Bridging Global and Local Models of Service-oriented Systems

Johannes Maria Zaha
University of Essen, Germany
Johannes.Zaha@sse.uni-due.de
Marlon Dumas Arthur H.M. ter Hofstede
Queensland University of Technology, Brisbane, Australia
(m.dumas,a.terhofstede)@qut.edu.au
Alistair Barros
SAP Research Centre, Brisbane, Australia
alistair.barros@sap.com
Gero Decker
Hasso-Plattner-Institute, Potsdam, Germany
gero.decker@hpi.uni-potsdam.de

***Abstract*— A service-oriented system is a collection of independent services that interact with one another through message exchanges. Languages such as the Web Services Description Language (WSDL) and the Business Process Execution Language (BPEL) allow developers to capture the interactions in which an individual service can engage, both from a structural and from a behavioral perspective. However, in large service-oriented systems, stakeholders may require a global picture of the way services interact with each other, rather than multiple small pictures focusing on individual services. Such global models are especially useful when a set of services interact in such a way that none of them sees all messages being exchanged, yet interactions between some services may affect the way other services interact. Unfortunately, global models of service interactions may sometimes capture behavioral constraints that can not be enforced locally. In other words, some global models may not be translatable into a set of local models such that the sum of the local models equals the original global model. Starting from a previously proposed language for global modeling of service interactions, this paper defines an algorithm for determining if a global model is locally enforceable and an algorithm for generating local models from global ones. It also shows how local models are mapped into templates of BPEL process definitions.**

I. Introduction

As the first generation of web service technology based on XML, SOAP, and WSDL reaches a certain level of maturity, a second generation targeting long-running collaborative business processes is gestating. In the first generation, web services are equated to sets of operations and message types (cf. WSDL). This conception reflects an emphasis on single request-response interactions. Meanwhile, the second generation of web service technology targets conversational interactions, with service descriptions capturing not only individual message exchanges and the underlying message types, but also dependencies between these exchanges, most notably behavioral dependencies.

This trend is evidenced by the emergence of languages for modeling and implementing services that engage in conversational interactions. These languages include the Business Process Modeling Notation (BPMN) [18] and the Business Process Execution Language for Web Services (BPEL) [1], which respectively target the design and the implementation phases of service development. Other relevant proposals in this space are the Web Services Choreography Description Language (WS-CDL) [10] and variants of UML Activity Diagrams such as BPSS/ebBP [6] and UML Profile for EDOC [14].

An analysis of approaches to conversational service modelling unveils two different approaches. On the one hand, interactions between conversational services can be modeled from the perspective of individual services, that is, each service model defines the set of messages that the service in question can send and receive; individual service models are then "stitched together" to capture conversations between services. This approach is suitable when building individual services or when service-enabling existing applications. However, in the early phases of the service development lifecycle, emphasis is not on building individual services but rather on identifying potential services and understanding their interactions. In these phases, stakeholders need a global picture of the way services interact with one another. Thus, models at this level need to emphasize the interactions between services and their interdependencies. Such "global models" are especially useful when a multiple parties interact in such a way that none of them sees all messages being exchanged, yet interactions taking place between some parties have an impact on the way other parties interact. WS-CDL is an example of a language for describing global models of service interactions (also known as *choreographies*).

The second author was funded by a fellowship co-sponsored by Queensland Government and SAP Research. The first and last authors contributed to this research while working at Queensland University of Technology and SAP Research respectively.

This paper is an extended and revised version of reference [22].

Given their complementarity, an approach to conversational service modeling that seamlessly integrates global and local views, is desirable. In previous work [21], we introduced a language, namely "Let's Dance", for modeling service interactions and their behavioral dependencies. Let's Dance supports service interaction modeling both from a global and from a local viewpoint. In a global (or choreography) model, interactions are described from the viewpoint of an ideal observer who oversees all interactions between a set of services. On the other hand, local models focus on the perspective of a particular service, capturing only those interactions that directly involve it. A possible usage scenario is one where global models are produced by analysts to agree on interaction scenarios from a global perspective, while local models are produced during system design and handed on to implementers. Implementers then refine the local models and/or use them to generate code templates (e.g. in BPEL). To ensure proper handover between these users, it is necessary to have a mapping from global to local models.

It turns out that not all global models can be mapped into local ones in such a way that the resulting local models satisfy the following two conditions: (i) they contain only interactions described in the global model; and (ii) they are able to collectively enforce all the constraints expressed in the global model. For example, consider a global interaction model containing: (i) a first interaction where an actor A sends a message to an actor B; (ii) a second interaction where an actor C sends a message to an actor D; and (iii) a constraint to the effect that the second interaction can not occur before the first one. An obvious question arises then: How can actors C and/or D know that the interaction between A and B has taken place in the absence of any interaction between actors A and B on the one hand, and actors C and D on the other? Thus, either the model needs to be enhanced with an interaction between A/B and C/D, or the sequential execution constraint will not be enforced. In WS-CDL, constraints that can not be enforced using the explicitly declared interactions are enforced by implicit interactions. Since we envisage business analysts "signing off" on global models defined in Let's Dance, we consider the option of introducing such "hidden interactions" implausible. Hence, tools for global service modeling need to ensure that global models are "locally enforceable", meaning that they can be translated into sets of local models satisfying the two conditions above. This paper presents an algorithm for determining whether or not a global model expressed in Let's Dance is locally enforceable. The paper also provides an algorithm to translate global models into local ones.

The paper is structured as follows. Section II gives an overview of the Let's Dance language. Sections III and IV respectively present the algorithms for determining the local enforceability of global models and for generating local models from global ones. Section V introduces an algorithm for generating BPEL code from local models. A tool supporting the Let's Dance language is presented in section VI. Section VII discusses related work and Section VIII concludes.

II. Language Overview

Let's Dance abstracts away from implementation details and avoids reliance on imperative programming constructs. In particular, the language does not rely on variable assignment, while conditions need not be written in an executable language. Instead, these are treated as free-text labels that are subsequently refined. Still, models defined in Let's Dance can be used to generate BPEL templates or to check that a service implementation conforms to the behavioral constraints captured in a model. This section provides an overview of Let's Dance. A more detailed description of Let's Dance and solutions for the service interaction patterns presented in [3] can be found in [21].

A. Language Constructs

A choreography consists of a set of interrelated service interactions corresponding to message exchanges. At the lowest level of abstraction, an interaction is composed of a message sending action and a message receipt action (referred to as communication actions). Send and receive actions are represented by non-regular pentagons that are juxtaposed to form a rectangle denoting an elementary interaction as illustrated in Figure 1. In this figure, each rectangle with an arrowhead in the middle denotes an elementary interaction. Each half of this symbol denotes a communication action (send or receive). The name of the message type being exchanged is written in the middle of the interaction (e.g. "Message M1"). A communication action is performed by an actor playing a role. This role is specified in the top corner of the symbol corresponding to a communication action. Roles are written in uppercase and the actor playing this role (the "actor reference") is written in lowercase between brackets. For example, the label "A(a1)" means that the communication action in question is performed by actor a1 performing role A.

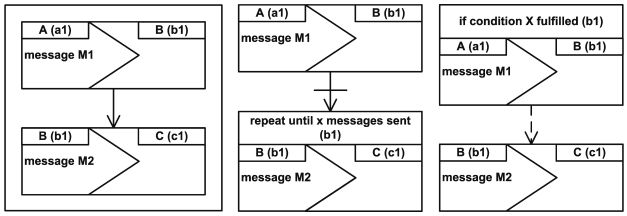

Fig. 1. Constructs of Let's Dance

Interactions can be inter-related using the constructs depicted in Figure 1. The relationship on the left-hand side is called "precedes" and is depicted by a directed edge: the source interaction can only occur after the target interaction has occurred. That is, after the receipt of a message "M1" by "B", "B" is able to send a message "M2" to "C". The rectangle surrounding these two interactions denotes a composite interaction, which can be related with other interactions with any type of relationship. The relationship at the center of the figure is called "inhibits", depicted by a crossed directed edge. It denotes that after the source interaction has occurred, the target interaction can no longer occur. That is, after "B" has

received a message "M1" from "A", it may not send a message "M2" to "C". The latter interaction can be repeated until "x" messages have been sent, which is indicated by the header on top of the interaction. The actor executing the repetition instruction is noted in brackets. Finally, the relationship on the right-hand side of the figure, called "weak-precedes", denotes that "B" is not able to send a message "M2" until "A" has sent a message "M1" or until this interaction has been inhibited. That is, the target interaction can only occur after the source interaction has reached a final status, which may be "completed" or "skipped" (i.e. "inhibited"). In the example, the upper interaction has a guard assigned, which is denoted by the header on top of the interaction. This interaction is only executed if the guard evaluates to true. The actor who evaluates the guard is noted in brackets.

B. Example

An example of a choreography corresponding to a loan application process is depicted in Figure 2. A client "c" sends a (loan) application to the loan department "l" of a financial institution. Once this interaction is completed, a composite interaction is enabled. This composite interaction contains two guarded sub-interactions. The two elementary interactions on the left-hand side take place only if a credit check is requested. If so, the loan department sends a message "check credit" to the Bureau of Credit Registration (BCR) and receives the credit information for the requested client. If no credit check is requested and the guard evaluates to false, both elementary interactions are inhibited. The two elementary interactions on the right-hand side are executed only if the loan department requests an optional insurance. Since a composite interaction is completed if all sub-interactions have been executed or inhibited, the succeeding interactions are enabled even if both guards inside the composite interaction evaluate to false. For connecting the two following interactions with the composite interaction, a connector has been used for multiple arrows. The loan department either sends a rejection of the application to the client or issues a request for payment with the payment department, whereby here a two-way inhibits relationship is used. If a request for payment is issued, payment notifications for each of the accounts nominated by the client are sent. This is captured through the repetition of the interaction using an (informally) specified condition in the box at the top corner of the interaction.

Figure 3 shows the loan application choreography from the viewpoint of actor "c" which plays the role of a client. This local model starts with the client sending an application to the loan department, followed by the receipt of either a rejection or payment notification. The two-way inhibits relationship in the model is derived: it does not explicitly appear in the global model. The same holds for the precedes relationship that targets the interaction for payment notification. This illustrates that local models may contain relationships that are derived from, but not explicitly represented in the global model.

C. Abstract Syntax

The abstract syntax of the language is formally captured by the following definition of a Let's Dance choreography.

