Validating Reconfigurations of Reo Circuits

Maurice H. ter Beek[1], Fabio Gadducci[2], and Francesco Santini[3]

[1] ISTI–CNR, Pisa, Italy
[2] Department of Computer Science, University of Pisa, Italy
[3] EPI Contraintes, INRIA, France

Abstract. We formalize dynamic reconfiguration of Reo circuits (multi-party interactions built from primitive channels) by graph transformation and apply it to a critical infrastructure controlling the business process of an e-banking scenario, in which reconfiguration is triggered as soon as the communication buffers reach specific predefined thresholds of congestion.

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.

We describe our current research on defining graph transformation based reconfigurations for Reo circuits [3]. We apply them to a real-world scenario from the Finance domain: a critical infrastructure controlling the business process of an e-banking system, inspired by a study of real-world requirements at Credit Suisse S.A. in Luxembourg [4], in which reconfiguration is triggered as soon as the communication buffers reach specific predefined thresholds of congestion.

In this scenario, people select shares they want to buy, put them into a basket and pay for them using electronic means like credit cards. Once in a while, an account statement is sent to clients by e-mail, summarizing their past transactions. As a case study, we show two different configurations of this e-banking system and a way to switch from one to the other (stemming from the assumption that the nature of the incoming requests may change over time). The first case represents a redundant yet safer configuration: if a request fails in either one of the two branches of the original configuration, the other branch provides a potential backup. In the second case, 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.

We control the switching by setting variables to certain values through a graph rewriting model. The configurations are defined in Reo, the modeling and simulation processes in Modelica (`http://modelica.org/`), a non-proprietary, object-oriented, equation-based language for modeling complex physical systems.

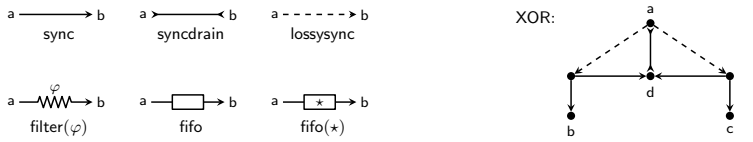

Fig. 1: Graphical syntax of common channels (left) and a circuit (right).

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.

Fig. 2: Channel behavior.

This paper is an extended abstract of [3]. In §2 we introduce Reo, while our e-banking scenario is introduced in §3. In §4 we define graph transformations for the dynamic reconfiguration described in our scenario and in §5 we evaluate their application. §6 contains our conclusions and ideas for future work. An appendix contains the basic notions on graph rewriting and the specific variant we use.

2 Reo

Reo facilitates the compositional construction of *circuits*: communication mediums coordinating interacting parties, each built from simple *channels*. A channel has exactly two *ends* that each have exactly one type: it is either a *source end* that accepts data items or a *sink end* that offers data items. Fig. 1(left) shows six primitive channels available for Reo users; Fig. 2 describes their behavior.

In a circuit, "data items" flow through "channels" (along edges) past "nodes". Nodes are logical points where execution over different channels is synchronized. A node is either *source*, *sink* or *mixed*, depending on whether all channel ends coinciding on that node are source ends, sink ends or a combination of both. 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. The XOR circuit in Fig. 1(right) implements an *exclusive router* of messages: its intuitive behavior is that data obtained as input through node a is non-deterministically delivered to one of the output nodes b or c.

In this paper we consider Reo's semantics based on *constraint automata* [2]: states represent the internal configurations of a circuit and transitions describe its atomic coordination steps. Formally, a transition is defined as a tuple of four elements: a source state, a *synchronization constraint*, a *data constraint* and a target state. A synchronization constraint specifies which nodes synchronize—i.e. through which nodes a data item flows—in some coordination

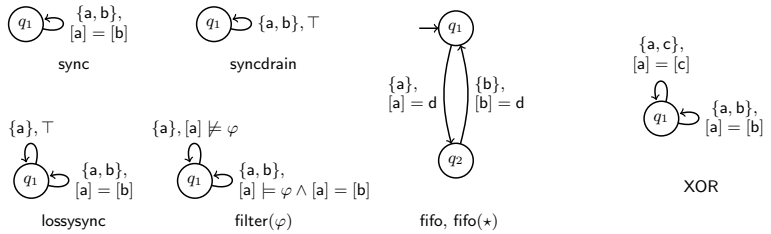

Fig. 3: Constraint automata for the common channels and the circuit in Fig. 1.

