WSOFT : an automatic testing tool for web services composition

Dung Cao, Patrick Félix, Richard Castanet

WSOTF: An Automatic Testing Tool for Web Services Composition

Tien-Dung Cao, Patrick Felix, and Richard Castanet
LaBRI - CNRS - UMR 5800, University of Bordeaux 1
351 cours de la libération, 33405 Talence cedex, France.
Email: {cao,felix,castanet}@labri.fr

***Abstract*—This paper presents an automatic conformance testing tool with timing constraints from a formal specification (TEFSM: Timed Extended Finite State Machine) of web services composition (WSOTF: Web Service composition, Online Testing Framework), that is implemented by an online testing algorithm. This algorithm combines simultaneously idea of test execution and debug to generate and simultaneously execute the test cases. In this tool, the complete test scenario (timed test case suite and data) is built during test execution. This tool focus on unit testing, it means that only the service composition under test is tested and all its partners will be simulated by the WSOTF. This tool also considers the timing constraints and synchronous time delay. We can also use this tool for debug that is not easy while we develop a composite of Web service.**

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

I. INTRODUCTION

As the software system, a web service (WS) can be tested by traditional software testing techniques such as: conformance testing, performance testing, availability testing, robustness testing, etc. There is not an exception for a WS composition. In general, the conformance testing is firstly applied to verify the conform of an implementation with its specification after the development of a WS. In the last years, many approaches and tools are developed for WS composition conformance testing [6–9, 17, 21, 22]. In these works, testing consists of there phases: test case generation, data generation (the complete test scenario is built), test execution and give the verdict. Here, we call these works be offline approach, it means that the complete test scenario is built before test execution. The test purpose is used in these works to generate test case. In the complex services, the number of test case suite that can cover all actions is very large, testing by test purpose is necessary. But with a service is less complex, this is difficult for tester because currently, there are not an automatic communication between the phases of there works. Moreover, testing by test purpose is not necessary because the number of test case suite that tester must execute to verify the conformation of a Service Under Test (SUT) and its specification is not very large. That can be automatically generated by the execution of N times and at each state, the next action will be randomly selected from the next possible action list of current action.

Another hand, the debug is a process of monitoring (collect the traces) while we develop a software that saves the execution traces, its data and analyses to find an error, mistake, failure in a system. Most bugs arise from mistakes and errors made by people in either a program's source code or its design. The debug of a web service composition is very difficult because a composite of Web service is a runtime system, its components (partner services) are invoked and integrated at the runtime.

This paper presents an automatic conformance testing tool with timing constraints from a formal specification (TEFSM: Timed Extended Finite State Machine) of web services composition (WSOTF: Web Service composition, Online Testing Framework), that combines simultaneously idea of test execution and debugger to generate and simultaneously execute the test cases. From the current action, we have a next possible action list (consists of input/output action and synchronous timed delay) based on the data value at the current action. If this list is empty, we arrive a final state. Else, if the set of input action and synchronous timed delay are not empty, we will randomly choice an action of this list, generate the data and execute the correspondent action (the test execution phase). After that, the current action will be updated by this action (the debug phase). The same of output action, we wait a message from SUT, check it and update current action. We call this be online testing approach. We choice the formal specification TEFSM [20] as input format of WSOTF because this formalism is closely related to timed automata [12] and permits to carry out timing constraints, clocks, state invariants on clocks and data variables. The specification of a WS composition as UML can be translated into TEFSM before using this tool. With a BPEL specification (the real implementation may be another language as Java, .NET, PHP, etc), we can use the rules in [10] to translate into TEFSM. This tool focus on unit testing, it means that only the service composition under test is tested and all its partners will be simulated by WSOTF. When WSOTF receives a request from SUT (partner invoke), it will randomly generate the correspondent SOAP message with its structure and returns it to SUT. This tool can also be used to debug a WS composition while its development.

