A BPEL observability enhancement method. *

Sébastien Salva[1] and Issam Rabhi[2]

[1] LIMOS CNRS UMR 6158, Université d'Auvergne,
Campus des Cézeaux, Aubière, FRANCE
`sebastien.salva@u-clermont1.fr`

[2] LIMOS CNRS UMR 6158, Université Blaise Pascal,
Campus des Cézeaux, Aubière, FRANCE
`rissam@isima.fr`

Abstract. WS-BPEL processes are usually overlapped in large business applications composed of several Web Services. Such applications are more and more developed with respect of quality processes. Testability is an important quality degree, which evaluates the fault detection coverage during the testing process and the testing cost. In this paper, we focus on a well-known testability criterion called observability, which evaluates if enough distinguishable events can be observed while testing. We study the observability of ABPEL (Abstract-BPEL) specifications and we describe some ABPEL observability degradation properties (ABPEL code patterns which do not respect the observability definition). From these, we propose an observability enhancement method which detects observability issues in ABPEL specifications and semi-automatically updates the code.

Keywords: BPEL, testability, observability, enhancement method

1 Introduction

Internet is emerging as a Web Service based platform where more and more organizations are implementing and deploying Business processes for the five last years. Web Services offer many advantages such as the resource virtualization or the externalization of functional code in a standardized way. They also represent the foundation stones of larger Business processes. WS-BPEL (Web Services Business Process Execution Language [3]) gathers these "stones" by providing an upper layer which orchestrates Web Services by describing the service 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.

Such processes require strict software life cycle and so entail the use of quality processes and models, like the CMMI one (Capability Maturity Model Integration [20]). Among these quality processes, the testing activity takes a solid place

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

since it costs as much as 50 % of the total development effort. To reduce this substantial cost, there is a significant trend in the study of methodologies for testable software development. Analyzing testability, all along the development life cycle, helps to achieve an application from which the testing stage is able to detect the largest number of bugs, with the less efforts in regards to bug detection and time.

This paper focuses on the ABPEL (Abstract BPEL) specification observability. This latter is a well-known testability criterion whose the purpose is to evaluate if enough distinguishable events can be observed during the testing process to give a significant verdict. When observability is degraded, testing methods may only cover a part of the implementation and give a distorted verdict [6]. Among the BPEL modeling languages (ABPEL, UML, Petri nets, process algebra, abstract state machines (ASM),etc. [2]), we base our choice on the ABPEL (Abstract BPEL) formalism since it is widely used in software development companies and with existing testing methods [1].

To the best of our knowledge, there are few works on ABPEL (or BPEL) observability in literature. Observability definitions are typically given on input/ouput models [8] whereas ABPEL is composed of structured and basic activities which can be nested, and gathers specific features (fault handlers, correlations, etc.). To link ABPEL specifications with observability, we analyze an intermediate model, the STS (Symbolic Transition System [7]) and the transformation rules, allowing to translate ABPEL specifications into STSs. From known STS observability issues, we begin to deduce the corresponding ABPEL observability degradation properties. Then, we propose an observability enhancement method which parses ABPEL specifications, recognizes observability issues and semi-automatically removes them by modifying the ABPEL code. The method goal is not to preserve the standard semantics of the ABPEL specification, but to achieve a more testable one, at the expense of other properties such as the performance for instance.

The remainder of this paper is structured as follow: Section 2 provides an overview on the BPEL language and on system testability. In Section 3, we describe the BPELtoSTS transformation and several BPEL observability degradation propositions. Section 4, presents the observability enhancement method. Section 5 concludes and gives some perspectives on BPEL testability.

2 BPEL and testability Overview

2.1 BPEL

Business Process Execution Language for Web Services (BPEL or BPEL4WS [3]) defines and manages business processes based on interactions of Web Services, called partners. BPEL describes the element declarations (exchanged messages, partner roles, links, etc.) and the partner orchestration which includes the internal details (states, variables, requests, etc.). BPEL proposes basic activities, to perform basic operations (invoke, receive, etc.), and structured activities (scope,

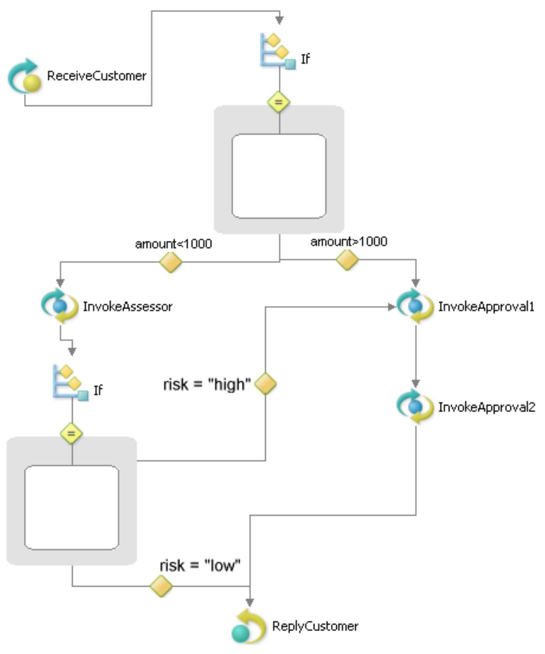

Fig. 1. The Loan approval BPEL process

flow, etc.) to set the activity structuring. Among its possibilities, it supports fault handler activities whose the purpose is to manage fault messages or other exceptions returned by external Web Services (the process can be compensated or terminated, etc.). BPEL processes also handle Web Service WSDL descriptions [4] which gather the required data to invoke a service, by defining the Web Service interfaces, the accessible methods, called operations, and the operation parameter/response types.

