Electronic Communications of the EASST
Volume 41 (2011)

Proceedings of the
Tenth International Workshop on
Graph Transformation and
Visual Modeling Techniques
(GTVMT 2011)

Distributed Port Automata

Christian Krause

14 pages

Guest Editors: Fabio Gadducci, Leonardo Mariani
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Distributed Port Automata

Christian Krause[*]

christian.krause@hpi.uni-potsdam.de
Hasso Plattner Institute (HPI), University of Potsdam, Germany

Abstract: Dynamic reconfigurations are a powerful approach for the adaption of component-based or service-oriented software systems at runtime. Important issues in this area are the problems of state transfer and state consistency, i.e., to determine the system state after a reconfiguration and to ensure that it is valid. To deal with these problems, we introduce *distributed port automata* in this paper. Distributed port automata combine structural and behavioral system properties and therefore allow to reason about dynamic reconfigurations. In our approach, we use an automata-based model for describing the behavior of the primitive building blocks in a system, and a graph-based model for describing its structure in terms of a network. We demonstrate how to derive the system semantics of a distributed port automaton and show that it is compositional. We consider an encoding of the coordination language Reo and show a new result on compositionality of flattening for distributed graphs.

Keywords: dynamic reconfiguration, distributed graph transformation, Reo

1 Introduction

A common approach in component-based and service-oriented software is to divide the system model into two orthogonal aspects: (i) the computation performed by a set of (black-boxed) components or services, and (ii) their coordination using some kind of ‘glue code’. The specification of the components or services is usually based on a behavioral, e.g. an automata-based, model. However, the composition and coordination of these functional building blocks is often done using graphical models, e.g. Petri nets or Reo connectors [Arb04]. Dynamic reconfigurations of such systems involve structural changes to these models at runtime. However, it is non-trivial to verify the impact of these structural modifications on the execution of the system. Specifically, when performing dynamic reconfigurations, two issues have to be dealt with: (i) the structural and logical integrity of the system has to be maintained, and (ii) it must be ensured that the system is in a consistent state after the reconfiguration (state transfer and consistency). For instance, it should be guaranteed that the reconfiguration is not performed within a critical section (e.g. a transaction), and that the system is not brought into a deadlock state.

In this paper, we introduce *distributed port automata* – an integrated structural and behavioral model for service and component coordination. We specify primitives using so-called *port automata* and combine them in a structural model based on distributed graphs. Specifically, we use the framework of distributed graph transformation [Tae99, EOP06], which enables us to model reconfigurations using algebraic graph transformation based on the double pushout approach [CMR$^+$97]. We then derive a semantics functor and show that it is compositional. For

[*] Supported by the research school in ‘Service-Oriented Systems Engineering’ at the Hasso Plattner Institute (HPI).

this purpose, we show that the flattening functor for distributed graphs is cocontinuous, i.e., it preserves colimits, which are the basis for gluing operations of (distributed) graphs. As applications, we consider a distributed port automata encoding of Reo and discuss a number of problems in the area of dynamic reconfigurations.

The work in this paper is a generalization of the approach in [Kra09]. Specifically, we replace the ad hoc connector model in [Kra09] by distributed graphs.

Organization: The rest of this paper is organized as follows. Section 2 recalls the basics of distributed graph transformation and includes a new result regarding flattening. Section 3 contains a brief introduction to Reo and the semantical model of port automata. Section 4 introduces distributed port automata and includes a discussion on composition and dynamic reconfiguration. Finally, Section 6 and 7 contain related work and conclusions.

Acknowledgements: The author of this paper is very grateful to Erik de Vink and Ulrike Golas for their proofreading.

2 Distributed graph transformation

We use the framework of distributed graph transformation, as introduced by Taentzer [Tae99], and later generalized to a notion of transformation of distributed objects by Ehrig et al. [EOP06]. In this section, we recall the basic notions of this framework and present a new result on compositionality of the flattening functor for distributed graph transformation.

Distribution of graphs can be described by adding a second layer 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 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.

We consider directed graphs $G = (V, E, s, t)$ with $s, t : E \to V$ source and target functions, and componentwise morphisms. Then, a distributed graph is defined as follows.

Definition 1 (distributed graph) A *distributed graph* (N, D) consists of a graph N, called the *network graph*, and a commutative functor $D : N \to \mathbf{Graph}$, where N is interpreted as a category.

The network graph N describes the topology of the system. The functor or *diagram* D 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 $D(e) : D(n) \to D(n')$. Following [EOP06], 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. We now recall the definition of morphisms for distributed graphs.

Definition 2 (distributed graph morphism) For two distributed 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 $f_D : D_1 \to D_2 \circ f_N$.

