Execution Semantics for Service Choreographies

Gero Decker[1], Johannes Maria Zaha[2], and Marlon Dumas[2]

[1] SAP Research Centre, Brisbane, Australia
g.decker@sap.com
[2] Queensland University of Technology, Brisbane, Australia
(j.zaha,m.dumas)@qut.edu.au

Abstract. A service choreography is a model of interactions in which a set of services engage to achieve a goal, seen from the perspective of an ideal observer that records all messages exchanged between these services. Choreographies have been put forward as a starting point for building service-oriented systems since they provide a global picture of the system's behavior. In previous work we presented a language for service choreography modeling targeting the early phases of the development lifecycle. This paper provides an execution semantics for this language in terms of a mapping to π-calculus. This formal semantics provides a basis for analyzing choreographies. The paper reports on experiences using the semantics to detect unreachable interactions.

1 Introduction

A trend can be observed in the area of service-oriented architectures towards increased emphasis on capturing behavioral dependencies between service interactions. This trend is evidenced by the emergence of languages such as the Business Process Execution Language for Web Services (BPEL) [1] and the Web Service Choreography Description Language (WS-CDL) [7].

There are two complementary approaches to capture service interaction behavior: one where interactions are seen from the perspective of each participating service, and the other where they are seen from a global perspective. This leads to two types of models: In a *global model* (also called a *choreography*) interactions are described from the viewpoint of an ideal observer who oversees all interactions between a set of services. Meanwhile, a *local model* captures only those interactions that directly involve a given service. Local models are suitable for implementing individual services while choreographies are useful during the early phases of system analysis and design.

This paper reports on ongoing work aimed at bridging these two viewpoints by defining a service interaction modeling language (namely *Let's Dance*) as well as techniques for analyzing and relating global and local models of service interactions. In previous work [14], we defined this language informally. This paper introduces a formal execution semantics for the language using π-calculus and discusses the analysis of models using this semantics.

The next section gives an overview of the Let's Dance language. The semantics and an example are given in Section 3 while Section 4 discusses the analysis of choreographies. In Section 5 related work is presented and section 6 concludes.

2 Language overview

2.1 Language Constructs

A choreography is 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). Communication actions are represented by non-regular pentagons (symbol ▭▷ for send and ▷▭ for receive) that are juxtaposed to form a rectangle denoting an elementary interaction. A communication action is performed by an actor playing a role. The role is indicated in the top corner of a communication action. Role names are written in uppercase while the actor playing this role (or more specifically: the "actor reference") is written in lowercase between brackets.

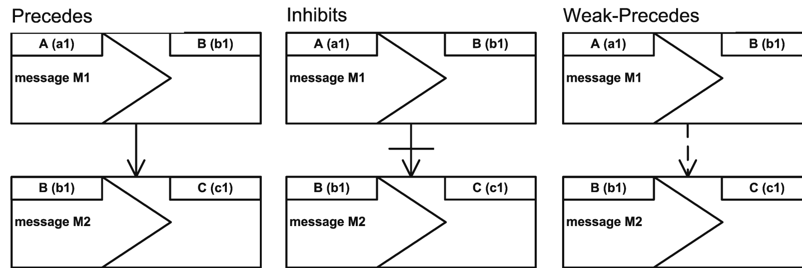

Fig. 1. Relationships in Let's Dance

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 middle relationship is called "inhibits", depicted by a crossed directed edge. It denotes that after the source interaction has taken place, the target interaction can no longer take place. That is, after "B" has received a message "M1" from "A", it may not send a message "M2" to "C". Finally, the relationship on the right-hand side, 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").

Interactions can be grouped into composite interactions as shown on the left-hand side of Figure 2. Composite interactions can be related with other interactions through precedes, inhibits and weak-precedes relationships. A composite

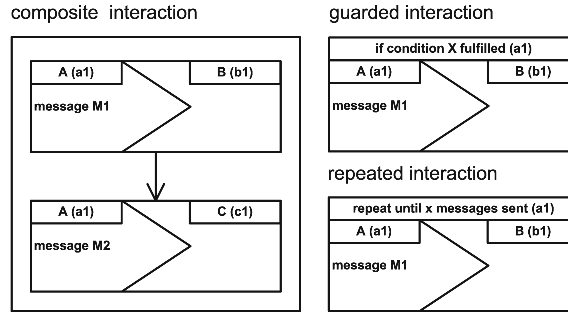

Fig. 2. Constructs of Let's Dance

interaction is completed if all sub-interactions have been executed or inhibited. The composite interaction in Figure 2 is completed if "A" has exchanged a message "M1" with "B" and a message "M2" with "C", since there is no way for the elementary interactions in question to be inhibited. The upper-right corner of Figure 2 shows a guard attached to an elementary interaction: The respective interaction is only executed if the guard evaluates to true. The actor evaluating the guard is named between brackets next to the guard. The last construct is depicted in the lower-right corner of Figure 2. It corresponds to the repetition of an interaction. Repetitions can be of type "while", "repeat until" or "for each" (the example shown in the figure is a "repeat until"). Repetitions of type "for each" have an associated "repetition expression" which determines the collection over which the repetition is performed. A repeated interaction (regardless of its type) has an associated stop condition. The actor responsible for evaluating the stop condition (and the repetition expression if applicable) is designated between brackets. Let's Dance does not impose a language for writing guards, stop conditions or repetition expressions. In this paper, we treat these as free-text.

