Online Testing Framework for Web services

Dung Cao, Patrick Félix, Richard Castanet, Ismail Berrada

Online Testing Framework for Web Services Composition

Tien-Dung Cao[1], Patrick Félix[1], Richard Castanet[1] and Ismail Berrada[2]
[1]LaBRI - CNRS - UMR 5800, University of Bordeaux 1
351 cours de la libération, 33405 Talence cedex, France.
Email: {cao,felix,castanet}@labri.fr
[2]L3I, La Rochelle University, 17042 La Rochelle, France.
Email: ismail.berrada@univ-lr.fr

***Abstract*—Testing conceptually consists of three activities: test case generation, test case execution and verdict assignment. Using online testing, test cases are generated and simultaneously executed. This paper presents a framework that automatically generates and executes tests "online" for conformance testing of the Web service composition described in BPEL. The proposed framework considers unit testing and it is based on a timed modeling of BPEL specification, a distributed testing architecture and an online testing algorithm that generates, executes and assigns verdicts to every generated state in the test case.**

***Keywords*-Web Services Composition, BPEL, Test Generation, Timed Extended Finite State Machine, Online Testing, Conformance Testing.**

I. Introduction

Conformance testing is a commonly used activity to improve the system reliability. It is a very time-consuming process that requires a great deal of expert knowledge of the systems being tested. The use of formal specifications provides a support for automating this process.

The ways to test a system can be classified into two basic groups. The most natural way, namely the active testing approach, consists of carrying out the test derivation from specifications. Test cases are then executed against the system under test and verdicts are assigned. Another possibility is the passive testing approach. The absence of observations allows only the validation of traces, and thus this approach checks that a trace of an implementation is a valid execution of the specification.

Web services testing is a kind of black-box testing where test cases are designed based on the interface specification of the Web services under test, not on its implementation. In this case, the internal logic does not need to be known at all, whether it's coded in Java, C# or BPEL. On the contrary, business process unit (a Web services composition) testing can be classified in two kinds:

- ***White-box*** approach. As BPEL is an executable language, the BPEL description of Web services composition is considered as the source code of the composition. It can be executed by any BPEL engine (Active BPEL, Oracle...). Test cases are then designed based on the internal logic of the service under test.
- ***Gray box*** approach (we call gray-box to difference with black-box testing of Web services because we know that are the interaction between a service under test and its partners). In this approach, a composite Web service can be coded in a different language from the specification, for instance, a BPEL specification is coded as a Java program (even BPEL). An implementation of the composite Web service is then tested without any information of its internal structure. Test cases are then generated only from the specification (WSDL and BPEL).

Testing conceptually consists of three activities: test case generation, test case execution and verdict assignment. Using online testing, test cases are generated and simultaneously executed [22]. In this paper, we focus on model-based unit testing [2] of Web services composition given by BPEL specifications. We consider conformance testing using the gray-box approach . We present a framework that automatically generates and executes tests "online" for Web service composition described in BPEL. In order to model the BPEL behaviors, the timing constraints, and data variables, the BPEL specification is transformed into the Timed Extended Finite State Machines (TEFSM) model. This formalism is closely related to timed automata [18] and permits to carry out timing constraints, clocks, state invariants on clocks and data variables for BPEL message. From this model, we propose an online testing algorithm to automatically generate test cases and simultaneously execute them to issue verdicts based on distributed testing architecture that is also proposed in this framework.

The rest of paper is organized as follows. Section II reviews some previous works on Web services composition testing. In section III, we give the definition of TEFSM that is used to model BPEL process, and some related notations. Section IV describes the relationship between BPEL concepts and TEFSM. Section V presents a framework that includes a timed modeling of BPEL specification, a test architecture, and an online testing algorithm. Finally,

section VI concludes the paper.

II. Related works

In the last years, several techniques and tools have been developed to test Web services. Various approaches for service composition testing were analyzed by [2] including unit testing, integration testing, black-box testing and white-box testing of choreographies and orchestrations.

- **Testing BPEL descriptions**. Jose Garcia-Fanjul et al [6] use the SPIN model checker to generate test cases specification for compositions given in BPEL. In order to systematically derive test suites, the transition coverage criterion is considered. Yongyan Zheng and Paul Krause [7, 9] model each BPEL activity by an automaton (also referred as Web Service Automaton). These automata are then transformed into Promela, the input format of the SPIN model checker. [10] use one more time the SPIN model checker to verify BPEL specification. However, the authors do not transform BPEL directly into Promela as in [6]. BPEL will be translated to guard conditions which are transformed to Promela. In all of these methods, test cases are generated from counterexamples given by the SPIN model checker. Transforming BPEL into Intermediate Format Language (IF) is presented in [14] and [15]. Timed test cases are generated using TestGen-IF tool [28]. [26, 27] present also a framework for white-box testing . However, the authors do not consider automatic test case generation [15].
- **Testing WSDL descriptions**. [4] present a test cases generation approach based on WSDL descriptions. R. Heckel and L. Mariani in [20] also present a test cases generation approach based on the definition of rules for each service. Some of open source tools for testing WSDL descriptions of Web services were developed such as soapUI [30] and TestMaker [31]. In [3], the authors propose a framework that extends UDDI (Universal Description, Discovery and Integration) registry role from the current one of a passive service directory, to sort of an accredited testing organism, which validates service behaviour before actually registering it.

