Connector Rewriting with High-Level Replacement Systems

Christian Koehler[1,2], Alexander Lazovik[1,3], Farhad Arbab[1]

CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

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 channels with well-defined behavior. These channels are mobile, i.e. can be dynamically created and reconfigured at run-time. Based on these language features, we introduce a high-level transformation system for Reo. We show how transformations of Reo connectors can be defined using the theory of *high-level replacement (HLR) systems*. This leads to a powerful notion of dynamic connector reconfiguration in Reo. Moreover, the rewrite rules are naturally expressed in Reo's visual syntax for connectors.
Applications of this framework are manifold, due to the generality of the field of coordination. In this paper we provide an example from the area of Service-oriented Computing.

Keywords: Reo, coordination, high-level replacement systems, adhesive categories, model transformation, service composition.

1 Introduction

Component composition is one of the greatest challenges in distributed, heterogenous environments. The concept of coordination directly addresses this problem by providing models for describing the necessary communication among the components in a composed system. In this sense, coordination languages 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. In this paper, we use Reo [1], an exogenous coordination language for distributed environments. 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.

[1] E-mail: {koehler,lazovik,farhad}@cwi.nl

[2] The work in this paper is supported in part by a grant from the GLANCE funding program of the Dutch National Organization for Scientific Research (NWO), through project WoMaLaPaDiA (600.643.000.06N09).

[3] This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

As software systems evolve over time, it is important to be able to dynamically adapt to changes in the interface of a component on the one hand, and its behavioral properties on the other. Further, there is often a need to reconfigure a complete structure of the system or even to redefine the composition of the components at run-time. In service-oriented architecture (SOA), adapting the service composition according to an issued user request is one of the possible examples of such system reconfiguration. In a typical scenario, the basic reference process is modified to satisfy the extended objectives of a particular client request.

Traditionally, applying changes to existing compositions is a very difficult task: it is a time-consuming, expensive procedure that can be done only by business domain experts. Adapting the existing composition for a particular customer usually requires plenty of time and cannot be done on-the-fly. The problem is even worse: different customers request different extensions, and organizations are stuck with maintaining separate processes for different groups of their customers. However, they still miss business opportunities because they are not able to satisfy clients that need dissimilar variations on top of the existing processes. But, if we look at what customers want, we see that there is always a simple base functionality that is requested by all of them, plus some specific options that are taken from a limited (yet potentially large) set of additional features. The number of possible combinations of optional features is enormous, making it impossible to maintain all possible process instantiations for all customers.

In this paper we propose a different approach based on high-level replacement systems. In this approach, business organizations need to maintain only a simple base system that is reconfigured using transformation rules when necessary. Each transformation rule (or set of rules) corresponds to one of the possible added value features of the business domain. For example, in a travel package scenario, booking a hotel and a flight represents basic functionality, while reserving some tourist guides or restaurants are additional features. The transformation rules are expressed using a separate high-level meta-language.

The idea of high-level replacement (HLR) systems is to define a set of rewrite rules, each consisting of a pattern, that must be matched and an associated template, describing the changes to be performed over the system. Additional application conditions may restrict the transformation further. A major advantage of this rewriting approach is that it allows to specify abstractly in which situations and how a system should be changed, including possible dependencies that need to be updated. Further, these rules cannot only be applied locally, but also globally, i.e. wherever the patterns match. Another important aspect of this approach is the granularity of changes that are made. Instead of sequentially performing local modifications, complex structural refactorings can be achieved in an atomic step. In this paper we apply the ideas of HLR systems to the coordination language Reo.

Organization: The rest of this paper is organized as follows. In Section 2 we describe related work. Section 3 contains an overview of coordination with Reo, including a short introduction to its dynamic aspects. Our running examples, introduced in Section 4 are taken from the field of service-oriented computing. Section 5 is an introduction to general high-level replacement systems. In Section 6, we give a formal definition of Reo connectors and show that these definitions give rise to a

HLR-system. We then extend the examples of Section 4 by rewrite rules in Section 7. In Section 8 we give a brief overview of the implementation of Reo. We conclude with remarks on possible future work in Section 9.

2 Related work

Reo [1] has been introduced as a general-purpose coordination language that is not restricted to a specific domain. A number of formal and informal descriptions of the dynamic features of Reo exist. A systematic discussion of connector reconfiguration in Reo is given in [9] and [10]. The approach includes a model checking algorithm for a reconfiguration logic, called *ReCTL**. Reconfigurations in these papers are performed directly using Reo's dynamic operations, like channel creation and node splitting. The transformation approach in this paper introduces a higher-level, algebraic view on connector reconfiguration. Reasoning through these transformations belongs to our future work.

