An observability enhancement method of ABPEL specifications.

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

Abstract—WS-BPEL is a language which aims to conceive Business applications orchestrating a set of Web Services. Designing such applications is a difficult task since many different concepts are involved (parallelism, partner interaction, fault handling, etc.). However, some Meta-engineering techniques may help designers to improve the quality level of a specification. One of them, called Testability, helps to assess how a specification is testable by mean of different criteria or to improve a specification so that the final implementation becomes easier to test. In this paper, we focus on a well-known testability criterion called observability, which evaluates whether the set of observable events (messages with communication data) is sufficient to conclude on a verdict during the testing process. 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

I. INTRODUCTION

A part of the Internet is emerging as a Web Service based platform where more and more organizations are implementing and deploying Business processes. 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 models, like the CMMI one (Capability Maturity Model Integration [15]). In such models, the testing activity takes a solid place since it costs as much as 50 % of the total development effort. To reduce this substantial cost, testability, which is a Meta-engineering technique, helps to assess several criteria on a specification by mean of factors and to improve the characteristics of the specification or of the test suite e.g., to reduce the test costs or to detect a greater number of bugs.

This paper focuses on the ABPEL (Abstract BPEL) specification observability. This latter is a well-known testability criterion whose purpose is to evaluate if the set of observable events is sufficient to give a verdict during the testing process [6]. We propose a method and a corresponding tool which recognizes specification parts degrading observability and which transforms them to conceive a more testable specification. Among the BPEL modeling languages (ABPEL, UML, Petri nets, process algebra, abstract state machines (ASM), etc.), we base our choice on the ABPEL formalism (Abstract BPEL, and more precisely Abstract Process Profile for Observable Behavior (APPOB)), since it is used with existing testing methods [1]. This one is a partially specified process that is not intended to be executed. It serves a descriptive role where concrete operational details are hidden.

The first issue raised by the ABPEL specification observability concerns the observability definition itself, which is typically given on input/output models [8] whereas ABPEL is composed of structured and basic activities which can be nested, and gathers specific features (fault handlers, correlations, etc.). A a consequence, we propose to analyze an intermediate model, the STS (Symbolic Transition System [7]) and especially the transformation rules of ABPEL specifications into STSs. From known STS observability issues, we recover with the transformation rules several ABPEL observability degradation properties. Then, from these latter, we propose an observability enhancement method which parses ABPEL specifications, recognizes observability issues and semi-automatically removes them. The method goal is not to preserve the standard semantics of the ABPEL specification, but to model 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 II provides an overview on the BPEL language and on testability. In Section III, we describe the BPELtoSTS transformation and several BPEL observability degradation propositions. Section IV, presents the observability enhancement method. Section V concludes and gives some perspectives on BPEL testability.

II. BPEL AND TESTABILITY OVERVIEW

A. 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, flow, etc.) to set the activity structuring. Among its possibilities, it supports fault handler activities whose 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]. 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". This graphical example conceals a more complex code, as shows Figure 2, where is given the ABPEL code of the two first activities.

Fig. 1. The Loan approval BPEL process

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. Nevertheless, simulating partners may be difficult and always increases the testing cost. Some testing methods use this architecture and some tools help to implement simulated partners by generating automatically partner stubs [2].

Fig. 2. A part of the BPEL code

B. Testability

Testability is a Meta-engineering technique which evaluates, with several criteria, the system capability to reveal its faults, the accessibility of its components and its testing cost [8]. Designers evaluate the specification testability after each life cycle step, so they can detect and modify the untestable parts. They can also estimate the testing cost. Testability can be used to model and to implement more testable systems, by improving the fault detection and the fault coverage [10]. It can also be evaluated to choose one specification (the most testable) among others.

Fig. 3. The test architectures

Fig. 4. Testability in software life cycle
Testability has been studied on different models (automata, UML models, logical circuits, relational models, etc.) [8], [10], [14]. These methods evaluate testability criteria one by one 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 models, observability is defined in [8] by:

**Definition II.1** A specification is observable if for each input given, 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 [9], some "C" and "Java" code transformations are proposed to improve the test set generation by manipulating input variables. In [13], Java applications are transformed into partial oracles which help to improve the test data generation.