For brevity, we will just write f for the network morphism f_N. By definition, 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* graph morphism of n. Furthermore, for every edge $n \xrightarrow{e} n'$ in N_1 the following diagram commutes.

$$\begin{tikzcd} D_1(n) \arrow[r, "D_1(e)"] \arrow[d, "f_n"'] & D_1(n') \arrow[d, "f_{n'}"] \\ D_2(f(n)) \arrow[r, "D_2(f(e))"] & D_2(f(n')) \end{tikzcd} \quad (1)$$

Distributed graphs and their morphisms form the category $\mathbf{Dis}(\mathbf{Graph})$. Due to the categorical definition, the concept of distribution can be generalized to other structures by considering functors $D : N \to \mathbf{C}$ into a category $\mathbf{C}$, giving rise to a category $\mathbf{Dis}(\mathbf{C})$ of *distributed objects* [EOP06].

Flattening of distributed graphs The flattening operation for distributed graphs glues together all local graphs along their shared subgraphs. It is a well-known fact that the flattening of a distributed graph or object (N,D) can be achieved by considering the colimit of D [Tae99] and that this extends to a functor $F : \mathbf{Dis}(\mathbf{C}) \to \mathbf{C}$, assuming $\mathbf{C}$ is cocomplete [LT05]. This definition is rather elegant since it defines flattening in terms of a universal property, and not by an algorithm or referring to an operational semantics.

Considering flattening of distributed graphs or objects is in particular interesting when distribution is used for representing a logical partitioning of an otherwise 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 $\mathbf{Dis}(\mathbf{C})$ and flattening the result should yield the same outcome as first flattening both distributed objects and then composing them in $\mathbf{C}$. Formally, we need to show that the flattening functor F preserves pushouts, or more generally, colimits.

Theorem 1 (flattening preserves colimits) *Let* $\mathbf{C}$ *be a cocomplete category. The flattening functor* $F : \mathbf{Dis}(\mathbf{C}) \to \mathbf{C}$ *has a right adjoint and is therefore cocontinuous.*

Proof. A detailed proof is given in the appendix. □

Distributed graphs or, more generally, distributed objects can be used to describe a logical partitioning of an otherwise flat structure. Due to Theorem 1, composition and transformation of distributed objects can be transparently implemented on the underlying flat structure. This result will also be crucial for the compositional semantics of distributed port automata in Section 4.

3 Channel-based coordination with Reo

Reo [Arb04] is a channel-based coordination language, with applications in component-based and service-oriented software systems. 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. Channels may impose constraints on the data flow at their ends. For instance, the communication through channels can be (a)synchronous and (un)buffered.

For the scope of this paper, we consider only a small set of primitives, summarized 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. Thus, the channel

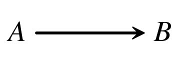
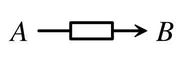
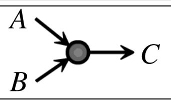
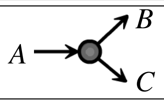
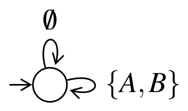
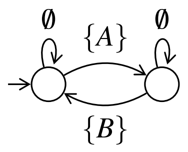
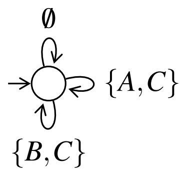
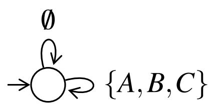
	Sync	*FIFO1*	*Merger*	*Replicator*
Graphical notation				
Port automaton				

Table 1: Graphical notation and port automata for some basic Reo primitives

blocks if the party at the sink end is not ready to receive data. The *FIFO1* channel is a directed, asynchronous channel with a buffer of size one. It reads a data item from its source ends, buffers it, and releases it again at its sink end. Channels in Reo can be joined together using nodes, which read data items from sink ends and write data items to source ends of channels that coincide in it. Nodes behave as non-deterministic *Merger*s on the sink ends and as (synchronous) *Replicator*s on the source ends. This means that a node non-deterministically reads a data item from one of the incoming sink ends and replicates it to all outgoing source ends without buffering it.

3.1 Building connectors

In Reo, channels and nodes are joined together to build so-called *connectors* which act as *glue code* between components or services and essentially enforce a communication protocol among them. This coordination of components and services is performed from outside and without their knowledge, which is, therefore, referred to as *exogenous* coordination.

An important aspect of Reo is that nodes do not buffer data items and, thus, allow synchrony to propagate through the connector. For instance, a sequence of *n Sync* channels joined together using nodes has the same qualitative behavior as a single *Sync*. Note also that Reo allows an arbitrary mixing of synchrony and asynchrony, which is also indicated by our first example.

