Testing Web Services Composition using the TGSE Tool

Dung Cao, Patrick Félix, Richard Castanet, Ismaïl Berrada

Testing Web Services Composition using the TGSE Tool

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

This paper proposes an approach to test (actively and passively) Web services composition described in BPEL using TGSE (Test Generation, Simulation and Emulation), that is a tool for generating test cases for Communicating Systems (CS). TGSE implements a generic generation algorithm allowing either test cases derivation or traces checking. It supports the description of one or several components with data and temporal constraints. First, 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. As our framework can handle both active and passive testing, on the one hand test cases are obtained by stimulating the CS. In this case, the exploration is guided by the use of test purposes modeled by TEFSM (a test purpose is considered as a part of the CS). On the other hand, TGSE can check whether a trace is valid according the specification or not. Finally, the Loan Web Service is used as a case study.

1. Introduction

BPEL (Business Process Execution Language) [1] is an emerging standard language to describe web service composition behavior. A BPEL process implements one Web service by specifying its interactions with other Web services (called partner services). The ways to test Web services composition can be classified into two basic groups. The most natural way, namely the active testing approach, the active testing is a testing method that interacts with the system under test by injecting test cases and analyzing the answers of the system regarding the expected results. Another possibility is the passive testing approach. The passive testing is a testing method based on a probe on the system in execution. There is no interaction with the system under execution. This method operates by checking properties on the logs provided by the probe.

Various approaches for Web services composition testing were analyzed by [2], including unit testing, integration testing, black box testing and white box testing of choreographies and orchestrations. For active testing, several test cases generation methods have been recently proposed for BPEL Web services [5, 6, 8, 9] even with timed constraints [15, 16]. These testing methods have given pertinent test cases. However, for passive testing, to our knowledge, the temporal constraints of Web services have not been considered. The methods proposed in [20, 21] focus only on monitoring techniques and semantic faults diagnosis of BPEL services.

A. Bucchiarone et al. [2] have defined two approaches for Web services composition testing:

- ***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...). Classical structural coverage criteria based on the source code can then be applied.
- ***Black box*** approach: In this approach, a composite Web service is actually coded in a different language from the specification. For instance, a BPEL specification is coded as a Java program. An implementation of the composite Web service is then tested without any information of its internal structure. Test cases are generated only from the specification.

In this paper, we focus on model-based testing of Web services composition given by BPEL specifications, and we consider the *black-box* approach.

Firstly, we present a timed modeling of BPEL specification based on Timed Extended Finite State Machine (TEFSM). The TEFSM formalism allows handling not only BPEL behaviors but also data and temporal constraints. In this model, we assign a time invariant for each timed activity of BPEL (for example: *wait*). We also describe how to translate BPEL specification into TEFSMs.

Secondly, for both active and passive testing, the TEFSMs corresponding to the BPEL specification and TGSE tool will be used. In fact, TGSE (Test Generation, Simulation and Emulation) [18, 19] is a toolkit developed by LaBRI within the RNRT Avrroes project and the European project Marie Curie RTN TAROT (MCRTN 505121). It contains

a test cases generator supporting communicating systems modeling and implementing a generic test generation algorithm. It supports the passive and active testing (with test purpose) of one or several components with data and temporal constraints. For active testing, test cases generation is based on simulation where the exploration is guided by test purposes. The TEFSM of BPEL specification and the test purpose will be modeled by a communicating system that is the input of the TGSE tool. For passive testing, the TGSE tool is used to check that a trace of an implementation is a valid execution of the BPEL specification. This trace is also modeled as a TEFSM and it is a component of the CS.

Finally, by presenting the Loan Web service case study, we show how our framework can be used in practice.

The remainder of this paper is organized as follows. Section 2 reviews some previous works on Web services composition testing. Section 3, we give some definitions about: TEFSM that is used to model BPEL process, a partial of TEFSM and a Communicating System. The section 4 describes the relationship between BPEL concepts and TEFSM. How to test a service composition using the TGSE tool is presented in the section 5. Section 6 reports our experimental result with the Loan Web Service. Finally, section 7 concludes the paper.