High-level replacement systems [16] have their roots in the field of graph transformation. They were introduced as a generalization of graph grammars to other high-level structures. Initially, the structures that were considered in this context were all graph-like, e.g. Petri nets, AHL-nets and hypergraphs. Although the theory also has been applied to algebraic specifications for instance. High-level replacement systems can be seen as an abstract meta-language for arbitrary transformations. An important advantage of HLR-systems is their generality and the sophisticated techniques for reasoning about the semantics of transformations. An extensive introduction to the theory of HLR-systems can be found in [15]. An application to the field of service-oriented computing can be found in [20]. There, the authors use graph transformation for service discovery. In this paper, we propose HLR-systems as the basis for business process customization.

The idea of on-the-fly modifications of a process is not new: for example, in [31,25] the authors propose configurable event-driven process chains (C-EPC) as an extended reference modeling language that allows capturing the core configuration patterns. A comprehensive discussion and an overview of variability in process models are provided in [19,4]. The relations and inheritance among processes and the possibilities of transforming one process to another are discussed in [5]. SAP [13,32] (as well as other Enterprise Systems reference models) uses reference models as application process models that document the functionality of off-the-shelf-solutions. However, these models are specific to domains for which their processes were initially developed, with only limited customization options. In contrast to the cited work, this paper emphasizes how customization happens, what parts are to be modified, and how the actual transformation is performed.

There are a number of alternative approaches to building customer-specific processes. For example, automatic service composition techniques allow delivering a user-specific process based on a set of pre-defined objectives and preferences. A configurable approach to service composition is proposed in [7]. To support dynamic composition of web services from existing web services and dynamic integration of their data, [34] proposes a data integration technique. The *HTN* planner SHOP2 was applied for web service composition in [35]. In [24] an XML Service Request

Language was introduced in order to express complex user goals and preferences over partially composed business processes. A number of approaches to service composition and coordination are based on service rich semantic descriptions, e.g., knowledge-based semantic web service composition [8], service discovery and composition based on semantic matching [29], and semi-automatic composition of web services based on semantic descriptions [33]. A general drawback of automatic composition which stops its wide adoption is the impossibility to keep the semantic descriptions of process activities up-to-date, making altering and introducing new service implementations difficult. In contrast, the approach presented in this paper does not rely on any rich semantic descriptions, making it much more feasible for real world distributed scenarios involving heterogenous computational entities with no or little available semantics.

3 Reo Overview

Reo is an exogenous coordination language wherein so-called *connectors* are used to coordinate components from outside, i.e. the components are not aware of the fact that they are coordinated. Complex connectors are composed out of primitive ones, called *channels*, with well-defined behavior, supplied by users. To build larger connectors, channels can be joined into nodes and, in this way, arranged in a circuit. Each channel type imposes its own constraints for the possible data flow at its ends, e.g. synchrony or mutual exclusion. The ends of a channel can be either source ends or sink ends. Source ends accept data into, and sink ends produce data out of their respective channels. While the behavior of channels is user-defined, nodes are fixed in their routing constraints. Data flow at a node occurs, iff

(i) exactly one of its coincident sink ends provides data, and

(ii) all of its coincident source ends are able to accept data.

A node atomically does a destructive read at the active sink end and replicates the data item at all attached source ends. The examples in Figure 1 illustrate how this behavior can be used to impose constraints at the border nodes of a connector and thereby, implement a certain connector behavior. This is possible due to the fact that the synchrony and exclusion constraints of channels and nodes propagate through the (synchronous regions of the) connector. This behavior leads to a certain context-awareness in connectors. A detailed discussion of this can be found in [9].

Figure 1 shows two connectors that involve in total four different kinds of channels. The *Sync* channel synchronously takes a data item from its source end and makes it available at its sink end. Its transfer can succeed only if the node at the sink end is ready to accept data. The *LossySync* has the same behavior, except that it does not block if the receiver cannot accept data. Instead, the data item is read and destroyed by the channel. The $FIFO_1$ is an asynchronous channel that has a buffer of size one. Unlike the prior channels, $FIFO$s are stateful primitive connectors. The *SyncDrain* channel is different than the already introduced ones. It has two source ends through which it can only consume data. Its behavior can be described as follows: if there are data items available at both ends, it consumes both of them synchronously.

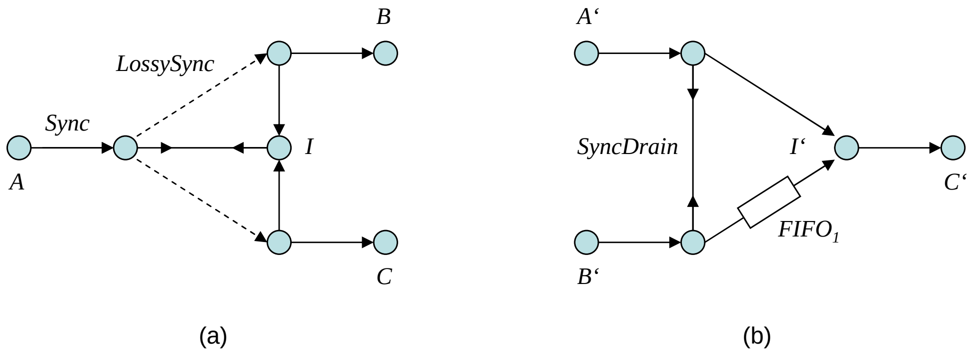

Fig. 1. (a) Exclusive router, (b) Ordering connector.

