Validating Reconfigurations of Reo Circuits in an e-Banking Scenario

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy
and
LIACS, Leiden University
The Netherlands
maurice.terbeek@isti.cnr.it

Fabio Gadducci
Department of
Computer Science
Largo Pontecorvo 3c
Pisa, Italy
gadducci@di.unipi.it

Francesco Santini
EPI Contraintes, INRIA
Domaine de Voluceau
Rocquencourt
Le Chesnay, France
francesco.santini@inria.fr

ABSTRACT

We formalize dynamic reconfiguration of Reo circuits (which can be thought of as multi-party communication infrastructures built from primitive channels) through graph transformation, and apply it to a scenario from the Finance domain: a critical infrastructure controlling the business process of an e-banking system. In this scenario, reconfiguration is triggered as soon as the communication buffers reach specific predefined thresholds of congestion. These constraints are implemented inside the Reo model by associating suitable predicates to channels, thus extending previous results on the use of graph transformation for the reconfiguration of Reo's graphical structures.

Categories and Subject Descriptors

G.2 [**Discrete Mathematics**]: Graph theory, Applications

General Terms

Design, Theory

Keywords

Graph rewriting; Coordination; Reo; Reconfiguration; QoS

1. INTRODUCTION AND MOTIVATIONS

Reo [1] is a channel-based exogenous coordination language wherein complex coordinators, called connectors, are compositionally constructed from simpler ones. This graphical language has a strong formal basis and promotes loose coupling, distribution, mobility and dynamic reconfigurability. Application designers can use Reo as a "glue code" language for the compositional construction of connectors that orchestrate the cooperative behavior of instances of components or services in a component-based system or a service-oriented application. A variety of semantic models exist for Reo [6]. This formal basis of Reo guarantees possibilities for both model checking and verification [7], as well as well-defined execution semantics of a service composition [11].

Graph transformation allows substructures of a graphical structure to be rewritten by rules consisting of patterns that define the changes to be performed on the structure whenever there is a matching sub-structure. Additional (positive or negative) conditions may limit the applicability of such rules. This flexibility allows an engineer to specify in an abstract way when and how a critical architecture or infrastructure can be reconfigured (and eventually execute the reconfigurations) [13]. Recently, there have been several approaches to the reconfiguration of component connectors in Reo [10, 9] based on graph transformation.

Our paper should be considered as a further contribution to the line of research sketched above. It describes graph transformation based reconfigurations for Reo circuits and applies them to a real-world scenario from the Finance domain: a critical infrastructure controlling the business process of an e-banking system. In this particular scenario, reconfiguration is triggered as soon as the communication buffers reach specific predefined thresholds of congestion. Since these constraints are implemented inside the Reo model by associating suitable predicates to channels: our work should then be considered as an extension of the most recent approaches to behavior preserving reconfigurations, as proposed in [9].

The e-banking scenario we consider here was inspired by a study of real-world requirements at Credit Suisse S.A. in Luxembourg [4]. It was specifically developed to fit the EM-Cube enterprise modeling framework presented in [3]. In this scenario, people select shares they want to buy, put them into a basket and, finally, pay for them using electronic means like credit cards. Once in a while, an account statement is sent to clients by e-mail, summarizing all their past transactions. As a running example, we show two different configurations of this e-banking system and a way to switch from one to the other. The first case represents a redundant, safer configuration: if a request fails in either one of the branches of the original configuration, the other branch can provide a potential backup. In the second configuration, on the other hand, we assume that incoming orders are i) not replicated, but instead processed simultaneously by an upper and a lower branch and ii) arbitrarily distributed to the available services, in order to maximize the overall capacity of requests that the IT infrastructure can process.

The motivation for allowing to switch between the aforementioned two configurations stems from the assumption

ISARCS'13, June 17–21, 2013, Vancouver, BC, Canada.

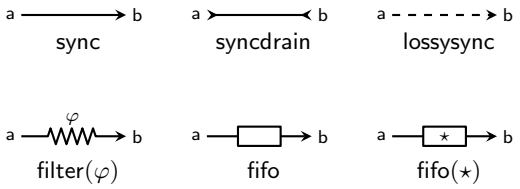


Figure 1: Graphical syntax of six primitive channels.

that the nature of the incoming requests may change over time. For instance, typical business situations may differ greatly between daytime and nighttime, just as well as there can in general be peak and off-peak times leading to different policies or different resources for the management of incoming requests. We plan to control the switching between configurations by setting some variables to certain values through a graph rewriting model. Each configuration is defined in the Reo language, and the modeling and simulation processes are accomplished in Modelica[1], a non-proprietary, object-oriented, equation based language for the modeling of complex physical systems.

The paper is organized as follows. In Section 2.1 we familiarize the reader with Reo, while in Section 2.2 we complete the required background by reporting the basic notions on graph rewriting. Our e-banking scenario is introduced in Section 3. In Section 4 we define graph transformations for the dynamic reconfiguration described in our scenario and in Section 5 we evaluate their application. Finally, in Section 6 we draw our conclusions and outline ideas for future work.

2. BACKGROUND

In the remainder of this section we summarize some background information that is needed to understand our contribution. We first briefly introduce the Reo coordination language [1], after which we report the basic notions of graph transformation [5].

2.1 Reo

The language *Reo* [1] facilitates compositional construction of *circuits*: communication mediums that coordinate interacting parties, each built from a number of simple *channels*. Every channel in Reo has exactly two *ends*, and each such end has exactly one of two types: a channel end either accepts data items—a *source end*—or it offers data items—a *sink end*.[2] Figure 1 shows six different primitive channels at the disposal of Reo users; Figure 2 describes their behavior in words. Interestingly, Reo does not fix which particular channels one may use to construct circuits with. Instead, Reo supports an *open-ended set of channels*, each of which exhibits unique behavior. This feature enables users of Reo to define their own channels, tailored to their specific needs.