III. 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 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.

![Fig. 5. An ABPEL process](image)

So, our reasoning consists in considering and analyzing an intermediate model, the STS (Symbolic Transition System [7]) and the transformation rules of 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. Last but not least, some works [12], [1] also introduce BPELtoSTS transformation methods.

In the following, we describe briefly the BPELtoSTS transformation, then we focus on the ABPEL observability degradations by introducing a list of propositions.

A. BPEL to STS transformation

A BPELtoSTS transformation has been proposed in [12]. This one is mainly based on rules, whose semantics are expressed with a process algebraic style. Nevertheless, nested activities are not supported, so scope, fault handlers 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 9. Basically, when a rule is composed of several parts, the upper one expresses a condition while the lower part describes the STS transformation. For instance, the if activity is translated by two rules: the second one copies the fault handler FH (resp. the compensation handler CH and the termination handler TH) of the if activity into each nested sub-activity. The first rule creates STS transitions as illustrated in Figure 6 where τ models an internal action. The sub-activities act₁,...,actₙ depicted in blue, are translated later by the transformation algorithm and the rules.

The transformation algorithm 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).

The BPELtoSTS transformation algorithm successively transforms 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 and not yet translated, 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 previous rule set. The BPELtoSTS transformation is detailed below.

Formally, we define that a BPEL activity belongs to SA ∪ 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 \( sa = ((sₐ₁,...,sₐₙ), FH_{sₐ}, CH_{sₐ}, TH_{sₐ}) \) ∈ SA is composed of a list \( (sₐ₁,...,sₐₙ) \) of activities in \( (SA \cup BA)^n \), of a fault handler set \( FH ∈ SA \), of a compensation handler set \( CH ∈ SA \) and of a termination handler set \( TH ∈ SA \). 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 \( ba = (bₐ₁,FH_{bₐ},CH_{bₐ},TH_{bₐ}) \) ∈ BA is composed of an action \( bₐ₁ \) (invoke, receive, ...), of a fault handler \( FH ∈ BA \), a compensation one \( CH ∈ SA \) and a termination one \( TH ∈ SA \). \( FH, CH \) and \( TH \) may be empty as well.

- a fault handler activity \( fh(catch(?f₁, act₁), ..., catch (?fₙ, actₙ), catchall(act)) → e ∈ SA \) is a structured activity triggered by received WSDL faults \( ?f₁,...,?fₙ \). \( e \) is the state reached by \( fh \) once terminated. We also denote \( FH → e \) a fault handler activity set where each activity \( fh₁ → e ∈ FH → e \) reaches \( e \) once terminated. Each fault is itself composed by a variable list (faultName, faultElement, faultMessageType).
BPEL to STS transformation algorithm:

1. We construct the initial STS composed of the transition
e_1 \xrightarrow{(p_1, ..., p_n)} \text{FH}_p \rightarrow e_{n+1} \text{CH}_p \rightarrow Tp_h \rightarrow e_{n+1} + 1 \text{ where } P \text{ is the initial BPEL process. } \text{FH}_p \rightarrow e_{n+1} \text{ means that if a fault handler } fh \in \text{FH}_p \rightarrow e_{n+1} \text{ is thrown, once this one ends, the execution proceeds from the state } e_{n+1}. \text{ We set } SA = \text{P}((p_1, ..., p_n), \text{FH}_p \rightarrow e_{n+1} \text{CH}_p \rightarrow Tp_h).