2.2 Example

Figure 3 shows a simple order management choreography involving an actor "b1" playing the role "Buyer" and an actor "s1" playing the role "Supplier". Each interaction has a label assigned to it for identification purposes (e.g. "P" for exchanging message "PaymentNotice" in the example). The first interaction to be enabled is "O", whereby a supplier receives a message from a seller (and thus these actor references are bound to specific actors). Following this interaction, two elementary interactions ("OR" and "CO") are enabled: one where the buyer receives a number of "Order Responses" from the supplier, and another where the buyer receives a "Cancel Order" message from the supplier.

Interaction "OR" has an associated stop condition which is evaluated by actor "s1" (the supplier). This repeated interaction is of type "repeat ... until" and it completes once the supplier has no more "Order Response" messages to send (i.e. once all the line items in the purchase order have been processed). If all order responses are exchanged before a "Cancel Order" message materializes, interac-

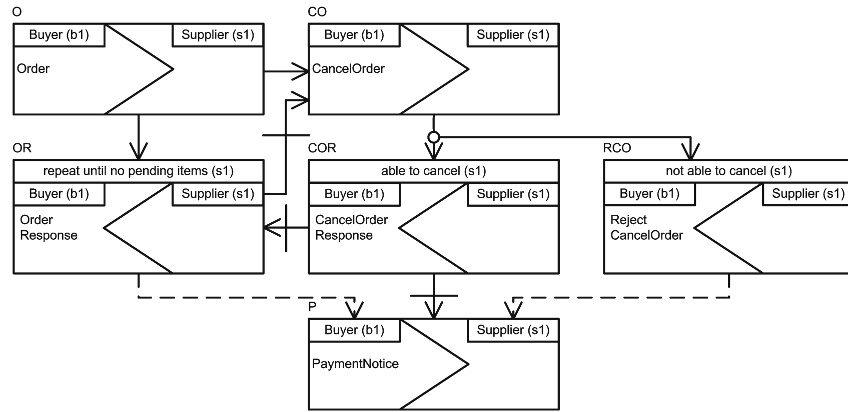

Fig. 3. Order Management Scenario

tion “CO” is inhibited. This entails that any interaction that follows it in the “Precedes” graph can no longer be performed. If on the other hand the “Cancel Order” message materializes while “Order Response” messages are still being exchanged, the supplier may either reject or accept the cancellation request. In case of acceptance, a “Cancel Order Response” is exchanged and all other potentially active interactions are inhibited (namely “OR” and “P”). If the cancellation’s is rejected, the supplier notifies it to the buyer (interaction “RCO”) and all remaining interactions are allowed to complete. The choreography (instance) completes normally after the buyer and the supplier have exchanged a payment notice (interaction “P”).

3 Formalization

3.1 Abstract Syntax

A *choreography* is a tuple (I, RI, RT, GI, A, c_0, *Precedes*, *WeakPrecedes*, *Inhibits*, *Parent*, *Performs*, *Evaluates*, *Executes*) such that:

- I is a set of Interactions
- $RI \subseteq$ I is a set of Repeated Interactions
- A function RT: RI $\rightarrow$ {w, r, fs, fc} linking repeated interactions to a repetition type, which is either While, Repeat Until, For-each Sequential or For-each Concurrent
- $GI \subseteq$ I is a set of Guarded Interactions
- A is a set of Actors
- $c_0 \in$ I is the top-level interaction of the choreography
- *Precedes*, *WeakPrecedes*, *Inhibits* $\subseteq$ I $\times$ I are three binary relations over the set of interactions I.
- *Parent* $\subseteq$ I $\times$ I is the relation between interactions and their sub-interactions.
- A function *Performs*: I $\rightarrow \wp$(A) linking interactions to actors

- A function *Evaluates*: GI $\rightarrow \wp$(A) linking guarded interactions to actors
- A function *Executes*: RI $\rightarrow \wp$(A) linking repeated interactions to actors

Not captured in the above definition are the notions of "conditional" and "repetition" expressions since these can be abstracted away when formalizing the control-flow semantics of the language. However, it is useful to have these in mind to understand certain choices in the semantics. Each guarded interaction is associated to a conditional expression (i.e. a boolean function) that determines whether the interaction is performed or not. In the abstract syntax, we only capture the actor responsible for evaluating this conditional expression (function *Evaluates*) and not the expression itself. Likewise, every repeated interaction is associated with a conditional expression (called the "stop condition") that when evaluated to true implies that the iteration must stop (in the case of "repeat" and "for each") or must continue (in the case "while"). Again, the abstract syntax only captures the actor responsible for evaluating this expression (function *Executes*). Finally, "for each" repeated interactions have a "repetition expression" attached to it that, at runtime, is used to compute the ordered collection over which the iteration is performed. The actor responsible for evaluating the "repetition expression" is the same that evaluates the "stop condition".

