Automatic Timed Test Case Generation for Web Services Composition*

Mounir Lallali [1], Fatiha Zaidi [2,3], Ana Cavalli [1], Iksoon Hwang [1]
[1] TELECOM SudParis - CNRS SAMOVAR
9 rue Charles Fourrier, F-91011 Evry, France.
Email: {Mounir.Lallali, Ana.Cavalli, Iksoon.Hwang}@it-sudparis.eu
[2] Univ Paris-Sud, LRI, UMR 8623, Orsay F-91405;
[3] CNRS, Orsay, F-91405.
Email: Fatiha.Zaidi@lri.fr

Abstract

In order to specify the composition of Web services, WS-BPEL was defined as an orchestrating language by an international standards consortium. In this paper, we propose a method to test composite Web service described in BPEL. As a first step, the BPEL specification is transformed into an Intermediate Format (IF) model that is based on timed automata, which enables modeling of timing constraints. We defined a conformance relation between two timed automata (of implementation and specification) and then proposed an algorithm to generate test cases. Test case generation is based on simulation where the exploration is guided by test purposes. The proposed method was implemented in a set of tools which were applied to a common Web service as a case study.

1. Introduction

Web services are Internet-based components published using standard interface description languages (e.g., WSDL [7]) and universally available via uniform communication protocols (e.g., SOAP [17]). The composition of Web services consists in combining existing Web services to define higher-level functionalities. Business Process Execution Language for Web Service (WS-BPEL) [20] is emerging as the standard composition language for specifying business process behavior based on Web services and became a *de facto* standard. A BPEL process implements one Web service by specifying its interactions with other Web services (called partners).

Testing is a fundamental step in any development process. The use of Web services for business, distributed or critical applications requires an increasing need for efficient testing approaches to detect faults and assess the quality attributes of Web services. For Web services composition, we can distinguish two types of testing: unit and integration testing [4, 18]. Unit testing is used to find bugs on an isolated (separated from partners) composite Web service. Integration testing is used to test this composite service in combination with its partners. In addition, conformance testing establishes that the implementation respects its specification.

A. Bucchiarone *et al.* [4] have defined two approaches for Web services composition testing:

- *White box* approach: In this approach, as BPEL is an executable language, the BPEL description of Web services composition is considered as the source code of the composition. Several structural criteria of coverage based on the code can be applied;
- *Black box* approach: In this approach, an implementation of a composite Web service is tested without any information of its internal structure. The composition of Web services is described by a formal/informal specification from which the tests are generated. This approach is adapted for rapid test scenario testing and quick Web Services prototyping [6].

In this paper, we focus on unit testing of an implementation of a Web services composition, based on *black box* approach and conformance testing. Note that this approach relies on the coverage of particular requirements of the system, which are called test purposes. The BPEL description of Web services composition is considered as the specification of what the system is expected to do and can be produced from the data and flow requirements, and from the WSDL descriptions. To generate tests from the BPEL specification, a formal model of BPEL is required. For that purpose, inspired on our previous transformation of BPEL into Timed Extended Finite State Machine for Web Service

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

(WS-TEFSM) [14], we propose a BPEL transformation into an Intermediate Format (IF) language which is associated to an efficient open-source simulator. Using this simulator, we can explore by an exhaustive simulation the state space of the model and generate test cases. Moreover, this language is based on communicating timed automata extended with variables, which can handle the BPEL constructs.

We propose a "black box" approach based on test purposes and on a conformance relation. To guide the test generation, we propose a test generation algorithm with (timed) test purposes. The conformance relation is defined as a *timed-traces-inclusion* relation. We have introduced in the IF simulator a *Hit-or-Jump* exploration strategy [5] to generate the test cases. The proposed methods are implemented by a set of tools, one is devoted to the BPEL transformation approach, and the other one to implement the test generation algorithm by extending the IF toolset [24].

The rest of this paper is organized as follows. Section 2 reviews some previous work on Web services composition testing. In section 3, BPEL, the IF automaton and the IF system are defined. Section 4 gives an overview of our timed testing method of Web services composition. Section 5 details the transformation of BPEL into IF model. Section 6 provides the conformance relation used in this testing method and the test case generation algorithm. The prototype tools that implement respectively the test generation algorithm and the BPEL transformation approach, are presented in section 7. A case study is presented in section 8. Finally, section 9 concludes the paper and outlines future work.

2. Related Work

In the last years, several techniques and tools have been developed to test Web services. Various approaches for service composition testing were analyzed by [4] including unit testing, integration testing, black box testing and white box testing of choreographies and orchestrations. Most of work on testing of Web services focused on unit testing of BPEL composition. A number of work has been proposed to test BPEL compositions based on model checking techniques and structural coverage criteria [10, 26]. [26] proposed a test generation framework for BPEL composition. The BPEL semantics were modeled by Web Service Automata (WSA) as an intermediate formal model and then the SPIN model checker and the NuSMV model checker were used to generate test cases. In [10], the BPEL composition was transformed into PROMELA directly, i.e., without any other intermediate formal model, and the SPIN model checker was used to generate test cases under transition coverage criterion.

Some of work presented frameworks for white box testing without model checker. [15,19] presented a framework for white box testing of BPEL composition. In [12] a formal model for structural-based workflow framework, which is not a BPEL, was introduced where structural-based testing is used. The problem of [12, 15, 19] is that they do not consider the automatic test case generation. Some of BPEL editors such as Oracle BPEL Test Framework [21] and ActiveBPEL Designer [1] provide manual unit testing of BPEL compositions. Some of work focused on testing the WSDL descriptions of Web services. [11] presented a test case generation approach based on WSDL descriptions. [23] proposed a framework for testing Web service based on WSDL descriptions and TTCN-3. Some of open source tools for testing WSDL descriptions of Web services were developed such as soapUI [8] and TestMaker [22].

