

Introduction

Webservices have gained enormous popularity in recent years. However, as it was the case with the nascent software engineering discipline, not much research has been dedicated to specify webservice formally. In this paper we focus on “locally implementable” webservice. A locally implementable webservice is one whose specification can be translated into sets of local views which, when put together, satisfy the global behavior of the specification. We want to find out which criterions preclude that a specification, called choreography here, is locally implementable, and will show how these criterions can be detected in REO (F. Arbab 2004) and Constraint Automata. The reason we use REO is twofold. First, due to extensive prior work on REO, sophisticated tools for REO circuits and the corresponding constraint automata are already available. Second, the visual accessibility of REO circuits allows us to detect non locally implementable circuits with the human eye.

The structure of this paper is as follows. In the first section, the terms choreography and projections are motivated, and an easy to use syntax for choreographies is presented. Surveying two papers on locally implementable issues, we extract criterions which render a given choreography non locally implementable and discuss ways to remedy these problems. In the next section, a mapping from this syntax into REO and constraint automata is put forward. A connection between the criterions for local non implementability and the corresponding REO structures (or constraint automata) is made in the third section. The paper ends in the fourth section, in which future work is outlined and a conclusion is drawn.

Criteria for local non implementable

There are two ways to specify webservice. Choreographies take a global view and specify the communication and observable behavior between a set of participants. Conversely, an orchestration only describes the local behavior of one participant, called role here. Choreographies can be transformed into several orchestrations by means of projection. In order to obtain an orchestration the choreography is projected onto a specific role. This procedure is repeated for each role, so that one projection is obtained for each role. However we will show later that such a projection is not always possible. We define a locally implementable orchestration as one which can be generated by a projection and which shows the same behavior as specified in the choreography.

CHOR, a language for choreographies

We now propose a simple language for choreographies, called CHOR, which is based on the languages presented in (Li, Huidiao und Geguang 2007) and (Qiu, et al. 2007) and formalizes choreography features.

$$A, B ::= \text{skip} \mid x@r := e \mid x@r1 \xrightarrow{c} y@r2 \mid A; B \mid A \parallel B \mid A \cap B \mid A < b > B \mid b * A$$

The language features basic and composed activities, denoted A and B , which represent the course of information exchange between different parties ($r1, r2, \dots, rn$) from a global view. The first three activities, *skip*, *assign* and *communication*, fall into the first category, whereas the rest, *sequential* and *parallel execution*, *nondeterministic* and *conditional choice*, and *loop* fall into the latter category.

Skip is the null activity, which does nothing at all. *Assign*, is a local activity which assigns the result of the evaluation of expression e to x , whereas *communication*, an interaction between $r1$ and $r2$, assigns the value of variable y in $r2$ to variable x in $r1$. *Sequential* and *parallel* execution refers to activities that are executed sequentially, in the first, and parallel in the latter case. *Choice* models a branch in the execution path, where, according to either the conditional expression b in the *conditional* case, or nondeterministically in the non *deterministic* case, either A or B is chosen as the future execution path. The *loop* repeats the sequence of activities A as long as the condition b becomes false.

A formal semantics for CHOR is not given, since the syntax is easy to understand and a straightforward behavior can be expected. (Li, Huidiao und Geguang 2007) and (Qiu, et al. 2007) propose similar grammars and suggest semantics based on transition rules and trace sets.

Criteria for non local implementable

As stated above, choreographies can be transformed into several orchestrations by means of projection. We defined a *locally implementable* choreography as one, whose projections, when joined, exhibit the same behavior as the choreography they were derived from. Only such choreographies are of interest for us, since the local implementable property allows us to work on a variety of interesting topics. The possibility of replacing parts of a complex distributed webservice, in case of failure for example, while retaining its proven semantics, is only one of many possibilities that arise.

We are hence interested in identifying locally implementable choreographies, and look for criteria which characterize non locally implementable choreographies. Surveying (Li, Huidiao und Geguang 2007) and (Qiu, et al. 2007), we extract five criteria for local non implementability. (Li, Huidiao und Geguang 2007) mention **continuity**, **single choice maker** and **loop termination** whereas (Qiu, et al. 2007) refer to **sequential composition** and **choice**.

(Proescholdt 2008) shows that **continuity** and **single choice maker** in (Li, Huidiao und Geguang 2007) are equivalent to **sequential composition** and **choice** in (Qiu, et al. 2007). The two papers are different in their treatment of **loops**, which are not covered in the latter, and **choice**, where the first distinguishes deterministic and nondeterministic versions, whereas the latter reunites them in a single choice construct. Thus, the three criterions **continuity**, **single choice maker** and **loop termination** remain to be considered.

Different types of local projections are possible. In the following text, we assume the projection to be the most straightforward implementation. The projection of a choreography onto a role r , removes all activities except the ones that interact with the role r , such as communications leading to or originating from r , or activities local in r . If choreography 1.1 for example is projected on r_1 , the resulting local representation only consists of a sending operation. All the other terms disappear.

We now formalize the constraints which ensure local implementability, to pave the way for detecting these criterions in REO. To this end the following notions are defined.

1. initiators: the participants which initiate a global activity A , denoted as $head(A)$.
2. terminators: the participants which conclude a global activity A , denoted as $tail(A)$.
3. participants: the participants which ever participate in an activity A , denoted as $part(A)$.

The three remaining criterions are listed below, each accompanied by an example illustrating the problem.

Continuity: a sequential activity $A;B$ is continuous if $head(B)$ is contained in $tail(A)$. Continuity requires for the initiator of the following activity terminate the preceding activity.

$$x1@r1 \xrightarrow{c1} y1@r2 ; x2@r3 \xrightarrow{c2} y2@r4 \quad 1.1.$$

The above choreography 1.1 is not locally implementable, since it cannot be guaranteed that the two communications take place in the order specified once they have been projected onto their local representations. This is because in the local representations of r_3 and r_4 all references to r_1 and r_2 are removed, and hence there is no way to enforce the specified order of activities.