Regarding online testing framework, [22, 23] present the T-Uppaal tool for model based testing of embedded real-time systems. T-Uppaal automatically generates and executes tests from the state machine model of the implementation under test (IUT) and assumes that environment specifies the required and allowed observable (real time) behaviors of the IUT. Finally, in [23] and [25], the authors introduce the TorX tool which is based on a timed extention of the ioco testing theory. These works are interested in input/output action and timing constraints more than the messages value (data). However, we can not use these frameworks for the web service composition testing because they do not support the SOAP message. Moreover, its input format (timed automata, Promela ...) do not support data variable or its data type is very poor. Another reason is: they can not also simulate a partner as a web services.

III. Preliminaries

BPEL specification can be described by means of formal models such as TEFSM [13, 18] (Timed Extended Finite State Machine). In this section, we introduce the TEFSM model and some related definitions.

Clocks and Constraints: A clock is a variable that allows to record the passage of time. It can be set to a certain value and inspected at any moment to see how much time has passed. Clocks increase at the same rate, they are ranged over $\mathbb{R}^+$, and the only assignments allowed are clock resets of the form *c:=0*. For a set *C* of clocks, and a set *V* of variables, the set of clock constraints Φ(C) is defined by the grammar: $\Phi := \Phi_1 | \Phi_2 | \Phi_1 \wedge \Phi_2, \Phi_1 := c \leq m, \Phi_2 := n \leq c$ where *c* is a clock of *C*, and *(n, m)* are two natural numbers. P(V) is a set of linear inequalities on V. Next, a n-tuple $(c_0, c_1, ..., c_n\ )$ (resp. $(v_0, v_1, ..., v_m)$) will be denoted by $\vec{c}$ (resp. $\vec{v}$).

Definition 1: (TEFSM): A TEFSM M is a tuple, M = (S, s_0, S_f, V, E $\cup\{\tau\}$, C, Inv, T) where:

- *S* = $\{s_0, s_1, ..., s_n\}$, is a finite set of states;
- $s_0 \in$ S is an initial state;
- S_f = $\{s_{f0}, s_{f1}, ..., s_{fm}\}$, is a finite set of final states;
- *V* is a finite set of data variables, $D_V^{|V|}$ is the data variable domain of V;
- *E* is a finite set of the events. *E* is partitioned into:
  - Input event of the form *?pl.op.msg*: the reception of the message (*msg*) for the operator (*op*) from the partner (*pl*);
  - Output event of the form *!pl.op.msg*: the emission of the message (*msg*) for the operator (*op*) to the partner (*pl*);
- τ is the internal event.
- *C* is a finite set of clocks including a global clock (never reset);
- *Inv*: S $\mapsto$ Φ(C) is a mapping that assigns a time invariant to states;
- $T \subseteq S \times E \times P(V) \vee \Phi(C) \times 2^C \times \mu \times S$ is a set of transitions where:
  - $P(\vec{v})\&\phi(\vec{c})$: are guard conditions on data variables and clocks;
  - $\mu(\vec{v})$: Data variable update function where $\mu : V \longmapsto D_V^{|V|}$;
  - $X \subseteq 2^C$: Set of clocks to be reset;

A transition $t = (s < e, [g], \{f; c\} > s') \in T$ represents an edge from state s to state s' on event *e*. g is a set of constraints over clocks and data variables, f is a set of data

update, and c is a set of clocks to be reset.

Definition 2: (Partial of TEFSM): Let M be a TEFSM. The partial machine of M is defined by PM = (S, s_{in}, S_{out}, V, E, C, Inv, T) where: (S, s_{in}, V, E, C, Inv, T) is a TEFSM and $S_{out} \subset$ S.

A partial of TEFSM is a TEFSM extended by input state s_{in} (representing the entering state of the partial machine and which replaces the initial state s_0) and a set of output states, S_{out} (representing the exit state of the partial machine).

IV. Relationship between BPEL concepts and TEFSM

BPEL [1] provides constructs to describe complex business processes that can interact synchronously or asynchronously with their partners. A BPEL process always starts with the *process* element (i.e the root of the BPEL document). It is composed of the following children: *partnerLinks, variables, activities* and the optional children: *faultHandlers, eventHandlers, correlationSets*. These children are concurrent. In our framework, we only model the activities of a BPEL process because we are only interested in the input/output messages and the conditions specification of input variables. An internal fault will be assigned a fail verdict. We use a *stop* variable for activities machine to terminate (assign to *true*) the rest activities if the termination is activated by an *exit* activity or the *throw* activity. The *scope* activity will be modeled as a *process*.

A. Messages

A BPEL variable is always connected to a message from a WSDL description of partners. In BPEL, a Web service that is involved in the process is always modeled as a *porType* (i.e. abstract group of operations (noted *op*) supported by a service). These operations are executed via a *partnerlink* (noted by *pl*). In our formalism, for instance, the input message *?pl.op.v* denotes the reception of the message *op(v)* (constructed from the operation *op* and the BPEL variable *v*) via the channel *pl*.

B. Basic activities

A basic activity can be one of the following: *receive, reply, invoke, assign, wait, empty, exit, throw*. Each basic activity is described by a partial machine (_ denotes the event of transition belong to $\{\tau\}$).