Example 1 *We consider a simple instant messenger application, shown in Figure 1. Two* Client *components exchange messages via a connector. Messages are sent into FIFO1 channels and are thus buffered. When they leave the buffer, they are synchronously replicated by the node behind the FIFO1 and sent to both clients. This can succeed only when both clients are ready to accept*

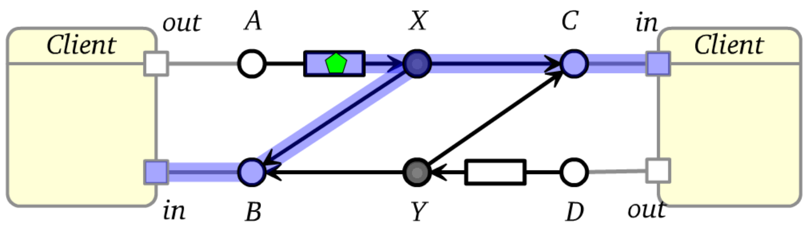


Figure 1: Instant messenger application modeled in Reo

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. In Figure 1, a message, depicted as a green token, is already buffered in the upper FIFO1 and is about to be sent to both clients. This synchronous dataflow is indicated by the blue highlighted parts of the connector. Note that the nodes X and Y are Replicators, whereas B and C act as Mergers.

3.2 Port automata semantics

Port automata [KC09] are the most basic semantical model for Reo. Although they cannot model data constraints, port automata already capture some of the key concepts required for defining channels and other primitives, i.e., synchrony, mutual exclusion and state. Port automata come equipped with a join-operator for composition, which allows to compute the semantics of a connector given the semantics of the primitives it is comprised of. Port automata have been introduced in [KC09] as an abstraction of constraint automata [BSAR06] and offer, because of their conciseness, the possibility of better comparison with other semantical models, such as Reo automata [BCS09]. In the following we define port automata formally.

Definition 3 (port automaton) A *port automaton* $PA = (Q, N, T, q_0)$ consists of a set of states Q, a set of port names N, a transition relation $T \subseteq Q \times 2^N \times Q$, and an initial state $q_0 \in Q$.

We usually write transitions as $p \xrightarrow{S} q$ with $p, q \in Q$ source and target states, $S \subseteq N$ the set of synchronously firing ports. The port automata for the primitives are given in the lower part of Table 1. We see immediately that the *Sync* channel as well as nodes (modeled using *Merger* and *Replicator* primitives) are stateless, i.e., they have only one state. Note that *Replicator*s basically synchronize all ends, whereas *Merger*s essentially implement mutual exclusion of the incoming ends. Note that the port automata for the primitives include explicit τ-transitions via the empty set of port names.

4 The category of port automata

To establish a categorical framework suitable for applying graph transformation methods, we introduce a notion of simulations for port automata in the following. A port automata simulation essentially consists of a mapping of states and transitions, and an inverse mapping of port names. We ensure consistency of firing events using a condition on the transitions of the port automata.

Definition 4 (port automata simulation) Let $PA_1 = (Q_1, N_1, T_1, q_0^1)$ and $PA_2 = (Q_2, N_2, T_2, q_0^2)$ be two port automata. A simulation $f = (f_Q, f_N, f_T)$ consists of functions $f_Q : Q_1 \to Q_2$, $f_N : N_2 \to N_1$ and $f_T : T_1 \to T_2$, such that:

- $f_Q(q_0^1) = q_0^2$
- $f_T\big(q_1 \xrightarrow{S_1} p_1\big) = \big(q_2 \xrightarrow{S_2} p_2\big)$ implies $q_2 = f_Q(q_1)$, $p_2 = f_Q(p_1)$ and $S_2 = f_N^{-1}(S_1)$

Note that port names are mapped in the opposite direction and that the condition $S_2 = f_N^{-1}(S_1)$ ensures consistency of firing events on all shared port names. We now consider an example.

Example 2 *An example of a port automata simulation is depicted in Figure 2. States q_0, q_2 are both mapped to p_0, and q_1 is mapped to p_1. The port name function is the inclusion map in the opposite direction. The transition via $\{B,C\}$ in the source is mapped to the transition via $\{B\}$. The transition via $\{C\}$ corresponds to the τ-step in the target automaton.*

Note that mapping a normal, i.e., non-empty transition to a τ-step essentially allows us to model interleaved semantics. Intuitively, the source automaton performs an (observable) step, whereas the target automaton takes a (silent) τ-transition. We define composition and identity of port automata simulations componentwise. The following lemma states that port automata and simulations form a category.

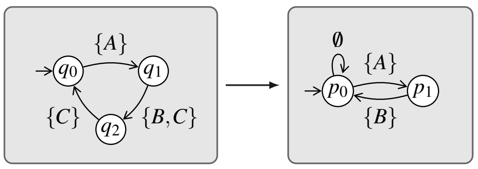


Figure 2: a port automata simulation

Lemma 1 (category of port automata) *Port automata and port automata simulations give rise to a category, denoted by* **PA**.