The exclusive router, shown in Figure 1.a, routes data from A to either B or C. The connector is only active if there is a write request pending at the source node A and there is at least one component attached to B or C that is ready to accept data. If both, B and C, are enabled the choice is made nondeterministically. This behavior can be explained by examining the possible data flow at the internal node I. It can accept data only from one of its attached sink ends. So one of the ends must not be active, which forces the corresponding *LossySync* to lose its data.

The second connector, shown in Figure 1.b, imposes an ordering on the dataflow from the source nodes A' and B' to the sink node C'. While the $SyncDrain$ synchronizes the inputs, the $FIFO_1$ stores the data item from B' and makes it available for the next step. Due to the fact that the $FIFO_1$ buffer cannot store more than one item, the connector cannot accept any new data from A' or B' in this configuration. The buffered element has to be removed before new input data can be processed.

Reo further includes a number of operations for changing the topology of a connector at run-time. Reo allows the dynamic creation of channels, splitting and joining of nodes, hiding internal nodes and more. The hiding of internal nodes plays an important role, because it allows to freeze the topology of a connector, such that it can only be used from outside with its published interface, but not changed anymore. The resulting connector can be viewed as a new primitive connector, since its internal structure is hidden and its behaviour is fixed.

Our focus in the rest of this paper will be on the basic operations that can be used for reconfiguring a connector. The proposed transformation approach addresses direct creation of channels and nodes, as well as connector reconfiguration using splitting and joining of nodes. To define complex transformations, we base our approach on the theory of so-called high-level replacement systems. An introduction to HLR-systems is given in Section 5. To motivate the use of such an abstract transformation approach, we first consider some examples from the field of Service-oriented Computing.

4 Running Example: Building a Travel Package

To illustrate our ideas we use a simple example from Service-oriented Computing that is based on the OTA standardized travel domain [27]. Service-oriented computing describes an architecture that uses loosely coupled services to support the requirements of business processes and users. Computational entities in a SOA environment are made available as independent services that can be accessed without any knowledge of their underlying implementation platform.

As the main coordination mechanism we rely on the channel-based exogenous coordination language Reo, introduced in Section 3. Reo supports a specific notion of compositionality that enables coordinated composition of individual services, as well as complex composite business processes. Accordingly, a coordinated business process consists of a set of web services whose collective behavior is coordinated by Reo connectors.

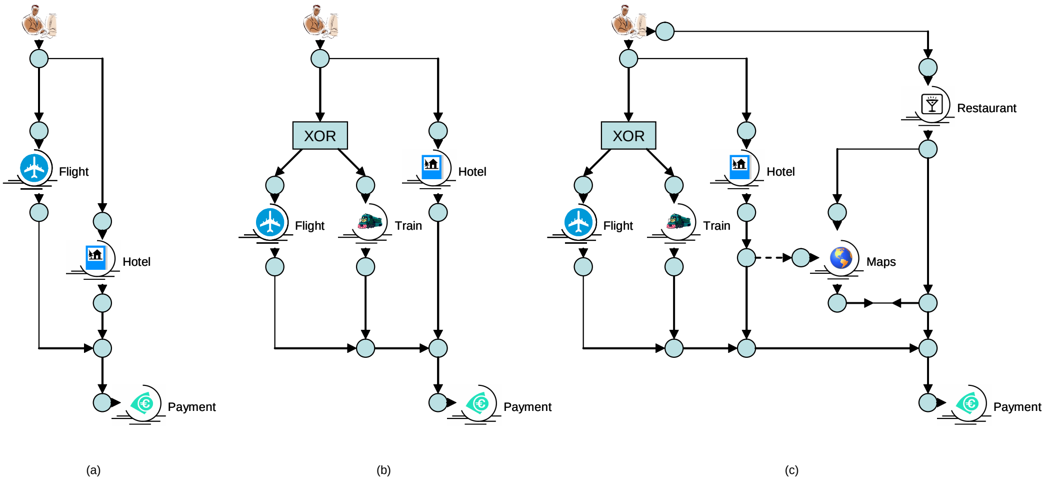

Fig. 2. Example processes.

OTA defines general XML-based message-oriented protocols for the travel domain. For demonstration purposes, we consider a simplified version that consists of reserving a hotel and booking transportation (flight or train in our simplified setting). A possible Reo coordination model is shown in Figure 2.a. To make this process work, we assumed to have a marketplace of properly described corresponding services. For example, a hotel reservation service is seen as an operation with one input and one output. An input message is taken from the user, while the output is directed to a payment service. We assume all issues regarding data compliance are resolved, since within the same business domain it is rather easy to achieve a common agreement on the protocol message types.

Consider a typical execution scenario. First, we try to reserve a hotel for some specific dates. We take the price and other information from the user, give it to the hotel service, and if the requested room is available, the execution proceeds to use a payment service. In parallel, a flight service is engaged for a booking in a similar way, and if the booking is succeeded, the payment service is used to pay for the tickets.