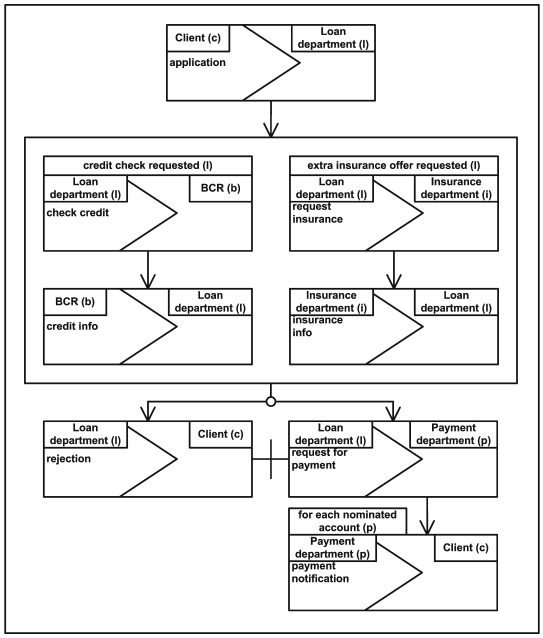

Fig. 2. Choreography of a loan application

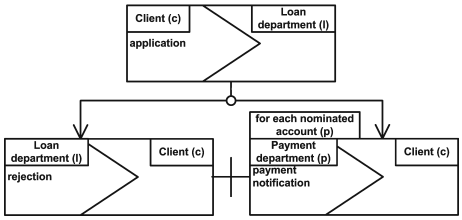

Fig. 3. Local model for the client

Definition 1. A *Choreography* (or *Global Model*) is a tuple (I, E, RE, GE, A, c_0, $Precedes$, $WeakPrecedes$, $Inhibits$, $Parent$, $Assignments$, $Performs$, $Conducts$) such that: *(a) I* is a set of Interactions; *(b) E* is a set of Expressions; *(c) RE* $\subseteq$ E is a set of Repeat-Expressions; *(d) GE* $\subseteq$ E is a set of Guard-Expressions; *(e) A* is a set of Actors; *(f)* $c_0 \in$ I is the top-level interaction of the choreography; *(g) Precedes*, *WeakPrecedes*, *Inhibits* $\subseteq$ I $\times$ I are three binary relations over the set of interactions I; *(h) Parent* $\subseteq$ I $\times$ I is the relation between interactions and their direct sub-interactions; *(i) Assignments* $\subseteq$ I $\times$ E is the relation between interactions and expressions; *(j) Performs*: I $\rightarrow \wp$(A) is a partial function linking interactions to actors; *(k) Conducts*: E $\rightarrow \wp$(A) is a partial function linking expressions to actors.

In the following definition, the symbol $Ancestor$ denotes the transitive closure of relation $Parent$, i.e. $Ancestor = Parent^+$. The sets *RI* $\subseteq$ I and *GI* $\subseteq$ I are used as abbreviations for Repeated Interactions and Guarded Interactions respectively and are defined as follows:

- $RI = \{r \in I \mid \exists e \in\ RE\ [(r, e) \in Assignments]\}$
- $GI = \{g \in I \mid \exists e \in\ GE\ [(g, e) \in Assignments]\}$

The definition of a well-formed choreography below captures certain syntactic constraints that exclude some incorrect choreographies. In the rest of the paper, we assume all choreographies to be well-formed.

Definition 2. A choreography $C = (I, E, RE, GE, A, c_0, Precedes, WeakPrecedes, Inhibits, Parent, Assignments, Performs, Conducts)$ is well-formed if:

- c_0 has no parent: $\neg\exists i \in I[i\ Parent\ c_0]$
- Each interaction other than the root has one and only one parent: $\forall i \in I \setminus \{c_0\}\ \exists! j \in I[j\ Parent\ i]$
- No relation that starts inside a repeated (composite) interaction crosses the boundary of this interaction: $\forall i, j \in I\ \forall k \in RI[(k\ Ancestor\ i \wedge (i\ Precedes\ j \vee i\ \textit{WeakPrecedes}\ j \vee i\ Inhibits\ j)) \rightarrow k\ Ancestor\ j]$
- There are no "precedence dependencies" between ancestors and descendants: $Ancestor \cap (Precedes \cup \textit{WeakPrecedes}) = \emptyset$
- There are no cyclic precedence dependencies: $Precedes \cup \textit{WeakPrecedes} \cup Parent$ is acyclic
- An interaction involves at most two actors: $\forall i \in I[1 \leq |Performs(i)| \leq 2]$

III. Local Enforceability

As previously mentioned, a choreography may include relationships that are not locally enforceable. For example, figure 4 shows a sample choreography with non-locally enforceable relationships. Specifically, the precedes relationship between the two interactions on top of the figure is not enforceable. This relationship denotes that the sending of a message "M2" can only occur after the sending of a message "M1" (as perceived by an ideal global observer). Since every communication action is carried out by a different actor, it is not possible for actors "c1" or "d1" to know when has a message "M1" been sent. On the other hand, the two-way inhibits relationship between the interactions labeled "M1" and "M3" is enforceable, since there are two common actors performing these interactions, namely "a1" and "b1", and these actors can enforce the inhibits relationship in their respective local models. The same holds for the precedes relationship connecting the interactions labeled "M3" and "M4". These two interactions only share one common actor, namely "a1", but since "a1" will not start sending a message "M4" until it has received a message "M3", the precedes relationship can be enforced in the local model of "a1". The source and target interactions of the weak-precedes relationship at the bottom of the figure also share a common actor, namely "e1". Nevertheless this relationship is not enforceable. Indeed, according to the meaning of the weak-precedes relation, message "M5" may need to be exchanged even if the source interaction of the weak-precedes relationship has been skipped. Thus, "e1" needs to know when this source interaction is skipped. In the depicted choreography the source interaction will be skipped if the interaction labeled "M1" occurs, because in this case the interaction labeled "M3" and therewith also the interaction labeled "M4" will never occur due to the inhibits relationship. Since "e1" is not involved in the interaction labeled "M1", it will never know that the source interaction of the weak-precedes relationship has been skipped. The situation would be different if the interaction labeled "M2" was performed by "b1" and "e1". In this case "e1" would know that the interaction for sending a message "M4" has been skipped, as soon as it sees a message "M2", and thus "e1" can enforce the weak-precedes relationship.

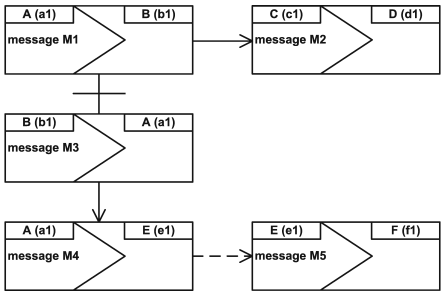

Fig. 4. Non-enforceable choreography

The above example illustrates the main issues for checking enforceability. In this section, we present an algorithm for checking the local enforceability of relationships between elementary interactions. This algorithm relies on two auxiliary algorithms: (i) an algorithm for "expanding" a choreography into an equivalent choreography in which every relationship involves only elementary interactions (as opposed to composite interactions); and (ii) an algorithm for detecting interactions in a choreography that will never be executed due to conflicting relationships in the choreography. The definitions of these algorithms rely on the following notations:

- $EI \subseteq I$ is the set of *Elementary Interactions* $EI = \{i \in I \mid \neg\exists j[i\ Parent\ j]\}$
- $CI \subseteq I$ is the set of *Composite Interactions* $CI = I \setminus EI$
- $Parent(i)$ is the parent of interaction i $\forall i \in I \setminus \{c_0\}\ Parent(i) =$ the only element in the set $\{p \in I \mid p\ Parent\ i\}$
- $Ancestors(i)$ is the set of ancestors of i $Ancestors(i) = \{a \in I \mid a\ Parent^+\ i\}$
- $Initial : CI \rightarrow \wp(EI)$ computes the set of elementary interactions in a composite interaction that are not target of control dependencies from other interactions in the same composite interaction: $Initial(ci) = \{ei \in EI \mid ci\ Ancestor\ ei \wedge \neg\exists k, m \in I[ci\ Ancestor\ k \wedge (k, m) \in Precedes \cup WeakPrecedes \wedge (m\ Ancestor\ ei \vee m = ei)]\}$
- $Final : CI \rightarrow \wp(EI)$ computes the set of elementary interactions in a composite interaction that are not the source of control dependencies leading to other interactions in the same composite interaction: $Final(ci) = \{ei \in EI \mid ci\ Ancestor\ ei \wedge \neg\exists k, m \in I[(ci\ Ancestor\ k) \wedge (m, k) \in (Precedes \cup \textit{WeakPrecedes}) \wedge (m\ Ancestor\ ei \vee m = ei)]\}$
- *MT* is the message type of an elementary interaction
- *Receiver*: $EI \rightarrow \wp(A)$ is a partial function linking elementary interactions to actors
- *Sender*: $EI \rightarrow \wp(A)$ is a partial function linking elementary interactions to actors
- *MessageType*: $EI \rightarrow MT$ is a partial function linking elementary interactions to message types
- *RepetitionType*: $RI \rightarrow (repeatUntil|while|forEach)$ is

a partial function linking repeated interactions to repetition types
- *Expression*: $I \rightarrow E$ is a partial function linking interactions to expressions
- *Evaluates*: $I \rightarrow \wp(A)$ is a partial function linking interactions to actors: $Evaluates(i) = \{a \in A \mid \exists e \in GI\ ((i,e) \in Assignments \wedge a \in Conducts(e))\}$
- *Children* = *Parent*$^{-1}$
- *TopLevelActivity* is a function returning the top element of a given BPEL code
- *B* is a Block with $EB = \{b \in dom(BM) | BM(b) = TopLevelActivity(receive(P,M)) \vee BM(b) = TopLevelActivity(invoke(P,M))\}$
- *EB* is an Elementary Block
- *BM* is a Block Mapping

A. Choreography expansion

The algorithm for expanding composite interactions is presented in Figure 5. This algorithm first adds every pair of interactions to the set of precedes relationships that has a composite interaction in between and where there exists a consecutive set of precedes relationships connecting these three interactions. This auxiliary construct is introduced in order to detect the enforceability of expanded interactions, introduced below. After that the relationships originating from a composite interaction are treated. Lines 2 to 7 of the algorithm denote the substitution of all relationships which source is a composite interaction. A composite interaction is completed if all sub-interactions have been completed or inhibited. Thus, a synchronization point in form of an interaction $Sync_{c,j}$ has to be added in order to be able to define whether the composite interaction has been completed or not. Thus, the actors executing the communication actions of this interaction have to be the common actors of all final interactions of the composite interaction and the actors executing the interaction that is the target of the relationship in question. It might be that there exists no common actor executing these interaction. In this case there is no actor assigned, which will be discovered in the enforceability algorithm presented later. In lines 6 and 7 new relationship involving this synchronization point are established: $Sync_{c,j}$ is the source of a new relationship of the considered type connecting to the target of the original relationship. Moreover it is the target of weak-precedes relationships from all final interactions of the considered composite interaction. In line 8 all relationships of the considered type which source is a composite interaction are deleted.

The second part of the algorithm deals with relationships targeting a composite interaction. In lines 9 to 11 all precedes- and weak-precedes relationships which target is a composite interaction are substituted. When a composite interaction is enabled, its initial sub-interactions should be enabled. Thus, precedes- and weak-precedes relationships are substituted with relationships of the respective type from the source of the original relationship to the initial interactions of the composite interaction. The last two lines of the algorithm depict the substitution of inhibits relationships which target is a composite interaction. In this case, the new relationships are established to every sub-interaction of the composite interaction.

