A Test Purpose and Test Case Generation Approach for SOAP Web Services

Sébastien Salva, Issam Rabhi
LIMOS CNRS UMR 6158
PRES Clermont University, Campus des Cézeaux
Aubière, FRANCE
sebastien.salva@u-clermont1.fr, rissam@isima.fr

***Abstract*—SOA Web services are now supported by most of major software development companies and industry. To be reliable, these ones require to be developed with respect to a complete software development life cycle and, in particular, they need to be tested. Test purpose-based methods are black box testing techniques which take advantage of reducing the time required for test derivation in comparison with exhaustive methods. Nevertheless, test purposes must be constructed manually. This paper proposes some test purpose generation methods for SOAP Web services, modelled by Symbolic Transition Systems (STS). Prior to generate test purposes, we augment the specification with the SOAP environment, to benefit from the messages generated by SOAP processors which give new information about the operations and the faults received. Then, we describe the test case generation from test purposes by synchronizing them with the specification. Test cases are finally translated into XML to be used later by the Soapui tool.**

***Keywords*-Stateful Web services; STS; SOAP; test purpose generation.**

I. Introduction

Web services represent a remarkable branch in the evolution of software development since they offer substantial advantages such as the externalization of Business or social applications available on the Internet, or the reuse of software accompanied by cost reduction. During the recent years, industry has embraced Web services as well-accepted channel for conducting E-Businesses on the Web.

Nevertheless, to ensure that Web services hold their promises, it is crucial that testing activities play an important role in the development process. Indeed, to achieve trustworthy Web services in an environment like the Internet, these ones must also be tested to check various aspects such as robustness, security and conformance which is the topic of this paper. Several testing methods concerning Web services testing have been proposed recently [1], [2], [3]. Some of them, dealing with conformance testing, are said exhaustive i.e., the test case selection is performed to ensure that a faulty implementation is detected by a least one test case. This exhaustiveness often implies to construct test cases for covering all the actions of a specification at least one time. So, the test case generation is often costly and eventually may lead to a state space explosion.

Test purpose-based methods represent an interesting alternative which solve the previous issues and which can be used to test various properties (not the whole system). The test selection is then guided and thereby reduced since test purposes target the test of some implementation parts only. But, although using this approach greatly reduces test costs, the main encountered issue is that test purposes are formulated manually. This task is particularly difficult when the system is large, has real-time constraints or is distributed. Few works propose to solve this issue. For instance, Henniger et al. propose to generate automatically test purposes for distributed systems [4] by considering the specific properties of these latter. None method has been proposed for service-oriented applications though. This is why we present, in this paper, some test purpose generation methods for stateful Web services to test the following specific properties: the operation existence, the critical states and the exception handling. To test them, we augment the specification, modelled with the STS formalism (Symbolic Transition System [5]), with the SOAP environment. Indeed, this one gives more information about the operations and the faults produced by Web services under test. Then, we describe the test case generation. We define a synchronous product which combines the specification and test purposes to produce test cases which can be executed on the implementation and which contain the test purpose properties. Finally, test cases are translated into XML to be executed with the Soapui tool [6], which is dedicated to the functional testing of Web services.

This paper is structured as follows: In Section II, we define both Web service and test purpose modelling. Section II-A provides an overview on some related works about Web service testing. We describe the advantages granted by SOAP for testing in Section III and define the specification completion. Test purpose generation methods are given in Section IV. Section V describes the testing method: we detail the test case generation and the testing framework. Finally, Section VI describes some experimentation results and Section VII concludes with some perspectives.

II. Web service and test purpose modelling

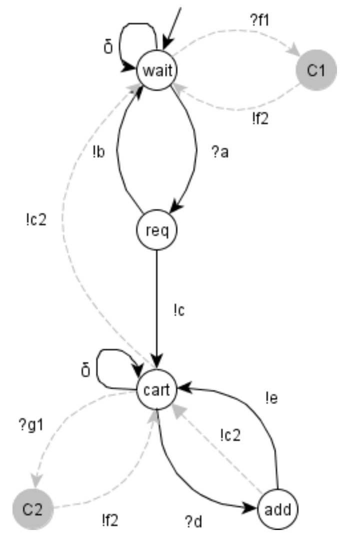
We formalize, in this paper, Web services with Symbolic Transition Systems (STS [5]). This extended automaton model associates a behaviour with a specification composed of transitions labelled by actions and of internal and external variables sets, which may be used to send or receive concrete values and to set guards which must be satisfied to fire transitions. Below, we only summarize the STS suspension definition where quiescence (the lack of observation) is taken into account with the δ symbol. The complete definition can be found in [5].

