Checking the Behavioral Conformance of Web Services with Symbolic Testing and an SMT Solver

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

[1] LRI; Univ. Paris-Sud, CNRS, Orsay France
[2] Univ. Évry Val d'Essonne, Evry, F-91000, France
{lina.bentakouk,pascal.poizat,fatiha.zaidi}@lri.fr

Abstract. Workflow-based service composition languages foster the rapid design and development of distributed applications. The behavioral verification of service Compositions has widely been addressed at design time, using model-checking. Testing is a complementary technique when it comes to check the behavioral conformance of a service implementation with respect to its specification or to a user or a service need. In this paper we address this issue with an automatic approach based on symbolic testing and an SMT solver.
keywords: services, orchestration, formal testing, test-case generation, WS-BPEL, transition systems, symbolic execution, SMT solver.

1 Introduction

Context. Web services are gaining industry-wide acceptance and usage as the main implementation of the Service Oriented Architecture. As such, they support the construction of distributed applications out of reusable heterogeneous and loosely coupled software entities, namely services. WS-BPEL [1] (BPEL for short) has been normalised as the language for Web service orchestration, *i.e.*, centralised service composition. BPEL is twofold. It may be used to describe orchestration implementations. Abstracting details and refereed to as Abstract BPEL (ABPEL), it may also be used to specify orchestrations or to publish information about the behaviour of a service, *i.e.*, the ordering in which its operations should be called.
Motivations. Since BPEL is a semi-formal workflow language, a significant research effort has been produced in the last years in order to propose formal models for orchestration verification [2]. Numerous work in this area have addressed model-checking, to check if properties are verified by an orchestration specification. Still, in presence of a black box implementation, one cannot retrieve a model from it. To help establishing the conformance wrt. a specification, testing should be used. However, to the contrary of model-checking, testing is incomplete. One rather focuses on generating the *good* test cases to search for errors. Testing enables one to ensure both that sub-services participating to an orchestration conform to their publicised behavioural interface, and that the orchestration itself conforms to the behavioural interface to be publicised after its deployment. Orchestration testing has mainly been addressed from a white-box perspective, assuming that the orchestration implementation source code is available. In practise, the source code is often not available as it constitutes an added-value for the

service providers. To overcome this limit, we propose a black-box conformance testing approach to automate the conformance checking of composed services. We base our approach on a symbolic treatment of data and on an efficient SMT solver.

Contributions. The contributions of our work are manifold. (*i*) Based on a behavioural specification, we produce a formal model, namely a Symbolic Transition System (STS) [3, 4] by means of transformation rules written in a process algebraic style [5]. We handle as well interfaces as behavioural parts of composed services. (*ii*) A particular accent is put on the processing of structured data types in such a way that they will be handled by a more powerful constraints solving tool compared to our previous work. (*iii*) The testing process is based on the definition of requirements that allow a user to specify the service behaviour to test. Based on requirements or user needs, the tester provides test purposes. We suppose these are given as STS too, yet, it has been shown that such transition system models could be obtained automatically either from abstract properties given in a temporal logic or from human-friendly descriptions such as sequence diagrams. (*iv*) We support the test of a real service orchestrator, in which we use some knowledge about the database of the partner services. This assumption is needed in order to provide pertinent data input to the tested orchestrated service. (*v*) We clearly explain our stepwise approach, which starts with the definition of a formal model of the specification and of the test purposes. Afterwards, we produce by means of a symbolic execution tree (SET) the test cases that cover the test purpose. The SET allows to avoid the unfolding of the STS product that would yield state space explosion. Then, an online oracle tester executes the test cases against the real implementation.

Outline. The remainder of the paper is structured as follows. The next section presents the related work. Then a running example is provided in Section 3. A formal approach is detailed in Section 4 in which we describe the use of test purposes. In Section 5 the automatic test case generation, based on the STS product and SET generation, is explained. In Section 6 we present the tools chain with an emphasis on the use of a solving tool. Finally Section 7 ends with conclusion and perspectives.

2 Related Work

Web service (WS) testing has been addressed differently depending on the available service description level and on the testing generation approach [6]. A simple but limited technique is to inject hand-crafted test cases on a service implementation using tools such as soapUI or BPELUnit [7]. However, these tools do not allow to do intensive testing, since they require a manual instantiation of the test cases. When the WS source code is available, white-box testing criteria can be used to generate test cases [8]. Sometimes, only service signatures are available. Still, this enables to test WS by generating relevant input data for each operation [9, 10]. In the later work, the control flow for Java control flow code is also considered. Going beyond WSDL WS, composite services, *e.g.*, orchestrations, feature a behavioural interface. It is important to take this information into account in the testing process.

In [11], they have addressed grey-box testing using a BPEL to IF (Intermediate Format) translation and extending the IF simulator to generate test cases given a test objective. A limitation is state space explosion in presence of complex data types. In [12]

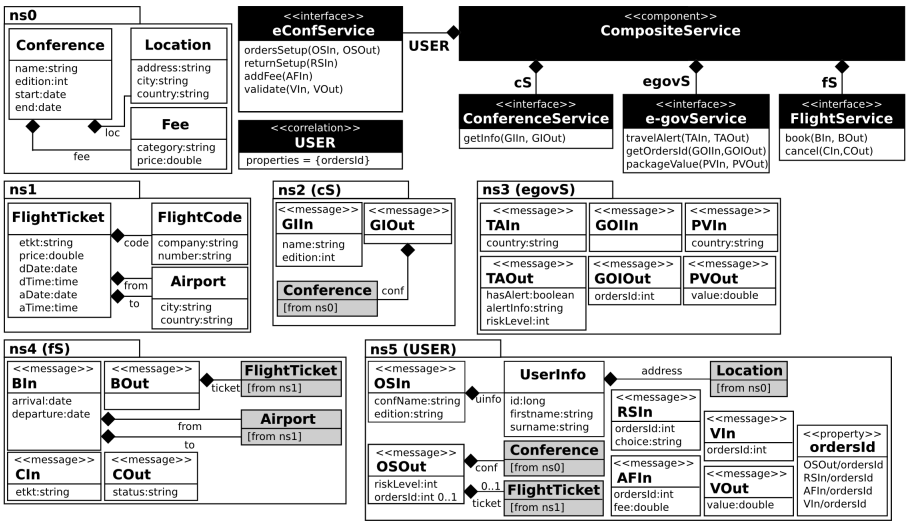


Fig. 1. e-Conference Example – Data and Service Architecture (UML Extended)

the authors propose an approach for compliance testing of the specification wrt. the implementation of a composed Web services, using Petri-nets. This approach allows to automatically generate test cases. However, they did not address the problem of generating input data for test.