2. Related Works

In the last years, there are several techniques and tools that 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. Jose Garcia-Fanjul et al [5] use the SPIN model checker to generate test cases for compositions given in BPEL. In order to systematically derive test suites, the transition coverage criterion is considered. Yongyan Zheng and Paul Krause [6] 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. [9] use one more time the SPIN model checker to verify BPEL specification. However, the authors do not transform directly BPEL into Promela as in [5]. BPEL will be translated to guard conditions which it is transformed to Promela. In all of these methods, test cases are generated from counterexamples generated by the SPIN model checker. Transforming BPEL into Intermediate Format Language (IF) is presented in [16]. Timed test cases are generated using TestGen-IF tool. [23, 24] present also a framework for white-box testing . However, the authors do not consider automatic test case generation [16].

Regarding passive testing, several methods have been proposed. [20, 21] propose a model-based approach to diagnose orchestrated Web service process. Firstly, the authors convert the Web service orchestration language, BPEL, into synchronized automata (discrete-event systems) as a formal description of the topology. Secondly, after an exception is thrown, the techniques in Artificial Intelligence provide ways to monitor and diagnose the execution trajectory based on the formal model and the observed evolution of the business process. However, timing constraints were not considered in these works.

3. Preliminaries

BPEL specification can be described by means of formal models such as TEFSM [14] (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 noted $\vec{c}$ (resp. $\vec{v}$).

Definition 1: (TEFSM): A TEFSM M is defined as a sextuple, M = (S, s_0, V, E $\cup\{\epsilon\}$, 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;
- *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*);
- *C* is a finite set of clocks including a global clock (never reseted);
- *Inv* : S $\mapsto$ Φ(C) is a mapping that assigns a time invariant to states ;
- $T \subseteq S \times E \times P(V) \wedge \Phi(C) \times 2^C \times \mu \times S$ is a set of transitions relation where:
  - $P(\vec{v}) \wedge \phi(\vec{c})$: are guard conditions on data variables and clocks;
  - $\mu(\vec{v})$: Data variable update function;
  - $X \subseteq 2^C$: 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 functions, and c is a set of clocks to be reset.

Definition 2: (Partial of TEFSM): Let M be a TEFSM. The partial 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 [14] 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). A TEFSM can also have a graphic format (see Fig 1).

Definition 3: (Communicating System): A Communicating System (CS) is a 5-tuple CS=(SP, SV, R, $M_{i,1\leq i\leq n}$, TP) where:

- *SP* is a finite set of shared parameters;
- *SV* is a finite set of shared variables;
- M_i = $(S_i, s_{0i}, V_i, E_i, C_i, Inv_i, T_i)$ is a TEFSM ;
- *TP* is a TEFSM representing the test purpose;
- *R* is a finite set of synchronization rules where each rule $\vec{r}$ is a vector n+1 elements;

A Communicating System declares a set of shared resources (parameters and variables), a set of automata, a set of rules describing the different possible synchronizations between the entities, and a test purpose modeled by an automaton.

4. 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 model a BPEL process as a communicating system with three automaton (*activity, faultHandler, eventHandler*). The synchronization rules will describe synchronous actions between them. 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 *faultHandler*. The *scope* activity will be model as a *process*. But in a *scope* activity, a *compensationHandler* can exist. In that case, a CS has four automaton.

4.1. 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*.

4.2. 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. To synchronize the faults with *faultHandler* machine, we add two transitions *!fault* and *?done* into each partial machine if *faultHandler* activity of process exists.

The 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})

The 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})

The 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})

The 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≤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≤dl),$(s_{out}, true)\}$, $\{t_1\}$)
  - t_1=(s_{in},< _ ,*[gc=dl & stop=false],*{$\emptyset$} >,s_{out})

The 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})

The 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})

The 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})

The 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})

4.3. Structured Activities

Structural activities are the *sequence, while, switch, flow, pick, repeatUntil, if* and *scope*. They take some partial machines $PM_{i,i\in[0,n]}$ (see Fig 1) and combine them to have a new partial machines.