The BPEL code is written by mean of the XML language. For readability reason, we only illustrate in Figure 1 a graphical specification of the *loan approval sample*, which is derived of the BPEL code which can be found in the WS-BPEL specification [3]. This graphical example conceals a more complex code, as shows Figure 3, where is given the ABPEL code of the two first activities. 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. According to the loan amount, different Web Service partners (”loan assessor”, ”loan approval1” and ”loan approval2”) are involved to check the client creditworthiness. The final response is either ”accept” or ”refused”.

Many works have been proposed on BPEL and some of them tackle especially to its validation [14, 21, 15, 9, 23, 13]. 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 [21], 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 [15], 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 [9], the authors transform BPEL specifications to PROMELA code, and then generate test cases by using the SPIN model-checking tool. In [23], 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 [13], 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.

2.2 Testability

Testability gathers several criteria which evaluate the system capability to reveal its faults, the accessibility of its components and its testing cost [8]. Designers evaluate the system testability after each life cycle step, so they can distinguish the testable system parts from the untestable ones. They can also assess the testing cost. Testability can be used to model and to implement more 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 [8, 11, 22, 18]) and timed ones (timed automata [19]). These methods evaluate testability criteria one by one by returning for instance, a factor called *degree* (a number between 0 and 1), the number of potential faults which can be detected, or the number of testability issues. Many criteria have been studied, depending on the system, but observability is commonly used. For input output systems, observability is defined in [8] by:

Definition 1. *A system is observable if for each input given to the system, a different output is observed.*

In testing methods, an observable specification is often required since it helps to determine the system internal state according to the observed outputs, during the test execution. To the best of our knowledge, few works have been proposed to improve testability by mean of specification transformation. In [10], some "C" and "Java" code transformations are proposed to improve the test set generation by manipulating input variables. In [17], Java applications are transformed into partial oracles which help to improve the test data generation.

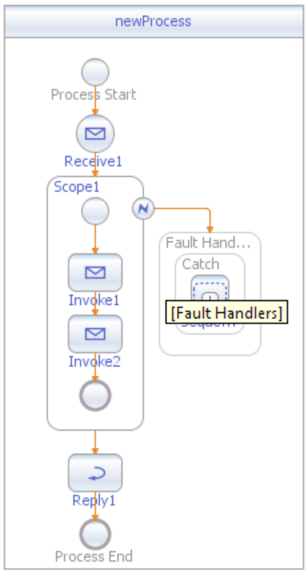

Fig. 2. An ABPEL process

```
<bpel:receive createInstance="yes" operation="request"
   partnerLink="customer" portType="lns:loanServicePT"
   variable="request">
   <bpel:sources>
     <bpel:source linkName="receive-to-assess">
       <bpel:transitionCondition>
         ($request.amount > 10000)
       </bpel:transitionCondition>
     </bpel:source>
     <bpel:source linkName="receive-to-approval">
       <bpel:transitionCondition>
        ($request.amount < 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>
```

Fig. 3. A part of the ABPEL code

3 BPEL observability study

Defining ABPEL observability degradations is not an obvious task since on the one hand we have an observability definition based on input/output events, and on the other hand we have the ABPEL language which is a melting pot of notions (partners, correlations, fault management, etc.), where structured and basic activities are nested. For instance, the scope activity of Figure 2 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.

So, our methodology consists in considering and analyzing an intermediate model, the STS (Symbolic Transition System [7]) and the transformation rules, allowing to translate ABPEL specifications into STSs. Then, from known STS observability issues, we search for the corresponding ABPEL observability degradation properties, thanks to the BPELtoSTS rules. We base our choice on the STS formalism because this one is widely used in testing methods [7, 1], and because STSs are state machine based models, where the observability issues are already known. Some works [16, 1] also introduce BPELtoSTS transformation methods.

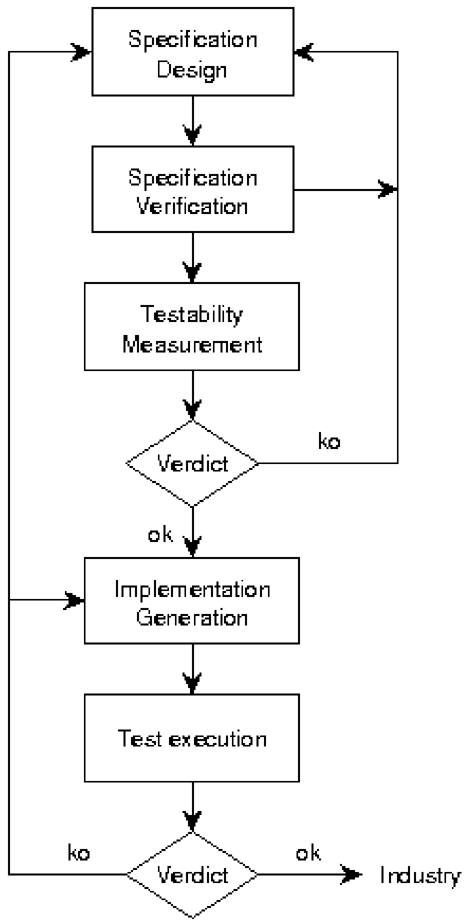

Fig. 4. Testability in software life cycle

In the following, we describe briefly the BPELtoSTS transformation. This one flattens each nested BPEL activity, spreads the fault handlers of one activity into each of its sub-activities, and eliminates the irrelevant activities for testability analysis (for instance, the ”empty” one). We obtain a model which can be analyzed with relative ease. Then, we focus on the ABPEL observability degradation and we introduce a non exhaustive list of propositions.