Single choice maker: a nondeterministic choice $A \wedge B$ is single choice made when $head(A) = head(B)$ and the cardinality of $head(A)$ is 1.

A deterministic choice $A < b > B$ (where b is the condition) is single choice made when $loc(b) = head(A) = head(B)$, where $loc(x)$ denotes the participants in the boolean expression x . Again, the $card(loc(b))$ must be 1.

Single choice maker demands that each branch of a choice must have the same initiator. Moreover, this initiator must be the only one to decide which branch to follow.

$$x1@r1 \xrightarrow{c1} y1@r2 \cap x2@r3 \xrightarrow{c2} y2@r4 \quad 1.2.$$

The above example 1.2 demands mutual exclusivity. However, by the same pattern as before, this mutual exclusivity cannot be guaranteed after a local projection.

Loop termination: a loop activity $b * A$ is loop terminated if $\text{loc}(b) = \text{part}(A)$. This means that each participant in A needs to have a corresponding loop condition in b .

$$z1@r1 > 3 * (x1@r1 \xrightarrow{c1} y1@r2 ; z1@r1 := z1@r1 + 1) \quad 1.3.$$

In this example 1.3, the loop condition is only evaluated by $r1$. The loop body includes an interaction between $r1$ and $r2$, thus the local implementation for $r2$ will loop forever due to lack of its relevant loop condition.

Remedying locally implementable problems

The cause for the locally implementable problems is the lack of synchronization between local projections that need to follow a specific order of activities. In some cases this problem can be solved by introducing additional communications into the choreography. We will now show which problems can be remedied and how this can be accomplished.

The **continuity** problem emerges in a sequential activity when the concluding roles of the first activity are not contained in the starting roles of the second activity. This can be remedied by appending additional communications to the end of the first activity so that the continuity condition is satisfied. Formally, a communication from each of the members of the set $\text{head}(B) - \text{tail}(A)$ to one of the members of $\text{tail}(A)$ is inserted into a sequential activity $A;B$. If more than one communication has to be added, they have to be placed into a parallel execution construct, because otherwise they will not figure in the $\text{tail}(A)$ set.

There is no way to remedy the **loop termination** and **single choice** maker, problems, since they constitute a generic problem.

Note that in regard to loop termination, the criterion suggested by (Li, Huidiao und Geguang 2007) might not be strict enough, since as mentioned in a footnote, there is a problem in synchronizing the projections of two or more conditions in the loop condition, once they have been projected to their corresponding roles. This problem is left for the developer to solve. However, we think that in order to synchronize the variables on a higher level, they would have to be equivalent. Therefore, enforcing just a single role in the loop condition (as in single choice maker) makes sense.

Mapping from CHOR to REO

In order to use existing REO tools to prove constraints like the absence of all of the three criteria which indicate non local implementability, a mapping from the choreography language CHOR into REO must be established. The grammar for CHOR consists of 8 activities. We will present a mapping for each of them, knowing that REO and Constraint Automata dispose of the join operation, allowing for reassembling the single parts of the grammar in REO. In finding mappings (Tasharo und Sirjani 2008) and (Samira, et al. 2008) prove to be handy. We adapt the constructs used there, yet removing features like exceptions, which are not part of CHOR.

REO consists of components that are connected via connectors which coordinate their activities. An REO activity starts, when data is received at its **start** port and finishes with sending data to its **end** port. Primitive connectors are channels which have two ends. There are two types of channel ends: source and sink. A source channel end accepts data into its channel, and a sink channel end dispenses data from its channel. Basic types of channels, used in this paper are: Synchronous Channel (Sync), Synchronous Drain (SyncDrain) and FIFO1. A Sync channel has a source and a sink. Writing a message succeeds at the source if and only if the receiving of a message succeeds at the same time at its sink. A SyncDrain has two sources. Writing a message succeeds at one of its sources if, and only if, writing a message succeeds on the other source. The FIFO1 channel has a source and a sink. It maintains a buffer with capacity of one. Writing a message succeeds on the source of a FIFO1 if and only if its buffer does not contain any messages. The taking of a message, emptying the buffer, succeeds on its sink if and only if its buffer already contains a message. Complex connectors are constructed by composition of simpler ones by applying a join operation.

A component can write data items to a source node that is connected to it. The write operation succeeds only if all source channel ends meet at the node accept the data item. The data item is then written to every source end coincident at the node. A source node, thus, acts as a replicator. A component can obtain data items, from an input operation, from a sink node connected to it. A take operation succeeds if at least one of the sink channels' ends which coincide at the node offers a data item; if more than one coincident channel end offers data items, one is selected nondeterministically. A mixed node nondeterministically selects and takes a suitable data items offered by one of its coincident sink channel ends and replicates it to all of its coincident source channel ends. A sink or mixed node, thus, acts as a nondeterministic merger.

An Exclusive Router (XR) has a start and two end ports. When a data item arrives at the start port, it only flows to one end port, depending on which one is prepared to consume it. A REO connector or circuit can be put inside a box (with end and start ports) to make a component.

