RAPPORT DE RECHERCHE

L R I

A FORMAL FRAMEWORK FOR SERVICE ORCHESTRATION TESTING BASED ON SYMBOLIC TRANSITION SYSTEMS

BENTAKOUK L / POIZAT P / ZAIDI F

Unité Mixte de Recherche 8623
CNRS-Université Paris Sud – LRI

04/2011

Rapport de Recherche N° 1542

CNRS – Université de Paris Sud
Centre d'Orsay
LABORATOIRE DE RECHERCHE EN INFORMATIQUE
Bâtiment 490
91405 ORSAY Cedex (France)

A Formal Framework for Service Orchestration Testing Based on Symbolic Transition Systems*

Lina Bentakouk[1], Pascal Poizat[1,2], and Fatiha Zaïdi[1]

[1] LRI ; Univ. Paris-Sud, CNRS
[2] Univ. Évry Val d'Essonne
{lina.bentakouk,pascal.poizat,fatiha.zaidi}@lri.fr

Abstract. The pre-eminent role played by software composition, and more particularly service composition, in modern software development, together with the complexity of workflow languages such as WS-BPEL have made composite service testing a topical issue. In this article we contribute to this issue with an automatic testing approach for WS-BPEL orchestrations. Compared to related work, we support WS-BPEL data computations and exchanges, while overcoming the consequential state explosion problem. This is achieved through the use of symbolic transition system models and their symbolic execution. Throughout the article, we illustrate our approach on a realistic medium-size example.
keywords: service composition, orchestration, formal testing, test-case generation, WS-BPEL, transition systems, symbolic execution.

1 Introduction

Service composition, and more specifically orchestration, has emerged as a cornerstone to develop added-value distributed applications out of reusable and loosely coupled software pieces. The WS-BPEL language [1], or BPEL for short, has become the de-facto standard for Web service orchestration and is gaining industry-wide acceptance and usage. This makes BPEL orchestration correctness a topical issue, all the more because of BPEL complexity. This has been partly addressed by automatic service composition or adaptation processes [2, 3]. Still, these usually assume that services convey semantic annotations, which is, currently, seldom the case. Service orchestrations are therefore developed in a more usual way, *i.e.*, from specifications which are thereafter implemented by service architects. Numerous model-based verification approaches have been proposed for BPEL, *e.g.*, translating BPEL to automata, Petri nets or process algebras [4]. These approaches are especially valuable to check if an orchestration specification is correct. Still, as far as the correctness of an implementation

* Version 1 – May, 15th, 2009. Part of this work (STS models, symbolic execution, test case generation and realisation based on coverage criteria) has been published in the proceedings of TESTCOM/FATES'09. This work is supported by the projects "PERvasive Service cOmposition" (PERSO) and "WEB service MOdelling and Validation" (WEBMOV) of the French National Agency for Research.

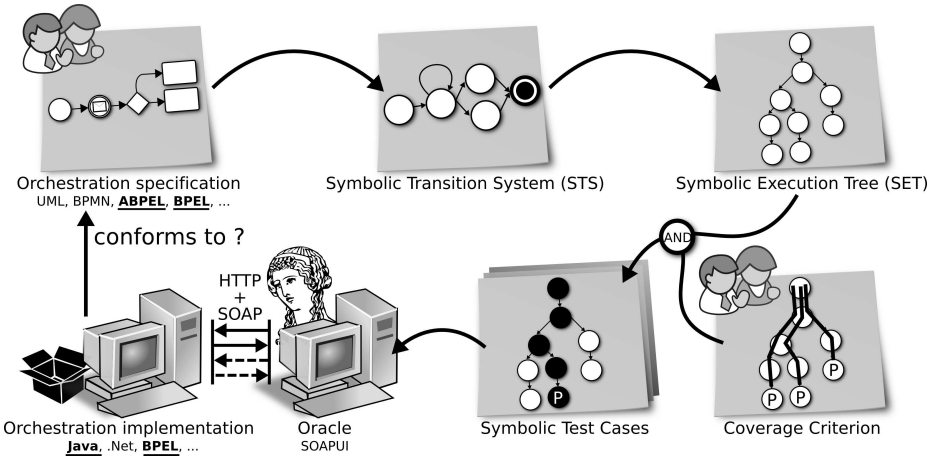

Fig. 1. Overview of the proposed framework

wrt. a specification is concerned, these approaches fall short as, *e.g.*, one may expect service providers to publicise (Abstract) BPEL descriptions of *what* their services do, but not *how* they do it. Here, testing comes as a solution to ensure (i) that some sub-service used in an orchestration really conforms to its publicised behavioural interface, and (ii) that the service orchestration itself conforms to the behavioural interface to be publicised after its deployment.

In this paper, we propose a formal framework for the testing of service orchestrations (Fig. 1). The orchestration specification is first translated into a formal model, namely a Symbolic Transition System (STS) [5]. In a second step, a Symbolic Execution Tree (SET) is computed from this STS. It represents (a finite subset of) the STS execution semantics, while avoiding the usual state-explosion problem in presence of unbounded data types, as used in full-fledged BPEL. Given a coverage criterion, we generate from the SET a set of execution paths which are finally run by a test oracle against the orchestration implementation.

With reference to related work, our contributions are manifold. A first contribution is the support for the rich XML-based data types available in BPEL. This is achieved first by relying on a symbolic model rather than on labelled transition systems (LTS) usually used –either directly or indirectly from process algebraic or Petri net descriptions– as BPEL models. LTS are known to cause over-approximation or unimplementable test cases (when data are simply abstracted away), and state explosion problems (when message parameters or variables are flattened wrt. their infinite domains). Both are avoided using STS. Symbolic execution [6] also takes part in avoiding state explosion by representing message parameters and variables using symbolic values instead of concrete data. A second contribution is the possibility to take different orchestration specification languages (UML, BPMN, (A)BPEL) into account. This is achieved thanks

to the STS model which has already proven to be valuable, *e.g.*, for UML [7], and is here used for BPEL.

Finally, we propose a comprehensive language-to-language model-based testing framework, while most model-based (verification or testing) approaches targeted at orchestration languages either ignore the retrieval of the formal model from the orchestration specification, ignore or over-simplify the rich XML-based data types of services, or do not tackle the execution of test cases against a running service implementation. As a consequence, we have applied our framework to two realistic medium-size case studies, including the extended version, presented herein, of the loan approval service [1].

The remaining of the paper is organised as follows. The following Section introduces our case study. The orchestration and STS models, together with the transformation from BPEL to STS, are presented in Section 3. The principles of symbolic execution and the computation of a finite SET from an STS are described in Section 4. Section 5 addresses the retrieval of symbolic test cases from a SET, our online testing algorithm, and tool support. Finally Section 6 discusses related work and we end in Section 7 with conclusions and perspectives.

2 Running Example

In this Section we introduce our xLoan case study. It is an extension of the well-known loan approval example presented in the BPEL standard [1], which usually serves for demonstration purposes in articles on BPEL verification. Our extensions are targeted at demonstrating our support for BPEL important features: complex data types, complex service conversations including message correlation, loops and alarms. Hence, more complex and realistic data types are used, to model user information, loan requests and loan proposals. The specified sub-services respectively deal with loan approval (BankService) and black listing (BlackListingService), with users not being blacklisted asking for low loans ($\leq 10,000$) getting loan proposals without requiring further approval. As-is, these services resemble the ones proposed in [1]. Yet, once a loan is accepted, proposals may be sent to the requester. Further communication then takes place, letting the requester select one proposal or cancel, which is then transmitted to BankService. If the selected offer code is not correct the requester is issued an error message and may try again (select or cancel). Timeouts are also modelled, and the bank is informed about cancelling if the requester does not reply in a given amount of time (2 hours).