3.1 BPEL to STS transformation

A BPELtoSTS transformation has been proposed in [16]. This one is mainly based on rules, whose the semantics are expressed with a process algebraic style. Nevertheless, nested activities are not supported, so scope, faulthandlers or process activities cannot be used. Consequently, we extend this work by redefining or completing the transformation rules. Most of them are given in Figure 8. Basically, when a rule is composed of several parts, the upper one expresses a condition while the lower part describes the STS transformation. Below, we also propose an associated transformation algorithm which handles them.

Formally, we define that a BPEL activity belongs to $SA \cup BA$, where SA is the set of structured activities and BA the set of basics activities, relative to the WS-BPEL specification:

– a **structured activity** $sa = ((sa_1, ..., sa_n), FH_{sa}, CH_{sa}, TH_{sa}) \in SA$ is composed of a list $(sa_1, ..., sa_n)$ of activities in $(SA \cup BA)^n$, of a fault handler set $FH \in SA$, of a compensation handler set $CH \in SA$ and of a termination handler set $TH \in SA$. Compensation and termination handlers are directly called by a fault handler when a fault occurs and are used to reset or to terminate the current process. FH, CH and TH may be empty,
– a **basic activity** $ba = (ba_1, FH_{ba}, CH_{ba}, TH_{ba}) \in BA$ is composed of an action ba_1 (invoke, receive, ...), of a fault handler $FH \in SA$, a compensation one $CH \in SA$ and a termination one $TH \in SA$. FH, CH and TH may be empty as well,
– a **fault handler activity** $fh(catch(?f_1, act_1), ..., catch\ (?f_n, act_n), catchall(act)) \rightarrow e \in SA$ is a structured activity triggered by received WSDL faults $?f_1, ..., ?f_n$. e is the state reached by fh once terminated. We also denote $FH \rightarrow e$ a fault handler activity set where each activity $fh_i \rightarrow e \in FH \rightarrow e$ reaches e once terminated. Each fault is itself composed by a variable list (faultName, faultElement, faultMessageType).

In summary, the BPELtoSTS transformation algorithm successively develops each structured activity to a graph of sub-activities. We start with the initial BPEL process activity which is developed to a first STS transition set. Then, each activity, encountered on a transition, is developed and so on, until there is no other activity to develop. To transform basic and structured activities into STS transitions, the algorithm refers to the rule set, given in figure 8.

BPEL to STS transformation algorithm

1. We construct the initial STS composed of the transition $e_1 \xrightarrow{P((p_1,...,p_n),FH_p\rightarrow e_{n+1},CH_p,TH_p)} e_{n+1}$ where P is the initial BPEL process. $FH_p\rightarrow e_{n+1}$ means that if a fault handler $fh \in FH_p\rightarrow e_{n+1}$ is thrown, once this one ends, the execution proceeds from the state e_{n+1}. We set $SA = P((p_1,...,p_n),FH_p\rightarrow e_{n+1},CH_p,TH_p)$,
2. Let the transition $e_1 \xrightarrow{SA} e_{n+1}$ with $SA = ((sa_1,...,sa_n),FH\rightarrow e_{n+1},CH, TH)$. We develop SA and construct the corresponding STS transitions:
   - if SA is a "while" (or "if", "pick") activity, we translate SA to $SA' = ((sa'_1,...,sa'_n),\emptyset,\emptyset,\emptyset)$ with one of the rules given in figure 8 composed of the condition $FH \neq \emptyset$. With this rule, we spread the fault handler set of the initial structured activity into each nested sub-activity. For instance, the fault handler set of a scope activity is spread into sub activities such as "while", "if" or "invoke". Then, we use one of the rules of figure 8 to translate SA' into STS transitions (rule with the condition $FH = \emptyset$),
   - if SA is a structured activity different from "while" (or "if", "pick"), for each $sa_i \in (sa_1,...,sa_n)$:
     - If $sa_i \in BA$, we use one of the rules given in figure 8,
     - if $sa_i \in SA$, we construct the transition $e_i \xrightarrow{sa_i} e_{i+1}$ with $sa_i = ((sa_{i1},...,sa_{im}),\ FH_{sa_i}\rightarrow e_{i+1} \cup FH\rightarrow e_{n+1}, CH_{sa_i} \cup CH,\ TH_{sa_i} \cup TH)$. As previously, we spread the fault handler of the initial structured activity into each nested sub-activity. We denote the fault handler $FH_{sa_i}\rightarrow e_{i+1}$ owing to the activity sa_i which ends at the state e_{i+1}. If a fault handler of sa_i, $fh\rightarrow e_{i+1} \in FH_{sa_i}\rightarrow e_{i+1}$ is caught, once executed, the process must proceed from the state e_{i+1}.
3. While it exists a undeveloped activity $e_i \xrightarrow{SA} e_j$, we use 2.

We applied the BPELtoSTS transformation on the specification example, depicted in Figure 1. We obtain the STS of Figure 5, which preserves the initial ABPEL specification semantic. For instance, the first "reply" activity is translated into the first transition from the location A1. A "fault handler" activity begins at the location A10. The outgoing transitions labeled by "?faultId" models a "catch" activity.

Below, we provide some definitions and notation to be used throughout the paper.

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 set L with l_0 the initial one, an internal 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.

