A preliminary study on BPEL process testability.*

Sébastien Salva
LIMOS CNRS UMR 6158
Université d'Auvergne
Campus des Cézeaux,Aubière, FRANCE
sebastien.salva@u-clermont1.fr

Issam Rabhi
LIMOS CNRS UMR 6158
Université Blaise Pascal
Campus des Cézeaux,Aubière, FRANCE
rissam@isima.fr

ABSTRACT

WS-BPEL is an OASIS standard language used for describing interactions in Service Oriented Architectures (SOA). BPEL processes are usually overlapped in large business applications composed of web services. Such applications are more and more developed with respect of quality processes, such as testability, which is the topic of this paper. Testability helps to model systems where bug detection is relatively easier and whose the testing cost is lower. In this paper, we focus on two well-known testability criteria, observability and controllability. To evaluate them, we propose to transform ABPEL specifications into STS and to apply existing methods. Then, from STS testability issues, we deduce some patterns of ABPEL testability degradation. These latter help to finally propose testability enhancement methods.

Keywords

BPEL, testability, observability, controllability, enhancement methods

1. INTRODUCTION

WS-BPEL, or Web Services Business Process Execution Language [6], is an OASIS standard language used for describing interactions in Service Oriented Architectures (SOA). In such architectures, services are proposed through components named *web services* which can be seen as independent object instances called by operations (methods). Such web services can be requested over a network like Internet and take place recently in the "Cloud" paradigm (virtualized resources where control or expertise is unneeded). BPEL implements Business processes over web services by orchestrating them by mean of a coordination logic: BPEL processes receive specific client application requests, call some web services according to conditions and variables defined in the process, and usually return a response.

BPEL processes are often written in large business applications gathering also many web services, composed themselves of many data management layers. For instance, such applications may be used to manage transaction of goods between customers and suppliers, or to make live statistics. Nowadays, more and more companies developing web services and BPEL processes follow several quality processes and models, like the CMMI one (Capability Maturity Model Integration). Software testability, which is one quality criterion, aims to achieve a testable application, that is an application from which the testing stage is able to detect the largest number of bugs, with the less efforts with regards to bug detection and time. Since, the testing process costs as much as 50 % of the total development effort, testability is seen as essential.

BPEL testability is a large field because this language provides a large set of mechanisms (fault handling, correlation,...), interacts with external web services and proposes parallel programming. So, we propose, in this paper, a preliminary testability study of sequential BPEL processes, modeled with the APBEL (Abstract BPEL) language. We focus on two well-known testability criteria, observability and controllability. To evaluate them, we transform an ABPEL specification into a corresponding STS (Symbolic Transition System) to flatten the nested BPEL activities, to spread fault handlers into sub-activities, and to retrieve irrelevant properties. We obtain a graph which can be analyzed with existing testability algorithms.

Then, from some known STS testability issues, we deduce the corresponding patterns of ABPEL testability degradation. These ones may help designers to write directly more testable ABPEL specifications. These ABPEL testability degradations are also used, in the following, to construct some enhancement methods which detect testability issues and modify semi-automatically ABPEL specifications. We apply them on a real BPEL example, the "Loan approval" which is given in the WS-BPEL specification [6].

The remainder of this paper is structured as follow: Section 2 provides an overview on the BPEL language and on system testability, while describing some related works. In section 3, we discuss on the testability evaluation, and introduce some ABPEL testability degradation propositions. Then, we describe, in section 4, some semi-automatic testability enhancement methods and an academic tool which implements them. Section 5 leads to several perspectives on BPEL testability by giving some ideas about other proper-

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

QuoMBat 2010, Paris, France

ties to analyze.

2. BPEL AND TESTABILITY OVERVIEWS

2.1 BPEL

BPEL defines and manages the process orchestrations, that is the interactions between web services, called *partners*. BPEL describes abstract and executable business processes. An abstract process describes the element declarations (exchanged messages, partner roles, links, ...) which are going to be used by this process. An executable BPEL process models the orchestration and includes the internal details (states, variables, requests,...). For this, BPEL gathers basic activities, to perform basic operations (invoke, receive,...), and structured activities (scope, flow,...) to set the activity structuring. Among its possibilities, it supports fault handler activities that manage fault messages or other exceptions returned by external web services (the process can be compensated or terminated,...). BPEL processes also handle web service WSDL descriptions which gather the required informations to invoke a service, by defining the web service interfaces, called enpoints, the accessible methods, called operations, and the operation parameter/response types.

We illustrate, in figure 1, a graphical example of BPEL process, the *loan approval sample*, which is derived of the one which can be found in the WS-BPEL specification. A part of the BPEL code, describing the two first activities is given in figure 2. In this process, a customer sends a request for a loan, the request gets processed, and the customer finds out whether the loan was approved. The partners involved are: web service "loan assessor", web service "loan approval1" and web service "loan approval2".

In the following, we consider gray box BPEL processes were the external interactions with the partners are observable. With such gray boxes, two kinds of test architectures are proposed. These ones differ by the number of PCO (point of control and observation). In the first one (figure 3 with continuous boxes), there is only one PCO, thus the web service interactions are uncontrollable. We can only stimulate the BPEL process from the client application. With the second architecture (figure 3 with dashed boxes), more PCO are added to control the messages sent from the partners, which need to be simulated. Each message sent to the BPEL process is then controllable. Some testing methods use this architecture and some tools help to implement simulated partners by generating automatically partner stubs [3].

Many works have been proposed on BPEL and some of them tackle especially to its validation [14, 23, 16, 10, 25, 15]. The authors of [14] propose a BPEL unit testing approach and introduce a tool prototype, named BPEL-Unit, which extends JUnit. The process, modeling web service interactions, is translated to JAVA collaboration classes and method calls, which are later tested by Junit. In [23], a method is proposed to model web service compositions with CSP (process algebra) by using a set of rules for translating compositions. Afterwards, other methods are introduced for model checking, model verification and model simulation by using FDR tool(Failures-Divergence Refinement). In [16], the authors use BPEL specifications, translated into Petri nets (Using BPEL2oWFN), and test the existence of a partner process, such that both can interact properly (interoperability test). In [10], the authors transform BPEL specifications to PROMELA code, and then generate test cases by using the SPIN model-checking tool. In [25], another test case generation framework for BPEL compositions is proposed. BPEL specifications are modeled by WSA (Web Service Automata). SPIN and NuSMV tools are used to generate test cases. In [15], the BPEL process is modeled by BFG (BPEL Flow Graph). Test cases are generated by covering each graph branch at least once. These ones are executed and traces are analyzed to give a verdict. A test selection minimization algorithm is also proposed.