There is no official graphical notation neither for orchestration architectures (WSDL interfaces and partner links), nor for the imported data types (XML Schema files) or the service conversation (BPEL <process> definition). For the former ones (Fig. 2) we use the UML notation that we extend with specific stereotypes in order to represent message types, correlations and properties (see Sect. 3 for their semantics). Moreover, XML namespaces are represented with packages. Additionally, there is currently an important research effort on relating the Business Process Modelling Notation (BPMN) with Abstract BPEL or

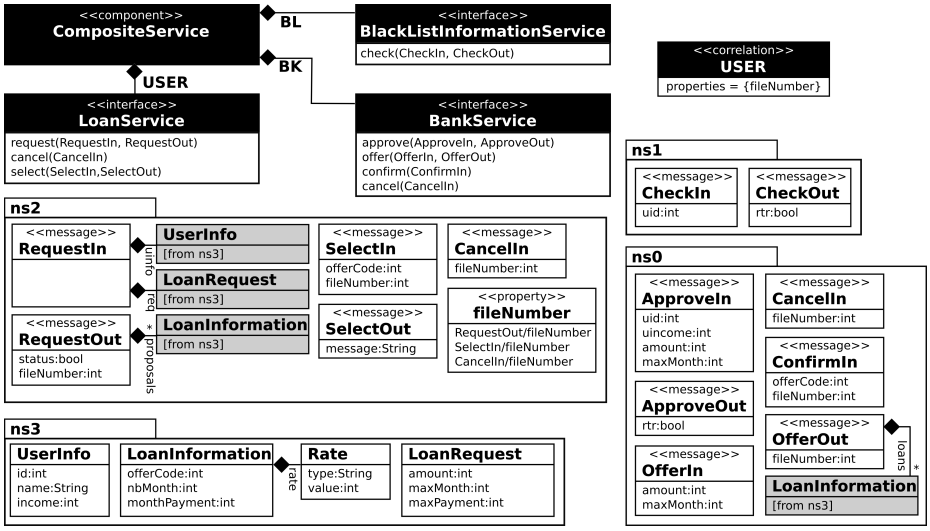

Fig. 2. xLoan Example – Data and Service Architecture

BPEL code [8]. Therefore, concerning the graphical presentation of service conversations, we take inspiration from BPMN, while adding our own annotations supporting relation with BPEL. Communication activities are represented with the concerned partnerlink (USER for the user of the orchestration, BK or BL for the two sub-services), operation, input/output variables, and, when it applies, information about message correlation.

Figure 3 presents the orchestration specification. The overall process is presented in Figure 3, upper part, while its lower part concerns the (potentially looping) subprocess, GL&S (*Get Loan and Select*), for loan proposal selection.

3 From BPEL to STS

In this Section we first present our service model. Then, we present the rules which are used to obtain a service model from a BPEL description.

3.1 Service Model

Services may expose information at different interface description levels. The basic level is the *signature level* where a service describes the set of operations it provides. Complex services, including state-full ones, additionally provide a *behavioural description* (conversation) of the way its operations should be called. In this work we focus on these two levels, which are the ones widely accepted and used, with respectively the WSDL and (A)BPEL languages.

Signatures. The *signature* of a service is described using a combination of XML schema (exchanged data structures) and WSDL (operations and messages). We model it as a tuple $\Sigma = (\mathcal{D}, \mathcal{O}, \mathcal{P}, in, out, err, \pi, \downarrow)$.

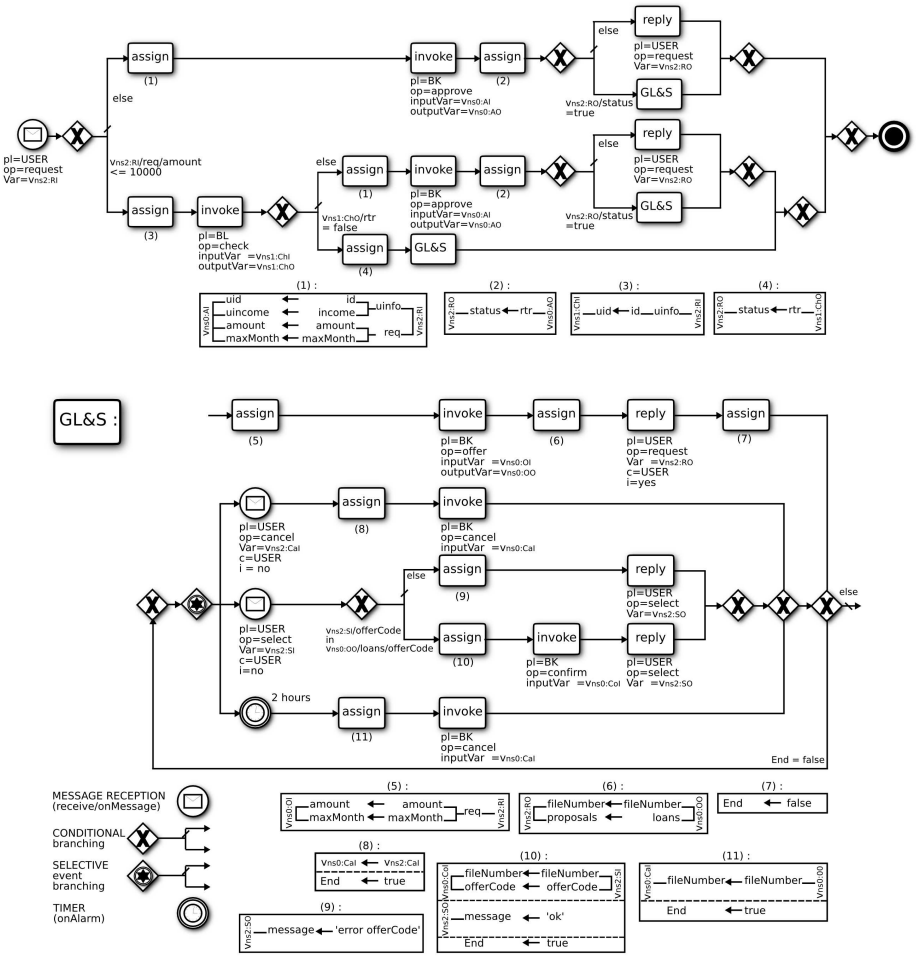

Fig. 3. xLoan Example – Orchestration Specification

$\mathcal{D}$ is a set of domains. $dom(x)$ denotes the domain of x. $\mathcal{O}$ is a set of (provided) operations. Operations may be either one-way or two-way. $in, out, err : \mathcal{O} \rightarrow \mathcal{D} \cup \{\perp\}$ denote respectively the input, output, or fault message of an operation ($\perp$ when undefined, *e.g.*, $out(o) = err(o) = \perp$ for any one-way operation o). $\mathcal{P}$ is a set of property names. Properties, together with property aliases and correlation sets, are important BPEL features that support the definition of sessions (see [1] and below, *Message Correlation*). $\downarrow$ is used to define property aliases for messages: for a message type m, $m \downarrow_p$ denotes the part in m messages that corresponds to property p. Finally, π is used to specify what two-way op-

erations in sub-services are supposed to do: $\pi(o)$, $o \in \mathcal{O}$, is a boolean formula relating o inputs and outputs.