[7] 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

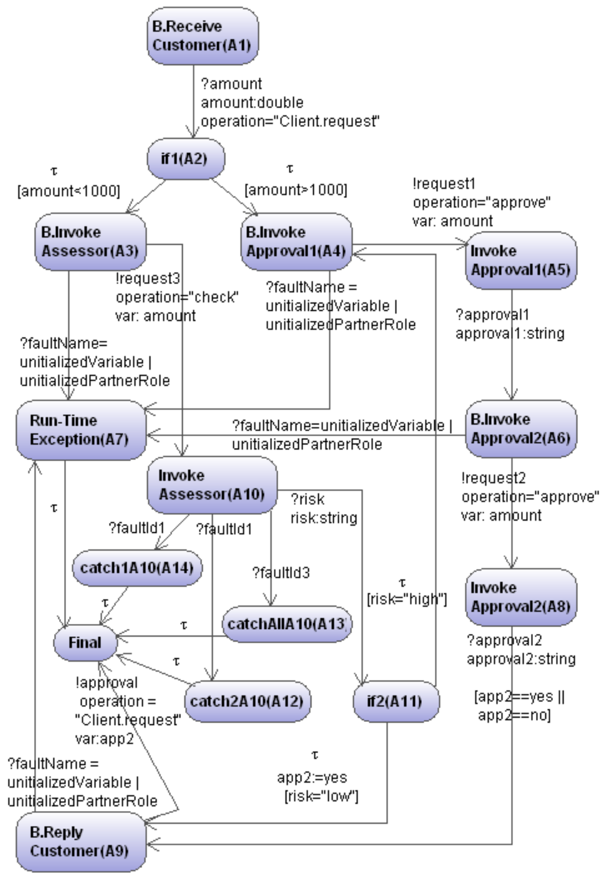

Fig. 5. An STS modeling an ABPEL process

the interaction variables of s to ground terms. For a reaction (s, η), η is observable in the message s. We consider that 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 $(\varphi_1(\eta) \wedge, ..., \wedge\, \varphi_i(\varrho_{i-1}, \eta'_1))$ true$\}$.

3.2 ABPEL observability degradation

In the following, we assume that the partners, taking part to the BPEL orchestration, are observable. The BPELtoSTS transformation rules, that we consider in the propositions, are those given in Figure 8.

Observability aims to evaluate the system internal state from its observed outputs. According to the definition given in Section 2.2, the system observability is degraded if different 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 (proofs are trivial).

- **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 degradation property, we have deduced two corresponding ABPEL degradations: "the lack of a reply activity at the end of ABPEL processes" and "the use of conditions with if activities which cannot be always satisfied".

Proposition 1. *An ABPEL specification not terminated by a "reply" (one-way "invoke") activity is not observable.*

Proof. Both "reply" and one-way "invoke" activities produce an STS transition $(l_j, l_{j+1}, !o, \varphi_j, \varrho_j)$. The following proof considers "reply" activities only. The reasoning is the same with one-way "invoke" ones.
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 [3]). The translation of *bpel* into STS (rule 2) 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 STS transitions, the first one labeled by an output symbol and the other one labeled by an input symbol (rule 8). The last "invoke" activity of *bpel* produces the STS transitions $(l_i, l_{i+1}, !o, \varphi, \varrho)$, $(l_{i+1}, l_{i+2}, ?e, \varphi_{i+1}, \varrho_{i+1})$. If this last "invoke" activity is not followed by a "reply" one, there is no path from l_{i+2} composed of output symbols. Consequently, no reaction can be observed, $\forall\eta, out(l_{i+2}, \eta) = \emptyset$. *bpel* is not observable.

Proposition 2. *An ABPEL specification, composed of an "invoke" (or "receive") activity followed by an "if" one $if((cond_1, act_1),..., (cond_n, act_n))$ 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 $(l, l', !o, \varphi, \varrho)$, $(l', l'', ?e, \varphi', \varrho')$ (rule 8). A "receive" activity is transformed by the last transition only. The "if" activity gives the transitions (rule 5) $\forall(1 \leq i \leq n)$ $(l'', l_i, \tau, \varphi_i, \emptyset)$ with $\varphi_i = cond_i$. Suppose that it exists η such as $\bigvee_{1\leq i\leq n} (\varphi_i(\eta))$ false. No transition can be fired. The location l'' is blocked and is said quiescent [7]. So, it exists a stimulus $(?e, \eta)$ such as $out(l'', \eta) = \emptyset$. Consequently, both the STS and the ABPEL specification are not observable.

The four following ABPEL testability degradation propositions result from the second STS degradation property. With STSs, this degradation is raised 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 specification 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 "a block of internal activities bound by two "receive" activities".

Proposition 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 [3], 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 (rule 9) $(l_i, l_{i+1}, ?fault_i, \emptyset, \varrho_i)$ and $(l_j, l_{j+1}, ?fault_j, \emptyset, \varrho_j)$, 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 (rule 8) $(l_{i+1}, l_{i+2}, !o, \emptyset, \varrho_{i+1})$, $(l_{j+1}, l_{j+2}, !o, \emptyset, \varrho_{j+1})$. Consequently, it exists two stimuli $(?fault_i, \eta_i)$, $(?fault_j, \eta_j)$ such as $out(l_{i+1}, \eta_i) = out(l_{j+1}, \eta_j) = (!o, \eta)$. The STS and the ABPEL specification are not observable.

Proposition 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 $?fault_j = (faultName = n_j, fault-Element = e_j, faultMessage = m_j\ Type = t_j)$ be two different faults. This activity is translated into the STS transitions (rule 9) $(l_i, l_{i+1}, ?fault_i, \emptyset, \varrho_i)$ and $(l_i, l_{i+1}, ?fault_j, \emptyset, \varrho_j)$, with $?fault_i \neq ?fault_j$. If this "catchall" activity is followed by an "invoke" one, we also have the transition $(l_{i+1}, l_{i+2}, !o, \emptyset, \varrho_{i+1})$ (rule 8) with $!o = (op, req, partner)$.