We call the act of "gluing" channel ends together to build circuits *composition*. One can think of composite circuits as graphs with nodes and edges (channels) and compare their behavior to plumbing systems. In such systems, "fluids" flow through "pipes and tubes" past "fittings and valves". Similarly, in Reo circuits "data items" flow through "channels"

[1] `http://modelica.org/`

[2] However, channels do not necessarily have *both* a source *and* a sink end: they can also have two source ends (as the syncdrain channel in Figure 1) or two sink ends.

Name	Behavior
sync	Atomically fetches an item on its source end a and dispatches it on its sink end b.
syncdrain	Atomically fetches (and loses) items on both of its source ends a and b.
lossysync	Atomically fetches an item on its source end a and, non-deterministically, either dispatches it on its sink end b or loses it.
filter(φ)	Atomically fetches an item on its source end a and dispatches it on its sink end b if this item satisfies the filter constraint φ; loses the item otherwise.
fifo	Atomically fetches an item on its source end a and stores it in its buffer.
fifo($\star$)	Atomically dispatches the item $\star$ on its sink end b and clears its buffer.

Figure 2: Channel behavior.

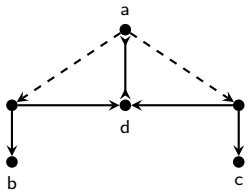


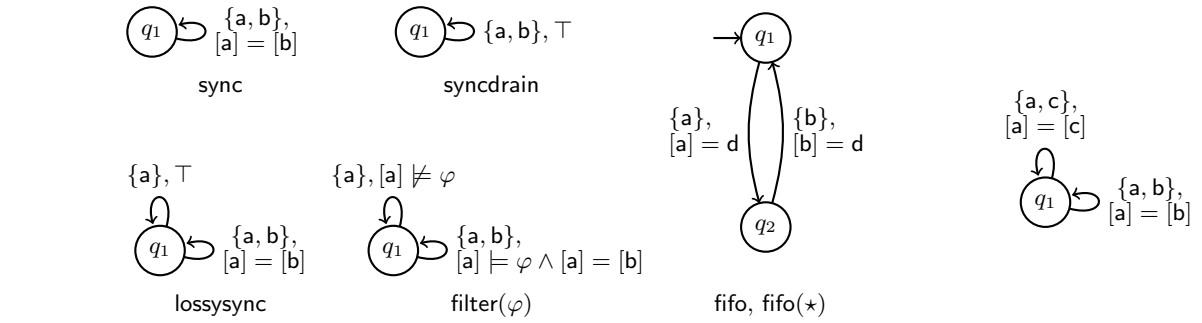
Figure 3: Graphical syntax of a XOR circuit.

(along edges) past "nodes". Usually, the interacting parties themselves supply the data items that flow through the circuits they communicate through. To this end, every circuit defines an interface. Such an interface consists of the *boundary nodes* of a circuit: parties write and take data items only to and from boundary nodes.

Nodes are used as execution logical points, where execution over different channels is synchronized. A node is either *source*, *sink* or *mixed*, depending on whether all channel ends that coincide on that node are source ends, sink ends or a combination of the two. Considering the example in Figure 3 (whose semantics is explained in the next paragraph), a is a source node, b and c are sink nodes and, finally, d is a mixed node. A mixed node non-deterministically selects and takes a suitable data item offered by one of its coincident sink channel ends and replicates it into all of its coincident source channel ends.

Figure 3 shows a circuit, named XOR, that one can construct from the channels in Figure 1. This circuit implements an *exclusive router* of messages: its intuitive behavior is that data obtained as input through node a is delivered to one of the output nodes b or c. If both b and c are willing to accept data, then node d non-deterministically selects which side of the connector will succeed in passing the data. We will repeatedly exploit the XOR circuit in our e-banking scenario in Section 3.

In general, one derives the behavior of a circuit from the behavior of the channels and nodes that it consists of: all circuits exhibit *compositionality*. In this paper we refer, when needed, to Reo's semantics based on *constraint automata* [2]. Constraint automata resemble classical finite state machines in the sense that they consist of finite sets of states and transitions. States represent the internal configurations of a circuit; transitions describe its atomic coordination steps. Formally, we define a transition as a tuple of four elements: a source state, a *synchronization constraint*, a *data constraint* and a target state. A synchronization constraint specifies



(a) Channels.

(b) Circuit: XOR.

Figure 4: The constraint automata related to the primitive channels in Figure 1 and to the XOR circuit in Figure 3.

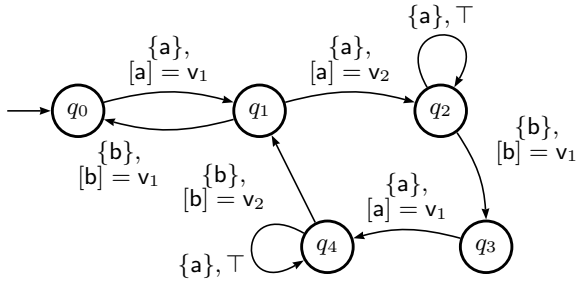


Figure 5: The behavior of a lossyNfifo channel with two slots.

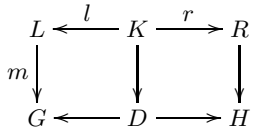


Figure 6: A rule application in the DPO approach.

which nodes synchronize—i.e., through which nodes a data item flows—in some coordination step; a data constraint specifies which particular data items flow in such a step. Figure 4 shows the constraint automata of the channels and circuits presented earlier in this section (in Figure 1 and Figure 3, respectively).

However, as said before, various *semantic models* exist to formally describe the behavior of Reo circuits [6]. These models, among other applications, enable one to reason about the correctness of Reo circuits. Kokash et al., e.g., employ the `mCRL2` toolkit to verify the correctness of Reo translations of business process models [7].