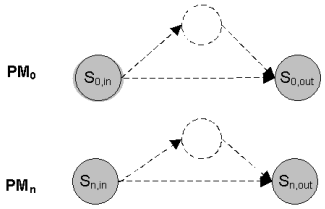

Figure 1. Partial machines

The partial machines of structural activities (*sequence, while, switch and pick*) are shown in Fig 2. In our framework, the *repeatUntil* activity will be modeled as a *while* activity. The conditional behavior *if* will be also modeled as a *switch* activity. The *eventHandler* activity will be model as pick activity. The *flow* activity allows specifying one or more concurrent activities [1]. It specifies the parallel execution of the flow partial TEFSM. The partial machine of *flow* finishes when all of its sub-partial machine finish. We use a boolean variable for each its sub-partial machine to examine the termination of each machine. The initial value of these variable is *false* (see Fig 3). The *links* defined in the *flow* activity permit to enforce precedence between these activities, i.e. it permits synchronization (see Fig 4).

The *faultHandlers* element combines a switch activity applied to various sequences of a *catch* or a *catchAll* activities and a sub-activity of the partial machine. The *catchAll* element is used to catch all faults not handled by the defined catch activities. Fig 5 models a *faultHandler* activity.

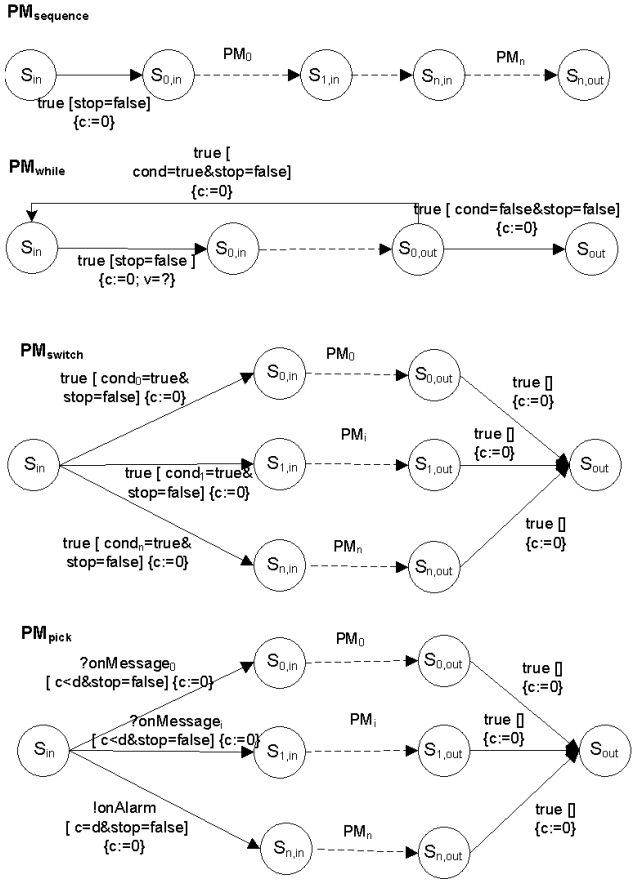

Figure 2. Modeling structural activities

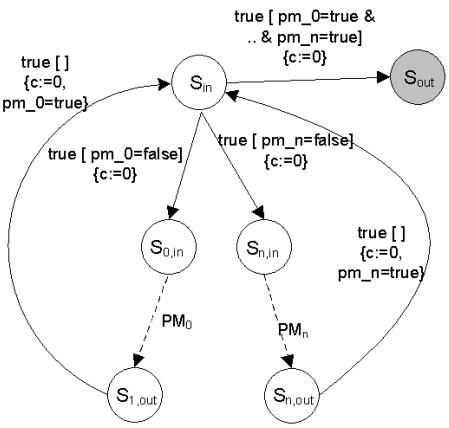

Figure 3. Modeling flow activity

4.4. Limitations

Our framework has the following limitations. The attributes *joinCondition*, *supressJoinFailure* of the *flow* activity are not treated. An activity with *correlation* will be model by adding a variable *status* of properties as in [15]. In that case, we add two transitions *!fault* and *?done* into the partial machine to handle the fault because the standard fault *correlationViolation* must be thrown [1] (i.e. synchronize the fault with *faultHandler* machine).