In our previous work [14], a timed modeling of BPEL composition, and a formal transformation of BPEL process into a new formalism WS-TEFSM were proposed. Actually, WS-TEFSM is not supported by any toolset, and consequently, it is not used in the proposed test generation method. However, we use the IF model [2] which is based on a communicating timed automata extended with variables and is supported by a toolset [24].

3. Preliminaries

We introduce in this section the BPEL language. We present the IF timed automaton and the IF system [3, 16] which are used to model BPEL composition and to describe the timed test case generation algorithm.

3.1. BPEL

BPEL [20] provides constructs to describe complex business processes that can interact synchronously or asynchronously with their partners. A basic process in BPEL is defined as a root element consisting of one or more child elements describing *partners*, a set of *variables* that keep the state of the process, *correlation Sets*, *fault* and *compensation* handlers and activities. These latter define the interaction logic of a process and its partners. The BPEL activities that can be performed by a business process instance are categorized into basic (e.g., *wait*, *empty*), messaging (e.g., *receive*, *reply*) and structured activities (e.g., *flow*, *pick*).

3.2. IF Timed Automaton

A communicating system described using IF language [2] is composed of active process instances running in parallel and interacting asynchronously through shared variables and *signals* via *signalroutes* or by direct addressing. A process instance can be created and destroyed dynamically during the system execution. It has local data and a private FIFO buffer. Each IF process is described as

a timed automaton extended with discrete data variables, communication primitives and urgency attributes on transitions, i.e., IF-TA.

Definition 1 (IF Timed Automaton) *The IF Timed Automaton is a tuple* $TA = (Q, Act, X, T, q_0)$ *where:* Q *is a finite set of states (including stable and unstable states),* q_0 *is the initial state,* Act *is a finite set of actions,* X *is a set of typed variables (including data variables, timers and clocks) and* $T \subseteq Q \times G(X) \times 2^{Act} \times U \times Q$ *is a set of transitions such that* $G(X)$ *is a set of Boolean guard conditions on data variables, timers and clocks, and* $U = \{eager, lazy, delayable\}$ *is the urgency set.*

The IF model considers the standard domains: $\mathbb{B}$ of Boolean, $\mathbb{Z}$, $\mathbb{R}$ and an infinite set of process identifiers. An *unstable* state is invisible at execution, but a *stable* state is atomic. The actions in Act represent observable actions. The label $\tau \notin Act$ denotes an *internal* action (that is unobservable). We note Act_τ the set $Act \cup \{\tau\}$ which includes signal input and output, assignment action, dynamic process creation and destruction. Each transition ($t = q \xrightarrow[u]{g\ a} q' \in T$) is annotated with a set of guards g, a set of actions $a \in Act_\tau$ and an urgency attribute $u \in U$.

The clocks and timers values are real numbers. They can be set or reset. Time progresses in states and transitions take zero time to be executed. The transition urgency [16] is used to control the time progress: (i) *eager transition*: is urgent as soon as it is enabled, and blocks time progress; (ii) *lazy transition*: is never urgent, and never block time progress; (iii) *delayable transition*: allows waiting as long as time progress does not disable it.

The semantics of an IF-TA TA is defined by a labeled transition system denoted $[TA] = (S, s_0, \Gamma, \Rightarrow)$ where $S = \{s_0, s, s', ...\}$ is a set of semantic states, s_0 is an initial state, Γ is the labels set (actions and delays) and $\Rightarrow$ is a transition relation (see [3, 16] for more details).

In these semantics, we can distinguish *discrete* and *timed* transition. The *timed* transition, denoted by $s \overset{d}{\Rightarrow} s \oplus d$, indicates that the IF timed automaton does not execute any action (does not change state), but increments the current value of the clocks and timers d. A *timed* transition does not block any transition. In the *timed* transition, the time progresses only in a *stable* state by steps until a positive amount of time $d \geq 0$. It progresses in this state only if: (i) there is no enabled *eager* transitions; and (ii) for each enabled *delayable* transition tr, $\exists d' \geq 0$ such that tr cannot be disabled while the time progress by d'. The *discrete* transition, denoted by $s \overset{a}{\Rightarrow} s'$ such that $q \xrightarrow[u]{g\ a} q' \in T$, indicates that if the guard g is *true*, then the automaton follows the transition by executing the action a, changing the current values of the data variables by executing all the assignments, changing the current values of the clocks and timers by executing all the time setting/resetting, updating the buffers contents of the system by consuming the first signals required by input actions and by appending all signals required by output actions, and moving in the next state q'.

To obtain an IF system model, IF timed automata can be composed by using an associative and commutative parallel operator $||$. Let TA_i be an IF timed automaton which models a process of the system, and $[TA_i]$ be its semantics. A system SYS is a set of processes such that $SYS = \{TA_1, TA_2, ..., TA_n\}$. Its semantics [3, 16] is defined by a labeled transition system $[SYS]$ where $[SYS] = (S, s_0, \Gamma, \Rightarrow) = [TA_1] \,||\, [TA_2] ... \,||\, [TA_n]$.

4. An Overview of Timed Test Generation Method for Web Services Composition

In general, BPEL can be considered to be either a reference specification or an executable specification. In this paper, we consider a BPEL specification as a reference specification. In order to describe a BPEL specification, it is needed to consider (i) the WSDL description [7] of the Web service and its partners and (ii) the control and data flow requirements of the Web services composition. Formal or semi-formal specification (e.g., UML, BPMN) of Web services composition can also be used, if it exists, to describe the BPEL specification. In this section, we explain briefly the proposed test generation method. In the proposed method, a set of test cases, which can be used in testing of time requirement, are generated automatically based on the BPEL specification of the Web services composition. Figure 1 illustrates the overall methodology.