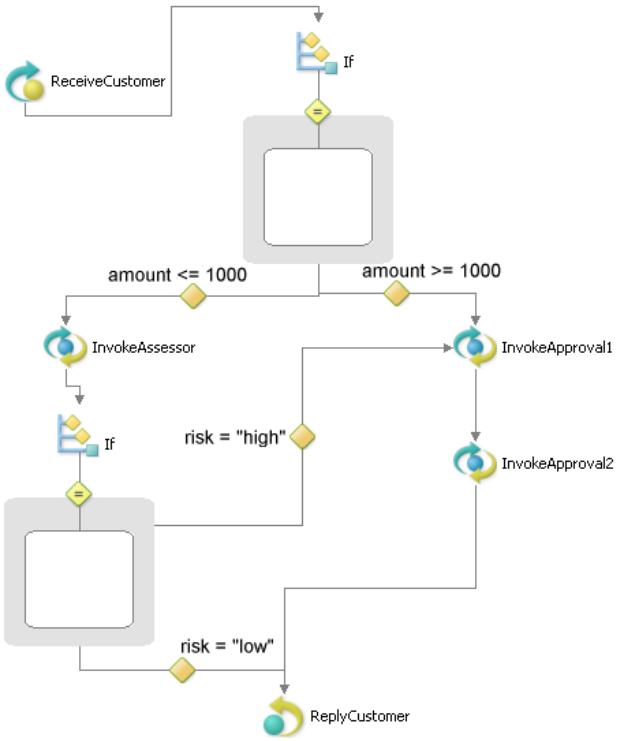

Figure 1: The Loan approval BPEL process

2.2 Testability

Testability gathers several criteria which evaluate the system capacity to reveal its faults, the accessibility of its components and its testing cost [9]. Testability can be blend into software life cycle as illustrated in figure 4. Designers evaluate testability of a system after each life cycle step. So, they can evaluate and anticipate the system parts which are testable and those which are not. They can also measure the testing cost.

Testability can be used to model and to implement testable systems, by improving the fault detection and the fault coverage [11]. It can also be evaluated to choose one specification (the most testable) among others.

Testability has been studied on different untimed models (automata, UML models, logical circuits, relational models [9, 11, 24, 18]) and timed ones (timed automata)[20]. These methods evaluate testability criteria one by one by returning, for instance, a factor called *degree* (a number between 0 and 1), but also the number of potential faults which can be detected, or the number of testability issues.

Many criteria have been studied, depending on the sys-

```
<bpel:receive createInstance="yes" operation="request"
            partnerLink="customer" portType="lns:loanServicePT"
            variable="request">
            <bpel:sources>
                <bpel:source linkName="receive-to-assess">
                    <bpel:transitionCondition>
                        ($request.amount &lt; 10000)
                    </bpel:transitionCondition>
                </bpel:source>
                <bpel:source linkName="receive-to-approval">
                    <bpel:transitionCondition>
                        ($request.amount &gt;= 10000)
                    </bpel:transitionCondition>
                </bpel:source>
            </bpel:sources>
        </bpel:receive>
        <bpel:invoke inputVariable="request" operation="check"
            outputVariable="risk" partnerLink="assessor"
            portType="lns:riskAssessmentPT">
            <bpel:targets>
                <bpel:target linkName="receive-to-assess" />
            </bpel:targets>
            <bpel:sources>
                <bpel:source linkName="assess-to-setMessage">
                    <bpel:transitionCondition>
                        ($risk.level = 'low')
                    </bpel:transitionCondition>
                </bpel:source>
                <bpel:source linkName="assess-to-approval">
                    <bpel:transitionCondition>
                        ($risk.level != 'low')
                    </bpel:transitionCondition>
                </bpel:source>
            </bpel:sources>
        </bpel:invoke>
```

Figure 2: A part of the BPEL code

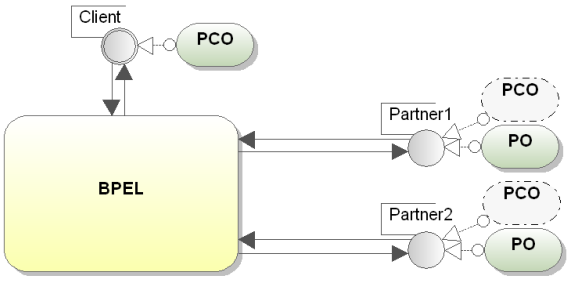

Figure 3: The test architectures

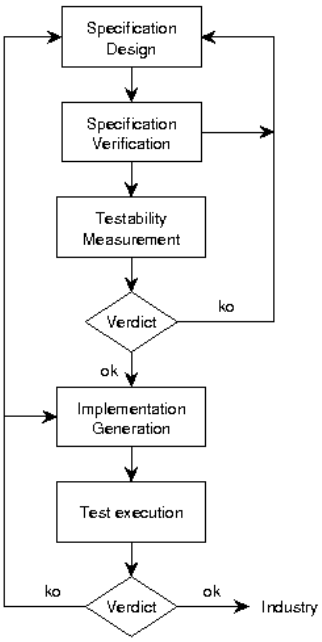

Figure 4: Testability in software life cycle

tem, but observability and controllability are commonly targeted:

- *Observability* aims to evaluate the system internal state according to its observed outputs. For input output systems, observability is defined in [9] by: "a system is observable if for each input given to the system, a different output is observed",
- *Controllability* denotes the ability to reach and to activate specific parts of a system. It is defined in [9] by "a system is controllable if for each observed output, it exists an input which forces the observation of this output".

3. BPEL TESTABILITY STUDY

The first raised question is which formalism to choose for BPEL process modeling? We found some different responses in literature [22]. BPEL processes can be modeled with:

- high level languages, such as BPMN, ABPEL (Abstract BPEL) or UML activity diagrams which describe the BPEL process overall behaviors by giving for instance the partner roles, the exchanged messages (requests and responses), and the conditions on these messages,
- lower level languages, such as Petri nets, automata (LTS,...) , process algebra, abstract state machines (ASM). These ones describe BPEL processes with more details (about variables, conditions, fault handling,...).