Proof. We verify the consistency condition of firing events for the identity: $S = id_N^{-1}(S)$. Similarly, for the composition of $f : PA_1 \rightarrow PA_2$ and $g : PA_2 \rightarrow PA_3$ it holds that:

$$(g \circ f)_N^{-1}(S_1) = g_N^{-1}\left(f_N^{-1}(S_1)\right) = g_N^{-1}(S_2) = S_3 \qquad \square$$

The port automaton with one state, an empty port name set and a τ-transition is the final object in **PA**, denoted by **1**. If there is a morphism between two port automata PA_1 and PA_2, we may also write $PA_1 \succeq PA_2$ for short. Similarly, if there exists a (categorical) isomorphism, we denote this by $PA_1 \cong PA_2$. Note that this notion of behavioral equivalence is stronger than standard equivalences based on, e.g., bisimulation [Mil89].

We define the composition of port automata now using pullbacks in **PA**. Note that since we model τ-steps explicitly, the composition is based on mere synchronization only. Interleavings of actions are modeled by synchronizations of an action and a τ-transition in the other automaton.

Theorem 2 (pullbacks of port automata) **PA** *has pullbacks which can be constructed componentwise. For a cospan $PA_1 \rightarrow PA_0 \leftarrow PA_2$ the pullback object is $PA_3 = (Q_3, N_3, T_3, q_0^3)$ with:*

$$\begin{tikzcd}
PA_0 & PA_1 \arrow[l, "f_1"'] & \\
PA_2 \arrow[u, "f_2"] & PA_3 \arrow[l, "g_2"'] \arrow[u, "g_1"'] & \\
& & X \arrow[ul, dotted, "h"] \arrow[uul, bend right, "h_1"'] \arrow[ull, bend left, "h_2"]
\end{tikzcd}$$

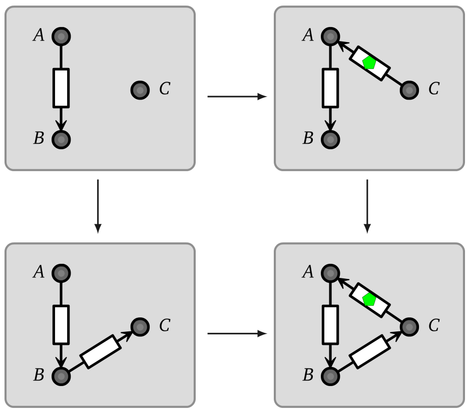
- *$Q_3 = Q_1 \times_{Q_0} Q_2$ (pullback in* **Set***)*
- *$N_3 = N_1 +_{N_0} N_2$ (pushout in* **Set***)*
- *$q_0^3 = \langle q_0^1, q_0^2 \rangle$, and*
- *T_3 defined by the following rule:*

$$\frac{f_{1,T}\left(q_1 \xrightarrow{S_1} p_1\right) = f_{2,T}\left(q_2 \xrightarrow{S_2} p_2\right) \qquad S_0 = f_{1,N}^{-1}(S_1) = f_{2,N}^{-1}(S_2) \qquad S_3 = S_1 +_{S_0} S_2}{\langle q_1, q_2 \rangle \xrightarrow{S_3}_3 \langle p_1, p_2 \rangle} \tag{2}$$

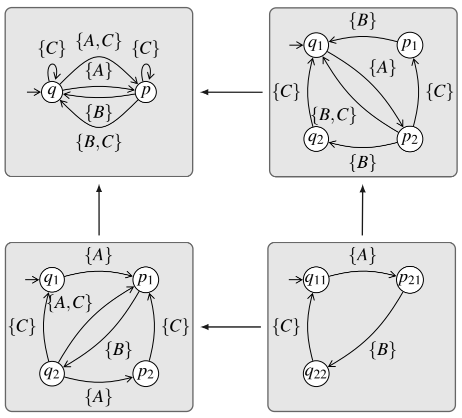
Proof. Due to the componentwise construction in **Set**, we only need to show that g_1, g_2 and h are valid **PA**-morphisms, i.e., we need to check the consistency condition for the firing ports. For g_1 we have by construction: $g_{1,N}^{-1}(S_3) = g_{1,N}^{-1}(S_1 +_{S_0} S_2) = S_1$, and analogously for g_2. For h, assume there is a transition via S in X that is, w.l.o.g., mapped to a transition via S_i in PA_i with $i = 1,2$. We need to show that there is a corresponding transition via S_3 in PA_3. Since the h_i are by assumption valid **PA**-morphisms, we have: $h_{i,N}^{-1}(S) = S_i$. Moreover, $f_1 \circ h_1 = f_2 \circ h_2$ and thus: $f_{1,N}^{-1}(S_1) = f_{2,N}^{-1}(S_2)$. Therefore, the premise of rule (2) is fulfilled and the transition exists. Since $h_i = g_i \circ h$ we also know that $h_N^{-1}(S) = S_3$. Thus, the consistency condition holds also for h. □

Note that $f_{1,T}(q_1 \xrightarrow{S_1} p_1) = f_{2,T}(q_2 \xrightarrow{S_2} p_2)$ implies $f_{1,Q}(q_1) = f_{2,Q}(q_2)$ and $f_{1,Q}(p_1) = f_{2,Q}(p_2)$. Port automata pullbacks generalize the join operator for constraint automata of [BSAR06], since they allow to join two automata not just over a set of shared port names, but over a whole automaton, which can be seen as a shared context. The semantics of the original join operator is a special case where PA_0 is stateless, i.e., it has only one state. Note also that the new composition operator is derived from our notion of simulation and phrased in terms of a universal property.