B. Reachability analysis

Well-formed choreographies in Let's Dance may contain *unreachable interactions*, that is, interactions that will never occur in any execution of the choreography. The presence of unreachable interactions makes the analysis of choreographies more difficult. Choreographies containing interactions with unreachable interactions are semantically incorrect, and it is thus normal to expect that they should be corrected prior to analysing them further in view of generating local models. Three patterns lead to an interaction i being unreachable.

1) Two interactions with vice-versa inhibits relations precede i (Figure 6(a)).
2) An interaction preceding i, inhibits i (Figure 6(b)).
3) An interaction j that always executes inhibits i and there is also a path of precedes and weak-precedes relations from that interaction to i (Figure 6(c)).

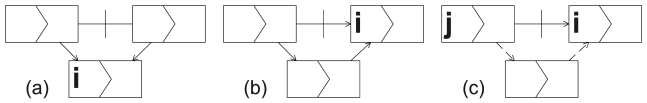

Fig. 6. Unreachable Interactions

An algorithm for detecting unreachable interactions is presented in Figure 7. This algorithm takes as input an expanded choreography (i.e. pre-processed by the expansion algorithm in Figure 5), and produces a set of unreachable interactions U. The algorithm makes use of the following auxiliary relations:

- $Prec = Precedes \cup WeakPrecedes$
- $Inhibits'$: a variant of $Inhibits$ that excludes certain relationships that do not have any effect. Such relationships are characterized by the fact that the target interaction can only occur after the source interaction has completed and thus the source never manages to actually inhibit the target (e.g. two interactions being connected with a precedes relationship in one direction and with a inhibits relationship in the opposite direction). Thus: $Inhibits' = Inhibits \setminus (Prec^{-1})^*$.

Lines 2-5 of the algorithm take care of patterns 1 and 2 identified above. Pattern 3 is more difficult to detect since there is the condition that interaction j has to always execute. Preceding guarded interactions (line 7) and targets of inhibits relations might prevent this (line 8). However, there are two cases where inhibits relations do not have any effect: (i) if the target always happens before the source (which we address by using $Inhibits'$ instead of $Inhibits$); or (ii) if the source is unreachable. To detect this latter case we proceed in two steps. For each interaction i, if there is an inhibits relationship targeting a preceding interaction j, we temporarily classify i as unreachable but we keep track that this classification is subject to revision. We do this by inserting a tuple into an auxiliary relation called $Depends$, indicating that the reachability of i "depends" on the reachability of the source of the inhibits relation targeting j (lines 12-13). Indeed, if the source of this inhibits relation turns out to be itself unreachable, and thus will never be executed, the inibits relationship in question will never impede the execution of j. In a second step, after

1: $Precedes := Precedes \cup \{(i,j) \in I \times I \mid \exists k \in CI\ [i\ Precedes\ k \wedge k\ Precedes^*\ j]\}$
2: **for each** $R \in \{Precedes, \textit{WeakPrecedes}, Inhibits\}$
3: **for each** $(c,j) \in R$ **where** $c \in CI$
4: $I := I \cup \{Sync_{c,j}\}$ (*this creates a new interaction $Sync_{c,j}$*)
5: $Performs(Sync_{c,j}) := \{a \in Actors \mid (\forall f \in Final(c)\ [a \in Performs(f)]) \wedge a \in Performs(j)\}$
6: $R := R \cup \{(Sync_{c,j}, j)\ \mid c\ R\ j\}$
7: $\textit{WeakPrecedes} := \textit{WeakPrecedes} \cup \{(i, Sync_{c,j})\ \mid i \in Final(c)\}$
8: $R := R \setminus (CI \times I)$
9: **for each** $R \in \{Precedes, \textit{WeakPrecedes}\}$
10: $R := R \cup \{(i,j) \in I \times I \mid \exists a \in Ancestors(j)\ [j \in Initial(a) \wedge i\ R\ a]\}$
11: $R := R \setminus (I \times CI)$
12: $Inhibits := Inhibits \cup \{(i,j) \in I \times I \mid \exists a \in Ancestors(j)\ [i\ Inhibits\ a]\}$
13: $Inhibits := Inhibits \setminus (I \times CI)$

Fig. 5. Algorithm for expanding relationships involving composite interactions

1: U := {}; Depends := {};
2: **for each** $(i,j) \in Precedes$ **sorted topologically**
3: **if** $(i \in U \vee \exists x \in I\ [x\ Inhibits\ j \wedge x\ Precedes^*\ i]$
4: $\vee \exists x,y \in I\ [x\ Inhibits\ y \wedge y\ Inhibits\ x \wedge x\ Precedes^*\ i \wedge y\ Precedes^*\ j])$ **then**
5: $U := U \cup \{j\}$
6: **for each** $(i,j) \in Inhibits$
7: **if** $\exists (x,y) \in WeakPrecedes\ [i\ Prec^*\ x \wedge y\ Prec^*\ j \wedge \neg \exists w \in GI(w\ Precedes^*\ x)]$ **then**
8: **if** $\neg \exists (v,w) \in Inhibits'\ [w\ Precedes^*\ i]$ **then**
9: $U := U \cup \{k \in I \mid j\ Precedes^*\ k\}$
10: **else**
11: $U := U \cup \{j\}$
12: **for each** $(v,w) \in Inhibits'$ **where** $w\ Precedes^*\ i$
13: $Depends := Depends \cup \{(j,v)\}$
14: **for each** $i \in I$ **where** $\exists j \in I(i\ Depends\ j)$
15: **if** $\neg \exists j \in I \setminus U(i\ Depends^+\ j \wedge \neg \exists k \in U(k\ Precedes^+\ j))$**then**
16: $U := U \cup \{k \mid i\ Precedes^*\ k\}$
17: **else**
18: $U := U \setminus \{i\}$

Fig. 7. Algorithm for detecting unreachable interactions

having fully populated the relation *Depends*, if it is found that all the interactions upon which the reachability of i depends have been classified as unreachable, i and all its successors are definitely classified as unreachable as well (lines 15-16). Otherwise i is classified as reachable (line 18).

In the rest of the paper, we restrict ourselves to choreographies without unreachable interactions, that is $U = \emptyset$.

C. Enforceability algorithm

The algorithm for checking local enforceability is presented in Figure 8. This algorithm implements a function that takes as parameter an expanded global model (i.e. a global model after applying the algorithm in Figure 5) and produces a set of pairs of interactions (i,j) such that there is a non-locally enforceable constraint between i and j. This set of pairs is named *UR*. Moreover, the algorithm produces as set of interaction *UI* that include guarded and repeated interactions which can not be enforced.

In lines 1-3 of the algorithm, each relationship is checked to ensure that there is at least one common actor involved both in the source and in the target interaction. If this is not the case, the relationship is added to the set of relationships that are not locally enforceable. The "for each" loop starting in line 4 deals with more complex requirements for weak-precedes relationships between pairs of interactions (i,j). The "for each"-loop from line 5 to line 8 checks each inhibits relationships that could cause the source of the weak-precedes relationship (i) to be skipped. The first part of the condition in the if-clause can only evaluate to true, if there is no direct or transitive precedes relationship between the interaction that can be inhibited (l) and interaction j, since in this case the enforceability is already ensured. The second part of the condition that has to be fulfilled in order to add the pair of interactions (i,j) to the set of interactions with a non-locally enforceable constraint starts after the $\wedge$-symbol in line 6 and ends with line 7. It evaluates to true if there exists an actor that is involved in the execution of interaction j and in an interaction that is involved in a path of consecutive precedes relationships going from the interaction that might be inhibited (l) to the source of the weak-precedes relationship (i), and this

1: **for each** $(i,j) \in Precedes \cup Inhibits' \cup WeakPrecedes$
2: **if** $Performs(i) \cap Performs(j) < 1$
3: **then** $UR := UR \cup \{(i,j)\}$
4: **for each** $(i,j) \in WeakPrecedes$
5: **for each** $k,l \in I$ **where** $k\ Inhibits'\ l \wedge l\ Precedes^*\ i$
6: **if** $\neg(l\ Precedes^+\ j) \wedge (\exists a \in Performs(j)\ \exists m \in I\ \neg\exists n \in I\ [l\ Precedes^*\ m \wedge m\ Precedes^*\ i \wedge$
7: $a \in Performs(m) \wedge a \in Performs(n) \wedge k\ Heralds\ n]$
8: **then** $UR := UR \cup \{(i,j)\}$
9: **for each** $k \in GI$ **where** $k\ Precedes^*\ i$
10: **if** $\neg(k\ Precedes^+\ j) \wedge (Performs(j) \cap Evaluates(k) \cap Performs(i) = \emptyset \vee \exists a \in Performs(j)\ \exists m \in I$
11: $[k\ Precedes^*\ m \wedge m\ Precedes^*\ i \wedge a \in Performs(m) \wedge a \notin Evaluates(k)])$
12: **then** $UR := UR \cup \{(i,j)\}$
13: **for each** $i \in (RI \cup GI)$
14: **if** $Coordinators(i) = \emptyset$
15: **then** $UI := UI \cup \{i\}$
16: **for each** $(k,j) \in Precedes \cup WeakPrecedes \cup Inhibits'$ **where** $i \notin Ancestors(j) \wedge i \in Ancestors(k)$
17: **if** $Performs(j) \cap Performs(k) \cap Coordinators(c) = \emptyset$
18: **then** $UR := UR \cup \{(k,j)\}$
19: **for each** $(j,k) \in Precedes \cup WeakPrecedes \cup Inhibits'$ **where** $i \notin Ancestors(j) \wedge i \in Ancestors(k)$
20: **if** $Performs(j) \cap Performs(k) \cap Coordinators(c) = \emptyset$
21: **then** $UR := UR \cup \{(j,k)\}$

Fig. 8. Algorithm for determining local enforceability

actor not being involved in the execution of an interaction following the source of the inhibits relationship (k). This additional condition formulates the necessity, that any actor that is involved in executing interaction j and that is involved in the execution of an interaction that might be skipped due to the inhibits relationship, must have knowledge about the result of the evaluation of the guard. Therefore, the interaction that provides this knowledge to the respective actor (n), must be executed in any possible instance of the choreography, which is expressed by the relating this interaction with a *Heralds* relationship. We say that an interaction x "heralds" another interaction y if in any run where interaction x occurs, interaction y necessarily occurs subsequently. If the above "heralds" relationship holds between j and some suitable n, holds, it can be assured that at least one of the actors involved in j will know that i has been skipped (since they would know that an alternative path was taken when interaction n eventually occurs), and thus the actor(s) in question would know that they can complete interaction i. In this case, the weak-precedes relationship between i and j is enforceable, otherwise it is not enforceable and the pair (i,j) is added to UR. An interaction may be skipped either due to the presence of guards or due to inhibits relationships, hence the relation $Heralds$ is defined as follows:

$$Precedes^* \setminus \{(i,j) \in I \times I \mid \exists z \in I\ [i\ Precedes^+\ z \wedge z\ Precedes^*\ j\ \wedge (z \in GI \vee \exists x,y \in I\ [x\ Inhibits'\ y\ \wedge y\ Precedes^*\ z]]\}$$

The "for each" loop in lines 9-12 checks each guarded interaction that directly or transitively precedes i. A guarded interaction (say k) and all interactions directly or transitively connected to it via a precedes relationship, will be skipped if the guard evaluates to false. This "path skipping" must be known by at least one of the actors that perform i and j. Thus, there must be a common actor involved in the execution of i and j, and at least one of these actors must evaluate the guard in question, so as to know whether the path is skipped or not. Additionally, any actor that is involved both in j and in an interaction lying on a path of consecutive precedes relationships from k to i, must also evaluate the guard. In other words, any actor involved in interaction j and in an interaction that may be skipped if the guard evaluates to false, must know the result of the evaluation of the guard. If any of these conditions is not fulfilled, the weak-precedes relationship from i to j is not locally enforceable.

