HAL
open science

Online Verification of Value-Passing Choreographies through Property-Oriented Passive Testing

Huu Nghia (hữu Nghĩa) Nguyễn, Pascal Poizat, Fatiha Zaïdi

To cite this version:

Huu Nghia (hữu Nghĩa) Nguyễn, Pascal Poizat, Fatiha Zaïdi. Online Verification of Value-Passing Choreographies through Property-Oriented Passive Testing. 9th International Symposium on High-Assurance Systems Engineering (HASE), Oct 2012, Omaha, United States. 10.1109/HASE.2012.15 . hal-01367296

HAL Id: hal-01367296

https://inria.hal.science/hal-01367296v1

Submitted on 15 Sep 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Online Verification of Value-Passing Choreographies through Property-Oriented Passive Testing*

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 roles played by peers in a collaboration. Choreography conformance testing aims at verifying whether a set of distributed peers collaborates wrt. a choreography. Such collaborations are usually achieved through information exchange, thus taking data into account during the testing process is necessary. We address this issue by using a non-intrusive passive testing approach based on functional properties. A property can express a critical (positive or negative) behaviour to be tested on an isolated peer (locally) or on a set of peers (globally). We support online verification of these kind of properties against local running traces of each peer in a distributed system where no global clock is needed. Our framework is fully tool supported.**

***Keywords*-choreography, conformance checking, passive testing, online verification, tools.**

I. Introduction

Context. *Choreography* is the description with a global perspective of *interactions* between *roles* in some collaboration. A choreography has two components: a set of roles, and the specification of the legal orderings of message exchanges between these roles. *Implementation* of choreography is a set of *services* (web services, organizations, humans) playing roles wrt. the choreography. The development process of peers can be top-down, where skeletons of peers are generated from choreography then completed by developers, or bottom-up, where peers may be written from scratch or be reused. These peers are then coordinated to fulfill the choreography.

Issues. One key issue with choreography, so-called *conformance*, is to check whether the implementation exhibits or not the behaviours specified in the choreography. Testing, consisting in performing experiments on implementation, is a mean to guarantee that the conformance relation is holding when the choreography was implemented. In distributed context, the control overall implementation under test (IUT) is difficult to achieve, it is even impossible in some situation, *e.g.*, system working 24/7 where the IUT can be running in their real environment. Therefore, the testing *should not disturb* its natural operation, since it might produce a wrong behaviour of the overall system. As a consequence, it is prefer to *detect faults as soon as they occur* to prevent wrong behaviours of overall system as soon as possible. Since IUT might run in real environment, so the test may be applied at any moment in the lifetime of IUT. Finally, interactions between peers may have parameters which represent exchanged information. Therefore, the correctness of *data exchange should be verified* in testing process.

Related Work. Formal testing can be differentiated as active or passive, according to its level of controllability. *Active testing* consists in applying a set of test cases to the IUT and then analyzing its reaction to emit a verdict. This method assumes a kind of controllability of the IUT through Points of Control and Observations (PCOs). *Passive testing* relies only on observing 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 Table I, we compare approaches for testing of choreography. Columns 2 and 3 focus on passive testing and property-oriented approach. To the contrary of active testing, the passive testing does not disturb the natural operation of the IUT. It is also of particular interest since we do not always have the ability to control an IUT. By using passive approach, testing can be done continuously and the peers in a collaboration can evolve dynamically. Following [1], passive testing approach is devised into two groups, naïve and property-oriented based approaches. The *naïve based approach* consisting in comparing the specification trace with the one of the implementation in a forward or backward manner. Whereas in *property-oriented based approach*, only critical properties, which are given either by expert or extract from specification model, are compared with the execution trace [2].

Columns 4 and 5 are relative to the data support in testing process. Some approaches, [5], [7] just *abstract away from data*. This is known to yield over-approximation issues, *e.g.*, false negatives in the verification process. Furthermore, the information exchange is usually in complex type, thus complex data treatment should be supported in testing. This was also a limitation of our previous work [4]. Column 6

* This work is supported by the Personal Information Management through Internet project (PIMI-ANR-2010-VERS-0014-03) of the French National Agency for Research

Table I
TESTING OF CHOREOGRAPHY

	Passive		Data & Value-Passing		Online	without g. clock
	supported	prop.	supported	complex		
[3]	no	no	yes	no	no	no
[4]	yes	no	no	no	no	no
[5]		yes	no	no	no	yes
[6]		yes	yes	no	yes	no
this paper		yes	yes	yes	yes	yes

and 7 is relative to the realization of testing. Online testing intends to be able to detect faults as soon as they are generated in IUT. Some testing approaches require a global clock to correlate local traces of peers.

Contributions. Our contributions are manifold. First, we define a formal model based on Symbolic Transition Graph with Assignment (STGA) [8] for both peers and choreography with supporting complex data types. We formalize our local and global properties, inspired from [5] by taking data into account. The local properties are used to test behaviours of one isolated peer wrt. its specification model, while the global properties test the collaboration of a set of peers wrt. its choreography model. A negative version of property is also introduced. The *positive property* is used to describe expected behaviors, while the *negative property* describes forbidden behaviors. If a model is available, we should verify the correctness of these properties wrt. its model. Since the model is equipped with data and loops, we use symbolic techniques in order to avoid the state space explosion problems (when message parameters or variables are flattened wrt.their infinite domains). Hence, messages parameters and variables are represented by symbolic values instead of concrete ones. Once the correctness of the properties is ensured, both of the local and global properties are verified against the local execution traces of the running IUT collected at each peer. Our verification process does not require a global clock since we assume that global properties express relation between several local properties and in particular by specifying relation between data among the choreography. Finally, we validate our approach by a case study in which peers are realized by Web services. The SOAP messages exchanged of each service are progressively collected as an XML stream. We translate our properties into XQuery to perform an online verification of the properties on the IUT XML stream. Finally, our framework is fully tool supported.