Definition 1 *A (suspension) Symbolic Transition System STS is a tuple $< L, l_0, V, V_0, I, \Lambda, \rightarrow>$, where:*

- *L is the finite set of locations, with l_0 the initial one,*
- *V is the finite set of internal variables, I is the finite set of external or interaction ones. We denote D_v the domain in which a variable v takes values. The internal variables are initialized with the assignment V_0, which is assumed to take an unique value in D_V,*
- *Λ is the finite set of actions, partitioned by $\Lambda = \Lambda^I U \Lambda^O$: inputs, beginning with ?, are provided to the system, while outputs (beginning with !) are observed from it. $a(p) \in \Lambda$ is an action where $p = (p_1, ..., p_k)$ is a finite set of external variables. We denote $type(p) = (t_1, ..., t_k)$ the type of the variable set p. δ denotes the quiescence i.e., the lack of observation from a location,*
- *$\rightarrow$ is the finite transition set. A transition $(l_i, l_j, a(p), \varphi, \varrho)$, from the location $l_i \in L$ to $l_j \in L$, also denoted $l_i \xrightarrow{a(p),\varphi,\varrho} l_j$ is labelled by $a(p) \in \Lambda \times I$, $\varphi \subseteq D_V \times D_p$ is a guard which restricts the firing of the transition. Internal variables are updated with the assignment $\varrho : D_V \times D_p \rightarrow D_V$ once the transition is fired.*

The STS model is not specifically dedicated to Web services. These latter may be invoked with methods called operations. This is why, for modelling, we assume that an action a in Λ represents either the invocation of an operation op which is denoted $opReq$ or the return of an operation op with $opResp$. For an STS S, we denote $\mathcal{OP}(S)$ the operation set found in Λ. We also assume that service handlers, which may be used to modify SOAP messages, are actions of the specification. An example is illustrated in Figures 1(a) and 2 (solid transitions only). This one describes a part of the Amazon Web Service devoted for e-commerce (AWSECommerceService). For sake of simplicity, we consider only two operations, "CartCreate" which creates a virtual cart composed of items, and "CartAdd", which fills the existing cart with new items. Initially, a customer has to create a cart by giving a correct identifier (AWSAccessKeyID), an item identifier and a quantity. If the cart is instantiated (the $CartCreate$ operation response is composed of the variable Isvalid equal to "true"), this one can be upgraded with the $CartAdd$ operation. Note that we do not include all the parameters for readability reasons.

An STS is also associated to an LTS (Labelled Transition System) to define its semantics. Intuitively, the LTS semantics corresponds to a valued automaton without symbolic variables: the states are labelled by internal variable values while transitions are labelled with actions and parameter values. The semantics of an STS $S =< L, l_0, V, V_0, I, \Lambda, \rightarrow>$ is expressed by an LTS $||S|| =< Q, q_0, \sum, \rightarrow>$.



(a) An STS specification

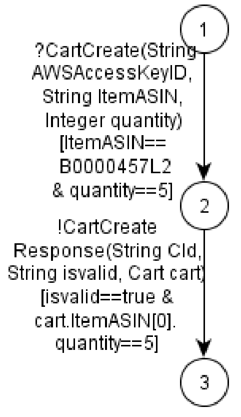

1
?CartCreate(String AWSAccessKeyID, String ItemASIN, Integer quantity) [ItemASIN== B0000457L2 & quantity==5]
2
!CartCreate Response(String CId, String isvalid, Cart cart) [isvalid==true & cart.ItemASIN[0]. quantity==5]
3
(b) A test purpose

Figure 1.

?a	?CartCreate(String AWSAccessKeyID, String ItemASIN, Integer quantity) id:=AWSAccessKeyID q:=quantity
!b	!CartCreateResponse(String[] errors, String isvalid) [isvalid==false & id<>"ID"]
!c	!CartCreateResponse(String CID, String isvalid, Cart cart) [isvalid==true & id=="ID" & q>0] CartId:=CID
!c2	!(soapfault,cause) [q≤0]
?d	?CartAdd(String AWSAccessKeyID, String ItemASIN, Integer quantity, String CID) [CID==Cartid] id:=AWSAccessKeyID q:=quantity
!e	!CartAddResponse(String isvalid) [isvalid==true & id=="ID" & q>0]
?f1	?CartAdd(String AWSAccessKeyID, String ItemASIN, Integer quantity, String CID)
!f2	!a(p) [a(p)≠(soapfault,"Client") & a(p)≠(soapfault,"the endpoint...")]
?g1	?CartCreate(String AWSAccessKeyID, String ItemASIN, Integer quantity)
?sc1	?CartCreate(String AWSAccessKeyID, String ItemASIN, Integer quantity) [ItemASIN==B0000457L2 & quantity==5] id:=AWSAccessKeyID q:=quantity
!sc2	!CartCreateResponse(String CId, String isvalid, Cart cart) g_1=[isvalid==true & id=="ID" & q>0 & cart.ItemASIN[0].quantity==5] CartId:=CID
!sc3	!CartCreateResponse(String CId, String isvalid) g_2=[isvalid==true & id=="ID" & q>0 & cart.ItemASIN[0].q $\neq$ 5] CartId:=CID
!sc4	!CartCreateResponse(String CId, String isvalid) [$\neg g_1$ & $\neg g_2$]