The rest of paper is organized as follows. Section II, we give the formal definition of TEFSM and conformance notation. Section III presents the detail of the WSOTF tool. A test application of Loan Approval Service using the WSOTF is shown in the section IV. Finally, section V concludes the paper and the future works.

II. Preliminaries

In this section, we present some definitions about Timed Extended Finite State Machine (TEFSM), that we use as input format of WSOTF, and the conformance notation that is applied in WSOTF to automatically check the conformation of an implementation of Web service composition with its specification.

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 in 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, $(c_0, c_1, ..., c_n\ )$ (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, V, E_τ, 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;
- *V* is a finite set of data variables, $D_V^{|V|}$ is the data variable domain of V;
- E_τ is a finite set of events. E_τ is partitioned into:
  - Input event $?a$ (E_I);
  - Output event $!b$ (E_O);
  - τ is the internal event.
- *C* is a finite set of clocks including a global clock gc (never reset);
- *Inv*: S $\mapsto$ Φ(C) is a mapping that assigns a time invariant to states;
- $T \subseteq S \times E_\tau \times P(V) \vee \Phi(C) \times 2^C \times \mu \times S$ 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;
- M is a deterministic machine;

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

For $a \in E_I \cup E_O$, we write $s \xrightarrow{a}$, iff $\exists s' \in S$ such that $s \xrightarrow{a} s'$. We write $s \xrightarrow{a_1,...,a_n} s'$ 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 $s \xrightarrow{\tau,...,\tau} s'' \xrightarrow{a} s''' \xrightarrow{\tau,...,\tau} s'$. We define $\Gamma = (E_I \cup E_O \times \mathbb{R}_+)$ 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. A *timed sequence* $\sigma \in Seq(\Gamma)$ is composed of actions a and non-negative reals d such that: $s \xrightarrow{(a,d)} s' \oplus d$, 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 2: (Conformance Relation): An implementation of Web services under test (denotes IUT_I) is conform to its specification (denotes IUT_S) $\iff \forall Trace(IUT_I)$, $\exists Trace(IUT_S)$ 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 at least a timed trace (from initial state s_0 to a final state) belonging to its specification. A fault verdict will be immediately assigned for current trace 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 $Trace(IUT\_I) \notin Trace(IUT\_S)$).

III. WSOTF: an automatic testing tool

This section presents an overview of WSOTF tool that consists of a test architecture, a test algorithm, the architecture of WSOTF and finally an experience example using WSOTF.

A. Test architecture

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 the client) communicate by exchanging *input* and *output* SOAP messages. When the SUT is being tested (unit testing), the tester plays the role of the environment (i.e. its partner services). In WSOTF, we use a central test architecture that the client and its partners will be simulated by only tester. This tester will send and receive the SOAP message to/from SUT and it is concentratelly controlled by a controller. The controller will generate the test case and give the verdict. The figure 1 shows an abstract model of SUT (A) (a client and two partners) and a correspondent central test architecture (B).

B. Test algorithm

In WSOTF, we distinguish two following transition types:

- 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

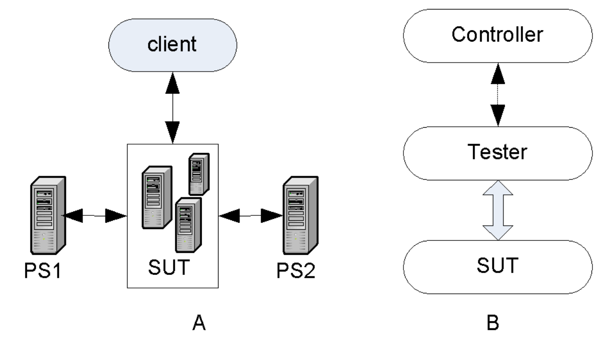


Figure 1. The abstract model of SUT and Central Test Architecture

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.