We base our choice on the ABPEL formalism because it is widely used in software development companies and with existing testing methods [2].

In the following, we first describe how to evaluate testability of ABPEL specifications. We focus on the ABPEL

observability and controllability degradations. From these latter, we propose several testability enhancement methods.

3.1 How can ABPEL specifications be assessed ?

Analyzing ABPEL specifications directly to assess testability criteria is not an easy task. They are composed of many information about partners, variables,... (see figure 2). But, above all, structured BPEL activities such as scope, process, or fault handler activities may be nested with other ones. For instance, the scope activity of figure 5 is composed of a fault handler which is dedicated to each sub-activity of the scope. Each sub-activity may be also composed of other sub-activities and fault handlers. And so on.

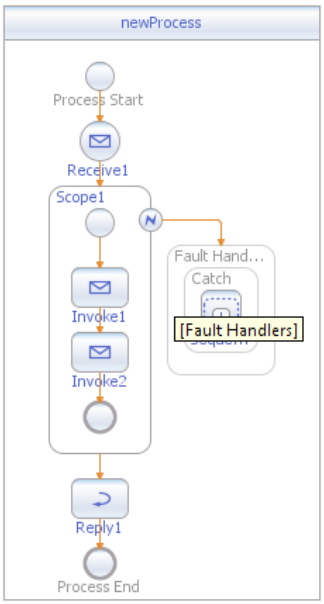

Figure 5: An ABPEL process

As a consequence, some works propose to transform ABPEL specifications into other formalisms [19, 5, 13, 2]. We propose, in this paper, to transform ABPEL specifications into STS (Symbolic Transition Systems [8]). STS offer a large formal background (definitions of implementation relations, test case generation algorithms,...) and are semantically close to UML state machines. Besides, some tools are proposed to translate UML state machines into STS [1]. Last but not least, a BPELtoSTS transformation has been proposed in [17, 2]. The latter aims to flatten each nested BPEL activity, to spread the fault handlers of one activity into each of its sub-activities, and to eliminate the irrelevant activities for testability analysis (for instance, the "assign", "empty" activities). Moreover, using STS offers the significant advantage to apply existing state machine testability analysis algorithms or tools.

A *Symbolic Transition System* STS is a tuple $< L, l_0, Var, var_0, I, S, \rightarrow >$, composed of symbols $S = S_I \cup S_O$: inputs, beginning with "?" are provided to the system, while outputs, (beginning with "!") are observed from it. An STS is composed of a location variable set Var, initialized by var_0 and of an interaction variable set I. Each transition $(l_i, l_j, s, \varphi, \varrho) \in \rightarrow$ from the location l_i to l_j, labeled by the symbol s, may update variables with ϱ and may have a guard φ on $Var \cup I$, which must be satisfied to fire the transition.

[8] defined the notion of stimulus (resp. reaction) which is a pair (s, η), where $s \in S_I$ is an input symbol (resp. $s \in S_O$ is an output one) and η is a mapping of the interaction variables of s to ground terms. For a reaction $(!o, \eta)$, η is observable in the message $!o$. A reaction is observed from BPEL processes either when a fault is thrown (with the "throw" activity) or with the "reply" and "invoke" activities when a partner operation is called with parameter values (η).

We also denote $out(l, \eta)$ the set of the first reactions reached from the location l with the values η. $out(l_1, \eta) = \{(!o_1, \eta'_1), ..., (!o_n, \eta'_n) \mid \forall 1 \leq i \leq n, \exists p = (l_1, l_2, e_1, \varphi_1, \varrho_1)...(l_i, l_{i+1}, !o_i, \varphi_i, \varrho_i)$ such as $(e_1, ..., e_{i-1}) \in S_I^{i-1}$ and $\wedge(\varphi_1(\eta), ..., \varphi_i(\varrho_{i-1}, \eta'_1))$ true$\}$.

We applied the BPELtoSTS transformation on the specification example, depicted in figure 1. We obtain the STS of figure 6. All the structured activities (scopes, if, fault handlers,...) were flatten. And all the fault handlers were spread to sub-activities. For instance, A "fault handler" activity begins at the location A11. Each outgoing transition models a "catch" activity.

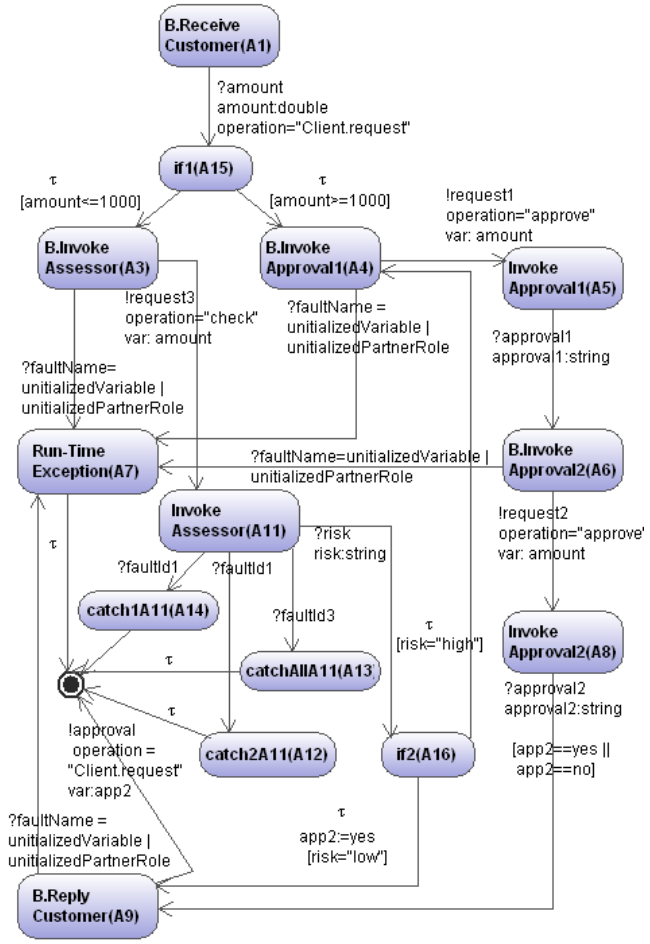

Figure 6: An STS modeling an ABPEL process

Once translated into STS, a BPEL process can be analyzed with relative ease to measure many state machine

based testability criteria, such as the observability, the controllability [11] or coverage criteria [4]. In the remainder of this paper, we will consider observability and controllability criteria only.