Example 3 *An example of a port automata pullback is depicted in Figure 3(b). The state maps are indicated by state names. Note that the resulting automaton in the bottom right actually includes more states which are suppressed here since they are unreachable. The automata in this pullback can be modeled using FIFO1 channels. In fact, the automata correspond to Reo connectors and the whole pullback corresponds to a structural gluing of these connectors, as shown in the pushout of Reo graphs in Figure 3(a).*



(a) connector graph pushout



(b) port automata pullback

Figure 3: categorical composition of connectors and port automata

We use the default notation for pullbacks of port automata, i.e., $PA_3 = PA_1 \times_{PA_0} PA_2$. Note that we have indirectly shown that **PA** has general limits, since it has pullbacks and a final object. The categorical construction using pullbacks furthermore includes the morphisms into the original port automata and thereby relates them with the result using simulations, which will be helpful in the context of dynamic reconfigurations.

5 The category of distributed port automata

Example 3 in the previous section already indicated a connection between the graph-based model of Reo connectors and the semantical counterpart in the category of port automata. We now provide a categorical model which integrates the structural aspects, i.e., the topology of a connector or network, and the semantics of the primitives it is comprised of. We use the theory of distributed graph transformation as in Section 2. Specifically, we consider the following category.

Definition 5 (category of distributed port automata) The category of *distributed port automata* is defined as $\mathbf{Dis}(\mathbf{PA}^{op})$.

Note that we consider $\mathbf{Dis}(\mathbf{PA}^{op})$ and not $\mathbf{Dis}(\mathbf{PA})$ since the **PA**-semantics is contravariant to the graph structure of networks, as indicated also in Example 3 where a pushout of Reo connectors is mapped to a pullback of the corresponding port automata. However, before investigating the properties of $\mathbf{Dis}(\mathbf{PA}^{op})$, we first show how to encode Reo into this model.

5.1 Reo connectors as distributed port automata

To encode Reo in $\mathbf{Dis}(\mathbf{PA}^{op})$ we map every primitive in a connector to its corresponding port automaton (cf. Table 1), and every node X to a stateless port automaton with one transition via X and one via the empty port name set. Note again that *Merger*s have to be modeled explicitly. The port automata for primitives and nodes are now considered as vertices in a network graph N. For every pair of a node and a connected primitive we create an edge between the corresponding vertices in the network graph. The edge points towards the port automaton of the primitive. However, it corresponds to the **PA**-simulation in the opposite direction which maps all transitions of the primitive that involve the connected node X to the self loop transition that includes X, and all other transitions to $\emptyset$. The reason for inverting the edges is, formally, the fact that we consider the category $\mathbf{Dis}(\mathbf{PA}^{op})$. Informally, it can be motivated by arguing that the edges in the network graph represent primarily structural mappings of the node names, which are contravariant for port automata simulations.

Example 4 *Figure 4 depicts the distributed port automaton for the instant messenger in Example 1. For simplicity, we have modeled the Replicator nodes X and Y using a single port and omitted the port automata for the nodes B and C, and all τ-transitions. Note also that the edges in the distributed port automaton correspond to* inverse *simulations. This is due to the fact that the edges model, primarily, structural mappings of ports.*

5.2 Composing distributed port automata

Our goal is to use double pushout graph transformation [CMR+97] for realizing reconfigurations of connectors modeled using distributed port automata. In essence, we are aiming at applying the theory of distributed graph transformation to our automata-based framework for component connectors. To enable composition and reconfiguration in our setting, we need to ensure that the category of distributed port automata, as defined above, has pushouts. We do this in the following lemma.