We now introduce a new primitive Reo channel, called lossyNfifo channel, which will also be extensively used in our e-banking scenario in Section 3. It can be graphically represented as a fifo channel with a dashed incoming arrow (see Figure 7). This channel represents an n-position buffer, whose informal semantics is that it is allowed to start losing data when the buffer is completely full; otherwise, all its slots are filled in order. In Figure 5 we formally explain its behavior by showing a constraint automaton for a 2-slot lossyNfifo, in which variables v_1 and v_2 represent the two slots. In states q_2 and q_4 the two slots are already full, but it is still possible to receive elements through port a, consequently discarding them (this is represented by the self-loops on q_2 and q_4). Operations on port b free one of the slots by consuming the first-stored message (it is a FIFO scheme) and then outputting it from the circuit to the following component connected to port b.

2.2 Graph Transformation

The generic name *algebraic graph transformation* indicates a class of formalisms adopted in the literature for the local manipulation of graphical structures. This is accomplished by choosing a set of rules, by precisely specifying the kind of objects you may want to manipulate and, finally, by denoting how a rule can be applied to an object.

The most well-known proposal is probably the so-called DPO approach [5]. It is summed up by the two squares in Figure 6. Assuming to have just ordinary graphs (as in Reo's graphical representation), the rule is specified by two graph morphisms, $l : L \to K$ and $r : K \to R$ (the former has to be injective, too), denoting how the left-hand side graph L is rewritten into the right-hand side graph R, while the intermediate graphs are used to keep track of the connection between the items of the left-hand side and that of the right-hand side.

The central idea is that both squares have to be somehow minimal. In the DPO approach, this is achieved by requiring the squares to be what in categorical terms is presented as a *pushout*: e.g., H is obtained as the disjoint union of D and R, with the proviso that the items which already occur in K are actually collapsed. Operationally, the application of a rule to a graph G consists of three steps. First, the match morphism $m : L \to G$ is chosen, providing there is an occurrence of L in G. Next, all items of G matched by $L \setminus l(K)$ are removed, and those items identified by $l \circ m$ are coalesced, leading to the context graph D. Finally, the items of $R \setminus r(K)$ are added to D, further coalescing those nodes identified by r, obtaining the derived graph H.

For our e-banking scenario, we will consider a simple variant of graphs (adopted early on for modeling reconfiguration in Reo; see, e.g., [10]) in which each edge is actually equipped

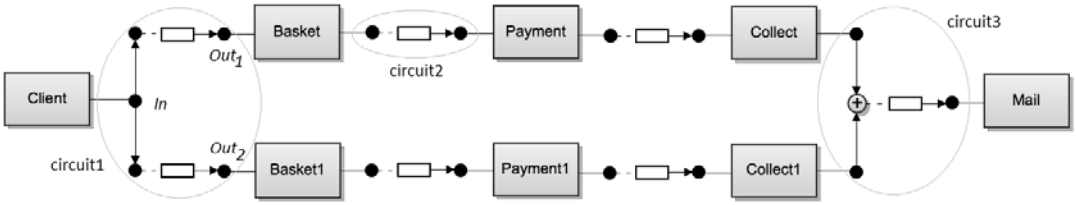


Figure 7: The complete Reo circuit diagram for Case 1.

with a predicate, and the morphisms between graphs actually preserve such predicates. All the relevant properties of the DPO approach (namely, the uniqueness of the result of a rule application for a given match morphism, the possibility to define a notion of parallel execution of rules, as well as some confluence properties; again, see [5]) continue to hold also for our variant.

3. e-BANKING SCENARIO

The critical architectural scenario we consider as running example was inspired by a study of real-world requirements at Credit Suisse S.A. in Luxembourg [4]. Among the alternatives, for its simplicity we opted for a business process that is about buying shares over the Internet by the help of an e-banking system. In this scenario, people select shares they want to buy, put them into a basket and, finally, pay for them using electronic means like credit cards. Once in a while, an account statement is sent to clients by e-mail summarizing all their past transactions.

In order to be able to discuss operational risks related to IT systems [4], we decided to model a service landscape, which implements and enables this business process. The model has been described using Reo and the obtained circuit is shown in Figure 7, which depicts "Case 1" (to distinguish it from the forthcoming model of "Case 2"). The routing between the different services is realized by circuits that internally implement different Reo connectors (see Section 2.1), which are ideally suited to support the exogenous coordination of services in an IT landscape. These circuits contain buffers (represented as the lossyNfifo channels introduced in Section 2.1), which can run full. In this case, incoming requests are dropped and, therefore, lost.

In this model, a stream of client orders flows into the system. Each order has a certain business value in euro cents, comes at a specific point in time and adds up to a concrete basket until that basket is complete.

3.1 Case 1

The input stream in model "Case 1", produced by a *client* service, is replicated by circuit 1 (in Figure 7) to be processed by an upper (*basket*, *payment* and *collect* services) and lower branch (*basket1*, *payment1* and *collect1* services). All these services are pairwise identical, considering the two branches. This is because whenever a request fails in either of the two branches, the other one always provides a potential backup. In each branch, the coordination between two consecutive services is modeled by circuit 2 (in Figure 7). The streams out of the two branches are merged (circuit 3 in Figure 7), collected and forwarded from time to time to a *mail* service, which sends out account statements by e-mail.

The *basket* services handle the bookkeeping of all incoming requests until their corresponding baskets are complete.