The testability measurement can be performed in different ways, such as counting the number of testability issues. For instance, consider the STS of figure 6. This one contains five observability and six controllability issues:

- Observability issues: the stimuli ($?amount$, $amount = 1000$) and ($?risk$, $risk = high, amount = 1000$) give the same reaction ($!request1$, $amount = 1000$). 2) In the same way ($?risk$, $risk = low$) and ($?approval2$, $app2 = yes$) are followed by the same reaction ($!approval$,$app2 = yes$). 3) The last observability issues are detected at the location A11 (Invoke Assessor), where several input messages modeling "catch" activities are not followed by output ones,

- Controllability issues: these ones are often the consequence of indeterminism states [11]. The STS of figure 6 has four indeterminism locations (locations A3 (invoke assessor), A4 (B.Invoke approval1), A6 (B. Invoke Approval2), A9(B. Reply Customer)) on account of partner roles not initialized in the APBEL specification. Another controllability issue is detected from the location A15 since the two conditions [$amount \leq 1000$] and [$amount \geq 1000$] are not exclusive. The last case is obtained from the location A11 where the same fault ?FaultId1 is labelled on two different transitions.

Such testability evaluations may help to choose the most testable specification among many. However, it does not identify testability issues on the ABPEL specification directly. These evaluations detect only issues in the corresponding STS specification. So, these ones do not help designers neither to write ABPEL processes with higher quality nor to improve them once testability issues are detected.

This is why we analyze, in the following, some properties degrading the STS testability from which we deduce the corresponding ABPEL testability degradation patterns.

In the following, we assume that the partners, taking part to the BPEL orchestration, are testable (observable and controllable). The BPELtoSTS transformation rules, that we consider in the propositions, are those given in [17, 2]. Due to lack of room, some of them are given in [21].

3.2 ABPEL observability degradation

Observability aims to evaluate the system internal state from its observed outputs. According to the observability definition, given in section 2.1, the system observability is degraded if some inputs are followed by the same outputs or by no output. For an STS $sts =< L, l_0, Var, var_0, I, S, \rightarrow >$, this can be written with two degradation properties which take into account stimuli and reactions. The first property means that the observability is degraded if no reaction is observed. The second one describes the case of different stimuli which produce the same reaction set.

- Observability degradation 1: if it exists a stimulus $(?e, \eta)$ such as $(k, l, ?e, \varphi, \varrho) \in \rightarrow$ and $out(l, \eta) = \emptyset$,

- Observability degradation 2: if it exists two stimuli $(?e_i, \eta_i) \neq (?e_j, \eta_j)$ with $(l_i, l'_i, ?e_i, \varphi_i, \varrho_i) \in \rightarrow$, $\varphi_i(\eta_i)$ true and $(l_j, l'_j, ?e_j, \varphi_j, \varrho_j) \in \rightarrow$, $\varphi_j(\eta_j)$ true such as $out(l'_i, \eta_i) = out(l'_j, \eta_j)$.

From the first STS rule, we have deduced two corresponding ABPEL testability degradations: "the lack of a reply activity at the end of ABPEL processes" and "the use of conditions, in "if" activities, which cannot be always satisfied".

Proposition 3.1 *An ABPEL specification not terminated by a "reply" (one-way "invoke") activity is not observable.*

Proof. "reply" and one-way "invoke" activities are both identical.
1. Consider an ABPEL process *bpel*, composed only of internal actions (for instance the "assign" or "empty" activities) and not terminated by a reply activity. An ABPEL process always begins by a "receive" activity (see [6]). The translation of *bpel* into STS produces one transition $(l, l', ?e, \varphi, \varrho)$ labeled by an input symbol ("receive" activity" followed by no transition labeled by an output one). So, $\forall \eta, out(l', \eta) = \emptyset$. *bpel* is not observable.
2. Now, consider an ABPEL process *bpel*, composed of partner interactions (the usual case) and not terminated by a reply activity. We have supposed that the associated partners are observable. Consequently, they always produce a response once called. An "invoke" activity is translated by two transitions in STS, the first one labeled by an output symbol and the other one labeled by an input symbol. The last "invoke" activity produces the STS transitions $(l_i, l_{i+1}, !o, \varphi, \varrho)$, $(l_{i+1}, l_{i+2}, ?e, \varphi_{i+1}, \varrho_{i+1})$. Without "reply" activity, this transition cannot be followed by another one labeled by an output symbol. Consequently, $\forall \eta, out(l_{i+2}, \eta) = \emptyset$. *bpel* is not observable. □

Proposition 3.2 *An ABPEL specification, composed of an "invoke" activity followed by an "if" one* $if((cond_1, act_1),\ldots, (cond_n, act_n))$ *which cannot be always satisfied* ($\exists \eta$, $\bigvee_{1 \leq i \leq n} cond_i(\eta)$ *false), is not observable.*

Proof. An invoke activity is transformed by the STS transitions $(l, l', !o, \varphi, \varrho)$, $(l', l'', ?e, \varphi', \varrho')$. The "if" activity yields the STS transitions $\forall (1 \leq i \leq n)$ $(l'', l_i, \tau, \varphi_i, \varrho_i)$ with $\varphi_i = cond_i$. Suppose that it exists η such as $\bigvee_{1 \leq i \leq n} (\varphi_i(\eta))$ false. None transition can be fired. The location l'' is blocked and is said quiescent [8]. So, it exists a stimulus $(?e, \eta)$ such as $out(l'', \eta) = \emptyset$. Consequently, the STS and the ABPEL specification are not observable. □

With the second STS degradation property, we found four corresponding cases of ABPEL testability degradation. With STS, this degradation is obtained if it exists two transitions $(l_i, l_{i+1}, ?e_i, \varphi_i, \varrho_i)$, $(l_j, l_{j+1}, ?e_j, \varphi_j, \varrho_j)$ from which the same reaction $(!o, \eta)$ is observed. This case can be derived from ABPEL specifications having: "two different catch activities followed by the same invocation", "a catchall activity, triggered by multiple faults, followed by the same invocation", "a pick activity with multiple onmessage branches followed by the same invocation", or having "two (or more) successive receive activities".

Proposition 3.3 *An ABPEL specification composed of a couple of non identical "catch" ("catchall") activities* ($catch_i$,

$catch_j$), $catch_i \neq catch_j$, followed by two "invoke" activities using the same operation and parameter values, is not observable.