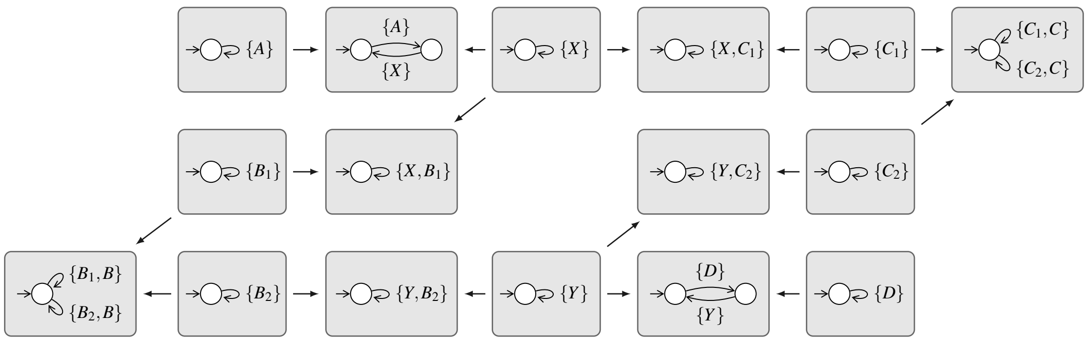


Figure 4: instant messenger modeled as distributed port automaton

Lemma 2 *The category* $\mathbf{Dis}(\mathbf{PA}^{op})$ *is cocomplete.*

Proof. $\mathbf{PA}$ is complete since it has pullbacks and a final object. Hence, $\mathbf{PA}^{op}$ is cocomplete. If $\mathbf{C}$ is (co)complete, then so is $\mathbf{Dis}(\mathbf{C})$ [EOP06]. Thus, $\mathbf{Dis}(\mathbf{PA}^{op})$ is cocomplete. □

In Figure 3(a) we have shown how pushouts can be used to glue Reo networks along a common subconnector. Note that this gluing is of a pure structural nature. In Section 5.1 and in particular in Figure 4 we have seen how Reo connectors can be encoded as distributed port automata. An important aspect of this encoding was that the topology of the network is modeled by the network graph of the distributed port automaton. Moreover, the local port automata model each primitive in the network.

Since $\mathbf{Dis}(\mathbf{PA}^{op})$ is cocomplete, we can compose distributed port automata using pushouts. Usually, the semantics of primitives is fixed. Therefore, the local morphisms (port automata simulations) are isomorphisms. This is, for instance, the case for Reo networks, but also for Petri nets. In this situation, the primitive port automata in the network nodes are not changed when composing two distributed port automata using pushouts. Thus, the composition is performed only on the network level and is of purely structural nature. The case where a local port automata morphism is not an isomorphism has applications as well. Essentially, it allows to model a refinement of primitives, e.g. refining a buffer with bag semantics to one with *FIFO* semantics.

5.3 Semantics of distributed port automata

Distributed port automata capture the semantics of each primitive and the topology of the connector. The semantics of a connector modeled as a distributed port automaton $(N,D) \in \mathbf{Dis}(\mathbf{PA}^{op})$ is given by the port automaton that corresponds to the colimit of the diagram D. This colimit glues together all node names and, since we have reversed the arrows, corresponds to a limit in $\mathbf{PA}$.

As discussed in Section 2, the colimit over a distributed graph or object can be interpreted as a flattening operation, which, moreover, extends to a flattening functor $F : \mathbf{Dis}(\mathbf{C}) \to \mathbf{C}$. In the case of distributed port automata, i.e., in the category $\mathbf{Dis}(\mathbf{PA}^{op})$, the flattening using colimits can, thus, be used to define the composite behavior of connectors in terms of a semantics functor.

Definition 6 (semantics functor) Let $F : \mathbf{Dis}(\mathbf{PA}^{op}) \to \mathbf{PA}^{op}$ the flattening functor for distributed port automata. By reverting the arrows, this induces the following contravariant functor:

$$Sem : \mathbf{Dis}(\mathbf{PA}^{op}) \to \mathbf{PA}$$

which is called the *semantics functor* for distributed port automata.

The following lemma states that the semantics of distributed port automata is compositional, i.e., it is compatible with composition of distributed port automata using pushouts, or more generally, with colimits.

Lemma 3 (compositional semantics) *The semantics functor* $Sem : \mathbf{Dis}(\mathbf{PA}^{op}) \to \mathbf{PA}$ *is compositional, i.e., it maps colimits of distributed port automata to limits of port automata.*

Proof. This holds since the flattening functor is cocontinuous (cf. Theorem 1). □

Thus, we have shown that a structural gluing of connectors in $\mathbf{Dis}(\mathbf{PA}^{op})$ which is realized as a pushout of the network graphs has a corresponding semantical join operation, i.e., a pullback of the respective port automata. Furthermore, Theorem 1 shows that structure and semantics of distributed port automata form a pair of adjoint functors.

5.4 Towards dynamic reconfigurations

Since distributed port automata integrate the structure and semantics of, e.g., Reo connectors and Petri nets, they can be used for problems occurring in the area of dynamic reconfigurations. An important issue is to determine the state of the system after a reconfiguration, which we refer to as the problem of *state transfer*. Moreover, it is crucial to ensure that the new system state is indeed a valid one, which we refer to as the problem of *state consistency*.