Let $(?fault_i, \eta_i)$, $(?fault_j, \eta_j)$ be two stimuli and η a ground term such as $\eta_i = \varrho_i(\eta)$ ($\eta_j = \varrho_j(\eta)$ respectively). 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 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 $?e_i \neq ?e_j$. The invoke activities give the two transitions $(l_i, l_{i+1}, !o, \varphi_{i+1}, \varrho_{i+1})$, $(l_j, l_{j+1}, !o, \varphi_{j+1}, \varrho_{j+1})$ (rule 6). We obtain the same case as the one described in Proposition 3. Thus, the STS and the ABPEL specification are not observable.

Proposition 6. *An ABPEL specification, composed of a block of $n \geq 0$ internal activities (no "invoke", "reply", "throw" activities) bound by two "receive" activities, is not observable.*

Proof. Such an activity block is transformed by STS paths such as $(l_1, l_2, ?e_1, \varphi_1, \varrho_1)$, $(l_2, l_3, e_2, \varphi_2, \varrho_1)$,..., $(l_{n-1}, l_n, e_{n-1}, \varphi_{n-1}, \varrho_{n-1})$, $(l_n, l_{n+1}, ?e_n, \varphi_n, \varrho_n)$ with $(e_2, ..., e_{n-1}) \notin S_O^{n-2}$ (no output message). If this activity block is not followed by an "invoke" (or "reply"or "throw") one, $\forall(\eta_k, \eta_l)$, such as $(\varphi_1(\eta_k) true)$ and $(\varphi_n(\eta_l) true)$, $out(l_2, \eta_k) = out(l_{n+1}, \eta_l) = \emptyset$. Otherwise, the activity block is followed by an "invoke" (or "reply" or "throw") activity which produces an STS transition $(l_j, l_{j+1}, !o, \varphi_j, \varrho_j)$ such as $\exists \eta_j, \varphi_j(\eta_j)$ true. $\forall(\eta_k, \eta_l)\ out(l_2, \eta_k) = out(l_{n+1}, \eta_l) = (!o_j, \eta_j)$. The second STS degradation property is satisfied, hence both the STS and the ABPEL specification are not observable.

4 BPEL observability enhancement

The previous observability degradation propositions are used here to provide some enhancement algorithms and a corresponding tool. This latter parses an ABPEL specification, detects observability degradations, according to the Propositions 1-6 and removes them. The tool architecture is given in Figure 6.

The ABPEL specification modification may require the update of some partner WSDL descriptions or of some source code parts. This is why we denote this enhancement method as semi-automatic. The different enhancement steps are detailed below. In the following, we denote *bpel*, an ABPEL specification:

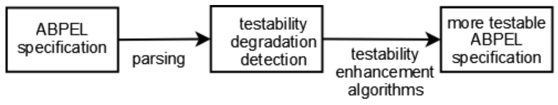
ABPEL specification
parsing
testability degradation detection
testability enhancement algorithms
more testable ABPEL specification

Fig. 6. An observability enhancement tool

- **"reply" activity addition:** we check that each branch of *bpel* ends with a "reply" (invoke-only) activity (Proposition 1). 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 $branch_i$" which is supposed to be a unique output message, not yet used in the specification (line 3). (the observability is not degraded with the addition of this output message),
- **"if" condition addition:** according to Proposition 2, an "if" activity $if((cond_1, act_1),..., (cond_n, act_n))$, which cannot be always satisfied
  ($\exists\eta \bigvee_{1\leq i\leq n} cond_i(\eta)$ false), may degrade the specification observability. We complete such an "if" activity by adding an "<else>" conditional branch which represents the disjunction of all other conditions. This new branch ends in a "reply" activity to not degrade the observability, in regards to Proposition 1 (an ABPEL process must end in a "reply" activity). More precisely, the algorithm, given in Algorithm 2, adds a branch which throws a new fault "$FLT : badCondition_k$" (line 4). This fault is caught by a new "catch" activity, added in the current "scope" block (lines 5-6). This activity is composed of a "reply" one sending the fault "$FLT : badCondition_k$" to the client side,
- **"invoke" activity distinction:** Propositions 3-5 refer to the same STS observability degradation: "2 different stimuli are followed by the same reaction". This reaction is produced by two "invoke" activities which call the same operation *op* with the same parameter values. We propose to distinguish the two partner calls (to produce different reactions), by completing *op* with a new variable, taking a different value according to the stimulus. For instance, in Proposition 3, the same reaction is observed on account of two "invoke" activities using the same operation after two different "catch" activities $catch_i(?fault_i, act_i)$, $catch_j(?fault_j, act_j)$. Here, the two "invoke" activities can be distinguished by the fault name. In this case, we add a variable $fault$ to *op* equals to the fault name.

  Propositions 4-5 are based on the same idea. We only give, in Algorithm 3, the methodology dedicated to Proposition 3. The other ones require only minor modifications. In Algorithm 3, 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 (line 1), we compute, from the corresponding STS, the values satisfying the execution of these invocation, with constraint solvers [12, 5] (lines 4-5). If at least one value satisfies both "invoke" activities (Proposition 3 verified) (line 6), we modify them by adding a new parameter equals to $?fault_i$ ($?fault_j$ respectively) (lines 7-8). According to Proposition 3, $?fault_i$ and $?fault_j$ are not identical, so we obtain different reactions when *op* is called. This modification also requires to update the WSDL description and the source code of the called partner. The constraint solvers construct values satisfying the guards of a specification path. We use the solvers in [12, 5] which work as external servers and manage most of the simple types, "string" included,

– **Asynchronous call modification:** asynchronous calls correspond to partner invocations whose the response receipt is delayed and where there is no need to wait before proceeding further. Two asynchronous calls are modeled by two first invoke-only activities $invoke_i(op_i, req_i, \emptyset, partner_i, corr_i)$ and $invoke_j(op_j, req_j, \emptyset, partner_j, corr_j)$, followed later by two "receive" ones $rcv_i(op_i, resp_i, partner_i, corr_i)$, $rcv_j(op_j, resp_j, partner_j, corr_j)$, linked by the same correlation sets. When the two "receive" activities are successive, the observability is degraded (Proposition 6). We propose here to insert $invoke_j$ between the two "receive" activities to obtain observable reactions after rcv_i (Proposition 6 no more satisfied). However, we cannot only move the $invoke_j$ activity in the APBEL code since this one depends on the variables req_j which may be updated in the process. Algorithm 4 replaces $invoke_j$ by an "assign" activity which copies the req_j variables into new ones, denoted $req2_j$ (line 3). These variables are only used with the $invoke_j$ activity which is inserted before rcv_j (line 4).