Partnership. An orchestration is built around a partnership, *i.e.*, a set of partner signatures, corresponding to required operations, and a set of signatures for the orchestration itself, corresponding to provided operations. In the sequel we suppose, without loss of generality, that an orchestration has only one of these, named USER. A *partnership* ρ is an ID-indexed set of signatures $\{\Sigma_i\}_{i \in ID}$, where ID is a set of names ($\mathsf{USER} \in ID$). Two domains with the same name, including the namespace part, in two signatures correspond to the same thing. More specifically, we suppose they have the same formal semantics.

Events. The semantics of a service conversation depends on message-based communication, which is modelled using *events*. An input event, $pl.o?x$, with $o \in \Sigma_{pl}$ and message variable x such that $dom(x) = in(o)$, corresponds to the reception of the input message of operation o from partner link pl. Accordingly, we define output events $pl.o!x$ ($dom(x) = out(o)$). Service calls may also yield errors (message faults). This is modelled with fault output events, $pl.o!!x$, and fault input events, $pl.o??x$ ($dom(x) = err(o)$). We omit BPEL port types here for simplicity reasons (full event prefixes would be, *e.g.*, $pl.pt.o?x$ for the first example above with input on port type pt). $Ev^?$ (resp. $Ev^!$, $Ev^{??}$, and $Ev^{!!}$) is the set of input events (resp. output events, fault input events, and fault output events). Ex is the set of internal fault events, that correspond to faults possibly raised internally (not in messages) by the orchestration process. We also introduce specific events: τ denotes non-observable internal computations or conditions, χ denotes time passing (time constraints in services are generally soft, hence discrete time is a valid abstraction), and $\surd$ denotes the termination of a conversation (end of a session). We define $Ev = Ev^? \cup Ev^! \cup Ev^{??} \cup Ev^{!!} \cup Ex \cup \{\tau, \chi, \surd\}$. We also define hd as $\forall * \in \{?, ??, !, !!\}$, $hd(pl.o * x) = pl.o$, and $hd(e) = e$ for any other e in Ev.

Orchestration. Different models have been proposed to support behavioural service discovery, verification, testing, composition or adaptation [4, 9–11]. They mainly differ in their formal grounding (Petri nets, transition systems, or process algebra), and the subset of service languages being supported. Here, we base on [12] due to its good coverage of the main BPEL language constructs. Moreover, its process algebraic style for transformation rules enables a concise yet precise and operational model, which is, through extension, amenable to symbolic execution. In this objective, we extend the [12] formalising to support a wider subset of BPEL (see below discussion *On BPEL coverage*), including data in computation and messages. As a consequence, our behavioural model grounds on (discrete timed) Symbolic Transition Systems (STS) in place of the discrete timed Labelled Transition Systems (dtLTS) presented in [12].

A *Symbolic Transition System* (STS), is a tuple $(\mathcal{D}, \mathcal{V}, S, s_0, T)$ where $\mathcal{D}$ is a set of domains (as in signatures), $\mathcal{V}$ is a set of variables with domain in $\mathcal{D}$, S is a non empty set of states, $s_0 \in S$ is the initial state, and T is a (potentially nondeterministic) transition relation, $T \subseteq S \times \mathcal{T}_{\mathcal{D}^{\mathrm{Bool}},\mathcal{V}} \times Ev \times 2^{Act} \times S$, with $\mathcal{T}_{\mathcal{D}^{\mathrm{Bool}},V}$ denoting boolean terms, Ev a set of events and Act a set of actions (of the form

$v := t$ where $v \in \mathcal{V}$ is a variable and $t \in \mathcal{T}_{\mathcal{D},\mathcal{V}}$ a term). The transition system is called symbolic as the guards, events, and actions may contain variables. An element (s, g, e, A, s') of T is denoted $s \xrightarrow{[g] \quad e \quad / \ A} s'$. When there is no guard (*i.e.*, it is true) it is omitted. The same yields for the actions. We impose that variables used in the STS transitions are defined in $\mathcal{V}$. STS have been introduced under different forms (and names) in the literature [5], to associate a behaviour with a specification of data types that is used to evaluate guards, actions and sent values. This role is played here by $\mathcal{D}$ which is a superset of all partners' domains. Consistency of the union is ensured by the above-mentioned restriction on domains sharing names.

An *orchestration* $\mathcal{O}rch$ is a couple $(\rho, \mathcal{B})$ where ρ is a partnership and $\mathcal{B}$ is an STS. We impose that $\mathcal{B}$ is correct wrt. ρ, *i.e.*, its set of events correspond to partner links and operations defined in ρ, which can be syntactically checked.

3.2 Transformation Rules

We support the main BPEL activities [1], abstracted from their concrete syntax as follows:

P,Q,R ::= basic | struct
basic ::= receive(pl,op,var) | reply(pl,op,var) | invoke(pl,op,inputvar[,outputvar])
| time | throw e | x[/path]:=Expr | empty | 0
struct ::= P;Q | if c then P[else Q] | while c {P} | repeat {P} until c
flow({P_i}) | scope(P,EH^d) | pick(EH^d)
EH^d ::= [{((pl_i,op_i,var_i),P_i)},(d,Q),{(e_j,R_j)}]

Communication activities (receive, reply, and invoke) specify the communications between service partners. *Time activities* (timeouts or watchdogs) can be reduced to a time passing activity, time, and the use of scopes. Faults are raised using throw. *Assignment activities* (:=), support data and computation, and operate between an XPath [13] expression (a variable and an optional path over it) and any expression (including XPath). empty and 0 denote respectively an empty and a terminated process. On top of these *basic activities*, BPEL defines workflow-based *structuring activities*: sequence (;), conditional (if), loops (while and until), parallelism (flow), scopes (scope) and multiple event processing (pick).

scope(P,EH^d) encapsulates an activity P with an event handler EH^d. An event handler is made up of sets of events, one for each type of event –received messages, timeouts related to a duration (d), or faults (e_j)[3] – and associated activities. scope behaves as P if none of the events happens and as a given sub-activity (some P_i, Q, or R_j) if the corresponding event happens. pick is treated as a scope[4]

The transformation rules (BPEL to STS) are presented in Table 1. Anonymous variables (va_i) are introduced to follow BPEL communication semantics.

[3] These correspond respectively in BPEL to onEvent and onAlarm in event handlers, and to fault handlers.

[4] We abstract from concrete syntax differences, *e.g.*, between onMessage in a pick and onEvent in scopes.