The constraints below are assumed to be satisfied by any Let's Dance model.

- Each interaction has one and only one parent: $\forall i \in I \mid \exists! j \in I[j \ Parent \ i]$
- No relation crosses the boundary of a repeated (composite) interaction: $\forall i, j \in I \ \forall k \in RI[(k \ Ancestor \ i \wedge (i \ Precedes \ j \vee i \ WeakPrecedes \ j \vee i \ Inhibits \ j)) \rightarrow k \ Ancestor \ j \vee k = j]$ (where $Ancestor = Parent^+$).

3.2 Background on π-calculus

The π-calculus is a process algebra for mobile systems [9]. In π-calculus, communication takes place between different π-processes. Names are a central concept in π-calculus. Links between processes as well as messages are names. This allows for link passing from one process to another. The scope of a name can be restricted to a set of processes but may be extruded as soon as the name is passed to other processes.

We will use the following syntax throughout the paper:

$$\begin{aligned} P &::= M \mid P|P' \mid (\nu\ z)P \mid !P \\ M &::= 0 \mid \pi.P \mid M + M' \\ \pi &::= \overline{x}\langle y \rangle \mid \overline{x} \mid x(y) \mid x \mid \tau \end{aligned}$$

Concurrent execution is denoted as $P|P'$, the restriction of the scope of z to P as $(\nu\ z)P$ and an infinite number of concurrent copies of P as $!P$. Inaction of a process is denoted as 0. A non-deterministic choice between M and M' as $M + M'$, sending y over x as $\overline{x}\langle y \rangle$, sending an empty message over x as $\overline{x}$ and receiving an empty message over x as x. The prefix $x(y)$ receives a name over x and continues as P with y replaced by the received name. τ is the unobservable

action. Communication between two processes can take place in the case of matching send- and receive-prefixes. Furthermore, we denote the parallel and sequential execution of the prefixes π_i, $i \in I$ as $\Pi_{i \in I}\ \pi_i$ and $\{\pi_i\}_{i \in I}$, respectively. To restrict the scope of the set of names z_i, $i \in I$ we use the abbreviation $[z_i]_{i \in I}$.

3.3 Formalization

We chose π-calculus for the formalization of Let's Dance since it has proved to be a suitable formalism for describing interactions in a service-oriented environment (cf. the formalization of the Service Interaction Patterns [8]). Although we do not exploit the full power of π-calculus in this paper, we are dependent on the concept of name passing as soon as correlation issues and actor reference passing find their way into the formalization. Also, conformance between global and local models – a central issue for choreographies in practice – calls for advanced reasoning techniques such as π-calculus' weak open bi-simulation.

To improve understandability, we decompose the formalization of an interaction into four levels covering different aspects as depicted in figure 4.

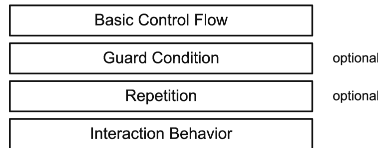

Fig. 4. Formalization levels for interactions

- *Basic Control Flow* covers the coordination between different interactions. The three different relationship types $Precedes$, $WeakPrecedes$ and $Inhibits$ and the notion of propagating skipping are formalized here.
- *Guard Condition* formalizes the possibility to skip an enabled interaction instance if a guard condition evaluates to false. Since evaluating the conditions themselves is not formalized, we introduced a non-deterministic choice. *Guard Condition* only applies to guarded interactions.
- *Repetition* covers the repetition types "while", "repeat", "for each (sequential)" and "for each (concurrent)". It only applies to repeated interactions.
- *Interaction Behavior* contains the formalization for elementary interactions and composite interactions. In the case of composite interactions enabling and skipping sub-interactions are formalized in this layer.

A π-process is introduced for each of these levels and for each interaction in a choreography. Communication between π-processes realizes the coordination between different interactions as well as between the different layers of each interaction. For inter-level-communication we introduce the private links *enable*, *complete* and *skip*. Figure 5 illustrates how these private links are used.

Sending a message over *enable* indicates that the interaction instance is enabled. Sending a message over *complete* back indicates that the interaction has executed successfully. *skip* is used to propagate skipping to sub-level-processes.