2. Let the transition e_1 \xrightarrow{SA} e_{n+1} + 1 \text{ with } SA = ((s_a_1, ..., s_a_n), \text{FH} \rightarrow e_{n+1} \text{CH}, T \text{H}). \text{ We develop } SA \text{ and construct the corresponding STS transitions: }

- if } SA \text{ is a "while" (or "if", "pick") activity, we translate } SA \text{ to } SA' = ((s_{a1}', ..., s_{an}'), 0, 0, 0) \text{ with one of the rules given in figure 9 composed of the condition } \text{FH} \neq 0. \text{ 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 9 to translate } SA' \text{ into STS transitions (rule with the condition } \text{FH} = 0),

- if } SA \text{ is a structured activity different from "while" (or "if", "pick"), for each } s_{ai} \in (s_{a1}, ..., s_{an}), \text{ we use one of the rules given in figure 9,}

- if } s_{ai} \in SA, \text{ we construct the transition } e_i \xrightarrow{s_{ai}} e_{i+1} \text{ with } s_{ai} = ((s_{a1}, ..., s_{am}), \text{FH}_sa_i \rightarrow e_{i+1} \cup \text{FH} \rightarrow e_{i+1} \cup \text{CH}_sa_i \cup \text{CH}, \text{TH}_sa_i \cup \text{TH}). \text{ As previously, we spread the fault handler of the initial structured activity into each nested sub-activity. We denote the fault handler } \text{FH}_sa_i \rightarrow e_{i+1} \text{ owing to the activity } s_{ai} \text{ which ends at the state } e_{i+1}. \text{ If a fault handler of } s_{ai}, \text{ fh} \rightarrow e_{i+1} \in \text{FH}_sa_i \rightarrow e_{i+1} \text{ is caught, once executed, the process must proceed from the state } e_{i+1} \text{.}

3. While it exists a undeveloped activity } e_i \xrightarrow{SA} e_j, \text{ we use 2.

We applied the BPELtoSTS transformation on the specification example, depicted in Figure 1. We obtain the STS of Figure 7, which preserves the initial ABPEL specification semantics. 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. Each outgoing transition labeled by "?faultId" models a catch activity belonging to the previous one.

Below, we recall briefly some definitions and notation to be used throughout the paper. A complete definition can be found in [7].

An Symbolic Transition System STS is a tuple < L, l_0, V ar, v ar_0, I, S, -> >, 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 V ar, initialized by v ar_0 and of an interaction variable set I. Each transition (l_i, l_j, s, \varphi, \varrho) \rightarrow \in \text{ from the location } l_i \text{ to } l_j, \text{ labeled by the symbol } s, \text{ has a guard } \varphi \text{ which must be satisfied to fire the transition and updates the internal variables with } s \text{ once fired.}

The notion of stimulus (resp. reaction) is a pair (s, \eta), where s \in S_I \text{ is an input symbol (resp. s \in S_O \text{ is an output one) and } \eta \text{ is a mapping of the interaction variables of } s \text{ to values. For a reaction } (s, \eta), \eta \text{ is observable in the message } s. \text{ 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 } \eta.\n
We also denote out(l, \eta) the set of the first reactions reached from the location l with the values \eta, out(l_1, \eta) = \{(l_0, \eta')_1, ..., (l_m, \eta')_m | \forall 1 \leq i \leq n, 3p = (l_1, l_2, e_1, \varphi_1, \varrho_1)...(l_{i-1}, l_i, \varphi_i, \varrho_i) \text{ such as } (e_1, ..., e_{i-1}) \in S_I^{i-1} \text{ and } (\varphi_i(\eta_i) \wedge \wedge, \text{ wedge}(\varrho_i(\eta_i - 1), \eta_i)) \text{ true.}

B. Observability degradation in ABPEL specifications

According to the definition given in Section II-B, the system observability is degraded if different inputs are followed by the same outputs or by no output. For an STS stx = < L, l_0, V ar, v ar_0, I, S, -> >, 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) \rightarrow out(l, \eta) = \emptyset,

- **Observability degradation 2:** if it exists two stimuli (?, e, \eta) \neq (?, e, \eta) with (l_k, l'_k, ?, e_k, \varphi_k, \varrho_k) \rightarrow, \varphi_k(\eta_k) \text{ true and } (l_j, l'_j, ?, e_j, \varphi_j, \varrho_j) \rightarrow, \varphi_j(\eta_j) \text{ true such as out(l'_i, \eta_i) = out(l'_j, \eta_j).}

From the first STS observability degradation property, we have deduced two corresponding ABPEL ones: "the lack of a reply activity at the end of BPEL processes" and "the use of conditions with if activities which cannot be always satisfied". The BPELToSTS transformation rules, that we consider in the propositions, are those given in Figure 9.
Proposition III.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 \((l_i, l_{i+1}, \text{lo, } \varphi, g_i)\). The following proof considers "reply" activities only. The reasoning is the same with one-way "invoke" ones.