BPEL	**STS**
empty	$\mathsf{empty} \xrightarrow{\surd} \mathsf{0}$
time	$\mathsf{p} \xrightarrow{\chi} \mathsf{p}$ with $\mathsf{p} \in \{\mathsf{time}, \mathsf{rec(pl,o,v_{in})}, \mathsf{send(pl,o,v_{out})}\}$
assign$^+$	$\mathsf{p1{:=}p2} \xrightarrow{\tau \ / \ p1:=p2} \mathsf{empty}$
throw	$\forall e \in Ex$ $\mathsf{throw\ e} \xrightarrow{e} \mathsf{0}$
rec$^+$	$\mathsf{rec(pl,o,v_{in})} \xrightarrow{pl.o?va_m \ / \ v_{in}:=va_m} \mathsf{empty}$ with $\exists o \in \mathcal{O}(\Sigma_{pl})$, $in(o) = m$
send$^+$	$\mathsf{send(pl,o,v_{out})} \xrightarrow{\tau \ / \ va_m:=v_{out}} \_ \xrightarrow{pl.o!va_m} \mathsf{empty}$ with $\exists o \in \mathcal{O}(\Sigma_{pl})$, $out(o) = m$
receive*	$\mathsf{receive(pl,o,v_{in})} = \mathsf{rec(pl,o,v_{in})}$
reply*	$\mathsf{reply(pl,o,v_{out})} = \mathsf{send(pl,o,v_{out})}$
invoke$^+$	$\mathsf{invoke(pl,o,v_{in})} = \mathsf{send(pl,o,v_{in})}$ $\quad$ $\mathsf{invoke(pl,o,v_{in},v_{out})} = \mathsf{send(pl,o,v_{in});rec(pl,o,v_{out})}$
sequence*	$\forall a \in Ev \backslash \{\surd\}$, $\dfrac{P \xrightarrow{[g] \ a \ / \ A} P'}{P;Q \xrightarrow{[g] \ a \ / \ A} P';Q}$ $\quad$ $\forall a \in Ev$, $\dfrac{P \xrightarrow{\surd} P' \wedge Q \xrightarrow{[g] \ a \ / \ A} Q'}{P;Q \xrightarrow{[g] \ a \ / \ A} Q'}$
if*	$\mathsf{if\ c\ then\ P\ else\ Q} \xrightarrow{[c] \ \tau} \mathsf{P}$ $\quad$ $\mathsf{if\ c\ then\ P\ else\ Q} \xrightarrow{[\neg c] \ \tau} \mathsf{Q}$
while*	$\mathsf{while\ c\ \{P\}} \xrightarrow{[c] \ \tau} \mathsf{P;while\ c\ \{P\}}$ $\quad$ $\mathsf{while\ c\ \{P\}} \xrightarrow{[\neg c] \ \tau} \mathsf{empty}$
until$^+$	$\mathsf{repeat\ \{P\}\ until\ c = P;while\ c\ \{P\}}$
flow$^+$	
flow internals	$\forall a \in Ev \backslash \{\chi, \surd\}$, $\dfrac{\exists j \in I, P_j \xrightarrow{[g] \ a \ / \ A} P'_j}{\mathsf{flow}(\{\mathsf{P_{i,i\in I}}\}) \xrightarrow{[g] \ a \ / \ A} \mathsf{flow}(\{\mathsf{P_{i,i\in I\backslash\{j\}}}\} \cup \{\mathsf{P'_j}\})}$
flow termination /time passing	$\dfrac{\forall i \in I, P_i \xrightarrow{\surd} P'_i}{\mathsf{flow}(\{\mathsf{P_{i,i\in I}}\}) \xrightarrow{\surd} \mathsf{empty}}$ $\quad$ $\dfrac{\exists J \neq \emptyset, J \subseteq I, \forall i \in J, P_i \xrightarrow{\chi} P'_i \wedge \forall i \in I \backslash J, P_i \xrightarrow{\surd} P'i}{\mathsf{flow}(\{\mathsf{P_{i,i\in I}}\}) \xrightarrow{\chi} \mathsf{flow}(\{\mathsf{P_{i,i\in I\backslash J}}\} \cup \{\mathsf{P'_{i,i\in J}}\})}$
scope*	let $EH^d = [\{((\mathsf{pl_i}, \mathsf{o_i}, \mathsf{v_i}), \mathsf{P_i})_{i\in I}\}, (\mathsf{d}, \mathsf{Q}), \{(\mathsf{e_j}, \mathsf{R_j})_{j\in J}\}]$, $O_I = \{(pl_i, o_i, v_i)_{i\in I}\}$, $\overline{O_I} = \{pl_i.o_i \mid (pl_i, o_i, v_i) \in O_I\}$, $E_J = \{e_{j,j\in J}\}$ in:
event handler	$\forall (pl_i, o_i, v_i) \in O_I$, $\dfrac{\forall a \in Ex \cup \{\chi, \surd\}, \neg(P \xrightarrow{a})}{\mathsf{scope(P,EH^d)} \xrightarrow{pl_i.o_i?va_m \ / \ v_i:=va_m} P_i}$ with $\exists o_i \in \mathcal{O}(\Sigma_{pl_i})$, $in(o_i) = m$
time passing /alarm	$\forall d > 1$, $\dfrac{P \xrightarrow{\chi} P' \wedge \forall a \in Ex \cup \{\tau, \surd\}, \neg(P \xrightarrow{a})}{\mathsf{scope(P,EH^d)} \xrightarrow{\chi} \mathsf{scope(P,EH^{d-1})}}$ $\quad$ $\dfrac{P \xrightarrow{\chi} P' \wedge \forall a \in Ex \cup \{\tau, \surd\}, \neg(P \xrightarrow{a})}{\mathsf{scope(P,EH^1)} \xrightarrow{\chi} \mathsf{Q}}$
fault handler /unsupported fault	$\forall e_j \in E_J$, $\dfrac{P \xrightarrow{e_j}}{\mathsf{scope(P,EH^d)} \xrightarrow{\tau} \mathsf{R_j}}$ $\quad$ $\forall e \in Ex \backslash E_J$, $\dfrac{P \xrightarrow{e}}{\mathsf{scope(P,EH^d)} \xrightarrow{e} \mathsf{0}}$
scope termination /scope internals	$\dfrac{P \xrightarrow{\surd}}{\mathsf{scope(P,EH^d)} \xrightarrow{\surd} \mathsf{0}}$ $\quad$ $\forall a \in Ev$, $\dfrac{hd(a) \notin (\{\chi, \surd\} \cup Ex \cup \overline{O_I}) \wedge P \xrightarrow{[g] \ a \ / \ A} P'}{\mathsf{scope(P,EH^d)} \xrightarrow{[g] \ a \ / \ A} \mathsf{scope(P',EH^d)}}$
pick	$\mathsf{pick(E) = scope(time,E)}$

Table 1. BPEL to STS rules, * /$^+$:extended/added wrt. [12] (BPEL to dtLTS).

Message correlation. Correlation is supported as an extension of Table 1 rules. First we modify the orchestration model to be $(\rho, \mathcal{B}, \mathcal{C})$ where $\mathcal{C}$ are correlation

sets, *i.e.*, a name and a set of associated properties denoted with *props*. These are used to correlate messages in-between service instances [1]. Sometimes a single property is used (*e.g.*, an identifier), but more generally this is a set (*e.g.*, name and surname). A correlation value is a value of a structured domain with items corresponding to the properties. For each correlation set c in $\mathcal{C}$, we have two variables, vcs_c and $vcs_c init$, in $\mathcal{V}(\mathcal{B})$. The communication activities parameter lists (pl,o,v) (in receive, reply, invoke, pick onMessage and scope onEvent) are extended to (pl,o,v,i,c) where c is the correlation name and i corresponds to correlation initiation (yes, no, or join). The STS semantics of communication activities is then extended in the following way.

On the initial transition we add action $vcs_c init := false$. We define $ccc(c, va_m) = (\bigwedge_{p \in props(c)} va_m/[m \downarrow p] = vcs_c/p)$ (correlation consistency constraint of c satisfied), $G_c^{\text{yes}}(va_m) = (vcs_c init = true)$, $G_c^{\text{join}}(va_m) = (vcs_c init = true \wedge \neg ccc(c, va_m))$, $G_c^{\text{no}}(va_m) = (vcs_c init = false \vee \neg ccc(c, va_m))$, $A_c^{\text{yes}}(va_m) = A_c^{\text{join}}(va_m) = \{vcs_c/p := va_m/[m \downarrow p]\}_{p \in props(c)} \cup \{vcs_c init := true\}$, and $A_c^{\text{no}}(va_m) = \emptyset$.