Receive Activity: <receive partnerLink=pl portType=pt operation=op variable=msg>

PM = ($\{s_{in}, s_{out}\}$, s_{in} , $\{s_{out}\}$, {*v, stop*}, {*?pl.op.msg*}, {c}, $\{(s_{in}, true), (s_{out}, true)\}$, $\{t_1\}$)

- t_1=(s_{in},<*?pl.op.msg,[stop=false],*{*c,v=msg*}>,s_{out})

Reply Activity: <reply partnerLink=pl portType=pt operation=op variable=msg>

PM = ($\{s_{in}, s_{out}\}$, s_{in} , $\{s_{out}\}$, {*stop*}, {*!pl.op.msg*}, {c}, $\{(s_{in}, true), (s_{out}, true)\}$, $\{t_1\}$)

- t_1=(s_{in},<*!pl.op.msg,[stop=false],*{c}>,s_{out})

Assign Activity: <assign> <from> v_2 </from> <to> v_1 </to> ... </assign>

PM = ($\{s_{in}, s_{out}\}$, s_{in} , $\{s_{out}\}$, {v_1,v_2,...,v_n,*stop*}, $\emptyset$, {c}, $\{(s_{in}, true), (s_{out}, true)\}$, $\{t_1\}$)

- t_1=(s_{in},< _ ,*[stop=false],*{*c*, v_1=v_2,...}>,s_{out})

Wait Activity: <wait (for=d | until=dl)>.

- <wait for=d>: PM = ($\{s_{in}, s_{out}\}$, s_{in} , $\{s_{out}\}$, {*stop*}, $\emptyset$, {c}, $\{(s_{in}$,c$\leq$d),$(s_{out}, true)\}$, $\{t_1\}$)
  - t_1=(s_{in},< _ ,*[c=d & stop=false],*{c} >,s_{out})
- <wait until=dl>: PM = ($\{s_{in}, s_{out}\}$,s_{in} , $\{s_{out}\}$, {stop}, $\emptyset$, {gc}, $\{(s_{in}$,gc$\leq$dl), $(s_{out}, true)\}$, $\{t_1\}$)
  - t_1=(s_{in},< _ ,*[gc=dl & stop=false],*$\emptyset$ >,s_{out})

Throw Activity: <throw faultName=fault/>

PM = ($\{s_{in}, s_{out}\}$, s_{in}, $\{s_{out}\}$, {*stop*}, $\emptyset$, $\emptyset$, $\{(s_{in}, true)$, $(s_{out}, true)\}$, $\{t_1\}$)

- t_1=(s_{in},<*!fault,[],*{*stop=true*}>,s_{out})

Exit Activity: <exit/>

PM = ($\{s_{in}, s_{out}\}$, s_{in} , $\{s_{out}\}$, {*stop*}, $\emptyset$, $\emptyset$, $\{(s_{in}, true)$, $(s_{out}, true)\}$, $\{t_1\}$)

- t_1=(s_{in},< _ ,*[],*{*stop=true*}>,s_{out})

Invoke Activity: <invoke partnerLink=pl portType=pt operation=op inputVariable=msg_in outputVariable=msg_out>

PM = ($\{s_{in}, s_1, s_{out}\}$, s_{in} , $\{s_{out}\}$, {*v_in, v_out, stop*}, {*!pl.op.msg_in, ?pl.op.msg_out*}, $\{c\}$, $\{(s_{in}, true)$, $(s_1, true)$ $(s_{out}, true)\}$, $\{t_1, t_2\}$)

- t_1=(s_{in},<*!pl.op.msg_in,[stop=false],*{c}>,s_1)
- t_2=(s_1,<*?pl.op.msg_out,[stop=false],*{c, *v_out=msg_out*}>,s_{out})

Empty Activity: <empty/>

PM = ($\{s_{in}, s_{out}\}$, s_{in} , $\{s_{out}\}$, {*stop*}, $\emptyset$, {c}, $\{(s_{in}, true)$, $(s_{out}, true)\}$, $\{t_1\}$)

- t_1=(s_{in},< _ ,*[stop=false],*{c}>,s_{out})

C. Structured activities

Structural activities are the *sequence, while, switch, flow, pick, repeatUntil, if* and *scope*. They take some partial machines $PM_{i,i\in[0,n]}$ and combine them to have a new partial machine. The partial machines (set of states and transitions) of structural activities (*sequence, while, switch and pick*) are shown in Fig 1. The set of variables, events, clocks are:

$$V = \{v_0, .., v_m\} \cup V_0 \cup ... \cup V_n,$$

$$E = \{e_0, .., e_l\} \cup E_0 \cup ... \cup E_n,$$

$$C = C_0 \cup ... \cup C_n$$

In our framework, the *repeatUntil* activity will be modeled as a *while* activity. The conditional behavior *if* will be