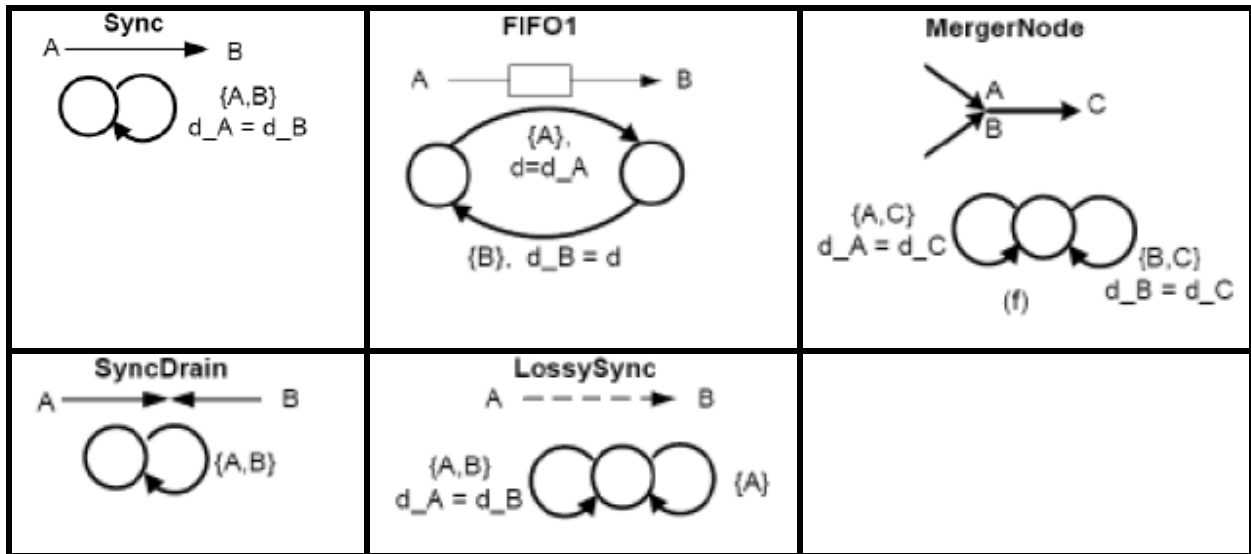


Figure 1 Constraint automaton for some basic REO channels

We use CHOR in this paper, due to its simplicity, not without mentioning that other languages such as WS-CDL, for which a mapping into REO exists (Tasharo und Sirjani 2008), can be used and are being used for modeling webservice choreographies. Due to the equality of the used constructs (basic and composed activities), our results also apply to them.

Mapping of the CHOR components into REO

We will now show how a CHOR choreography is mapped into REO. CHORs syntax consists of 8 activities. In addition, **roles** and **variables** have to be modeled. We chose to represent roles as REO components. A role might have a limited number of variables, which have to be accessed from outside. In order to address them, a component has two ports for every variable. One port is for accessing the variables values, the other one for assigning a value to it. Successive reads on the first port, result in the variables value unless it is overwritten by a write on the second port. **Constants** are represented as components which constantly emit the same value. We also use a number of other predefined components to model conditional expressions and basic arithmetic operations, like **addition** and **greater than**.

First, the mappings for the basic activities **skip**, **local assignment** and **remote assignment** are introduced. We also depict the corresponding constraint automata (CA). The CA circuits will ease automatic analysis of a REO circuit later.

The **skip** construct can easily be modeled by a FIFO1 circuit, which connects start and end.

The **local assignment** requires two variables a and a' of the role X , where the content of the latter is assigned to the former. In Figure 2 we show the circuit of a variable, whereas Figure 3 depicts how to assign the value of a' to a .

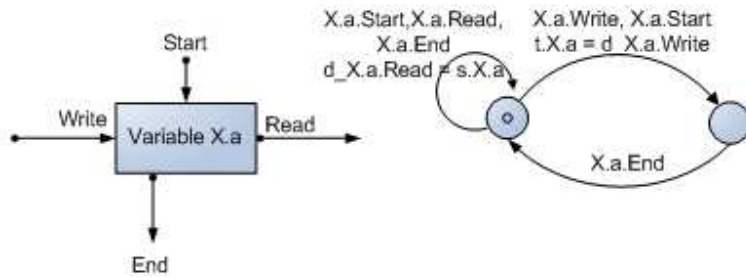


Figure 2 REO circuit and constraint automaton for a variable

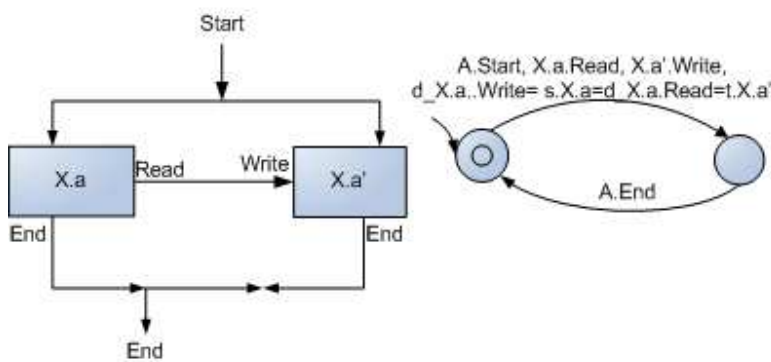


Figure 3 REO circuit and constraint automaton for assignment

The **remote assignment** (communication) assigns the value of variable a in X to the variable b in Y .

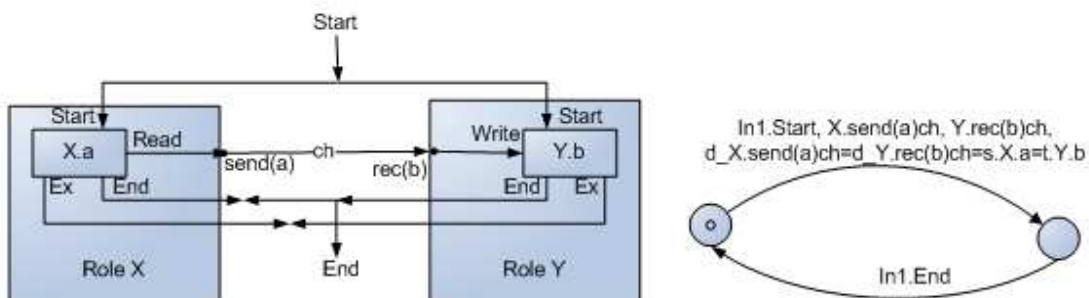


Figure 4 REO circuit and constraint automaton for remote assignment

Second, the REO circuits of composed activities sequential and parallel execution, choice and loop are shown.

A **sequential** activity is modeled in REO by connecting the start ports of an activity to the end ports of the following activity. A sequencer activates the components sequentially.