We replace each transition $s \xrightarrow{pl.o*va_m \quad / \ A} s'$ $(* \in \{?, ??\})$ by:

$s \xrightarrow{pl.o*va_m \quad / \ A} s''$ (reception),

$s'' \xrightarrow{[\neg G_c^{\text{i}}(va_m)] \quad \tau \quad / \ A_c^{\text{i}}(va_m)} s'$ (correlation ok), and

$s'' \xrightarrow{[G_c^{\text{i}}(va_m)] \quad \tau}$ throw bpel : correlationViolation (correlation error)

We also replace each transitions $s \xrightarrow{\tau \quad / \ A} s' \xrightarrow{pl.o*va_m \quad / \ A} s''$ $(* \in \{!, !!\})$ by:

$s \xrightarrow{\tau \quad / \ A} s'$ (assignments),

$s' \xrightarrow{[\neg G_c^{\text{i}}(va_m)] \quad pl.o*va_m \quad / \ A \cup A_c^{\text{i}}(va_m)} s''$ (correlation ok and emission), and

$s' \xrightarrow{[G_c^{\text{i}}(va_m)] \quad \tau}$ throw bpel : correlationViolation (correlation error).

Message faults. Message faults are supported as an extension of rules presented in Table 1. Concerned activities are reply and invoke. For reply, we add the reply(pl,o,fn[,v_{err}]) form, where fn is the fault name and v_{err} is an (optional) error variable. This form is transformed as follows:

reply(pl,o,fn[,v_{err}]) = send(pl,o,fn[,v_{err}]), with

send(pl,o,fn[,v_{err}]) $\xrightarrow{\tau \quad / \ \text{v}_{\text{m}} := v_{\text{err}}}$ _ $\xrightarrow{pl.o!!_{fn} v_{\text{m}}}$ empty with $\exists o \in \mathcal{O}(\Sigma_{pl})$, $err(o) = m$.

The synchronous invoke is transformed as follows:

invoke(pl,o,v_{in},v_{out})= send(pl,o,v_{in});rec+(pl,o,v_{out}), with two rules for rec+:

one for the reception of a correct message, rec+(pl,o,v_{in}) = rec(pl,o,v_{in}), and one for the reception of a fault message,

rec+(pl,o,v_{in}) $\xrightarrow{pl.o??_{\text{fn}} v_{\text{m}} \quad / \ v_{\text{in}} := v_{\text{m}}}$ throw fn with $\exists o \in \mathcal{O}(\Sigma_{pl})$, $err(o) = m$.

The catch construct of synchronous invoke is not directly supported but it can be simulated using a fault handler in a scope around the invoke.

On BPEL coverage. We have presented the input/output variable communication scheme. The from/to part communication scheme may be treated in the same way thanks to our explicit use of BPEL anonymous variables. The catch construct of synchronous invoke is not directly supported but it can be simulated using a fault handler in a scope around the invoke. With reference to [12], we

support data (in computation, conditions and message exchange), message correlation, message faults, parallel processing (flows) and the until activity. Links, the foreach activity, fault variables in fault handlers, compensations handlers, and termination handlers are future work.

Application. The STS obtained from the example presented in Section 2 is

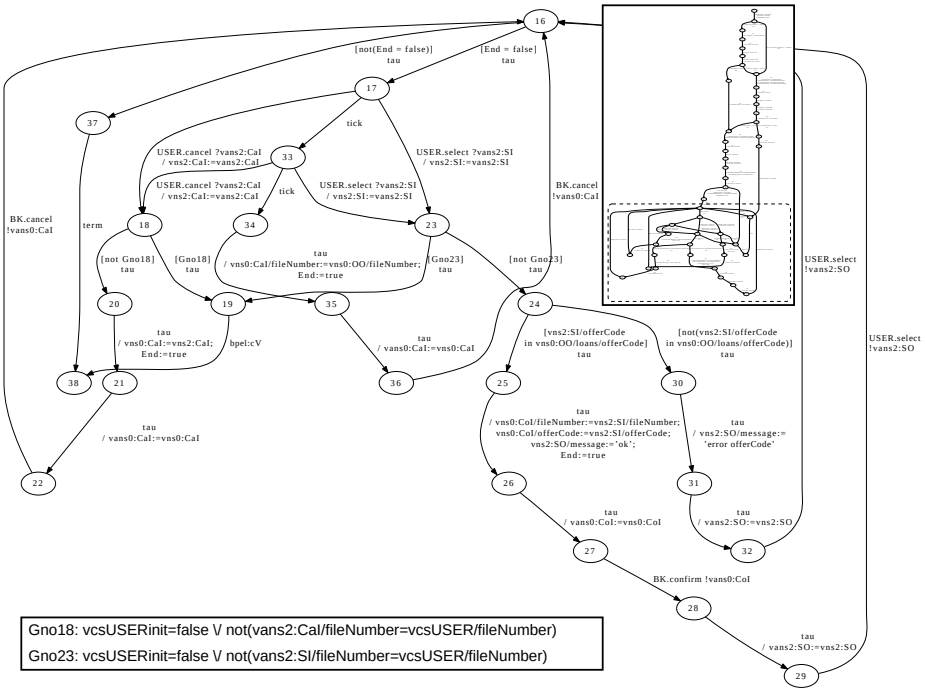

Fig. 4. xLoan Example – Symbolic Transition System

presented in Figure 4 where tau (resp. tick, term) denote τ (resp. χ, $\surd$). The zoom corresponds to the while part. One may notice states 16 (while condition test), 17/33 (pick), 34 (onAlarm timeout), and 18/23 (correlation testing).

4 Symbolic Execution

Symbolic execution [6] (SE) is a program analysis technique that has been originally proposed to overcome the state explosion problem when verifying programs with variables. SE represents values of the variables using symbolic values instead of concrete data [14]. Consequently, SE is able to deal with constraints over symbolic values, and output values are expressed as a function over the symbolic input values. More recently these techniques have been applied to the verification of interacting/reactive systems, including testing [14–16].

SE-Trees. The SE of a program is represented by a *symbolic execution tree* (SET), where nodes, $\mathcal{N}_{\text{SET}}$, are tuples $\eta_i = (s, \pi, \sigma)$ made up of the program counter, s, the symbolic values of program variables, σ, and a path condition, π. Let $\mathcal{V}_{\text{symb}}$ be a set of (symbolic) variables (representing symbolic values), disjoint from the program variables, $\mathcal{V}$ ($\mathcal{V} \cap \mathcal{V}_{\text{symb}} = \emptyset$). σ is a map $\mathcal{V} \to \mathcal{V}_{\text{symb}}$. The path condition (PC) is a boolean formula with variables in $\mathcal{V}_{\text{symb}}$. The PC accumulates constraints that the symbolic variables must fulfil in order to follow a given path in the program.

Since we apply SE to the STS obtained from orchestrations, the program counter is an STS state, and $\mathcal{V}$ corresponds to the STS variables (either simple, message type, anonymous, or correlation variables from BPEL). The edges of the SET, $\mathcal{E}_{\text{SET}}$, are elements of $\mathcal{N}_{\text{SET}} \times Ev_{\text{symb}} \times \mathcal{N}_{\text{SET}}$ (may be non deterministic), where Ev_{symb} corresponds to the STS events (Ev) with symbolic variables in place of variables.