We applied this enhancement method on the ABPEL specification illustrated in Figures 1,5. This one is composed of six observability issues: (1) no reaction is observed from the location A2 with the stimulus (τ, $amount = 1000$). (2,3) both the stimuli ($?risk$, $risk = low$) and ($?approval2$, $app2 = yes$) are followed by the same reaction ($!approval$,$app2 = yes$). (4,5,6) The last observability issues are detected at the location A10 (Invoke Assessor), where three input messages modeling "catch" activities are not followed by output ones and where two messages "?faultid1" give the same observation. Once the enhancement method applied, we obtain a new ABPEL specification and its corresponding STS, illustrated in Figure 7. This one is much more observable since the observability degradation number is lowered to one. Indeed, three different "reply" activities (from locations A12, A13 and A14) have been added, two identical "catch(?faultid1)"

Algorithm 1: "reply" activity addition

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

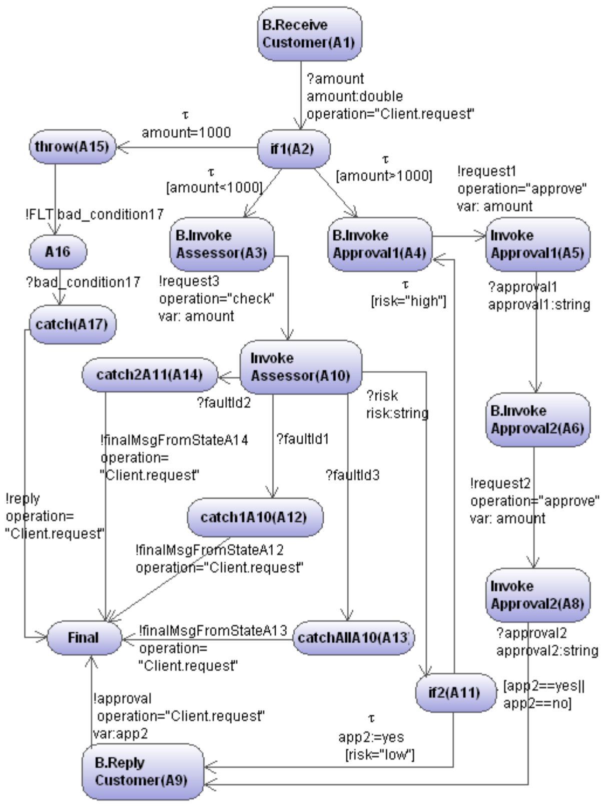

Fig. 7. The new STS specification

Algorithm 2: "if" condition addition

```
  input  : ABPEL specification *bpel*
1 foreach "invoke" activity I followed by an "if" activity
  $if_k((conf_1, act_1), ..., (cond_n, act_n))$ do
2     Compute $V$ such as $\forall v \in V \bigvee_{1 \le i \le n} (cond_i(v))$ false with constraint solvers ;
3     if $V \neq \emptyset$ then
4         Add "<else> <throw faultName= "$FLT : badCondition_k$"/></else>";
5         Add this "catch" activity in the faulthandler of the "scope" activity
          containing $if_k$: "<catch faultName= "$FLT : badCondition_k$"> REPLY
          </catch>";
6         where REPLY=$reply(resp, partner, op)$ with partner=client, op= client
          operation used for calling the ABPEL process,
          resp="$FLT : badCondition_k$";
```

Algorithm 3: "catch" activity distinction

```
input : ABPEL specification $bpel$
1 if it exists $catch_i(?fault_i, act_i)$, $catch_j(?fault_j, act_j)$ in bpel followed by the
  same operation call op with the parameter types $(p_1, ..., p_m)$ then
2     Compute $sts =< L, l_0, Var, var_0, I, S, \rightarrow>$;
3     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
4         Compute with solvers the value set $V_1$ over $(p_1, ..., p_m)$ such as $\forall v \in V_1$,
          $(\varphi_0(v) \wedge \varphi_1(\varrho_0)\wedge,..., \wedge\varphi_i(\varrho_{i-1}))$, $true$;
5         Compute $V_2$ from the path $p_j$;
6         if $V_1 \cap V_2 \neq \emptyset$ then
7             Add to the first invoke activity a string parameter "fault" with the
              value "$?fault_i$" ;
8             Add to the second one a string parameter "fault" with the value
              "$?fault_j$" ;
```

Algorithm 4: Asynchronous call modification

```
input : ABPEL specification $bpel$
1 foreach successive receive activities $rcv_i(op_i, resp_i, part_i, corr_i)$,
  $rcv_j(op_j, resp_j, part_j, corr_j)$, with $corr_i \neq corr_j \neq \emptyset$ do
2     Find the invokeonly activity $invoke_j(op_j, req_j, \emptyset, part_j, corr_j)$;
3     Replace $invoke_j$ by an "assign" activity copying $req_j$ into $req2_j$;
4     Insert $invoke_j(op_j, req2_j, \emptyset, part_j, corr_j)$ between $rcv_i$ and $rcv_j$, ;
```

activities (location A10) have been distinguished with "?faultid1", ?faultid2, and the "if" condition has been completed (location A2 with $amount = 1000$).

