Passive Conformance Testing of Service Choreographies

Huu Nghia Nguyen
LRI UMR 8623 CNRS,
Orsay, France;
Univ. Paris-Sud,
Orsay, France
huu-nghia.nguyen@lri.fr

Pascal Poizat
LRI UMR 8623 CNRS,
Orsay, France;
Univ. Evry Val d'Essonne,
Evry, France
pascal.poizat@lri.fr

Fatiha Zaïdi
LRI UMR 8623 CNRS,
Orsay, France;
Univ. Paris-Sud,
Orsay, France
fatiha.zaidi@lri.fr

ABSTRACT

Choreography supports the specification, with a global perspective, of the interactions between the roles played by partners in a collaboration. These roles are the basis for the implementation of the collaboration, by developers and/or software architects, as a set of distributed communicating peers. An issue is to check for the conformance of the implementation with reference to the choreography specification. We address this issue with a passive testing approach. It tackles the peculiarities of choreography implementations through non-intrusiveness, support for black-box peers without source code being available, and both local and global conformance. Several languages have been proposed for choreography. We chose *Chor* since it is both expressive and abstract enough to suit the requirements of a specification language. Further, it can be seen as an abstraction of the standard Web service choreography language, WS-CDL. In this paper we present both the formal framework of our approach and our tool support for one possible implementation model, Web service choreographies.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Monitors, Testing tools*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages, Service-oriented architecture (SOA)*

General Terms

Design, Verification

Keywords

Choreography, Web services, conformance checking, passive testing, tool.

1. INTRODUCTION

Business processes and applications are no longer monolithic but are built from the reuse and composition of other processes or software entities. This trend has increased with the emergence of component and service architectures such as Web services. A centralized point of view may be taken on collaboration, which nicely suits service orchestration. However, modern processes and applications are much more collaborative in nature. Hence, they should be specified and implemented in a collaborative way too. This is where choreography may help.

A *choreography* is the description from a global perspective of the interactions between roles played by peers (components, services, organizations, humans) in some collaboration. Several languages and notations do support choreography specification: BPMN, MSC, UML, WS-CDL to give some. Following [9], they can be classified using two dimensions. They can be abstract (*e.g.*, BPMN, MSC, UML) or concrete (*e.g.*, WS-CDL). Specification languages are used to describe *what* peers should (or should not) do in a collaboration, rather than *how* they should do it. Therefore, abstract choreography languages are better candidates for choreography specification. The second dimension [9] is related to the underlying model. In *interconnected interface models* (*e.g.*, BPMN collaboration diagrams, MSC, UML sequence diagrams, Reo), conversations are defined at (each) peer level and interactions are defined by roughly connecting conversations.

To the contrary, in *interaction models* (*e.g.*, BPMN 2.0 new choreography diagrams, UML collaboration diagrams), interactions between peers are the basic building blocks. From a designer perspective, interaction models better suit the needs of choreography specification due to their global perspective. In this paper, we chose the *Chor* [17] language for the specification of choreographies. While being simpler than more general purpose languages such as BPMN or UML, it is expressive and abstract enough to enable one to specify collaborations. *Chor* is based on an interaction model and can also be seen as an abstraction of the WS-CDL standard. Further, *Chor* defines both a choreography language, a role language, and projections between global and local (role) descriptions supporting the formal treatment of collaborations at both viewpoints.

An issue with choreography is the relation with implementation. Peers may be written from scratch or be reused, and are then coordinated to fulfil the choreography. This relates to automatic service composition and orchestration [13]. An alternative is the generation of peer skeletons from choreographies, *e.g.*, [17, 12]. Skeletons are then completed by developers to build a running system. In any case, an important issue, so-called conformance, is to check whether the implementation exhibits or not the behaviours specified in the choreography. When peer code is available, or when behavioural interfaces of the peers are provided, verification using model-checking or behavioural equivalences is an alternative. In such a case, formal testing is of great help. It enables one to check whether an implementation conforms or not to a specification without requiring an access to its code.

In this paper we propose a formal framework for testing whether an implementation conforms or not to a choreography specification. Conformance is checked both at the local and at the global level. Local conformance represents whether some peer implementation plays or not correctly its role. Global conformance ensures that interactions between peers follow the prescription of the choreography or if they diverge from the envisioned collaboration. Our approach is totally tool supported and demonstrated on a medium-size case study.

The rest of this paper is organized as follows. We give preliminaries on the *Chor* language in Section 2. Conformance is discussed and formalized in Section 3. Our testing architecture is introduced in Section 4 and Section 5 presents its implementation and our tools. Section 6 discusses related work. We end with conclusions and perspectives in Section 7.

2. THE CHOR LANGUAGE

Chor includes basic and structuring activities. *skip* denotes a do-nothing action, while $c^{[i,j]}$ represents a basic interaction on some medium c between two roles of the choreography, namely i and j, called performers of the interaction. Since *Chor* is concerned about the abstract specification of the collaboration, the instantiation of some interaction medium, and details associated to it (*e.g.*, exchanged data) is part of the developer duties. This is can be, *e.g.*, using message and message parts in a Web service framework. We restrict to the observable fragment of *Chor*, *i.e.*, we do not take into account the specification at the global level of local non-observable actions. Structuring in *Chor* is achieved using sequencing (;), exclusive choice ($\sqcap$) and parallel flows ($\|$).

Role requirements are described in a dialect of *Chor* called role languages (*Role* for short) with the only difference that a global interaction $c^{[i,j]}$ corresponds in role i (resp. j) to an emission denoted $c^{[i,j]}!$ (resp. reception denoted $c^{[i,j]}?$). The semantics of a *Chor* (local or global) specification C is given in terms of its set of all specification traces, *trace set* for short, that represent all possible run of the specification [17]. In the sequel, $^\frown$ denotes trace concatenation and $\bowtie$ denotes trace interleaving. Further in a trace, $\boxtimes$ denotes a deadlock (*i.e.*, a blocking termination).