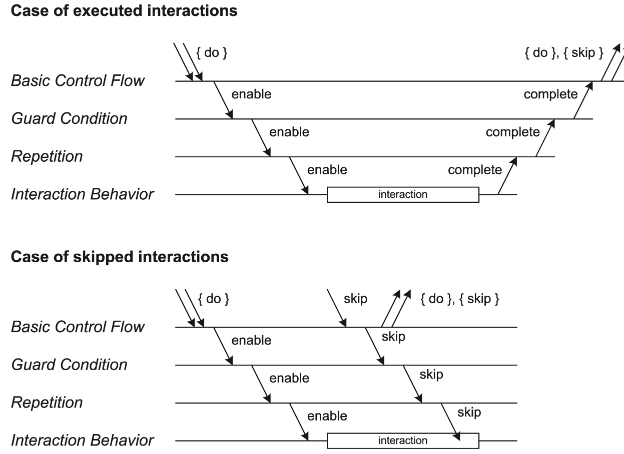

Fig. 5. Inter level communication

Formalization of Basic Control Flow. For every $A \in I$ the formalization of the corresponding interaction instances is:

$$
\begin{aligned}
A = (\nu\ perform, enable, complete, skip)(\{do_A\}_{i \in P} \ .\ \overline{perform} \quad && (1)\\
\mid (perform \ .\ (\overline{enable} \ .\ (complete \ .\ A_{completed} \quad && (2)\\
+skip_A \ .\ (A_{skipped} \mid !skip_A))) \quad && (3)\\
+skip_A \ .\ (perform \ .\ A_{skipped} \mid !skip_A)) \quad && (4)\\
\mid InnerProc(enable, complete, skip)) \quad && (5)\\
A_{completed} = \{\overline{skip_i}\}_{i \in Q} \ .\ (\overline{done_{Parent(A)}} \mid \Pi_{i \in R}\ \overline{do_i}) \mid !skip_A \quad && (6)\\
A_{skipped} = \overline{skip} \ .\ \{\overline{skip_i}\}_{i \in S} \ .\ (\overline{done_{Parent(A)}} \mid \Pi_{i \in R}\ \overline{do_i}) \quad && (7)
\end{aligned}
$$

$$
\begin{aligned}
\textbf{where}\ P &= \{x \in I \mid x\ Precedes\ A \vee x\ WeakPrecedes\ A \vee x = Parent(A)\}\\
Q &= \{x \in I \mid A\ Inhibits\ x\}\\
R &= \{x \in I \mid A\ Precedes\ x \vee A\ WeakPrecedes\ x\}\\
S &= \{x \in I \mid A\ Precedes\ x\}
\end{aligned}
$$

$$
InnerProc = \begin{cases} Guard_A & \textbf{if } A \in GI \\ NoGuard_A & \textbf{if } A \notin GI \end{cases}
$$

The names do_A and $skip_A$ are introduced for the coordination between A and all interactions that are the source of a relation where A is the target:

- *Precedes:* If the source interaction has completed an empty message is sent over do_A. If the source interaction was skipped then first a message is sent over $skip_A$ and then another message over do_A. This order is crucial for ensuring that first skipping is propagated before enabling takes place.
- *WeakPrecedes:* A message is sent over do_A if the source interaction has completed or was skipped.
- *Inhibits:* A message is sent over $skip_A$ if the source interaction has completed.

For every $Precedes$ and $WeakPrecedes$ relation a message over do_A has to arrive before anything else can happen inside the interaction instance. That is why the private name *perform* was introduced (lines 1, 2, 4). Even if a message over $skip_A$ arrives before all messages over do_A have arrived (line 4) the process has to wait for the remaining messages before sending a message over *perform*.

When all do_A-messages arrive before a $skip_A$-message, the interaction instance is enabled and an empty message is sent over *enable* to the process of the layer below (line 2). Once the interaction instance is enabled, the instance either completes (a *complete*-message is received) or a $skip_A$-message arrives. The latter causes the instance to be skipped immediately without waiting for the completion of the execution. In the first case, i.e. the instance completes, the follow-up actions in $A_{completed}$ apply which consist of first sending $skip_A$-messages to all target interactions of outgoing $Inhibits$-relations. Then *do*-messages are sent to all target interactions of outgoing $Precedes$- and $WeakPrecedes$-relations. *done*-messages will be explained in the section “Interaction behavior”.

In the case where a $skip_A$-message arrives before all do_A-messages have arrived, the alternative in line 4 is chosen. After the *perform*-message has arrived (i.e. that all do_A-messages have arrived) the follow-up actions in $A_{skipped}$ apply. After skipping is propagated to the lower levels, *skip*-messages are sent to all target interactions of $Precedes$-relations. Finally, *do*- and *done*-messages are sent like it was already the case in $A_{skipped}$.

$!skip_A$ serves as a “garbage collector” for $skip_A$-messages that arrive without causing any effect: After the instance has already completed (line 6) or after a $skip_A$-message has already caused skipping the instance (lines 3, 4).

Example. Interaction OR from Figure 3 is not guarded and has one incoming $Precedes$-relation, one incoming $Inhibits$-relation, one outgoing $Inhibits$-relation and one outgoing $WeakPrecedes$-relation which leads to the following π-processes:

$$\begin{aligned}
OR = {} & !(\nu\ perform, enable, complete, skip)(do_{OR} \,.\, \overline{perform} \\
& \mid (perform \,.\, (\overline{enable} \,.\, (complete \,.\, OR_{completed} \\
& \qquad\qquad + skip_{OR} \,.\, (OR_{skipped} \mid !skip_{OR}))) \\
& \quad + skip_{OR} \,.\, (perform \,.\, OR_{skipped} \mid !skip_{OR})) \\
& \mid NoGuard_{OR}(enable, complete, skip)) \\
OR_{completed} = {} & \overline{skip_{CO}} \,.\, \overline{do_P} \mid !skip_{OR} \\
OR_{skipped} = {} & \overline{skip} \,.\, \overline{do_P}
\end{aligned}$$

Interaction instance lifecycle. When observing the communication between the *Basic-Control-Flow*-layer-process and the process of the level below we can easily identify the state an interaction instance is in. Figure 6 depicts the life cycle of an interaction instance.