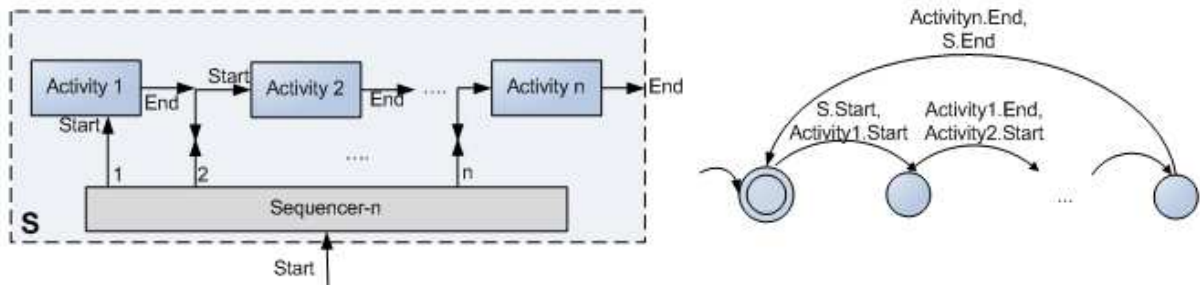


Figure 5, REO circuit and constraint automaton for sequential activities

The **parallel execution** of two or more activities is modeled by connecting the start port of the component to the start ports of all activities.

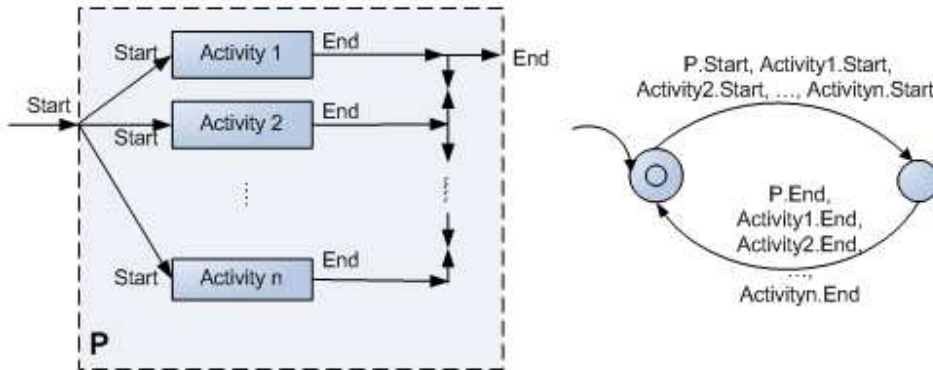


Figure 6 REO circuit and constraint automaton for parallel execution

There are two different types of **choice**: **non-deterministic**- and **conditional choice**. For non deterministic choice, we use an X-Router, which selects the future execution path non-deterministically. For the conditional choice, a component modeling the boolean condition is inserted before the X-Router. This additional component models the Boolean- and arithmetic expressions of the condition.

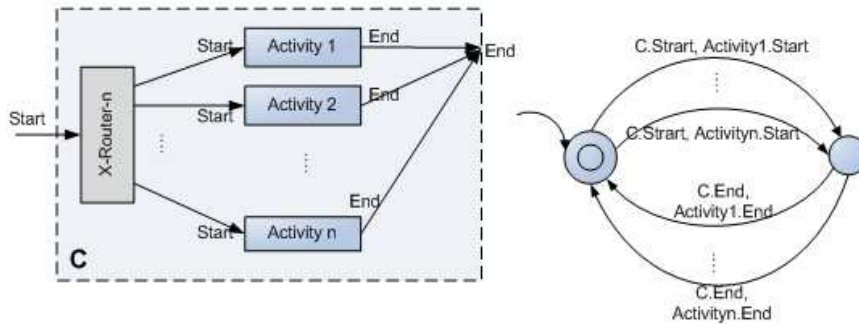


Figure 7 non-deterministic choice REO circuit and constraint automaton

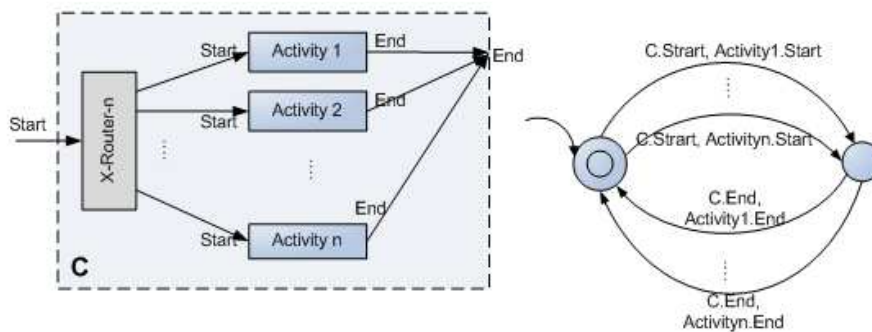


Figure 8, REO circuit for conditional choice with constraint automaton

For modeling the **loop** construct, a while loop can be used. It consists of a condition evaluator which is similar to the conditional choice above, with one branch leading to the loop activity and the other one to the end. The end channel of the loop activity leads the execution back to the beginning. As well as this, the end ports are connected with synchronized FIFO1 channels. A FIFO1 and synchronization channel prevent the component from restarting before the last activity has concluded.

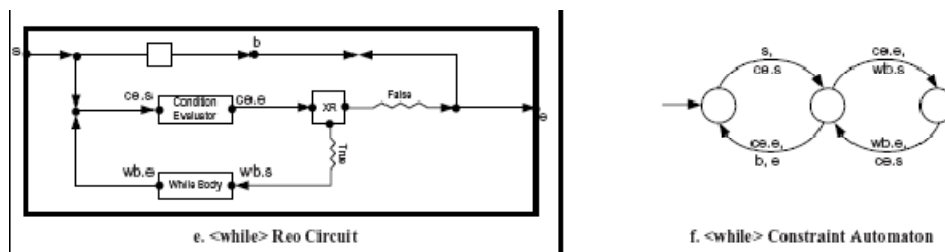


Figure 9 REO circuit for while loop with constraint automaton

Detecting non local implementability in REO and constraint automaton

After introducing the CHOR syntax and considering three criterions which impede local implementability, we will now show how non local choreographies can be detected in constraint automata. We build CA out of REO circuits which represent choreographies, exploiting REO's operational semantics in constraint automata (CA) (Arbab, et al. 2006) and examine how the criterions can be detected in CA. We will see that the main problem for local non implementability is the loss of control flow, caused by the local projection on a role. This means that control flow which is given implicitly by the CHOR choreography, is lost after the local projection. The following sequence of activities in distinct roles is an example for this.

$$e@r1 ; e@r2 ; e@r3$$

It is taken for granted in the global view that the second activity is only started after the first one has concluded. However, since the local projection removes activities irrelevant to the local role, in the local views the activities appear out of context. In the preceding example the projection on $r1, r2, r3$ produces three single local activities 'e'. Since there is no communication whatsoever between the roles, it can no longer be guaranteed that activity e in $r1$ will be the first operation, and e in $r3$ the last.