The requirements for each role of a choreography can be obtained using projection (*nproj*) hiding the interactions that do not concern the role of interest and orienting the other ones; *i.e.*, for a role k, the projection of $c^{[i,j]}$ gives $c^{[i,j]}!$ if $k = i$, $c^{[i,j]}?$ if $k = j$, and *skip* otherwise.

Example 1. Let us take the example of a collaboration involving two roles, a client (R_1) and a travel agency (R_2). The client first issues a request to the agency (c_1) which then gives back train information (c_2) and plane information (c_3). This is modelled in *Chor* as the specification $C_1 = c_1{}^{[1,2]}; (c_2{}^{[2,1]} \parallel c_3{}^{[2,1]})$, whose semantics, $[\![C_1]\!]$, is:

$$\begin{aligned}
[\![C_1]\!] &= [\![c_1{}^{[1,2]}]\!]^\frown([\![c_2{}^{[2,1]}]\!] \bowtie [\![c_3{}^{[2,1]}]\!]) \\
&= \{\langle c_1{}^{[1,2]}\rangle\}^\frown\{\langle c_2{}^{[2,1]}, c_3{}^{[2,1]}\rangle, \langle c_3{}^{[2,1]}, c_2{}^{[2,1]}\rangle\} \\
&= \{\langle c_1{}^{[1,2]}, c_2{}^{[2,1]}, c_3{}^{[2,1]}\rangle, \langle c_1{}^{[1,2]}, c_3{}^{[2,1]}, c_2{}^{[2,1]}\rangle\}
\end{aligned}$$

Using *nproj*, we may obtain a process for each role in C_1: $R_1 = nproj(C_1, 1) = c_1{}^{[1,2]}!; (c_2{}^{[2,1]}? \parallel c_3{}^{[2,1]}?)$, $R_2 = nproj(C_1, 2) = c_1{}^{[1,2]}?; (c_2{}^{[2,1]}! \parallel c_3{}^{[2,1]}!)$. The trace sets of R_1, R_2, and $R = R_1 \parallel R_2$, that represent respectively all possible executions of R_1, R_2, and their collaboration:

$$\begin{aligned}
[\![R_1]\!] &= \{\langle c_1{}^{[1,2]}!, c_2{}^{[2,1]}?, c_3{}^{[2,1]}?\rangle, \langle c_1{}^{[1,2]}!, c_3{}^{[2,1]}?, c_2{}^{[2,1]}?\rangle\} \\
[\![R_2]\!] &= \{\langle c_1{}^{[1,2]}?, c_2{}^{[2,1]}!, c_3{}^{[2,1]}!\rangle, \langle c_1{}^{[1,2]}?, c_3{}^{[2,1]}!, c_2{}^{[2,1]}!\rangle\} \\
[\![R_1 \parallel R_2]\!] &= \{\langle c_1{}^{[1,2]}, c_2{}^{[2,1]}, c_3{}^{[2,1]}\rangle, \langle c_1{}^{[1,2]}, c_3{}^{[2,1]}, c_2{}^{[2,1]}\rangle\}
\end{aligned}$$

3. CONFORMANCE RELATION

Formal methods provide many techniques to verify conformance between a specification and an implementation, *e.g*, using behavioural equivalences and preorders [6]. This has been applied recently to component and service based architectures [21]. However, in our context, we have an important constraint: the implementation source code is un-available since it is made up of distributed black-box peers. Some approaches suppose that such peers have behavioural interfaces, but in practice this is seldom the case. A recent proposal enables to retrieve such interfaces from black-box services using testing [7]. Still, this can be an intrusive technique, *i.e.*, that cause change on the service due to the active nature of the test being used. In our context we base on non intrusive passive testing techniques. Implementations may only be observed and checked for conformance using their logs, which are (linear) sequences of observations. This advocates for the use of a trace equivalence or a trace preorder.

Further, the relation between a choreography specification C and an implementation I can be seen with two mirror perspectives. In the former perspective, it is *the coordination middleware* that is tested. We are interested in the fact that I strictly enforces (over connected peers) what is described in C. I should then exhibit at least (or exactly) the behavior described in C. This may be supported using active testing of service orchestrations [8]. In the second perspective, it is *the cooperation of the peers* that is tested. We are interested in the fact that the peers do not interact in some other ways than what is specified in C. This corresponds to a passive testing using logs at the peers' locations. In such a case, we impose that the traces of I are included in the C ones. In this paper, we focus on this second perspective. We may now give our formal definition of the conformance of an implementation with reference to a specification. For this we base on trace preorder.

Definition 1. The preorder relation, denoted with $\preceq$, between two traces is defined recursively as follows: (i) $\langle\rangle \preceq \langle\sigma_2\rangle$; and (ii) $\langle\alpha, \sigma_1\rangle \preceq \langle\alpha, \sigma_2\rangle$ iff $\langle\sigma_1\rangle \preceq \langle\sigma_2\rangle$.

Given two trace sets T_1 and T_2, we write $T_1 \preceq T_2$ iff $\forall t_1 \in T_1, \exists t_2 \in T_2 \mid t_1 \preceq t_2$.