The "for each" loop in lines 13-21 deals with special requirements for repeated interactions. This part of the algorithms uses an auxiliary function called *Coordinators* that retrieves a set of actors. For elementary interactions, this set contains all actors that perform a given interaction and that evaluate the guard or execute the repetition instruction if any. For composite interactions, this set contains all actors that are involved in the execution of all initial and all final interactions, and that are involved in the evaluation of the guard or the execution of the repetition instruction respectively.

$$\begin{aligned} &Coordinators(i) := \textbf{ if } i \in EI \\ &\textbf{then } \{a \in Actors \mid \exists e \in E\ [(i,e) \in Assignments \wedge \\ &a \in Performs(i) \wedge a \in Conducts(e)] \\ &\textbf{else } \{a \in Actors \mid \exists e \in E, \exists x \in I, \\ &\exists y \in I\ [(i,e) \in Assignments \wedge x \in Initial(i) \wedge \\ &y \in Final(i) \wedge a \in \{Performs(x) \cap Performs(y) \\ &\cap Conducts(e)\}]\} \end{aligned}$$

In lines 14 and 15 the set of coordinators is checked. If this set is empty, the respective interaction is added to the set of unenforceable interactions. The for-each loop from line 16 to

18 iterates over all relationships, where a sub-interaction of the considered interaction is the source of a relationship to an interaction that is not a sub-interaction of the considered interaction. For such relationships to be enforceable, there must exist an actor involved in the interactions j and k that is part of the set of coordinators of the considered interaction. In all other cases, there exists no actor that is able enforce the relationship in question. Thus, the relationship would be added to the set of unenforceable relationships. The last three lines of the algorithm are very similar to the ones before: the only difference is that in this case all relationships are examined, where the target of a relationship is a sub-interaction of the interaction in question and the source is not.

Since the enforceability checking algorithm is far from trivial, a closer examination of its computational complexity is warranted. We first observe that the **for each** loop in lines 1 to 3 encodes a simple iteration over the set of tuples in the union of the *Precedes*, *Inhibits* and *WeakPecedes* relations. Below, we use symbol E to designate this union. Thus, the complexity of this first loop is bounded by $O(|E|)$. The second **for each** loop (lines 4 to 12) iterates over the tuples in *WeakPrecedes*. For each of these tuples, it performs one backward graph traversal along *Precedes* relation and for each element visited in this backward traversal that is the target of an inhibits relationship, the algorithm performs a forward traversal starting from the source of this inhibits relationship (lines 5-8). A similar procedure is followed in lines 9-12 to check for guarded interactions, so the complexity of lines 9-12 as it is in the same order as that for lines 5-8. If we implement graph traversals using depth-first search, we can bound the complexity of a traversal by $O(|I| + |E|)$, and since the traversals are nested, their combined complexity is bounded by $O((|I|+|E|)^2)$. Hence, the complexity of the **for each** loop in lines 4 to 12 is bounded by $O(|I| \times (|I|+|E|)^2)$. Finally, the third **for each** loop in lines 13 to 21 iterates over the set of guarded or repeated interactions. In each iteration, the algorithm visits each tuple in the union of the *Precedes*, *Inhibits* and *WeakPecedes* relations and tests if the elements in the interactions in this tuple are ancestors of the guarded or repeated interaction being examined. Assuming we materialize the $Ancestors$ function as a hash table beforehand, this test can be done in constant time, and the complexity of this loop is bounded by $O(|I| \times |E|)$. In summary, the complexity of the algorithm is dominated by the complexity of the second outer **for each** loop which is $O(|I| \times (|I| + |E|)^2)$. This means it is realistic to check enforceability even for models involving hundreds of interactions.

IV. Generating Local Models

The aim of the algorithm for generating local models is to provide a local view for each of the actors participating in a choreography. These local views should only include elementary interactions where at least one of the communication actions is executed by the actor for which the local view is generated. The challenge in generating local views is the derivation of the correct relationships between interactions. We illustrate this issue using the global view depicted in Figure 9. There are three participating actors in this global model: "a1" playing the role "A", "b1" playing the role "B" and "c1" playing the role "C".

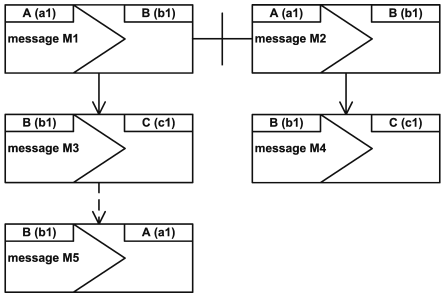

Fig. 9. Sample global model

The local view for actor "b1" is equal to the whole choreography since this actor participates in every interaction. Meanwhile, the local model of actor "a1" consists of the two interactions labeled "M1" and "M2" related via a two-way inhibits relationship, plus the elementary interaction labeled "M5". The latter interaction is not directly related with "M1" in the global model, yet, in order to preserve the semantics of the global model, the local model of "a1" needs to explicitly render the "derived" weak-precedes relationship between "M1" and "M5" (see the local model of actor "a1" in Figure 10). Finally, the local model of "c1" consists of the two interactions labeled "M3" and "M4". These interactions are unrelated in the choreography, yet, in the local model of "c1" they are related via a two-way inhibits relationship (see Figure 10), as only one of these two interactions will occur.

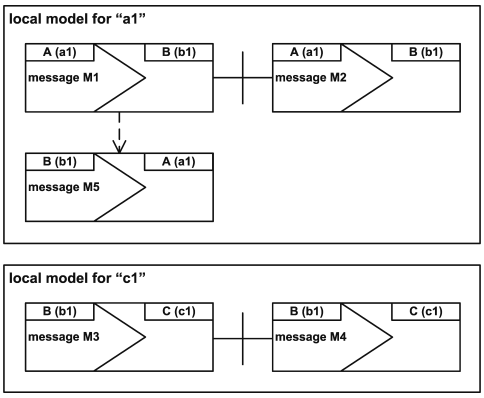

Fig. 10. Generated local models

The algorithm in Figure 11 describes the generation of the local model for a given actor in a choreography. Following the idea of abstracting from certain process steps in process algebra by viewing them as "silent" or τ-steps and then reducing the process to an equivalent one without τ-steps, the algorithm proceeds by viewing elementary interactions in which no communication action is executed by a considered actor as a silent or τ-interaction. A set of reduction rules are then applied to remove these τ-interactions. The algorithm presented below implements a procedure that, given a global model and an actor *a*, computes the local model corresponding

to the view of a on the assigned global model. The "for each"-loop in line 2 iterates over every τ-interaction in the considered local view and tries to apply the six reduction rules to this τ-interaction. The reduction rules can be integrated in three groups, which equals to the three "for each"-loops starting in lines 3, 10, and 17: the first one for τ-interactions that are the target of a precedes relationship, the second one for τ-interactions that are the target of a weak-precedes relationship and the third one for τ-interactions that are the source of a precedes relationship. Each of these three loops consists of two further for-each loops, whereby each of them represents the second relationship of a combination of two relationships to be reduced. Only with this structure it is possible to consider every possible combination of relationships and therewith not miss any possible reduction.

```
1: Procedure LocalModel(a: Actor)
2:   for each τ ∈ I where ¬a ∈ Performs(τ)
3:     for each i ∈ I where i Precedes τ
4:       for each j ∈ I where τ Precedes j
5:         Precedes := Precedes ∪ {(i, j)}
6:         if ∃e ∈ E[(τ, e) ∈ Assignments]
7:         then Assignments := Assignments ∪ {(j, e)}
8:       for each j ∈ I where τ WeakPrecedes j
9:         WeakPrecedes := WeakPrecedes ∪{(i, j)}
10:    for each i ∈ I where i WeakPrecedes τ
11:      for each j ∈ I where τ WeakPrecedes j
12:        WeakPrecedes := WeakPrecedes ∪{(i, j)}
13:      for each j ∈ I where τ Precedes j
14:        WeakPrecedes := WeakPrecedes ∪{(i, j)}
15:        if ∃e ∈ E[(τ, e) ∈ Assignments]
16:        then Assignments := Assignments ∪ {(j, e)}
17:    for each i ∈ I where τ Precedes i
18:      for each j ∈ I where i Inhibits' τ
19:        Inhibits' := Inhibits' ∪ {(j, i)}
20:        if ∃e ∈ E[(τ, e) ∈ Assignments]
21:        then Assignments := Assignments ∪ {(i, e)}
22:      for each j ∈ I where τ Inhibits' j
23:        Inhibits' := Inhibits' ∪ {(i, j)}
24:        if ∃e ∈ E[(τ, e) ∈ Assignments]
25:        then Assignments := Assignments ∪ {(i, e)}
26:    for each i ∈ I
27:      for each R ∈ {Precedes, WeakPrecedes,
28:                    Inhibits'}
29:        R := R \ {(i, τ), (τ, i)}
30:    I := I \ {τ}
```

Fig. 11. Algorithm for deriving local models from choreographies

The first reduction rule applies for every combination of two consecutive precedes relationships where the considered τ-interaction is in between. From this structure a precedes relationship between the source of the first and the target of the second precedes relationship can be derived. Additionally, if the τ-interaction has a guard or a repetition instruction assigned, then this expression is assigned to the interaction that is the target of the second precedes relationship. This pushing of expressions to following interactions is executed in four of the six reduction rules and might lead to expressions that are assigned to interactions which do not involve the actors nominated for evaluating this expression. In this case, the expression is considered as opaque. The second rule applies for a weak-precedes relationship as second relationship. In this case a weak-precedes relationship between the source of the precedes relationship and the target of the weak-precedes relationship can be derived.