1. Consider an ABPEL specification bpel, composed only of internal actions (for instance the "assign" or "empty" activities) and not terminated by a reply activity. bpel always begins by a "receive" activity (see [3]). The translation of bpel into STS (rule 2) produces one transition \((l, l', \tau, e, g)\) 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 suppose that bpel is 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 \((l_i, l_{i+1}, \text{lo, } \varphi, g_i), (l_{i+1}, l_{i+2}, \tau, e, \varphi_{i+1}, g_{i+1})\). 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 III.2  An ABPEL specification, composed of an "invoke" (or "receive") activity followed by an "if" one if\((cond_1, act_1), \ldots, (cond_n, act_n)\) which cannot be always satisfied \((\exists \eta, \bigwedge_{1 \leq i \leq n} \text{cond}_i(\eta) = \text{false})\), is not observable.

Proof:  An "invoke" activity is transformed by the STS transitions \((l, l', \text{lo, } \varphi, g), (l', l'', \text{?e, } \varphi', g')\) (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 \(\varphi_i = \text{cond}_i\). Suppose that it exists a stimulus \(\eta\) such as \(\bigwedge_{1 \leq i \leq n} (\varphi_i(\eta) = \text{false})\).

No transition can be fired. The location \(l''\) is blocked and is said quiescent [7]. So, it exists a stimulus \((\text{?e, } \eta)\) such as \(out(l'', \eta) = \emptyset\). Consequently, both the STS and the ABPEL specification are not observable. ■

The four following propositions result from the second STS observability degradation property. With STSs, this degradation is raised if it exists two transitions \((l_i, l_{i+1}, \text{?e}_i, \varphi_i, g_i), (l_j, l_{j+1}, \text{?e}_j, \varphi_j, g_j)\) from which the same reaction \((\text{lo, } \eta)\) 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 "a block of internal activities bound by two "receive" activities".

Proposition III.3  An ABPEL specification composed of a couple of non identical "catch" ("catchall") activities \((\text{catch}_1, \text{catch}_2)\), \(\text{catch}_1 \neq \text{catch}_2\), 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 = \((\text{fault}_\text{Name}, \text{fault}_\text{Element}, \text{fault}_\text{Message} \text{Type})\). Two different "catch" activities \((\text{catch}_1, \text{catch}_2)\) are translated by two STS transitions (rule 9) \((l_i, l_{i+1}, \text{?fault}, \emptyset, g_i)\) and \((l_j, l_{j+1}, \text{?fault}, \emptyset, g_j)\). Consequently, it exists two stimuli \((\text{?fault}, \eta_1), (\text{?fault}, \eta_2)\) such as \(out(l_{i+1}, \eta_1) = out(l_{j+1}, \eta_2) = (\text{lo, } \eta)\). The STS and the ABPEL specification are not observable. ■

Proposition III.4  An ABPEL specification composed of a "catchall" activity, triggered by multiple faults and followed by an "invoke" activity whose operation call is independent of the triggered fault, is not observable.

Proof:  A "catchall" activity catchall\(\{(\text{?fault}_1, \text{?fault}_2), \ldots, \text{act}\}\) is a "catch" one, triggered by all the faults received in a fault handler, not already caught by other "catch" activities. Let \(\text{?fault}_1 = (\text{fault}_\text{Name} = n_1, \text{fault}_\text{Element} = e_1, \text{fault}_\text{Message} = m_1, \text{Type} = t_1)\) and \(\text{?fault}_2 = (\text{fault}_\text{Name} = n_2, \text{fault}_\text{Element} = e_2, \text{fault}_\text{Message} = m_2, \text{Type} = t_2)\) be two different faults. This activity is translated into the STS transitions (rule 9) \((l_i, l_{i+1}, \text{?fault}, \emptyset, g_i)\) and \((l_j, l_{j+1}, \text{?fault}, \emptyset, g_j)\), with \(\text{?fault}_1 \neq \text{?fault}_2\). If this "catchall" activity is followed by an "invoke" one, we also have the transition \((l_{i+1}, l_{i+2}, \text{lo, } \emptyset, g_{i+1})\) (rule 8) with \(\text{lo} = \text{(op, req, partner)}\).