To solve this problem out, we propose to rely on symbolic models and their symbolic semantics. Symbolic testing approaches have been investigated for several years, *e.g.*, in [13–15]. Our test purpose driven test case generation is closely related to [13, 14]. However, except for [15] (still, from a theoretical and generic point of view), the above mentioned approaches have not addressed application to the component or service composition domain. This application requires a comprehensive language-to-language approach with, *e.g.*, transformation rules supporting a workflow-based language with complex data types and communication mechanisms. We have proposed in [4] a WS symbolic testing approach based on specification coverage criteria. Here, this work is extended to formally propose a more precise and stepwise approach driven by symbolic test purpose which allows a user to test specific part of an orchestration. In other words, the test purposes described in this paper refer to behavioral functionalities that a tester wants to check. Further, in [4] we also used the UMLtoCSP [16] tool to solve constraints, which required a very fine tuning of the variables' domains. Moreover, a more complex model transformation between this tool and ours was needed, which is no more the case with the Z3 SMT solver.

3 Running Example

In this Section we present our e-Conference case study. This medium-size orchestrated service provides functionalities to researchers going to a conference such as information

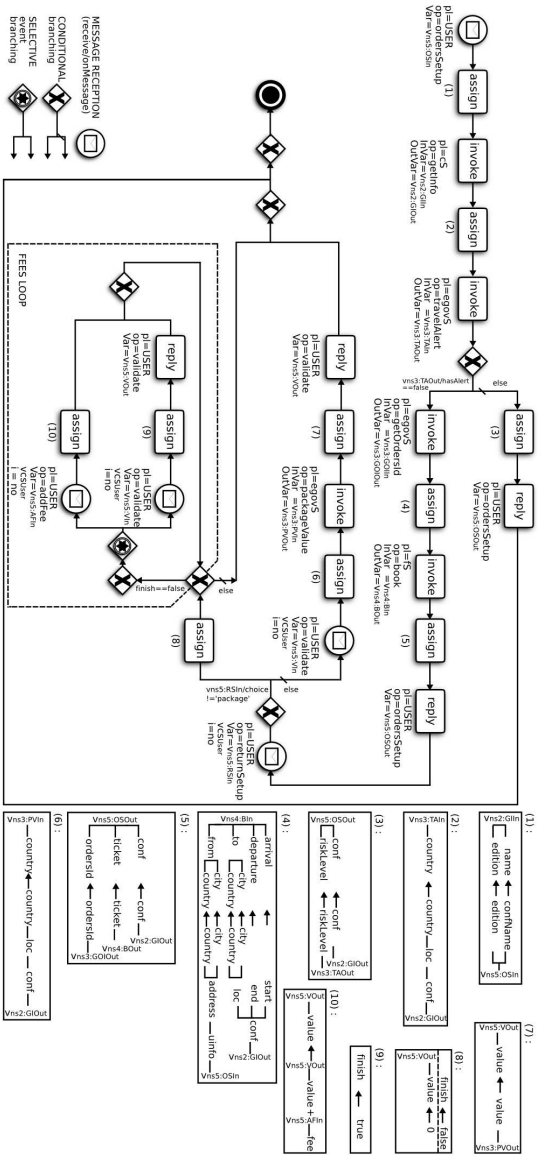


Fig. 2. e-Conference Example – Orchestration Specification (Inspired and extended from BPMN)

about the conference, flight booking and fees refunding. Parts of the orchestration, *e.g.*, the e-governance rules, correspond to reality. Other parts, *e.g.*, the sub-service used to book plane tickets, represent a simplified yet realistic version of it.

e-Conference is based on three sub-services: ConferenceService, FlightService, and e-govService. Its specification is as follows. A user starts the process by providing the conference name and edition, together with personal information (ordersSetup operation). Conference information is first retrieved using ConferenceService then e-govService is invoked to check if any travel alerts exist in the conference country. If there is one, the user is informed and the process stops. If not, an orders id is asked to e-govService, a plane ticket is bought using FlightService and all gathered information (orders id, conference and plane) is sent back to the user who may then go to the conference. Upon return, the user may choose to be refunded from either a fees or package basis (returnSetup operation). In both cases, the user will end by validating the mission (validate operation) and e-Conference will reply with refunding information. If fees basis has been chosen, the user will be able to send information on fees (using addFee operation several times).

Figure 1 exhibits our extension of the UML notation corresponding to the e-Conference orchestration. Within this diagram we highlight the stereotypes for message types, correlations and properties to represent the orchestration architecture and the imported data (XML schema files structured in namespaces ns_x). Concerning orchestration specification shown in Figure 2, 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, cS, egovS, and fS for the sub-services), operation, input/output variables, and, when it applies, information about message correlation.

4 Formal Models for Test Case Generation

In this section we briefly present our formal model for service orchestration, the motivations and steps that lead us to reuse the same model to represent test purposes.

4.1 Service Orchestration Model

Different models have been proposed to support behavioural service discovery, verification, testing, composition or adaptation [2, 17, 11, 18]. They mainly differ in their formal grounding (Petri nets, transition systems, or process algebra), and the subset of service languages being supported. We base on [19] due to its wide coverage of 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. With reference to [19], we support data (in computation, conditions and messages), message faults (enabling a service to inform its partners about internal errors), message correlation (enabling BPEL engines to correlate messages in-between service instances), flows (parallel processing) and the until activity. More specifically, support for data yields grounding on (discrete time) Symbolic Transitions Systems and their symbolic execution, rather than on (discrete time) Labelled Transition Systems. Unlike the model described in [3] we have modified the WS-STS model to take into account

the specific characteristics of Web Services.