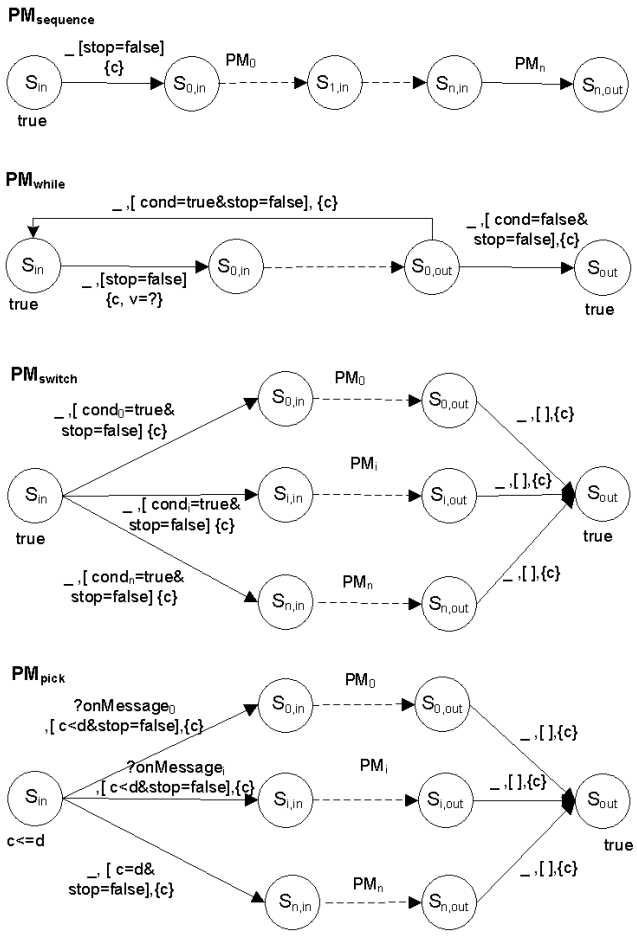

Figure 1. Modeling structural activities

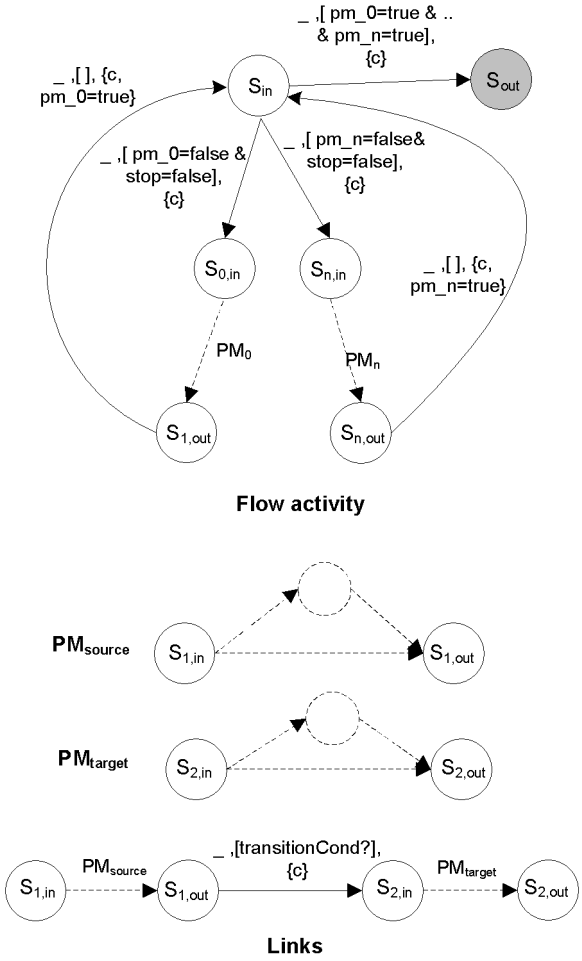

Figure 2. Modeling Flow activity and Link variables

also modeled as a *switch* activity, and the *eventHandler* activity will be model as a *pick* activity. The *flow* activity allows describing one or more concurrent activities [1]. It specifies the parallel execution of flow corresponding to partial TEFSMs. The partial machine of a *flow* finishes when all of its sub-partial machine finish. So when a sub-partial machine finishes, it changes into a temporary state to wait the rest of sub-partial machine finish before moving into s_{out}. We choice this temporary state is s_{in} of *flow*'s partial machine. This is easy to check the output actions of a flow activity from service under test and carry out its by a sequence because the test framework can work as a sequence upon time order while it receives the parallel requests. We use a boolean variable for each sub-partial machine to examine the termination of each machine. The initial value of these variables is *false*. The *links* defined in the *flow* activity allow to enforce precedence between these activities, i.e. it allow synchronization. We add a transition to enforce precedence between these partial machines. The partial machine of a flow activity and link variables are shown in Fig 2.

D. Limitations

Our framework has the following limitations: The attributes *joinCondition*, *supressJoinFailure* of the *flow* activity are not treated. An activity with *correlation* property will be modeled by adding a variable *status* of properties as in [14]. In many case, may be we do not use the *correlation* property because we test only single session.

V. Online Testing Framework

A. Conformance Relation

In order to capture the notion of conformance between implementations under test (IUT) of a Web service and its specification, we will use a conformance relation. Before doing this, we need some additional notations:

- The *action* transition (discrete transition), denoted by $s \xrightarrow{a} s'$ such that $s \xrightarrow{a[g]\{u\}} s' \in T$, indicates that if the guard (on variables and clocks) g is true, then the automaton fires the transition by executing the input/output action a, changing the current values of the data variables by executing all assignments, changing

the current values of the clocks and timers by executing all time setting/resetting, updating the buffers contents of the system by consuming the first signal required by input actions and by appending all signals required by output actions, where u is a set of update functions, and moving into the next state s'.

- The *timestamps* transition (timing transition), denoted by $s \overset{c=d}{\rightarrow} s'$ where c is a local clock (or $s \overset{gc=dl}{\rightarrow} s'$ where gc is a global clock), indicates that TEFSM will move to state s' when local clock c reaches a time duration d (or global clock gc reaches a deadline (dl)), for example, the transition of the *wait* activity.

For $a \in E$, we write $s \xrightarrow{a}$, iff $\exists s' \in S$ such that $s \xrightarrow{a} s'$. We write $s \overset{a_1,...,a_n}{\longrightarrow} s'$ iff $\exists s_1, s_2, ..., s_{n-1} \in S$ such that $s \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 ... s_{n-1} \xrightarrow{a_n} s'$. We write $s \overset{a}{\Rightarrow}$, iff $\exists s', s'', s''' \in S$ such that $s \overset{\tau,...,\tau}{\longrightarrow} s'' \xrightarrow{a} s''' \overset{\tau,...,\tau}{\longrightarrow} s'$. We define $\Gamma = (E \times \mathbb{R}_+)$ as the set of observable actions (actions and delays) and $\Gamma_\tau = (E_\tau \times \mathbb{R}_+)$ as the set of observable actions including internal actions (i.e. $E_\tau = E \cup \{\tau\}$). A *timed sequence* $\sigma \in Seq(\Gamma)$ is composed of actions a and non-negative reals d such that: $s \overset{(a,d)}{\rightarrow} s' \oplus d$, where d is a timing delay. Let $\Gamma' \subseteq \Gamma$ and $\sigma \in Seq(\Gamma)$ be a timed sequence. $\pi_{\Gamma'}(\sigma)$ denotes the projection of σ to Γ' obtained by deleting from σ all actions not present in Γ'. The *observable timed traces* of an IUT is defined by: $Trace(IUT) = \{\pi_\Gamma(\sigma) | \sigma \in Seq(\Gamma_\tau) \wedge s_0 \overset{\sigma}{\Rightarrow}\}$.