PROOF. In the WS-BPEL specification [6], two "catch" activities are said identical if their received faults are identical too. Each "catch" activity is triggered by a received fault $?f = (faultName, faultElement, faultMessage\ Type)$. Two different "catch" activities ($catch_i$, $catch_j$) are translated by two STS transitions $(l_i, l_{i+1}, ?fault_i, \varphi_i, \varrho_i)$ and $(l_j, l_{j+1}, ?fault_j, \varphi_j, \varrho_j)$, with $?fault_i \neq ?fault_j$. If $catch_i$ and $catch_j$ are followed by the same operation call, using the same parameter value η, the STS is composed of two transitions $(l_{i+1}, l_{i+2}, !o, \varphi_{i+1}, \varrho_{i+1})$, $(l_{j+1}, l_{j+2}, !o, \varphi_{j+1}, \varrho_{j+1})$ such as $\vee(\varphi_{i+1}(\eta), \varphi_{j+1}(\eta))$ true. Consequently, it exists two stimuli $(?fault_i, \eta_i)$, $(?fault_j, \eta_j)$ such as $out(l_{i+1}, \eta) = out(l_{j+1}, \eta) = (!o, \eta)$. The STS and the ABPEL specification are not observable. □

Proposition 3.4 *An ABPEL specification composed of a "catchall" activity, triggered by multiple faults and followed by an "invoke" activity whose the operation call is independent of the triggered fault, is not observable.*

PROOF. A "catchall" activity $catchall((?fault_i, ?fault_j), act)$ is a "catch" one, triggered by all the faults received in a fault handler, not already caught by other "catch" activities. Let $?fault_i = (faultName = n_i, faultElement = e_i, faultMessage = m_i\ Type = t_i)$ and $?fault_j = (fault - Name = n_j, faultElement = e_j, faultMessage = m_j\ Type = t_j)$ be two different faults. This one is translated into the STS transitions $(l_i, l_{i+1}, ?fault_i, \varphi_i, \varrho_i)$ and $(l_i, l_{i+1}, ?fault_j, \varphi_j, \varrho_j)$, with $?fault_i \neq ?fault_j$. If this "catchall" activity is followed by an "invoke" one, we also have the transition $(l_{i+1}, l_{i+2}, !o, \emptyset, \varrho_{i+1})$ with $!o = (op, req, partner)$.

Let η be a value obtained before the "catchall" activity and $(?fault_i, \eta_i)$, $(?fault_j, \eta_j)$ be two stimuli. η_i (respectively η_j) is obtained from η by applying the variable update ϱ_i (ϱ_j) ($\varrho_i : faultName_i = n_i, faultElement_i = e_i, faultMessage_i = m_i\ Type_i = t_i$). If op does not handle the fault, the variables req are independent of the ones in $\varrho_i \wedge \varrho_j$. The mapping of req to ground terms equals to η. Consequently, it exists two stimuli $(?fault_i, \eta_i)$, $(?fault_j, \eta_j)$ such as $out(l_{i+1}, \eta_i) = out(l_{j+1}, \eta_j) = (!o, \eta)$. This corresponds to the second degradation property. The STS and the ABPEL specification are not observable. □

Proposition 3.5 *An ABPEL specification composed of a "pick" activity with multiple "onmessage" branches followed by "invoke" activities using the same operation and parameter values, is not observable.*

PROOF. A "pick" activity with two "onmessage" branches is translated by the STS transitions $(l, l_i, ?e_i, \varphi_i, \varrho_i)$, $(l, l_j, ?e_j, \varphi_j, \varrho_j)$, with $?e_i \neq ?e_j$ and $(l_i, l_{i+1}, !o, \varphi_{i+1}, \varrho_{i+1})$, $(l_j, l_{j+1}, !o, \varphi_{j+1}, \varrho_{j+1})$. We obtain the same case as the one described in proposition 3.3. Thus, the STS and the ABPEL specification are not observable. □

Proposition 3.6 *An ABPEL specification, composed of two (or more) successive "receive" activities, is not observable.*

PROOF. k successive "receive" activities are translated by k STS transitions $(l_1, l_2, ?e_1, \varphi_1, \varrho_1)$,..., $(l_k, l_{k+1}, ?e_k, \varphi_k, \varrho_k)$. If these "receive" activities are not followed by an "invoke" ("reply") one, $\forall(\eta_1, ..., \eta_k) out(l_2, \eta_1) = ... = out(l_{k+1}, \eta_k) = \emptyset$. Otherwise, these ones are followed by an "invoke" ("reply") activity which produces an STS transition $(l_n, l_{n+1}, !o_n, \varphi_n, \varrho_n)$ such as $\exists \eta_n, \varphi_n(\eta_n)$ true. $\forall(\eta_1, ..., \eta_k)$ such as $\bigwedge_{1 \leq i \leq k} (\varphi_i(\eta_i)\ \text{true}), out(l_2, \eta_1) = ... = out(l_{k+1}, \eta_k) = (!o_n, \eta_n)$. Consequently, The STS and the ABPEL specification are not observable. □

3.3 BPEL controllability degradation

Controllability depends firstly on the test architecture features. As we suggest in section 2.1, using the test architecture composed of real partners (architecture composed of one PCO and of two PO, figure 3 with continuous boxes) involves to an uncontrollable BPEL process. Indeed, messages, which are computed and returned by the partners to the BPEL process, cannot be controlled by a tester. The single PCO (point of control and observation) is on the Client side only. This lack of controllability can be solved by using a test architecture composed on PCO where partners are simulated. With this one, returned responses are input messages given by the tester, so these ones are controllable. In the following, we consider this king of architecture.

Controllability is also influenced by other properties. Determinism is one of them. The more indeterministic a system is, the less controllable it becomes [11].

An STS $sts =< L, l_0, Var, var_0, I, S, \rightarrow >$ is undeterministic if it exists a location $l \in L$ such as:

1. it exists two transitions $(l, l_i, e, \varphi_i, \varrho_i)$, $(l, l_j, e, \varphi_j, \varrho_j)$ labelled by the same symbol and η with $(\varphi_i(\eta) \wedge \varphi_j(\eta))$ true,
2. it exists two transitions $(l, l_i, !o_i, \varphi_i, \varrho_i)$, $(l, l_j, !o_j, \varphi_j, \varrho_j)$ with $!o_i \neq !o_j$ and η with $(\varphi_i(\eta) \wedge \varphi_j(\eta))$ true,
3. it exists two transitions $(l, l_i, ?e, \varphi_i, \varrho_i)$, $(l, l_j, !o, \varphi_j, \varrho_j)$ and η with $(\varphi_i(\eta) \wedge \varphi_j(\eta))$ true.