STS modelling. A *Web Service Symbolic Transition System* (WS-STS), is a tuple $(\mathcal{D}, \mathcal{V}, S, s_0, T)$ where $\mathcal{D}$ is a set of domains (data type specifications), $\mathcal{V}$ 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, $T \subseteq S \times \mathcal{T}_{\mathrm{Bool},\mathcal{V}} \times Ev \times seq(Act) \times S$, with $\mathcal{T}_{\mathrm{Bool},\mathcal{V}}$ denoting Boolean terms possibly with variables in $\mathcal{V}$, Ev a set of events represents the messages communication. $\mathcal{D}$ and $\mathcal{V}$ are often omitted when clear from the context (*e.g.*, $\mathcal{V}$ are variables used in transitions). A reception or the return of an invocation event noted $pl.o?x$ corresponds to the reception of the input message x from the partner pl of the operation o. We omit BPEL port types for simplicity reasons (full event prefixes would be, *e.g.*, $pl.pt.o?x$). Accordingly, we define a reply or invocation event $pl.o!x$ corresponds to an emission of an output message. $Ev^?$ (resp. $Ev^!$) is the set of reception events (resp. emission events). Ex is the set of internal fault events, that corresponds 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 and $\surd$ denotes the termination of a conversation (end of a session). For reasons of simplicity we denote $Ev = Ev^? \cup Ev^! \cup Ex \cup \{\tau, \surd\}$. $seq(Act)$ is a set of actions denoting computation (data processing) that will be executed in a sequential way (of the form $v := t$ where $v \in \mathcal{V}$ is a variable and $t \in \mathcal{T}_{\mathcal{D},\mathcal{V}}$ is a term).
The transition system is called symbolic as the guards, events, and actions may contain variables. $(s, g, e, A, s') \in T$ is also written $s \xrightarrow{[g]e/A}_T s'$ or simply $s \xrightarrow{[g]e/A} s'$ when clear from the context. The WS-STS moves from the state s to the state s' if the event e occurs and the guard g is satisfied and some actions can be performed. 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 WS-STS transitions are variables from BPEL and anonymous variables used for interacting with other services. Notice that each data type (structured or scalar) of BPEL corresponds to an element of $\mathcal{D}$ and each variable of BPEL corresponds to a variable in $\mathcal{V}$. An orchestration is built around a partnership, *i*.e., a set of (partners) signatures corresponding to required operations and a set of signatures corresponding to provided operations. In the sequel, we suppose, without loss of generality, that an orchestration has only one of the later, named USER. STS have been introduced under different forms (and names) in the literature [3], to associate a behaviour with a specification of data types that is used to evaluate guards, actions and sent values. This role is played by $\mathcal{D}$ which is a superset of all partner's domains. Transformation rules from BPEL to WS-STS are provided in [5].
Application. From the e-Conference specification (see Figure 2), we obtain the STS in Figure 3 (49 states, 57 transitions) where tau (resp. term) denote τ (resp. $\surd$). The zoom (grey states) corresponds to fees loop. One may notice states 39 (while condition test) and 40 (pick). In states $41/45$ it is checked if incoming messages (validate or addFee) come from the same user than the previous ones in the conversation (ordersSetup and returnSetup). When it is not the case (correlation failure) an exception is raised (in state 26). Variables names, in our example are prefixed with namespaces (*e.g.*, vns5:VOut is the variable storing values of type ns5:VOut) to help the reader.

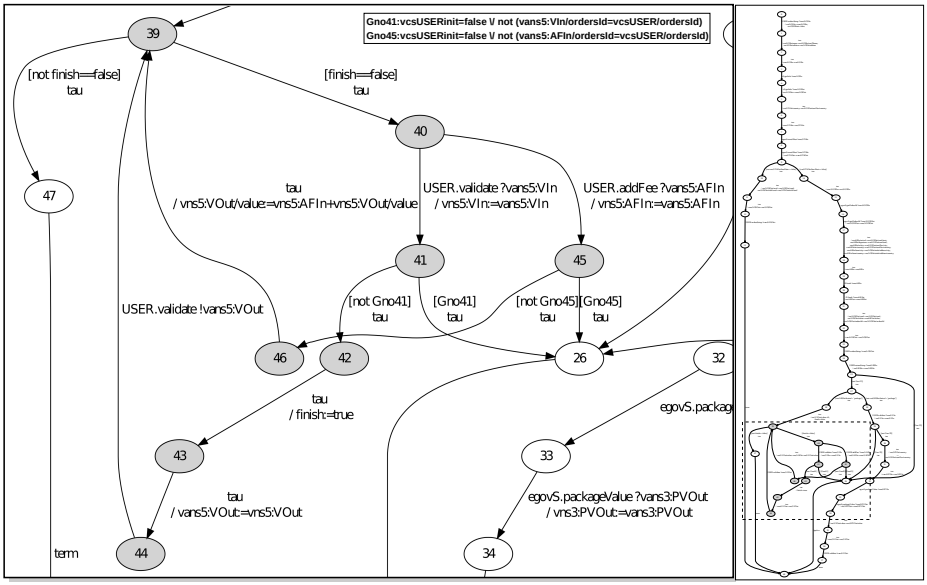


Fig. 3. e-Conference Example – Orchestration model (STS)

4.2 Test Purpose Model

A Test Purpose (TP) is a set of functional properties allowed by the specification and that one is interested to test. Usually, the formal model for TP follows the one used for the system specification. Hence Labelled Transition Systems (LTS) is the most popular model for TP. However in our case it will be formalised as an STS according to the specification model. Note that a TP can also be modelled using Linear Temporal Logic (LTL) to be more abstract. The average user may prefer more user friendly notation *eg*. MSC or UML sequence diagrams [20, 21] that describe the interactions between system components. In both case we can get back to transition system model, LTL can be transformed in Buchï automata [22] while, MSC and UML sequence diagrams can be transformed in LTS [23].

To represent formally requirements as a test purpose we were inspired by the work of [13]. However, the way to express a test purpose is simpler because we don't need *reject* states to specify an undesired behaviour. Thus the WS-STS resulting product contains only the paths that run through an accept state.

TP models are defined according to the orchestration (specification) models they refer to. Therefore, given an orchestration model $\mathcal{B} = (\mathcal{D}_\mathcal{B}, \mathcal{V}_\mathcal{B}, S_\mathcal{B}, s_{0_\mathcal{B}}, T_\mathcal{B})$, a TP for $\mathcal{B}$ is an WS-STS $TP = (\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$ with some constraints. TP may use a set of additional variables $\mathcal{V}_I$ for expressiveness (see *Application*, below), disjoint from $\mathcal{B}$ variables. $\mathcal{V}_{TP} = \mathcal{V}_I \cup \mathcal{V}_\mathcal{B}$ where $\mathcal{V}_I \cap \mathcal{V}_\mathcal{B} = \emptyset$, accordingly $\mathcal{D}_{TP} \supseteq \mathcal{D}_\mathcal{B}$, with $\forall t\ s \xrightarrow{[g]e/v:=t} s' \in T_{\mathcal{TP}}\ v \in \mathcal{V}_I$. Assignments in TP can only operate on $\mathcal{V}_I$.
The events labelling TP transitions correspond to the $\mathcal{B}$ ones. More specifically, we