The first step is to transform the BPEL description into IF model. Transformation can be done automatically by a prototype tool, called BPEL2IF. More detailed explanations about the transformation and the prototype tool are given in section 5 and section 7 respectively. In order to generate test cases, we need to describe test purposes. A test purpose represents a specific behavior of the specification (i.e., IF model), including timing requirement and is composed of a set of actions or clock values that refer to BPEL process variables, timeouts, activities duration, messaging BPEL constructs, e.g., *receive*, *onMessage*, etc. Once we have the IF model of the Web services composition and the test purposes, we can generate test cases automatically using TestGen-IF. The detailed explanation of the TestGen-IF and the test case generation algorithm, which is based on partial state space exploration guided by test purposes, are presented in section 7 and section 6.2 respectively. Finally, a conformance relation, which is called *timed-traces-inclusion*, is presented in section 6.1.

In the proposed test generation method, the IF model is used for the following reasons: (i) the IF language is based on a communicating timed automata extended with variables (ii) the IF is supported by a toolset [2] which enables the simulation of the process execution and the validation

of the model, and (iii) it provides programming APIs, such as exploration strategies, allowing us to implement our own functionalities.

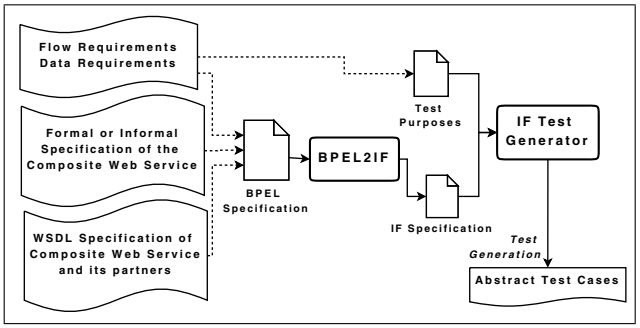

Figure 1. Timed Testing Method of Web Services Composition

5. Transforming BPEL into IF

In this BPEL transformation approach, we consider data, predicates, messages and partner link handling, basic and structured activities, fault propagation, termination and synchronization of activities, scopes, BPEL process element, WSDL interface of BPEL process clients and partners. Each BPEL activity is described as an IF process. In particular, a non basic activity is transformed into an IF process which can be dynamically created its sub-activities processes. Note that each IF process can be the parent of other IF processes or a child of another one. The BPEL process is described by an IF system which its IF processes are executed in parallel and interact asynchronously through *signals* (IF messages) via *signalroutes* (communication buffers) [24]. In this section, we present the transformation of the main constructs and functionalities of BPEL including data, partner links, fault handlers, some basic and structured activities (e.g., flow, pick), synchronization of activities, fault and termination handling, and finally BPEL process and its client and partners. Note that in this section, we use the Loan Web service given in Section 8.

5.1. Data, Message and Partner Links

A BPEL variable can be declared as XML schema element, XML schema type and WSDL message type. These types (defined in the WSDL interfaces) are transformed in simple or complex IF types by using the IF type constructors (enumeration, range, array and record).

In the IF simulator, all the possible input parameters are given during simulation. As we cannot control those values, in general it is common to be faced to state explosion problem. In order to reduce the problem size, we limit the values of some input parameters. Note that choosing those parameters should be done carefully by an expert who has the best knowledge of the Web services.

The BPEL predicates (e.g., join condition attribute of the *flow target* element), which call external functions or contain variables, are replaced by Boolean variables. These predicates are mutually exclusive, e.g., conditional branches of *if* activity. For this end, we define (as in [26]) a set of logical constraints on these predicates. Their resolution permits to generate Boolean values covering all activities branches. Because the BPEL partner links are bidirectional, each WSDL partner link type is transformed in one or more IF enumeration types which contains the partner link name, and its associated *portType* and *operation*. For example, the *client* partner link type is described by two IF types:

```
type plType_fromClient = enum LoanService_LoanService_initiate
type plType_toClient = enum LoanService_LoanServiceCallback_onResult;
```

Each WSDL message type is described as an IF *signal* (a message used to communicate between IF processes). For instance, the request message: *<**message** name="RequestMessage"> <part name="payload" element="s1:loanApplication"/> < **/message**>* is described as ***signal** RequestMessage(plType_fromClient,RequestMessageType)* where *RequestMessageType* is a complex type with *payload* as member, and *plType_fromClient* is a partner link type.

Each BPEL partner link can be associated to one or more IF *signalroutes* (i.e., communication buffers used between IF processes [24]). However, the BPEL partner links are bidirectional when IF *signalroutes* are unidirectional. For this reason, we associate, for each partner link, at least two *signalroutes* (*fromPartnerLink* and *toPartnerLink*). The first one is used to transport the input messages while the second one is used to transport the output messages. Let *RequestMessage* and *ResultMessage* be two messages. For instance, the partner link *Client* used in *receive* and *invoke* activities (of the Loan service) is associated to the two following *signalroutes* (i.e., *fromClient* and *toClient*) where *env* is the IF environment [3, 16] and *IntermediateEnv* is an intermediate environment process (see Section 5.9):

***signalroute** fromClient()*	***signalroute** toClient()*
***from** IntermediateEnv **to** ReceiveProcess **with** RequestMessage;*	***from** invokeProcess **to** env **with** ResultMessage;*

5.2. Fault Propagation

An IF process (describing a BPEL activity) forwards a *fault* message to its parent process when it receives a *fault* message from its sub-processes (describing its sub-activities). The *fault* message is propagated until the fault handler of one of the enclosed scope handles this fault. We model this propagation by allowing each IF process (de-

scribing an enclosing activity) to receive a *fault* message in each state and send it to its parent.

5.3. Termination

In BPEL, the termination is activated by an *exit* activity (first case) or when a fault is thrown by an *invoke* activity or a *throw* activity (second case). We use *terminate* and *done* messages to handle the termination as follows:

- *Case 1*: when an *exit* activity is reached, the *exit* process assigns *true* to the *exit* variable associated to the IF process corresponding to the BPEL process element. This latter initiates the termination of its children processes by propagating the *terminate* message. Before terminating an IF process, all of its children must be terminated. For this, the parent process propagates the *terminate* message to its children and waits to their termination, i.e., receiving *done* message from all its children. If a terminated IF process has no children or all its children have terminated (normally or abnormally), it stops immediately its execution and sends a *done* message to its parent.
- *Case 2*: each IF process finishes its control flow when it receives a *fault* message and sends this message to its parent. If one of the enclosing scopes can handle this fault, the fault propagation is stopped.