SET edge computation. The SET is computed in a BFS fashion as follows. The root is $(s_0, true, \sigma_0)$ where s_0 is the STS initial state and σ_0 is the mapping of a fresh variable for each variable of the STS. Each transition $s \xrightarrow{[g] \quad e \quad / \ A} s'$ then corresponds to an edge $(s, \pi, \sigma) \xrightarrow{e'} (s', \pi', \sigma')$, computed as follows:

1. **guard:** $\pi^G = \pi \wedge g[\sigma(v_i)/v_i]_{v_i \in vars(g)}$ ($\pi^G = \pi$ if there is no guard)
2. **event:** e', $\sigma^E = \begin{cases} pl.o * v_s, & \sigma[v \to v_s] \text{ if } e = pl.o * v, * \in \{?, ??\} \\ pl.o * \sigma(v), & \sigma \quad \text{if } e = pl.o * v, * \in \{!, !!\} \\ e, & \sigma \quad \text{otherwise} \end{cases}$
   with $v_s = new^1(\mathcal{V}_{\text{symb}}, \sigma)$. If e is a sub-service invocation return ($e = pl.o * v_{out}, * \in \{?, ??\} \wedge pl \neq \mathsf{USER}$), we set $\pi^E = \pi(o)[\sigma^E(v_{in})/in, v_s/out]$, where $e = pl.o!v_{in}$ is the label of the (unique) transition before the one we are dealing with, to take into account the operation specification. Else, $\pi^E = \pi^G$.
3. **actions** ($A = \{x_i/path_i := t_i\}_{i, i \in \{1,\ldots,n\}}$)**:**
   $\pi_i^A = \pi_{i-1}^A \wedge (v_{s_{x_i}}/path_i = t_i[\sigma^E(v_j)/v_j]_{v_j \in vars(t_i)})$
   with $\Delta = \{x \in \mathcal{V} \mid (x/path_i := t_i) \in A\}$, $\{v_{s_x}\}_{x \in \Delta} = new^{\#\Delta}(\mathcal{V}_{\text{symb}}, \sigma^E)$, $\sigma' = \sigma^E\{[v_{s_x}/x]\}_{x \in \Delta}$, $\pi_0^A = \pi^E$, and $\pi' = \pi_n^A$.

where $vars$ denotes the variables in a term, $new^n(\mathcal{V}_{\text{symb}}, \sigma)$ denotes the creation of n new (fresh) symbolic variables wrt. σ, $t[y/x]$ denotes the substitution of x by y in t, and $\sigma[x \to x_s]$ denotes σ where the mapping for x is overloaded by the one from x to x_s. Δ is the set of variables that are modified by the assignments. For each of these, we have a new symbolic variable. We denote $may(\eta)$, $\eta \in \mathcal{N}_{\text{SET}}$, the set $\{e \mid \exists (\eta, e, \eta') \in \mathcal{E}_{\text{SET}}\}$.

Feasible paths. Edges with inconsistent path conditions may be cut off while computing the SET. For this, we check when computing a new node η if $\pi(\eta)$ is satisfiable (there exists a valuation of variables in π such that π is true, if not, we cut the edge off). This is known to be an undecidable problem in general. Therefore, if the constraint solver does not yield a solution (or a contradiction) in a given amount of time, we cut the edge off and we issue a warning specifying that the test process is to be incomplete.

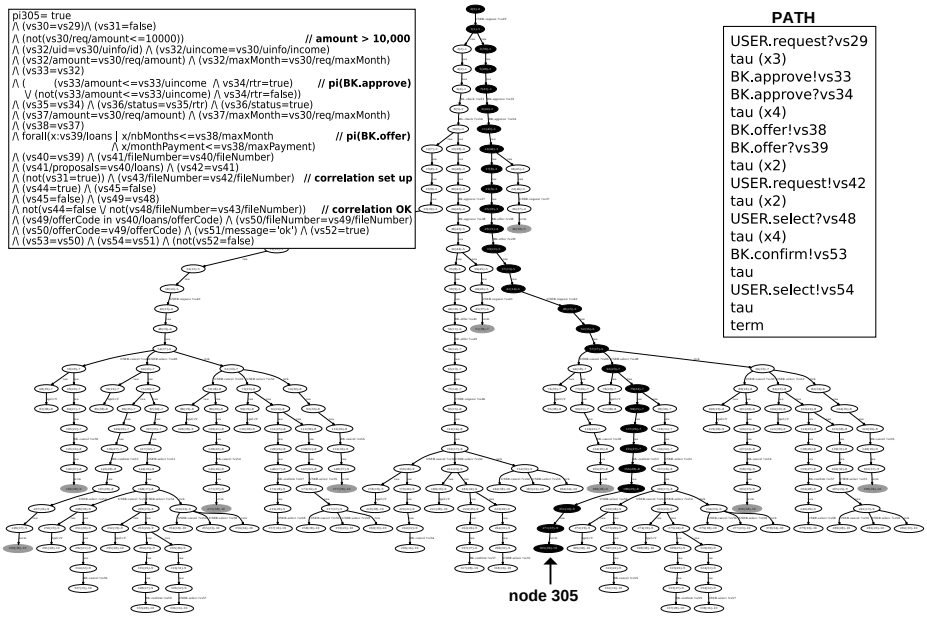

Fig. 5. xLoan Example – Symbolic Execution Tree (k=10, τs not counted)

Path length criterion. STS may contain loops that would cause SET unboundedness. To solve this issue out, we take into account a path length criterion while computing the SET. Given a constant k, we stop the SET computation at some node whenever this node is at k edges from the SET root. In order to take into account the fact that interactions are the most important part of orchestrations, only non-τ transitions can be counted. Different cutting criteria have been proposed in the literature, *e.g.*, the inclusion criterion in [16], where the SET computation stops at one node when an equivalent node (in terms of program variable valuations validating the path constraint) is on the path to the SET root. Yet, this does not prevent the need to a path length criterion [16].

Application. The SET computed from the Figure 4 STS is presented in Figure 5. There are 10 leaves corresponding to termination (in gray). The zoom presents the path (in black) we use for demonstration in the next Section. Its final node is number 305, and its path condition, π_{305} is also given in the Figure.

5 Test Case Realisation and Testing Architecture

In this Section we present the way symbolic test cases are realised and executed on an orchestration implementation. The distinctive features of our approach can be summarised as follows:

- **Functional testing with a SET path coverage criterion.** For the time being, we support two criteria: all paths with length $n \leq k$, and all complete paths (a path ended with $\surd$) with length $n \leq k$. Paths are constructed in a DFS way.
- **Symbolic input-output trace inclusion.** The conformance relation we use is trace inclusion extended to our symbolic context. It can be related to the conformance relation defined in [15, 16] that inspired our work (see *Related Work*). However, we do not require input-enabledness since, for Web services, an exception is returned whenever an unexpected event is received by the service.
- **Online realisation of symbolic test cases.** Test case realisation is performed step by step, by interacting with the Service Under Test (SUT). This is to avoid emitting erroneous verdicts. Take a path $p?x.p!y$, with $\sigma = \{x \rightarrow v_{s_0}, y \rightarrow v_{s_1}\}$ and $\pi = v_{s_0} > 2 \wedge v_{s_1} > v_{s_0}$. Realisation all-at-once would yield a realised path $p?v_{s_0}, p!v_{s_1}$ with, *e.g.*, $\{v_{s_0} \rightarrow 3, v_{s_1} \rightarrow 4\}$. Suppose now we send message p with value 3 to the SUT and that it replies with value 5. We would emit a *Fail* verdict ($5 \neq 4$), while indeed 5 would be a correct reply ($5 > 3$).
- **Branching awareness.** The SUT may send different outputs at some point in its execution due to non-determinism (*e.g.*, due to a flow activity in the implementation). Outputs are non-controllable events from the point of view of the tester. Therefore, a path is realised in the context of its SET (see Alg. 1).