In the examples below, we will notice that CHOR and REO use different ways to model control flow. Whereas in CHOR the control flow, such as sequencing, is given implicitly through the ordering of the code, in REO a sequencer and REO channels are used to coordinate the sequential execution. Since the same channel construct is also used for communication (Figure 3), there exists no way to distinguish a channel which is used for coordination from one used for communication. This will turn out to be a problem, because the presence of the non local implementable criterions is linked to missing communication channels in REO. We will see that this problem affects all three cases, and will propose a naming scheme, to distinguish between communication and coordination.

We will now list the three criterions considered above, and show how their presence can be detected in REO and CA.

Continuity in REO

First, the problem of **continuity** will be illustrated. We consider the following choreography.

$$x1@r1 \xrightarrow{c1} y1@r2 ; x2@r3 \xrightarrow{c2} y2@r4 \quad 1.4.$$

A B

As shown in the previous section, this choreography is not locally implementable. Recall the sets *head* and *tail* and the condition for continuity; *head(B) is contained in tail(A)* with *tail(A)* being {r2} and *head(B)* being {r3}, which is obviously not contained in {r2}. It is clear that the condition fails. After projection, the local implementations of roles r3 and r4 have no idea of r1 and r2, and nothing can hence guarantee that, as required by the choreography, they are executed after r1 and r2 have run.

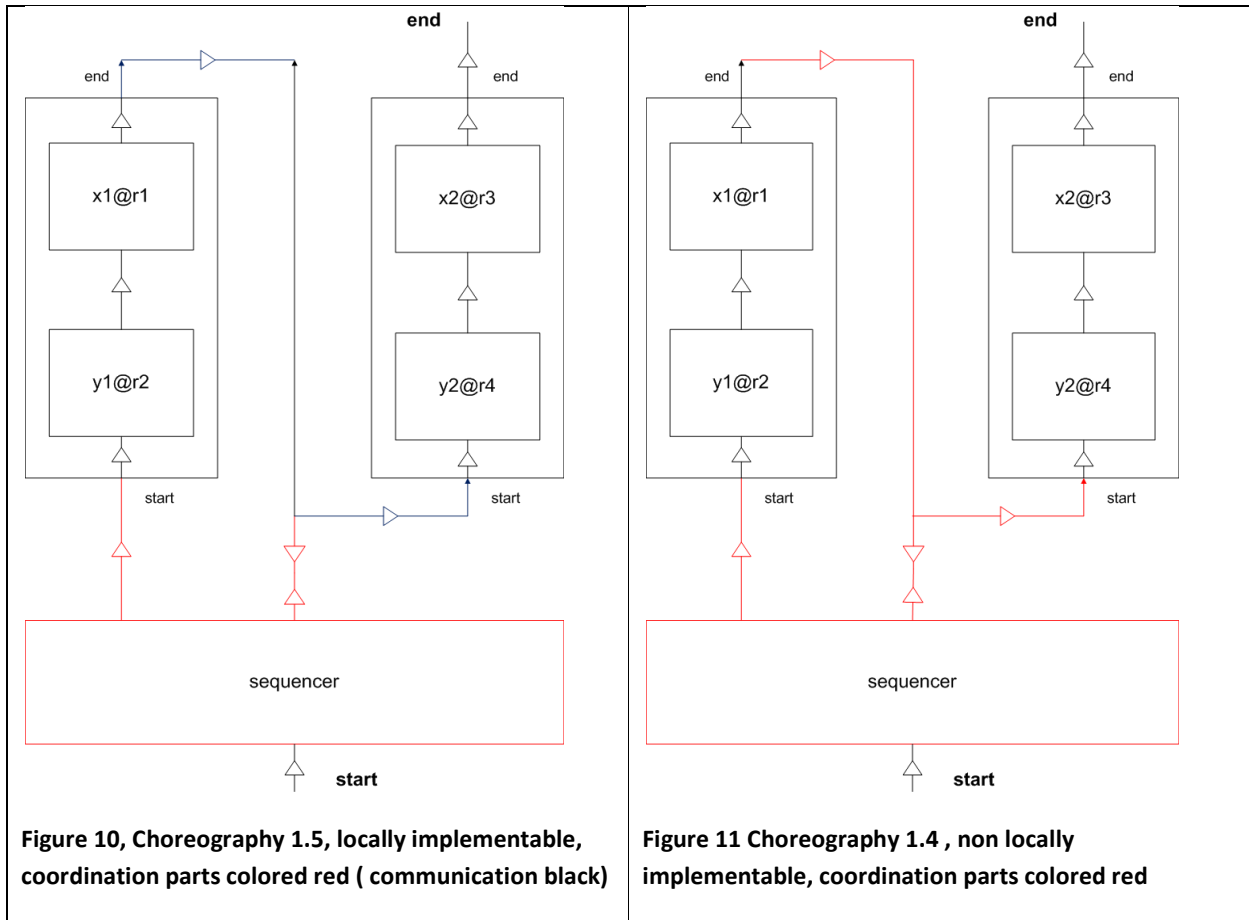
We now show how the violation of the condition can be seen in REO. The actual problem in choreography 1.4 is the sequence of r2 and r3, which are only separated by a semicolon in CHOR, to indicate the sequence. In REO this is represented by a channel between r2 and r3 and a sequencer, as explained in Figure 5. However, choreography 1.5, which is locally implementable, results in the same REO circuit as can be seen in Figure 10 and 11. This is because the communication between r2 and r3 is also represented by a channel between them; the before mentioned problem of unity of communication and coordination.