Overview. The remainder of the paper is organized as follows. We present our formal models of choreography and peers based on STGA in Section II. We then formalize our notion of local and global property in Section III. Section IV introduces the implementation of our framework and some experiments. We end with conclusions and perspectives.

II. A FORMAL MODEL FOR SERVICE CHOREOGRAPHIES

In our framework, we assume that the properties are provided by the standards or by the choreography experts. They will be then checked on the execution traces (log) of the IUT which are collected at running time. In order to reason about the format of the logs and also about the one of properties, we need to define formally a model of the IUT. Furthermore, if the model of IUT is available, we has to check the correctness of properties wrt. the model to ensure no divergence between them. In this section, we briefly introduce some notions about STGA which can be used to specify distributed systems with a global perspective, *i.e.*, *choreographies*, and to describe the pieces of a distributed implementation, *i.e.*, *peers*. Due to this multi-purpose objective, it is first presented in terms of an abstract alphabet, E. We then explain how E can be realized for the different purposes.

Symbolic Transition Graph with Assignments (STGAs). Let Val be the set of data values, ranged over by v, and Var be the set of variables, ranged over by x, y, z, x_1, etc. We use $dom(x)$ to represent domain of variable x, *i.e.*, $dom(x) \subseteq Val$. $DTerm$ is a set of data expressions, ranged over by t. $BTerm$ is a set of boolean expressions, ranged over by ϕ. We assume that $Var \cup Val \subseteq DTerm$, $t = t' \in BTerm$ for any $t, t' \in DTerm$. $BTerm$ is closed under the usual operators $\wedge, \vee, \neg$.

A STGA is a transition system where each state is associated with a set of free variables and each transition may be guarded by a boolean expression $\phi \in BTerm$ that determines if the transition can be fired or not. A *guarded transition* is labelled by a triple (ϕ, e, A), *e.g.*, $s \xrightarrow{[\phi]\, e/A} s'$ represents a guarded transition from state s to state s' with a guard ϕ, an event e, and an action A. An *event* e takes the form $o(l_1 = x_1, \ldots, l_n = x_n)$, rewritten as $o(l_i = x_i)$ for short, where o represents the control part and the composite date exchange is represented by a set of $l_1 = x_1, \ldots, l_n = x_n$. Each field of this data structure is pointed by a label l_i and its value is the one of the variable x_i. We also introduce specific event τ which denotes non-observable internal computations. An *action* A is a sequence of assignments. An assignment takes the form $x := t$. Thus, the action $A = (x_1 := t_1; x_2 := t_2; \ldots; x_m := t_m)$, will be executed in a sequential manner. We denote A_x as $\{x_1, .., x_m\}$ and $A_{\bar{t}}$ as $\{t_1, \ldots, t_m\}$. We use $fv(a)$ and $bv(a)$ to denote respectively the set of free and bound variables used in some expression a. These sets for an event will be detailed later.

Definition 1: STGA is a tuple $\mathcal{M}(E) = (S, s_0, T)$ where, S is a non empty set of states, each state s having an associated set of free variables $fv(s)$, $s_0 \in S$ is the initial state, and T is a set of transitions. If $s \xrightarrow{[\phi]\, e/A} s'$ is a transition of T then $fv(\phi) \cup fv(e) \cup fv(A_{\bar{t}}) \subseteq fv(s)$, $fv(s') \subseteq fv(s) \cup bv(e) \cup A_x$ and $bv(e) \cap (A_x \cup fv(A_{\bar{t}})) = \emptyset$

In a transition $s \xrightarrow{[\phi]\, e/A} s'$, we can omit ϕ if it is true, and A if there is no assignment. In the above definition, neither $A_x \subseteq fv(s)$ nor $fv(e) \subseteq A_x$ is required.

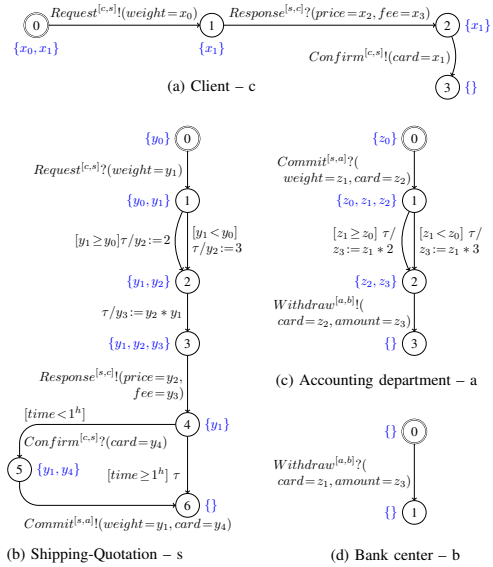

Figure 1. STGAs of four services

Service model. A service model describes the behaviour of a service in a collaboration. The events of service a can be abstracted as sending and reception, denoted respectively with $o^{[a,b]}!(l_i = x_i)$ and $o^{[a,b]}?(l_i = x_i)$, where b is another service, *i.e.*, $a \neq b$. A *service model* for a service a, a set of services ID with $a \in ID$, a set of operations O, a set of label L, and a set of variables V, is an element, $\mathcal{M}^S$, of $\mathcal{M}(E)$ with $E = \{o^{[a,b]}!(l_i = x_i), o^{[a,b]}?(l_i = x_i) \mid o \in O \wedge b \in ID \wedge a \neq b \wedge l_i \in L \wedge x_i \in V\} \cup \{\tau\}$. Sets of free and bound variables of sending and reception are defined as follows: $fv(o^{[a,b]}!(l_i = x_i)) = bv(o^{[a,b]}?(l_i = x_i)) = \bigcup\{x_i\}$, and otherwise both $fv(e)$ and $bv(e)$ are empty.