The *terminate* and *done* messages must have a higher priority than a *fault* message which has a higher priority than a normal incoming message (generated from BPEL). The *empty*, *throw* and *rethrow* activities may be allowed to complete, and. the started *exit* activity must not be terminated [20]. We detail the termination of each activity below when we describe the IF process of each activity.

5.4. Synchronization of Activities

Flow activity provides concurrency and synchronization dependencies between its sub-activities. This synchronization is expressed by a *link* construct. Each BPEL activity has optional nested standard elements *source* and *target* [20] that define a link which connects two activities and can change the sequential order of activities. An activity may declare itself to be the target (respectively the source) of one or more links by including one or more target (respectively source) elements. An activity can be the source of multiple links, thus allowing multiple branches to be executed in parallel. Each link can have an associated *transition condition* attribute. The source sets the link guard to the logic value of *transition condition* or to *true* if this attribute is not specified. The target activity may have a *join condition* attribute specified. It is executed only if its attribute is evaluated to *true*. If the *join condition* is not specified, it is interpreted as an *or* logical operator between the incoming links.

In order to not complicate the task of the *source*, *target* and *flow* activities, we used a specific IF process, called *linksManager*, to handle the links and the synchronization of the *flow* sub-activities. This *linksManager* uses source (respectively target) messages to communicate with source (respectively target) activities. Each target or source activity declares itself to *linksManager*. When a source activity finishes, for each link, it evaluates the guard and sends a *source* message to *linksManager* with the guard value. *LinksManager* sends a *target* message with this guard value to the target activity which must wait until the source activities finish. When a target activity receives a target message of all incoming links, it evaluates its guard (i.e., *join condition* or logical formula on incoming links). When this guard is not satisfied, a target activity propagates the *join failure* fault to its parent.

5.5. Basic Activities

Basic activities [20] describe elemental steps of the BPEL process behavior. Basic activities are: *invoke*, *receive*, *reply*, *assign*, *wait*, *empty*, *exit*, *throw* and *rethrow*. Each basic activity is described by a simple IF process that executes one step and sometimes handles its forced termination. The *wait* activity is described by an IF process that delays for a certain period of time or until a certain deadline by using a *eager* urgency and a *clock* guard. The *exit* activity is described as an IF process that terminates immediately the IF process instance of the BPEL process by assigning to the *exit* variable (of the BPEL process element) the value *true*. The IF process of the *invoke*, *receive* or *reply* activity uses the IF communicating actions (i.e., *input* and/or *output*). The *assign* process uses the IF assignment operation.

The IF process for the *wait* activity (< *wait for="PT30S"*/ >) is illustrated in the Figure 2. This process initializes its local *clock* and stops after waiting for 30 seconds. It terminates when it is interrupted by a *terminate* signal.

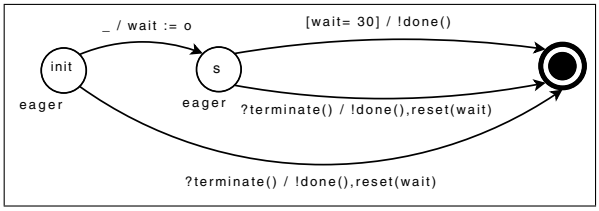

Figure 2. The wait process

We note that all the input actions of the IF processes describing the messaging constructs (*invoke*, *receive* and *onMessage*) are not urgent and never blocks time progress. They have a *lazy* urgency (see Section 3.2).

5.6. Structured Activities

Structured activities describe the order in which its sub-activities are executed [20]. *Sequence*, *if*, *while* and *repeat Until*, and serial *for each* activities provide a sequential control. *Flow* activity provides concurrency and synchronization between activities. *Pick* activity provides a choice controlled by events. We detail the transformation of *flow* and *pick* activities into IF processes.

5.6.1. Flow Activity

The *flow* activity allows to specify one or more activities to be performed concurrently [20]. The links defined in the *flow* activity permit to enforce precedence between sub-activities, i.e., synchronization. The *flow* activity is described as an IF process that creates simultaneously the IF processes of all the enclosed sub-activities. The synchronization of the *flow* sub-activities is handled by the l*inksManager* process described in section 5.4. The *flow* process completes when all sub-activities are completed. It is interrupted when it receives a *terminate* or a *fault* message. In this case, it terminates its behavior and applies a forced termination to its active sub-activities processes.

5.6.2. Pick Activity

The *pick* activity waits for one event occurrence, then executes the associated activity [20]. It is described as an IF process that waits the occurrence of one event, creates the IF process of the associated activity of each event and waits for its termination. The event of the *pick* activity has two forms: *OnMessage* considered as an IF input action and *onAlarm* considered as an IF waiting action. When the *pick* activity process is interrupted, by receiving a *terminate* or *fault* message, it terminates immediately when no branch is selected, or the forced termination is applied to the IF process of the selected branch sub-activity. All the input actions (modeling the *onMessage* elements) of the *pick* process are not urgent and do not block time progress. These input actions have a *lazy* urgency. In the contrary, the time-out action (modeling the *onAlarm*) is always urgent and it has a *eager* urgency.

5.7. Defined Fault Handlers

The *fault handler* of a *scope* or a *BPEL process* is a set of catch clauses defining how the *scope* should respond to different types of faults [20]. The *fault handlers* are described as an IF process which combines an *if* activity applied to various sequences of *catch* or a *catchAll* activities (conditional branches) and the creation of its IF sub-activities processes. Each *catch* branch is considered as a comparison between the propagated fault and its handled fault. The *catchAll* branch is used to catch all the faults that are not handled by the defined *catch* branch.