5.1 Online Testing Algorithm

Online testing is presented in Algorithm 1. Its input is the SET with a distinguished symbolic path we want to test. The algorithm then animates the path by interacting, over messages for the USER partnerlink, with the SUT. Accordingly, input (resp. output) events in the path correspond to messages sent (resp. received) by the tester. Generation of data in sent messages and checking of data in received messages is supported using constraint solving over a Path Condition (PC). Initially PC corresponds to the path condition (π) in the last node of the path we test. The treatment of the path edges is then as follows.

- **Input events.** The tester has to generate a message to be sent to the SUT. For this, PC is solved (always succeeds, or the edge would have been cut off in the SET computation). We use the instantiation of the event variable, x_s, to send the message, and to update the PC. If the sent message yields an exception, we return a *Fail* verdict, else we pass to the next edge.
- **Output events.** The treatment of output events corresponds to message reception in the tester. Whenever an emission by the SUT is foreseen, a timer, TAC, is set up. Then three cases may occur. (i). If the timer elapses, we return a *Fail* result. (ii). If we receive the expected message before this, we update the PC with this new information and try to solve it. If it succeeds we continue to the next edge. If it fails we return a *Fail* verdict. If we do not

Algorithm 1: Online Testing Algorithm

```
Data: SET + a distinguished path p, path p = n_1 l_1 n_2 l_2 ... l_{k-1} n_k ;
begin
    π = π_k; i := 1; rtr := Pass ;
    while i < k and rtr = Pass do
        switch l_i do
            case USER.e?x_s
                val := (SOLVE(π)[x_s]);
                try {send (e(val)); π := π ∧ x_s = val;}
                catch (e ∈ Ex) { rtr := Fail; }
            case USER.e!x_s
                start TAC;
                try {receive (e(val)); π = π ∧ (x_s = val);
                     if ¬SOLVE(π) then rtr := Fail; }
                catch (timeout_TAC) {rtr := Fail;}
                catch (receive e') { if e' ∈ may(η_i) then rtr := Inconclusive;
                                     else rtr := Fail;}
            case χ
                wait(1 unit of time);
            otherwise
                skip;
        i := i + 1;
    return rtr;
end
```

get a result in a given amount of time we return an *Inconclusive* verdict (not in the Algorithm for simplicity). (iii). If we receive an unexpected event, we check in the SET if it is due to the specification non-determinism. If not, we return a *Fail* verdict. If it is the case, we return an *Inconclusive* verdict and the test path needs to be replayed in order to exhibit the behaviour that this test path characterises (for this we assume SUT fairness). Fault output events are supported in the same way

– **Time passing (χ)** corresponds to the passing of one unit of time. Accordingly, the tester waits for this time before going to the next event in the path. The unit of time is computed from the specification (one hour in our example). Other events are skipped (see below, *Discussion*, about an alternative for gray-box testing).

5.2 Tool Support and Application

In this part, we end the application to the xLoan example, focusing on the online testing algorithm and on tool support. Our approach is automated by means of prototypes written in the Python language that serve as a proof of concept. As far as constraint solving is concerned, we chose the UML2CSP tool [17], which supports OCL constraint solving over UML class diagrams, that correspond to

the constraints we have on XML schema data. Additionally, UML2CSP is able to generate witness object diagrams when the constraints are satisfiable. More precisely, in order to reuse this tool, we proceed as follows:

- we translate XML schema definitions into an UML class diagram, see Figure 2. Additionally, domain limits are set up in UML2CSP according to uniformity hypotheses, *e.g.*, here we have set maxMonth:{12,24,36} and maxPayment:[1000..100000]). This step is done only once.
- to check if some π is satisfiable *before* sending a message, an additional root class (Root) is created wrt. the UML diagram, with as many attributes as symbolic variables in π. The π constraint is translated in OCL. If π is satisfiable, UML2CSP generates an object diagram. From it we get data for the variable of interest to be sent (step $val := (SOLVE(\pi)[x_s])$ in Alg. 1).
- to check if some π is satisfiable *after* receiving a message, we perform as before, but adding an OCL constraint enforcing that the symbolic variable of the reception is equal to the data effectively received (steps $\pi = \pi \wedge (x_s = val)$ and $\neg SOLVE(\pi)$ in Alg. 1).
- cutting infeasible paths in the SET computation is a sub-case of satisfaction before message sending (the generated object diagram is discarded).
- strings are treated as integers which represent an index in an enumerated type. This corresponds to a set of specific string constants for the test.

For the time being, interaction with UML2CSP is manual. The automation of this step is under process as part of an Eclipse plug-in we are developing.

Experiments have been applied on an implementation of xLoan which is not isomorphic to its specification as, *e.g.*, the BPEL code relies on additional boolean variables rather than on the workflow structure to decide if the subprocess for loan proposal selection or cancelling is applicable. The implementation size is 246 lines long (186 XML tags). In order to demonstrate our online algorithm, we take one of the 10 complete paths we have in the SET (see Fig. 5). Due to lack of room we focus on the first interaction steps of the path (loan request, loan reply). The first call to UML2CSP with the end path condition, π_{305}, enables one to retrieve values for the RequestIn message (id, name, income, amount, maxMonth, maxPayment), *e.g.*, the part of the path condition relative to the requested amount generates the `Context Root inv PC : not(self.vs30.req.amount<=10000)` OCL constraint, and value 10001 for the request amount. We then generate the message data in Figure 6, left, and send it using SOAPUI[5]. The corresponding received message is in Figure 6, right. We translate this data as an OCL constraint and solving it with UML2CSP we are able to show that it is a correct output. We may then proceed generating data for a correct offer selection (offerCode=1).

5.3 Discussion

We have experimented the application of our framework for functional testing based on structural criteria. We think that it could be used in other contexts.

[5] `http://www.soapui.org/`

```
<soapenv:Envelope xsi:...="http:... >
<soapenv:Body>
<ns2:RequestIn>
  <ns3:uInfo>
    <id>1</id>
    <name>Simpson</name>
    <income>10002</income>
  </ns3:uInfo>
  <ns3:req>
    <amount>10001</amount>
    <maxMonth>12</maxMonth>
    <maxPayment>1000</maxPayment>
  </ns3:req>
</ns2:RequestIn>
</soapenv:Body>
</soapenv:Envelope>
```

```
<soapenv:Envelope xsi:...="http:... >
<soapenv:Body>
<ns2:RequestOut>
  <status>true</status>
  <fileNumber>1</fileNumber>
  <ns3:proposals>
    <offerCode>1</offerCode>
    <nbMonths>12</nbMonths>
    <monthPayment>918</monthPayment>
    <ns3:rate>
      <type>fixed</type>
      <value>10</value>
    </ns3:rate>
  </ns3:proposals>
</ns2:RequestOut>
</soapenv:Body>
</soapenv:Envelope>
```

Fig. 6. xLoan Example – a Sent and a Received Message (parts of)

First, one may need to test –at design-time, or at run-time for dynamic service binding– if a SUT may serve as a sub-service, being given a specification of the way one would like to communicate with it. This could be achieved by generating a partner service description [18] for this specification, and then apply our technique. Besides, supporting test objectives described as a directed acyclic STS with leaves tagged using $\{Pass, Fail\}$ could be performed using the STS product [19] of this STS with the specification STS prior to SET generation.