Example 1: Let us present a running example with four services: c (client), s (shipping-quotation – Sq), a (accounting department – Ad), and b (bank center - Bc). Their STGAs are shown in Figure 1. The sets of free variables attached at initial states of these STGAs state that the client, the Sq, and the Ad services work with parameters $\{x_0, x_1\}$, $\{y_0\}$ and $\{z_0\}$ respectively, while the Bc service works without parameter. The client wants to ship some good, (s)he issues a request shipping to the Sq service by providing the weight of goods to be sent, then it receives a response indicating its price and a fee to pay. If the client agrees with this price, then the client will send its credit card number to the Sq service. In the Sq service side, after receiving the request, based on the received weight, and its price list (for sake of simplicity, we only consider two prices, 2 and 3), it will calculate a fee, then respond with the fee to the client. After that, it will wait to receive the client credit card number during one hour, then it will commit the client information to the Ad service. The Ad service will withdraw money from the Bc service based on the received information from the Sq service.

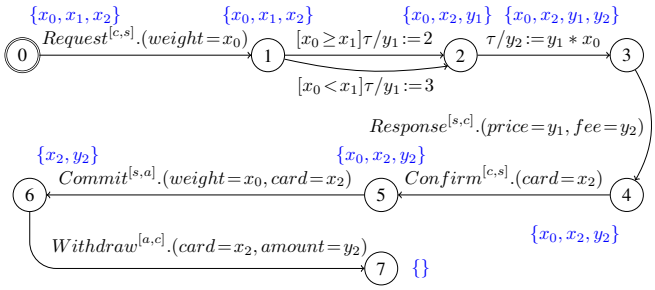

Figure 2. Shipping choreography

Choreography model. A choreography model describes, with a global perspective, the legal interactions among roles played by peers of a distributed system. In a choreography, each peers is identified by a unique name. The basis of an interaction-model choreography description is the interaction event. An interaction o from role a to role b is denoted by $o^{[a,b]}.(l_i = x_i)$. A *choreography model* for a set of services ID, a set of operations O, a set of labels L, and a set of variables V, is an element of $\mathcal{M}(E)$ with $E = \{o^{[a,b]}.(l_i = x_i) \mid o \in O \wedge a \in ID \wedge b \in ID \wedge a \neq b \wedge l_i \in L \wedge x_i \in V\} \cup \{\tau\}$. The set of free variables of the interaction $o^{[a,b]}.(l_i = x_i)$ is $\bigcup\{x_i\}$, whereas its set of bound variables is empty.

Example 2: Let us consider a *shipping choreography* as in Figure 2. This choreography presents the global behaviour of the four services described in Example 1. It states that the *weight* in *Commit* interaction is the one in *Request* interaction. It guarantees also that the *amount* issued from Ad to Bc services is the same with the *fee* responded by the Sq service to the client.

Semantics of STGA. The semantics of STGA are introduced in [8] by both symbolic and ground semantics. We choose late–ground semantics in our framework since we will work on implementation trace which is a sequence of messages. We concretize by using an evaluation function. An *evaluation* $\rho \in Eval$ is a mapping from Var to Val. We denote $\rho(t)$ as the evaluation of expression t by ρ. Obviously, $\rho(t) \subseteq Val$ and $\rho(\phi) \in \{true, false\}$. We write $\rho \models \phi$ to indicate $\rho(\phi) = true$. Composition of evaluations ρ and ρ' is denoted by $\rho.\rho'$ such that $\rho.\rho'(t) = \rho(\rho'(t))$.

A (finite) *path*, π, of an STGA $\mathcal{M}(E)$ is a sequence of consecutive transitions in $\mathcal{M}(E)$. *Runs of a path*, $\rho(\pi)$, is created by applying ρ on π as depicted by rules in Figures 3. Its result is a sequence of ground transitions where there is no guard, *i.e.*, it is true, each state consists of a state of STGA, an evaluation, and concrete event, called message in which each of its parameter is a constant v. A message m is a structure $o(l_1 = v_1, \ldots, l_n = v_n)$ where o is the control part and $\bigcup\{l_i = v_i\}$ represents the data part. A *trace* is a sequence of messages created from a $\rho(\pi)$ by removing all the states without changing the messages order.

$$(SEND)\quad \frac{s \xrightarrow{[\phi]\ o!(l_i = x_i)/A} s'}{s_\rho \xmapsto{o!(l_i = \rho(x_i))} s'_{A.\rho}}\ \rho \models \phi \qquad (INTERACT)\quad \frac{s \xrightarrow{[\phi]\ o.(l_i = x_i)/A} s'}{s_\rho \xmapsto{o.(l_i = \rho(x_i))} s'_{A.\rho}}\ \rho \models \phi$$

$$(RECEIVE)\quad \frac{s \xrightarrow{[\phi]\ o?(l_i = x_i)/A} s'}{s_\rho \xmapsto{o?(l_i = v_i)} s'_{A.\rho.\bigcup\{x_i \mapsto v_i\}}}\ \rho \models \phi \quad \forall v_i \in dom(x_i) \qquad (TAU)\quad \frac{s \xrightarrow{[\phi]\ \tau/A} s'}{s_\rho \xmapsto{\tau} s'_{A\rho}}\ \rho \models \phi$$

Figure 3. Semantics of an STGA

Definition 2: The semantic of a STGA $\mathcal{M}(E)$ is the set of all its traces $[\![\mathcal{M}(E)]\!]$.

Example 3: $m = Response^{[s,c]}!(price{=}2, fee{=}6)$ is a message of the Sq service model in Figure 1(b). A trace of this model is $\langle Request^{[c,s]}?(weight = 3), \tau, \tau, Response^{[s,c]}!(price{=}2, fee{=}6)\rangle$, while the trace $\langle Request^{[c,s]}?(weight = 3), \tau, \tau, Response^{[s,c]}!(price = 2, fee{=}10)\rangle$ is not.