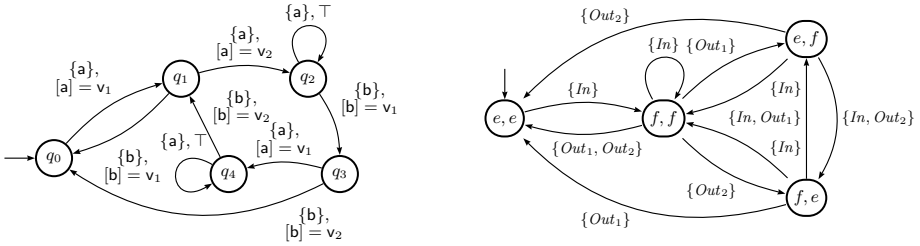

Fig. 4: Behavior of a lossyNfifo channel with two slots (left) and behavior of circuit 1 in Fig. 5 considering the two lossyNfifo channels to have one slot (right). The state labels denote the filling status of the two fifo channels: *f*ull or *e*mpty. The edge labels In, Out_1 and Out_2 stem from those shown in Fig. 5.

step; a data constraint specifies which particular data items flow in such a step. Figure 3 shows the constraint automata for the channels and circuits presented in Fig. 1.

Our case study extensively uses a new primitive Reo channel, called lossyNfifo, graphically represented in Fig. 5 as a fifo channel with a dashed incoming arrow. It models an n-position buffer, whose informal semantics allows it to start losing data when the buffer is completely full; otherwise, all its slots are filled in order.

In Fig. 4(left) we formalize its behavior (for $n = 2$) as a constraint automaton, in which variables v_1 and v_2 represent the two slots. In states q_2 and q_4 both are full, but it is still possible to receive elements through port a, and then discarding them (modeled 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.

3 A Case Study

We model a service landscape which implements and enables the business process of the aforementioned e-banking system. The model is described using Reo, resulting in the circuit in Fig. 5 for "Case 1" below. Routing between services is realized by circuits that internally implement different Reo connectors and that contain buffers (lossyNfifo channels) that can run full. In "Case 1", incoming requests are dropped and lost. We assume a stream of client orders flowing into the system, each has a certain business value in euro cents, comes

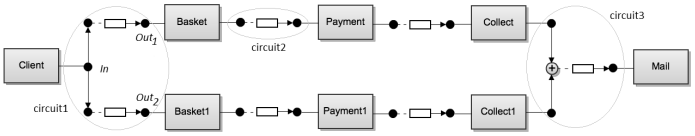

Fig. 5: The complete Reo circuit diagram for Case 1.

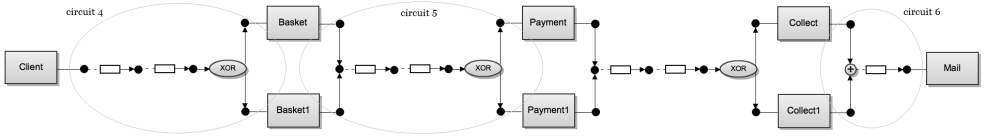

Fig. 6: The complete Reo circuit diagram for Case 2.

at a specific point in time and adds up to a concrete basket until that basket is complete.