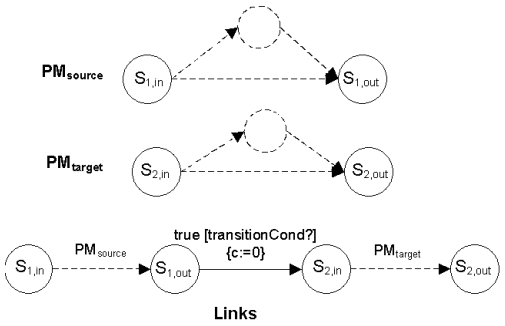

Figure 4. Modeling Links

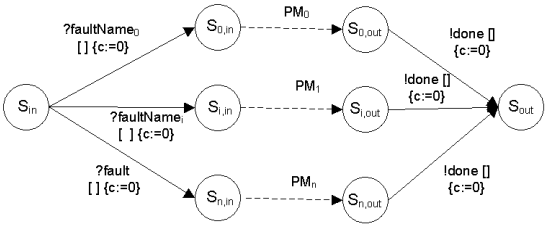

Figure 5. Modeling faultHandler

5. Testing Services Composition Using TGSE

In this section, we study how to test the service composition using the TGSE tool. Two approaches, namely active and passive testing are considered in our framework.

5.1. An Overview of Testing Services Composition Using TGSE

In our framework, we use TGSE to test Web services composition described in BPEL, both for active and passive approaches. To do this, we start by transforming the BPEL description into TEFSM. This transformation can be done automatically by a prototype tool (or by hand) using rules described in the section 4. On active testing approach, a test purpose is requested to guide the test cases generation. On passive testing approach, we use TGSE to check whether a trace is valid according the specification. Fig 6 illustrates the main lines of our methodology.

The current version of TGSE does not supported time invariants on state. Thus, we use only timing constraints on transitions. Moreover, it only supports integer and boolean data types. So, we use many variables to model a BPEL message.

5.2. Active Testing

In order to generate test cases using TGSE, a test purpose must be defined. This test purpose will be modeled as a TEFSM and its actions will be synchronized with the corresponding actions in each TEFSM.

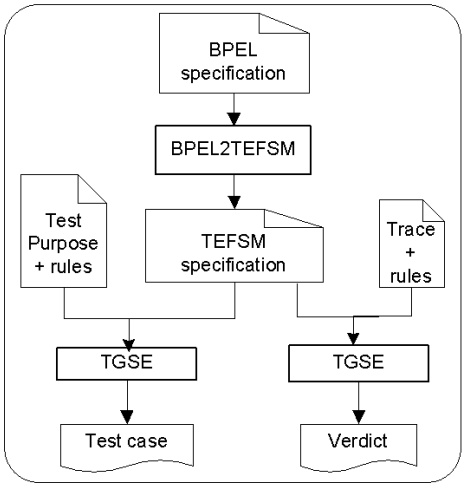

Figure 6. Overrall Method of Testing Services Composition

As TGSE supports timing constraints, we can use a timed test purposes (i.e. test purpose with some timed requirements) to generate timed test cases [16]. In the case, *faultHandler* activity and *eventHandler* activity of a BPEL process will exist (recall that these activities are optional children of BPEL process). The communicating system (CS) will be composed of the three TEFSMs ($M_{activities}, M_{faultHandler}, M_{eventHandler}$) and the test purpose. Thus, each synchronization vector $\vec{r}$ of the CS will have four elements.

TGSE will generate a test case in the XML format satisfying the test purpose. Note that, if a transition condition of a TEFSM depends on input value of messages, then we will use a parameter as a value.

5.3. Passive Testing

TGSE allows runtime coverage of transitions based on guard conditions (by examining variable values). This is very helpful for our purpose of checking BPEL service traces. In our approach, we will model an execution trace as a test purpose (i.e. also referred as a TEFSM) by giving real values to each input message. In that case, all BPEL variables become shared variables. After modeling the BPEL specification and the trace as TEFSMs and declaring their synchronization rules, we use TGSE to verify this system. If the TGSE output is a sequence, it means that this trace is a valid trace of the BPEL specification.

6. A Case Study

In this section, we study an example of the Loan Web Service that is described in 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).