From these indeterministic properties, we found three corresponding testability degradation properties in ABPEL specifications having: "partner roles not initialized", "two successive if activities whose the conditions may be satisfied simultaneously" or having "faulthandler activities composed of identical catch ones".

Proposition 3.7 *"invoke" activities, depending on partners whose the role is not initialized, involve indeterministic and uncontrollable ABPEL processes.*

PROOF. According to the WS-BPEL specification [6], "invoke" activities, whose the partner role is not initialized, must be composed of a "catch" activity triggered by the fault "uninitializedPartnerRole". This activity is translated to the two STS transitions $(l, l_i, !o, \varphi_i, \varrho_i)$, $(l, l_j, ?fault, \varphi_j, \varrho_j)$ with $\varphi_i = \varphi_j = \emptyset$. So, $\forall \eta, (\varphi_i(\eta) \wedge \varphi_j(\eta))$ true. This corresponds to the third indeterminism case. Thus, the corresponding STS and the ABPEL specification are not deterministic and not controllable. □

Proposition 3.8 *An ABPEL specification, composed of an "if" ("switch") activity gathering two conditional branches $if((cond_1, act_1), (cond_2, act_2))$ where the condition $\vee(cond_1, cond_2)$ may be satisfied simultaneously ($\exists \eta \mid (cond_1(\eta) \wedge cond_2(\eta))true$), is not controllable.*

PROOF. The "if" (or "switch") activity $if((cond_1, act_1), (cond_2, act_2))$ is translated by the STS transitions $(l, l_1, \tau, \varphi_1, \emptyset)$, $(l, l_2, \tau, \varphi_2, \emptyset)$ with $\varphi_1 = cond_1$ ($\varphi_2 = cond_2$). Suppose that it exists η such as $(\varphi_1(\eta) \wedge \varphi_2(\eta))$ true. These transitions represent the first indeterminism case given previously. Consequently, the ABPEL process is not deterministic and not controllable. □

Another indeterminism case is raised on account of the "flexibility", allowed in the "catch" activity construction by the WS-BPEL specification [6]. Indeed, it is granted to construct several identical "catch" activities composed by the same fault identification (faultName, faultElement, faultMessageType).

Proposition 3.9 *An ABPEL process, composed of a "faulthandler" activity gathering two identical "catch" activities, is not controllable.*

PROOF. Let $catch_k((faultName_k, faultElement_k, faultMessage_k), act_k)$ and $catch_l((faultName_l, faultElement_l, faultMessage_l), act_l)$ be two "catch" activities of the same fault handler. These "catch" activities are translated by two STS transitions $(l, l_i, ?fault_i, \varphi_i, \emptyset)$, $(l, l_j, ?fault_j, \varphi_j, \emptyset)$, with $\varphi_i = \varphi_j = \emptyset$. If $(faultName_i, faultElement_i, faultMessage_i)$, $(faultName_j, faultElement_j, faultMessage_j)$ are identical, in accordance with the WS-BPEL specification (identical values, two absent values are identical), then $?fault_i = ?fault_j$. So, it exists η with $(\varphi_i(\eta) \wedge \varphi_j(\eta))$
true such as two transition can be fired. This corresponds to the first indeterminism case, given previously. The ABPEL process is undeterministic and uncontrollable. □

4. BPEL TESTABILITY ENHANCEMENT

The previous propositions, describing ABPEL testability degradation patterns, are used here to propose some testability enhancement methods and a corresponding tool. This one parses an ABPEL specification, detects testability degradations, according to the propositions 3.1-3.9 and removes them. The tool architecture is given in figure 7.

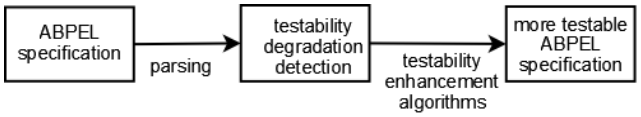

Figure 7: A testability enhancement tool

The modification of ABPEL specifications by these enhancement methods may require the update of some partner WSDL descriptions or code parts. This is why we denote them as semi-automatic. These enhancement methods are detailed below. In the following, we denote *bpel*, an ABPEL specification.

4.1 BPEL observability enhancement

We improve the ABPEL specification observability by focusing on the propositions 3.1 and 3.3. The first enhancement method adds a "reply" activity when this one is missing. The second one distinguishes the observable behaviour of "catch" activities.

- **"reply" activity addiction:** We check that each branch of the ABPEL specification ends with a "reply" (invoke-only) activity. If one is missing, we complete the specification with a "reply" activity modeling a response to the client which has called the ABPEL process. The corresponding algorithm is given in Algorithm 1. The response sent to the client is composed of the message "final message from $location_i$" which is supposed to be a unique output message, not yet used in the specification. (the observability is not degraded with the addiction of this output message),

- **"catch" activity distinction:** if two "catch" activities $catch_i(?fault_i, act_i)$, $catch_j(?fault_j, act_j)$ are followed by two "invoke" activities using the same operation *op*, we compute, from the corresponding STS, the values satisfying the execution of these invocation, with constraint solvers [12, 7]. If a value satisfies both the two "invoke" activities (proposition 3.3 verified), we modify them by adding a new parameter to *op*, equals to $?fault_i$ ($?fault_j$ respectively). According to the proposition 3.3, since $?fault_i$ and $?fault_j$ are not identical we obtain two different operation calls and thus different reactions. However, this modification also requires to update the
WSDL description and the code of the called partner. The algorithm is given in Algorithm 2. The constraint solvers construct values satisfying the guards of a specification path and hence satisfying its execution. We use the solvers [7] and [12] which works as external servers. The solver [12] manages "String" types, and the solver [7] manages most of the other simple types.

input : ABPEL specification *bpel*
Compute $sts =< L, l_0, Var, var_0, I, S, \rightarrow >$ from *bpel*
if $\exists l_i \xrightarrow{e,\varphi,\varrho} l_f$ *with* $e \in S_I \cup \{\tau\}$ *and* l_f *a final location*
then
  Add a reply activity $reply(param, partner, op)$ in *bpel* with partner=client, op= client operation used for calling the ABPEL process, param="last message from $branch_i$"
end

Algorithm 1: "reply" activity addiction

4.2 BPEL controllability improvement

We improve the controllability of ABPEL specifications by considering the propositions 3.7 and 3.9. First, we add, when needed, the missing partner roles. This solution removes the faults whose the type is "uninitializedPartnerRole" and so an indeterminism case, described in proposition 3.7. This upgrade requires some data from the partner WSDL descriptions. Then, referring to proposition 3.9, we try to distinguish each fault in the same "faulthandler" activity. In the same way, this solves an indeterminism issue. This modification requires some partner modifications.