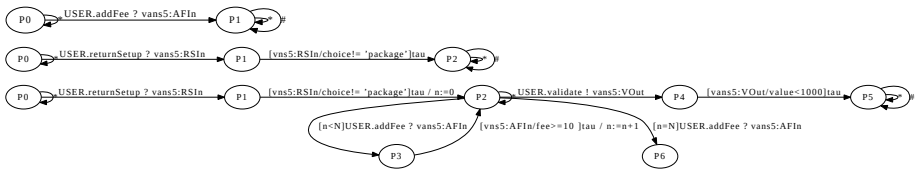


Fig. 4. e-Conference Example - Test Purposes (STS)

impose for simplicity sake that variables used in message exchanges (events of the form pl.op . . .) correspond to the ones in $\mathcal{B}$. This constraint can be lifted using substitutions. TP also introduce a specific event, *. Transitions labelled with * may neither have a guard, nor actions, and are used to abstract in TP one or several $\mathcal{B}$ transitions that are not relevant for the expression of the requirement. A TP defines a specific set of states, $Accept$ ($Accept \subseteq S_{TP}$), that denotes TP satisfaction. Those states have transitions labelled by an event $\#$. Finally, we impose that TP is consistent with $\mathcal{B}$, *i.e.*, TP symbolic traces are included in $\mathcal{B}$ ones. This can be checked using symbolic execution (see Sect. 5.2), where we also have to check that the path condition corresponding for the TP trace implies the path condition of the $\mathcal{B}$ trace.

Application. Let us assume one wants to focus on testing refunding on a fees basis. Possible TPs are given in Figure 4. In the first TP, one may note the use of the addFee transition, leading to an acceptance state (Labelled with #). This specifies addFee must be part of the generated test cases. Several transitions in the specification could be done before and after the addFee one, therefore, the two TP states are equipped with a * loop. The * loop is interpreted as being the execution of any other communication messages (including calls to sub-services), except those explicitly expressed. The first TP requires that there is at least one addFee in the test cases. One may want to take also into account the case where there are none. This can be specified as in the second TP. There, one relies on the returnSetup transitions that carry user requests relative to the return mode (package or fees basis). In order to specify it is the later which is required, a guarded transition is used (choice should be different from 'package'). Note that the guard is put on a τ transition after the reception in order to be consistent with (symbolic) execution semantics: the guard can only be evaluated once the variable has been received. The last TP, used in the sequel, is more realistic and demonstrates TPs expressiveness with four requirements: (i) return is on a fees basis, (ii) each fee is greater or equal to 10, (iii) the total amount for the mission is less than 1000, and (iv) there are at most N fees added. The later requires to use a TP additional variable, n, to count addFee iterations.

5 Automatic Test Case Generation

5.1 WS-STSs Product

To generate test cases we have to take into account constraints specified both in the specification and in the TPs. This is achieved using WS-STS (symbolic) product.

Given a specification model $\mathcal{B} = (\mathcal{D}_{\mathcal{B}}, \mathcal{V}_{\mathcal{B}}, S_{\mathcal{B}}, s_{0_{\mathcal{B}}}, T_{\mathcal{B}})$, and a test purpose $TP =$

$(\mathcal{D}_{TP}, \mathcal{V}_{TP}, S_{TP}, s_{0_{TP}}, T_{TP})$, their product, $Prod = \mathcal{B} \otimes TP$, is the WS-STS $(\mathcal{D}_{Prod}, \mathcal{V}_{Prod}, S_{Prod}, s_{0_{Prod}}, T_{Prod})$ where $\mathcal{D}_{Prod} = \mathcal{D}_{TP}$, $\mathcal{V}_{Prod} = \mathcal{V}_{TP}$, $S_{Prod} \subseteq S_{\mathcal{B}} \times S_{TP}$, $s_{0_{Prod}} = (s_{0_{\mathcal{B}}}, s_{0_{TP}})$, and T_{Prod} is built using four rules:

$$\bullet\ \forall e \in \{\tau, \chi, \sharp\},\quad \frac{s_{TP} \xrightarrow{[g]e/A}_{TP} s'_{TP}}{(s_{\mathcal{B}}, s_{TP}) \xrightarrow{[g]e/A}_{Prod} (s_{\mathcal{B}}, s'_{TP})} \qquad \bullet\ \forall e \in Ev^{?} \cup Ev^{!} \cup Ex \cup \{\surd\},\quad \frac{s_{\mathcal{B}} \xrightarrow{[g_{\mathcal{B}}]e/A_{\mathcal{B}}}_{\mathcal{B}} s'_{\mathcal{B}},\ s_{TP} \xrightarrow{[g_{TP}]e/A_{TP}}_{TP} s'_{TP}}{(s_{\mathcal{B}}, s_{TP}) \xrightarrow{[g_{\mathcal{B}} \wedge g_{TP}]e/A_{TP};A_{\mathcal{B}}}_{Prod} (s'_{\mathcal{B}}, s'_{TP})}$$

$$\bullet\ \forall e \in \{\tau, \chi, \sharp\},\quad \frac{s_{\mathcal{B}} \xrightarrow{[g]e/A}_{\mathcal{B}} s'_{\mathcal{B}}}{(s_{\mathcal{B}}, s_{TP}) \xrightarrow{[g]\tau/A}_{Prod} (s'_{\mathcal{B}}, s_{TP})} \qquad \bullet\ \forall e \in Ev^{?} \cup Ev^{!} \cup Ex \cup \{\surd\},\quad \frac{s_{\mathcal{B}} \xrightarrow{[g_{\mathcal{B}}]e/A_{\mathcal{B}}}_{\mathcal{B}} s'_{\mathcal{B}},\ s_{TP} \xrightarrow{*}_{TP} s'_{TP},\ \not\exists\, s_{TP} \xrightarrow{[g_{TP}]e/A_{TP}}_{TP} s'_{TP}}{(s_{\mathcal{B}}, s_{TP}) \xrightarrow{[g_{\mathcal{B}}]e/A_{\mathcal{B}}}_{Prod} (s'_{\mathcal{B}}, s'_{TP})}$$

The two rules on the left side denote that TP (resp. $\mathcal{B}$) evolves independently for non observable events. The first rule on the right side corresponds to synchronising between TP and $\mathcal{B}$. Finally, the second rule on the right side defines the * semantics. It corresponds to observable events but for the ones that are captured by the first rule on the right side. To enforce the acceptance states semantics, the product is cleaned up by pruning states (and related transitions) that are not co-reachable from acceptance states, *i.e.*, any s such that $\not\exists s' = (s'_{\mathcal{B}}, s'_{TP}) \in S_{Prod}\ .\ s \rightarrow *s' \wedge s' \in Accept$. Notice that a cleaning of the WS-STS product is performed to keep only the paths that pass through an accept state.