Indeed, while implementing a choreography, the developer may have to add important additional exchanges or synchronizing activities in the peers. This is especially the case with non-realizable choreographies. Take for example the specification $C = c_1{}^{[1,2]}; c_2{}^{[3,4]}$. Projecting it on its roles we get $R_1 = c_1{}^{[1,2]}!$, $R_2 = c_1{}^{[1,2]}?$, $R_3 = c_2{}^{[3,4]}!$, and $R_4 = c_2{}^{[3,4]}?$. Implementing the specification using these four peers as-is, *i.e.*, $I = R_1 \parallel R_2 \parallel R_3 \parallel R_4$, the developer cannot prevent that R_3 and R_4 interact on c_2 before R_1 has sent c_1 to R_2: trace $\langle c_2{}^{[3,4]}, c_1{}^{[1,2]}\rangle$ is in $[\![I]\!]$ while it is not in $[\![C]\!] = \{\langle c_1{}^{[1,2]}, c_2{}^{[3,4]}\rangle\}$. Therefore, the developer may decide to add a synchronizing message between R_2 and R_3, c_{sync}, to enforce the choreography, *i.e.*, replacing R_2 and R_3 above respectively by

${c_1}^{[1,2]}?; {c_{\text{sync}}}^{[2,3]}!$ and ${c_{\text{sync}}}^{[2,3]}?; {c_2}^{[3,4]}!$. However, in such a case, we would not have conformance, *i.e.*, $[\![I]\!] \not\preceq [\![C]\!]$. To support this, we formally define conformance as follows.

Definition 2. Given a specification S and an implementation I. We have $I\ conf\ S$ iff $[\![I]\!] \downharpoonright_{acts(S)} \preceq [\![S]\!]$, where $acts(S)$ is the set of all activities of S and $\downharpoonright$ is the filter operator, *i.e.*, $t \downharpoonright_X$ (or $T \downharpoonright_X$) retains only the elements of X in t (or T) while preserving their order.

Before going on, let us stress a basic assumption that we make on the relation between a choreography, C, and an implementation of it, I. We suppose a one-to-one function between the roles in C and the peers in I: each role R_i is implemented by exactly one peer P_i, and each peer P_i implements exactly one role R_i. This enables us to relate peer communications in log files to role activities in specifications. Based on the formal definition of conformance we proposed above, we may now define conformance between a choreography implementation and a choreography specification.

Definition 3. Given a choreography C with n roles, $R_i = nproj(C, i)$, and an implementation I with n peers, P_i, I conforms to C, denoted $I\ conf\ C$, iff the following two conditions hold:
(i) local conformance: $P_i\ conf\ R_i$, for every $i = 1..n$, and
(ii) global conformance: $(P_1 \parallel P_2 \parallel \ldots \parallel P_n)\ conf\ C$.

Example 2. Let us take again the C_1 choreography specifications from Example 1, and examine the conformance of an implementation I_1 made up of two peers, P_1 and P_2, whose logs are $t_1 = \langle {c_1}^{[1,2]}!; {c_2}^{[2,1]}? \rangle$ and $t_2 = \langle {c_1}^{[1,2]}?; {c_2}^{[2,1]}!; {c_2}^{[2,1]}! \rangle$ respectively. P_2 obviously does not conform to R_2 since in the later only one ${c_2}^{[2,1]}!$ may happen, while two ones appear in the log. P_1 has realized ${c_1}^{[1,2]}!; {c_2}^{[2,1]}?$, *i.e.*, an unique reception of ${c_2}^{[2,1]}?$. The second message ${c_2}^{[2,1]}!$ sent by P_2 may have been lost or cancelled. Hence P_1 conforms to R_1. In the model composition $P = P_1 \parallel P_2$, only the ${c_1}^{[1,2]}$ and ${c_2}^{[2,1]}$ interactions have been done. While P conforms to C_1 (Def 2), I_1 does not conform to C_1 since P_2 does not conform to R_2 (Def. 3).

4. IMPLEMENTATION

4.1 Testing Approach and Tool Support

In *active testing*, the tester interacts with the Implementation Under Test (IUT) by sending inputs (messages) and observing outputs (messages too). This method assumes a kind of controllability of the implementation through Points of Control and Observations (PCOs). Observing the outputs and comparing them to the expected ones, *i.e.*, those described by the specification, a verdict can be emitted. A *pass* verdict establishes the conformance of the implementation to its specification and a *fail* the contrary. *Passive testing* is a software testing method that relies only on observations on the running IUT. In passive testing the tester does not send messages to the IUT. It only observes the exchange (sending and reception) of messages between the IUT and its partners, through Points of Observation (POs). These observations will be compared to the specification in order to emit a verdict.

In both cases, active and passive testing, the implementation is considered as a black box, which means that the internal structure of the implementation is not known and no source code is available. The term "passive" relates to the fact that the tests do not disturb the natural operation of the IUT, to the contrary of "active" testing. Passive testing is also of particular interest since we do not always have the ability to control an IUT. Using passive testing in our work, testing can be done continuously and the peers in a collaboration can evolve dynamically. Such a seamless monitoring activity is a less costly activity as it does not require to make the IUT unavailable during the testing process.

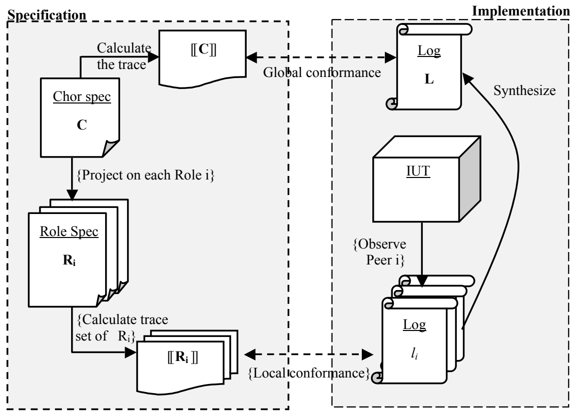

Figure 1: Testing Choreography Implementations

Our testing approach is represented by Figure 1. In this figure, repeated activities are denoted with the "{" and "}" symbols, *e.g.*, {*Project on each Role i*} means that the projection will be repeated on each role R_i of C. At the specification level, from a choreography specification C, we can obtain directly its trace set ($[\![C]\!]$). Afterwards, the set of local requirements R_i of its roles is obtained by using the natural projection function. From R_i we have also its trace set ($[\![R_i]\!]$). At the implementation level, we can obtain local implementation traces, called logs and denoted as l_i, from each PO. From n collected logs, we synthesize a global log, denoted as L. Global log, L, and local logs, l_i, will be checked against $[\![C]\!]$ and $[\![R_i]\!]$ to establish global conformance and local conformance, respectively. By Definition 3, the IUT conforms with C if and only if both global and local conformance are achieved.