Each interaction instance starts in the state *initialized*. Now a message over either *enable* or *skip* can be sent. In the case of *skip* the interaction instance is skipped and cannot execute any more. In the case of *enable* a message over

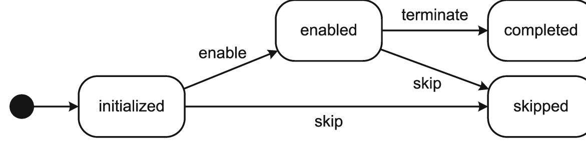

Fig. 6. Interaction instance life cycle

complete or *skip* can be sent. As already mentioned *complete* indicates that the interaction has executed successfully. Therefore, the instance changes to the state *completed*. A message over *skip* results in skipping the instance.

Formalization of Guard Conditions.

$$Guard_A(e, c, s) = (\nu\ enable, complete, skip)(s\ .\ \overline{skip} \tag{1}$$

$$|\ (e\ .\ (\tau_0\ .\ \overline{skip_A} + \tau_0\ .(\overline{enable}\ .\ complete\ .\ \overline{c} \tag{2}$$

$$|\ InnerProc(enable, complete, skip))))) \tag{3}$$

$$NoGuard_A(e, c, s) = InnerProc(e, c, s) \tag{4}$$

$$\textbf{where} \quad InnerProc = \begin{cases} While_A & \textbf{if } A \in RI \wedge RT(A) = w \\ Repeat_A & \textbf{if } A \in RI \wedge RT(A) = r \\ ForEachSeq_A & \textbf{if } A \in RI \wedge RT(A) = fs \\ ForEachConc_A & \textbf{if } A \in RI \wedge RT(A) = fc \\ NoRepetition_A & \textbf{if } A \notin RI \end{cases}$$

The links e, c and s are used for the communication with the *Basic Control Flow* layer. The new names *enable*, *complete* and *skip* serve as communication links to the process of the layer below.

A guard will not be evaluated until the interaction instance is enabled. That is why a message has to be received over e before the non-deterministic choice can take place (line 2). If the first alternative is chosen a *skip*-message is sent back to the *Basic-Control-Flow*-layer-process which causes the interaction instance to be skipped. If the second alternative is chosen the layer below is enabled.

Example. Interaction *COR* from Figure 3 leads to the following π-process:

$$\begin{aligned} Guard_{COR}(e, c, s) = (\nu\ enable, complete, skip)(s\ .\ \overline{skip} \\ |\ (e\ .\ (\tau_0\ .\ \overline{skip_A} + \tau_0\ .(\overline{enable}\ .\ complete\ .\ \overline{c} \\ |\ NoRepetition_{COR}(enable, complete, skip))))) \end{aligned}$$

Interaction *OR* from Figure 3 is translated as

$$NoGuard_{OR}(e, c, s) = Repeat_{OR}(e, c, s)$$

Formalization of Repetitions. "While" and "For each (sequential)" have identical semantics at the level of abstraction of control flow. In both cases the interaction instance is executed an arbitrary number of times. We assume that the repetition will terminate at some point in time. "Repeat until" repetitions have similar

semantics as “While” except that in this case the interaction instance is executed at least once. The formalization below is based on recursion.

$$\begin{aligned}
While_A(e,c,s) = ForEachSeq_A(e,c,s) &= (\nu\ enable, complete, skip)\\
&\quad (s\ .\ \overline{skip} \mid e\ .\ R)\\
R &= \tau_0\ .\ \overline{c} + \tau_0\ .\ (\overline{enable}\ .\ complete\ .\ R\\
&\qquad \mid InnerProc(enable, complete, skip))\\
Repeat_A(e,c,s) &= (\nu\ enable, complete, skip)(s\ .\ \overline{skip} \mid e\ .\\
&\quad (\overline{enable}\ .\ complete\ .\ R\\
&\quad \mid InnerProc(enable, complete, skip)))\\
R &= \tau_0\ .\ \overline{c} + \tau_0\ .\ (\overline{enable}\ .\ complete\ .\ R\\
&\quad \mid InnerProc(enable, complete, skip))
\end{aligned}$$

$$\textbf{where} \quad InnerProc = \begin{cases} Elementary_A & \textbf{if } A \notin CI \\ Composite_A & \textbf{if } A \in CI \end{cases}$$

“For each (concurrent)” is the most complex type of repetitions. All interaction instances are executed concurrently. Informally, when a repeated interaction is performed, one instance of the contained interaction is started for each element in the collection obtained from the evaluation of the repetition expression. These instances execute concurrently. Each time that one of these instances completes, the stop condition is evaluated. If the stop condition evaluates to true, the execution of the remaining instances is stopped and the execution of the repeated interaction is considered to be completed.