The next two reduction rules apply when a τ-interaction is the target of a weak-precedes relationship: in the first rule with the τ-interaction being the source of an weak-precedes relationship, in the second rule with the τ-interaction being the source of a precedes relationship. In both cases a weak-precedes relationship is derived, connecting the source of the original weak-precedes relationship and the target of the second original relationship. In the latter case any expression assigned to the τ-interaction is assigned to the interaction that is the target of the precedes relationship. The last two reduction rules apply for combinations of one precedes and one inhibits relationship. The first one covers the cases where the τ-interaction is the source of a precedes relationship and the target of an inhibits relationship. This leads to a derived inhibits relationship where the source of the original inhibits relationship is the source and the target of the original precedes relationship is the target of the derived relationship. The last reduction rule covers all combinations of relationships where the considered τ-interaction is the source of a precedes and of an inhibits relationship. In this case an inhibits relationship can be derived, having the target of the original precedes relationship as source and the target of the original inhibits relationship as target. Both reduction rules involving inhibits relationships push down expressions to the target of the precedes relationship. Having applied all reduction rules, the considered τ-interaction and all relationships in which it is involved are deleted (lines 18-22).

V. Mapping Local Models to BPEL code

This section describes a method for transforming a local model into a skeleton of a BPEL process definition. The mapping is defined in terms of 12 transformation rules corresponding to different "structural patterns" of Let's Dance models. When a match is found between a model fragment and the pattern captured in a rule, the fragment is replaced by a "block" which acts as a placeholder for the BPEL code corresponding to the fragment. The block is treated as an elementary interaction for the purpose of applying further rules. By repeating this process, we can reduce a local model into a single block with its associated BPEL code. The order in which rules are presented is important since some rules are more specific than others and generate better BPEL code.

None of the proposed rules is able to deal with relationships that cross the boundary of a composite interaction. Prior to applying the mapping rules, composite interactions in the local model should be expanded using the algorithm of Figure 5.

A. Mapping Rules

a) Substitution functions: The two functions $\sigma_{x\to y}(C)$ and $\sigma_{x\to y}(R)$ defined below encode basic transformation steps

that a choreography will undergo when applying a mapping rule. The first function takes as input a choreography, and produces a new choreography identical to the original one except that any occurrence of a given element *x* is replaced by another element *y*. Thus, the new element is added to the set of interactions *I*, the replaced element is removed, and the $Precedes_c$, $WeakPrecedes_c$, *Inhibits'*, $Parent_c$, and $Assignments_c$ relations are updated accordingly. By element, we mean either an interaction (elementary or composite) or a block. The second auxiliary function (which is in the definition of the first one) takes as input a binary relation (e.g. a $WeakPrecedes_c$ relation) and replaces any occurrence of a given element *x* by another element *y*. It is worth noting that repeated applications of these substitution functions may lead to undesired self-loops. For example, if in the original choreography there existed a precedes relationship between *x* and *y* and both *x* and *y* are replaced by a new element *z*, we would obtain a relation where *z* precedes itself. To avoid this situation, function $\sigma_{x\to y}(C)$ is defined in such a way that it removes any self-loop created by a replacement.

$\sigma_{x\to y}(C) = ((I \cup \{y\}) \setminus \{x\}, \sigma_{x\to y}(Precedes_c) \setminus \{(y,y)\}, \sigma_{x\to y}(WeakPrecedes_c) \setminus \{(y,y)\}, \sigma_{x\to y}(Inhibits'_c) \setminus \{(y,y)\}, \sigma_{x\to y}(Parent_c), \sigma_{x\to y}(Assignments_c))$

$\sigma_{x\to y}(R) = (R \cup \{(y,z) \mid (x,z) \in R\} \cup \{(z,y) \mid (z,x) \in R\} \setminus \{(x,z) \mid (x,z) \in R\} \setminus \{(z,x) \mid (z,x) \in R\}$

The mapping rules are depicted in Figure 12 and Figure 13 and formally described below. To represent BPEL code, we use an abbreviated BPEL notation. The association between "blocks" introduced by the mapping and their corresponding BPEL code fragment is captured by a function called ***BM*** – i.e. $BM(b)$ is the code corresponding to block b.

b) Rule 1: This rule deals with elementary interactions. The rule can be divided in two parts: the first part handles the substitution of elementary interactions where the sender is the actor for which the local model was generated (hereafter called the *actor of interest*). The second part deals with elementary interactions where the receiver is the actor of interest.

$$\frac{i \in EI, self = Sender(i)}{(C, BM) \to \begin{array}{c}(\sigma_{i\to b_i}(C),\ BM \cup \\ \{(b_i, invoke(Receiver(i), MessageType(i)))\})\end{array}}$$

$$\frac{i \in EI, self = Receiver(i)}{(C, BM) \to \begin{array}{c}(\sigma_{i\to b_i}(C),\ BM \cup \\ \{(b_i, receive(Sender(i), MessageType(i)))\})\end{array}}$$

c) Rule 2: This rule deals with composite interactions, which are captured in BPEL using the *flow* construct. As previously discussed, we assume that the boundary of the composite interaction is not crossed by any relationship. In addition, the children of the composite interaction must not to be related between them, and they must all be blocks obtained through previous applications of mapping rules. This is guaranteed by checking that all children are already part of the domain of the block mapping function ***BM***.

$$\frac{\begin{array}{c}i \in CI, Children(i) = \{bi_1, ..., bi_n\}, \\ Children(i) \setminus dom(BM) = \emptyset, \forall i,j \in \{bi_1, ..., bi_n\} : \\ (i,j) \notin (Precedes_c \cup WeakPrecedes_c \cup Inhibits'_c)\end{array}}{(C, BM) \to \begin{array}{c}(\sigma_{i\to b_i}(C),\ BM \cup \\ \{(b_i, flow(BM(bi_1), ..., BM(bi_n)))\})\end{array}}$$

d) Rule 3: A second Let's Dance pattern that can be mapped into the *flow* construct is defined below. Two blocks i and j can be transformed into a *flow* block if they both have the same set of predecessors and the same set of successors and if each block is connected with the same type of relationship (whether $Precedes$ or $WeakPrecedes$). Furthermore, both blocks must not have assigned any guard or repetition instruction (i.e. they do not appear in the domain of function $Assignments$), and they must not be the source or target of any inhibits relationship.

$$\frac{\begin{array}{c}i,j \in dom(BM), \\ \forall k \in I : (k,i) \in Precedes_c \Leftrightarrow (k,j) \in Precedes_c, \\ \forall k \in I : (k,i) \in WeakPrecedes_c \Leftrightarrow (k,j) \in WeakPrecedes_c, \\ \forall k \in I : (i,k) \in Precedes_c \Leftrightarrow (j,k) \in Precedes_c, \\ \forall k \in I : (i,k) \in WeakPrecedes_c \Leftrightarrow (j,k) \in WeakPrecedes_c, \\ \nexists (i,k) \in Assignments_c, \nexists (j,k) \in Assignments_c, \\ i,j \notin dom(Inhibits'_c), i,j \notin range(Inhibits'_c)\end{array}}{(C, BM) \to \begin{array}{c}(\sigma_{i\to b_{i,j}}(\sigma_{j\to b_{i,j}}(C)),\ BM \cup \\ \{(b_{i,j}, flow(BM(i), BM(j)))\})\end{array}}$$

e) Rule 4: A repeated block of type *repeatUntil* can be mapped to the BPEL construct with the same name if it has no children. I.e., all existing sub-interactions of the interaction to which the repetition instruction is assigned (if any) have been previously mapped, including the interaction itself.

$$\frac{\begin{array}{c}i \in RI \cap dom(BM), \\ RepetitionType(i) = repeatUntil, Children(i) = \emptyset\end{array}}{(C, BM) \to \begin{array}{c}(\sigma_{i\to b_i}(C),\ BM \cup \\ \{(b_i, repeatUntil(BM(i), Expression(i)))\})\end{array}}$$

f) Rule 5: A repeated block of type *while* is mapped in a way similar to rule 4.

$$\frac{\begin{array}{c}i \in RI \cap dom(BM), \\ RepetitionType(i) = while, Children(i) = \emptyset\end{array}}{(C, BM) \to \begin{array}{c}(\sigma_{i\to b_i}(C),\ BM \cup \\ \{(b_i, while(Expression(i), BM(i)))\})\end{array}}$$

g) Rule 6: A repeated block with repetition type *forEach* is mapped in a way similar to rules 4 and 5.

$$\frac{\begin{array}{c}i \in RI \cap dom(BM), \\ RepetitionType(i) = forEach, Children(i) = \emptyset\end{array}}{(C, BM) \to \begin{array}{c}(\sigma_{i\to b_i}(C),\ BM \cup \\ \{(b_i, forEach(Expression(i), BM(i)))\})\end{array}}$$

h) Rule 7: Two blocks that are connected with a precedes relationship and that are not repeated, can be mapped into a BPEL *sequence* if the following requirements are fulfilled. First, the target of the precedes relationship must not have any guard assigned. This is required because such guards can not be simply "lifted" to the new block created by this rule: if we did so, the guard would also constrain the execution of the source of the precedes relationship rather than only affecting its target. Second, the source of the precedes relationship must have no other outgoing arcs and the target must have no other incoming arcs. Finally, none of the two blocks must be source or target of an inhibits relationship since lifting any existing inhibits relationships to the new block would not preserve the

scope of the inhibits.

$$\frac{\begin{array}{c} i,j \in dom(BM), i,j \notin dom(RI), \\ (i,j) \in Precedes_c, j \notin dom(Assignments_c) \\ \mid \{(i,k) \mid (i,k) \in (Precedes_c \cup WeakPrecdes_c)\} \mid = 1, \\ \mid \{(k,j) \mid (k,j) \in (Precedes_c \cup WeakPrecdes_c)\} \mid = 1, \\ i,j \notin dom(Inhibits'_c), i,j \notin range(Inhibits'_c) \end{array}}{(C, BM) \rightarrow \begin{array}{c} (\sigma_{i \rightarrow b_{i,j}}(\sigma_{j \rightarrow b_{i,j}}(C)),\ BM \cup \\ \{(b_{i,j}, sequence(BM(i), BM(j)))\}) \end{array}}$$

i) Rule 8: A guarded block can be mapped to an *ifThenElse* statement in BPEL provided that: (i) the guarded block has no children; and (ii) it is not the source of a precedes relationship. The guard expression of the block becomes the branching condition of the *ifThenElse* statement, the BPEL code associated with the block itself is placed in the "then" branch of the *ifThenElse* statement, and an empty action is placed in the "else" branch. It is important that the guarded block does not "strongly precedes" another interaction (cf. the second constraint above). Indeed, any such interaction must be skipped if the guard evaluates to false, and thus, must be embedded in the "then" branch rather than being left outside the *ifThenElse* statement. Thus, a rule causing the absorption of any such interaction into the guarded block (e.g. the generation of a *sequence* BPEL activity using rule 7) must be applied first, before this rule can be applied.