Our approach is fully supported by a tool that supports *Chor* specification parsing, trace semantics retrieval, global log synthesis, and conformance checking. We have also implemented a monitoring module for the Apache ODE BPEL engine in order to retrieve local logs.

4.2 Observation of SOAP Messages

The approaches to capture SOAP messages in the context of Web services (WSs) can be classified into three groups. The first approach injects some modules into the WS engine in order to extract the expected information, *e.g.*, [23, 15, 20, 16]. The second approach intends to implement a software which is functions as a SOAP "proxy" and which is independent and external to the WS engine, *e.g.*, SoapUI[1] or Membrane SOAP/HTTP Monitor[2]. All incoming and outgoing SOAP messages of the WS must be directed to this SOAP proxy, which then forwards the messages to their destination while

[1]http://www.soapui.org/
[2]http://www.membrane-soa.org/soap-monitor/

conserving a copy of them. The last approach sniffs passively the SOAP messages which are transferred in the network by means of sniffers such as tcpdump[3] or wireshark[4]. The two last approaches capture the SOAP messages when they are outside of a WS engine, and are as a consequence independent from it. This means that they can be used for different types of WS engines. However these two approaches, and also more generally all approaches which only capture the SOAP messages outside WS engines, miss some important features which are necessary for testing. They cannot guarantee that the captured SOAP messages will be sent to the destination and that these messages are accepted by the WS partner. Another problem is that they cannot know which instance of a WS sends (or receives) the captured messages. Let us consider the following example. Web service R_1 can send its requests either to R_2 or R_3. This is described as $R_1 = {c_1}^{[1,2]}! \sqcap {c_2}^{[1,3]}!$. This example raises the problem a "false negative" verdict in case we capture two consecutive messages c_1 and c_2 at the PO of R_1. The verdict has to be a *fail* if these messages are sent by the same instance of the WS R_1, but if they are sent by two different instances the verdict has to be *pass*.

With its advantages we chose the first approach to implement our tool to collect the SOAP messages. The architecture of our monitor consists of a module integrated into Apache ODE, a WS-BPEL compliant WS orchestration engine. It allows to capture all messages which are sent or received by a monitored service and to record it into a log file. However as this module is integrated into the ODE engine, it may cause a limitation by considering only the monitoring of services which are implemented in WS-BPEL. To overcome this issue, we have also implemented a kind of wrapper which adds a WS-BPEL layer for services which are not WS-BPEL Web services. Hence, we are able to capture and to log all the SOAP messages that are transmitted between the monitored peer and its partners.

Each peer is observed by a monitor that captures all the input and output messages of the peer. The captured messages that do not concern the testing choreography will be discarded by the tester. When a message is sent or received by a peer of the IUT, an *observation* is recorded immediately in the log file. A *Chor* interaction $c^{[i,j]}$ means that a message c is transferred from R_i to R_j. As a consequence, an observation contains the sender, the receiver, and the type of the message. Moreover, at the specification (role) level, an interaction is equipped with "!" or "?" to indicate if it is a sending or a reception. Hence, we annotate observations accordingly in logs. To ease the reconstruction of the order between observations of different logs l_i, an observation also contains the time of the observation. Furthermore, to correlate the observations of a sending message and of the corresponding reception one, we inject, at the sending moment, an identity in the header of SOAP messages. Formally, the observation is defined as follows. Given an IUT which consists of n peers $P = \{P_1, \ldots, P_i, \ldots, P_n\}$, an *observation* is a tuple $ob = (act, id, t, s, r, m)$ where $act \in \{SEND, RECEIVE\}$ indicates a sending or a reception, id is a message identity, t is the reception or sending time, $s, r \in P$ with $s \neq r$ are the sender and the receiver, and m is the message. A log $l_i = \langle ob_1, ob_2, \ldots, ob_m \rangle$ for a peer P_i is a sequence of all the observations of messages which are sent/received by P_i to/from others peers of the IUT.

[3]http://www.tcpdump.org/

[4]http://www.wireshark.org/

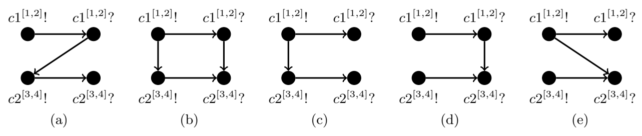

Figure 2: Sendings and Receptions Correlation

4.3 Global Log Synthesis

We have a set of local logs l_i and we need to synthesize a global log in order to perform global conformance testing, *i.e.* compare this global log with the trace set of the choreography. As proposed in [24], we assume that clocks of peers are synchronous to support the construction of an order between two observations that have happened in two different local logs. This assumption typically holds, *e.g.*, when collaborations are deployed over clouds. Communication is synchronous in *Chor*. This means that the sending and the reception events for an interaction happen at the same time. In an implementation there is usually a delay between a sending and the corresponding reception. A choreography $C = {c_1}^{[1,2]}; {c_2}^{[3,4]}$ means that the passing of message c_1 from R_1 to R_2 should happen before the passing of message c_2 from R_3 to R_4. Figure 2 represents all possible correlations between the sending and the reception time of messages c_1 and c_2, where the order of execution is denoted with an arrow "$\rightarrow$". For example, in Figure 2(a), the sending of c_2 only happens after the reception of c_1. One can note that the case represented in Figure 2(a) is the strongest one, *i.e.*, it implies the other cases. For example, if ${c_1}^{[1,2]}?$ happens before ${c_2}^{[3,4]}!$ as in Figure 2(a), we can infer that ${c_1}^{[1,2]}!$ happens before ${c_2}^{[3,4]}!$ as in Figure 2(c) because of execution order transitivity (*i.e.*, $a \rightarrow b \wedge b \rightarrow c \Rightarrow a \rightarrow c$). Hence case (a) is the default in our tool. However one may select any other case in the tool.