The trouble is that 1.4 is non locally implementable, whereas 1.5 is, (*Tail(A) = {r2}, head(B) = {r2}, tail(B) = {r3}, head(C) = {r3}*) making it impossible to distinguish a non locally implementable REO circuit from a locally implementable one. We conclude that, without further help, it is impossible to detect the continuity property by looking at a REO circuit.

$$x1@r1 \xrightarrow{c1} y1@r2 ; y11@r2 \xrightarrow{c11} y11@r3 ; x2@r3 \xrightarrow{c2} y2@r4 \quad 1.5.$$

Remedying unity of communication and coordination

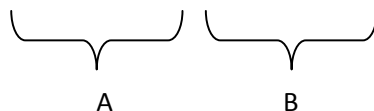
In order to remedy this problem we propose the following scheme. The unity of communication and coordination channels is removed by marking channels which are used for communication, such as the ones which are used in the mapping of a remote assignment in Figure 4. Below this will be done by painting communication channels red. If both a coordination and a communication take place between two roles, the communication takes precedence, which means that the channel is **not** marked. This does not pose a problem to REO, and the mapping rules which were put forward above can be adapted easily. Consequent algorithms will now be able to distinguish between communication and coordination channels. It is hence possible to detect missing communication despite the presence of a coordination channel.



Parallel execution and sequences

The previous example constitutes the best case of the continuity problem, since the head and tail sets consisted of only one role. In the general case, more than one role can be in the sets head or tail. This occurs when multiple roles are executed in parallel on one or both sides of a sequence. In this case, the continuity condition requires each role on the right side to have a connection channel to at least one role on the right side, in order to satisfy the “*head(B) is contained in tail(A)*” condition.

$$e1@r1 \parallel e2@r2 ; e3@r3 \parallel e4@r2$$



The above choreography is an example for this case. Both $\text{head}(B) = \{r3, r2\}$ and $\text{tail}(A) = \{r1, r2\}$ have multiple roles. However, $\text{head}(B)$ is not completely contained in $\text{tail}(A)$, which makes the continuity condition fail.

Formulated in REO this condition is as follows. If one or more roles on the right hand side of a sequence (B) do not have a communication channel to the left side (A), the continuity constraint is violated. As before, the coordination circuits of the sequencer component preclude a distinction between communication and coordination.

Continuity in REO, finally!

With these measures in place, we can now proceed to finally proposing an algorithm which detects a violation of the **continuity** constraint in a REO circuit.

In the first step we remove all channels which are related to coordination from the circuit. This is equivalent to removing the **non red** parts from Figure 10 or 11.

In the second step, we check if there still is a channel between distinct roles. The remarks about parallel execution made above are hereby taken into account. If there are one or more missing channels, the corresponding choreography is violated.

Looking at Figure 10 and 11 we can see that it is now possible to distinguish a non locally implementable from a locally implementable REO circuit.

Single choice maker in REO

The requirement of single choice maker is twofold. First, each selective branch must have one single identical role as an initiator. This means that, after a conditional branch the execution path must continue with the same role, regardless of which path is followed. Moreover, this initiator has to be the only one which decides which branch will be chosen. This means that the decision which execution path is followed must only depend on one role, which is also the initiator role. In the case of conditional choice this means that the Boolean condition must be evaluated by this single initiator only.

The two conditions can be seen easily in REO. The equality of the initiators is equivalent to a choice component whose two execution paths (for true and false) connect with the same role. The uniqueness can be guaranteed, if the choice component only considers variables of one role. This means that channels can only lead to one role. In the case of the conditional choice, the variables used in the conditional expression, can only be of one role. Therefore, if the conditional expression connects to other roles than the initiator role, the condition is violated.

The above description leads directly to an algorithm capable of detecting the single choice maker criterion.

Figures 12 and 13 illustrate examples of REO circuits for non locally implementable and locally implementable choreographies. Both make use of choreography 1.6, which shows an example of a choreography that violates the non single choice maker condition.

$$x@r1 ::= 3 < x@r1 > 100 > x@r2 \xrightarrow{c^2} y@r3 \quad 1.6.$$

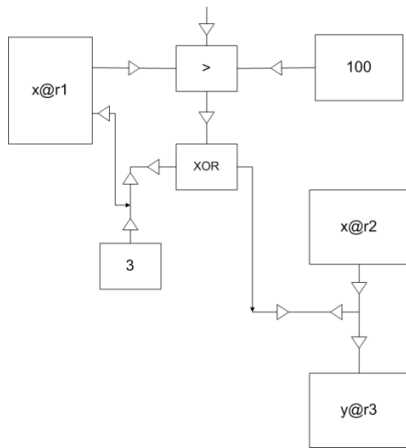


Figure 12, choreography 1.6, the single choice criterion is violated

We now consider choreography 1.7, which fulfills the requisitions of single choice maker.