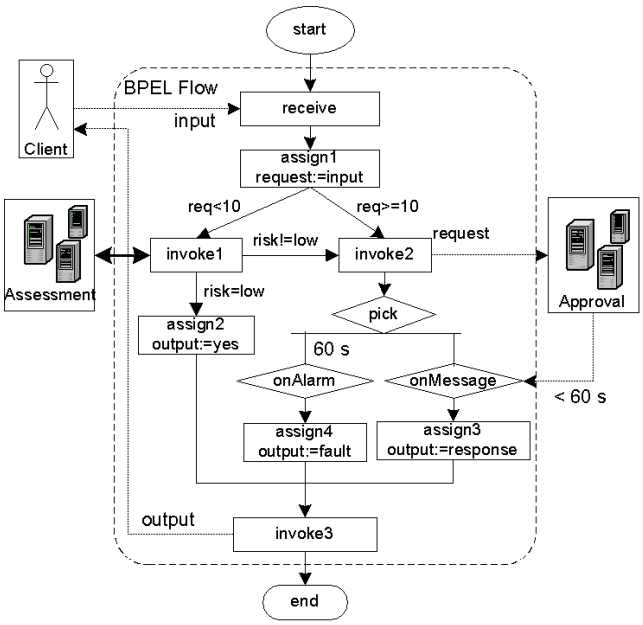

Figure 7. The Loan Web Service

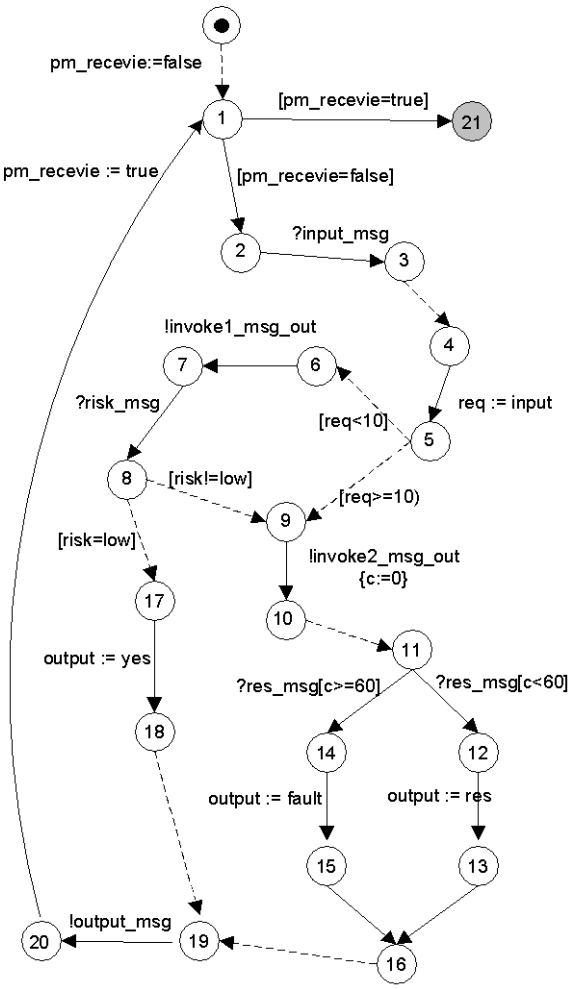

Figure 8. TEFSM of Loan Web Service

6.1. The TEFSM Specification of the Loan Service

Using the rules of the section 4, we have a TEFSM of Loan Web Service of the figure 8 (The dashed lines denote transitions of link variables). In this example, we do not use *stop* variable because the *exit* activity and the *faultHandler* activity do not exist.

In TGSE, an TEFSM is described by: number of state, an initial state, a list of clock variables and a list of transition. Each transition *t* is composed of :

1) source_state(id, name);
2) target_state(id, name);
3) event (*nop* denotes an internal event);
4) guard condition on clocks (# denotes true);
5) guard condition on variable (# denotes true);
6) reset clocks (# denotes empty);
7) update variables (# denotes empty);

The figure 9 describes TEFSM of Loan Web Service in TGSE input format. The value of variables: *request ,risk* and *response* of Approval service is used as parameters (i.e. *p_input, p_risk and p_res*).