Algorithm 1 represents the synthesis of global log based on the first case. This algorithm is divided into two parts. The first part (lines 1–17) creates a new list of observations by merging n local logs. The observations in this list are sorted by the order of execution time. In fact, each local log has partial ordered, so that this part is basically to sort n arrays which are already sorted. The second part (lines 18–27) is to synthesize the global observations from the sorted list of local observations corresponding to the case represented in Figure 2(a), which imposes that, in logs, the reception of a message has to be adjacent to the sending of this message. This means that there is nothing (neither sending nor reception of other messages) which can happen in this interval.

4.4 Testing Algorithm

Before presenting the algorithms for global and local conformance verification, we present Algorithm 2 which verifies the preorder relation between a log and a trace. It is the implementation of Definition 1.

The algorithm for global conformance verification is presented in Algorithm 3. The implementation to realize the choreography may need additional interactions. In this case, we only keep the interactions which are present in the specification by using the filtering function. As soon as we find a trace of the choreography such that the global log is its preorder then this log passes the conformance check and the algorithm is stopped. If we visit the whole trace set without finding any trace of the specification such that the global log is its preorder, this means that the implementation is not exhibiting a behavior represented by the specification, which entails to return a *fail*. The algorithm for local conformance verification is similar. The only difference is that we do not have to synthesize the global log from n local logs, *i.e.*, line 2 in Algorithm 3.

Algorithm 1: Synthesis of observations ($synthesisObservations(L)$)

```
   Input: A set of $n$ logs $L = \langle l_1, l_2, \ldots, l_n \rangle$
   Output: A log $l = \{ob_1, ob_2, \ldots, ob_m\}$
1  /* Get global order of observations                    */
2  $t = \langle\rangle$ ;
3  $index_1 = index_2 = \ldots = index_n = 1$ ;
4  while $true$ do
5      $j = -1$ ;
6      for $i = 1$ to $n$ do
7          if $index_i \leq length(l_i)$ then
8              $o = getElementAt(l_i, index_i)$ ;
9              if $j = -1$ then
10                 $min = o$ ;
11                 $j = i$ ;
12             if $o.t < min.t$ then
13                 $min = o$ ;
14                 $j = i$ ;
15     if $j = -1$ then break;
16     $t = t \cup \{min\}$ ;
17     $index_j = index_j + 1$ ;
18 /* Synthesis of global log                               */
19 $l = \langle\rangle$ ;
20 for $i = 1$ to $length(t)$ do
21     $o_1 = getElementAt(t, i)$ ;
22     $o_2 = getElementAt(t, i + 1)$ ;
23     if $o_1.act = SEND$ and $o_2.act = RECEIVE$ and $o_1.id = o_2.id$ then
24         $o$ = new $Observation$(null, $o_1.id$, null, $o_1.s, o_1.r, o_1.m$) ;
25         $l = l \cup \{o\}$ ;
26         $i = i + 1$ ;
27 return $l$ ;
```

Algorithm 2: Preorder verification ($isPreorder(l, t)$)

```
   Input: A trace $t = \langle \alpha_1, \alpha_2, \ldots, \alpha_n \rangle$
   Input: A log $l = \langle ob_1, ob_2, \ldots, ob_m \rangle$
   Output: true if $l \preceq t$ else false
1  if $m > n$ then return false;
2  for $i = 1$ to $m$ do
3      if $ob_i.s \neq \alpha_i.s$ or $ob_i.r \neq \alpha_i.r$ or $ob_i.m \neq \alpha_i.m$ then return false;
4  return true ;
```

Algorithm 3: Global conformance verification

```
   Input: A Chor specification $C$
   Input: A set of $n$ logs $L = \{l_1, l_2, \ldots, l_n\}$
   Output: Verdict (pass or fail)
1  $T = [\![C]\!]$ ;
2  $l$ = synthesisObservations($L$) ;
3  $l = l \downarrow_{act(C)}$ ;
4  foreach trace $t$ of the $T$ do
5      if $isPreorder(l, t)$ then return pass;
6  return fail ;
```

5. EXPERIMENTS

5.1 Test Case Experiment

We have experimented our approach and our tools on several medium-size case studies. For each of them, we have defined several correct implementations of its *Chor* choreography, and then we have performed mutations at the level of the implementation peers representative of errors in the development/implementation process. These mutations can be categorized as follows: adding (a), removal (r), replacement (x), and reordering (o) of interactions, and change (c) of structuring operators.

We present in Table 1 results on one of our case studies, related to the online buying and delivery of goods (see below). The rows of the table correspond to different correct implementations (marked with –) and to different mutants. Its columns corresponds to the inputs which are a *Chor* specification defined by the number of: peers (# Peers), the number of interactions (# Int.) and the number of the traces (# Traces); and a set of logs represented by the number of observations before (#1) and after (# 2) the filtering on the set of interactions defined in the roles of the *Chor* specification (see Def. 2). The remaining columns are devoted to the kind of mutations being performed (Mutations), the testing verdicts (local, P_i, and global, P) and the duration of the testing process. As far as the verdicts are concerned, an implementation or a peer may be conform while not achieving completely the envisioned behaviour. This may correspond to a potential deadlock situation. It is detected by our tool and denoted by (⊠).