$$\frac{\begin{array}{c} i \in GI \cap dom(BM), \\ Children(i) = \emptyset, \nexists (i,j) \in Precedes_c \end{array}}{(C, BM) \rightarrow \begin{array}{c} (\sigma_{i \rightarrow b_i}(C),\ BM \cup \\ \{(b_i, ifThenElse(Expression(i), \\ BM(i), empty))\}) \end{array}}$$

j) Rule 9: The rule below captures a pattern where two elementary blocks are connected via a two-way inhibits relationship. This pattern can be mapped into an *ifThenElse* statement in BPEL, if both elementary blocks have the same set of predecessors and the same set of successors and if each element is connected via either only *Precedes* relationships or only *WeakPrecedes* relationships. Moreover, the two blocks must not be involved in any other inhibits relationship except the two-way inhibits relationship identified in the pattern. The branching condition is *opaque* (i.e. it is left unspecified), since there is not enough information to determine which block should be executed. To make the process executable, a developer would have to decide how to transform this opaque condition into a fully specified condition. In doing so, the developer may choose to turn this *ifThenElse* statement into a *pick* statement, meaning that the decision of which branch to take is determined by which interaction occurs first.

$$\frac{\begin{array}{c} (i,j) \in Inhibits'_c, (j,i) \in Inhibits'_c, i \in EB, j \in EB \\ \forall k \in I : (k,i) \in Precedes_c \Leftrightarrow (k,j) \in Precedes_c, \\ \forall k \in I : (k,i) \in WeakPrecedes_c \Leftrightarrow (k,j) \in WeakPrecedes_c, \\ \forall k \in I : (i,k) \in Precedes_c \Leftrightarrow (j,k) \in Precedes_c, \\ \forall k \in I : (i,k) \in WeakPrecedes_c \Leftrightarrow (j,k) \in WeakPrecedes_c, \\ \mid i \cap dom(Inhibits'_c) \mid = \mid j \cap dom(Inhibits'_c) \mid = 1, \\ \mid i \cap range(Inhibits'_c) \mid = \mid j \cap range(Inhibits'_c) \mid = 1 \end{array}}{(C, BM) \rightarrow \begin{array}{c} (\sigma_{i \rightarrow b_{i,j}}(\sigma_{j \rightarrow b_{i,j}}(C)),\ BM \cup \\ \{(b_{i,j}, ifThenElse(opaque, BM(i), BM(j)))\}) \end{array}}$$

This pattern describes only one case where a two-way inhibits relationship can be mapped into an *ifThenElse* block. There are a three minor variants of this pattern, where one of the two blocks has no successor, or none of them has a successor, but we do not show them as they are very similar.

The above nine rules capture specific structural patterns.

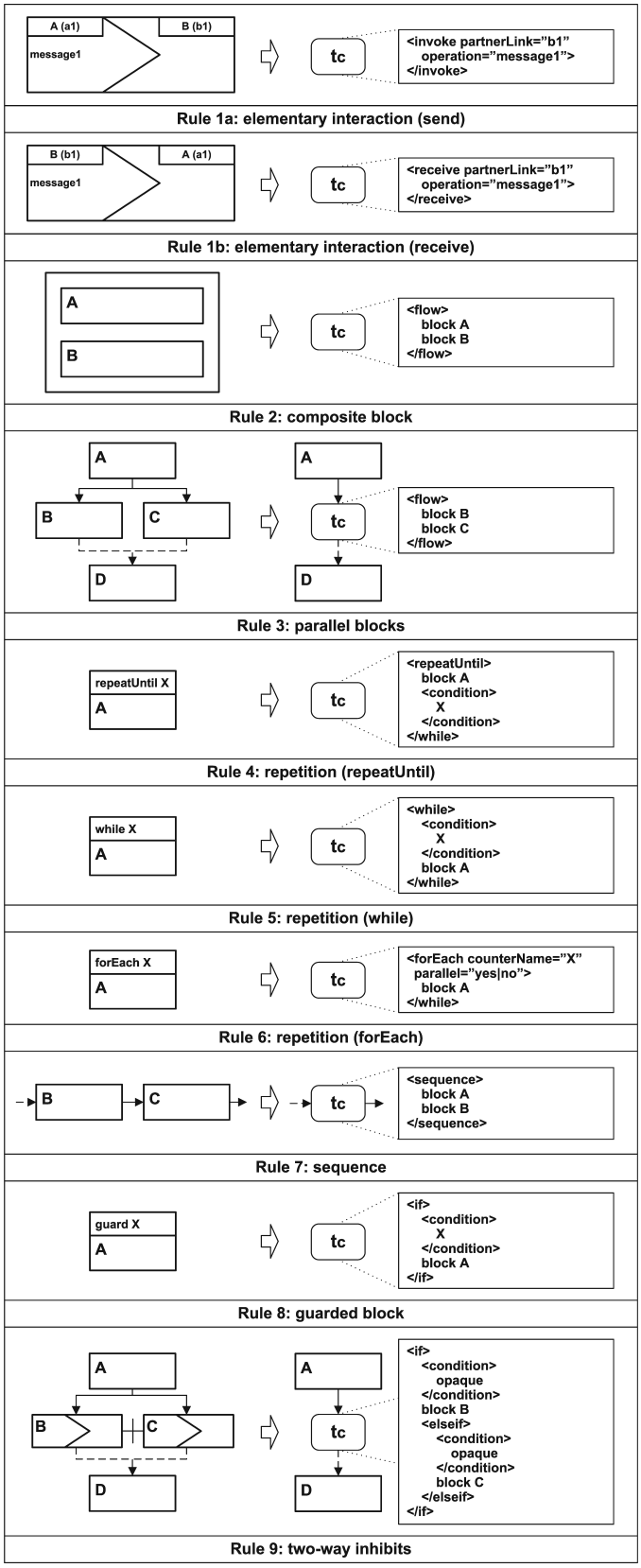

Fig. 12. Mapping rules 1 to 9

While they tend to generate structured BPEL code, they are not able to translate Let's Dance models with arbitrary topologies. To ensure completeness of the mapping, we provide three rules that can map inhibits, precedes and weak-precedes relationships, regardless of the context. These rules generate less structured and thus less readable code, and will only be applied when the above rules are not applicable.

k) Rule 10: Any two blocks connected by an inhibits relationship is mapped into two new blocks: one for the source and one for the target. The first block consists of a sequence in which the original source block is followed by the production of a self-addressed message. This self-addressed message acts as a trigger for an event handler included in the second block. The second block consists of a scope containing the second activity and an event handler. If the self-addressed message is produced, the event handler catches it and interrupts the scope, or if the scope has not yet been reached, the message is buffered until the scope is reached and the scope is exited immediately when reached. The inhibits relationship linking the matched blocks is deleted in the transformed model.

$$\frac{i,j \in dom(BM), (i,j) \in Inhibits'_c}{(C, BM) \rightarrow \begin{array}{l}(\sigma_{i\rightarrow b_i}(\sigma_{j\rightarrow b_j}(DelInhibits(C,(i,j))),\ BM \cup \\ \{(b_i, sequence(BM(i), invoke(self, inh_j))), \\ (b_j, scope(BM(j), eventHandler(inh_j, exScope)))\})\end{array}}$$

In this rule, $invoke(self, inh_j)$ stands for a BPEL action whereby the process sends an empty message of type inh_j to itself, while $exScope$ is an action that causes the enclosing scope to be exited. Function $DelInhibits$ takes as input a choreography and a pair of interactions connected by an inhibits relationship, and produces a new choreography identical to the original one but without this inhibits relationship.

l) Rule 11: Any two blocks connected via a precedes relationship can be mapped into a BPEL *link*. Any guard assigned to the target of the precedes relationship is mapped into a transition condition. The join condition of the target activity should evaluate to false if a negative token is passed along this link. Accordingly, this link is added as a conjunct to the join condition of the target BPEL activity. In this rule, we assume that the source activity is not a guarded interaction while the target activity is a guarded interaction, but it is straightforward to define variants of this rule to deal with the cases where there is a guard attached to the source activity or the case where no guard is attached to the target interaction.

$$\frac{i,j \in dom(BM), (i,j) \in Precedes_c}{(C, BM) \rightarrow \begin{array}{l}(\sigma_{i\rightarrow b_i}(\sigma_{j\rightarrow b_j}(DelPred(C,(i,j))),\ BM \cup \\ \{(b_i, addSource(BM(i), l, Expression(j))), \\ (b_j, addTarget(BM(j), l, joinCond(BM(j)) \wedge l))\})\end{array}}$$

Function $addSource$ takes as input a BPEL activity, a link identifier and a transition condition. The activity is made the source of the link and the transition condition is attached to this link. Similarly, function $addTarget$ sets the target of a link to be the activity given as first parameter, and associates a join condition given as third parameter to this activity.

m) Rule 12: Any two blocks connected by a weak-precedes relationship can be mapped into a BPEL *link*. This is similar to the previous rule, but the join condition for the target block does not include the newly added link, because the target actvitity will be executed regardless of whether the source activity is executed or not.

$$\frac{i,j \in dom(BM), (i,j) \in WeakPrecedes_c}{(C, BM) \rightarrow \begin{array}{l}(\sigma_{i\rightarrow b_i}(\sigma_{j\rightarrow b_j}(DelWkPred(C,(i,j))),\ BM \cup \\ \{(b_i, addSource(BM(i), l, Expression(j))), \\ (b_j, addTarget(BM(j), l, joinCond(BM(j))))\})\end{array}}$$

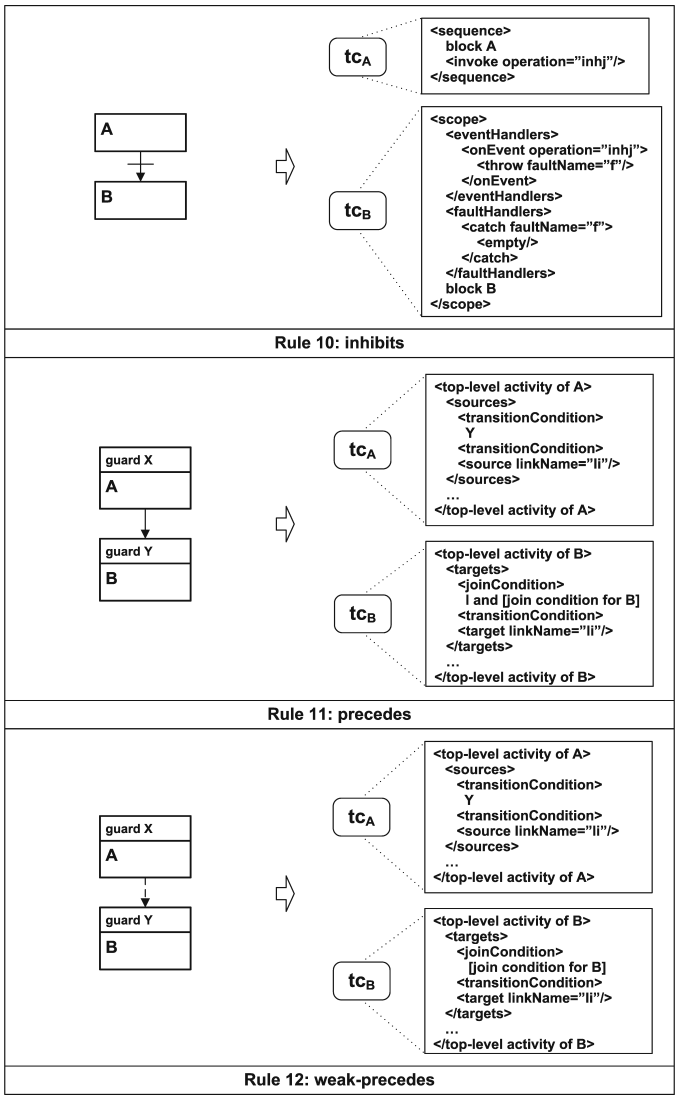

Fig. 13. Mapping rules 10 to 12

Each rule application reduces the size of the local model until there only remains one block with its associated BPEL code. To get more readable code, the rules should be applied in order. First, the rules are scanned from the first to the last one to determine if they match a fragment of the model. If a match is found, the rule is applied and the search for the next rule to be applied re-starts with the first rule.