5.8. BPEL Process Element

A BPEL process always starts with the *process* element, i.e., the root of the BPEL document. It is composed of the following optional children: *partner links*, *variables*, *correlation sets*, *fault handlers*, *compensation handlers* and *event handlers*. Note that *compensation handlers* are not considered in this work. The *process* element contains one main activity declaration representing the process workflow definition. This BPEL *process* element is described as an IF process which creates two sub-processes, respectively, of its primary activity and its event handlers. The transformation of its optional children is detailed above. When the IF process of the BPEL *process* element receives a *fault* or when its exit variable is assigned to *true* by the *exit* process, it applies a forced termination to the IF processes of its primary activity and event handlers by propagating a *terminate* message and waits for their termination. Afterwards in the case of *fault* message reception, the IF process of the BPEL *process* element creates the IF process of its fault handlers to handle the fault and waits for its termination.

5.9. BPEL Client and Partners

The client and the partners of a BPEL process are considered as the environment of the IF system. In IF, the communication between two IF processes and the communication between an IF process and the environment are handled in a different way. Each IF process has its own FIFO queue and the messages from other processes are stored in this queue. As the communication is asynchronous it may take time to consume the messages in the FIFO queue. The messages from the environment are, however, consumed as soon as the environment sends a message, i.e., the communication between the IF environment and IF process is synchronous.

In the proposed transformation, the IF process of BPEL messaging constructs (in particular *receive*, synchronous *invoke*, *onMessage* and *onEvent*) can receive the messages from the environment as well as from other processes (e.g., *done*, *fault* and *terminate* signals). In this case the order of consumption of the messages cannot be guaranteed to be the same of its reception as is received. The messages in the queue, which is already received from other IF processes, may need to wait for the processing of newly incoming messages from the environment. In order to solve this problem, we introduce an intermediate environment process, called *IntermediateEnv*. Every message from the environment is sent to this *IntermediateEnv* process and then it passes each message to the appropriate destination. As each signal is defined to have only one destination in the proposed model,

the Intermediate process can distribute the incoming messages from the environment to proper destinations. By introducing of the *IntermediateEnv* process, we can guarantee the order of consumption of messages as all the messages coming from the environment are passed to the FIFO queue of the IF process.

6. Timed Test Case Generation Using Test Purposes

In this section, we define the concept of timed traces and timed test cases of the IF system, and the conformance relation (inspired by [9, 13]). Finally, we detail our timed test case generation algorithm.

6.1. Conformance Relation and Timed Test Case

In the following, we define $\Gamma = Act \cup \mathbb{R}_+$ as a set of observable labels (actions and delays), and $\Gamma_\tau = Act_\tau \cup \mathbb{R}_+$ as a set of labels included internal actions. Let $SYS = \{TA_1, TA_2, ..., TA_n\}$ be a system, and $[SYS] = (S, s_0, \Gamma_\tau, \Rightarrow)$ its semantics. Let $s, s_0, s_1, ..., s_n \in S$ be semantic states. Let $Seq(\Gamma) = (\Gamma)^*$ be the set of all finite timed sequences over Γ. A *timed sequence* $\sigma \in Seq(\Gamma)$ is composed of actions a and non-negative reals d such as: $s \stackrel{d}{\Rightarrow} s \oplus d$ and $s \stackrel{a}{\Rightarrow} s'$ (*timed* and *discrete* transitions defined in Section 3.2). $\sigma_\varepsilon \in Seq(\Gamma)$ is the *empty* sequence.

Let $\Gamma' \subseteq \Gamma$ and $\sigma \in Seq(\Gamma)$ a timed sequence. $\pi_{\Gamma'}(\sigma)$ denotes the projection of σ to Γ' obtained by deleting from σ all actions not present in Γ'. $Time(\sigma)$ denotes the sum of all delays in a sequence σ. Let $\sigma = \sigma_1.\sigma_2...\sigma_n$ are a timed sequence, $s \stackrel{\sigma}{\Rightarrow}$ is used to denote that there exists s_n and $s \stackrel{\sigma_1}{\Rightarrow} s_1 \stackrel{\sigma_2}{\Rightarrow} s_2... \stackrel{\sigma_n}{\Rightarrow} s_n$. A state $s \in S$ is reachable if $\exists \sigma \in Seq(\Gamma) \mid s_0 \stackrel{\sigma}{\Rightarrow} s$. The set $Reach(SYS)$ denotes the reachable states of the system *SYS*. This system *SYS* is *non-blocking* if $\forall s \in Reach(SYS)\ \exists d \in \mathbb{R}_+\ \exists \sigma \in Seq(\Gamma_\tau) \mid time(\sigma) = d \wedge s \stackrel{\sigma}{\Rightarrow}$. The *non-blocking* property ensures that a system *SYS* does not block time if it not executes any action. The *observable timed traces* of a system *SYS* is defined by: $Traces(SYS) = \{\pi_\Gamma(\sigma) \mid \sigma \in Seq(\Gamma_\tau) \wedge s_0 \stackrel{\sigma}{\Rightarrow}\}$.

Let SYS_S and SYS_I two systems which model respectively the specification of a composite Web service (a BPEL description) and its implementation. We assume that SYS_S and SYS_I are *non-blocking*. In our testing method, we consider *timed-traces-inclusion* noted $\preccurlyeq_{Traces}$ as *conformance* relation where the time delays is considered to be observable actions [13]. The *conformance* relation $SYS_S \preccurlyeq_{Traces} SYS_I$, requires each observable sequence of SYS_S to be an observable sequence of SYS_I: $(SYS_S \preccurlyeq_{Traces} SYS_I) \Leftrightarrow (Traces(SYS_S) \subseteq Traces(SYS_I))$.