6.2. Test Purposes

6.2.1. Test purposes for Scenario #1. This scenario tests the communication between BPEL and Assessment service. Firstly, the Loan process is initiated by receiving an input from the client. It continues receiving the response (i.e. risk_msg) of the Assessment service. Finally, response to the client. Fig 10 gives the test purpose for scenario #1 according to TGSE input format. The rules list for this test purpose are: {*<?input_msg, ?input>, <?risk_msg,?risk>, <!output_msg, !output>*}.

6.2.2. Test purposes for Scenario #2. This scenario tests the communication between BPEL and Approval service with timing constraints. The Loan process is initiated by receiving an input from the client. It receives the response of the Approval service after 30 seconds. Finally, it sends this response to the client. The test purposes for scenario #2 is formulated in TGSE as in Fig 11. The rules list for this test

```
P_AUTO bpel
{
nb_states: 22
initial_state: 0
clocks: t

(0,init), (1,flow_in), nop, #, #, #, pm_receive=0
(1,flow_in), (2,receive_in), nop, #, pm_receive[0,0], #, #
(1,flow_in), (21,flow_out), nop, #, pm_receive[1,1], #, #
(2,receive_in), (3,receive_out), ?input_msg, #, #, #, #
(3,receive_out), (4,assign1_in), nop, #, #, #, #
(4,assign1_in), (5,assign1_out), nop, #, #, #, req=p_input
(5,assign1_out), (6,invoke1_in), nop, #, req[-inf,10[, #, #
(6,invoke1_in), (7,invoke1_s1), !invoke1_msg_out, #, #, #, #
(7,invoke1_s1), (8,invoke1_out), ?risk_msg, #, #, #, risk=p_risk
(5,assing1_out), (9,invoke2_in), nop, #, req[10,+inf], #, #
(9,invoke2_in), (10,invoke2_out), !invoke2_msg_out, #, #, #, #
(10,invoke2_out), (11,pick_in), nop, #, #, h:=t, #
(11,pick_in), (12,assign3_in), ?res_msg, t[0,60[, #, #, #
(11,pick_in), (14,assign4_in), ?res_msg, t[60,+inf], #, #, #
(12,assign3_in), (13,assign3_out), nop , #, #, #, out=p_res
(14,assign4_in), (15,assign4_out), nop , #, #, #, out=-1
(13,assign3_out), (16,pick_out), nop, #, #, #, #
(15,assign4_out), (16,pick_out), nop, #, #, #, #
(8,invoke1_out), (9,invoke2_in), nop, # , risk[1,1] , #, #
(8,invoke1_out), (17,assign2_in), nop, # , risk[0,0] , #, #
(17,assign2_in), (18,assign2_out), nop, #, #, #, out=1
(18,assign2_out), (19,invoke3_in), nop, #, #, #, #
(16,pick_out), (19,invoke3_in), nop, #, #, #, #
(19,invoke3_in), (20,invoke3_out), !output_msg, #, #, #, #
(20,invoke3_out), (1,flow_in), nop, #, #, #, pm_receive=1
}
```

Figure 9. TEFSM specification of the Loan Service for TGSE

```
TESTER test_purpose1
{
    nb_states = 4
    initial_state = 0
    final_state = 3
   (0, init), (1, state1), ?input, #, #, #, #
   (1, state1), (2, state2), ?risk,#, #, #, #
   (2, state2), (3, finish), !output, #, #, #, #
}
```

Figure 10. Test Purpose of Scenario #1 in TGSE

purpose are: {<*?input_msg, ?input*>, <*?res_msg,?res*>, <*!output_msg, !output*>}.

6.3. Test Cases

The test cases that are generated using TGSE cover also internal actions. For instance, a test case for scenario #1 is:

$0 \overset{nop}{\rightarrow} 1 \overset{nop}{\rightarrow} 2 \overset{input\_msg}{\longrightarrow} 3 \overset{nop}{\rightarrow} 4 \overset{nop}{\rightarrow} 5 \overset{nop}{\rightarrow} 6 \overset{invoke1\_msg\_out}{\longrightarrow} 7 \overset{risk\_msg}{\longrightarrow} 8 \overset{nop}{\rightarrow} 17 \overset{nop}{\rightarrow} 18 \overset{nop}{\rightarrow} 19 \overset{output\_msg}{\longrightarrow} 20.$

In our case, we focus on *black-box* testing, it means that we covers only the input events and the output events. We

```
TESTER test_purpose2
{
    nb_states = 4
    initial_state = 0
    clocks: t
    final_state = 3
   (0, init), (1, state1), ?input, #, #, #, #
   (1, state1), (2, state2), ?res, t[30,30], #, #, #
   (2, state2), (3, finish), !output, # , #, #, #
}
```

Figure 11. Test Purpose of Scenario #2 in TGSE

are not interested in the internal events. So, from TGSE result, we will drop internal actions. The test cases for each scenario are shown in Fig 12.