$$x@r1 ::= 3 < x@r1 > 100 > x@r1 \xrightarrow{c^2} y@r3 \quad 1.7.$$

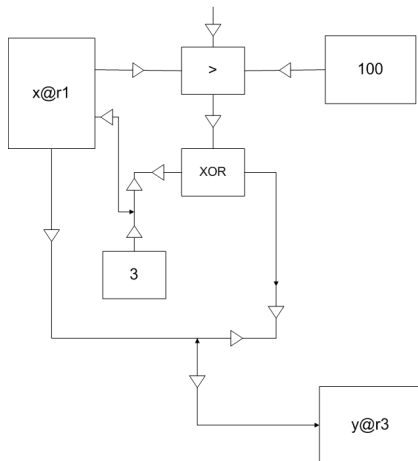


Figure 13, choreography 1.7, it is locally implementable

Loop termination in REO

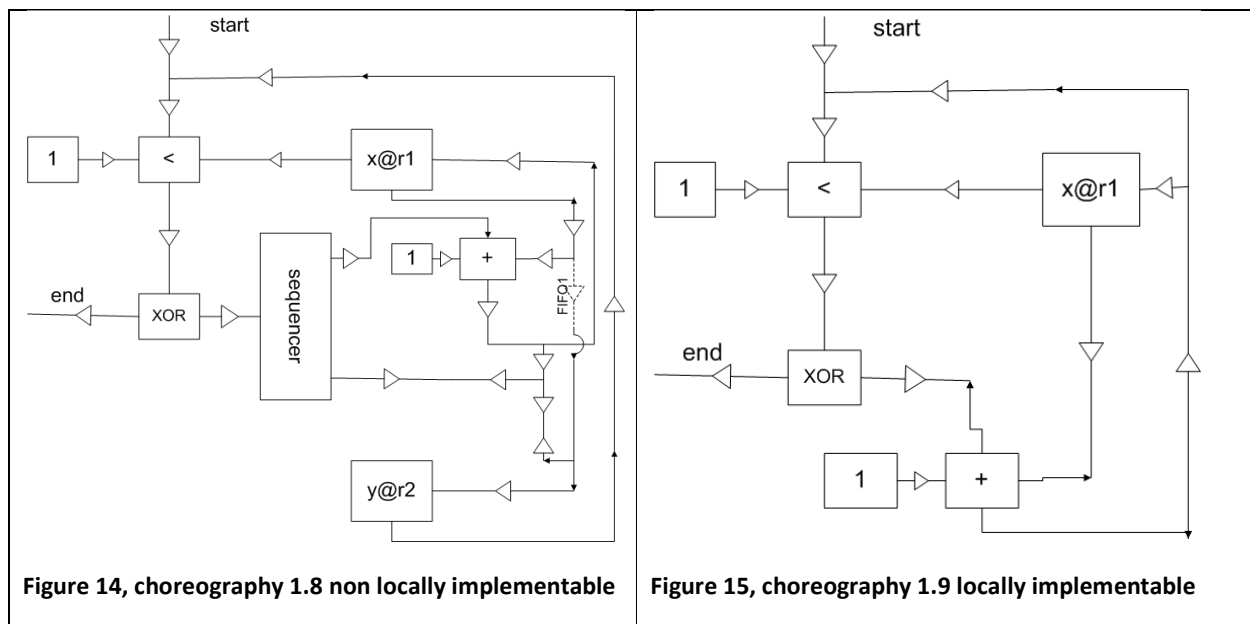
The loop termination criterion states that each role that appears in the loop activity must also have a representation in the loop condition. We restricted the condition even further, so that only one single role is allowed inside the loop condition. This means that the loop body can only contain activities inside a single role.

We will now describe how this is reflected in a REO circuit. First, channels originating in the loop's condition must lead only to variables or activities in a single role. Second, the loops activity can only consist of channels linking local variables or leading back to the loop condition. A communication or coordination channel leading from one role to another violates the condition. Last but not least, the roles in the condition and in the activity must be identical.

As an example we consider choreography 1.8 , which is non locally implementable and choreography 1.9 which is.

$$x@r1 > 1 * (x@r1 = x@r1 + 1; x@r1 \xrightarrow{c2} y@r2) \quad 1.8.$$

$$x@r1 > 1 * (x@r1 = x@r1 + 1) \quad 1.9.$$



Both choreographies fulfill the first condition, that the conditional part takes only one role into account: The channel leading from X@r1 to the less than sign. However, in choreography 1.3 the second

constraint is violated since there is a channel leading from the condition to a sequencer. The sequencer invokes activities in r2 and r1, which is a violation of the second condition.

In the preceding section, we have pointed out how non local implementable criteria can be found in REO circuits. However, the algorithms proposed were not immediately applicable, since we lack an adequate representation of REO circuits so far. This problem will be solved in the next section, where we will outline how to detect the three criteria in Constraint Automata (CA), which are proposed as semantics for REO (Arbab, et al. 2006).

Detecting non locally implementable criteria in constraint automata

We will now discuss ways to detect non locally implementable issues automatically. To this end we use Constraint Automata (CA), which we derive directly from REO circuits. We use choreography 1.1 as an example, and will show how the **continuity** criterion can be found. The other two criteria are not elaborated, but can be found using similar procedures.

The choreography 1.1 is non locally implementable due to its violation of the **continuity** requirement. However, as seen in Figure 10 and Figure 11, the REO circuits look similar, since coordination and communication are not distinguished. We will now show that this distinction can be made. We consider the CA schemata of choreographies 1.4 and 1.5, which we build by assembling the basic CA circuits which accompanied the basic REO circuits in the mapping section. While the two circuits look equal in REO, the additional communication c11 is represented in the CA schema by an additional state, as shown in Figure 16 and Figure 17.