- **partner role addiction:** For each "invoke" activity $invoke(mess, resp, partner, op)$, we check that the partner role is described in the BPEL section "PartnerLink". If not, we update it, in condition that all the partner WSDL descriptions are provided. Once the

input : ABPEL specification *bpel*
if *it exists two "catch" activities* $catch_i(?fault_i, act_i)$, $catch_j(?fault_j, act_j)$ *in bpel followed by the operation call op with the parameters* $(p_1, ..., p_m)$ **then**
  Compute $sts =< L, l_0, Var, var_0, I, S, \rightarrow >$ from *bpel*
  foreach *paths* $p_i = l_0 \xrightarrow{e_0,\varphi_0,\varrho_0} l_1$,..., $l_i \xrightarrow{?fault_i,\varphi_i,\varrho_i} l_{i+1} \xrightarrow{!op,\emptyset,\varrho_{i+1}} l_{i+2}$ *and* $p_j = l_0 \xrightarrow{e'_0,\varphi'_0,\varrho'_0} l'_1$,..., $l'_j \xrightarrow{?fault_j,\varphi'_j,\varrho'_j} l'_{j+1} \xrightarrow{!op,\emptyset,\varrho'_{j+1}} l'_{j+2}$ **do**
    Compute with solvers V_1 the value set over $(p_1, ..., p_m)$ such as $\forall v \in V_1, \wedge(\varphi_0(v), \varphi_1(\varrho_0)$,..., $\varphi_i(\varrho_{i-1}), \varrho_{i+1} true)$
    Compute V_2 from the path p_j
    if $V_1 \cap V_2 \neq \emptyset$ **then**
      Add to the first invoke activity a string parameter "fault" with the value "$?fault_i$"
      Add to the second invoke activity a string parameter "fault" with the value "$?fault_j$"
    end
  end
end

Algorithm 2: "catch" activity distinction

partner role is declared, the indeterminism case described in proposition 3.5 is removed. The algorithm is given in Algorithm 3,

- **fault distinction in fault handlers:** We check that each "faulthandler" is not composed of two "catch" activities $catch_i(?fault, act_i)$, $catch_j(?fault, act_j)$ triggered by the same fault. Otherwise, we try to differentiate these faults either by modifying the message type or by naming them differently (if the type cannot be modified, we modify the fault name). With this mere modification, we distinguish the input faults and we remove an indeterminism case (proposition 3.9). However, this modification implies to update the WSDL description of the partner which sends this fault. The algorithm is given in Algorithm 4.

input : ABPEL specification *bpel*
foreach *"invoke" activity* $invoke(mess, resp, partner, op)$ **do**
  if *partner has not a role in the BPEL "partnerLink" section* **then**
    add <partnerLink name= *partner_name*
    partnerRole="*partner_name*Provider"
    partnerLinkType="ns:*partner_name*"/>
  end
  , with "ns" a new variable equals to the web service WSDL URL (xmlns:ns="http://...")
end

Algorithm 3: PartnerRole addiction

We applied these enhancement methods on the "Loan approval" example of figure 6 to produce a new specification. The corresponding STS is illustrated in figure 8. This new specification is much more testable since the testability degradation number is reduced to 4 instead of 11. Three different "reply" activities (locations A12, A13 and A14) and three

input : ABPEL specification *bpel*
foreach *"faulthandler" activity composed of the catch activities* $catch_1(fault_1, act_1)$,..., $catch_n(fault_n, act_n)$ **do**
  // $?fault_k = (faultName_k, faultElement_k, faultMessageType_k)$
  if *it exists* $catch_i(fault_i, act_i)$, $catch_j(fault_j, act_j)$ *with* $fault_i == fault_j$ **then**
    if $MessageType_i == null$ **then**
      $MessageType_i = type$ in (string, integer,...) such as $\forall (1 \leq k \neq i \leq n), faultMessageType_k \neq type$
    end
    else
      Add a random integer value at the end of $faultName_i$
    end
  end
end

Algorithm 4: Fault distinction

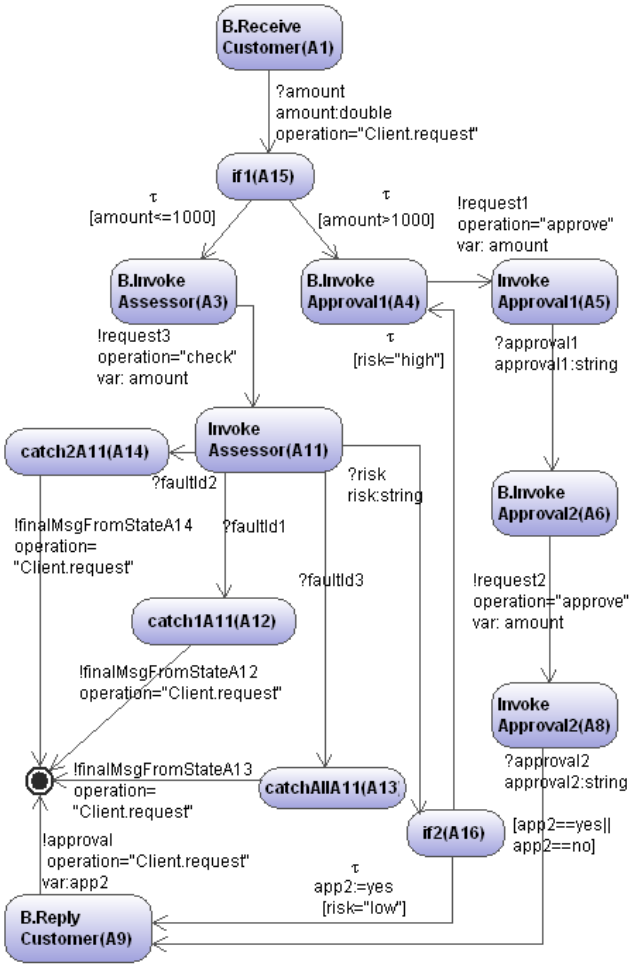

Figure 8: The modified STS

partner roles (locations A3, A6 and A9) have been added. Two identical "catch" activities (location A11) have been distinguished.