Definition 3: (Conformance Relation): An implementation of Web services under test (denotes IUT_I) is conform to its specification (denotes IUT_S) $\Longleftrightarrow \forall Trace(IUT_I)$, $\exists Trace(IUT_S)$ such that $Trace(IUT_I) \subseteq Trace(IUT_S)$.

An implementation of Web services under test is conform to its specification, Iff for each timed trace that are generated by the implementation, we must find a least of timed trace belong to its specification.

B. Test architecture

Usually, to perform unit testing of a composite Web service, it is necessary to simulate its partners. This is due to following reasons [27]:

1) In unit testing phase, some of the dependent partner services are still under development.
2) Some partner services are developed and governed by other enterprises. Sometimes it is impossible to obtain the partner services' source code and related modules, and set up the running environment for the testing.
3) Even we have had some partner services ready for use, simulated ones are sometimes still preferred because they could generate more interaction scenarios with less effort.
4) Another case, we cannot test directly some partner services that are ready for use because it generates the data perturbation.

It is straightforward to derive a test architecture that describes the test environment consisting of simulated services. A Web service under test (SUT) and its partners (including also client) communicate by exchanging *input* and *output* messages. When the SUT is being tested, the tester plays the role of the environment (i.e. its partner services).

In our work, we consider the tester as a Web service because the partners of SUT are also Web services. A Web service tester contains one controller component and a set of test execution components that represent the partner services of SUT. The controller will generate test cases (sequences of input/output message and timing delays) and send/receive input output message to/from each test execution component. Each test execution component may receive a message from the controller and sends it to SUT, or receives a message from SUT and forward it to the controller. The controller analyses the result and assigns a verdict to the test case execution. We assume here that SUT has N partner services and one client. The test architecture is shown in figure 3.

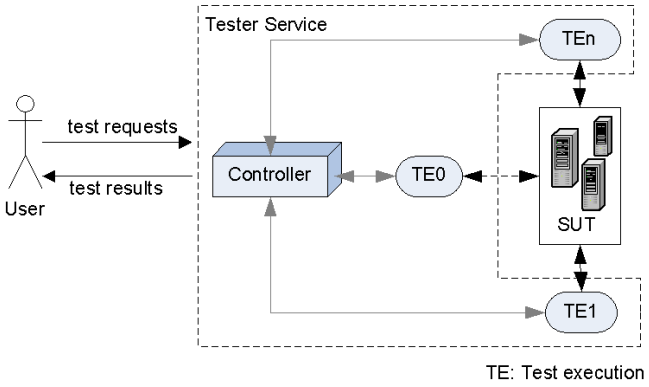

Figure 3. The test architecture

C. Online testing algorithm

In offline testing, test cases are generated from the model where the complete test scenarios and verdicts are computed before execution. In contrast, *online testing* [22–25] combines test generation and execution: only a single test primitive is generated from the specification at a time, and executed on the SUT. An observed test run is a timed trace consisting of an alternating sequence of input/output messages and timing delays.

The tester can perform two basic actions: either send an input to the SUT, or wait for an output for a duration d. If the tester observes an output at time $d' \leq d$, then it checks whether this is legal according to the specification. If not, a timeout fault will be raised as a fail verdict. In the case of flow activity, may be have many output for a duration d. So that, we use a queue (Q) to save all of output. The tester will wait an output by checking this queue either empty. The variable values will be updated by the data from input/output message. Note that, in a given state, the guard condition of a transition has one of three values: $true$, $false$ or $undefined$ when the variables values on guard are undefined. If the

invariant of the state s is $c \leq m$ (c is a local clock), then the value of the function $invariant(s)$ will be m (i.e. $invariant(s)$ = m). The online testing algorithm for Web services composition is shown in the figures 4 and 5. This algorithm will generate a fault verdict if exists an output or delay that are not belong to any observable timed traces of its specification (i.e. $\exists Trace(IUT\_I), \forall Trace(IUT\_S)$ such that $Trace(IUT\_I) \notin Trace(IUT\_S)$).

Procedure getNextAction(State s) // list of transitions
BEGIN
  $result := \emptyset$; // transitions list
  $queue := \{t \in T | t.src = s \wedge t.guard! = false\}$;
  WHILE $queue \neq \emptyset$ **DO**
    $trans := queue.pop()$
    $T := T \setminus \{trans\}$;
    IF $trans$ is an input output transition or a timestamps transition THEN
      $result.add(trans)$;
    ELSE
      $temp := \{t \in T | t.src = trans.target \wedge t.guard! = false\}$;
      $queue.add(temp)$;
  OD
  return $rerult$;
END

Figure 5. Test generation and execution (part 2/2)

D. Testing framework design

In this section, we present an example of the framework implementation design. We have implemented our framework on a local machine (i.e. the test execution components and the controller component are installed on the same machine). The architecture is shown in Fig 6. The framework consists of six main elements:

1) compiler: loads and compiles input format (TEFSM);
2) analyzer: analyze the WSDLs to extract informations of partner services about: operator, port, synchronous, asynchronous etc;
3) test execution that sends and receives the SOAP messages to/from the SUT;
4) data generation from the WSDLs and XML Schema;
5) data update function API; and
6) the controller managing the test execution components and the data generation.