5 Conclusion

This paper proposes an observability enhancement method of ABPEL specifications. From known STS observability issues, we have deduced some corresponding ABPEL observability degradation propositions. These ones can be used to directly write more testable ABPEL specifications or to evaluate observability. From these propositions, we provide several observability enhancement algorithms, which have been implemented in an academic tool. This one parses the specification, detects observability degradation patterns and modifies the ABPEL code semi-automatically.

We have chosen to improve observability by modifying the ABPEL specification and sometimes the interface of the involved partners. Instead of changing these interfaces, another solution would be to add a "logger" service whose the role would be to give new observable reactions. This solution is interesting with the *"reply" activity addition* algorithm but we still need to modify the partner interfaces with the *"invoke" activity distinction* one. Nevertheless, we will explore the benefit of a "logger" service in a future work.

The ABPEL observability degradation set, that we have presented, is not exhaustive. For instance, there are many cases where two stimuli produce the same reaction. Some of them can be identified to provide new enhancement algorithms. Other cases are more difficult to classify. For instance, the issue detected in the location A9 (Figure 5 (($?risk$, $risk = low$) and ($?approval2$, $app2 = yes$) give the same reaction ($!approval$,$app2 = yes$)), depends on different activities and may require a strong modification of the specification behaviour.

Many other quality criteria could be also studied such as the *controllability*, which evaluates whether an implementation can be controlled enough to obtain test results, the *execution time*, which evaluates the testing cost according to the minimal and maximal execution times, or the *accessibility* of BPEL parts.

This observability enhancement method does not guarantee to preserve the standard semantics of an ABPEL specification but to achieve a more testable one. So, in a future work, it would be interesting to discuss about the trade-off between the observability improvement and the degradation of other properties, such as the performance.

References

1. Bentakouk, L., Poizat, P., Zaïdi, F.: A formal framework for service orchestration testing based on symbolic transition systems. 21th IFIP International Conference on Testing of Communicating Systems 5826/2009, 16–32 (2009)
2. van Breugel, F., Koshika, M.: Models and verification of bpel (2006)
3. Consortium, O.: Ws-bpel v2.0 (April 2007), http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf
4. Consortium, W.W.W.: Web services description language (wsdl) (2001)
5. Een, N., Sörensson, N.: Minisat (2003), http://minisat.se
6. Escobedo, J.P., Gaston, C., Gall, P., Cavalli, A.: Observability and controllability issues in conformance testing of web service compositions. In: 21st IFIP WG 6.1 International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop. pp. 217–222. Springer-Verlag (2009)
7. Frantzen, L., Tretmans, J., Willemse, T.: Test Generation Based on Symbolic Specifications. In: Grabowski, J., Nielsen, B. (eds.) Formal Approaches to Software Testing – FATES 2004. pp. 1–15. No. 3395 in Lecture Notes in Computer Science, Springer (2005), http://www.cs.ru.nl/ lf/publications/FTW05.pdf
8. Freedman, R.S.: Testability of software components. IEEE transactions on Software Engineering 17(6) (june 1991)
9. García-fanjul, J., Tuya, J., Riva, C.D.L.: Generating test cases specifications for bpel compositions of web services using spin. In: Workshop on WebServices Modeling and Testing. pp. 83–94 (2006)
10. Harman, M., Baresel, A., Binkley, D., Hierons, R.M., Hu, L., Korel, B., McMinn, P., Roper, M.: Testability transformation - program transformation to improve testability. In: Formal Methods and Testing. Lecture Notes in Computer Science, vol. 4949, pp. 320–344. Springer (2008)

11. Karoui, K., Dssouli, R., Cherkaoui, O.: Specification transformations and design for testability. In: IEEE Globecom'96, Londre (Nov 1996)
12. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis. pp. 105–116. ACM, New York, NY, USA (2009)
13. Li, Z.J., Tan, H.F., Liu, H.H., ZHU, J., Mitsumori, N.M.: Business-process-driven gray-box soa testing. In: IBM systems Journals. pp. 457–472 (2008)
14. Li, Z.J., Sun, W.: Bpel-unit: Junit for bpel processes. In: Service-Oriented Computing ICSOC. LNCS, vol. 4294, pp. 415–426. Springer-Verlag (november 2006)
15. Lohmann, N., Massuthe, P., Stahl, C., Weinberg, D.: Analyzing interacting bpel processes. Lecture Notes in Computer Science, vol. 4102, pp. 17–32. Springer-Verlag (october 2006)
16. Mateescu, R., Rampacek, S.: Formal modeling and discrete-time analysis of bpel web services. In: Lecture Notes in Business Information Processing, EOMAS 2008. vol. 10, pp. 179–193. Springer-Verlag (Jun 2008)
17. McMinn, P.: Co-testability transformation. In: Schlingloff, H., Vos, T.E.J., Wegener, J. (eds.) Evolutionary Test Generation. No. 08351 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany (2009)
18. Narasimhan, V.L., Parthasarathy, P.T., Das, M.: 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. Salva, S., Fouchal, H.: Some parameters for timed system testability. Computer Systems and Applications, ACS/IEEE International Conference on 0, 0335 (2001)
20. Software-Engineering-Institute: Capability maturity model integration (2010), http://www.sei.cmu.edu/cmmi/
21. Xiwu, G., Zhengding, L.: A formal model for bpel4ws description of web service composition. Wuhan University Journal of Natural Sciences pp. 1311–1319 (2006)
22. Yves, B.B., Traon, Y.L., Sunyé, G.: Testability analysis of a uml class diagram. In: In Proceedings of the Ninth International Software Metrics Symposium (METRICS03. pp. 54–66. IEEE Computer Society (2002)
23. Zheng, Y., Zhou, J., Krause, P.: An automatic test case generation framework for web services. JSW 2(3), 64–77 (2007)