This process is simple, but it works for most users. However, even typical scenarios are usually more complicated with more services involved. Our simple process may be additionally enriched with services that the average user travelling abroad may benefit from, e.g., restaurants, museums, information offices, etc. It is difficult, or even nearly impossible, to put all services within the same process: users require different services only a few of which are shared. This makes it difficult to add new travel options, since only a limited number of users are actually interested in additional services. We demonstrate one of the possible complex requests that goes beyond the functionality of our simple process:

> *A trip to Lisbon is planned for the time of a conference; a hotel must be reserved and a travel ticket booked. The client prefers to have an alternative transportation option, e.g., train, in case the conference location is close. The client also requests to visit a restaurant close enough to the hotel for a dinner, if possible.*

For such a complex yet typical goal with a large number of loosely coupled services, hard-coded business process specifications cannot be used effectively. The problem is that the number of additional services is enormous, and every concrete user may be interested only in a few of them. Having these considerations in mind, it is sensible to design a business process to contain only basic services (as we did in Figure 2.a) with a number of external services (or other processes) that are not directly a part of the process, but the user may want them as added value, e.g., museums and places to visit, or making reservations in restaurants.

The Reo connector that satisfies the introduced user request is shown in Figure 2.c. The transition from the initial base process is done in two steps. At first we introduce a train service into the process as a transportation alternative to a flight service (see Figure 2.b). In the second step, we add a restaurant service that is optionally requested by the user. A map service is used to ensure that a restaurant is close enough to the requested hotel. The resulting process is shown in Figure 2.c.

To make this scenario work requires a mechanism to define the transformation rules that change the shape of the process to satisfy the user request in a fast and convenient way. This approach requires a language for high-level descriptions of possible process modifications according to the users' goals. In this scenario, the user is an actor who triggers the transformation rules. In general, the triggering events may come from different sources, such as other services, the underlying platform, or actions during the process execution itself, allowing runtime on-the-fly reconfigurations.

Traditional approaches to business process customization support small incremental modifications of the original process. However, applying atomic customization operations is difficult, time-consuming and error-prone. Moreover, each modification requires a skillful process designer who knows the corresponding business domain. All this makes it impossible to apply the traditional approaches to adapt a process to a user request.

In the following sections we introduce a high-level replacement approach which allows us to modify a process quickly, even dynamically on-the-fly, if necessary. The general idea is to group the related modifications into one or more transformation rules that are triggered when needed. Using a high-level replacement system, a process is adapted to a user request in the following way. Initially, we have a simple process with basic functionality (as the one introduced in Figure 2.a) and a repository of transformation rules that define possible process modifications. When a user issues his request, its corresponding rules are fetched from the repository. In our example, two rules apply: adding an alternative transportation (train service) to a flight, and adding a restaurant service that is close to the hotel. The latter may consist of two rules: add a travel option with a parameter restaurant; and a rule that uses a map service to ensure that the restaurant is located near the hotel.

5 High-Level Replacement Systems

High-level replacement (HLR) systems, introduced in [16], arose as a generalization of the theory of graph transformation to other high-level structures, e.g. Petri-nets, AHL-nets and algebraic specifications. In the following, we apply this theory in the field of coordination to allow us to define transformations for Reo connectors conveniently.

In high-level replacement systems, transformations are defined abstractly, based on concepts from category theory. An overview of HLR-systems can be found in [15] and [28]. We recall now the most important definitions. First we give a formal definition of *productions*, which are abstract representations of transformation rules. The second concept, called *derivations*, describes the application of a transformation rule w.r.t. a given input.

Definition 5.1 [Production] A *production* p is a span of morphisms

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

in a category $\mathbf{C}$. A production is called *linear*, if l and r are monos.

Productions are interpreted as rewrite rules. To support this intuition, we usually assume linear productions, i.e. the morphisms l, r are injective. In that case, the object K can be viewed as a substructure of both L and R. The object L is referred to as the left-hand side and R as the right-hand side of the rewrite rule. K can be thought of as a gluing object of the production, which means that it is the part that remains invariant during rule application. To apply a rewrite rule, the following steps must be performed:

(i) match L as a substructure of an input object M,

(ii) delete all parts of M that are matched by L, but are not in K,

(iii) add all parts of R that are not in K.

In other words, in a production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$, the morphism l determines which parts of the matched pattern should be deleted, and r determines which new parts should be added. The categorical formulation of this principle is the concept of so-called Double-Pushout (DPO) derivations. Pushouts are a categorical construction that describe the gluing of two structures.

Definition 5.2 [Derivation] Given an object M in a category $\mathbf{C}$, a production $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and a morphism $m : L \to M$, called *match*, a *derivation* $M \overset{p,m}{\Rightarrow} N$ is a diagram in $\mathbf{C}$

$$\begin{tikzcd}
L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "c"'] \arrow[dr, phantom, "(2)"] & R \arrow[d, "n"] \\
M & C \arrow[l, "g"] \arrow[r, "h"'] & N
\end{tikzcd}$$

where (1) and (2) are pushouts.