Additionally, one could have a gray-box point of view over orchestrations using the information available in the SET. We are experimenting another version of Algorithm 1 where communications with the partners are no longer skipped by the tester which receives invocations of the SUT and replies as if it was the sub-service (generating data respecting the π boolean formula in the sub-service signature specification). For this, we use the same constraint satisfaction approach as presented before.

6 Related Work

The state of practice in service testing has been limited for a long time to the use of tools such as SOAPUI or BPELUnit [20] that release from the burden of the translation into SOAP messages, operation calling and test management. However, test-cases were mainly generated using empirical approaches, often without automation.

In the last years, the software testing community has started to get involved in the service field. As a consequence, several works have tried to bridge the gap between current practice in service testing and state-of-the-art formal and automated software testing. One way to address this issue is to focus on the service signatures, *i.e.*, their WSDL description [21]. This enables to test operations independently. However, WSDL does not provides neither a semantic information on services nor a behavioural description of them, which is important in presence of composite (orchestration) services.

To the contrary, we chose to focus on this complementary part of orchestrations. This point of view has been adopted by several works, through white-box

testing. Approaches based on control-/data-flow coverage criteria are presented in [22–24]. Classical testing data-flow criteria are revisited in [22] in order to be suitable to services. A control-flow algorithm is combined in [23] with rewriting graphs to support XPath expressions. The use of BPEL model-checking for white-box test case generation has been proposed in [25, 26]. In both cases, test cases are considered as counter-examples and generated, according to several coverage criteria, with the SPIN model-checker. While [25] uses an intermediary model, [26] transforms directly BPEL into Promela. All these white-box approaches assume that the implementation source code is available.

In earlier work [10], we have addressed gray-box testing using translation of BPEL into the IF language and the extension of the IF simulator to generate tests according to a test objective. However, the use of data domain enumeration yield state explosion. To circumvent this problem, we propose in this paper a black-box testing approach using translation into STS and symbolic execution. We took inspiration from previous work on symbolic testing [19, 16, 15, 27]. The works in [19, 16], still, did not address components or services. Application to this domain has first been proposed in [15], and later in [27], from a theoretical and generic point a view, without a specification/implementation language in mind. In our work, we propose a comprehensive language-to-language approach with BPEL as target language. Accordingly, compared to the above-mentioned works, we take into account BPEL specific features in test-case derivation.

7 Conclusion and Perspectives

With the development of service reuse through their aggregation in added-value composite services, the testing of orchestrations has become a topical issue. In this paper we have presented a framework for orchestration testing based on symbolic transition systems which, compared to related work, supports the rich XML-based data types used in (Web) services without suffering from state explosion issues. This framework also proposes a comprehensive language-to-language approach, as it deals with both the retrieval of formal models from real service specification languages and with the execution of test cases using the SOAPUI API. Although we have presented here the transformation rules from BPEL to STS, we advocate that transformations can be defined from other languages with workflow features (UML, BPMN) and accordingly provide the software architect with a richer specification environment.

Ongoing work is relative to the re-engineering of our tool prototypes in Java and their integration in a plug-in extension of the Eclipse BPEL Designer, following [28]. A first perspective of our work is to support conformance testing based on test objectives. This would provide a valuable alternative to the path length criterion when dealing with infinite SET. Another perspective is relative to the supported basic XML types, currently only integers and strings. Using String databases, as proposed in [21], in place of simple enumerated values would enhance the tests relevance. Finally, supporting additional parts of BPEL, in-

cluding compensation/termination handlers and string operators would enable to target a wider range of BPEL specifications.

References

1. OASIS: Web Services Business Process Execution Language (WSBPEL) Version 2.0. Technical report, OASIS (April 2007)
2. Rao, J., Su, X.: A Survey of Automated Web Service Composition Methods. In: Proc. of SWSWPC. (2004)
3. Dumas, M., Benatallah, B., Motahari Nezhad, H.R.: Web Service Protocols: Compatibility and Adaptation. IEEE Data Eng. Bull. **31**(3) (2008) 40–44
4. ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal Methods for Service Composition. Annals of Mathematics, Computing & Teleinformatics **1**(5) (2007) 1–10
5. Poizat, P., Royer, J.C.: A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic. Journal of Universal Computer Science **12**(12) (2006) 1741–1782
6. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7) (1976) 385–394
7. Attiogbé, C., Poizat, P., Salaün, G.: A Formal and Tool-Equipped Approach for the Integration of State Diagrams and Formal Datatypes. IEEE Transactions on Software Engineering **33**(3) (2007) 157–170
8. Ouyang, C., van der Aalst, W., Dumas, M., ter Hofstede, A.: Translating BPMN to BPEL. Technical Report BPM-06-02, BPM Center Report (2006)
9. Bucchiarone, A., Melgratti, H., Severoni, F.: Testing Service Composition. In: Proc. of ASSE. (2007)
10. Lallali, M., Zaïdi, F., Cavalli, A., Hwang, I.: Automatic Timed Test Case Generation for Web Services Composition. In: Proc. of ECOWS. (2008)
11. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. In: Proc. of ICSOC. Volume 5364 of LNCS. (2008)
12. Mateescu, R., Rampacek, S.: Formal Modeling and Discrete-Time Analysis of BPEL Web Services. In: Advances in Enterprise Engineering I. Volume 10 of Lecture Notes in Business Information Processing., Springer (2008) 179–193
13. W3C: XML Path Language (XPath) Version 1.0. Technical report, W3C (1999)
14. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Proc. of TACAS. Volume 2619 of LNCS. (2003)
15. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Proc. of FATES/RV. Volume 4262 of LNCS. (2006)
16. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic Execution Techniques for Test Purpose Definition. In: Proc. of TESTCOM. Volume 3964 of LNCS. (2006)
17. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In: Proc. of ASE. (2007)
18. Kaschner, K., Lohmann, N.: Automatic Test Case Generation for Interacting Services. In: Proc. of ICSOC 2008 Workshops. Volume 5472 of Lecture Notes in Computer Science. (2009)
19. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic Test Selection based on Approximate Analysis. In: Proc. of TACAS. Volume 3440 of LNCS. (2005)
20. Mayer, P.: Design and Implementation of a Framework for Testing BPEL Compositions. PhD thesis, Leibnitz University, Germany (2006)

21. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: Towards Automated WSDL-Based Testing of Web Services. In: Proc. of ICSOC. Volume 5364 of LNCS. (2008)
22. Bartolini, C., Bertolino, A., Marchetti, E., Parissis, I.: Data Flow-Based Validation of Web Services Compositions: Perspective and Examples. In: Architecting Dependable Systems. Volume 5135 of LNCS. (2008)
23. Mei, L., Chan, W., Tse, T.: Data Flow Testing of Service-Oriented Workflow Applications. In: Proc. of ICSE. (2008)
24. Li, Z., Sun, W., Jiang, B., Zhang, X.: BPEL4WS Unit Testing: Framework and Implementation. In: Proc. of ICWS. (2005)
25. Zheng, Y., Zhou, J., Krause, P.: An Automatic Test Case Generation Framework for Web Services. Journal of Software **2**(3) (2007) 64–77
26. García-Fanjul, J., Tuya, J., de la Riva, C.: Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In: Proc. of WS-MaTe. (2006)
27. Frantzen, L., Huerta, M., Kiss, Z., Wallet, T.: On-The-Fly Model-Based Testing of Web Services with Jambition. In: Proc. of WS-FM. Volume 5387 of LNCS. (2009)
28. H.Foster, S.Uchitel, J.Magee, J.Kramer: WS-Engineer: A Tool for Model-Based Verification of Web Service Compositions and Choreography. In: Proc. of ICSE. (2006)