Our case study is described in *Chor* as follows: $Order^{[1,2]}$; $(Reject^{[2,1]} \sqcap Confirm^{[2,1]}; Payment^{[1,2]}; (Invoice^{[2,1]} \parallel Shipment^{[2,3]}; Postage^{[2,3]}; Distribution^{[3,1]}))$. The implementation of this specification has been done with three Web services, Customer (P_1), Supplier (P_2), and Shipper (P_3), running on three ODE engines.

The first three rows correspond to correct implementation logs. Row 1 corresponds to a case where the supplier rejects the order. It is hence much shorter, in observations, than rows 2 and 3. These correspond to the supplier accepting the order. The only difference between them is the order in which $Invoice$ is done *w.r.t.* the rest of the choreography (*Shipment*, etc.), *i.e.*, the order in which parallel actions (∥ in the *Chor* specification) are done.

In row 4, a new message is sent in P_3 to inform P_2 about the goods being delivered to the client ($Inform^{[3,2]}!$). Since this is a new message *w.r.t.* the choreography, it is filtered out, and peer P_3 is still conform to its role. The same yields for the whole implementation. In row 5, the corresponding message is also added in P_2 ($Inform^{[3,2]}?$). Again, it is filtered out and the implementation is correct. In row 6, P_2 is mutated in order not to send $Postage^{[2,3]}$ to P_3 anymore. Since it is at the end of the P_2 role, this one is still conform to its role. Further, P_3 blocks waiting for it (hence it does not send $Distribution^{[3,1]}$), but also conforms to its role. Then, in row 7, both P_2 and P_3 have agreed not to use $Postage^{[2,3]}$ (it is removed in both peers). While P_2 stays conform, P_3 is not conform anymore, since it sends $Distribution^{[3,1]}$ before having received $Postage^{[2,3]}$, which is forbidden by its role.

Finally, we have a last mutation in row 8. We replace $Shipment^{[2,3]}!; Postage^{[2,3]}!$ by $Postage^{[2,3]}!; Shipment^{[2,3]}!$ in P_2. We also replace $Shipment^{[2,3]}?; Postage^{[2,3]}?$ by $Shipment^{[2,3]}? \parallel Postage^{[2,3]}?$ in P_3. Now P_2 and P_3 may interact doing $Shipment^{[2,3]}$ before $Postage^{[2,3]}$. This is detected in the log of P_2, the one of P_3, and in the global log, hence all of these are not conform since this contradicts the specification.

We have experimented our approach on bigger case studies. The biggest one is a mutant with 7 peers, 11 distinct

Table 1: Online buying case study

	Mutation	*Chor* specification			Obs		Verdict				Duration
		# Peers	# Int.	# Traces	# 1	# 2	P_1	P_2	P_3	P	(seconds)
1	–	3	8	5	5	4	✓	✓	✓	✓	0.013
2	–	3	8	5	16	14	✓	✓	✓	✓	0.017
3	–	3	8	5	16	14	✓	✓	✓	✓	0.018
4	(a) in P_3	3	8	5	17	14	✓	✓	✓	✓	0.016
5	(a) in P_2 & P_3	3	8	5	18	14	✓	✓	✓	✓	0.015
6	(r) in P_3	3	8	5	11	10	✓(⊠)	✓(⊠)	✓(⊠)	✓(⊠)	0.014
7	(r) in P_2 & P_3	3	8	5	14	12	✓	✓(⊠)	×	×(⊠)	0.014
8	(o) in P_2 & (c) in P_3	3	8	5	16	14	✓	×	×	×	0.018

interaction channels, and 116,640 traces in the choreography trace set. We have 52 observations (40 after filtering), and the testing time is 2.92 seconds. We have observed that the testing time is greater for fail mutants since in the worst case we have to check all of the choreography traces to give a verdict (see Alg. 3).

5.2 Generic Experiments

We have evaluated our testing approach by conducting practical experiments. Our aim was to assess the scalability (computation time) with reference to the number of messages, and this both for correct and incorrect logs. Our experiments can be explained as follows. We produce automatically a *Chor* specification by choosing the number of roles, the number of messages, and the operator types. Consequently, logs are produced automatically from this specification by applying the projection for each role. We then inject faults inside the logs in order to analyse the impact of the presence of faults on the time of the testing process. Different kinds of faulty logs (mutants) have been produced. In the experiments presented in Figure 3, we inject faults corresponding to the reordering of messages. The positions of faults are selected randomly. For sake of simplicity, we present here the experiments only for two peers.

Figure 3 shows the time (in milliseconds) of verifications with $1,000$, $2,000$, $5,000$, $10,000$, and $20,000$ messages, and for correct or incorrect logs. Each measure is computed from the average of 50 runs. Experiments are relative to the local verifications (for respectively peer 1 and peer 2), the global conformance verification (including global log synthesis), and the whole conformance testing process (sum of the 3). These experiments have been performed on a 2GHz Intel Core 2 Duo MacBook laptop with 4GB of RAM. The times shown in Figure 3 do not take into account the time for the computation of the traces of the *Chor* specification since it depends only on this specification. For $1,000$, $2,000$, $5,000$, $10,000$, and $20,000$ consecutive messages, this would be 43, 172, $1,099$, $3,292$, and $14,305$ milliseconds respectively.

In Figure 3, we can observe that the local testing time is very small for any % of faults. In the worst case (upper bound), *i.e.*, for a correct log, we have to check the whole of it. The computation time for different % of faults in the global conformance testing is close to the case without faults. This is due to the important part of it used for the global log synthesis that grows with the number of messages. This directly impacts the overall testing time. Still, with a maximum time of 139 milliseconds for $20,000$ messages, we believe that our approach is scalable.