The notation $M \overset{p,m}{\Rightarrow} N$ can be interpreted in the following way: an input M is transformed, using the rule p and a match m, to the output N. The idea of high-level replacement categories is to define a set of axioms, called HLR-conditions, that not only allow DPO-Rewriting, but also ensure a number of useful properties and theorems. We recall now the most important properties of HLR-categories. For a detailed discussion see [15].

Glueing Condition An arbitrary production is not always applicable with respect to a given match. Formally, this problem corresponds to the existence of the pushout complement C in Def. 5.2. The so-called *glueing condition* states under which circumstances C exists and hence, the rewriting rule can be applied.

Local Church-Rosser deals with parallel and sequential independence of derivations. The specified conditions ensure that two productions applied to the same structure in any order or even in parallel, lead to the same result.

Concurrency While Local Church-Rosser deals with sequentially and parallel independent derivations, the concurrency theorem addresses rules that cannot be applied in arbitrary order. In particular, the theorem gives constructions for composing and decomposing sequentially dependent productions.

High-level replacement categories are a rich meta-language for general transformation systems. However, in order to instantiate an HLR-system with concrete structures, it is necessary to show that a language fulfills the requirements of an HLR-category. An overview of these HLR-conditions is given in [28]. The rather large number of requirements has been verified for many examples, including graphs, hypergraphs, Petri nets and algebraic specifications. However, we do not list these conditions here. Instead, we base our approach on another concept that is closely related to HLR-categories. In 2005, Lack and Sobociński introduced the concept of so-called *adhesive categories* [23]. The definitions made there turned out to be sufficient to prove almost all requirements of high-level replacement categories. In the following section, we use this result to instantiate Reo as an HLR-category.

6 Connector Rewriting

In this section we give formal definitions for the structural aspects of Reo connectors. Next, we show that these definitions are appropriate to derive a high-level replacement system.

Definition 6.1 [Connector] A *connector* $C = (N, P, E, node, prim, type)$ consists of a set N of *nodes*, a set P of *primitives*, a set E of *primitive ends* and functions

- $prim : E \to P$, assigning a primitive to each primitive end,
- $node : E \to N$, assigning a node to each primitive end,

- $type : E \to \mathbf{2} := \{src, snk\}$, assigning a type to each primitive end.

The structure of a connector can be described as a hypergraph with an additional coloring for the ends of the hyperedges. Primitives correspond to hyperedges and the primitive ends are the points where primitives coincide on nodes. Moreover, these primitive ends are colored, in the way that the function $type : E \to \mathbf{2}$ determines whether they are source or sink ends.

Definition 6.2 [Connector morphism] For two connectors C_1, C_2, a *connector morphism* $h = (h_N, h_P, h_E) : C_1 \to C_2$ is a tuple of functions $h_N : N_1 \to N_2$, $h_P : P_1 \to P_2$, $h_E : E_1 \to E_2$, such that the following diagrams commute:

$$\begin{tikzcd}
E_1 \arrow[r, "h_E"] \arrow[d, "node_1"'] \arrow[dr, phantom, "="] & E_2 \arrow[d, "node_2"] & E_1 \arrow[r, "h_E"] \arrow[d, "prim_1"'] \arrow[dr, phantom, "="] & E_2 \arrow[d, "prim_2"] & E_1 \arrow[r, "h_E"] \arrow[d, "type_1"'] \arrow[dr, phantom, "="] & E_2 \arrow[d, "type_2"] \\
N_1 \arrow[r, "h_N"'] & N_2 & P_1 \arrow[r, "h_P"'] & P_2 & \mathbf{2} \arrow[r, "id_{\mathbf{2}}"'] & \mathbf{2}
\end{tikzcd}$$

Connectors with their morphisms form the category **Con** of connectors. Continuing the interpretation of connectors as special hypergraphs, the definition of connector morphisms preserves the hypergraph structure and the type information of the primitive ends. This incorporates the invariant that source ends can be mapped only to source ends and sink ends only to sink ends. The definition of connector morphisms can also be used to define typed connectors and typed rewrite rules[4]. Like the DPO-approach, the idea of this kind of typing originates in the field of graph transformation [15]. Type information can be used to restrict the set of primitives used in a connector and further its topology. Intuitively, a connector C is typed by another connector T, if there is a morphism $t : C \to T$. In this setting, the connector C may use only those primitives that also occur in T. Further, typed morphisms and typed rewrite rules must preserve this information. A possible scenario may permit a component to attach to only a specific type of channels or that two different channel types may never join together. More information on typed graph transformation can be found in [15].

Based on the results by Lack and Sobociński [23], we can show that connectors as defined above indeed form a high-level replacement category. Hence, we can rely on the results and methodologies of the theory of HLR-systems.