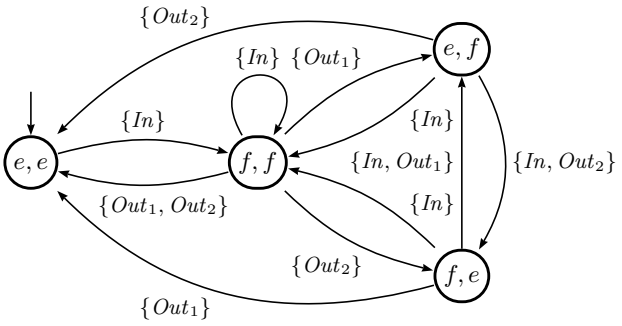


Figure 8: The behavior of circuit 1 in Figure 7 considering the two lossyNfifo channels as having only one slot. The state labels denote the filling status of the two fifo channels: *f*ull or *e*mpty. The edge labels In, Out_1 and Out_2 stem from those shown in Figure 7.

Once a basket is complete, the whole basket is sent to the *payment* service. Both services can fail. This behavior is realized by failure distributions: each service has its own. The *collect* services collect all successfully paid baskets and flush them out to the *mail* service, based on a given timetable. In addition, the *payment* services and the *mail* services use different delay distributions.

The node labeled with "+" in circuit 3 (in Figure 8) is a *join* node. This type of node is often used to perform some kind of computation over the incoming data, before routing it to its source channel ends. In Figure 8 it is used to send to the *mail* service only one of the two identical requests replicated in the two branches; if one (or both) of them was lost before, this check is superfluous.

As an example, Figure 8 shows the behavior of circuit 1 (in Figure 7) as a constraint automaton in which data constraints are omitted for the sake of clarity, i.e., only the synchronization constraints on nodes In, Out_1 and Out_2 are shown. These nodes represent the endpoints where the *client*, *basket* and *basket1* components perform their offer and accept operations on circuit 1.

3.2 Case 2

We now focus on a different model, which we call "Case 2". In this case, we assume that incoming orders are not replicated and simultaneously processed by an upper and a lower branch, but they are arbitrarily distributed to the available services in order to maximize the overall capacity of requests that the IT landscape can process. The main difference compared with model "Case 1" in Figure 7 is the use of different Reo circuits to coordinate the services being used. In fact, services behave exactly in the same way. In Figure 9 we show the complete Reo circuit diagram related to "Case 2".

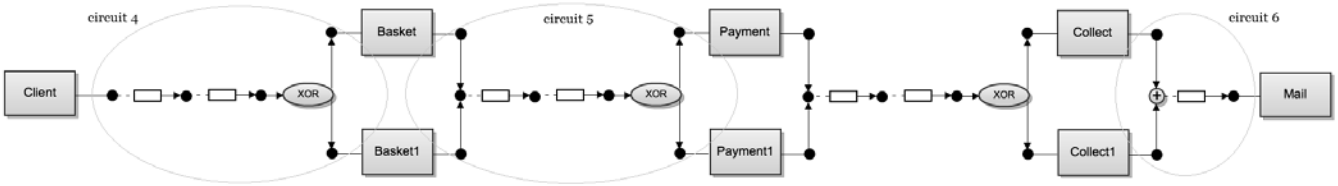


Figure 9: The complete Reo circuit diagram for Case 2.

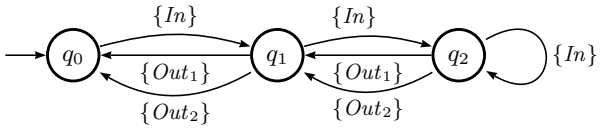


Figure 10: The behavior of circuit 4 in Figure 9, considering the two lossyNfifo channels as having only one slot.

Note that in Figure 9 we keep the join node as used in Figure 7, even if it is unnecessary since it could be replaced by a simple standard node (because in "Case 2" we do not replicate the baskets and, consequently, we do not need to check whether the same basket has already been served before by the *mail* service). This simplifies the rewriting proposed in Section 4, since we may avoid to design a rule for the rewriting of this circuit. Nevertheless, in the implementation that will be shown in Section 5 we implement the simpler circuit without the *join* node of "Case 1".

In addition, in Figure 10 we show the behavior (as a constraint automaton) of circuit 4 (see Figure 9), which replaces circuit 1 in Figure 7 as the first step of the data flow. Note also that the two lossyNfifo channels in Figure 9 are positioned *before* the XOR sub-circuit. This is because if they were positioned *after* the XOR, then the latter could send a message to a full buffer and lose it. As a consequence, in Figure 9 messages are lost only if both queues are full.

Assuming that the nature of the incoming requests may change over time (e.g., typical business situations may differ greatly between daytime and nighttime), we may need to implement a sort of switch that either selects model "Case 1" or model "Case 2" for processing the incoming requests. Different loads of requests (i.e., shares in this scenario) may lead to the loss of a large percentage of purchases, thus lowering the trust of customers in the offered service. This is precisely what makes the infrastructure critical. In [4] the authors measured the different resilience of both "Case 1" and "Case 2". From a theoretical point of view, such a switch (in between model "Case 1" and model "Case 2") enables the dynamic reconfiguration of the Reo circuits in the given IT landscape, by either selecting the first or the second configuration. In the sequel, this dynamic reconfiguration is modeled and implemented by graph transformation.

4. REWRITING FOR RECONFIGURING

The use of graph transformation has been proposed early on for the reconfiguration of Reo specifications. After all, a circuit is described by means of a graphical structure and the substitution of its components (or whole sub-circuits) can thus be naturally modeled by the manipulation of edges (or sub-graphs, respectively).

The most elaborate proposal has been illustrated in [8], extended as [9]. In order to guarantee the reconfiguration to be sound with respect to the circuit behavior, the authors equip each item of the graphical representation of a circuit with an automaton, guaranteeing that the transformation actually preserves it. Our goal here is much simpler. We only consider the graphical representation, and we enrich each of its edges with a predicate. This represents a sort of *Quality of Service* (QoS) measure, possibly an abstraction of an automaton, which is considered to be the signal for the start of the reconfiguration.

Our proposal is thus an immediate extension of the one presented in [9]. For the sake of simplicity, we avoid considering the presence of automata, which would simply be preserved by our reconfiguration, and we focus instead on the additional predicates. The improved performance of the reconfigured model (i.e., "Case 2") is then made explicit in the simulation presented in the next section.