In our framework, we use the TEFSM as an input specification. Hence, we develop a prototype (called BPEL2TEFSM) using rules of section IV to transform the BPEL description into a TEFSM specification.

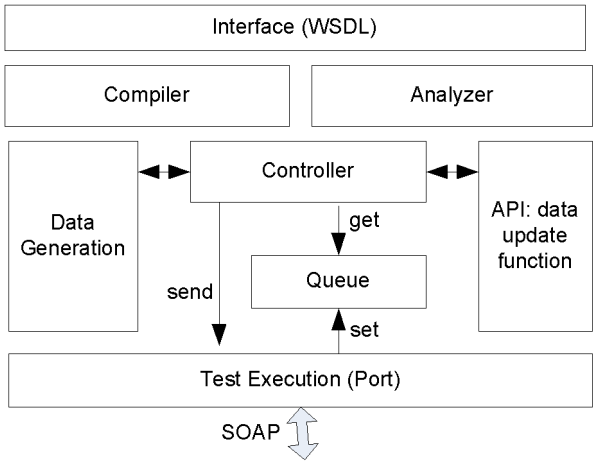

Figure 6. Online testing framework architecture

1) Test execution component: Each test execution component represents a partner service (i.e. simulating partner service). However, we do not know its internal structure, so, we will generate automatically its output messages (of partner services) based on the message type of WSDL specification. Constraints on variable will be applied on these output messages to satisfy a test case selection. In our framework, the role of each test case execution is very simple. Each test execution component receives the SOAP message from the controller and sends it to SUT by setting SOAP message into the queue Q, and it receives a SOAP message from SUT and forward it to the controller. All of test execution components have the same role, but we created one test execution component for each partner service because it has a different address. These test execution components are created by the controller based on the partner link number of BPEL specification.

2) Data generation: We use the Bai et al [4] method, based on XML Schema data type, to generate test data with constraints on input variables from tester. Typically the data constraints only cover a part of the required data. The rest can be derived automatically or by patterns using random data generation tools. This component will be controlled by the controller within the $generate\_data()$ function of figure 4.

3) Controller: The controller implements the online testing algorithm of figures 4 and 5. As the tester is a web service (asynchronous), we need a WSDL specification for the tester. In this WSDL specification, we define two functions:

1) $start(TEFSM\ M, int\ N, int\ tmax)$ to receive a test request from the client with three parameters: a TEFSM M, test execution number N, and network timeout, $tmax$, for synchronous action;
2) $finished()$ to return the result that is a list of test case and its verdict;

Online Testing Algorithm
Input: - $M = (S, s_o, S_f, V, E, C, Inv, T)$ //specification of the SUT
- Test execution number : N.
- Time to wait for an output message from the SUT: $tmax$; //network timeout
Output: Timed test cases suite and its corresponding verdicts.
BEGIN
$state := s_0$; //current state
$counter := 0$;
$tcList := \emptyset$; // a test case list
$tc := \emptyset$; // a test case
$Q := \emptyset$; // a queue to keep all SOAP message that are forwarded by test execution components
WHILE $counter \leq N$ **DO**
- $acList := getNextAction(state)$; //list of transitions, see part 2
IF $acList = \emptyset$ THEN
IF $tc \neq \emptyset$ THEN //found a test case
- $tcList.add(tc)$;
- $tc.empty()$;
- $state := s_0$ //restart (search a new test case)
- $counter + +$;
ELSE
IF $itList = \{acList.inputTrans() \cup acList.timedTrans()\} \neq \emptyset$ THEN
- randomly choose t $\in itList$;
IF $t \in acList.inputTrans()$ THEN
- $i\_msg := generate\_data(t.action)$; //generate data for input message
- randomly delay n times units based on guard condition of t on clock
- $send\_to\_test\_execution(i\_msg)$; // send an input message to SUT
- $update\_variable(i\_msg)$; // update variables with the input message value
- $tc.add(i\_msg, n)$
ELSE // t $\in$ acList.timedTrans() (i.e. timestamps transition)
- delay d times units
- $tc.add(delay = d)$ //add delay into test case
- $state := t.target$ // moving into new state
ELSE //waiting an output message from the SUT
- $d := invariant(state) = true?tmax : invariant(state)$
- sleep for d time units or wake up if Q is not empty at $d' \leq d$
IF (o_msg = $Q.pop()$) is not null THEN
//find a transition t such that $t.action = o\_msg$ from current state;
IF $search(o\_msg, state) = true$ THEN
- $update\_variable(o\_msg)$;
- $state := t.target$ // moving into new state
- $tc.add(o\_msg, d')$
ELSE
- exit(); //current test case is fails
ELSE
- exit(); //current test case is fails (clock constraints)
OD
END

Figure 4. Test generation and execution (part 1/2)

E. An example of online testing result

In this section, we illustrate our framework using the Loan Web Service (fig. 7). This process receives an input from the client. If this *input* is less than 10, it invokes the synchronous Assessment Service and receives a *risk* result. In the case, this *risk* is *low*, so it sends a *yes* response *yes* to the client. Otherwise (i.e. *input≥10 or risk!=low*), it invokes the asynchronous Approval Service by sending a request and uses a BPEL *pick* activity for one of the following cases: (1) to receive an asynchronous response from the partner service and send this response to client; (2) to send a timeout fault to client if there is not response from the partner service after a duration (e.g. 60 seconds).