In the sequel, we consider only observables messages, *i.e.*, they are no τ messages and observable traces, *i.e.*, they contain only observable messages, since those are what we observe from the execution of an IUT, *i.e.*, we cannot observe internal activity representing by τ in the model. An observable trace is obtained from $[\![\mathcal{M}(E)]\!]$ by removing τ events while preserving the order of other messages. $[\![\mathcal{M}(E)]\!]$ is overridden as the set of all (observable) traces of model $\mathcal{M}(E)$. *Log* (execution traces) recorded from an IUT will have the format of an observable trace.

III. Property Formalization

In this section we formally define properties. The property represents behaviours the IUT is expected to satisfy. Since behaviours to be tested are usually far fewer than the behaviours of IUT, this approach reduces not only a lot of processing, but also allow the tester to focus on critical behaviours of the IUT [1], [9]. We define *local property* $\mathcal{P}$ to express behaviours to be tested at the level of one service, while *global property* $\overline{\mathcal{P}}$ is used to express collaborations and/or relation between data among services to be tested. The negative version of properties is also presented to guarantee that the IUT *does not* perform some special behaviour which can lead to erroneous behavior. These properties are then checked against execution logs of IUT to emit a testing verdict. Local property checking requires only local log of the corresponding service, while a set of local logs are required for global one.

A. Local Property

Our property is expressed as IF-THEN clause, *e.g.*, *IF context THEN consequence*, *i.e.*, each time a *context* is satisfied then the *consequence* must appear. For example, each time the Sq service in Figure 1 receives a *Request* from the client, then it must respond. Furthermore, message exchanges carry information which can be validated under some condition describing by a boolean expression, *e.g.*, the *fee* response must be equal to the multiplication of the *price* response and the requested *weight*.

A message is an instance of an event (under an evaluation), *i.e.*, an event expresses a class of messages. We use candidate event, which is a pair event/predicate, e/ϕ, to represent a sub-class containing messages which are instances of the event e that are satisfied by ϕ.

Definition 3: A candidate event (CE) is a pair $o(l_1 = x_1, \ldots, l_n = x_n)/\phi(x_1, \ldots, x_n)$, denoted by $o(\bar{x})/\phi(\bar{x})$, where $\phi(x_1, \ldots, x_n)$ is a boolean expression on $\{x_1, \ldots, x_n\}$. A CE $o(\bar{x})/\phi(\bar{x})$ is called to be validated by message $o'(l'_i = v_i)$ iff $o' == o \wedge \bigwedge l_i = l'_i \wedge \rho \models \phi$, with $\rho = \bigcup\{x_i \mapsto v_i\}$.

By extension, we write $o(\bar{x})/\phi(\bar{x}_1, \bar{x})$ to present that this CE may depend on another CE $o_1(\bar{x}_1)/\phi(\bar{x}_1)$. The predicate can be omit if it is true.

Example 4: $CE_1 = Request^{[c,s]}?(weight = x)/(x > 0)$ expresses a class of received *Request* message of the Sq service from the client such that the value of the requested *weight* parameter is positive.

Definition 4: Local property P is described by the form:

$$\mathcal{P} ::= Context \xmapsto{(d)} Consequence \qquad \text{(positive)}$$

$$\neg\mathcal{P} ::= Context \xmapsto{(d)} \neg Consequence \qquad \text{(negative)}$$

where d is a positive integer, *Context* is a sequence of CEs, *e.g.*, $\langle e_1(\bar{x}_1)/\phi_1(\bar{x}_1), \ldots, e_n(\bar{x}_n)/\phi_n(\bar{x}_1, \ldots, \bar{x}_n)\rangle$, and *Consequence* is a set of CEs, *e.g.*, $\{e'_1(\bar{y}_1)/\phi'_1(\bar{x}_1, \ldots, \bar{x}_n, \bar{y}_1), \ldots, e'_m(\bar{y}_m)/\phi'_d(\bar{x}_1, \ldots, \bar{x}_n, \bar{y}_m)\}$.

This definition allows to express that each time when the $Context$ is satisfied then the $Consequence$ must or must not (depending on the formula type, *i.e.*, $\mathcal{P}$ or $\neg\mathcal{P}$) be validated after at most d messages. The $Context$ is satisfied when all of its CEs are satisfied while the $Consequence$ is satisfied when there exists at least one CE which is satisfied, *i.e.*, the $Consequence$ is not satisfied when all of its CEs are not satisfied.

Example 5: Let us take some examples of properties:

$$P_1^s ::= \langle Request^{[c,s]}?(_)\rangle \xmapsto{(1)} \{Response^{[s,c]}!(_)\}$$

$$P_2^s ::= \langle Request^{[c,s]}?(weight{=}y_1)\rangle \xmapsto{(1)} \{Response^{[s,c]}!(price{=}y_2, fee{=}y_3)/(y_3 == y_1 * y_2)\}$$

$$P_1^a ::= \langle Commit^{[s,a]}?(weight{=}x_1, card{=}x_2)\rangle \xmapsto{(1)} \{Withdraw^{[a,b]}!(card{=}y_1, amount{=}y_2)/(y_1 == x_2 \wedge (y_2 == x_1{*}2 \vee y_2 == x_1{*}3))\}$$

The Sq service is guaranteed by the first three properties. P_1^s guarantees that the Sq service always responds to its received request. P_2^s details the property P_1^s by adding a predicate which guarantees that the computation of *fee* in the Sq service is correct. P_1^a guarantees that the Ad service transfers exactly the credit card number from the Sq service to the Bc service and there are two prices 2 and 3 which can be applied.

Negative property is introduced as the reverse of positive one. If a model of the IUT is available, we can easily obtain a positive property which corresponds to a negative one and vice versa. A negative property corresponding to P_1^s is:

$$\neg P_{11}^s ::= \langle Request^{[c,s]}?(_)\rangle \overset{(1)}{\longmapsto} \neg\{Confirm^{[c,s]}?(_), Commit^{[s,a]}!(_)\}$$

This property states that the Sq service has not to receive a *Confirm* or send a *Commit* immediately after receiving a *Request* from the client.