Consider the rule presented in Figure 11: the three circuits composing the rules are arranged horizontally, and the morphims are suggested by the position of each item. The only ambiguities concern the isolated nodes in the intermediate graph, which are mapped into the rightmost and leftmost node of the right-hand side. The presence of the f is a predicate that is signaling the possible congestion of the channel, i.e., it may be full. It can easily be obtained as an abstraction of the channel behavior by stating, e.g., that more than a certain amount of slots of the buffer are actually in use (i.e., filled). As soon as the threshold is exceeded, the rule becomes enabled and it may be applied. The manipulation tends to replace the (first) component of the duplicated branch in order to turn it into a single branch with twice the buffering capacity. The need of reconfiguring also the connections between components can be achieved by temporarily switching off and on the basket components, as long as they are empty: the switching is modeled by the deletion and recreation of the component after the step, while the presence of a predicate e on the basket component signals that the rule can be applied only after the baskets have indeed been emptied.

The application of the rule to the circuit of Figure 7 is presented in Figure 12. The result of the reconfiguration is the change in behavior of the overall circuit, even if the semantics of each channel is preserved. Now the initial part of the circuit has passed to a linear treatment of the client input, and this is reflected in the overall behavior. One of the lossyNfifo channels verifies the predicate f, and this fact enables the application of the rule. The baskets are still empty, while the predicates on the other edges are not considered here: they are the same before and after the reconfiguration steps, and they were not relevant for the enabling of the rule.

Further applications of rule 2 (in Figure 13) and rule 3 (in Figure 14) will turn the circuit into the one presented in

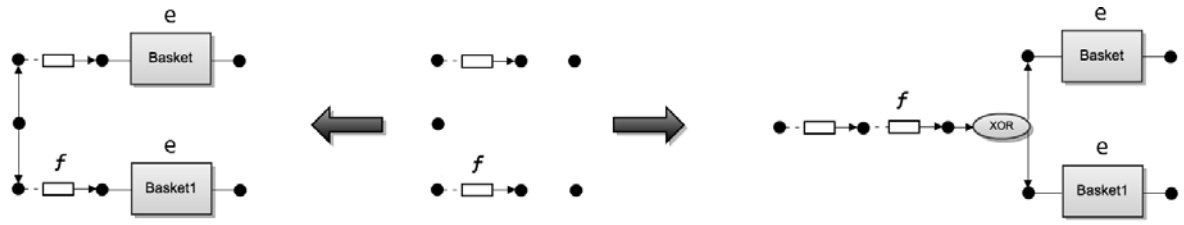


Figure 11: Rewriting rule 1.

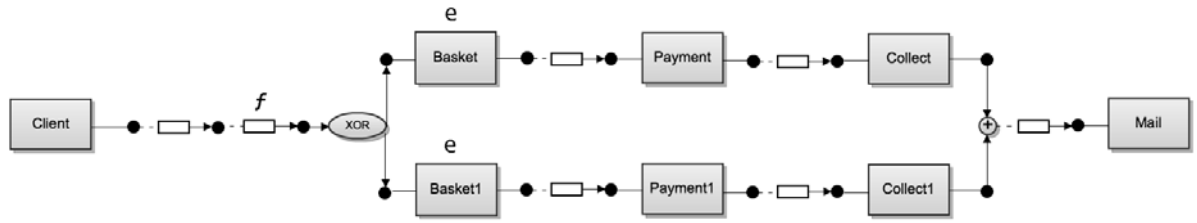


Figure 12: Rewriting of Figure 7 obtained by applying rule 1 (in Figure 11).

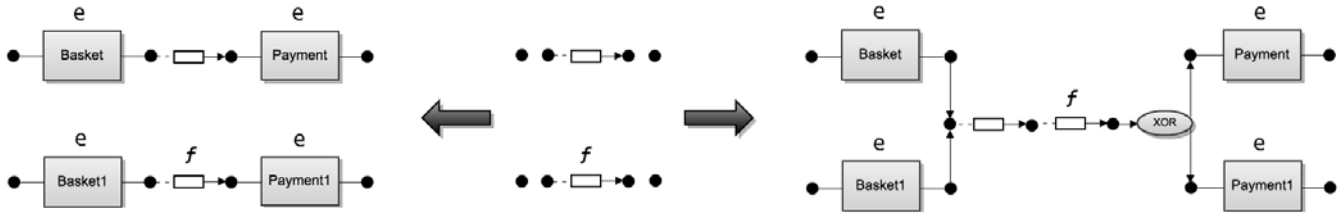


Figure 13: Rewriting rule 2.

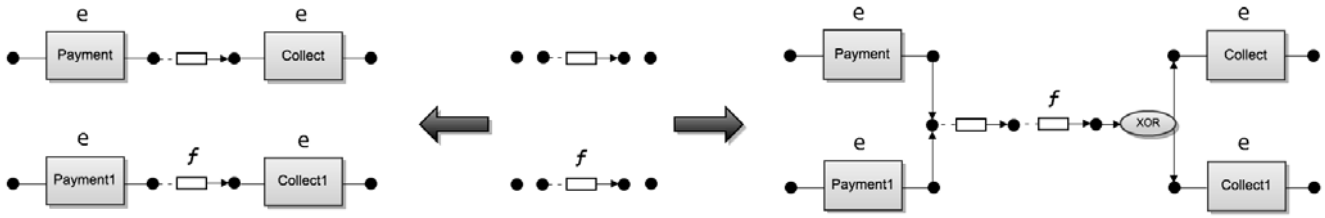


Figure 14: Rewriting rule 3.