6. RELATED WORK

Checking the conformance of service choreographies has mainly been addressed from a verification point of view, using model-checking or behavioural equivalence techniques [12, 17, 19, 18, 4]. Among them, some authors promote the use of *Chor* as a specification language [17, 19]. The main issue addressed in these works is to check whether a set of collaborating peers exhibit the same behaviours than a global choreography specification, or to give techniques to retrieve peer skeletons satisfying this property.

Formal verification and testing are complementary. The former enables one to be sure that design artifacts (choreography specification, role requirements, etc.) are correct. However, when it comes to check the correctness of a choreography implementation, with black box peers for which no source code or model (behavioural interface) is available, testing is required. Numerous works have addressed the testing of services, including composite ones [8]. These works apply if one wants to check the conformance of a coordination media, implemented as a single orchestration, with reference to a choreography specification. In fact, choreographies are more collaborative and distributed in nature. Few works address the testing of full-fledged choreographies. Reo is a very expressive language for the implementation of coordinating component connectors [1]. In [1], the authors propose a fault-based technique to generate test cases from a specification given in terms of pre- and post-conditions. Passing tests on an implementation is not addressed. In [11], an approach for the input-output testing of a coordinator specified using Reo is presented. This is an active testing approach, while we use passive testing not to interfere with the running IUT. Further, Reo is dedicated at the implementation of coordinators, hence it is rather an interconnected model of choreography and can be used to test a centralized coordinator choreography implementation (*e.g.*, an orchestrator - coordinator). To the contrary, in this work, we use a simpler language, *Chor*, with an interaction model, and we test the coordinated distributed peers. An approach for WS-CDL testing in which dynamic symbolic execution is used to generate test inputs and assertions is presented in [25]. However, this is an active testing approach that considers the IUT as a white box.

The passive testing of choreographies is an alternative when control over the choreography is not possible (only observing is possible). Passive testing has been intensively used in the area of protocol testing [5, 3] based on the definition of invariants that represent the set of properties to be checked on the log. The testing process is performed off-line on a log. Recently, passive testing approaches have been proposed to test Web service orchestrations [14], still with an invariant-based approach, but where timing constraints have been added. Testing is performed by using a pattern-matching analysis without defining an explicit conformance relation. As far as choreography is concerned, a black box choreography

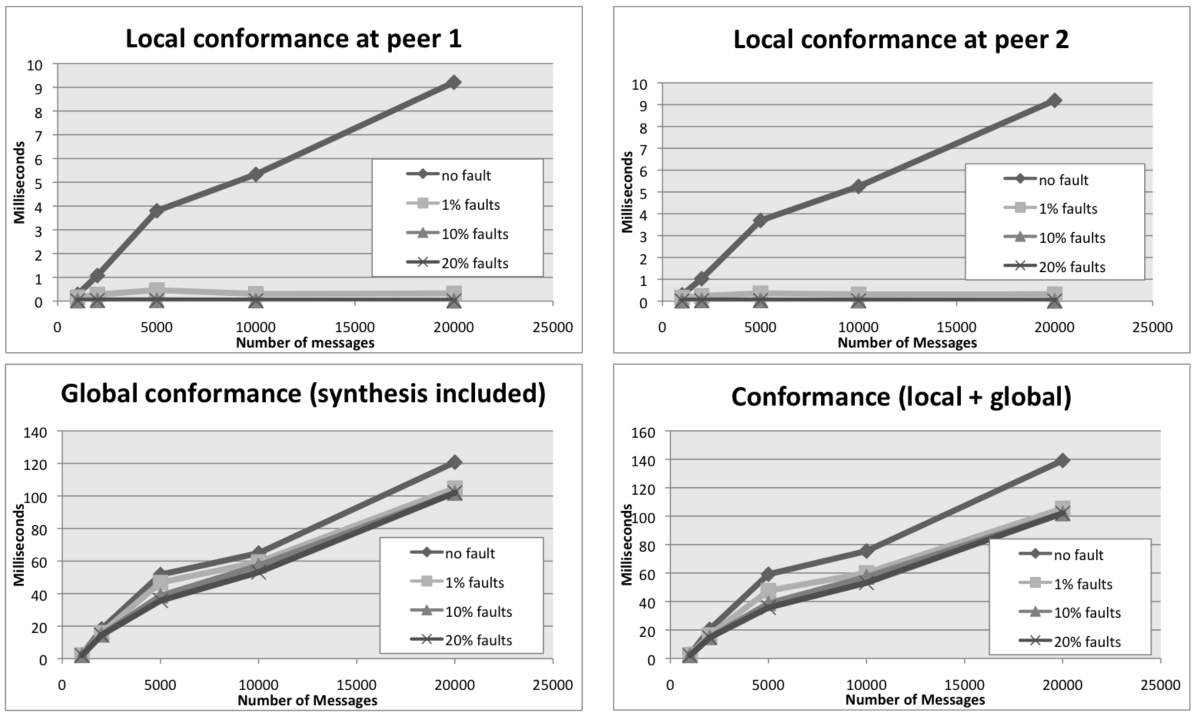

Figure 3: Tester Scalability

testing approach that extends [3] is presented in [2] and is based on the checking of invariants too. Local invariants are used to check peer (local) logs, while global invariants are used to express global-level properties. In [10], the authors propose a runtime monitoring and verification technique for choreography constraints expressed in LTL. With reference to [2, 10], we want to check the conformance of an IUT *w.r.t.* a specification expressed in a choreography language rather than in an abstract logic. Further, the global invariants used in [2] cannot detect a violation of the execution order among peers, and the approach in [10] focuses on the subset of choreographies which constrain the sequence of message exchanged by one specific partner with its peers. Moreover, compared to [2], we explicit our conformance relations.