Verification of local properties against log. Given a (potentially infinite) log $log = \langle m_1, \ldots, m_i, \ldots\rangle$, and a property $P = \langle e_1/\phi_1, \ldots, e_n/\phi_n\rangle \overset{(d)}{\longmapsto} \{e'_1/\phi'_1, \ldots, e'_m/\phi'_m\}$, we define the semantics, *i.e.*, the returned verdict, of the property P on the log log by means of an algorithm, not described here by lack of room. The algorithm works as follows. It is based on a *Check* function which takes as inputs the execution log and a property to be checked. In a property, a later CE may depend on a former one, consequently, verification of a message may require the presence of its precedence. Since we can forward-only read data in continuous stream mode, we need to create buffer which contains some fragment of messages stream, what we call a *window*. The created windows contain the first message validating the first CE of the context property and the following messages the $n + d$ next messages. Once a window is created, the verification process on the window can start in parallel with other created windows. In case of a positive property, the verdict *Fail* is emitted only when no message in the d-next messages of the log satisfies any CEs of *Consequence*. The expected verdicts (*Pass* or *Fail*) can be given only when the *Context* is validated. An *Inconclusive* verdict is emitted only if the *Context* cannot be matched. The algorithm needs to collect minimum $n + d$ messages for validating the property (n messages for *Context* and then d messages for *Consequence*). There are maximally $(n + d) + (n + d - 1)$ messages registered in memory.

B. Global Property

The local property is used to express some behaviour of a service and is tested by using only log of the service itself. However, some kinds of fault cannot be detected by local property. Let us take an example by considering the number precision problem in the Ad service of our running example as following. When the Ad service received $Commit^{[s,a]}?(card = x, weight = y)$ from the Sq service, value of y will be rounded to one digit, *e.g.*, the Sq service sends $weight = 4.96$ but the Ad service will consider the received weight as 5. In consequence, if the Ad service is configured with $z_0 = 5$, the price 2 will be applied instead of the price 3. This kind of fault in the Sq service can be detected since its response contains *price*, *e.g.*, by property P_2^s. But, the *Withdraw* event sent by the Ad service does not contain the *price*, this cannot be detected by verifying the relation between *amount* and *weight*. However as required by the choreography model in Figure 2, the price applied on the Ad service has to be the one applied by the Sq service, *i.e.*, if price 3 is applied by the Sq service then it must be also applied on the Ad service and the same holds with price 2. In such a case, local log of only the Ad service is not sufficient. We need to analyse several local properties on a global log to detect such a fault.

In our previous work [4], [10], the global log of the choreography IUT $\mathcal{C}$ was constructed by the synthesis of services local logs of a choreography by assuming the existence of a global clock, *e.g.*, the IUT is running in a cloud. The global clock allows to know the total order of messages from the set of local logs. In this work, our *global log* is just constructed by grouping local logs in a set. Since no synthesis is required, we do not need global clock.

Definition 5: The global property is described wrt. the following grammar:

$$\overline{\mathcal{P}} ::= SET \Longmapsto SET' \mid \neg\overline{\mathcal{P}} ::= SET \Longmapsto \neg SET'$$

where SET and SET' are two set of local properties.

Verification of global properties on global log. We firstly formalize global log as follows:

Definition 6: Let $log_1, \ldots, log_n$ be n local logs recorded from n different services. We define the global log, $\overline{\log}$, of these n services as the set of their logs. Local log of service i in global log is given by $\overline{\log}\lfloor_i$.

The semantics of global property $\overline{P} = SET \Longmapsto SET'$ on global log $\overline{\log}$ are presented in Table II.

Example 6: We use a global property as defined below to guarantee that price 2 is being applied by both Sq and Ad services.

$$\begin{aligned}\overline{P}_1 ::= &\{\langle Request^{[c,s]}?(weight = x_1)\rangle \overset{(1)}{\longmapsto} \\ &\{Response^{[s,c]}!(price = x_2, fee = x_3)/(x_3 == x_1 * 2)\}\} \\ \Longmapsto &\{\langle Commit^{[s,a]}?(weight = y_1, card = y_2)\rangle \overset{(1)}{\longmapsto} \\ &\{Withdraw^{[a,b]}!(card = y_3, amount = y_4)/ \\ &(y_2 == y_3 \wedge y_4 == y_1 * 2)\}\}\end{aligned}$$

IV. Implementation and Evaluation

The architecture of our online verification system is depicted on Figure 4 [1]. Each local tester is attached to a service to be tested. Local tester will collect input/output messages of its service at a point of observation (PO) which is put at a position such that all exchanged messages from/to the service are captured. Based on collected log, the local tester will verify its local properties and will give local verdict. Since the global properties consist of elements which have the same format with local properties so we use local testers to verify these elements, then the results will be used by the global tester to emit a verdict for global properties based on Table II. For such reasons, in this section we focus on the implementation of local tester.