Application. The product of the orchestration (Figure 3) with the third TP (Figure 4) is given in Figure 5. It has 68 states and 85 transitions (89 states and 119 transitions before pruning). Its set of symbolic traces is a subset of TP one (hence also of the orchestration one). One may note for example that receiving addFee is possible only if done less than N times (see guard [n<N] in the transition outsourcing from state 45), and that the condition on fee values is also taken into account (states $48/54/61/67$).

5.2 Symbolic Execution Tree

Symbolic execution [24] (SE) 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 [25]. 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 [25, 26, 14].

The SE of a program is represented by a *symbolic execution tree* (SET), where nodes, $\mathcal{N}_{\mathrm{SET}}$, 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 $\mathcal{V}_{\mathrm{symb}}$ be a set of (symbolic) variables (representing symbolic values), disjoint from the program variables, $\mathcal{V}$ ($\mathcal{V} \cap \mathcal{V}_{\mathrm{symb}} = \emptyset$). σ is a map $\mathcal{V} \rightarrow \mathcal{V}_{\mathrm{symb}}$. The path condition (PC) is a Boolean formula with

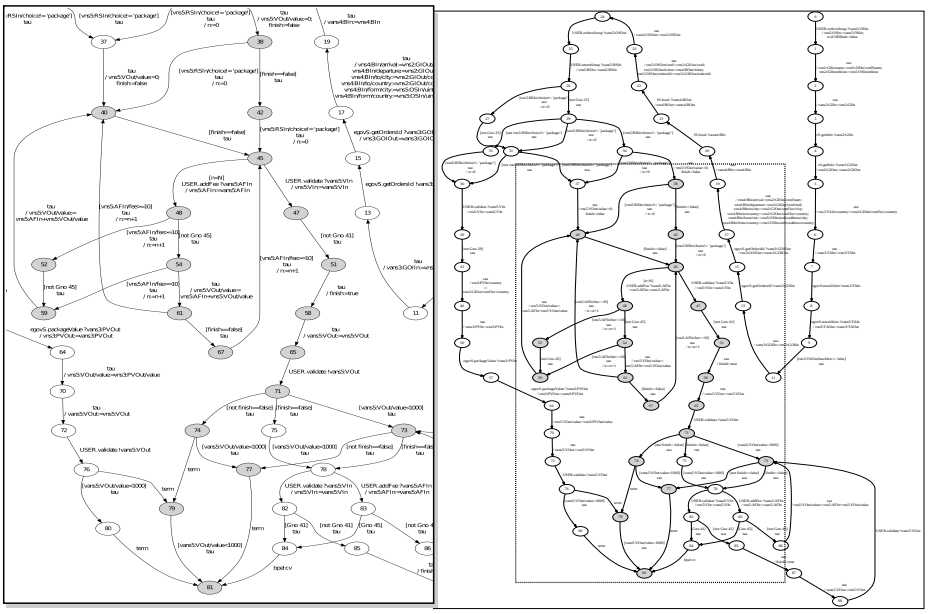


Fig. 5. e-Conference Example - Product (STS)

variables in $\mathcal{V}_{\text{symb}}$. The PC accumulates constraints that the symbolic variables must fulfill in order to follow a given path in the program. Since we apply SE to an WS-STS product, the program counter is an WS-STS state, and $\mathcal{V}$ corresponds to the WS-STS product variables. 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 WS-STS product events (Ev) with symbolic variables in place of variables. Each feasible path of a SET (i.e., each path in this SET with a PC that can be solved) represents a possible test case.

5.3 SET computation

The SET is computed in a BFS fashion as follows. The root is $(s_0, true, \sigma_0)$ where s_0 is the WS-STS product initial state and σ_0 is the mapping of a fresh variable for each variable of the WS-STS ($\sigma_0 : \forall v \in \mathcal{V}, v \mapsto newVar$) and $\pi_0 = true$. Each transition $s \xrightarrow{[g]e/A} s'$ then corresponds to an edge $(s, \pi, \sigma) \xrightarrow{e'} (s', \pi', \sigma')$, computed as described in [4].

1. **guard:** $\pi^G = \pi \wedge g\sigma$ ($\pi^G = \pi$ if there is no guard)
2. **event:** e', $\sigma^E = \begin{cases} pl.o?v, \sigma[v \rightarrow v_s] & \text{if } e = pl.o?v \\ pl.o!\sigma(v), \sigma & \text{if } e = pl.o!v \\ e, \sigma & \text{otherwise} \end{cases}$
   with $v_s = new(\mathcal{V}_{\text{symb}}, \sigma)$. If e is a sub-service invocation return ($e = pl.o?v_{out}$

$\wedge pl \neq$ 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 ($\pi(o)$). 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. Note that we suppose without losing of generality that in practise one assign with parallel instructions is executed sequentially.

We denote $may(\eta)$, $\eta \in \mathcal{N}_{\text{SET}}$, the set $\{pl.o!v \mid \exists \eta \xrightarrow{L} *\eta' \xrightarrow{[g]pl.o!v/A} \eta"\}$ with L a sequence of labels such that the corresponding word (keeping only the event in labels), contains only non observable events or communication events with partners ($\{\tau, \chi, \surd\} \cup \{pl.o * v \mid * \in \{?, !\} \wedge pl \neq USER\}$). This set will be used later on for the test verdict emission.

Pruning infeasible paths. Edges with inconsistent path conditions are 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.

Pruning redundant paths. A WS-STS product may contain loops that would cause SET unboundedness. To solve this issue out, we propose two compatible techniques. We can first 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, this technique is inspired by the k bounded model checking [27]. The user can re-start the SET computation process with $k + 1$ for more test cases.

A complementary approach is to use the *inclusion criterion* as proposed by [14]. Let us consider $\eta = (s, \pi, \sigma)$, a reachable node in the SET. Solving the associate path condition π means that there exists at least a value for each symbolic variable satisfying the constraint with regards to the σ mapping ($\mathcal{V} \to \mathcal{V}_{\text{symb}}$). Such constraint allows several interpretations (combinations of possible values). $\mathcal{M}_\eta^\mathcal{V}$ represents the set of all the possible interpretations for the variables $\mathcal{V}$ of the node η. Consider now, an other node $\eta' = (s, \pi', \sigma')$ which has the same state as η. We say that η' is included in η ($\eta' \subseteq \eta$), when $\mathcal{M}_{\eta'}^\mathcal{V} \subseteq \mathcal{M}_\eta^\mathcal{V}$. That means that the set of interpretations of the variables of $\mathcal{V}$ belonging to η is bigger than the one belonging to η', so $\pi'\sigma' \implies \pi\sigma$. If the two nodes are on the same branch, we can prune the sub-tree associated to the node η' as it is included in the sub-tree of the node η. We can then stop the generation of the SET at the node η'.