Theorem 6.3 *Connectors and their morphisms form a HLR-category.*

Proof. It is known that hypergraphs form a HLR-category [30]. To verify this also with our specific notion of connectors, we use the result that *presheaves* are adhesive categories [23]. The definitions for connectors and their morphisms have been chosen such as to form a presheaf over the small category

$$\begin{tikzcd}
N & E \arrow[l, "node"'] \arrow[r, "prim"] \arrow[d, "type"] & P \\
 & \mathbf{2} &
\end{tikzcd}$$

[4] Here, the term *typed* has nothing to do with the function in Def. 6.1.

where we restrict the functor images of **2** to be binary sets. Hence, connectors form an adhesive category. To show that it is also a HLR-category, one more requirement has to be satisfied, namely that there must be an initial object [23]. The empty connector fulfills this requirement. □

With this result, we can define transformation rules for Reo. Based on our running example, we present a set of rules for a business process customization in the following section.

7 Example Rules

In this section we define transformation rules for the example introduced in Section 4. The scenario was based on an informal user request, which is used to customize a process.

In the following, we use a visual notation to define rewrite rules. However, the underlying formalism consists of the definition of connectors, connector morphisms and HLR-productions only. Each rule is represented in a separate figure, which consist of two parts: a left-hand side (LHS) and a right-hand side (RHS). The left-hand side is the pattern to be matched. The dashed arrows define the mapping to the right-hand side. This mapping implicitely defines the gluing structure K in Def. 5.1. Every node and every channel (primitive connector), that is mapped to the right-hand side will be preserved during the transformation. Parts in the LHS, that are not mapped, will be deleted. Parts in the RHS, that are not in the image of the mapping are added.

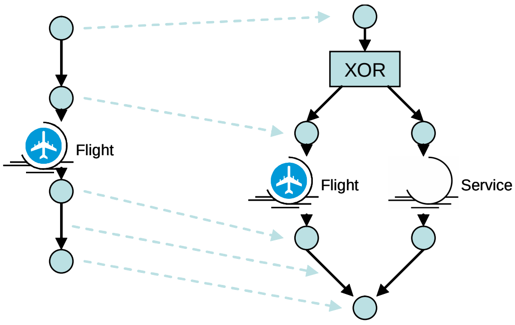

Fig. 3. Rewrite rule: Add flight alternative.

The first rule, shown in Figure 3, introduces an alternative transportation to a process. The pattern consists of a flight, that is already part of the initial process. The node mappings indicate that the upper *Sync* channel in the left-hand side is replaced by an exclusive router. In this example, we hide the internal structure of the exclusive router, which has been described in Section 3. Note also that the rule is parameterized by a service, that is derived from the user request. In our particular example, this parameter is a train service. After application of this rule to the original base process, we have a process with a train as an alternative as shown in Figure 2.b.

The second part of the request, booking a restaurant that is close to the hotel, is a bit more complicated. It consists of two rules shown in Figure 4 and 5. With the first

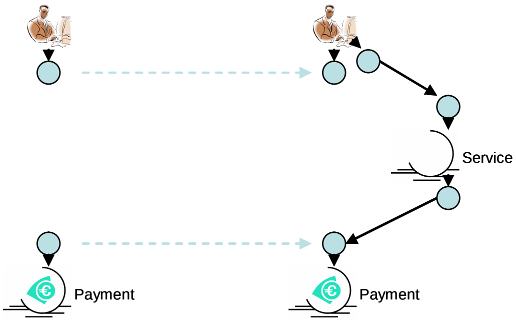

Fig. 4. Rewrite rule: Add travel package option.

rule, we add a travel package option which is a restaurant service in our example. The second rule allows us to synchronize two services based on their location. An external map service is used to check if the locations of the two services are nearby. The map service works as follows: it produces some output whenever two input messages refer to close locations, and does nothing otherwise. Thus, the second service successfully executes only if the map service has some output. After the application of this rule with a hotel and restaurant as its parameters, the resulting process looks as in Figure 2.c.

From the formal point of view, each of the presented example rules corresponds to a linear production (cf. Def. 5.1). This incorporates the constraint that the mappings, indicated by the dashed arrows, introduce a 1:1 correspondence between nodes or channels. Consequently, the rules apply only the expected basic operations, such as node / channel creation or deletion. Dropping the requirement of linear productions introduces a new kind of operations, i.e., splitting or joining of elements. In particular, it is possible to define node splittings and node joins, as specified in the Reo language. However, using this kind of operations also introduces new requirements that must be ensured at run-time. For instance, when splitting a node into two new nodes, what happens to the channels that coincide on this node must be well-defined.