Our framework is detailled in a stepwise manner:

1) *Properties definition*. Standards or protocol experts provide the implementation behaviors to be tested, which

[1]The tool is available at http://www.lri.fr/~nhnghia/prop-tester/

Table II
CHECKING $\overline{P} ::= SET \Longmapsto SET'$ AND $\neg\overline{P} ::= SET \Longmapsto \neg SET'$ ON GLOBAL LOG $\overline{\log}$

$Check(\overline{\log}, \overline{P})$	$Check(\overline{\log}, \neg\overline{P})$	Condition
Pass	*Fail*	$\forall P_a \in SET : Check(\overline{\log}\lfloor_a, P_a) = Pass \Rightarrow \exists P_{a'} \in SET' : Check(\overline{\log}\lfloor_{a'}, P_{a'}) = Pass$
Fail	*Pass*	$\forall P_a \in SET : Check(\overline{\log}\lfloor_a, P_a) = Pass \Rightarrow \nexists P_{a'} \in SET' : Check(\overline{\log}\lfloor_{a'}, P_{a'}) = Pass$
Inconclusive	*Inconclusive*	$\exists P_a \in SET : Check(\overline{\log}\lfloor_a, P_a) \neq Pass$

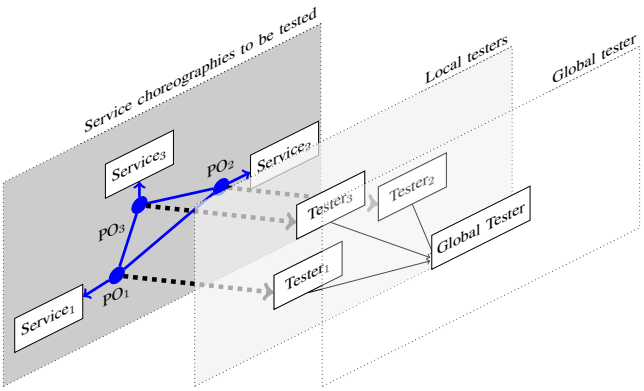

Figure 4. Architecture of the Verification System

are formulated as property according to the Definition 4 and 5.

2) *Correctness of property*. If specification model of the IUT is available, then the properties will be then formally verified on the model guaranteeing that they are correct wrt. the requirements.
3) *Translating property into XQuery*. A property is translated into an XQuery such that it returns *false* iff the property is violated, and *true* iff the property is validated. The *Inconclusive* verdict of the property will be emitted by the tester when the end of stream of the log is reached without any delivered verdict.
4) *Extraction of execution traces*. An observer is put at each service level to sniff all of its (input and output) messages exchanged with its partners. Each time the observer captures a message, if the message is related to the properties to be tested, then it is sent through an opened pipeline between the tester and the PO to the tester, where it will be verified by an XQuery processor.
5) *Properties tested on the execution traces*. The properties tested in XQuery form will be executed by MXQuery[2] processor on the XML stream supplying by the observer. Based on result of the query, the verdict (*Pass*, *Fail*, or *Inconclusive*) will be emitted.

[2]http://mxquery.org

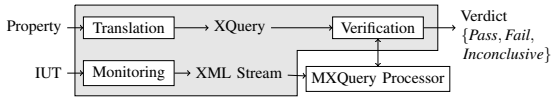

Figure 5. Online verification of local property

In this verification, the step 1, 2 and 3 are done only one time, while the step 4 and 5 are done in a continuous way online. The Figure 5 exhibits the steps 3,4, and 5. In the sequel, we will present all the steps of the framework in the order, except that step 4 will be explained before step 3 since the translation into XQuery depends on the format of the log file. Moreover, we will present some experiments executed on the service choreography presented in the example of Section II.

A. Correctness of property

Correctness of a local property. When a model of a service is available, we suppose that the service (the implementation) must conform to its model, *i.e.*, the logs of the service must conform to its traces. The correctness of local property wrt. a service model is guaranteed by the following definition.

Definition 7: Let $\mathcal{M}^S$ be a service model and P be a property, we say P is correct (return true) wrt. $\mathcal{M}^S$ if:

- $\forall tr \in [\![\mathcal{M}^S]\!]$, $Check(tr, P) \in \{Pass, Inconclusive\}$, and
- $\exists tr \in [\![\mathcal{M}^S]\!]$, $Check(tr, P) = Pass$

Since we cannot always compute all traces of $\mathcal{M}^S$, *i.e.*, it may be infinite due to the unboundedness of the model equipped with loops. To overcome this issue, we put limit k which will cut the length of paths. For the data types, we avoid the state space explosion by avoiding the unfolding of the model, the obtained paths are symbolic paths, *i.e.*, no concrete values are given to variables. The verification of the correctness is performed by the Algorithm 1. We firstly find all paths of $\mathcal{M}^S$ being compatible with the events of the *context* of P. Two events are compatible iff they have the same control part and list of parameter labels. For each found path, we add the predicates of each CE of P to the guard of the transition of the compatible event with the one of the CE if this is a sending, while for the reception the predicate is added in the next transition. One of the two special transitions are added to each path, line 13 and 24, to distinguish if the path is or not compatible also with the *consequence* of P. Z3[3] SMT solver is used, line 26–27 to determine whether the cumulated predicates are satisfiable, *i.e.*, it exists a set of instances (values) for the variables of the path.

[3]http://research.microsoft.com/en-us/um/redmond/projects/z3/

Algorithm 1: Checking correctness of local property

Data: an STGA $\mathcal{M} = (S, s_0, T)$, a number $k > 0$, and a Property $P = \langle e_1/\phi_1, \ldots, e_n/\phi_n \rangle \overset{(d)}{\longmapsto} \{e'_1/\phi'_1, \ldots, e'_m/\phi'_m\}$

Result: true/false (property P correct/incorrect)