Figure 2. Specification symbols

Test purposes describe the test intention. We assume that they are composed of specification properties which should be met in the implementation under test. Usually, test purposes do not represent complete specification paths. Therefore, they are often synchronized with the specification to generate executable test cases. Consequently, for a specification $S =< L, l_0, V, V_0, I, \Lambda, \rightarrow>$, we also formalize a test purpose with a deterministic and acyclic STS $TP =< L_{TP}, l0_{TP}, V_{TP}, V0_{TP}, I, \Lambda, \rightarrow_{TP}>$ where internal variables of the test purpose and the specification are exclusive ($V \cap V_{TP} = \emptyset$), $\rightarrow_{TP}$ is composed of transitions modelling specification properties. So, for any transition $l_j \xrightarrow{a(p),\varphi_j,\varrho_j} l'_j \in \rightarrow_{TP}$, it exists a transition $l_i \xrightarrow{a(p),\varphi_i,\varrho_i} l'_i \in \rightarrow$ and a value set $(x_1, ..., x_n) \in D_I$ such that $\varphi_j \wedge \varphi_i(x_1, ..., x_n) \models$ true. A test purpose example for the AWSECommerceService is illustrated in Figure 1(b). This one aims to create a cart composed of five items whose identification number (ASIN) is *B0000457L2*.

A. Related work on Web service testing

Some test purpose-based methods dealing with Web services can be found in literature. These ones propose to transform and adapt an initial specification to be used with existing test purpose-based techniques [2], [3].

These test purpose-based method assume having an existing test purpose set constructed manually. Few works have also proposed automatic test purpose generation techniques: for instance, in [4], some generation techniques are proposed for distributed systems by identifying significant action sequences of the distributed components. From each sequence, a test purpose is generated. To our knowledge, none method has been given for service-oriented applications. However, these ones are composed of specific properties, e.g., SOAP faults, operations with data or exceptions. Therefore,these applications require new test purpose generation techniques which take into consideration these properties. We introduce, in this paper, some of these techniques for stateful Web services, which aim at testing the operation existence, the critical states and the exception handling. Then, we define a synchronous product to achieve the test case selection.

III. The advantages offered by the SOAP environment for testing

Web services are deployed in specific environments, e.g., SOAP for SOAP Web services, to structure messages in an interoperable manner and to manage operation invocations. In particular, the SOAP environment consists in a SOAP layer which serializes messages with XML and of SOAP receivers (SOAP processor + Web services) [7] which is software, in Web servers, that consumes messages. The SOAP processor is a Web service framework part which represents an intermediary between client applications and Web services and which serializes/deserializes data and calls the corresponding operations.

SOAP processors complete Web service behaviours by adding new messages, called SOAP faults, which give details about the faults raised in the server side. They return SOAP faults composed of the causes "Client" or "the endpoint reference not found" if services or operations do not exit. SOAP processors also generate SOAP faults when a service instance triggers exceptions. In this case, the fault cause is equal to the exception name. However, exceptions correctly managed in the specification and in the service code (with try...catch blocks) are distinguished from the unhandled ones since a correct exception handling produces SOAP faults composed of the cause *SOAPFaultException* only. So, SOAP faults can also be used to test whether the exception handling is correct by identifying the received causes. Consequently, taking into account these messages while generating test purposes sounds very interesting to check the satisfaction of some service properties. So, we propose to augment the specification with SOAP faults. We denote $(soapfault, cause)$ a SOAP fault where the variable *cause* is the reason of the SOAP fault receipt.