Distributed port automata provide means for reasoning about state transfer and state consistency. To illustrate this, we revisit Example 3 which shows how a pushout of Reo networks at the structural level (Figure 3(a)) corresponds to a pullback of port automata at the semantical level (Figure 3(b)). We now interpret this as an application of a simple reconfiguration rule which adds a full *FIFO1* between the matched nodes C and A. Essentially, we assume that Figure 3(a) is the right part of a DPO diagram. In this view, the corresponding pullback of port automata can be seen as an application of a 'semantical reconfiguration rule' in the category of port automata.

In such an approach, we can deduce the effect of an application of a purely structural reconfiguration rule on the connector semantics. For instance, assume that the connector to be reconfigured (lower left automaton in Figure 3(b)) is in its initial state q_1, in which both *FIFO1*s are empty. The image in the left-hand side of the rule (upper left automaton) is state q. Moreover, we assume that the *FIFO1* to be added by the rule is initially full, i.e., the selected state in the right-hand side of the rule is q_1. This information is sufficient to deduce the state of the connector after the reconfiguration. The pullback construction given in Theorem 2 yields as new target state q_{11} in the resulting automaton. Thus, we can determine the state after a reconfiguration, which provides us with a means to solve the problem of state transfer.

Now assume that before the reconfiguration, the *FIFO1*(A,B) and *FIFO1*(B,C) are already full, i.e., the automaton in the upper right part is in state p_1 and the automaton in the lower left part is in state p_2. Both states are mapped to state p in the upper left automaton, and therefore correspond to state p_{12} in the lower right automaton, which is not shown because it is unreachable from the initial state. In this particular state, all three *FIFO1*s are full and the connector would run into a deadlock. Therefore, it is crucial to check in which state the connector currently is, before reconfiguring it. In essence, this is the problem of state consistency. In the distributed port automata approach, we can characterize reconfigurations which yield consistent connector states by demanding that the target state of the reconfigured connector must be reachable from the initial (or the current) state.

6 Related work

Constraint automata [BSAR06] are a compositional model for Reo [Arb04]. Our port automata are an abstraction of constraint automata where data constraints are ignored. The compositionality result in [BSAR06] says that the semantics of a connector can be computed out of the semantics of its constituent primitives. Our notion of compositionality is more general since it works with arbitrary gluings of connector graphs. In fact, we generalize the join operation of [BSAR06] by allowing to join two automata along a common context automaton.

Graph transformation based reconfigurations for Reo are considered in [KMLA11]. 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 is not compositional, since graph grammars in general are not compositional.

A model for distributed connectors and their reconfigurations is suggested in [KAV09]. Similar to the approach in this paper, the distributed connector model is based on distributed graph transformation. However, the semantics of connectors is not considered and, thus, dynamic reconfigurations cannot be modeled.

A basic logic for reasoning about connector reconfigurations in Reo, including a model checking algorithm is the topic of [Cla08]. Different than the work in this paper, the author uses a formalization of connectors, which is particularly not a graph model. Moreover, the reconfiguration operations are rather low-level and provide no means for a rule-based definition of reconfigurations.

We also mention some related work on Petri nets. A marking graph semantics of Petri nets is considered in [PER01]. 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 considered in [BCEH05]. A categorical approach to automata-based semantics for Petri nets is considered in [DS02]. However, this approach is more restrictive than our port automata model, since concurrent actions imply interleaved semantics.

7 Conclusions

We have presented distributed port automata – a model for component-based software systems, which uses (i) an automata-based model for specifying the semantics of the 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 a network. We have shown that the flattening functor for distributed graphs can be used to derive the semantics of distributed port automata and, moreover, that it is compositional. For this purpose, we have shown that the flattening functor has a right-adjoint.

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.

Bibliography

[Arb04] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science* 14:329–366, 2004. doi:10.1017/S0960129504004153

[BCEH05] P. Baldan, A. Corradini, H. Ehrig, R. Heckel. Compositional semantics for open Petri nets based on deterministic processes. *Mathematical Structures in Computer Science* 15:1–35, 2005.

[BCS09] M. Bonsangue, D. Clarke, A. Silva. Automata for context-dependent connectors. In *Coordination Models and Languages*. LNCS 5521, pp. 184–203. Springer, 2009.

[BSAR06] C. Baier, M. Sirjani, F. Arbab, J. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming* 61(2):75–113, 2006. doi:10.1016/j.scico.2005.10.008

[Cla08] D. Clarke. A basic logic for reasoning about connector reconfiguration. *Fundamenta Informaticae* 82(4):361–390, 2008.

[CMR$^+$97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. *Handbook of Graph Grammars and Computing by Graph Transformation*. Chapter Algebraic approaches to graph transformation I: Basic concepts and double pushout approach, pp. 163–245. World Scientific, 1997.