Application. Among the 85 paths in the SET ($k = 15$) of the product STS (see Figure 5), there are 7 complete paths corresponding to the coverage of the last test purpose

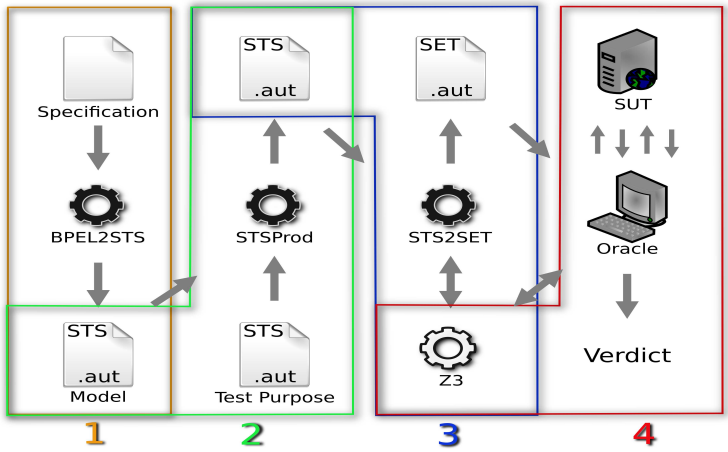


Fig. 6. Overview of the tools chain

described in Section 4.2. A path example is:

```
USER.ordersSetup?vs36 tau(x2) cS.getInfo!vs40
cS.getInfo?vs41 tau(x2) egovS.travelAlert!vs44
egovS.travelAlert?vs45 tau(x2) egovS.getOrdersId!vs47
egovS.getOrdersId?vs48 tau(x2) fS.book!vs51
fS.book?vs52 tau(x2) USER.ordersSetup!vs55
USER.returnSetup?vs58 tau(x5) USER.addFee?vs69
tau(x4) USER.validate?vs74 tau(x3) USER.valivate!vs77
tau(x2) term.
```

6 Tools Support and Experimentations

Figure 6 exhibits the different steps of our approach that are supported by tools. In step 1: from a specification represented as a BPEL file we use the BPEL2STS tool to generate an STS model. Followed by step 2: the produced model and a test purpose STS (in the same format) are provided as inputs files to our STSprod tool that computes the product. In step 3: the STS2SET tool takes as input the STS product and uses the Z3 solver [28] to compute the SET. Finally in step 4: using the SET paths we execute the test cases as described in the test cases realisation with an automated call to Z3 but the oracle algorithm is exercised manually. All intermediate formats follows the Aldebaran format (.aut) which supports a graphical representation with the CADP tool [29].

SMT Solver Use during the SET construction. Our approach relies on a symbolic execution semantic of our WS-STS. Compared to work that enumerate the possible values of variables [11] for the test cases, we rely on a solving tool to instantiate the path condition. Enumeration techniques can be faced to the state space explosion problem and false verdicts can be emitted as explained later on. In our case we can overcome the state space explosion problem by avoiding the unfolding of the WS-STS and with our on line testing algorithm we can obtain the possible values by a call to the solving tool and hence instantiate the values by interacting step by step with the service under

test. To solve the constraints we use an efficient SMT (Satisfiability Modulo Theories) solving tool, i.e. Z3. Z3 allows a direct use of arithmetic in Boolean formulae. Z3 is called by a satisfiability function. First when we compute a node η of the SET related to a path condition π that has been augmented by a condition because of a guard in the WS-STS (an if or a $while$), a call to the satisfiability function is performed. An SMT file is built to be given to Z3. For the inclusion criterion depicted above for the pruning of redundant paths, we also call this function to check the satisfiability of $\pi'\sigma' \Longrightarrow \pi\sigma$ and the complexity is on the number of edges from η to η'.

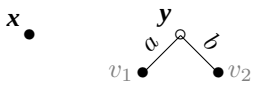
Symbolic test case extraction. Symbolic test cases correspond to the SET paths. However, it may be relevant to test only paths leading to orchestration termination ($\surd$) even if it is not strongly required for Web services since different instances are run for each test case. Due to our k path length criterion in the SET computation, it follows that symbolic test cases have a length $n \leq k$. Notice that we can increase the value of k if we did not find errors during the execution of tests.

Online realisation with a constraint solving tool. Since Web services are reactive systems, test case realisation has to be performed step by step, by interacting with the Service Under Test (SUT). This is to avoid emitting erroneous verdicts. Take a path $pl.o?x.pl.o!y$, 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 ($5 > 3$). Concretely, the tester is implemented as a reactive process that mirrors the observable behaviour of the test case. We begin with the PC in the last state of the SET path corresponding to the test case. Whenever the tester has to send a message to the SUT (USER.o? in the test case), PC is solved to get correct data values to send. Whenever the tester has to receive a message from the SUT (USER.o! in the test case) a timeout is run. If it ends before we receive the message there is an *Inconclusive* verdict. If we receive the message from the SUT, data is extracted from it and the PC is updated with a correspondence between the reception variable(s) and the data, and PC is then checked to see if the data is correct or not. In both cases, we rely on a constraint solving tool, the Z3 solver.

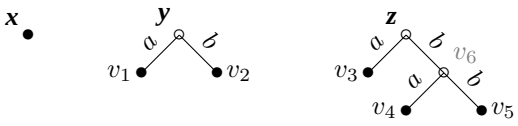
Call of the Z3 Solver for the Online Realisation. We present in the following how the solving function based on the call of the Z3 tool is used by the online algorithm to retrieve values in order to execute the tests. The solving function takes as inputs variables and their types, a predicate represented as a conjunction of clauses and a set of words that will be used as a small database to assign a string to a string variable. This function supports the use of the following datatypes: boolean ($Bool$) integer (Int) double ($Real$) and string ($String$). For sake of simplicity we use only the type Int in the sequel. First, the function builds for each variable its associated tree. Each tree is constructed according to its variable type. Hence, if it is a simple type variable then the associated tree will be reduced to a simple type leaf. Else if it is a complex type variable, then an isomorphic tree will be constructed. We need to manipulate tree types to be compliant with simple or complex types of XSD schema. Let us assume a variable x, a variable y and a complex type named T_1. T_1 consists of a part a and another part b, both are integer. We provide the following annotation for this complex type: $T_1 : \{a : Int,$

$b : Int\}$. The variables x and y have the following types: $x :: Int$ and $y :: T_1$. The corresponding tree for x will be reduced to an integer leaf and the one associated to y is a tree with an integer leaf through a and an integer leaf through b. Notice that we use the term *Labels* to refer to a or b (the label a and the label b). We give below a graphical view of the trees.