BPEL	STS transformation rules
$assign(var\_update)$ or $empty$	$e_i \xrightarrow{\tau,\emptyset,\varrho} e_{i+1}, \varrho = var\_update$
$receive(op, resp, partner, corr,$ $FH, CH, TH)$	$\dfrac{fh_k \in FH}{e_i \xrightarrow{?(op,resp,partner,corr),[c_i==corr],\varrho(resp)} e_{i+1} \wedge e_i \xrightarrow{fh_k} e_{n+1}}$
$reply(op, req, partner, corr,$ $FH, CH, TH)$	$e_i \xrightarrow{!(op,req,partner,corr),\emptyset,(c_i:=corr)} e_{i+1}$
$throw(!f, FH, CH, TH)$	$\dfrac{fh_k \in FH}{e_i \xrightarrow{!f,\emptyset,\emptyset} e_i 1 \xrightarrow{fh_k} e_i fk}$
$if((cond_1, act_1), ...,$ $(cond_n, act_n), FH, CH, TH)$ or $switch$	$\dfrac{1 \leq l \leq n \wedge FH = \emptyset}{e_i \xrightarrow{\tau,[cond_l],\emptyset} e_{il} \xrightarrow{act_l} e_{i+1l}}$ $\dfrac{1 \leq l \leq n \wedge FH \neq \emptyset}{if(((cond1, act'_1),...,(cond_n, act'_n)),\emptyset,\emptyset,\emptyset) \text{ with } act'_k =}$ $\overline{((act'_{k1},...,act'_{km}), FH_{act_k} \cup FH, CH_{act_k} \cup CH, TH_{act_k} \cup TH)}$
$pick((mess_1, act_1), ...,$ $(mess_n, act_n), onalert(t, act_{n+1}),$ $FH, CH, TH)$ with $mess_k = ($ $op_k, resp_k, partner_k, corr_k)$	$\dfrac{1 \leq k \leq n \wedge FH = \emptyset}{e_i \xrightarrow{?mess_k,\emptyset,\varrho(resp_k)} e_{ik} \xrightarrow{act_k} e_{i+1k}}$ $\overline{e_i(reset\_timer(t)) \xrightarrow{\tau,[t>=Time],\emptyset} e_{in+1} \xrightarrow{act_{n+1}} e_{i+1n+1}}$ $\overline{\wedge e_i \xrightarrow{\tau,[t<TIME],\emptyset} e_i}$ $\dfrac{1 \leq k \leq n+1 \wedge FH \neq \emptyset}{pick(((m_1, act'_1),...,(m_n, act'_n),(onalert(t, act_{n+1'})),\emptyset,\emptyset,\emptyset)}$ $\overline{\text{with } act'_k = ((act'_{k1},...,act'_{km}), FH_{act_k} \cup FH,}$ $\overline{CH_{act_k} \cup CH, TH_{act_k} \cup TH)}$
$while(cond, act, FH, CH, TH)$	$\dfrac{FH = \emptyset}{e_i \xrightarrow{[cond]} e_k \xrightarrow{\tau,[act],\emptyset} e_i \wedge e_i \xrightarrow{\tau,[\neg cond],\emptyset} e_{i+1}}$ $\dfrac{1 \leq k \leq n \wedge FH \neq \emptyset}{while(cond,(act'_1,...,act'_n),\emptyset,\emptyset,\emptyset) \text{ with } act'_k = ((act'_{k1},...,act'_{km}),}$ $\overline{FH_{act_k} \cup FH, CH_{act_k} \cup CH, TH_{act_k} \cup TH)}$
$invoke(op, req, resp, partner,$ $corr, FH, CH, TH)$	$\dfrac{resp \neq null, fh_k \rightarrow e_m \in FH}{e_i \xrightarrow{!(op,req,partner),\emptyset,\varrho=(c_i:=corr)} e_l \wedge}$ $\overline{e_l \xrightarrow{?(op,resp,partner),[c_i==corr],\emptyset} e_{i+1} \wedge e_l \xrightarrow{fh_k} e_m}$ $\dfrac{resp = null}{e_i \xrightarrow{!(op,req,partner)} e_{i+1}}$
$fh(catch_1(?f_1(n_1, e_1, m_1), act_1)$ $, ..., catch_n(?f_n((n_n, e_n, m_n)),$ $act_n), catchall(?f(n_{n+1}, e_{n+1},$ $m_{n+1}), act))$	$\forall 1 \leq l \leq n, e_i \xrightarrow{?f_l,\emptyset,\varrho_l} e_{il} \xrightarrow{act_l} e_{ilf}$ $e_i \xrightarrow{?f[f \neq f_1,...,f \neq f_n],\varrho_n} e_{in+1} \xrightarrow{act} e_{n+1f}$ $\varrho_l = (faultName_l = n_l, faultElt_l = e_l,$ $faultMess_l = m_l)$
$scope((act_1, ..., act_n),$ $FH, CH, TH)$ or $process$ or $sequence$	$\forall 1 \leq l \leq n e_l \xrightarrow{act_l} e_{l+1}$ $\forall 1 \leq k \leq n,\ act'_k = ((act'_{k1}, ..., act'_{km}),$ $FH_{act_k} \rightarrow e_{k+1} \cup FH \rightarrow e_{n+1},$ $CH_{act_k} \cup CH, TH_{act_k} \cup TH)$

Fig. 8. BPELtoSTS transformation rules