B. Example

In section II-B we introduced a choreography for a loan application process. The local model for the loan department is depicted in the top left corner of Figure 14. This local model is similar to the whole choreography, since the only interaction

that does not involve the loan department is the payment notification. The step-by-step application of the mapping rules on this local model is depicted in Figure 14.

The first applicable rule is the one for mapping elementary interactions where the loan department is the sender of a message (Rule 1a). This rule can be applied to four elementary interactions, namely the two guarded interactions and the two interactions at the bottom that are connected via a two-way inhibits-relationship. The corresponding BPEL code is similar for blocks t_c^{1a} to t_c^{1d} in the top right corner of Figure 14. Thus, the code skeleton is only presented for the block named t_c^{1a}:

```
<invoke partnerLink="BCR(b)"
   operation="check credit">
</invoke>
```

The next rule to be applied is the one for elementary interactions for which the loan department is the receiver of a message (Rule 1b). This rule can be applied to the three remaining elementary interactions, which leads to the blocks named t_c^{2a} to t_c^{2c}. The resulting choreography involves only composite interactions and elementary blocks, where the latter ones stand for BPEL skeletons (cf. middle right of Figure 14). Again, the BPEL code is similar for all block, so we only present the code skeleton for block t_c^{2a}:

```
<receive partnerLink="Client(c)"
   operation="application">
</receive>
```

The generated choreography involves a composite interaction that consists of two sequences, whereby the first block of each sequence has a guard assigned. Thus, each of these sequences can be mapped into one BPEL block by applying rule 7. The two guards that were assigned to both of the first blocks are assigned to the to the corresponding block, that is generated by applying the rule. The BPEL code for block t_c^{3a} is defined as follows:

```
<sequence>
   <invoke partnerLink="BCR(b)"
      operation="check credit">
   </invoke>
   <receive partnerLink="BCR(b)"
      operation="credit info">
   </receive>
</sequence>
```

Having generated the two sequences and having assigned the two guards to the new blocks, it is possible to apply rule 8. This rule integrates the guard for block t_c^{3a} and t_c^{3b}. The generated BPEL code encapsulates the guard as a condition in an if-statement. Below is the code for block t_c^{4a}.

```
<if>
   <condition>
      credit check requested
   </condition>
   <sequence>
      <invoke partnerLink="BCR(b)"
         operation="check credit">
      </invoke>
      <receive partnerLink="BCR(b)"
         operation="credit info">
      </receive>
   </sequence>
</if>
```

Both blocks t_c^{4a} and t_c^{4b} are now the only elements within the composite interaction in the middle of the choreography. Thus, rule 2 can be applied that summarizes both blocks under a flow-construct and generates block t_c^5. Since this block is the only successor of block t_c^{2a}, these two blocks can be used to generate a sequence with applying rule 7. The BPEL code for block t_c^6 in the lower left corner of Figure 14 summarizes these two steps and is listed below:

```
<sequence>
   <receive partnerLink="Client()c"
      operation="application">
   </receive>
   <flow>
      <if>
         <condition>
            credit check requested
         </condition>
         <sequence>
            <invoke partnerLink="BCR(b)"
               operation="check credit">
            </invoke>
            <receive partnerLink="BCR(b)"
               operation="credit info">
            </receive>
         </sequence>
      </if>
      <if>
         <condition>
            extra insurance offer requested
         </condition>
         <sequence>
            <invoke partnerLink="BCR(b)"
               operation="check credit">
            </invoke>
            <receive partnerLink="BCR(b)"
               operation="credit info">
            </receive>
         </sequence>
      </if>
   </flow>
</sequence>
```

Block t_c^6 has two successors that are related with a two-way inhibits relationship and both successors are elementary blocks. Since the loan department is the sender of both of these messages, the elementary blocks can be mapped into a if-elseif-statement with applying rule 9. The BPEL code for block t_c^7 is defined as follows:

```
<if>
   <condition> opaque </condition>
   <invoke partnerLink="Client(c)"
      operation="rejection">
   </invoke>
   <elseif>
      <condition> opaque </condition>
      <invoke partnerLink="Payment department"
         operation="request for payment">
      </invoke>
   </elseif>
</if>
```

Finally, block t_c^8 is generated by applying rule 7. This rule embraces block t_c^6 and t_c^7 within a sequence- statement.

VI. Implementation

This section introduces a tool, namely "Maestro for Let's Dance", that enables analysts to capture and analyze Let's Dance choreographies, and to generate local models.

Figure 15 shows a screenshot of the tool. The palette on the left-hand side contains the diagram elements. The main

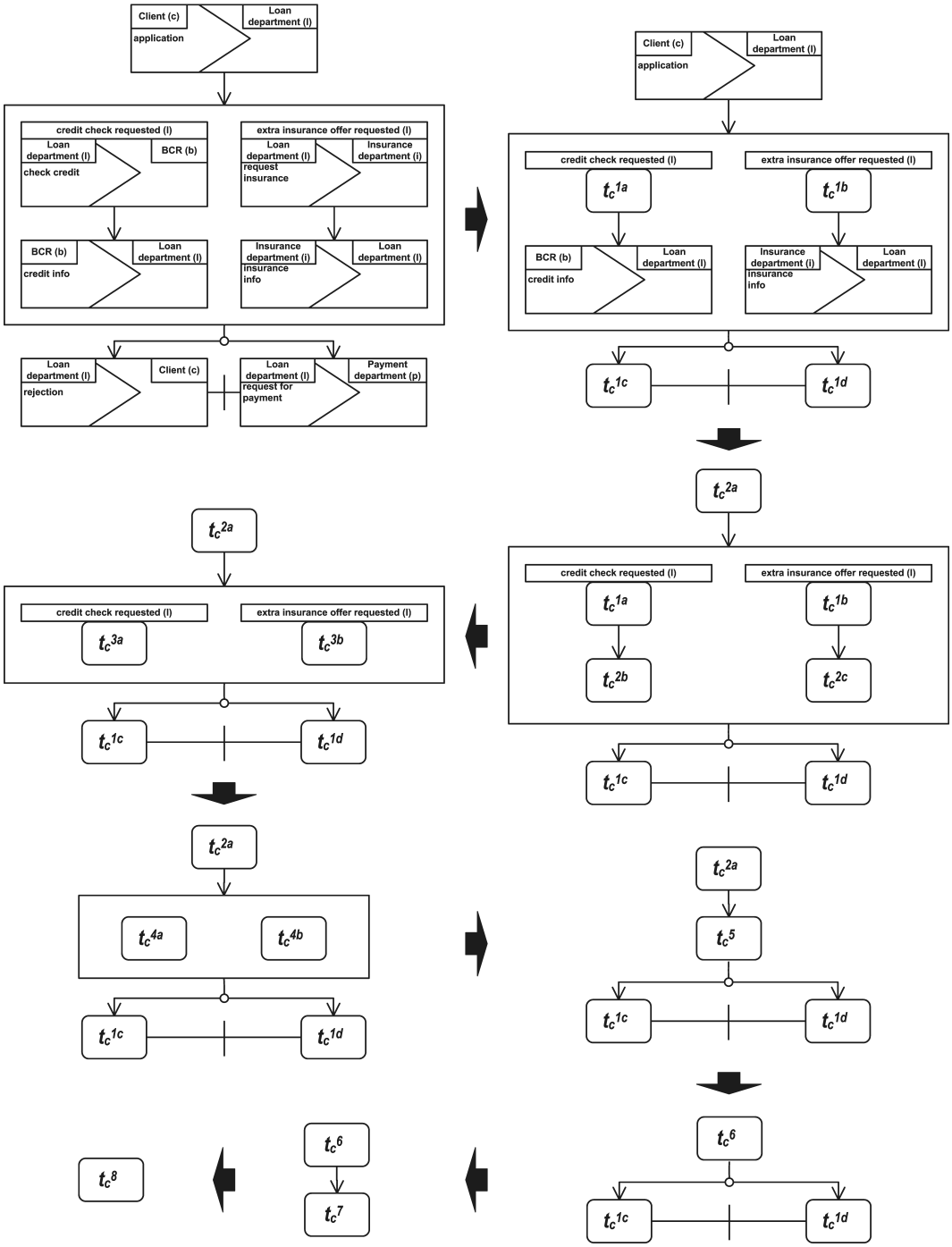

Fig. 14. Applying the mapping rules

drawing area is in the middle. On the right, there is a pane for editing the properties of the selected element and a navigator that provides an overview of the diagram. Analysis and step-by-step animation functionality can be accessed via the menus.

The tool incorporates a cardinality checker that is able to identify how many times an interaction can occur, at least and at most, within one execution of a choreography. The cardinality checker can help modelers to detect semantical errors, such as unreachable interactions or interactions that may be skipped against the modeler's intent.

The enforceability checker identifies those relationships between interactions that are not locally enforceable. Since domain analysts are supposed to sign off on a choreography model it is undesirable to automatically introduce implicit interactions to ensure the fulfilment of constraints. Instead, the modeler is warned of such issues, so that (s)he can refine the model as needed. The exact relationships that cause the enforceability property to be violated are highlighted using red lines. This way, the modeler can modify these relationships or add additional interactions to address the enforceability issues.

In order to give choreography modelers a better idea of the semantics of their models, the tool offers the possibility to

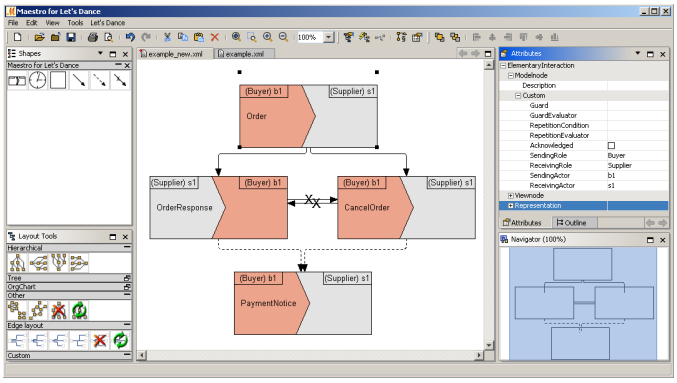

Fig. 15. Screenshot of the Let's Dance tool

animate choreography instances step-by-step and to observe intermediate states. In a given state of a choreography execution, there can be multiple enabled interactions. These enabled interactions are highlighted in green by the tool so that the modeler can choose which interaction should execute next and/or which ones should be skipped.