1 Get all paths Π_1 of $\mathcal{M}$: each starts from s_0, and its n last observable events are compatible with $\langle e_1, \ldots, e_n \rangle$, and its length isn't greater than k;
2 **if** $(\Pi_1 == \emptyset)$ **then** **return false** ; `// not found any path`
3 $\Pi_3 := \emptyset$;
4 **foreach** $\pi_1 \in \Pi_1$ **do**
5 $\quad \pi'_1$ is minimum path at the end of π_1 compatible with $\langle e_1, \ldots, e_n \rangle$;
6 $\quad \varphi :=$ **true**; $\quad k := 1$; `// for each CE in context`
7 $\quad$ **foreach transition** $tr_j = s \xrightarrow{[\phi]\, e/A} s' s \xrightarrow{[\phi]\, e/A} s' \in \pi'_1,\ j = 1$ **to** $length(\pi'_1)$ **do**
8 $\quad\quad$ **if** e *is compatible with* e_k **then**
9 $\quad\quad\quad \varphi := \varphi \wedge \bigwedge(x_i == y_i)$, with e is $o(l_i = x_i)$ and e'_k is $o(l_i = y_i)$
$\quad\quad\quad$ **if** e *is reception* **then** update tr_j with $\phi := \phi \wedge \varphi$; $\quad \varphi := \phi_k$;
10 $\quad\quad\quad$ **else** update tr_j with $\phi := \phi \wedge \phi_k \wedge \varphi$; $\quad \varphi :=$ **true**;
11 $\quad\quad k := k + 1$;
12 $\quad s \xrightarrow{[\phi]\, e/A} s'$ is the last transition of π ; $\quad$ // $\Rightarrow e \equiv e_n$
13 $\quad$ Get all paths Π_2 of $\mathcal{M}$: each starts from s', its length is not greater than d, and the its last event is in $\{e'_1, \ldots, e'_m\}$;
14 $\quad$ **if** $(\Pi_2 == \emptyset)$ **then** $\Pi_3 := \Pi_3 \cup \{\pi_1 \frown (s' \xrightarrow{[\varphi]\, \times} s_\times)\}$;
15 $\quad$ **else** **foreach** $\pi_2 \in \Pi_2$ **do**
16 $\quad\quad k := 1$; `// for each CE in consequence`
17 $\quad\quad$ **foreach transition** $tr_j = s \xrightarrow{[\phi]\, e/A} s' \in \pi_2,\ j = 1$ **to** $length(\pi_2)$ **do**
18 $\quad\quad\quad$ **if** e *is compatible with* e'_k **then**
19 $\quad\quad\quad\quad \varphi := \varphi \wedge \bigwedge(x_i == y_i)$, with e is $o(l_i = x_i)$ and e'_k is $o(l_i = y_i)$
20 $\quad\quad\quad\quad$ **if** e *is reception* **then** update tr_j with $\phi := \phi \wedge \varphi$; $\varphi := \phi_k$;
21 $\quad\quad\quad\quad$ **else** update tr_j with $\phi := \phi \wedge \phi'_k \wedge \varphi$; $\quad \varphi :=$ **true**;
22 $\quad\quad\quad k := k + 1$;
23 $\quad\quad \Pi_3 := \Pi_3 \cup \{\pi_1 \frown \pi_2 \frown (s'' \xrightarrow{[\varphi]\, \checkmark} s_\checkmark)\}$;
24 $b :=$ **false**;
25 **foreach** $\pi_3 \in \Pi_3$ **do**
26 $\quad$ **if** *exist an evaluation* ρ *s.t. state* $s_\times$ *exists in* $\rho(\pi_3)$ **then** **return false**;
27 $\quad$ **else** **if** *exist an evaluation* ρ *s.t. state* $s_\checkmark$ *exists in* $\rho(\pi_3)$ **then** $b :=$ **true**;
28 **return** b;

The correctness of global property. Since the global property consists of local properties whose correctness are verified on local traces, we override the projection function "$\downharpoonleft$" applied for projecting global trace to local traces.

Definition 8: Let $\mathcal{M}^C$ be a choreography model, and $tr \in [\![\mathcal{M}^C]\!]$. We override the projection of tr on service a as:

$$proj(o^{[a,b]}.(l_i = x_i), d) = \begin{cases} \langle o^{[a,b]}!(l_i = x_i) \rangle & \text{if } d = a \\ \langle o^{[a,b]}?(l_i = x_i) \rangle & \text{if } d = b \\ \langle\rangle & \text{otherwise} \end{cases}$$

$$tr\downharpoonleft_a = \begin{cases} \langle\rangle & \text{if } length(tr) = 0 \\ proj(head(tr), a) \frown tail(tr)\downharpoonleft_a & \text{otherwise} \end{cases}$$

Definition 9: Given a choreography model $\mathcal{M}^C$, a set of n service models $\{\mathcal{M}^S_1, \ldots, \mathcal{M}^S_n\}$, and a global property $\overline{P} = SET \mapsto SET'$. $\overline{P}$ is correct wrt. $\mathcal{M}^C$ iff:

- $\forall P_a \in SET \cup SET'$ of service a having model $\mathcal{M}^S_a$, $\exists tr \in [\![\mathcal{M}^S_a]\!]$, $Check(tr, P_a) = Pass$
- $\forall tr \in [\![\mathcal{M}^C]\!]$, $Check(tr, \overline{P}) \in \{Pass, Inconclusive\}$
- $\exists tr \in [\![\mathcal{M}^C]\!]$, $Check(tr, \overline{P}) \in \{Pass\}$

In the definition above, the two last conditions can be also verified based on Algorithm 1 where the notion compatible event is modified as follows. A sending or reception event is compatible with an interaction if it is compatible with the projection of the interaction.

B. Extraction of Execution Traces

We extend our monitor in [4] to collect SOAP messages exchanging among services of choreography. There are several monitoring frameworks for Web services, *e.g.*, [11]–[13] Each service is attached by one observer at a point of observation such that it can capture all SOAP messages from and to its service. We then put body parts of captured message side by side as an example in Example 7 in which, we add also *tstamp* attribute representing time stamp at the capturing moment. This *tstamp* allows us to verify time condition, *e.g.* timeout condition in our example where after the *Response* event the *Confirm* event can only happen before one hour.