Case 1. The input stream, produced by a *client* service, is replicated by circuit 1 for processing by an upper and a lower branch of pairwise identical services. When a request fails in either branch, the other provides a backup. In each branch, coordination between two consecutive services is modeled by circuit 2. Their output streams are merged, collected and forwarded from time to time to a *mail* service, which sends out account statements by e-mail, by circuit 3. The *basket* services handle the bookkeeping of all incoming requests until their corresponding baskets are complete, in which case the whole basket is sent to the *payment* service. Both services can fail, which is a behavior realized by their respective failure distributions. The *collect* services collect all successfully paid baskets and flush them out to the *mail* service, based on a given timetable. The *payment* services and the *mail* services use different delay distributions.

Fig. 4(right) shows the behavior of circuit 1 as a constraint automaton in which data constraints are omitted, i.e. only the synchronization constraints on nodes *In*, Out_1 and Out_2 are shown. These represent the endpoints where the *client*, *basket* and *basket1* components perform their offer and accept operations.

Case 2. We now assume that incoming orders are no longer replicated and simultaneously processed by two branches, but they are arbitrarily distributed to the available services to maximize the overall capacity of requests that can be processed. The main difference w.r.t. "Case 1" is the use of different Reo circuits to coordinate the services being used. In fact, services behave in exactly the same way. In Fig. 6, the complete Reo circuit diagram for "Case 2" is shown.

We now implement a sort of switch that either selects model "Case 1" or "Case 2" for processing 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. From a theoretical point of view, a switch enables the dynamic reconfiguration of the Reo circuits.

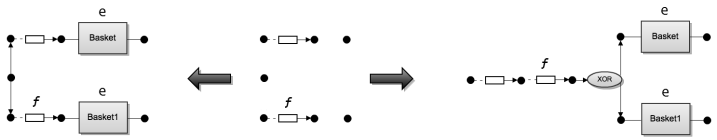

Fig. 7: Rewriting rule 1.

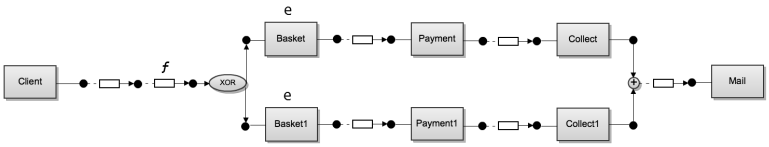

Fig. 8: Rewriting of Fig. 5 obtained by applying rule 1.

4 Rewriting for Reconfiguring

Graph transformation was adopted early on for reconfiguring Reo specifications. We use a simple variant (cf. Appendix) of an elaborate proposal [5, 6]. A rewriting rule is specified by two graph morphisms, denoting how the left-hand side is rewritten into the right-hand side, while using the intermediate graphs to track the connection between the items on both sides. We only consider graphical representations of circuits and enrich their edges with a predicate, preserved by the graph morphisms, representing a *QoS* measure: the signal to start reconfiguring.

In rule 1, depicted in Fig. 7, the three graphs (circuits) composing it are arranged horizontally and the morphims are suggested by the position of each item. The only ambiguity concerns the isolated nodes in the intermediate graph, which are mapped into the rightmost and leftmost node of the right-hand side. A predicate f, signaling the possible congestion of the channel, is easily 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 (filled). When the threshold is exceeded, the rule becomes enabled and it may be applied. It tends to replace the (first) component of the duplicated branch to turn it into a single branch with twice the buffering capacity. The need of reconfiguring also the connections between components is achieved by temporarily switching off and on the basket components, as long as they are empty: this is modeled by deleting and recreating the component after the step, while a predicate e on the basket component signals that the rule can be applied only after the baskets were indeed emptied.

The result of applying rule 1 to the circuit of Fig. 5 is depicted in Fig. 8. The reconfiguration has changed the behavior of the overall circuit, even if the semantics of each channel is preserved. The initial part of the circuit has now 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 ignored: they are the same before and after the reconfiguration steps and thus irrelevant for the enabling of the rule. Subsequent applications of rule 2 (Fig. 9) and rule 3 (Fig. 10) will turn

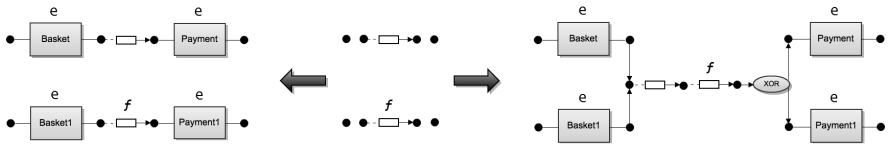

Fig. 9: Rewriting rule 2.