As we present in the introduction section, from the current action, we have a next possible transition list (consists of action transitions and timestamps transitions) based on the data value at the current action. If this list is empty, we arrive a final state. Else, if the set of input action and synchronous timed delay (timestamps transitions) are not empty, we will randomly choice a transition (note t) of this list, generate the data and execute the correspondent action (the test execution phase) if t belongs to the set of input action, else (t belongs to the set of timestamps transitions) we do a synchronous time. After that, the current action will be updated by this action (the debug phase). The same of output action, we wait a message from SUT, check it (give a fail verdict if the receivable message is not correct) and update current action. A timeout verdict is also given if we do not receive a message after a duration. After execution of each transition action (input/output), we must update the data variable with values of the SOAP message. The detail of algorithm is provided in [10]. In the current version, we process the activities of a flow activity as the sequence activities. The next version, we will improve this algorithm to process simultaneously the activities of a flow activity.

C. WSOTF architecture

The detail architecture of WSOTF is shown in the figure 2 that consists of five main components: loader, data generation, data update function, executer and test execution.

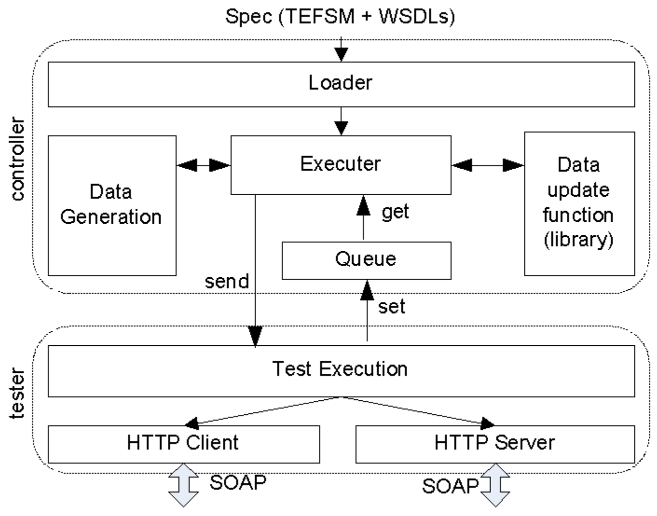


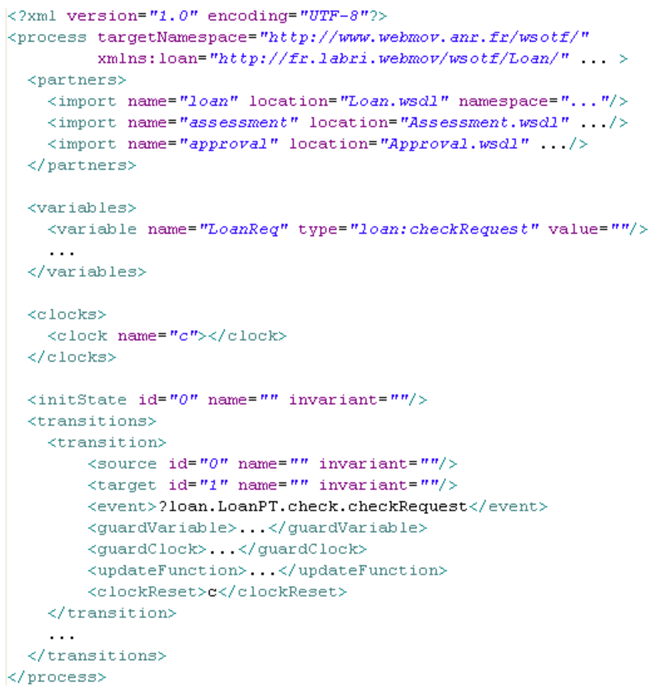
Figure 2. Architecture of WSOTF

1) **loader:** loads the input format and analyses the wsdls to get the informations of partners;
2) **data generation:** we reuse the code of SoapUI [24] to generate SOAP format. The data for each field of SOAP message is randomly generated or use a default value. This depends on the configuration file;
3) **data update function:** define the update functions for the variables;
4) **executer:** implementation of online testing algorithm to generate test case, control test execution and assign the verdict. It uses the data generation module and the data update function module to generate SOAP message and update value of variables;
5) **test execution:** this consists of two components: a http client to invoke the request into SUT (the client request or partner callback in the case asynchronous services, the result is return on a different port) and http server to receive and return the message from SUT on the same port. When test execution receives a SOAP message from SUT, it sets this message into a queue to wait a processing of the executer. It receives directly SOAP messsage from the executer and sends it to SUT;