A Timed test case is an observable timed trace that validates some timed requirements (e.g., timed test purposes) and generate a *pass* or *fail* verdict. It generates the *pass* verdict if all test purposes are satisfied, else it generates the *fail* verdict.

6.2. Timed Test Case Generation Algorithm

In this section, we define an algorithm to generate timed test cases from timed test purposes and a composite Web service specification given in IF system (i.e., a parallel composition of IF timed automata modeling the BPEL process element and its sub-activities). This algorithm is based on *Hit-or-Jump* strategy which is a generalization of random walks and structured search algorithms. For this test case generation, we have adapted *Hit-or-Jump* strategy to IF timed automaton. We assume here that the IF system modeling the specification of the composite Web service and its implementation are *non-blocking* models. The generated timed test cases are used to check the conformance of a composite Web service implementation to its specification.

Starting from the initial state of the IF system and considering a search *depth limit* and a set of timed test purposes to be satisfied (represented by events or timed constraints), a partial search is conducted from the current state s_i of the reachability graph until:

- *Hit Condition*: reached a state s_j (a Hit) where one or more test purposes are satisfied. The sequence from s_i to s_j is then concatenated to the test sequence, the test purposes set is updated and a partial search is started again from s_j;
- *Jump Condition*: reached a search *depth limit* without satisfying a test purpose. One leaf node (i.e., a state s_j) of the partial search tree is then selected, the sequence from s_i to s_j is concatenated to the test sequence, and a partial search is started again from s_j.

The algorithm terminates when all the test purposes are satisfied or there is no transition to explore. The construction of the complete system reachability graph is not required by this algorithm. The test case generation algorithm is illustrated in Figure 3.

7. Implementation and Prototype Tools

We have developed two prototype tools. The first tool, called BPEL2IF, is used to transform BPEL into IF. The second tool, called TestGen-IF, is used to generate timed test cases from an IF specification and test purposes.

Initial condition:

- The system is in an initial state $s_0 = (q_0, \upsilon_0, \nu_0, \omega_0)$;
- The set of timed test purposes to be satisfied is $TP = \{tp_1, tp_2, ..., tp_m\}$.
- The timed test sequence seq is empty, i.e., $seq = \sigma_\varepsilon$;
- The search *depth limit*;

Termination: The algorithm terminates when all the timed test purposes are satisfied, i.e., $TP = \emptyset$;

Execution:
Repeat

① **Hit**: From the current system state s_i conduct a search by exploring all possible transitions: *discrete* transition $s_i \stackrel{a}{\Rightarrow} s_{i+1}$ or *timed* transition $s_i \stackrel{d}{\Rightarrow} s_i \oplus d$ until ⓐ or ⓑ:

ⓐ Reach a state s_j such that $s_i \stackrel{\sigma}{\Rightarrow} s_j$ and **forall** k such that $s_j \models tp_k : TP = TP \setminus \{tp_k\}$ - a Hit. **Then**:

(i) Concatenate the sequence σ from s_i to s_j to the test sequence: $seq := seq.\sigma$;
(ii) Move to ①.

ⓑ Reach a search *depth limit*. **Then** move to ②.

Until ($TP = \emptyset \vee$ "no transition to explore").
IF $TP = \emptyset$ **Then return** *seq* **else** "no test sequence!".

② **Jump**:

(i) A partial searched tree has been constructed, rooted at s_i;
(ii) Examine all the tree leaf nodes, and select one (s_j such that $s_i \stackrel{\sigma}{\Rightarrow} s_j$) uniformly and randomly;
(iii) Concatenate the sequence σ from s_i to s_j to the test sequence: $seq := seq.\sigma$;
(iv) Arrive at s_j - a Jump.
(v) Move to ①.

Figure 3. Test Case Generation Algorithm

7.1. BPEL2IF tool

BPEL2IF can transform into IF the Web services composition described in BPELWS 1.1 or in WS-BPEL 2.0. It implements our BPEL transformation approach defined above in section 5. A BPEL and WSDL descriptions of a composite Web services and a WSDL description of its partners are given as inputs to BPEL2IF. The WSDL documents are used to transform the data types, messages, *portTypes* and partner links into IF data, *signals* and *signalroutes*. The BPEL document is processed as a tree where each node corresponds to a BPEL construct. A depth-first walk of the BPEL tree is performed and associated XSL transformation rules are applied to each node (i.e., BPEL construct) in order to produce an IF specification document. BPEL2IF uses two different transformation rules, one for each BPEL version. It is implemented in *Perl* and *XSL* languages.

7.2. TestGen-IF Tool

The TestGen-IF tool is based on the IF simulator [24]. It uses the IF simulator libraries which provides some functionalities for on-the-fly state-space traversal. TestGen-IF implements the timed test case generation algorithm presented in section 6.2. An IF specification document (.if file) and timed test purposes document (.tp file), and the jump depth value, i.e., *maxdepth* parameter, are given as inputs to TestGen-IF. During the test generation as in [25], when a timed test purposes is satisfied, a *Hit* message, the test purpose description and the number of timed test purposes to be satisfied are displayed. The *jump* information is also displayed when a jump is done. The generation terminates when all the timed test purposes are satisfied or when the system exploration terminates. This tool execution generates two files: "*output.stat*" containing statistics about the test generation process (number of jumps and visited states, generation duration, test case length, etc.), and "*output.sequence*" (in Aldebaran format) containing the timed test cases. This tool is implemented in the same implementation language of the IF simulator, i.e., C++ language.

7.2.1. Test Purposes Formulation in TestGen-IF

In order to formulate timed test purposes, we define some constraints used in the TestGen-IF tool: (i) *States constraints*: expressing that a system can be in a certain state; (ii) *Actions constraints*: corresponding, in particular, to signals actions (e.g., input signal, output signal) and describe that an action can be executed in a state (optionally) at a certain time; (iii) *Clock constraints*: expressing that a clock can have a certain value, optionally in a certain state.