Our example shows a possible scenario of how transformation rules can be used to build a travel package according to a user request. However, this very specific

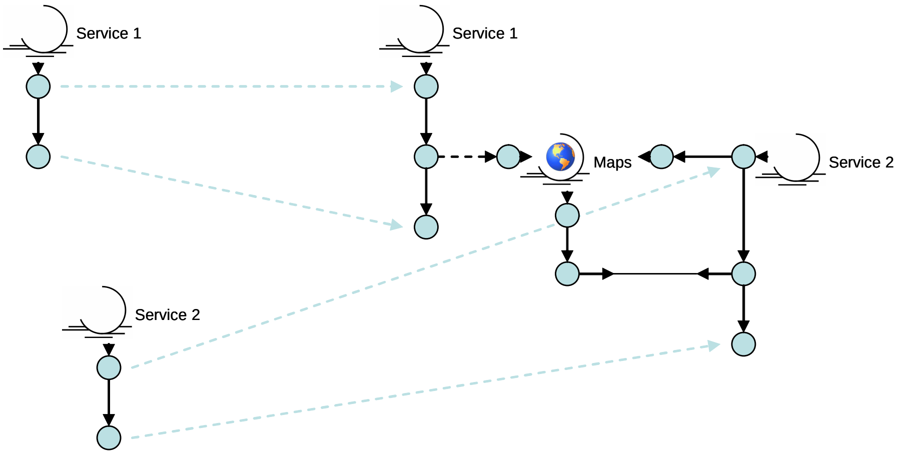

Fig. 5. Rewrite rule: Add map synchronization.

scenario should not hide the fact that i) coordination with Reo is not limited to the domain of services, and ii) the idea of high-level replacement systems is a general transformation meta-language that can be instantiated with arbitrary structural models.

8 Implementation

The examples given in this paper show that a good tool support is crucial for any kind of application in the field of coordination. The Eclipse Coordination Tools (ECT) project [14,22] aims at unifying a number of existing Reo-related tools into one integrated environment. ECT is implemented in Java as a set of plug-ins for the Eclipse platform [5] . Currently the framework consists of the following parts:

(i) graphical editors for Reo connectors and constraint automata [2],

(ii) simulation plug-in for Reo, that generates Flash animations on the fly,

(iii) conversion tool from Reo connectors to constraint automata,

(iv) Java code generation from constraint automata,

(v) model checking tool for constraint automata [21].

The existing Java code generation plug-in produces a centralized implementation of connectors using simple threads. Ongoing work also includes an implementation based on constraint programming, and a distributed implementation of Reo, which will be implemented using the Actors extension of the Scala language [18].

Since the Reo coordination tools are implemented using standard Eclipse modeling framework, we can directly use existing model transformation frameworks to define connector transformations. In particular, we use the Tiger EMF Model Transformation Framework (EMT) [6] which is an implementation of the graph-based HLR transformation approach, as presented in this paper. Our Reo model can be imported into the Tiger framework, where transformations are defined using a visual editor. Future work includes an integration of the run-time libraries of the transformation framework with the implementation of Reo.

9 Conclusions and Future Work

Reo is an expressive language for coordinating autonomous software components. Applying high-level replacement systems to Reo, we show how evolving software systems can be maintained and complex reconfigurations can be realized. As shown in our running example, the proposed approach successfully applies to a number of challenges in the field of service-oriented computing.

In the next step of our work, we plan to investigate to what extent it is possible to reason about the changes in the behavior of evolving connectors. In particular, it is interesting to extract a class of transformations that preserve certain aspects of the behavior of a connector. Since the semantics of Reo connectors can be given with so-called *constraint automata* [2], behavior preserving transformations

[5] http://www.eclipse.org

can be classified by postulating bisimilarity between the input and the result of a transformation (cf. [17]).

In classical HLR-systems, transformations are not limited to a single rule application. In general, multiple rules can be applied in combination (sequentially or even in parallel). The so-called negative application conditions can be used to define patterns that must not be matched to apply a rule. As mentioned already before, type information for connectors can be used to restrain the topologies of connectors. Further, it is possible to enforce a protocol that determines when to apply which rule and how often. All these concepts should be considered in the context of coordination, since they truely increase the expressiveness of transformations.

When transformation rules are used to achieve some objectives, e.g., build a user request, some transformation rules cannot be applied together, since one rule may potentially neglect the effects of previously applied transformations. It is possible to use AI techniques such as planning or formal verification to find a sequence of transformation rules that satisfies the provided objectives. In general, it is interesting to have a mechanism for analysis of various kind of relations between transformation rules: dependencies, mutual exclusiveness, joint effects, etc.

From the point of service-oriented computing, the scenarios provided in this paper are not exhaustive. For example, one may use HLR techniques to specify the composition of several processes as a set of transformations, to apply business rules that change the structure of a process, or to have run-time reconfigurations to deal with changes in the infrastructure. To make full use of high-level replacement systems in SOA, we also plan to address issues of QoS, service-level agreements, security, transactional behavior, and compensation. Initial work in this area has been done already in [3] and [26].

Our immediate future work involves the implementation of the proposed framework, i.e., the deployment of components and connectors in distributed environments.

References

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

[2] F. Arbab, C. Baier, J. Rutten, and M. Sirjani. Modeling component connectors in Reo by constraint automata. Technical report, CWI Technical Report SEN-R0304, ISSN 1386-369X, 2003.