Let \(\text{?fault}_1, \eta_1\), \(\text{?fault}_2, \eta_2\) be two stimuli and \(\eta\) a ground term such as \(\eta = g_i(\eta_1) = g_j(\eta_2)\) respectively. If \(\text{op}\) does not handle the fault, the variables \(\text{req}\) are independent of the ones in \(\{\text{gi} \land \text{gj}\}\). The mapping of \(\text{req}\) to ground terms equals to \(\eta\). Consequently, it exists two stimuli \((\text{?fault}_1, \eta_1), (\text{?fault}_2, \eta_2)\) such as \(out(l_{i+1}, \eta_1) = out(l_{j+1}, \eta_2) = (\text{lo, } \eta)\). This corresponds to the second degradation property. The STS and the ABPEL specification are not observable. ■

Proposition III.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 \((l_i, l_{i+1}, \text{?e}_i, \varphi_i, g_i), (l_j, l_{j+1}, \text{?e}_j, \varphi_j, g_j)\), with \(\text{?e}_i \neq \text{?e}_j\). The invoke activities give the two transitions \((l_i, l_{i+1}, \text{lo, } \varphi_{i+1}, g_{i+1}), (l_j, l_{j+1}, \text{lo, } \varphi_{j+1}, g_{j+1})\) (rule 6). We obtain the same case as the one described in Proposition III.3. Thus, the STS and the ABPEL specification are not observable. ■

Proposition III.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 \((l_1, l_2, \text{?e}_1, \varphi_1, g_1), (l_2, l_3, \text{?e}_2, \varphi_2, g_2), \ldots, (l_{n-1}, l_n, \text{?e}_{n-1}, \varphi_{n-1}, g_{n-1}), (l_n, l_{n+1}, \text{?e}_n, \varphi_n, g_n)\) with \(\{e_2, \ldots, e_{n-1}\} \notin \text{out}_2^{n-2}\) (no output message). If this activity block is not followed by an "invoke" (or "reply" or "throw") one, \(\forall \eta_1, \eta_2\), such as \(\varphi(\eta_1)\text{true}\) and \(\varphi(\eta_2)\text{true}\), \(out(l_2, \eta_1) = out(l_{n+1}, \eta_2) = \emptyset\). Otherwise,
the activity block is followed by an "invoke" (or "reply" or "throw") activity which produces an STS transition \((l_1, l_2, i_0, \varphi_1, g_1)\) such as \(\exists \eta_j \varphi_j(\eta_j)\) true, \(\forall (\eta_j, \eta_j)\) out\((l_1, \eta_j) = \text{out}(l_{n+1}, \eta_j) = (l_2, \eta_j)\).

The second STS degradation property is satisfied, hence both the STS and the ABPEL specification are not observable.

IV. ABPEL SPECIFICATION OBSERVABILITY ENHANCEMENT

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

The ABPEL specification modifications 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:

- **"reply" activity addition:** we check that each branch of \(bpel\) ends with a "reply" (invoke-only) activity (Proposition III.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\(_n\)," 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 III.2, an "if" activity \(if((\text{cond}_1, \text{act}_1), \ldots, (\text{cond}_n, \text{act}_n))\), which cannot be always satisfied (\(\exists \eta \bigvee_{1 \leq i \leq n} \text{cond}_i(\eta)\) 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 III.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: badConditions\(_k\)" (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: badConditions\(_k\)" to the client side.