Timeouts and deadlines can be described by clock constrains. The flow requirements can be described by states and actions constraints. We will extend this test purposes formulation by adding data constraints, duration action constraints and complex clock constraints. For instance, the constraint "***action*** $=$ *input sg* ***in*** *state* $= s$ ***when clock*** $c = d$" describes that the signal *sg* can be received in the state s when the clock $c = d$.

7.2.2. Timed test cases

The timed test cases are generated from the output file "*output.seq*" of TestGen-IF. This file contains the timed test sequences generated from the IF specification. To obtain a timed test cases, these test sequences should be filtered according to the input actions, output actions and delays.

8. Case Study: The Loan Web Service

In this section, we use the Loan Web service (see Figure 4) which is given as case study in [21]. This process receives a loan application document from the User. It invokes the asynchronous Credit Rating service by sending the document and uses a BPEL *pick* activity for one of the following cases: (1) to receive an asynchronous response from the partner service; (2) to set the credit rating of the loan document (default loan) to -1 after a timeout (e.g., 30 seconds). The value -1 represents the impossibility of the

credit rating evaluation. The partner service sets the credit rating according to the loan amount and returns the loan application document to the Loan service. If the loan amount is greater than 10000, it waits 30 seconds for the partner service to process it before raising a timeout. The partner service sends an event message (processing progress status) before sending its response. Finally, the Loan service sends the loan application document to the User.

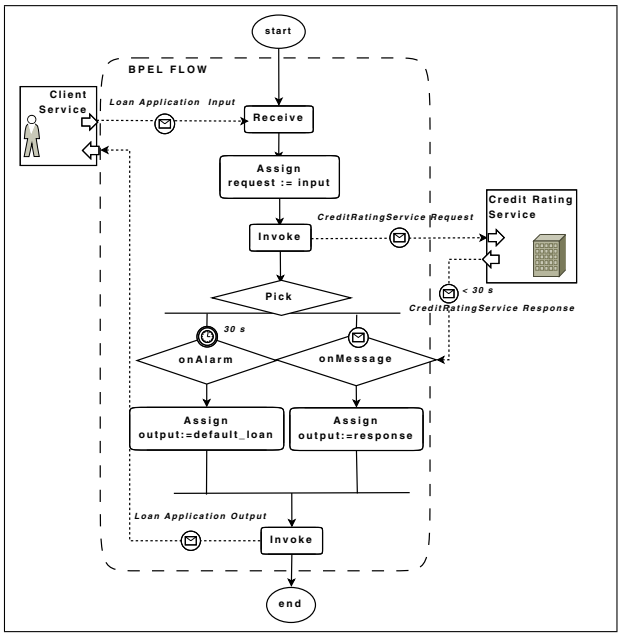

Figure 4. The Loan Web Service

We use our testing method to generate timed test cases for the Loan service. Our objective is to test the two *pick* activity branches. We define, for instance, the two following test scenarios: (1) *Scenario #1:* receive a response of the Credit Rating service after 5 seconds and reply with this response to the User; (2) *Scenario #2*: wait 30 seconds for a response from the Credit Rating service before replying with a default loan to the User.

8.1. The Specification of the Loan Service

The IF specification is the result of the transformation of the BPEL specification of the Loan service and the WSDL description of its partner (i.e., the Credit Rating service). It contains 10 processes describing the BPEL process and its sub-activities. It contains also the declaration of 5 *signals* describing the BPEL messages, 3 internal *signals* (i.e., *done*, *fault* and *terminate*), and 4 *signalroutes* associated to the BPEL partner links. The User and the Credit Rating services are described by the IF environment, i.e., *env*. Some metrics of the IF specification of the Loan service are given in Table 1. The loan application document is extracted from the WSDL description of the Loan service, and is described as an IF *record* type given by: ***type*** *loanApplication* **= *record*** *SSN string; email string; customerName string; loanAmount real; carModel string; carYear string; creditRating integer;* ***endrecord***.

Lines	480	Processes	10
Signalroutes	5	Signals	7
States	260	Transitions	431
Clocks	1	Global Variables	7

Table 1. Metrics of the IF specification of the Loan Service

In our case study, we reduce the *string*, the *real* and the *integer* types to a small range. A loan application value sent by the User (respectively by the Credit Rating service) is denoted by *loan_request* (respectively *loan_response*). *default_loan* denote the loan application value sent by the Loan service to the User after timeout (i.e., a credit rating of the loan has a value -1).

8.2. Timed Test Purposes

The timed test purposes for the two scenarios are described according to the TestGen-IF formulation.

8.2.1. Test purposes for Scenario #1

The Loan service is initiated by receiving the loan application document from the *user*. It receives the response of the Credit Rating service after 5 seconds. Finally, it sends this loan application document to the User. The test purposes for *scenario #1* is formulated as:

$TP_1 = \{tp_1, tp_2, tp_3\}$
$tp_1 = \{$***action*** $=$ *input LoanServiceRequestMessage*$\}$;
$tp_2 = \{$***action*** $=$ *input CreditRatingServiceResultMessage* ***when clock*** $c = 5\}$;
$tp_3 = \{$***action*** $=$ *output LoanServiceResultMessage*$\}$.

8.2.2. Test purposes for Scenario #2

The Loan service is initiated by receiving the loan application document from the User. It waits a response of the Credit Rating service for 30 seconds. After, it sends a default loan to the User. The timed test purposes for *scenario #2* is formulated as:

$TP_2 = \{tp_1, tp_2, tp_3\}$
$tp_1 = \{$***action*** $=$ *input LoanServiceRequestMessage*$\}$; $tp_2 = \{$***clock*** $c = 30\}$;
$tp_3 = \{$***action*** $=$ *output LoanServiceResultMessage*$\}$.

8.3. Timed Test Cases

The two timed test cases are generated (from the IF specification of the Loan service and the timed test purposes for each scenario) by using TestGen-IF. They are filtered according to observable actions (input, output and delays).