5. CONCLUSION

This paper proposes a preliminary study on ABPEL testability and some propositions describing patterns of ABPEL testability degradation. These ones can be used to directly write more testable ABPEL specifications or to evaluate observability and controllability criteria. We also propose some testability enhancement methods, which have been implemented in an academic tool. This one parses the specification, detects testability degradation cases and modifies it semi-automatically.

This preliminary work may lead to several perspectives on account of the large possibilities of the BPEL language. For instance, other criteria can be studied such as:

- the *execution time*, which evaluates the testing cost according to the minimal and maximal execution times. Evaluating it is one step. But it is also possible to reduce it, by translating independent sequential activities into parallel ones without modifying the whole process behaviour. Improving this metric may however lead to other controllability issues since concurrent tasks are less controllable,
- the *completeness*, which gathers the completeness of the states on the input message set, the completeness of the condition guards in a "if" activity or the completeness of the received messages (especially the received faults). Incomplete processes may lead to deadlocks. For instance, if a value does not satisfy any condition in a "if" activity, the conditions are incomplete (conditions which do not cover the complete domain of a variable), and the process hangs at this activity in a deadlock. And the greater the number of potential deadlocks is, the less testable the BPEL process is too.

This work does not take into account concurrent BPEL processes. It could be extended by modifying/updating the ABPEL transformation to a concurrent system modeling language (synchronized STS, UML activity diagrams ?). The main issue here is that the state number of an intermediate specification may explode on account of the synchronization of the concurrent processes. Testability of parallel BPEL processes also offers new challenging issues: indeed, we don't have one execution handling a symbolic variable set but an execution set composed of parallel processes, handling simultaneously shared variables. The previous testability definitions may become insufficient and metric evaluation may be much more complicated.

6. REFERENCES

[1] Magicdraw homepage. In *http://www.magicdraw.com*.

[2] L. Bentakouk, P. Poizat, and F. Zaïdi. A formal framework for service orchestration testing based on symbolic transition systems. *In TESTCOM/FATES 2009 - 21th IFIP International Conference on Testing of Communicating Systems, LNCS*, 5826/2009:16–32, 2009.

[3] A. Bertolino, G. Angelis, L. Frantzen, and A. Polini. Model-based generation of testbeds for web services. In *TestCom '08 / FATES '08: Proceedings of the 20th IFIP TC 6/WG 6.1 international conference on Testing of Software and Communicating Systems*, pages 266–282, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] L. Briand, Y. Labiche, and Q. Lin. Improving the coverage criteria of uml state machines using data flow analysis. In *Software Testing, Verification and Reliability (STVR) journal*. Wiley interscience, 2009.

[5] A. Brogi and R. Popescu. From bpel processes to yawl workflows. In LNCS, editor, *Web Services and Formal Methods*, september 2006.

[6] O. Consortium. Ws-bpel v2.0. April 2007. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.

[7] N. Een and N. Sörensson. Minisat. 2003. http://minisat.se.

[8] L. Frantzen, J. Tretmans, and T. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer, 2005.

[9] R. S. Freedman. Testability of software components. *IEEE transactions on Software Engineering*, 17(6), 1991.

[10] J. García-fanjul, J. Tuya, and C. D. L. Riva. Generating test cases specifications for bpel compositions of web services using spin. In *Workshop on WebServices Modeling and Testing*, pages 83–94, 2006.

[11] K. Karoui, R. Dssouli, and O. Cherkaoui. Specification transformations and design for testability. In *IEEE Globecom'96, Londre*, Nov. 1996.

[12] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[13] M. Lallali, F. Zaidi, and A. Cavalli. Transforming bpel into intermediate format language for web services composition testing. In *NWESP '08: Proceedings of the 2008 4th International Conference on Next Generation Web Services Practices*, pages 191–197, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Z. J. Li and W. Sun. Bpel-unit: Junit for bpel processes. In *Service-Oriented Computing, ICSOC*, pages 415–426, november 2006.

[15] Z. J. Li, H. F. Tan, H. H. Liu, J. ZHU, and N. M. Mitsumori. Business-process-driven gray-box soa testing. In *IBM systems Journals*, pages 457–472, 2008.

[16] N. Lohmann, P. Massuthe, C. Stahl, and D. Weinberg. Analyzing interacting bpel processes. In *Lecture Notes in Computer Science*, pages 17–32, october 2006.

[17] R. Mateescu and S. Rampacek. Formal modeling and discrete-time analysis of bpel web services. In *Lecture Notes in Business Information Processing, EOMAS 2008*, volume 10, pages 179–193. Springer, June 2008.

[18] V. L. Narasimhan, P. T. Parthasarathy, and M. Das. Evaluation of a suite of metrics for component based software engineering (cbse). *The Journal of Issues in*

Informing Science and Information Technology (iisit), 6(5/6):731–740, 2009.

[19] J. M. Org, J. Mendling, and J. Ziemann. Transformation of bpel processes to epcs. In *In: Proc. of the 4th GI Workshop on Event-Driven Process Chains*, 2005.

[20] S. Salva and H. Fouchal. Some Paramèters for Timed System Testability. In *ACS/IEEE International Conference on Computer System and Applications, AICCSA'01 (Beirut, Lebanon)*, June 2001.

[21] S. Salva and I. Rabhi. A preliminary study on bpel process testability. In *http://sebastien.salva.free.fr/sr10.pdf*.

[22] F. van Breugel and M. Koshika. Models and verification of bpel. 2006.

[23] G. Xiwu and L. Zhengding. A formal model for bpel4ws description of web service composition. *Wuhan University Journal of Natural Sciences*, pages 1311–1319, 2006.

[24] B. B. Yves, Y. L. Traon, and G. Sunyé. Testability analysis of a uml class diagram. In *In Proceedings of the Ninth International Software Metrics Symposium (METRICS03*, pages 54–66. IEEE Computer Society, 2002.

[25] Y. Zheng, J. Zhou, and P. Krause. An automatic test case generation framework for web services. In *Journal of Software*, pages 64–77, September 2007.