Let $S =< L, l_0, V, V_0, I, \Lambda, \rightarrow>$ be a Web service specification. S is completed by means of the algebraic operation $addsoap$ in S which completes the specification with SOAP faults as stated previously. The result is a new STS $S^\uparrow$. The operation $addsoap$ is defined as follow: $addsoap$ in $S =_{def} S^\uparrow =< L_{S^\uparrow}, l_0, V, V_0, I, \Lambda_{S^\uparrow}, \rightarrow_{S^\uparrow}>$ where $L_{S^\uparrow}$, $\Lambda_{S^\uparrow}$ and $\rightarrow_{S^\uparrow}$ are defined by the following rules:

- $$\frac{\dfrac{l_1 \xrightarrow{?opReq(p),\varphi,\varrho}_S l_2, l_1 \xrightarrow{?op'Req(p),\varphi',\varrho'} l \notin \rightarrow_S,}{l_1 \xrightarrow{?op'Req(p),\emptyset,\emptyset}_{S^\uparrow} l', l' \xrightarrow{!a(p),\varphi,\emptyset}_{S^\uparrow} l, \varphi = [a(p) \neq} \quad l' \notin L_S}{(soapfault,\text{"}CLIENT\text{"}) \wedge a(p) \neq (soapfault, \text{"the endpoint..."})]}$$
- $$\frac{l \xrightarrow{?opReq(p),\varphi,\varrho}_S l', \varphi' = \bigwedge_{l' \xrightarrow{!opResp_i(r_i),\varphi_i,\varrho_i}_S l'_i} \neg\varphi_i}{l' \xrightarrow{!(soapfault,cause),\varphi',\emptyset}_{S^\uparrow} l}$$

The first rule completes the initial specification on the input set by assuming that each unspecified operation request returns a SOAP fault message. The second rule completes the output set by adding, after each transition modelling an operation request, a transition labelled by a SOAP fault. Its guard corresponds to the negation of the guards of transitions modelling responses. A completed specification is illustrated in Figure 1(a) (with solid and dashed transitions). Note that unhandled exceptions are caught by specific exceptions called *unhandledException* in Java or C# and translated later by SOAP faults composed of the *unhandledException* cause. So, unhandled exceptions are supported by our work since we differentiate them from the SOAPFault exceptions thanks to the received SOAP faults.

IV. Automatic Test Purpose generation methods

Although test purposes sound interesting to reduce test costs, these ones also raise an important drawback since they are usually formulated manually. So, we contribute to solve this issue by introducing some automatic generation techniques for Web services. We propose three test purpose generation approaches which aim at testing the operation existence, the critical locations, and the exception handling.

A. Operation existence testing

This approach generates test purposes for testing whether operations in $\mathcal{OP}(S^{\uparrow})$, with $S^{\uparrow}$ an STS specification, are implemented and can be invoked. With the completion of the specification, detailed in the previous section, it becomes possible to test the existence of any operation, even those which do not return any response, i.e., any observable reaction. Indeed, if an operation is not implemented as it is described in the specification, the SOAP processor will return a SOAP fault composed either of the cause "Client" or of the cause "the end point reference not found". So, for a specification $S^{\uparrow} =< L_{S^{\uparrow}}, l0_{S^{\uparrow}}, V_{S^{\uparrow}}, V0_{S^{\uparrow}}, I_{S^{\uparrow}}, \Lambda_{S^{\uparrow}}, \rightarrow_{S^{\uparrow}} >$, the test purpose set is given by:
$TP = \bigwedge_{op \in \mathcal{OP}(S^{\uparrow})} \{tp =< L, l_0, V, V0, I_{S^{\uparrow}}, \Lambda_{S^{\uparrow}}, \rightarrow >$ where $\rightarrow = \{l_0 \xrightarrow{?opReq(p),\emptyset,\emptyset} l_1, l_1 \xrightarrow{!a(p),\varphi,\emptyset} l_2$, with $\varphi = [a(p) \neq (soapfault, "Client") \wedge a(p) \neq (soapfault,$ "the end point reference not found"$)]\}$

The specification of Figure 1(a) is composed of two operations, so we obtain two test purposes. These ones will be synchronized later with the specification to test any operation invocation.

B. Critical location testing

The second technique aims at testing the specification critical locations. This method is especially suitable when the specification locations have a precise meaning. It is not obvious to set which location is critical since no general and formal definition is given in literature. So, in this paper, we suggest that the critical locations are those the most potentially encountered in the acyclic specification paths. Algorithm 1, derived from the DFS (Depth First path Search) one, returns the critical location set CS, from a specification. Then, for each critical location $l \in CS$, we construct test purposes to test all the outgoing transitions of l.