The filtered timed test cases are given in Figure 5. For instance, the first action of the two test cases indicates that the IF process of the Loan service receives the "*loan_request*" message from the *partnerLinkType "LS"* with *portType = "LS"* and *operation = "initiate"*. These parameters will be used, in future work, to automate the derivation of executable test cases and for their deployment.

TEST CASE 1 FOR SCENARIO #1

1. *?LoanServiceRequestMessage* (*"LS_LS_initiate", loan_request*)
2. *!CreditRatingServiceRequestMessage* (*"CRS_CRS_initiate", loan_request*)
3. ***delays** = 5*
4. *?CreditRatingServiceResultMessage* (*"CRS_CRSCallback_onResult",loan_response*)
5. *!LoanServiceResultMessage* (*"LS_LSCallback_onResult", loan_response*)

TEST CASE 2 FOR SCENARIO #2

1. *?LoanServiceRequestMessage* (*"LS_LS_initiate", loan_request*)
2. *!CreditRatingServiceRequestMessage* (*"CRS_CRS_initiate", loan_request*)
3. ***delays** = 30*
4. *!LoanServiceResultMessage* (*"LS_LSCallback_onResult", default_loan*)

Figure 5. The Abstract Test Cases

9. Conclusions and Future Work

This paper proposed a unit testing method for Web services composition based on *black box* approach and conformance testing. The BPEL description is considered as the specification of a Web service composition. In order to generate tests from the BPEL specification, we proposed to transform it into an Intermediate Format (IF). IF can model the BPEL constructs and can handle fault, event and termination. We proposed a test case generation algorithm with timed test purposes to guide the test generation from IF specification. The tests are produced according to a timed-traces inclusion relation (conformance relation) between the specification and the implementation of a composite Web service. The proposed methods have been implemented in two tools; the first performs the transformation of BPEL into IF model, while the second generates automatically the timed test cases. Our future work is to define test execution architecture for the generated test cases. We will also investigate the extension of the proposed test generation method to handle integration testing as will as the choreography of Web services.

References

[1] Active Endpoints. ActiveBPEL Designer. http://www.activevos.com/products.php.

[2] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis. The IF toolset. In *SFM-04*, volume 3185 of *LNCS*, pages 237–267. Springer-Verlag, june 2004.

[3] M. Bozga and Y. Lakhnech. IF-2.0 Common Language Operational Semantics, September 2002. www-if.imag.fr/tutorials/semantics.ps.gz.

[4] A. Bucchiarone, H. Melgratti, and F. Severoni. Testing Service Composition. In *Proceedings of ASSE'07*, Mar del Plata, Argentina, August 2007.

[5] A. Cavalli, D. Lee, C. Rinderknecht, and F. Zaidi. Hit-or-Jump: An Algorithm for Embedded Testing with Applications to IN Services. In *Proceeding of FORTE/PSTV'99*, Beijing, China, 1999.

[6] Cross Checknet Networks. SOA Testing Tools. http://www.crosschecknet.com/soa_testing_black_white_gray_box.php.

[7] E. Christensen et al. Web Services Description Language Ver. 1.1, March 2001. http://www.w3.org/TR/wsdl.

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

[9] H. Fouchal, E. Petitjean, and S. Salva. Testing timed systems with timed purposes. *RTCSA*, 00:166, 2000.

[10] J. Garca-Fanjul, J. Tuya, and C. de la Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In *WS-MaTe'06*, pages 83–94, Palermo, Italy, 2006.

[11] S. Hanna and M. Munro. An approach for specification-based test case generation for web services. *AICCSA'07*, pages 16–23, May 2007.

[12] M. Karam, H. Safa, and H. Artail. An Abstract Workflow-Based Framework for Testing Composed Web Services. *AICCSA'07*, pages 901–908, May 2007.

[13] M. Krichen and S. Tripakis. An Expressive and Implementable Formal Framework for Testing Real-Time Systems. In *TestCom*, pages 209–225, 2005.

[14] M. Lallali, F. Zaidi, and A. Cavalli. Timed Modeling of Web Services Composition for Automatic Testing. In *Proceedings of SITIS'07*, Shangai, China, December 2007.

[15] Z. Li, W. Sun, Z. B. Jiang, and X. Zhang. BPEL4WS Unit Testing: Framework and Implementation. *ICWS*, 0:103–110, 2005.

[16] M. Bozga and J.-C. Fernandez and L. Ghirvu and S. Graf and J.P. Krimm and L. Mounier and J. Sifakis. IF: An Intermediate Representation for SDL and its applications. In *SDL Forum*, pages 423–440, 1999.

[17] M. Gudgin et al. Simple Object Access Protocol (SOAP) Ver. 1.1, June 2003. http://www.w3.org/TR/soap12/.

[18] P. Mayer. *Design and Implementation of a Framework for Testing BPEL Compositions*. PhD thesis, Leibniz University, Hannover, September 2006.

[19] P. Mayer. *Design and Implementation of a Framework for Testing BPEL Compositions*. PhD thesis, Leibniz University, Hanover, Germany, September 2006.

[20] OASIS Standard. WSBPEL Ver. 2.0, April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[21] ORACLE. Oracle BPEL Test Framework. http://download.oracle.com/docs/cd/B31017_01/integrate.1013/b28981/testsuite.htm.

[22] PushToTest. TestMaker. http://www.pushtotest.com/.

[23] S. Troschtz. *Web Service Test Framework with TTCN-3*. PhD thesis, University of Gttingen, Germany, 2007.

[24] Verimag/IMAG. IF Toolset. www-if.imag.fr/.

[25] E. Vieira. *Automated Model-Based Test Generation for Timed Systems*. PhD thesis, Telecom SudParis, Evry, 2007.

[26] Y. Zheng, J. Zhou, and P. Krause. An Automatic Test Case Generation Framework for Web Services. *Journal of Software*, 2(3):64–77, September 2007.