TEST CASE 1 FOR SCENARIO #1
1. *?input_msg (p_req=0)*
2. *!invoke1_msg_out*
3. *?risk_msg (p_risk=0)*
4. *!output_msg*

TEST CASE 2 FOR SCENARIO #2 *(first time)*
1. *?input_msg (p_req=0)*
2. *!invoke1_msg_out*
3. *?risk_msg (p_risk=1)*
4. *!invoke2_msg_out*
5. *?res_msg (p_res=0)*
6. *!output_msg*

TEST CASE 2 FOR SCENARIO #2 *(second time)*
1. *?input_msg (p_req=10)*
2. *!invoke2_msg_out*
3. *?res_msg (p_res=0)*
4. *!output_msg*

Figure 12. The Abstract Test Cases

Note 1: TGSE has sixteen selection modes (i.e. from p0 to p15) concerning the chose of transitions, automata, and synchronization rules . Here, we have used the p15 mode (all random) to run this example. The values of parameters generated randomly are saved in the OutputLp.out file.

7. Conclusions

In this paper, we have presented a methodology for the TGSE tool to test Web Service Composition described in BPEL language. Two test approaches are considered: active testing (generating test cases) and passive testing (verifying traces). We have given some definitions on Timed Extended Finite State Machine (TEFSM), a partial of TEFSM and a Communicating System (CS). TEFSM that we proposed can

enable modeling of BPEL behaviour, and data and timing constraints. We also define some rules to transform a BPEL specification into TEFSMs that is the components of a CS. The Loan Web Service example is used to illustrate our method. In a future work, we will attempt to use this tool to handle integration testing as well as the choreography of Web services.

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, Aug 2007.

[3] A. Gill, "Introduction to the Theory of Finite-State Machines", *Published by McGraw-Hill Book Co..*, 1962.

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

[5] 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*. 2006.

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

[7] Y. Zheng, J. Zhou, P. Krause, "Analysis of BPEL Data Dependencies", *EUROMICRO Conference on Software Engineering and Advanced Applications*, 2007.

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

[9] 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.

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

[11] 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.

[12] E. Bayse, A. Cavalli, M. Nunez, F. Zaidi, "A Passive Testing Approach based on Invariants: Application to the WAP", *Computer Networks*, 48, pp 247 - 266, 2005.

[13] 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.

[14] M. Lallali, F. Zaidi, A. Cavalli, "Timed modeling of web services composition for automatic testing", *3rd ACM/IEEE International conference on Signal-Image technologies and Internet-Based Systems*, China, 2007.

[15] 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*, 2008.

[16] 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.

[17] T. Higashino, A. Nakata, K. Taniguchi and A. Cavalli, "Generating Test Case for a Timed I/O automaton model", *International Workshop on Testing of Communicating Systems*. Budapest, Sep 1999.

[18] I. Berrada and P. Félix, "TGSE : Un outil générique pour le test", *Proc. of CFIP'2005*, March, 2005.

[19] I. Berrada, "Modélisation, Analyse et Test des systems communicants contraintes temporelles : Vers une approche ouverte du test", *PhD thesis of University Bordeaux 1*, Dec 2005.

[20] Y. Yan, Y. Pencole, M.O. Cordier, A. Grastien, "Monitoring Web Service Networks in a Model-based Approach", *3rd European Conference on Web Services*, Vxj, Sweden. 14 - 16 Nov 2005.

[21] Y. Yan, P. Dague, "Monitoring and Diagnosing Orchestrated Web Service Processes", *Proceedings of the 2007 IEEE International Conference on Web Services*, Salt Lake City, Utah, USA. Jul 9-13, 2007.

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

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

[24] 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.