The formalization of the “For each (concurrent)” construct below is inspired from the π-formalization for the workflow pattern “Multiple Instances with a-priori Runtime Knowledge” given in [10]. We introduce a linked list of processes that use links c for notifying the previous process in the list that the interaction instance has completed successfully and sk to notify the next process that the instance has been skipped. Figure 7 illustrates this. There can be cases where not all instances have to completed before the repetition is considered to be completed. Arbitrary stop conditions can be defined for a repetition and after a given instance completes a non-deterministic choice either leads to waiting for *complete* or sending a message over st right away. The latter results in propagating *stop*-messages that lead to the completion of the repeated interaction. The formalization of this construct is given below. The Symbol *InnerProc* is defined as for “While” repetitions (see above).

$$\begin{aligned}
ForEachConc_A(e,c,s) &= (\nu\ comp, sk)(e\ .R(comp, comp, sk) \mid comp\ .\ \overline{c} \mid s\ .\ \overline{sk})\\
R(c,st,s) &= (\nu\ comp, stop, sk)(\tau_0\ .\ \overline{c} + \tau_0\ .\ (R(comp, stop, sk)\\
&\quad \mid (\nu\ enable, complete, skip)(s\ .\ (\overline{sk} \mid \overline{skip})\\
&\quad \mid \overline{enable}\ .\ (stop\ .\ (\overline{st} \mid \overline{skip}) + complete\ .\\
&\qquad (\tau_0\ .\ (comp.\overline{c} + stop.\overline{st}) + \tau_0\ .\ (\overline{st} \mid \overline{sk})))\\
&\quad \mid InnerProc(enable, complete, skip))))
\end{aligned}$$

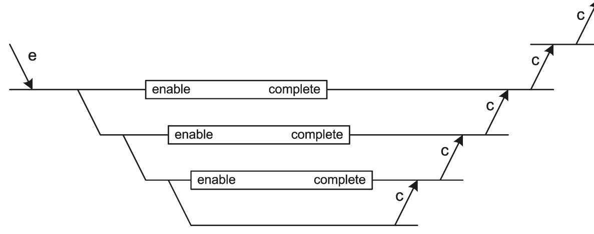

Fig. 7. Linked list of processes realizing three concurrent interaction instances

Example. π-processes for interaction OR from Figure 3:

$$\begin{aligned} Repeat_{OR}(e,c,s) &= (\nu\ enable, complete, skip)(s.\overline{skip} \mid e.(\overline{enable}.complete.R \\ &\qquad \mid Elementary_{OR}(enable, complete, skip))) \\ R &= \tau_0\ .\ \overline{c} + \tau_0\ .\ (\overline{enable}\ .\ complete\ .\ R \\ &\qquad \mid Elementary_{OR}(enable, complete, skip)) \end{aligned}$$

Formalization of Interaction Behavior. Elementary interactions are assumed to be atomic. Since we only focus on the control flow aspects of choreographies we do not incorporate an actual communication between two business partners in the π-formalization. We simply denote this interaction as $\tau_{interact}$.

$$\begin{aligned} Elementary_A(e,c,s) &= e\ .\ \tau_{interact}\ .\ \overline{c} \mid s \\ Composite_A(e,c,s) &= (\nu\ [do_i]_{i\in Q}, [done_i]_{i\in Q}, [skip_i]_{i\in Q})(\Pi_{B\in Q}\ B \\ &\qquad \mid e\ .(\Pi_{i\in P}\ \overline{do_i} \mid \{done_A\}_{i\in P}\ .\ \overline{c}) \\ &\qquad \mid s\ .\ (\Pi_{i\in P}\ \overline{skip_i}\ .\ \overline{do_i})) \end{aligned}$$

$$\begin{aligned} \textbf{where}\quad P &= \{x \in I \mid A = Parent(x)\} \\ Q &= \{x \in I \mid A \in RI \wedge A\ Ancestor\ x \\ &\qquad \wedge \neg\exists y \in RI(A\ Ancestor\ y \wedge y\ Ancestor\ x)\} \end{aligned}$$

Sub interactions of composite interactions must not execute until the parent interaction has been enabled. In the formalization of *Basic Control Flow* we have seen that *do*-messages are expected from all source interactions of *Precedes*- and *WeakPrecedes*-relations as well as from all direct parent interactions.

As soon as an interaction instance has completed or is ready for propagating skipping a *done*-message is sent to the direct parent interaction. These *done*-messages are collected in the $Composite_A$ process and as soon as all messages have arrived the *complete*-message is sent to the super-level-process. This behavior guarantees that all sub interaction instances are already in one of the states completed or skipped before further enabling and skipping takes place for outgoing relations from A.

In case of the receipt of a *skip*-message, *skip*-messages are sent to all sub interactions. If a sub interaction has already completed or has been skipped, this message does not have any effect.