[3] F. Arbab, T. Chothia, S. Meng, and Y.-J. Moon. Component connectors with qos guarantees. In *Proceedings of 9th International Conference on Coordination Models and Languages, Coordination'07, LNCS 4467*, pages 286–304, 2007.

[4] F. Bachmann and L. Bass. *Managing Variability in Software Architecture.* ACM Press, 2001.

[5] T. Basten and W. M. P. van der Aalst. Inheritance of behavior. *J. Log. Algebr. Program.*, 47(2):47–145, 2001.

[6] E. Biermann, K. Ehrig, C. Koehler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical defnition of in-place transformations in the eclipse modeling framework. In *Model Driven Engineering Languages and Systems (MoDELS'06)*, 2006.

[7] F. Casati, M. Sayal, and M. Shan. Developing e-services for composing e-services. In *13th Int. Conf. on Advanced Information Systems Engineering (CAiSE)*, 2001.

[8] L. Chen, N.R. Shadbolt, C. Goble, F. Tao, S.J. Cox, C. Puleston, and P. Smart. Towards a knowledge-based approach to semantic service composition. In *2nd Int. Semantic Web Conf. (ISWC2003)*, LNCS 2870. Springer, 2003.

[9] D. Clarke. Reasoning about connector reconfiguration I: Equivalence of constructions. Technical report, CWI Technical Report SEN-R0506, 2004.

[10] D. Clarke. Reasoning about connector reconfiguration II: Basic reconfiguration logic. *Proc. of FSEN 2005. Tehran, Iran*, 2005.

[11] D. Clarke, D. Costa, and F. Arbab. Connector colouring I: Synchronisation and context dependency. *Proc. of Foclasa 2005, San Francisco*, 2005.

[12] A. Corradini, H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, and A. Wagner. Algebraic approaches to graph transformation - Part I: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, pages 247–312. World Scientific, 1997.

[13] T. Curran and G. Keller. *SAP R/3 Business Blueprint: Understanding the Business Process Reference Model*. Upper Saddle River, 1997.

[14] Eclipse coordination tools. http://homepages.cwi.nl/~koehler/ect.

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

[16] H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Mathematical Structures in Computer Science*, 1(3):361–404, 1991.

[17] H. Ehrig and B. König. Deriving bisimulation congruences in the DPO approach to graph rewriting. In *FoSSaCS*, pages 151–166, 2004.

[18] P. Haller and M. Odersky. Actors that Unify Threads and Events. In *International Conference on Coordination Models and Languages*, Lecture Notes in Computer Science (LNCS), 2007.

[19] G. Halmans and K. Pohl. Communicating the variability of a software-product family to customers. *Software and System Modeling*, 2(1):15–36, 2003.

[20] R. Heckel and A. Cherchago. Structural and behavioral compatibility of graphical service specifications. *Journal of Logic and Algebraic Programming*, 2006. To appear.

[21] S. Klueppelholz and C. Baier. Symbolic model checking for channel-based component connectors. In *FOCLASA'06*, 2006.

[22] C. Koehler, A. Lazovik, and F. Arbab. ReoService: coordination modeling tool. In *ICSOC-07*, 2007.

[23] S. Lack and P. Sobociński. Adhesive categories. In *Foundations of Software Science and Computation Structures, FoSSaCS '04*, volume 2987 of *LNCS*, pages 273–288. Springer, 2004.

[24] A. Lazovik, M. Aiello, and M. Papazoglou. Planning and monitoring the execution of web service requests. *Journal on Digital Libraries*, 2005.

[25] J. Mendling, J. Recker, M. Rosemann, and W. M. P. van der Aalst. Towards the interchange of configurable EPCs. In *EMISA*, pages 8–21, 2005.

[26] S. Meng. Qccs: A formal model to enforce QoS requirements in service composition. In *Accepted by 1st IEEE and IFIP International Symposium on Theoretical Aspects of Software Engineering, TASE'07*, 2007.

[27] OTA. Open travel alliance. http://www.opentravel.org/.

[28] J. Padberg. Survey of high-level replacement systems. Technical report, Technical University of Berlin, 1993.

[29] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *Int. Semantic Web Conf. (ISWC)*, pages 333–347, 2002.

[30] D. Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In M.J. Plasmeijer M.R. Sleep and M.C.J.D. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, pages 201–213, 1993.

[31] M. Rosemann and W. M. P. van der Aalst. A configurable reference modelling language. *Inf. Syst.*, 32(1):1–23, 2007.

[32] SAP. Online documentation mySAP ERP. http://help.sap.com, 2005.

[33] E. Sirin, B. Parsia, and J. Hendler. Filtering and selecting semantic web services with interactive composition techniques. *IEEE Intelligent Systems*, 19(4), 2004.

[34] S. Thakkar, J. Ambite, C. Knoblock, and C. Shahabi. Dynamically composing web services from on-line sources. In *AAAI Workshop Intelligent Service Integr.*, 2002.

[35] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating DAML-S web services composition using SHOP2. In *ISWC-03*, 2003.