D. Testing a web service using the WSOTF

To test a Web service composition, before we must translate its specification (UML, BPEL, etc.) into the input format of WSOTF. The figure 3 shows an input format example of WSOTF. This format consists of partner section that declares the partners name and location of wsdl specification, variables list (variable types are: int, $boolean$ or message type of SOAP that is defined in WSDL), local clock, initial state, and a list of transitions. Each transition consists of seven fields:

source state, target state, event name, guard on variable, guard on clock, data update function and local clock to be reset. In WSOTF, we use the form *?pl.pt.op.msg* to represent a format of an input action that means the reception of the message (*msg*) for the operator (*op*) of the portTyte *pt* from the partner (*pl*) and the form *!pl.pt.op.msg* represents a format of an output action (resp. the emission of the message (*msg*) for the operator (*op*) of the portTyte *pt* to the partner (*pl*)). The result of WSOTF (traces including interval time between two actions and its correspondent verdict) is saved in a xml file. This tool allows declare (an enumeratation) the value of some fields of SOAP message in an enumetation file that can use to test by purpose (fix the condition for each branch), correlation data (current version not support yet the correlation data functions) or to debug. At each excution time, WSOTF requests a number N (integer) and it repeats N times to generate N traces if there is not error and the correspondent verdict is $PASS$. WSOTF will stop immediately if it finds an error (message receive incorrect, or timeout).



```
<?xml version="1.0" encoding="UTF-8"?>
<process targetNamespace="http://www.webmov.anr.fr/wsotf/"
         xmlns:loan="http://fr.labri.webmov/wsotf/Loan/" ... >
  <partners>
    <import name="loan" location="Loan.wsdl" namespace="..."/>
    <import name="assessment" location="Assessment.wsdl" .../>
    <import name="approval" location="Approval.wsdl" .../>
  </partners>

  <variables>
    <variable name="LoanReq" type="loan:checkRequest" value=""/>
    ...
  </variables>

  <clocks>
    <clock name="c"></clock>
  </clocks>

  <initState id="0" name="" invariant=""/>
  <transitions>
    <transition>
        <source id="0" name="" invariant=""/>
        <target id="1" name="" invariant=""/>
        <event>?loan.LoanPT.check.checkRequest</event>
        <guardVariable>...</guardVariable>
        <guardClock>...</guardClock>
        <updateFunction>...</updateFunction>
        <clockReset>c</clockReset>
    </transition>
    ...
  </transitions>
</process>
```

Figure 3. Input format of WSOTF

E. Using the WSOTF as a debugger

After developement of a web service composition, suppose the implementation has conformed with its specification, a question is: how does it run? to solve this question, in general, we execute it and use a debugger to monitor its behaviour (collect the traces). To do this in a web service composition is very difficult because: 1) a composite of Web service is a runtime system, its components (partner services) are invoked and integrated at the runtime. Moreover, the data of partner services will be effected while we execute the web service composition. 2) we must install the prope on the server to collect the traces. In WSOTF, all partner services are simulated by the WSOTF. This allows us use WSOTF to execute a web service composition without effect to its partners. The WSOTF collects a set of traces after it executes on the service. This is a reason to use the WSOTF as a debugger.