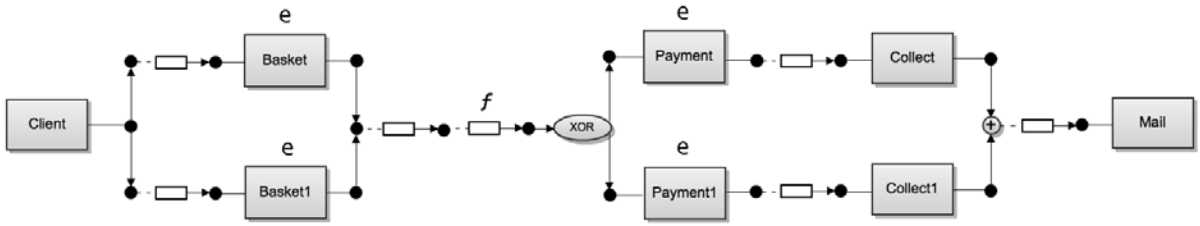


Figure 15: Rewriting of Figure 7 obtained by applying rule 2 (in Figure 13).

Figure 9. Note, however, that there is no forced sequence of reconfigurations. Any order might be chosen, as long as the single rules are enabled. This could, e.g., occur if the payment is delayed and the items from the basket start to fill the buffers. A possible solution is shown in Figure 15, where the initial part of the circuit still duplicates the data, while the second part no longer does so.

5. SIMULATION AND VALIDATION

Dymola[3] is a simulation engine that supports the simulation of Modelica models. Modelica is a freely available, special-purpose object-oriented language for the specification of large, complex and heterogeneous physical systems. It realizes hierarchical model composition, encompasses libraries of truly reusable components and connectors as well as composite non-causal connections. Modelica is suited for multi-domain modeling such as, e.g., mechatronic models in robotics, automotive and aerospace applications involving mechanical, electrical, hydraulic and control sub-systems, process-oriented applications and the generation and distribution of electric power. Models in Modelica are mathematically described by differential, algebraic and discrete equations. Modelica is designed such that available, specialized algorithms can be used to enable the efficient handling of models having more than a hundred thousand equations. Modelica is suited and used for hardware-in-the-loop simulations and embedded control systems.

In Modelica, the basic structuring element is a *class*. There are seven restricted classes with specific names such as, e.g., *model*, *type* (a class which is an extension of built-in classes, such as *Real*, or of other defined types) and *connector*, i.e., a class which does not have equations and can be used in connections; connectors are graphically connected by the links between components like, e.g., in Figure 16 and Figure 17. For a valid model it is fully equivalent to, e.g., replace the model and type keywords by the keyword class, because the restrictions imposed by such a specialized class are fulfilled by a valid model.

Handling large models means careful structuring in order to reuse model knowledge. A model is built-up from:

- basic components: Real, Integer, Boolean, String, etc.;
- structured components, for hierarchical structuring;
- component arrays, to deal with real matrices, arrays of sub-models, etc.;
- equations and/or algorithms (assignment statements);
- connections;
- functions.

Dymola supports graphical compositions of Modelica models, and fast simulation with symbolic pre-processing of them.

In Figure 16 and Figure 17 we show the block-oriented Dymola implementation for the two Reo circuit diagrams in Figure 7 and Figure 9, respectively.

[3]`http://www.3ds.com/products/catia/portfolio/dymola`

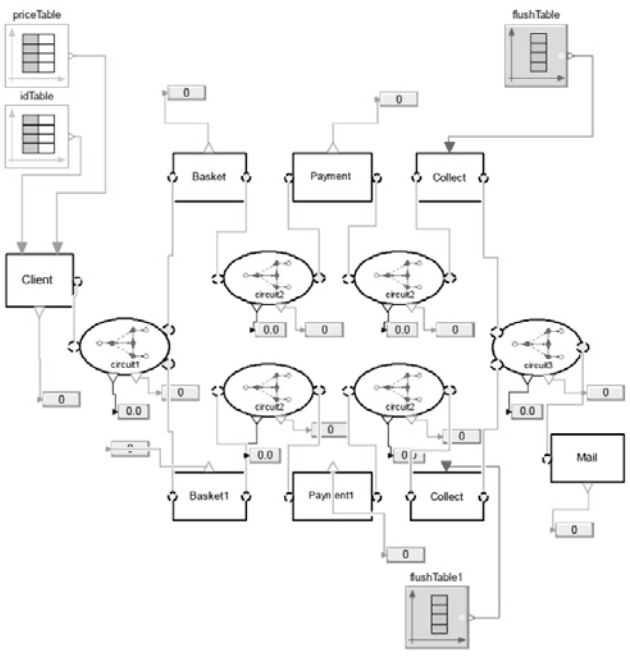


Figure 16: Case 1 as implemented in Dymola [4].

5.1 Case 3

Figure 18 shows the Modelica model "Case 3" in which we apply the rewriting that switches from "Case 1" to "Case 2" in case of need, i.e., when the maximum filling percentage among all lossyNfifo channels in the currently used Case is above a given (by the system designer) warning threshold. Circuit 7 in Figure 18 models the Reo coordination between the two cases: in our simulation we will start delivering data to "Case 1", after which we will switch to "Case 2", as we have formally shown in Section 4. The "Rewriting Rule" block in Figure 18 models the rewriting from "Case 1" to "Case 2": when at least one of the lossyNfifo channels in "Case 1" is full for more than a threshold percentage (this is the triggering condition), then circuit 7 waits for the complete emptying of all the lossyNfifo channels before it resumes routing the incoming requests to "Case 2". In the meantime, pending requests from the *client* service are stored inside circuit 7.