The tool can also generate the local models for a given choreography. For each actor a new diagram containing all interactions that involve this actor is generated and displayed in a separate tab. The generated local models are only valid if the original choreography is locally enforceable.

The prototype's functionality has been tested by coding all 14 "order management" choreographies defined by the xCBL consortium [19] as well as examples corresponding to the Service Interaction Patterns [3].

VII. Related Work

The description of global service interaction models (also called choreographies) has been the subject of intensive research and standardization. This had led to various special-purpose languages such as WS-CDL, BPSS/ebBP and EDOC (see references in Section I). In contrast, the issue of local enforceability of global models has received little attention. Existing formalizations of WS-CDL (e.g. [20]) skirt the issue of local enforceability of choreographies. Instead, they assume the existence of a state (i.e. a set of variables) shared by all participants. Participants synchronize with one another to maintain the shared state up-to-date. This effectively means that some interactions take place between services to synchronize their local view on the shared state, and at least some of these interactions are not explicitly defined in the choreography. In the worst case, this could lead to situations where a business analyst signs off on a choreography, and later it turns out that in order to properly execute this choreography a service provided by one organization must interact with a service provided by a competitor, unknowingly of the business analyst. A similar approach is followed in [5], where the authors formalize relationships between choreographies and orchestrations (i.e. local models). Meanwhile, in defining a translation from WS-CDL to BPEL, [13] adopts a different approach: control dependencies between interactions that can not be enforced in the local models are simply ignored. In this paper, we argue that neither approach (introducing hidden interactions or ignoring unenforceable dependencies) is satisfactory. Instead, we propose tool support as an approach to detect unenforceability of choreographies.

Let's Dance draws upon previous work in the area of workflow and architecture description languages. In particular, the "weak precedes" construct is inspired from (though not equivalent to) the "weak sync" construct defined in the ADEPT workflow definition language [17]. Meanwhile, the "inhibits" construct is inspired from the "disabling" construct of the Interactive Systems Description Language (ISDL): an architecture description language that supports the specification of behavioral dependencies between component interactions. A discussion on the application of ISDL for service choreography modeling is presented in [15]. However, the suitability of ISDL for capturing complex service interactions (e.g. involving multicast) is unproven.

Another family of languages that have been proposed for capturing service interactions are the so-called "semantic web service" description languages. This family of languages includes OWL-S [12] and WSMO [16]. In these languages, a service is described as a set of facts and logic rules covering three aspects: capability, non-functional properties, and interface. Interface descriptions are akin to local models in Let's Dance. However, while semantic web service descriptions are suitable in view of applying automated reasoning techniques (e.g. automated planning), their suitability for domain analysis and systems design is questionable. Domain analysts do not typically describe services down to the level of details required for non-trivial automated reasoning.

In [4], the authors consider the use of state machines to capture local models of service interactions. While state machines lead to simple models for sequential scenarios, they may lead to spaghetti-like models when used to capture scenarios with parallelism and cancellation. Nonetheless, state machines are suitable for reasoning about service models. Hull & Su [9] survey a number of approaches to capturing service interactions using communicating state machines. None of the proposals covered by this survey addresses the issue of local enforceability of global models. Instead, service interactions are described as collections of interconnected local models.

Foster et al [8] suggest the use of Message Sequence Charts (MSCs) for describing global service interaction models. These global models are converted into local models expressed as FSMs for analysis purposes. It should be noted that MSCs are a notation for describing behavior scenarios as opposed to full behavior specifications. In particular, basic MSCs do not allow one to capture conditional branches, parallel branches, and iterations. Extensions to MSCs to capture complex behavior have been defined, but in realistic cases they lead to cluttered diagrams since MSCs are based on lifelines which are meant to capture sequencing rather than branching.

Another stream of related work concerns the decentralized execution of service-oriented processes [2], [11]. A service-oriented process in this context is a set of interrelated tasks whose execution involves multiple services. Service-oriented processes can be specified for example as executable BPEL processes (cf. [11]) or as UML activity diagrams (cf. [2]).

In state-of-the-art service-oriented middleware, such as those based on BPEL, a centralized execution engine coordinates the interactions between all services involved in a process. This engine is responsible for interacting with each service, sending requests and receiving replies as dictated by the process model. Meanwhile, in decentralized execution approaches such as the ones cited above, the services involved in a process are themselves responsible for routing the flow of control. For example, one service performs the first task in the process and it then directly hands over control to (say) two other services that execute the next tasks, and so on. This decentralization has advantages in situations where it is undesirable to hand over full control to a single entity, such as in mobile environments [2]. The paradigm however differs from the global modeling approach put forward in this paper. Conceptually, service-oriented processes are modeled from the perspective of one entity, the provider of the overall process, who receives requests for executing the process and delegates the execution of tasks to the "component services". In contrast, in service choreographies, no actor plays a dominant role over the others.

VIII. Conclusion

This paper has put forward the issue of local enforceability of global service interaction models (choreographies). An algorithm is proposed that analyzes the relationships between interactions defined in a choreography and identifies those that are not enforceable. A second algorithm serves the purpose of generating local models for each actor participating in a given choreography. These local models can be used as a basis for generating behavioral interfaces and executable service descriptions. We have shown how local models can be mapped to BPEL. In separate work [7] we have formalized the control-flow constructs of the language using π-calculus.

We have implemented a toolset that supports the modeling of Let's Dance choreographies, and their analysis. Future work will aim at further validating the proposed language by designing and implementing the proposed model transformations for generating code templates in BPEL from local models. Validation on large-scale choreographies will also be sought. Another aspect worth investigation is the incorporation of transactional, and especially, atomicity aspects into the Let's Dance language. Indeed, faults occuring during the execution of a choreography may require that the effects of previous interactions be rolled back. The Let's Dance language could be extended with dedicated constructs that would help choreography designers to capture such rollbacks.

References

[1] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana: *Business Process Execution Language for Web Services, version 1.1*, May 2003. Available at: `http://www-106.ibm.com/developerworks/webservices/library/ws-bpel`

[2] L. Baresi, A. Maurino, S. Modafferi. "Workflow Partitioning in Mobile Information Systems". *Proceedings of the IFIP TC8 Working Conference on Mobile Information Systems (MOBIS)*, Oslo, Norway, September 2004. Springer, pp. 93–106.

[3] A. Barros, M. Dumas, and A. H.M. ter Hofstede. "Service Interactions Patterns. *Proceedings of the International Conference on Business Process Management (BPM)*, Nancy, France, September 2005. Springer, pp. 302–218.

[4] B. Benatallah, F. Casati, F. Toumani, and R. Hamadi. "Conceptual Modelling of Web Service Conversations". *Proceedings of International Conference on Advanced Information Systems Engineering (CAiSE)*, Velden, Austria, June 2003. Springer, pp. 449–467.

[5] N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, G. Zavattaro. "Choreography and Orchestration Conformance for System Design". *Proceedings of International Conference on Coordination Models and Languages (COORDINATION)*, Bologna, Italy, June 2006. Springer, pp. 63–81.

[6] J. Clark, C. Casanave, K. Kanaskie, B. Harvey, N. Smith, J. Yunker, K. Riemer (Eds). *ebXML Business Process Specification Schema Version 1.01*, UN/CEFACT and OASIS Specification, May 2001. `http://www.ebxml.org/specs/ebBPSS.pdf`.

[7] G. Decker, J.M. Zaha, M. Dumas. "Execution Semantics for Service Choreographies". *Proceedings of 3rd International Workshop on Web Services and Formal Methods (WS-FM)*, Vienna, Austria, September 2006. Springer, pp. 163–177.

[8] H. Foster, S. Uchitel, J. Magee, J. Kramer. "Tool Support for Model-Based Engineering of Web Service Compositions". *Proceedings of the IEEE International Conference on Web Servies (ICWS)*, Orlando FL, USA, July 2005. IEEE Computer Society, pp. 95–102.

[9] R. Hull, J. Su. "Tools for composite web services: a short overview". *SIGMOD Record* 34(2): 86-95, 2005.

[10] N. Kavantzas, D. Burdett, G. Ritzinger, and Y. Lafon. *Web Services Choreography Description Language Version 1.0*, W3C Candidate Recommendation, November 2005. `http://www.w3.org/TR/ws-cdl-10`.

[11] R. Khalaf, F. Leymann. "Role-Based Decomposition of Business Processes using BPEL". *Proceedings of the International Conference on Web Services (ICWS)*, Chicago IL, USA, September 2006. IEEE Computer Society, pp. 770–780.

[12] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, N. Srinivasan, K. Sycara. Bringing Semantics to Web Services: The OWL-S Approach. In *Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition (SWSWPC 2004)*, San Diego, California, USA, July 2004. Springer, pp. 26–42.

[13] J. Mendling, M. Hafner. "From Inter-organizational Workflows to Process Execution: Generating BPEL from WS-CDL". *Proceedings of the OTM Workshops*, Agia Napa, Cyprus, November 2005. Springer, pp. 506–515.

[14] Object Mangement Group (OMG): *UML Profile for EDOC*. `http://www.omg.org/technology/documents/formal/edoc.htm`

[15] D.A.C. Quartel, R.M. Dijkman, M. van Sinderen: "Methodological support for service-oriented design with ISDL". *Proceedings of the 2nd Intenational Conference on Service-Oriented Computing (ICSOC)*, New York NY, USA, November 2004. Springer Verlag, pp 1–10.

[16] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. "Web Service Modeling Ontology". *Applied Ontology* 1(1):77–106, 2005.

[17] M. Reichert, P. Dadam. "ADEPTflex – Supporting Dynamic Changes of Workflows Without Losing Control". *Journal of Intelligent Information Systems* 10(2): 93-129, 1998.

[18] S. White. *Business Process Modeling Notation (BPMN) – Version 1.0*, May 3 2004, `http://www.bpmi.org`.

[19] xCBL Consortium. *Order Management Choreographies*, June 2003. `http://www.xcbl.org/xcbl40/documentation.shtml`.

[20] H. Yang, X. Zhao, Z. Qiu, G. Pu , and S. Wang. *A Formal Model for Web Service Choreography Description Language*. Preprint, School of Mathematical Sciences, Peking University, January 2006 `http://www.math.pku.edu.cn:8000/var/preprint/7021.pdf`

[21] J.M. Zaha, A. Barros, M. Dumas, A. ter Hofstede "Let's Dance: A Language for Service Behavior Modeling". *Proceedings of the Fourteenth International Conference on Cooperative Information Systems (CoopIS)*, Montpellier, France, October 2006. Springer, pp. 145–162.

[22] J.M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, G. Decker. "Service Interaction Modeling: Bridging Global and Local Views". *Proceedings of the Tenth International Conference on Enterprise Distributed Object Computing (EDOC)*, Hong Kong, China, October 2006. IEEE Computer Society, pp. 45–65.