Example 7: Captured log of the Sq service.

```
<message source="c" destination="s" direction="reception"
   name="Request" tstamp="1">
    <weight>3</weight>
</message>
<message source="s" destination="c" direction="sending"
  name="Response" tstamp="3">
    <price>2</price>     <fee>6</fee>
</message>
...
```

C. Translating Property into XQuery

The SOAP messages exchanged between Web services are in XML format. One can refer to record execution traces (*log*) as an XML document to take advantage of standardized XML tools, *e.g.*, XML Query Language (XQuery), to analyze it. XQuery is a language for finding and extracting elements and attributes from XML data. We use window clauses in XQuery to slice the log into segments called window. The log of the IUT can be considered as a data stream consisting of continuous messages with time-varying arriving and unpredictable rates. Hence, the log processing requires real-time treatment, fast mean response time, and low memory consumption.

The query in Example 8 represents a local property of Example 5 (P^s_2). Let us note that the translation of the property into Xquery is done automatically, the algorithm for lack of room is not depicted here. Line 2–5 creates a sequence of windows, each window is represented by a variable $win. The variable $stream points to a log in stream mode and $stream//message is used to denote all message elements in the log. A window $win is a sub sequence of $stream for which the *start* and *end* conditions are applied. XQuery uses XPath syntax to express specific parts of an XML element. Our window starts at a message such that its destination is the Sq service, *i.e.*, "s", its name is "Request'. The window size is 2. The size can be less than 2 if we reach the end of $stream. Each created window realized by an Xquery from line 2–5 is encapsulated by another Xquery and will be referenced by $w, which has in charge to perform the

checking of the property of the created window (Line 7–8). For the example, at Line 7 it verifies whether a message in window $w such that its position is greater than 1 and it satisfied the conditions in line 8. If both holds a *Pass* verdict is emitted.

Example 8: Transforming property P_2^s into XQuery

```
1  for $w in (
2   for sliding window $win in $stream//message
3   start $s at $spos when $s/@name eq "Request" and $s/
      @direction eq "reception" and $s/@receiver eq "s"
4   end $e at $epos when $epos − $spos eq 1
5   return <window>{$win}</window> ) return
6  for $e1 in $w/message[1] return
7   (some $e11 in $w//message[position() > 1] satisfies
8    $e11/@name eq "Response" and $e11/@source eq "s" and
       $e11/@direction eq "sending" and number($e11/fee)
       eq number($e11/price) ∗ number($e1/weight) )
```

D. Evaluation

To evaluate the performance of our tools, we realized a series of experiments. For the verification of each property in Example 5, we generate 50 logs files for each length 1,000, 2,000, 5,000, and 20,000 messages. Each of these logs contain a randomly created sequence of the messages corresponding to the shipping-quotation service and the Ad service for the test of properties P_1^s, P_2^s, and P_1^a respectively. Generated log is then sent as an XML stream to our tool, as described in Figure 5. Since the performance of our tool depends on the one of the MXQuery processor, we do not intend to benchmark MXQuery processors, but rather to get early experiences of online verification of our approach. Figure 6 shows the average of time processing (milliseconds). These experiments have been realized on a Macbook Air–Mid 2011 laptop with a CPU 1.7GHz Core i5 and 4GB of RAM. These results show that the maximum time processing per message are 2.8 milliseconds. Furthermore, the time decreases when length of log increases. This tends to show that our framework can actually be done in real time.

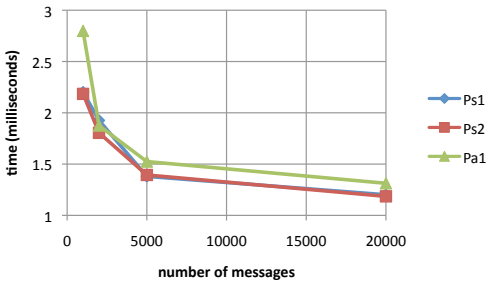

Figure 6. Tester Scalability

V. Conclusion

We have presented a formal framework to perform online verification of service choreographies based on property-oriented passive testing, where not only control part of interactions between services are tested but also its value passing. In our framework, local or global properties are verified against a formal model if it exists. Afterwards, they are tested on a local log of a service for local properties. As global properties are defined as a set of local properties, they are used to verify some behaviour at global level of the choreography, in a global log. Our framework is fully tool supported by taking advantage of XQuery engine to verify message exchanges online which are formatted as an XML stream, and by taking advantage also of Z3 SMT solver to avoid state explosion problem when validating properties against model with value-passing. As future work, on one hand, we would like to obtain service requirements (service models for instance) from a choreography model by using some projection function; and on the other hand, we intend to introduce more abstraction in the definition of our properties.

References

[1] B. T. Ladani, B. Alcalde, A. Cavalli, and B. T. Landi, "Passive Testing - A Constrained Invariant Checking Approach," in *Proc. of TESTCOM'05*, 2005, pp. 9–22.

[2] E. Bayse, A. R. Cavalli, M. Núñez, and F. Zaïdi, "A Passive Testing Approach based on Invariants: Application to the WAP," *Computer Networks*, vol. 48, no. 2, pp. 235–245, 2005.

[3] 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.

[4] H. N. Nguyen, P. Poizat, and F. Zaïdi, "Passive Conformance Testing of Service Choreographies," in *Proc. of SAC'12*, 2012.

[5] C. Andrés, M. Emilia Cambronero, and M. Núñez, "Passive Testing of Web Services," in *Proc. of WS-FM'10*, 2010.

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

[7] 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.

[8] Z. Li and H. Chen, "Computing Strong/Weak Bisimulation Equivalences and Observation Congruence for Value-Passing Processes," in *Proc. of TACAS99*, 1999.

[9] S. Li, J. Wang, W. Dong, and Zhi-Chang Qi, "Property-Oriented Testing of Teal-Time Systems," *Proc. of APSEC'04*, 2004.

[10] F. Zaïdi, E. Bayse, and A. Cavalli, "Network Protocol Interoperability Testing based on Contextual Signatures and Passive Testing," in *SAC '09*, 2009.

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

[12] 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*, vol. 2, no. 3, pp. 223–244, 2009.

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