- **"invoke" activity distinction:** Propositions III.3-III.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 III.3, the same reaction is observed on account of two "invoke" activities using the same operation after two different "catch" activities \(\text{catch}_1(? \text{fault}_1, \text{act}_1), \text{catch}_1(? \text{fault}_2, \text{act}_2)\). Here, the two "invoke" activities can be distinguished by the fault name. In this case, we add a variable \(\text{fault}\) to \(op\) equals to the fault name. Propositions III.4-III.5 are based on the same idea. We only give, in Algorithm 3, the methodology dedicated to Proposition III.3. The other ones require only minor modifications. In Algorithm 3, if two "catch" activities \(\text{catch}_1(? \text{fault}_1, \text{act}_1), \text{catch}_1(? \text{fault}_2, \text{act}_2)\) 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 \([11], [5]\) (lines 4-5). If at least one value satisfies both "invoke" activities (Proposition III.3 verified) (line 6), we modify them by adding a new parameter equals to \(?\text{fault}_1, (?\text{fault}_2\) respectively) (lines 7-8). According to Proposition III.3, \(?\text{fault}_1\) and \(?\text{fault}_2\) 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 \([11], [5]\) which work as external servers and manage most of the simple types, "string" included.

- **Asynchronous call modification:** asynchronous calls correspond to partner invocations whose 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 \(\text{invoke}_j(op_j, \text{req}_j, \emptyset, \text{partner}_j, \text{corr}_j)\) and \(\text{invoke}_j(op_j, \text{req}_j, \emptyset, \text{partner}_j, \text{corr}_j)\), followed later by two "receive" ones \(\text{rcv}_j(op_j, \text{resp}_j, \text{partner}_j, \text{corr}_j)\), \(\text{rcv}_j(op_j, \text{resp}_j, \text{partner}_j, \text{corr}_j)\), linked by the same correlation sets. When the two "receive" activities are successive, the observability is degraded (Proposition III.6). We propose here to insert \(\text{invoke}_j\) between the two "receive" activities to obtain observable reactions after \(\text{rcv}_j\) (Proposition III.6 no more satisfied). However, we cannot only move the \(\text{invoke}_j\) activity in the ABPEL code since this one depends on the variables \(\text{req}_j\) which may be updated in the process. Algorithm 4 replaces \(\text{invoke}_j\) by an "assign" activity which copies the \(\text{req}_j\) variables into new ones, denoted \(\text{req}_2\) (line 3). These variables are only used with the \(\text{invoke}_j\) activity which is inserted before \(\text{rcv}_j\) (line 4).

We applied this enhancement method on the ABPEL specification illustrated in Figures 1.7. For sake of simplicity, we only illustrate the modification on the STS specification since the XML based ABPEL specification is less readable. The initial specification is composed of six observability issues: (1) no reaction is observed from the location A2 with the stimulus \((\tau, \text{amount} = 1000)\). (2,3) both the stimuli \((?\text{risk}, \text{risk} = \text{low})\) and \(?\text{approval2}, \text{app2} = \text{yes}\) are followed by the same reaction \((?\text{approval}, \text{app2} = \text{yes})\). (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 "?\text{faultId1}\" give the same observation. Once the enhancement method applied, we obtain a new ABPEL specification and its corresponding STS, illustrated in Figure 8. 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 A15)
Algorithm 2: "if" condition addition

input : ABPEL specification bpel
1 foreach "invoke" activity I followed by an "if" activity if(s,(cond1, act1),...,(condn, actn)) do
2 Compute V such as $\forall v \in V \left( \bigwedge_{1 \leq i \leq n} \neg (\text{true}) \right)$ false with constraint solvers;
3 if V $\neq \emptyset$ then
4 Add "<else> <throw faultName= "FLT : badConditionk"/></else>";
5 Add this "catch" activity in the faulthandler of the "scope" activity containing if(s):
6 where REPLY=reply(resp, partner, op) with partner=client, op= client operation used for calling the ABPEL process, resp="FLT : badConditionk";

A14) have been added, two identical "catch(?fault1)" activities (location A10) have been distinguished with "?fault1", ?faultid2, and the "if" condition has been completed (location A2 with amount = 1000).