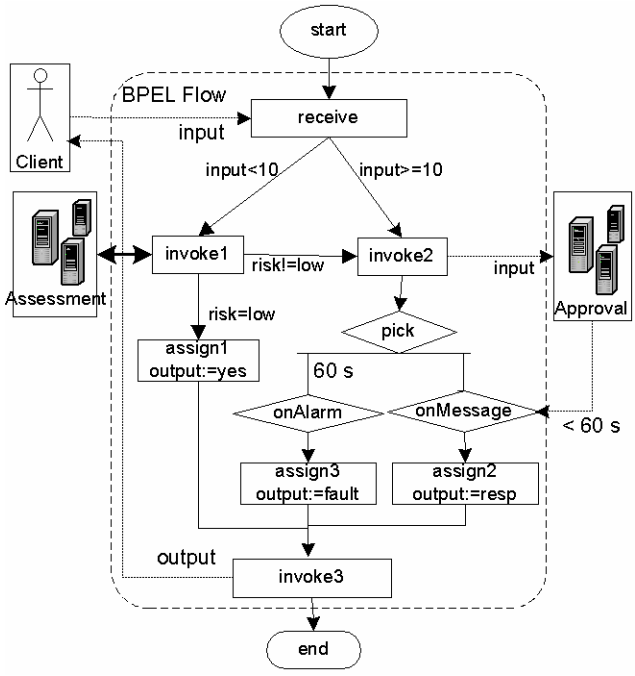

Figure 7. The Loan Web Service

The figure 8 introduces the TEFSM of the Loan Web Service, where c is a local clock. We assume here that:

1) all types of messages are integers;
2) the value of the input messages (i.e. input_msg, risk, response) and timing delay for each input message are randomly generated;
3) the test execution number is five (i.e. N=5).

The algorithm will finish either when there is a fault or the test execution number reaches the limit. We have a following timed test cases list with the format (message name (value), timing delay) (Fig 9):

VI. Conclusions

To address the problem of Web services testing, this paper proposes an online testing framework for Web services composition described in BPEL. We mainly covered four

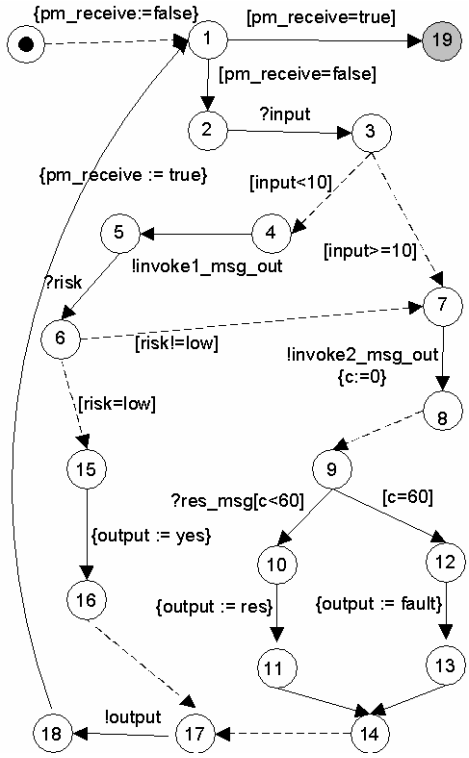

Figure 8. TEFSM of the Loan Web Service

1. $(?input(8), 0) \rightarrow (!invoke1\_msg\_out(8), 2) \rightarrow (?risk(1), 2) \rightarrow (!invoke2\ \_msg\_out(8), 0) \rightarrow (?response(1), 30) \rightarrow (!output(1), 30)$
2. $(?input(1), 0) \rightarrow (!invoke1\_msg\_out(1), 2) \rightarrow (?risk(0), 2) \rightarrow (!output(1), 2)$
3. $(?input(5), 0) \rightarrow (!invoke1\_msg\_out(5), 5) \rightarrow (?risk(1), 6) \rightarrow (!invoke2\_msg\_out(5), 0) \rightarrow (delay = 60) \rightarrow (!output(-1), 1)$
4. $(?input(10), 0) \rightarrow (!invoke1\_msg\_out(10), 2) \Longrightarrow FAIL.$

Figure 9. Test Result

topics: modeling BPEL specification by a TEFSM, a test architecture, an online testing algorithm which generates and executes test cases, and an example of testing framework design. We focus on unit testing of an implementation of a Web services composition, based on *gray-box* approach and conformance testing.

These are some of limitations in our framework. Several test cases can be not selected because the algorithm randomly selects the test cases. Moreover, it is limited by the time execution number. In the case of *flow* activity, if the service invokes many actions on parallel and it valides the timing contraints. May be these timing contraints are not

valided because our framework works (with a flow activity) as a sequence.

In future works, we plan to work on integration testing [2] that it is aimed at exercising the interaction among components and not just single units. Hence, an integration test case involves the execution of several components.

ACKNOWLEDGMENT

This Research is supported by the French National Agency of Research within the WebMov Project http://webmov.lri.fr

REFERENCES

[1] OASIS. Web Services Business Process Execution Language (BPEL) Version 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html

[2] A. Bucchiarone, H. Melgratti, and F. Severoni, *"Testing Service Composition"*, In Proceedings of ASSE07, Mar del Plata, Argentina, August 2007.

[3] A. Bertolino, A. Polini, "The Audition Framework for Testing Web Services Interoperability", *Proc of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications*, 2005.