IV. Application: Loan Approval Service

This example consists of a simple loan approval service [2]. Customers of the service send loan requests, including personal information and amount being requested. Using this information, the loan service executes a simple process resulting in either a ”loan approved” message or a ”loan rejected” message. The decision is based on the amount requested and the risk associated with the customer. For low amounts of less than 10 a streamlined process is used. In the streamlined process low-risk customers are approved automatically. For higher amounts, or medium and high-risk customers, the credit request requires further processing. For each request, the loan service uses the functionality provided by two other services (Approval and Assessment). In the streamlined process, used for low amount loans, a risk assessment service is used to obtain a quick evaluation of the risk (low or $high$) associated with the customer. A full loan approval service (possibly requiring direct involvement of a loan expert) is used to obtain assessments when the streamlined approval process is not applicable. In this example, to test with a synchronous time delay, we use the approval service as an asynchronous service. It means that loan approval service will send the request into approval service on a port and receives the response on another port. A fault will be sent to client if there is not a response under a duration (20 seconds). We deploy this service on the Active-Bpel engine [25] and use WSOTF to test it. From this specification of this service, we model manually it specification by a TEFSM in the figure 4. Because the return value of risk assessment service is either low or $high$, so we declare the value of this field in the enumeration file ($risk = low; high$). We also declare the min and the max of $integer$ to generate amount are 0 and 20. The other field will be randomly generated. The figure 5 shows the test result of Loan Approval Service using the WSOTF tool with the parameter $N = 10$. With ten traces in the figure 5, we found four different traces that is four test cases. So that, we can increase the value of parameter N to repeat many times to cover all possible test cases (in this example is five).

* **Note:** we also apply this tool to test the Travel Reservation Service of Netbean. The xsd chemas of this application is very complex with many option fields. The receivable result is less effect because the branch conditions based on the appearance of these option fields while WSOTF

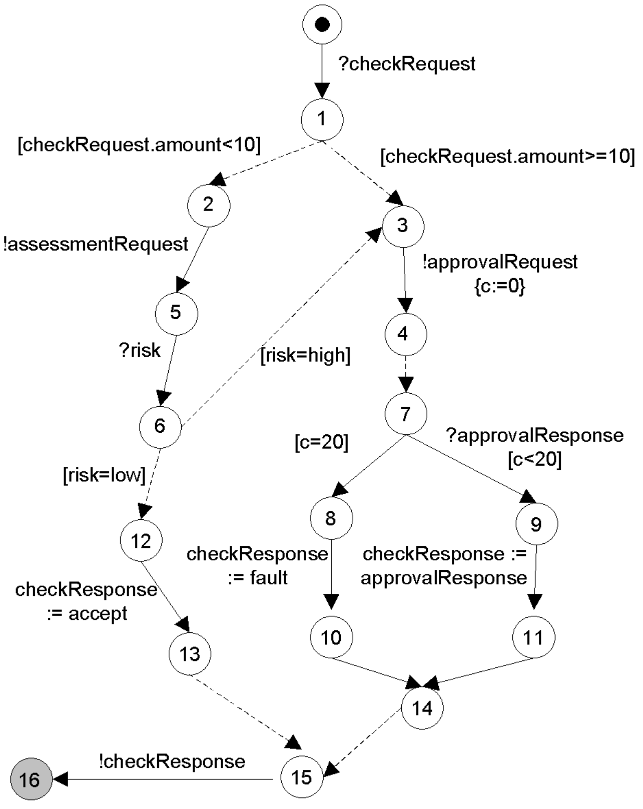


Figure 4. A formal specification of Loan Approval Service (TEFSM)