Once the trees are constructed, a new variable is assigned to each leaf of the tree. These new variables will be used as pointers when the instantiation of variables of complex type will be performed. In fact, each path of the tree starting from the root to a leaf is represented with a unique variable as shown below:



Once the previous step is achieved, we process the predicate. As we said above a predicate is a conjunction of clauses. Theses clauses may represent an equivalence operations between variables or/and paths of trees as well as arithmetic and logic operations. A path in predicate clause is represented with an $XPath$ expression. If we consider the same previous variables x and y and the following predicate: $x = 3 + 4 \wedge y/b = x$, the clause $y/b = x$ expresses the equivalence between the value of the variable x and the value of the location pointed by the path y/b.
If the $XPath$ expression y/b does not correspond to a leaf, but to a sub-tree then a new variable is created to represent this sub-tree. We would like to point out that if the $XPath$ expression leads to a leaf, then the path is replaced by its associated variable.
Let us consider z an input variable (as x and y) and a new complex type T_2 where: $T_2 : \{a : Int, b : T_1\}$ and $z :: T_2$. The associated predicate to those variables is specified as: $x = 3 + 4 \wedge y/b = x \wedge z/b = y$, in this case a new variable will be created by the program to represent z/b. The graphical view is given below.

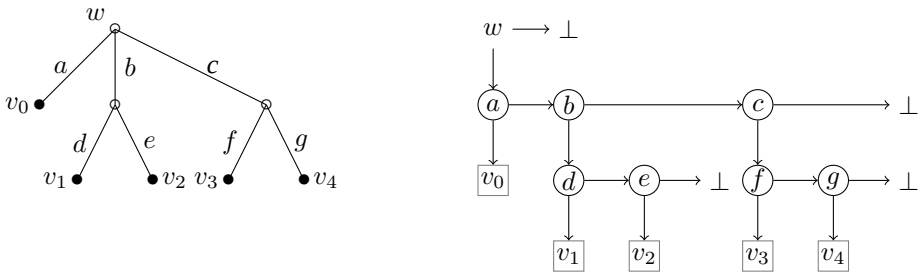


The next step is to submit the predicate using the new variables instead of the $XPath$ expressions to the Z3 solver. The function generates an SMT input file for Z3. This input file represents a problem to be submitted to Z3 and has to satisfy the following BNF rules (note that we use the `teletype font` to indicate the part that will appear in the Z3 input file).

Problem ::= Prefix Theory Var-Declaration Constraint-Formula Suffix
Prefix ::= " `( benchmark data` "
Theory ::= Labels-Declaration String-Declaration Tree-Declaration
Var-Declaration ::= " `:extrafuns (` " *(variableName variableType)*$^+$ " `)`"
Constraint-Formula ::= "`:formula ( and` " *variable-Structure constraintClauses*"`)`"
Suffix ::= " `)`"
Labels-Declaration ::= " `:datatypes ((label` " $Labels^*$ " `)` "
String-Declaration ::= " `(String` ("$word^+$ | "`noString`)" `)`
Tree-Declaration ::= "`(tree (nil) (IntLeaf (val Int))`
`(RealLeaf (val Real))(StringLeaf (val String))`
`(BoolLeaf (val Bool))`
`(cons (firstChildLab label) (firstChild tree)`
`(siblingChilds tree))))`"

A problem is divided into three main sub-parts: the *Theory*, the *Var-Declaration* and the *Constraint-Formula*. The *Theory* is used to specify the string words allowed as a given database and also to specify the structure of a complex type variable. The second part, i.e. the *Var-Declaration*, allows to declare each variable and its type. The different types can be: Int, $Real$, $Bool$ or a $tree$ type. Finally the *Constraint-Formula* sub-part allows to represent at first the structure of a complex variable type as specified by the theory (this corresponds to the "*variable-Structure*" in the rules above). Secondly the predicate (or the constraint), presented as a conjunction of clauses, will be written into a particular form (this corresponds to the *constraintClauses* in the rules above). To explain this last point, if we consider the very simple example $x = 3 + 4$, the corresponding constraint with the concrete syntax of Z3 will be written as follows:
$(= x\ (+\ 3\ 4))$.

The $Labels^*$ represent the set of tree labels of complex type variables, there is no labels for simple type variable. $word^+$ represent the words given as input to the program. The *Tree-Declaration* defines a tree structure. We give below an example of a tree given as an input to the solving function and how we encode it with the concrete syntax of Z3. We give a linear representation of the tree, with the label connected to its ancestor, its first children with its descendants and all the siblings of this children and their descendants. The nil constructor represents an empty tree.
Example. Let us consider the variable w with a tree as graphically represented below. The associated translation in the concrete syntax of Z3 is as follows. We give also the graphical representation of the encoding of the tree in Z3. Let us note that in the Z3 graphical representation we have the nil symbol associated to w for the empty tree for the sibling of the root w. Nevertheless, in the constraints below we simplify the structure by avoiding the nil for w as we are interested in the tree accessible through the label connected to w.
$(= w\ (cons\ a\ (IntLeaf\ v_0)\ (cons\ b\ (cons\ d\ (IntLeaf\ v_1)\ (cons\ e\ (IntLeaf\ v_2)\ (nil)))(cons\ c\ (cons\ f\ (IntLeaf\ v_3)\ (cons\ g\ (IntLeaf\ v_4)\ (nil))))\ (nil))))$



Once the SMT input file for Z3 is generated, we submit it to the solver. This one can return an instantiation of variables if the predicate is satisfiable (sat), an unsatisfiable answer ($unsat$) otherwise. An other answer could be given which is the unknown response ($unknown$) if the solver cannot give a response in a given amount of time. The solving function retrieves the values returned by the solver and assigns them to the variables. In the case of variables represented as trees, the leaves variables will receive the different values and thus complete the construction of trees. The instantiated variables will be used during the test case execution. The function can be easily extended to interact with different solving tool such as Alt-Ergo [30] for instance. We need to add some methods for the dumping of the problem to be solved in the syntax accepted by the tool and also to add a method to retrieve the responses of this tool.