[4] X. Bai, W. Dong, W.T. Tsai, Y. Chen, "WSDL-Based Automatic Test Case Generation for Web Services Testing", *Proc of the IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, pp 207 - 212, Beijing October 2005.

[5] Arthur Gill, "Introduction to the Theory of Finite-State Machines", *Published by McGraw-Hill Book Co..*, New York, 1962.

[6] Jose Garcia-Fanjul, Javier Tuya, Claudio de la Riva, "Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN", *International Workshop on Web Services Modeling and Testing*. WS-MaTe 2006.

[7] Y. Zheng, P. Krause, "Automata Semantics and Analysis of BPEL", *International Conference on Digital Ecosystems and technologies*. DEST 2007.

[8] Y. Zheng, J. Zhou, P. Krause, "A Model Checking based Test Case Generation Framework for Web Services", *International Conference on Information Technology*. ITNG 2007.

[9] Y. Zheng, J. Zhou, P. Krause, "An Automatic Test Case Generation Framework for Web Services", *JOURNAL OF SOFTWARE*, VOL. 2, NO. 3, SEPTEMBER 2007.

[10] X. Fu T. Bultan J. Su, "Analysis of Interacting BPEL Web Services", *International Conference on World Wide Web*. May 17 - 22, 2004, New York, USA.

[11] A. Wombacher, P. Fankhauser, and E. Neuhold, "Transforming bpel into annotated deterministic Finite state automata for service discovery" *Procs of ICWS04*, 2004.

[12] R. Kazhamiakin, P. Pandya, and M. Pistore, "Timed modeling and analysis in web service compositions", *The First International Conference on Availability, Reliability and Security*, vol. Volume 0, pp. 840 846, 2006.

[13] M. Lallali, F. Zaidi, A. Cavalli, "Timed modeling of web services composition for automatic testing", *The 3rd ACM/IEEE International conference on Signal-Image technologie and internet-Based Systems (SITIS'2007)*, China 16 - 19 december 2007.

[14] M. Lallali, F. Zaidi, A. Cavalli, "Transforming BPEL into Intermediate Format Language for Web Services Composition Testing", *The 4th IEEE International Conference on Next Generation Web Services Practices*, October, 2008

[15] M. Lallali, F. Zaidi, A. Cavalli, Iksoon Hwang, "Automatic Timed Test Case Generation for Web Services Composition", *Sixth European Conference on Web Services*. Dublin, Ireland, Nov 12 - 14, 2008.

[16] T. D. Cao, P. Felix, R. Castanet, I. Berrada, "Testing Web Services Composition using the TGSE Tool", *IEEE 3rd International Workshop on Web Services Testing (WS-Testing 2009)*, July 7, 2009, Los Angeles, CA, USA.

[17] C. Bartolini, A. Bertolino, E. Marchetti, A. Polini, "WS-TAXI: a WSDL-based testing tool for Web Services", *International Conference on Software Testing Verification and Validation*, April 1 - 4, 2009, Denver, Colorado - USA.

[18] R. Alur, D. L. Dill, "A Theory of Timed Automata", *Theory of Computer Science* .vol 126, no 2, pp 183 - 235 , 1994.

[19] ChangSup Keum, Sungwon Kang, In-Young Ko Jongmoon Baik and Young-Il Choi "Generating Test Cases for Web Services Using Extended Finite State Machine", *TestCom 2006*, New York, NY, USA, May 16 - 18, 2006

[20] R. Heckel and L. Mariani, "Automatic conformance testing of web services", *Fundamental Approaches to Software Engineering*, pp. 34 - 48, LNCS 3442, 2 - 10 April, 2005.

[21] H. Huang, W. T. Tsai, R. Paul, Y. Chen, "Automated Model Checking and Testing for Composite Web Services", *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pp.300-307, 2005.

[22] M. Mikucionis, K. G. Larsen, B. Nielsen, "T-UPPAAL: Online Model-based Testing of Real-time Systems", *19th IEEE International Conference on Automated Software Engineering*, pp 396 - 397. Linz, Austria, Sept 24, 2004.

[23] K. G. Larsen, M. Mikucionis, B. Nielsen, "Online Testing of Real-time Systems Using UPPAAL", *Formal Approaches to Testing of Software*. Linz, Austria. Sept 21, 2004

[24] G. J. Tretmans and H. Brinksma "TorX: Automated Model-Based Testing", *First European Conference on Model-Driven Software Engineering*, Nuremberg, Ger-

many, Dec 11 - 12, 2003.
[25] H. Bohnenkamp and A. Belinfante, “Timed Testing with TorX”, *Formal Methods 2005*, LNCS 3582, pp. 173 - 188, 2005.
[26] P. Mayer, “Design and Implementation of a Framework for Testing BPEL Compositions”, *Master thesis, Leibniz University*, Hannover, Germany, Sep 2006.
[27] Z. Li, W. Sun, Z.B. Jiang, X. Zhang, “BPEL4WS Unit Testing: Framework and Implementation”, *Proc of the IEEE International Conference on Web Service (ICWS’05)*, pp 103 - 110, 2005.
[28] A. Cavalli, Edgardo Montes De Oca, W. Mallouli, M. Lallali, “Two Complementary Tools for the Formal Testing of Distributed Systems with Time Constraints”, *12th IEEE International Symposium on Distributed Simulation and Real Time Applications*, Canada, Oct 27 - 29, 2008.
[29] BPELUnit - The Open Source Unit Testing Framework for BPEL. http://www.se.uni-hannover.de/forschung/soa /bpelunit/
[30] EVIWARE. soapUI. http://www.eviware.com/
[31] PushToTest. TestMaker. http://www.pushtotest.com/