If A is a repeated interaction we have to start executing the π-processes for all sub interactions at this point in time. That way we create multiple interaction instances for each sub interaction (one instance per repetition cycle). By creating new *do*- and *skip*-names we make sure that the inter-interaction-instance-coordination takes place within the same repetition cycle.

Creating new names only in the case of repeated interactions also implements the fact that $Precedes$-, $WeakPrecedes$- and $Inhibits$-relations can cross the boundaries of a composite interaction i if i is not repeated.

Example. Interaction OR from Figure 3 leads to the following formalization:

$$Elementary_{OR}(e, c, s) = e \,.\, \tau_{interact} \,.\, \overline{c} \mid s$$

Putting it all together. We have shown how the behavior of individual interaction instances can be expressed using π-calculus. It now only takes a small step to come to a π-formalization of a whole choreography C:

$$C = (\nu\ [do_i]_{i \in P}, [done_i]_{i \in P}, [skip_i]_{i \in P})\ \Pi_{A \in P}\ A$$
$$\textbf{where}\quad P = \{x \in I \mid \neg\exists y \in RI(y\ Ancestor\ x)\}$$

The π-processes for all interaction instances that are not sub interactions of repeated interactions are executed in parallel. Sub interactions of repeated interactions are executed in the $Composite_A$-process.

4 Reachability analysis for choreographies

The translation of Lets Dance choreographies into π-processes as it is shown in the previous section allows for reasoning on these choreographies. A typical means to examine π-processes is to use bi-simulation equivalence. The first definitions for bi-simulation, namely early and late bi-simulation, were introduced by Milner, Parrow and Walker ([9]). However, the most prominent definition for bi-simulation was introduced by Sangiorgi ([11]) and is called open bi-simulation. Using this bi-simulation equivalence relation $\sim_o$ we know whether two π-processes have the same transition behavior and thus simulate each other.

In the case of weak open bi-simulation the non-observable transitions are ignored. This bi-simulation definition is suitable for our purposes: We want to focus on certain interactions in our choreographies and consider everything else as non-observable to the bi-simulation analysis.

One interesting property of an interaction in a choreography is whether it is reachable (i.e. it may execute successfully) or if it is not-reachable (i.e. it never executes). If we examine an elementary interaction it is sufficient to check whether $\tau_{interact}$ may be executed. However, according to the definition given at the beginning of section 3 this action is unobservable. To change this we can replace the $\tau_{interact}$ of the interaction in question by the send-prefix $\overline{interact}$. The π-process would look like

$$Elementary_A(e, t, s) = e \,.\, \overline{interact} \,.\, \overline{t} \mid s$$

If we now define the link *interact* to be the only observable part in the choreography then we can compare it to other π-processes. E.g. a comparison to the π-process 0 tells us whether an interaction is reachable or not. If the choreography is weak open bi-simulation related to 0 the interaction in question is not-reachable otherwise the interaction must be reachable.

Doing bi-simulation analysis using tools such as the Mobility Workbench ([12]) is not possible if the choreography contains repeated interactions. The formalizations in the previous section introduce a non-deterministic choice for the stop condition of repetitions which causes the tool to run into an infinite loop. However, we can simply omit the *Repetition* layer for the reachability analysis while preserving correct results.

5 Related work

Industry-driven initiatives have attempted to standardize notations for global description of service interactions. An early attempt was BPSS [5] where global models are captured as flows of interactions using flowchart-like constructs. WSCI [2] represents another approach wherein global service interaction models are defined as collections of inter-connected local models. Control dependencies are described within each individual local model. A formal semantics of a subset of WSCI is sketched in [3]. More recently, the WS-CDL initiative [7] led to a language that follows the line of BPSS insofar as global service behavior is described as flows of interactions. WS-CDL goes further than BPSS in the level of details at which interaction flows are described. In fact, WS-CDL can be seen as a programming-in-the-large language for Web services: it deals with global interactions as the basic primitive but relies on imperative programming constructs such as variable assignment, sequence and block-structured choice and parallel branching. Several formal semantics of WS-CDL or subsets thereof have been defined. Yang et al. [13] propose a small-step operational semantics of WS-CDL. It is not clear however that this semantics provides a suitable basis for reasoning about service choreographies, such as determining whether or not a local model complies to a choreography, or performing reachability analysis as discussed above. Other authors have defined subsets of WS-CDL and captured them in terms of process calculi. Busi et al. [4] define a formal language corresponding to a subset of WS-CDL and use it as a foundation to capture relationships between choreographies and local models. It is unclear though that this formalization provides a suitable basis for automated analysis of choreographies.

Unlike WS-CDL, Let's Dance does not target application developers, but rather analysts and designers. Accordingly, it avoids reliance on imperative programming constructs with which analysts are usually unfamiliar. Still, Let's Dance models can be used for simulation and verification as discussed above.