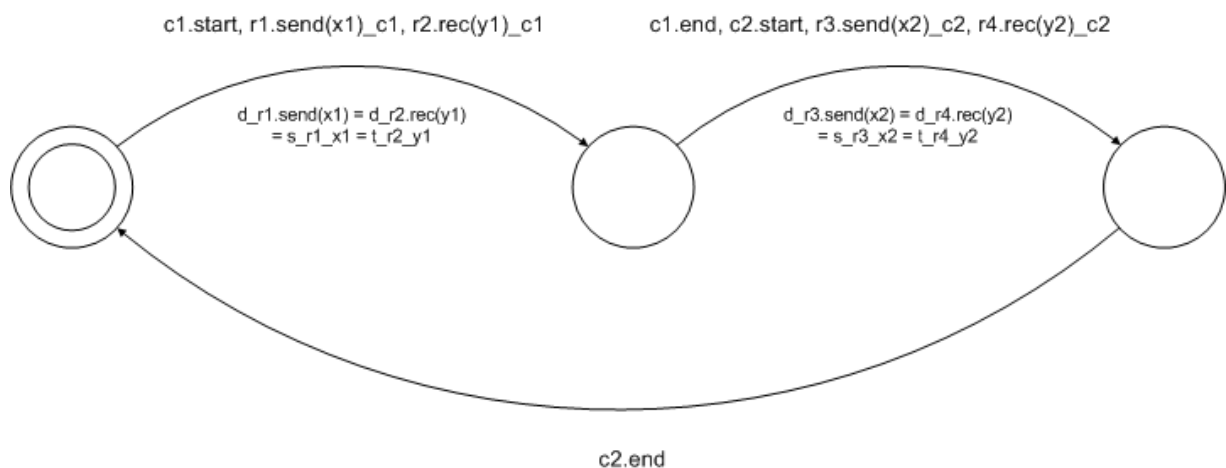


Figure 16, CA of choreography 1.4

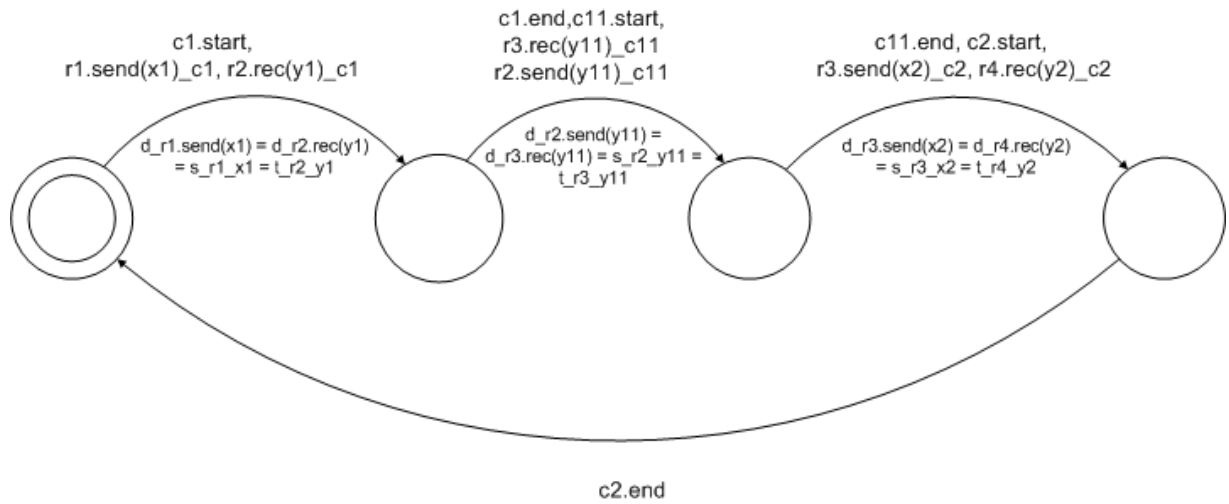


Figure 17, CA of choreography 1.5. Note the additional state representing the communication between r2 and r3

Detecting the difference becomes feasible now. In this case, the difference between Figure 17 and Figure 16 is that the latter can be traversed backwards from the end to the start state. The rules according to which the CA can be traversed are as follows. Begin at the end state. For each transition which ends here, check whether one of the following conditions is fulfilled. I) the transition is only guarded by an end activity. II) if the transition is guarded by a send activity of role r, then the state from which this transition originates from, must have an incoming transition which is guarded by a reception activity of the same role r. Note that this requires a one state look-ahead. If either I) or II) are fulfilled, mark the transition as ok. If all incoming transitions are ok, descend into each of them and repeat the procedure.

The proposed algorithm is in fact a depth-first tree search algorithm with loop protection, applied to a state automaton, and any current implementation can be used. Conditions I) and II) reflect the equality of the head and tail sets, as required by the **continuity** criterion.

We note that the above definition also includes the cases where the head or tail sets contain more than one role. The head and tail sets have more than element due to parallel execution and choice. However, these two cases are included in the above definition. As can be seen in Figures 6 and 7, the CA for choice adds one transition for each branch (normally two) and parallel execution adds additional guards to the current transition. These two cases are covered by the algorithm above.

Conclusion and further work

In this paper we have presented the choreography language, CHOR, which can be used to model basic choreographies. We considered local implementability issues, and presented three criteria which need to be fulfilled to make choreographies locally implementable. In order to detect these criteria in a given choreography, we put forward a mapping from CHOR to REO and constraint automata, to ease detection, either visually in REO or computationally in Constraint Automata.

We saw that the visual detection of the non local implementable criteria is difficult, especially when more complex choreographies are modeled. Automatic detection in constraint automata is possible, as shown in the example, but needs more work.

Further work consists of implementing a tool that maps a given CHOR choreography into REO, draws the corresponding circuit and outputs constraint automata. If this tool could be extended to cover WS-CDL as well, real world applications could be tested for locally implementable issues.

Bibliography

Arbab, Farhad. „Reo: A Channel-based Coordination Model.“ 2004.

Arbab, Farzad, Marjan Sirjani, C. Baier, and J.J.M.M. Rutten. „Modeling component connectors in Reo by constraint automata.“ *FOCLASA'03*. 2006.

Li, Jing, Zhu Huidiao, and Pu Geguang. „Conformance Validation between Choreography and Orchestration.“ *First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering(TASE'07)*, 2007.

Proescholdt, Timo. „Report about the Internship at the Formal Methods Lab / Webservice formalization.“ 2008.

Qiu, Zongyan, Xiangpeng Zhao, Chao Cai, and Hongli Yang. „Towards the Theoretical Foundation of Choreography.“ 2007.

Samira, Tasharofi, Vakilian Mohsen, Zilouchian Moghaddam Roshanak, and Sirjani Marjan. „Modeling Web Service Interactions Using the Coordination Language Reo.“ 2008.

Tasharo, Samira, and Marjan Sirjani. „Formal Modeling and Conformance Validation for WS-CDL using Reo and CASM.“ *Electronic Notes in Theoretical Computer Science*, 2008.