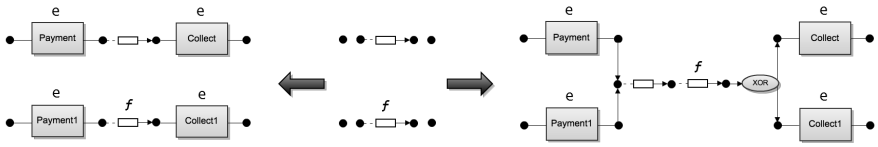

Fig. 10: Rewriting rule 3.

the circuit into that of Fig. 6.

5 Simulation and Validation

Dymola (`http://www.3ds.com/products/catia/portfolio/dymola`) is a tool supporting graphical compositions of Modelica models and their fast simulation with symbolic pre-processing. Figure 11(left) shows such a model of "Case 3", which implements the rewriting that switches from "Case 1" to "Case 2" whenever needed, i.e. when the maximum filling percentage among all lossyNfifo channels exceeds a (designer's choice) warning threshold. Circuit 7 models the Reo coordination between Cases 1 and 2. Our simulation below starts delivering data to "Case 1", after which it switches to "Case 2", as formalized in §4. The "Rewriting Rule" block models rewriting "Case 1" to "Case 2": when at least one of the lossyNfifo channels in "Case 1" is filled for more than a threshold percentage (the triggering condition), then circuit 7 waits for the complete emptying of all lossyNfifo channels before it resumes routing the incoming requests to "Case 2".

The two outputs from the "Rewriting Rule" block are needed to *i*) open/close the dispensing of requests from circuit 7 and *ii*) decide which of the two cases circuit 7 is routing requests to. When switching cases this block thus uses these two discretely valued parameters to first close ($open = 0$) the output flow of messages to "Case 1", and then, once all pending messages are served, switch the second flag (*case1or2*) to start routing messages to "Case 2" (reopening the output flow to it, 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.

Fig. 11(right) shows a chart obtained by simulating "Case 3" with Dymola until time 420s. To limit the time we set the threshold to 10%, meaning the triggering condition is fired when the maximal filling status among all lossyNfifo channels is at least 10%. The chart shows a curve for each variable `fifoCongestion`, `case1or2` and `open`. The first represents the only input of "Rewriting Rule", while the other two represent its two outputs (cf. Fig. 11(left)). The point indicated by a star represents time 135s, when

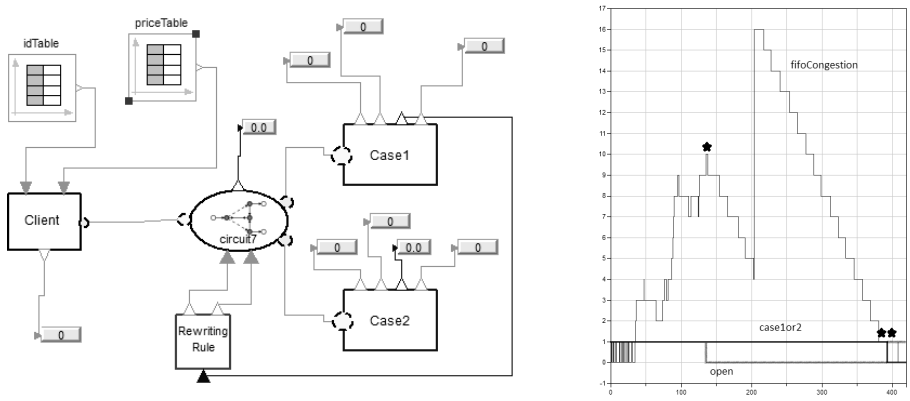

Fig. 11: Dymola: modeling "Case 3"(left) and simulating "Case 3" (right).