The two outputs from the "Rewriting Rule" block (in Figure 18) are needed to *i*) open/close the dispensing of requests from circuit 7 and *ii*) decide which of the two cases it is that circuit 7 routes requests to. Therefore, in case of switching between the two cases, the "Rewriting Rule" block uses these two discretely valued parameters to first close ($open = 0$) the output flow of messages to "Case 1", after which it then, once all pending messages have been served inside this case, switches the second flag (*case1or2*) in order to start routing messages to "Case 2" (opening the output flow to it again, i.e., $open = 1$). Its single input, whose value is continuously sampled, conveys the maximal filling status (in percentages) among all lossyNfifo channels in "Case 1"; the threshold is set to this value.

Figure 19 shows the chart obtained by simulating "Case 3" until time 420 (seconds). To limit this time we set $\alpha = 10\%$, which means the triggering condition is fired when the maximal filling status among all lossyNfifo channels is at least

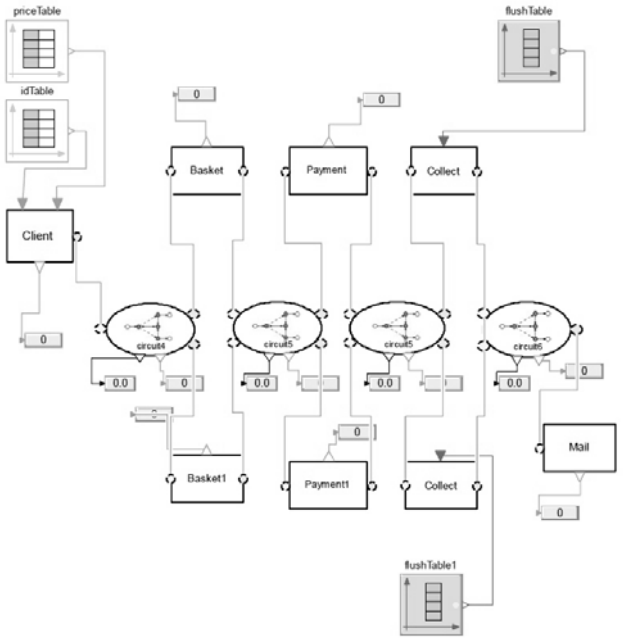


Figure 17: Case 2 as implemented in Dymola [4].

10%. The chart shows three curves, one for each variable `fifoCongestion`, `case1or2` and `open`. The first represents the only input of "Rewriting Rule", while the other two represent its two outputs (see Figure 18). The point indicated by a star in Figure 19 represents time 135s, when the 10% threshold is reached and circuit 7 stops routing messages to "Case 1", i.e., `open` becomes equal to 0 (it is now closed). When all lossyNfifo channels in "Case 1" are empty, at time 392s (indicated by two stars in Figure 19), circuit 7 starts routing messages again (i.e., `open` = 0), this time to "Case 2" (i.e., from having value 1, now `case1or2` becomes 0).

Finally, note that at time 200s, the *collect* services flush out their stored baskets (see Section 3) to circuit 3 (see Figure 7): this is the reason why `fifoCongestion` reaches 16% in Figure 19.

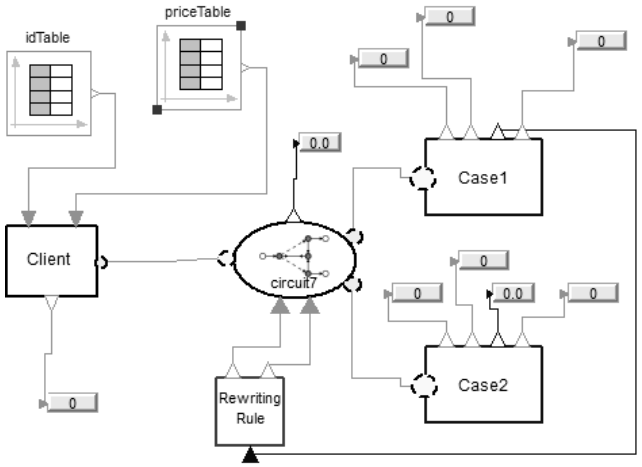


Figure 18: Case 3 as implemented in Dymola.

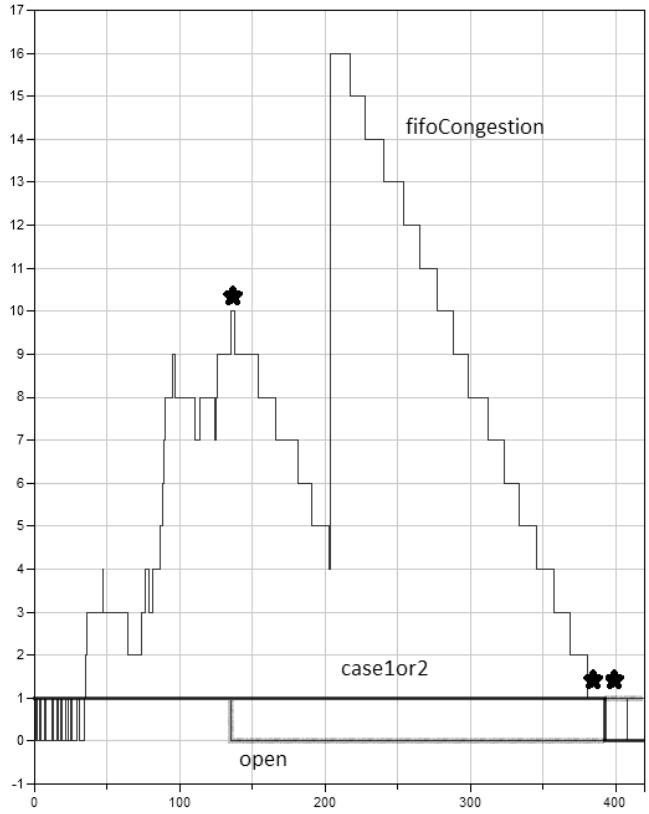


Figure 19: The chart obtained by simulating "Case 3". The star points to the instant at which our rewriting condition $\alpha = 10\%$ is reached, while the two stars indicate the moment of switching from "Case 1" to "Case 2", i.e., when rewriting has been completed.