Observability and controllability. When the tester receives an unexpected message (USER.o! in the test case), it does not always corresponds to an error (and hence, to a *Fail* verdict). This is due first to non-controllability of the SUT outputs (one cannot prevent a SUT to send a message). Moreover, due to non observable events ($\{\tau, \chi, \surd\}$) and internal communications with sub-services in orchestrations, the state we are in the test case execution, say η, does not always correspond to the state in which a correct implementation may have evolved. Therefore, whenever we receive an unexpected message, we check if it is in $may(\eta)$. If it is the case, we produce an *Inconclusive* verdict, else a *Fail* verdict. Notice that the *Inconclusive* verdict related to $may(\eta)$ is a consequence of the reception of an event related to the test of another branch of the test purpose. The conformance relation considered is a symbolic input-output trace inclusion as defined in [14].

Application. For the example e-Conference (see Section 3), we have defined five test purposes that allow to cover the different functional behaviour of the service. One is related to a demand of a user with a country having a high alert. Two others are related to an order with a country with no alert and with either a fees or package refund. The two last are related to orders with fees refund and a variation on the number of addfee. For each test purpose, we have produced several test cases, and only one can be executed with an uniformity hypothesis on the different values of the variables. In this work, we focus on generating test cases and solving constraints in order to first compute the SET then use the instantiate data returned by the solver to interact with the implementation. The interaction with the implementation can be done using the soapUI tool as depicted in [4].

7 Conclusion and Perspectives

The emergence of business models based on service composition calls for dedicated verification techniques. In this paper we have presented a formal testing framework addressing this issue. Formal models are retrieved from a real service composition language, WS-BPEL, thanks to a comprehensive set of transformation rules. Using symbolic models and their symbolic execution semantics, model state space explosion can be avoided. We do not only address test case generation but also tackle the way test cases are run against a service implementation. Using symbolic test purposes, the designer of an orchestration, or anyone interested in reusing it in a value-added composition, may have the testing process focus on functional properties of interest. Our framework is automated using prototypes. Our framework has been applied to realistic-size case studies. The main perspective of this work is to go beyond a pure black-box approach, defining an off-context testing approach where the test oracle would simulate orchestration partners. We also plan to define a language based on workflow (like BPMN) to express the test purposes.

Acknowledgement. This work was supported by the project "WEB service MOdelling and Validation" (WEBMOV) of the French National Agency for Research.

References

1. OASIS: Web Services Business Process Execution Language (WSBPEL) Version 2.0. Technical report, OASIS (April 2007)
2. ter Beek, M.H., Bucchiarone, A., Gnesi, S.: Formal Methods for Service Composition. Annals of Mathematics, Computing & Teleinformatics **1**(5) (2007) 1–10
3. Poizat, P., Royer, J.C.: A Formal Architectural Description Language based on Symbolic Transition Systems and Modal Logic. JUCS **12**(12) (2006) 1741–1782
4. Bentakouk, L., Poizat, P., Zaïdi, F.: A Formal Framework for Service Orchestration Testing based on Symbolic Transition Systems. In: Proc. of TESTCOM. (2009) 16–32
5. Bentakouk, L., Poizat, P., Zaïdi, F.: A Formal Framework for Service Orchestration Testing based on Symbolic Transition Systems. Long version, in P. Poizat Webpage (2009)
6. Bozkurt, M., Harman, M., Hassoun, Y.: Testing Web Services: A Survey. Technical report, King's College London (2010)
7. Mayer, P.: Design and Implementation of a Framework for Testing BPEL Compositions. PhD thesis, Leibnitz University, Germany (2006)
8. 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)
9. Bartolini, C., Bertolino, A., Marchetti, E., Polini, A.: Towards Automated WSDL-Based Testing of Web Services. In: Proc. of ICSOC. (2008)
10. Hallé, S., Hughes, G., Bultan, T., Alkhalaf, M.: Generating interface grammars from wsdl for automated verification of web services. In: ICSOC/ServiceWave. (2009) 516–530
11. Lallali, M., Zaïdi, F., Cavalli, A., Hwang, I.: Automatic Timed Test Case Generation for Web Services Composition. In: Proc. of ECOWS. (2008)
12. Kaschner, K., Lohmann, N.: Automatic test case generation for interacting services. In: ICSOC Workshops. (2008) 66–78

13. Jeannet, B., Jéron, T., Rusu, V., Zinovieva, E.: Symbolic Test Selection based on Approximate Analysis. In: Proc. of TACAS. (2005)
14. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic Execution Techniques for Test Purpose Definition. In: Proc. of TESTCOM. (2006)
15. Frantzen, L., Huerta, M., Kiss, Z., Wallet, T.: On-The-Fly Model-Based Testing of Web Services with Jambition. In: Proc. of WS-FM. (2009)
16. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a Tool for the Formal Verification of UML/OCL Models using Constraint Programming. In: Proc. of ASE. 547–548
17. Kovács, M., Varró, D., Gönczy, L.: Formal analysis of BPEL workflows with compensation by model checking. Int. Journal of Computer Sciences & Engineering (2008) 35–49
18. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. In: Proc. of ICSOC. (2008)
19. Mateescu, R., Rampacek, S.: Formal Modeling and Discrete-Time Analysis of BPEL Web Services. In: Advances in Enterprise Engineering I, vol. 10. LNBIP (2008) 179–193
20. Uchitel, S., Kramer, J., Magee, J.: Synthesis of behavioral models from scenarios. IEEE Trans. Softw. Eng. **29**(2) (2003) 99–115
21. Ziadi, T., Hélouët, L., Jézéquel, J.M.: Towards a uml profile for software product lines. In: Proc. of Software Product-Family Engineering. (2003) 129–139
22. Gastin, P., Oddoux, D.: Fast LTL to Büchi Automata Translation. In: Proc. of CAV '01. (2001)
23. Pickin, S., Jard, C., Jéron, T., Jézéquel, J.M., Traon, Y.L.: Test synthesis from uml models of distributed software. IEEE Trans. Software Eng. **33**(4) (2007) 252–269
24. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM **19**(7) (1976) 385–394
25. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized Symbolic Execution for Model Checking and Testing. In: Proc. of TACAS. (2003)
26. Frantzen, L., Tretmans, J., Willemse, T.A.C.: A Symbolic Framework for Model-Based Testing. In: Proc. of FATES/RV. (2006)
27. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers **58** (2003) 118–149
28. Moura, L.D., Bjørner, N.: Z3: An efficient smt solver. In: Proc. of TACAS. (2008)
29. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A Toolbox for the Construction and Analysis of Distributed Processes. In: CAV. (2007) 158–163
30. Conchon, S., Contejean, E., Kanig, J.: Ergo : a theorem prover for polymorphic first-order logic modulo theories (2006) http://ergo.lri.fr/papers/ergo.ps.