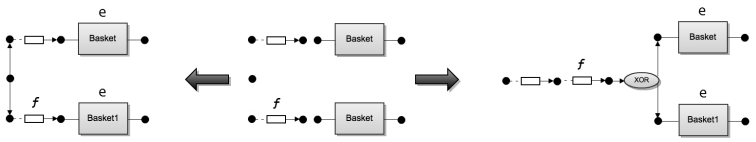

Fig. 12: An "on-the-fly" version of rewriting rule 1.

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), 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). Note that at time 200s, the *collect* services flush out their stored baskets to circuit 3, which is why `fifoCongestion` reaches 16%.

6 Discussion and Future Work

The scenario in this paper exemplifies the usefulness of (a simple variant of) graph transformation and its relevance in the context of Reo. The predicates used could be made more expressive, e.g. by associating values from a bounded partial order or a semiring, as in soft constraint satisfaction problems, to allow a more complex notion of rule matching. This could lead to symbolic graphs [7].

We could think of alternative reconfiguration rules, like for rule 1 in Fig. 12: The *basket* service connectors are included in the intermediate graphs and no predicate is required to hold. Intuitively, this corresponds to a kind of on-the-fly reconfiguration by which the state of the basket need not be empty, yet it is preserved during rewriting. This solution seems to increment flexibility, but the graph morphism on the left would no longer be injective, requiring a more difficult variant of the standard operational description of the rewriting step to preserve the uniqueness of the result of a rule application after a match is chosen.

References

1. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3) (2004) 329–366
2. Baier, C., Sirjani, M., Arbab, F., Rutten, J.: Modeling component connectors in Reo by constraint automata. Sci. Comput. Program. **61**(2) (2006) 75–113
3. ter Beek, M.H., Gadducci, F., Santini, F.: Validating Reconfigurations of Reo Circuits in an e-Banking Scenario. In Malek, S., ed.: ISARCS'13, ACM (2013) 39–47
4. Brandt, C., Santini, F., Kokash, N., Arbab, F.: Modeling and Simulation of Selected Operational IT Risks in the Banking Sector. In Klumpp, M., ed.: European Simulation and Modelling Conference, EUROSIS-ETI (2012) 192–200
5. Krause, C.: Distributed Port Automata. In Gadducci, F., Mariani, L., eds.: GT-VMT'11. Volume 41 of Electronic Communications of the EASST., EASST (2011)
6. Krause, C., Giese, H., de Vink, E.P.: Compositional and behavior-preserving reconfiguration of component connectors in Reo. J. Vis. Lang. Comput. **24**(3) (2013) 153–168
7. Orejas, F.: Symbolic graphs for attributed graph constraints. J. Symb. Comput. **46**(3) (2011) 294–315

Appendix: 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 to manipulate and by denoting how a rule can be applied to an object.

The best known proposal is probably the so-called DPO approach. It is summed up by the two squares in Fig. 13. 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 those of the right-hand side.

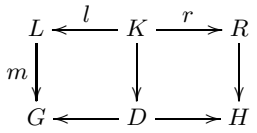

Fig. 13: A rule application in the DPO approach.

The idea is that both squares must be 'minimal'. The DPO approach achieves this by requiring the squares to be what in categorical terms is called 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. Second, all items of G matched by $L \setminus l(K)$ are removed, and those items identified by $l \circ m$ are coalesced, leading to context graph D. Third, 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 considered a simple variant of graphs adopted early on for modeling reconfiguration in Reo, in which each edge is actually equipped with a predicate and the morphisms between graphs actually preserve such predicates. All the relevant properties of the DPO approach (e.g., the uniqueness of the result of a rule application for a given match morphism) continue to hold also for our variant.