Several authors have considered the use of communicating state machines as a basis for modeling global models of service interactions [6]. While state machines lead to simple models for sequential scenarios, they usually lead to spaghetti-like models when used to capture scenarios with parallelism and cancellation. Thus,

state machines may provide a suitable foundation for reasoning about service interactions, but their suitability for choreography modeling is questionable.

6 Conclusion and outlook

This paper has introduced a formal semantics for a service interaction modeling language, namely Let's Dance, which supports the high-level capture of both global models (i.e. choreographies) and local models of service interactions. The semantics is defined by translation to π-calculus. At present, the semantics focuses on control-flow aspects. However, π-calculus is well-suited for capturing actor bindings and passing binding information across actors. Ongoing work aims at extending the current semantics along this direction.

The presented semantics has been used as a blueprint for the implementation of a simulation engine and as a basis for analyzing Let's Dance choreographies. We have shown in this paper how weak open bisimulation can be used to check reachability of interactions. Ongoing work aims at applying a similar technique to compliance checking, i.e. checking whether a local model complies to a choreography. However, when interactions in the choreography and those in the local models do not map one-to-one, or when a local model implements several choreographies, pure weak open bi-simulation approaches reach their limits.

Another problem that deserves further attention and can be addressed on the basis of the formalization is that of *local enforceability* of choreographies [15]. It turns out that not all choreographies defined as flows of interactions (the paradigm adopted in Let's Dance) can be mapped into local models that satisfy the following conditions: (i) the local models contain only interactions described in the choreography; and (ii) they collectively enforce all the constraints in the choreography. Proposals around WS-CDL skirt this issue. 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. Thus, certain interactions take place between services for the sole purpose of synchronizing their local view on the shared state and these interactions are not defined in the choreography. In the worst case, this leads to situations where a business analyst signs off on a choreography, and later it turns out that to execute this choreography a service provided by one organization must interact with a service provided by a competitor, unknowingly of the analyst. Thus, it is desirable to provide tool support to analyze choreographies to determine whether or not they are enforceable by some set of local models. In [15], we have defined an algorithm for determining local enforceability of Let's Dance choreographies. The formal semantics presented in this paper can provide a basis for validating the correctness of the transformation rules encoded in this algorithm.

Acknowledgments. The second author is funded by SAP. The third author is funded by a fellowship co-sponsored by Queensland Government and SAP.

References

1. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, S. Weerawarana: *Business Process Execution Language for Web Services, version 1.1*, May 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel
2. A. Arkin et al.: *Web Service Choreography Interface (WSCI) 1.0*, 2002. www.w3.org/TR/wsci/
3. A. Brogi, C. Canal, E. Pimentel, A. Vallecillo: *Formalizing Web Service Choreographies*. In Proceedings of 1st International Workshop on Web Services and Formal Methods, Pisa, Italy, February 2004, Elsevier.
4. N. Busi, R. Gorrieri, C. Guidi, R. Lucchi, G. Zavattaro: *Choreography and Orchestration Conformance for System Design*. In Proceedings of 8th International Conference on Coordination Models and Languages (COORDINATION'06), Bologna, Italy, June 2006, Springer Verlag.
5. 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
6. R. Hull, J. Su: *Tools for composite web services: a short overview*. SIGMOD Record 34(2): 86-95, 2005.
7. N. Kavantzas, D. Burdett, G. Ritzinger, Y. Lafon. *Web Services Choreography Description Language Version 1.0*, W3C Candidate Recommendation, November 2005. http://www.w3.org/TR/ws-cdl-10
8. G. Decker, F. Puhlmann, M. Weske. *Formalizing Service Interactions*. In Proceedings of BPM 2006, Vienna, Austria, 2006, Springer Verlag.
9. R. Milner, J. Parrow, D. Walker. *A calculus of mobile processes*, Information and Computation, 100:1–40, 1992.
10. F. Puhlmann, M. Weske: *Using the π-Calculus for Formalizing Workflow Patterns*. In Proceedings of BPM 2005, Nancy, France, September 2005, pp 153-168, Springer Verlag.
11. D. Sangiorgi: *A theory of bisimulation for the π-calculus*. In Acta Informatica 16(33): 69-97, 1996.
12. B. Victor, F. Moller, M. Dam, L.H. Eriksson: The Mobility Workbench. Uppsala University, 2006. http://www.it.uu.se/research/group/mobility/mwb
13. H. Yang, X. Zhao, Z. Qiu, G. Pu, S. Wang: *A Formal Model for Web Service Choreography Description Language (WS-CDL)* Preprint, School of Mathematical Sciences, Peking University, January 2006. www.math.pku.edu.cn:8000/var/preprint/7021.pdf
14. J. M. Zaha, A. Barros, M. Dumas, A. ter Hofstede: A Unified Language for Service Behavior Modeling. Preprint # 4468, Faculty of IT, Queensland University of Technology, February 2006. http://eprints.qut.edu.au/archive/00004468
15. J. M. Zaha, M. Dumas, A. ter Hofstede, A. Barros, G. Decker: *Service Interaction Modeling: Bridging Global and Local Views*. In Proceedings of the 10th International EDOC Conference, Hong Kong, 2006