[DS02] M. Droste, R. M. Shortt. From Petri nets to automata with concurrency. *Applied Categorical Structures* 10(2):173–191, 2002. doi:10.1023/A:1014305610452

[EOP06] H. Ehrig, F. Orejas, U. Prange. Categorical foundations of distributed graph transformation. In *ICGT'06*. LNCS 4178, pp. 215–229. Springer, 2006. doi:10.1007/11841883

[KAV09] C. Koehler, F. Arbab, E. de Vink. Reconfiguring distributed Reo connectors. In *WADT'09*. LNCS 5486, pp. 221–235. Springer, 2009. doi:10.1007/978-3-642-03429-9

[KC09] C. Koehler, D. Clarke. Decomposing port automata. In *SAC'09*. Pp. 1369–1373. ACM, New York, NY, USA, 2009. doi:10.1145/1529282.1529587

[KMLA11] C. Krause, Z. Maraikar, A. Lazovik, F. Arbab. Modeling dynamic reconfigurations in Reo using high-level replacement systems. *Science of Computer Programming* 76(1):23–36, 2011.
doi:10.1016/j.scico.2009.10.006

[Kra09] C. Krause. Integrated structure and semantics for Reo connectors and Petri nets. In *ICE'09*. EPTCS 12, p. 57. 2009.

[LT05] J. de Lara, G. Taentzer. Modelling and analysis of distributed simulation protocols with distributed graph transformation. In *ACSD'05*. Pp. 144–153. 2005.
doi:10.1109/ACSD.2005.27

[Mil89] R. Milner. *Communication and concurrency*. Prentice Hall International, 1989.

[PER01] 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*. LNCS 2128, pp. 230–249. Springer, 2001.

[Tae99] G. Taentzer. Distributed graphs and graph transformation. *Applied Categorical Structures* 7:431–462, 1999.
doi:10.1023/A:1008683005045

A Proof for Theorem 1

Proof. We consider the functor $G : \mathbf{C} \to \mathbf{Dis}(\mathbf{C})$ which maps an object $X \in \mathbf{C}$ to the distributed object $(1, (1 \mapsto X)) \in \mathbf{Dis}(\mathbf{C})$ and a morphism $f : X \to X'$ to $(id_1, (f))$, where 1 is the terminal object in **Graph**. We have $F \dashv G$ since there is a bijective correspondence:

$$\Phi_{X,Y} : \hom_{\mathbf{C}}(FY, X) \to \hom_{\mathbf{Dis}(\mathbf{C})}(Y, GX)$$

that is natural in $X \in \mathbf{C}$ and $Y = (N, D) \in \mathbf{Dis}(\mathbf{C})$. The flattening functor F associates the colimit to a distributed object. Thus, FY is the colimit of the diagram D together with $\mathbf{C}$-morphisms $(y_n : D(n) \to FY)_{n \in N}$. Now, for a $\mathbf{C}$-morphism $h : FY \to X$ we have the $\mathbf{Dis}(\mathbf{C})$-morphism $\Phi_{X,Y}(h) = (!_N, (h \circ y_n)_{n \in N}) : Y \to GX$ where $!_N : N \to 1$ is the terminal map for N in **Graph**. The mapping $\Phi_{X,Y}$ is bijective since all $\mathbf{Dis}(\mathbf{C})$-morphisms $Y \to GX$ are of the above form.

Now let $f : X \to X'$ a $\mathbf{C}$-morphism and $g : Y' \to Y$ a $\mathbf{Dis}(\mathbf{C})$-morphism. The morphism $Gf : GX \to GX'$ is given as above. $Fg : FY' \to FY$ is the unique morphism into the colimit object FY. Now we need to show the following naturality condition:

$$Gf \circ \Phi_{X,Y}(h) \circ g \quad \overset{!}{=} \quad \Phi_{X',Y'}(f \circ h \circ Fg) \quad : \quad Y' \to GX'$$

We write $Y' = (N', D')$ and $g = (g_{N'}, (g_m)_{m \in N'})$. Moreover, let $(y'_m : D'(m) \to FY')_{m \in N'}$ the $\mathbf{C}$-morphisms into the colimit of Y'. We now exploit the componentwise composition of $\mathbf{Dis}(\mathbf{C})$-

morphisms:

$$
\begin{aligned}
Gf \circ \Phi_{X,Y}(h) \circ g &= (id_1, (f)) \circ (!_N, (h \circ y_n)_{n \in N}) \circ \big(g_{N'}, (g_m)_{m \in N'}\big) \\
&= \Big(id_1 \circ\, !_N \circ g_{N'}, (f \circ h \circ y_{g_{N'}(m)} \circ g_m)_{m \in N'}\Big) \\
&= \Big(!_{N'}, (f \circ h \circ y_{g_{N'}(m)} \circ g_m)_{m \in N'}\Big) \\
&= \big(!_{N'}, (f \circ h \circ Fg \circ y'_m)_{m \in N'}\big) \qquad (3) \\
&= \Phi_{X',Y'}(f \circ h \circ Fg)
\end{aligned}
$$

Equality (3) holds since Fg is the unique morphism into the colimit FY. □