For what concerns the simulation parameters, as time in seconds and number of baskets in input (more baskets form a single and complete account statement, as described in Section 3), in [4] the simulation process has been performed for $3,500$ seconds (almost one hour) by processing $10,000$ different baskets in input. The whole simulation procedure takes around 10–15 "real" minutes under these conditions. In this paper, however, it was sufficient to run the simulation for only 500 seconds in order to switch from one case to the other. Therefore, this approach scales quite well, also if we consider that we are interested in rewriting Reo circuits during critical events, which usually last for a relatively short time. Moreover, we can stress the circuits by ad-hoc crafting the initial state of the simulation from the beginning (in some known cases), thus reducing the time needed to experience the reconfiguration. This can be done, e.g., by launching the simulation with already partially-filled lossyNfifo channels.

6. CONCLUSIONS AND FUTURE WORK

In this contribution, we investigated the use of graph transformation for the reconfiguration of Reo circuits. Our proposal fits with the currently adopted approaches, and we additionally introduced the use of suitable predicates on channels in order to fine-tune the timing of rule applications. We then discussed an appropriate scenario and implemented

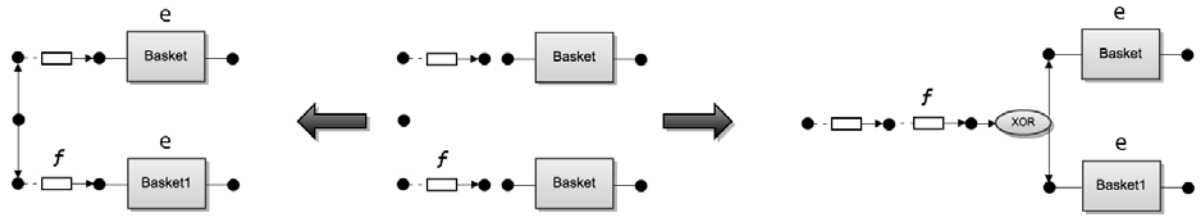


Figure 20: An "on-the-fly" version of rewriting rule 1 (in Figure 11).

a simulation of the circuit in the Dymola software, showcasing the improvement of the circuit's performance after reconfiguration.

The scenario exemplifies the usefulness of (a simple variant of) graph transformation, and its relevance in the context of Reo. The predicates are simply thought of as abstraction of the constraint automata associated to the graph items. They could be made more expressive, e.g. by associating values from a bounded partial order or a semiring, as in soft constraint satisfaction problems, thus allowing for a more complex notion of rule matching. Possibly, this could lead to symbolic graphs [12].

We close by indicating an alternative presentation for the reconfiguration rules. In Figure 20 an alternative version of rule 1 is given (cf. Figure 11). The *basket* service connectors are now included in the intermediate graphs and no predicate is required to hold. Intuitively, this corresponds to some kind of on-the-fly reconfiguration according to which the state of the basket is not required to be empty, yet it is preserved during rewriting. Even if this solution seems to increment flexibility, the graph morphism $l : L \to K$ is no longer injective, though, thus requiring a more difficult variant of the standard operational description of the rewriting step in order to preserve, e.g., the uniqueness of the result of a rule application after a match is chosen.

7. ACKNOWLEDGMENTS

The authors wish to thank Christoph Brandt from the Technische Universität Berlin (`cbrandt@cs.tu-berlin.de`) for his valuable suggestions and support during the simulation experiments in Dymola.

Francesco Santini carried out his work during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme, which is supported by the Marie Curie Co-funding of Regional, National and International Programmes (COFUND) of the European Commission. Maurice ter Beek conducted his work while on sabbatical leave at Leiden University. He gratefully acknowledges the hospitality and support during his stay in Leiden. Maurice ter Beek and Fabio Gadducci were supported by the Italian Ministry of Instruction, University and Research project CINA (PRIN 2010LHT4KM).

8. REFERENCES

[1] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

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

[3] C. Brandt and F. Hermann. Conformance Analysis of Organizational Models: A New Enterprise Modeling Framework using Algebraic Graph Transformation. *International Journal of Information System Modeling and Design*, 4(1), 2013.

[4] C. Brandt, F. Santini, N. Kokash, and F. Arbab. Modeling and Simulation of Selected Operational IT Risks in the Banking Sector. In M. Klumpp, editor, *European Simulation and Modelling Conference*, pages 192–200. EUROSIS-ETI, 2012.

[5] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1: Foundations, pages 163–245. World Scientific, 1997.

[6] S.-S. Jongmans and F. Arbab. Overview of Thirty Semantic Formalisms for Reo. *Scientific Annals of Computer Science*, 22(1):201–251, 2012.

[7] N. Kokash, C. Krause, and E. P. de Vink. Reo + mCRL2: A framework for model-checking dataflow in service compositions. *Formal Aspects of Computing*, 24(2):187–216, 2012.

[8] C. Krause. Distributed Port Automata. In F. Gadducci and L. Mariani, editors, *Graph Transformation and Visual Modeling Techniques*, volume 41 of *Electronic Communications of the EASST*. EASST, 2011.

[9] C. Krause, H. Giese, and E. P. de Vink. Compositional and behavior-preserving reconfiguration of component connectors in Reo. *Journal of Visual Languages & Computing*, 24(3), 2013.

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

[11] S. Meng and F. Arbab. Web Services Choreography and Orchestration in Reo and Constraint Automata. In Y. Cho, R. L. Wainwright, H. M. Haddad, S. Y. Shin, and Y. W. Koo, editors, *Symposium on Applied Computing*, pages 346–353. ACM, 2007.

[12] F. Orejas. Symbolic graphs for attributed graph constraints. *Journal of Symbolic Computation*, 46(3):294–315, 2011.

[13] M. Wermelinger and J. L. Fiadeiro. A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155, 2002.