7. CONCLUSIONS AND FUTURE WORK

Choreography plays an important role in the specification of collaborations. An issue, known as choreography conformance checking, is to check whether a set of distributed communicating peers implements correctly or not some collaboration specification. To support this activity, we have introduced a formal passive testing approach that enables one to check for choreography conformance both at the local (peer) and at the global (collaboration) level, without interfering with the implementation being tested. Our approach also supports the use of black-box peers, for which no source code or behavioural interface is available, which is often the case with peers developed by different third-parties.

We have implemented a comprehensive tool support for our approach and, as a proof of concept, we have developed a monitoring module for collaborations implemented with peers encapsulated as composite WS. A first perspective is to propose support for further implementation architectures. We rely on *Chor* for choreography specification since it is both expressive, abstract and based on an interaction model of choreography. Another perspective is to study the application of our approach to BPMN, since this standard has been very recently extended into version 2.0 to support choreography as a first-class entity. There are several monitoring frameworks for WS, *e.g.*, [23, 15, 20, 16]. In our work, we use an ad–hoc monitoring module to enable global log synthesis. A last perspective is to use instead a choreography-specific monitoring technique [22] in order to enable global log synthesis while logging only events related to the choreography specification.

8. REFERENCES

[1] B. K. Aichernig, F. Arbab, L. Astefanoaei, F. S. de Boer, M. Sun, and J. Rutten. Fault-Based Test Case Generation for Component Connectors. In *Proc. of TASE'09*, 2009.

[2] C. Andrés, M. Cambronero, and M. Núñez. Passive Testing of Web Services. In *Proc. of WS-FM'10*, 2010.

[3] C. Andrés, M. G. Merayo, and M. Núñez. Applying Formal Passive Testing to Study Temporal Properties of the Stream Control Transmission Protocol. In *Proc. of SEFM*, 2009.

[4] S. Basu and T. Bultan. Choreography Conformance via Synchronizability. In *Proc. of WWW'11*, 2011.

[5] E. Bayse, A. R. Cavalli, M. Núñez, and F. Zaïdi. A Passive Testing Approach based on Invariants: Application to the WAP. *Computer Networks*, 48(2):235–245, 2005.

[6] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra.* Elsevier, 2001.

[7] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic Synthesis of Behavior Protocols for

Composable Web Services. In *Proc. of ESEC/FSE'09*, 2009.

[8] M. Bozkurt, M. Harman, and Y. Hassoun. Testing Web Services: a Survey. Technical report, King's College London, 2010.

[9] G. Decker, O. Kopp, and A. Barros. An Introduction to Service Choreographies. *Information Technology*, 50(2):122–127, 2008.

[10] S. Hallé and R. Villemaire. Runtime Monitoring of Web Service Choreographies using Streaming XML. In *Proc. of SAC'09*, 2009.

[11] N. Kokash, F. Arbab, B. Changizi, and L. Makhnist. Input-output Conformance Testing for Channel-based Service Connectors. In *Proc. of PACO'11*, 2011.

[12] J. Li, H. Zhu, and G. Pu. Conformance Validation between Choreography and Orchestration. In *Proc. of TASE'07*, 2007.

[13] A. Marconi and M. Pistore. Synthesis and Composition of Web Services. In *Proc. of SFM'09*, 2009.

[14] G. Morales, S. Maag, A. R. Cavalli, W. Mallouli, E. M. de Oca, and B. Wehbi. Timed Extended Invariants for the Passive Testing of Web Services. In *Proc. of ICWS*, 2010.

[15] O. Moser, F. Rosenberg, and S. Dustdar. Non-Intrusive Monitoring and Service Adaptation for WS-BPEL. In *Proc. of WWW'08*, 2008.

[16] O. Moser, F. Rosenberg, and S. Dustdar. VieDAME - Flexible and Robust BPEL Processes through Monitoring and Adaptation. In *Proc. of ICSE'08*, 2008.

[17] Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the Theoretical Foundation of Choreography. In *Proc. of WWW'07*, 2007.

[18] G. Salaün, T. Bultan, and N. Roohi. Realizability of Choreographies using Process Algebra Encodings. In *Proc. of IFM'09*, 2009.

[19] G. Salaün and N. Roohi. On Realizability and Dynamic Reconfiguration of Choreographies. In *Proc. of WASELF'09*, 2009.

[20] J. Simmonds, Y. Gan, M. Chechik, S. Nejati, B. O'Farrell, E. Litani, and J. Waterhouse. Runtime Monitoring of Web Service Conversations. *IEEE Transactions on Services Computing*, 2(3):223–244, 2009.

[21] M. ter Beek, A. Bucchiarone, and S. Gnesi. Formal Methods for Service Composition. *Annals of Mathematics, Computing & Teleinformatics*, 1(5):1–10, 2007.

[22] B. Wetzstein, D. Karastoyanova, O. Kopp, F. Leymann, and D. Zwink. Cross-Organizational Process Monitoring based on Service Choreographies. In *Proc. of SAC'10*, 2010.

[23] G. Wu, J. Wei, and T. Huang. Flexible Pattern Monitoring for WS-BPEL through Stateful Aspect Extension. In *Proc. of ICWS'08*, 2008.

[24] F. Zaidï, E. Bayse, and A. Cavalli. Network Protocol Interoperability Testing based on Contextual Signatures and Passive Testing. In *Proc. of SAC'09*, 2009.

[25] L. Zhou, J. Ping, H. Xiao, Z. Wang, G. Pu, and Z. Ding. Automatically Testing Web Services Choreography with Assertions. In *Proc. of ICFEM'10*, 2010.