1.$?checkRequest[amount = 5] \rightarrow !assessmentRequest \rightarrow ?risk[low] \rightarrow !checkResponse$
2.$?checkRequest[amount = 12] \rightarrow !approvalRequest \rightarrow ?approvalResponse \rightarrow !checkResponse$
3.$?checkRequest[amount = 8] \rightarrow !assessmentRequest \rightarrow ?risk[low] \rightarrow !checkResponse$
4.$?checkRequest[amount = 15] \rightarrow !approvalRequest \rightarrow delay = 20 \rightarrow !checkResponse$
5.$?checkRequest[amount = 9] \rightarrow !assessmentRequest \rightarrow ?risk[low] \rightarrow !checkResponse$
6.$?checkRequest[amount = 1] \rightarrow !assessmentRequest \rightarrow ?risk[high] \rightarrow !approvalRequest \rightarrow delay = 20 \rightarrow !checkResponse$
7.$?checkRequest[amount = 10] \rightarrow !approvalRequest \rightarrow delay = 20 \rightarrow !checkResponse$
8.$?checkRequest[amount = 16] \rightarrow !approvalRequest \rightarrow delay = 20 \rightarrow !checkResponse$
9.$?checkRequest[amount = 6] \rightarrow !assessmentRequest \rightarrow ?risk[low] \rightarrow !checkResponse$
10.$?checkRequest[amount = 8] \rightarrow !assessmentRequest \rightarrow ?risk[high] \rightarrow !approvalRequest \rightarrow delay = 20 \rightarrow !checkResponse$

Figure 5. The test result of Loan Approval Service

generates a SOAP message either with all option fields or not.

V. Conclusions and Future works

To address the problem of Web services testing, this paper proposes a testing tool for web services composition from a formal specification that combines simultaneously idea of test execution and debugger to generate and simultaneously execute the test cases. In the WSOTF tool, the complete test scenario (timed test case suite and data) is built during test execution. This tool focus on unit testing, it means that only the service composition under test is tested and all its partners will be simulated by the WSOTF. The timing constraints and synchronous time delay are considered in this tool.

In the future works, we will extend TEFSM with a set of state properties (SP): $S \rightarrow \{on,\ off,\ null\}$ is a mapping that assigns a property to states. A state has the property on represents s_{in} of flow activity (resp. off represents s_{out}). And next, we improve the algorithm to process simultaneously the activities of a flow activity. When a state has the property be on, all of next actions will be simultaneously processed instead of randomly select an action and only process it. Otherwise, a graphic version will be developed to easy design input format and easy review the result.

Acknowledgment

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

References

[1] Web Services Description Language 1.1. http://www.w3.org/TR/wsdl
[2] 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
[3] A. Bucchiarone, H. Melgratti, and F. Severoni, *"Testing Service Composition"*, In Proceedings of ASSE07, Mar del Plata, Argentina, August 2007.
[4] 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.
[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, J. Zhou, P. Krause, “A Model Checking based Test Case Generation Framework for Web Services”, *International Conference on Information Technology*. ITNG 2007.

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

[9] 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.

[10] T. D. Cao, P. Felix, R. Castanet, I. Berrada, “Online Testing Framework for Web Services Composition”, *IEEE 3rd International Conference on Software Testing, Verification and Validation*, April 6-9, 2010, Paris, France. (submitted)

[11] 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.

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

[13] 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.

[14] 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

[15] G. J. Tretmans and H. Brinksma “TorX: Automated Model-Based Testing”, *First European Conference on Model-Driven Software Engineering*, Nuremberg, Germany, Dec 11 - 12, 2003.

[16] H. Bohnenkamp and A. Belinfante, “Timed Testing with TorX”, *Formal Methods 2005*, LNCS 3582, pp. 173 - 188, 2005.

[17] P. Mayer, “Design and Implementation of a Framework for Testing BPEL Compositions”, *Master thesis, Leibniz University*, Hannover, Germany, Sep 2006.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] Lina Bentakouk, Pascal Poizat, Fatiha Zadi, “A Formal Framework for Service Orchestration Testing Based on Symbolic Transition Systems”, *TESTCOM/FATES 2009*, Nov 2-4 2009, Eindhoven, the Netherlands.

[23] BPELUnit - The Open Source Unit Testing Framework for BPEL. http://www.se.uni-hannover.de/forschung/soa /bpelunit/

[24] EVIWARE. soapUI. http://www.eviware.com/

[25] Active Endpoints. The Active-Bpel engine. http://www.activevos.com/community-open-source.php