V. Conclusion

This paper proposes an observability enhancement method of ABPEL specifications, which detects testability degradations and which modifies ABPEL specifications and guides designers to improve them. We have proposed some ABPEL observability degradations, which detects testability degradations and guides designers to improve them. It has been shown that the 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 addition 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. It exists other cases where two stimuli produce the same reaction, and so other algorithms can be proposed. Moreover, many other quality criteria could be also studied e.g., the controllability which evaluates whether an implementation can be controlled to execute test cases, the execution time which evaluates the testing cost according to the minimal and maximal execution times, or the accessibility of ABPEL parts.

This observability enhancement method does not guarantee to preserve the standard semantics of an ABPEL specification but to conceive 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


<table>
<thead>
<tr>
<th>BPEL</th>
<th>STS transformation rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign(var_update) or empty</td>
<td>[ e_i \xrightarrow{\text{#}, \text{#}} e_{i+1}, D = \text{var_update} ]</td>
</tr>
<tr>
<td>receive(op, resp, partner, corr, ( FH, CH, TH ))</td>
<td>[ e_i \xrightarrow{(\text{op,resp,partner,corr}), [e_i = \text{corr}], \text{g(resp)}} e_{i+1} \wedge e_i \xrightarrow{f_{\text{b}}} e_{i+1} ]</td>
</tr>
<tr>
<td>reply(op, req, partner, corr, ( FH, CH, TH ))</td>
<td>[ e_i \xrightarrow{[\langle \text{op,req,partner,corr} \rangle, \text{#}(e_i = \text{corr})], \delta} e_{i+1} ]</td>
</tr>
<tr>
<td>throw(!f, ( FH, CH, TH ))</td>
<td>[ e_i \xrightarrow{[\langle \text{op,req,partner,corr} \rangle, \text{#}(e_i = \text{corr})], \delta} e_{i+1} ]</td>
</tr>
<tr>
<td>if((cond, act), ( \ldots )), ( cond_n, act_n ), ( CH, TH ), or switch</td>
<td>[ e_i \xrightarrow{[\langle \text{cond}, \text{#} \rangle, \langle \text{act} \rangle]} e_{i+1} ]</td>
</tr>
<tr>
<td>pick((mess, act), ( \ldots )), enalert(l, act_n+1), ( FH, CH, TH ) with messk = (op_k, resp_k, partner_k, corr_k)</td>
<td>[ e_i \xrightarrow{[\langle \text{mess}, \text{#} \rangle, \text{g} \langle \text{resp} \rangle, \text{act} \rangle]} e_{i+1} ] \quad \text{with act} = ((act_1, \ldots, act_{k_m}), FH_{act_k} \cup FH, CH_{act_k} \cup CH, \text{act}_k \cup \text{TH})</td>
</tr>
<tr>
<td>while(cond, act, ( FH, CH, TH ))</td>
<td>[ e_i \xrightarrow{[\langle \text{cond} \rangle, \text{#} \rangle]} e_{i+1} ]</td>
</tr>
<tr>
<td>invoke(op, req, resp, partner, corr, ( FH, CH, TH ))</td>
<td>[ e_i \xrightarrow{[\langle \text{op,req,resp,partner} \rangle, \text{#}(e_i = \text{corr})], \delta}] e_{i+1} ]</td>
</tr>
<tr>
<td>fh(catch1(?f1(n1, e1, m1), ( act_1 )), ( \ldots )), catchn(?fn((m_n, e_n, m_n)), ( act_n )), catchall(?f(n_{n+1}, e_{n+1}, ( m_{n+1} ), ( act )))</td>
<td>[ e_i \xrightarrow{[\langle \text{fh} \rangle, \text{#}]} e_{i+1} ]</td>
</tr>
<tr>
<td>scope((act_1, \ldots, act_n), ( FH, CH, TH )) or process or sequence</td>
<td>[ e_i \xrightarrow{[\langle \text{fh} \rangle, \text{#}]} e_{i+1} ]</td>
</tr>
</tbody>
</table>

Fig. 9. BPELtoSTS transformation rules