The test purpose set, expressed below, is composed of specification paths finished by output actions to observe the implementation reactions while testing. For a specification $S^{\uparrow} =< L_{S^{\uparrow}}, l0_{S^{\uparrow}}, V_{S^{\uparrow}}, V0_{S^{\uparrow}}, I_{S^{\uparrow}}, \Lambda_{S^{\uparrow}}, \rightarrow_{S^{\uparrow}} >$, the test purpose set is given by:
$TP = \bigwedge_{l \in CS} \{tp =< L, l_0, V, V0, I_{S^{\uparrow}}, \Lambda_{S^{\uparrow}}, \rightarrow >$ where $\rightarrow$ is constructed with the following rules:

- $$\frac{l \xrightarrow{!a(p),\varphi,\varrho}_{S^{\uparrow}} l', a(p) \neq \delta}{l_0 \xrightarrow{!a(p),\varphi,\phi(\varrho)} l'}$$
- $$\frac{l \xrightarrow{?a(p),\varphi,\varrho}_{S^{\uparrow}} l', p = l' \xrightarrow{a_1(p),\varrho_1,\varphi_1}_{S^{\uparrow}} l'_1 \ldots}{l_0 \xrightarrow{?a(p),\varphi,\phi(\varrho)} l', l' \xrightarrow{a_1(p),\varphi_1,\phi(\varrho_1)} l'_1 \ldots}$$
$$\frac{l'_{n-1} \xrightarrow{a_n(p),\varphi_n,\varrho_n}_{S^{\uparrow}} l'_n, a_n(p) \in \Lambda^O_{S^{\uparrow}} / \{\delta\}}{l'_{n-1} \xrightarrow{a_n(p),\varphi_n,\phi(\varrho_n)} l'_n}$$

In both rules, we use a renaming function $\phi : V \rightarrow V'$, $\phi(v) \rightarrow v'$ to obtain exclusive test purpose internal variables. The first rule is used when an outgoing transition, from a critical location, is labelled by an output. In this case, this transition is added to the test purpose. The second rule is used when a transition is labelled by an input. The test purpose is completed with this transition followed by a specification path finished by an output.

Algorithm 2, given below, constructs a test purpose set from one critical location l. For each outgoing transition t of l, if t is labelled by an output action then t is a test purpose (rule R_1). Otherwise, we extract a path p with the *Cover* procedure such that the test purpose $t.p$ is finished by a transition labelled by an output action (rule R_2).

In the specification of Figure 1(a), we have two critical locations req and $cart$. For each, one test purpose is constructed, with the previous rules, whose purpose is to test all the outgoing transitions with paths finished by an output action. For instance, for the req location, we obtain a straightforward test purpose composed of one transition labelled by !b and another one labelled by !c.

Algorithm 1: Critical location search

```
1  Critical(STS,location);
   input  : An STS $S =< L, l_0, V, V0, I, \Lambda, \rightarrow >$, the
            initial location $l_0$
   output: A location set $CL$
2  $\forall t \in \rightarrow, label(t)$ :="UNEXPLORED";
3  foreach $t = (l, l_i, a_i, \varrho_i, \varphi_i) \in OutgoingTransition(l)$
   do
4      if Label(t) == "UNEXPLORED" then
5          Label(t):="VISITED";
6          Count(l):=Count(l)+1;
7          Critical($S, l_i$);
8      else
9          Count($l_i$):=Count($l_i$)+1 ;
10 $CL := \{$location $l \mid$ Count($l$) $\leq \frac{\sum_{l_i \in L} Count(l_i)}{card(L)}\}$;
```

C. Exception handling testing

As described in Section III, SOAP processors return SOAP faults when exceptions are triggered in a Web ser-

Algorithm 2: Test purpose generation dedicated to critical locations

```
1  TPgen(STS,location);
   input  : An STS
            $S^\uparrow =< L_{S^\uparrow}, l0_{S^\uparrow}, V_{S^\uparrow}, V0_{S^\uparrow}, I_{S^\uparrow}, \Lambda_{S^\uparrow}, \to_{S^\uparrow}>$,
            a critical location $l \in L_{S^\uparrow}$
   output : A test purpose
            $tp =< L, l0, V, V0, I_{S^\uparrow}, \Lambda_{S^\uparrow}, \to>$
2  $\forall t \in \to_{S^\uparrow}, label(t)$ :="UNEXPLORED";
3  foreach $t_i = l \xrightarrow{a(p),\varphi,\varrho} l_i \in \to_{S^\uparrow}$ do
4      if $a(p) \in \Lambda^O_{S^\uparrow}$ then
5          $\to := \to \cup \{l_0 \xrightarrow{a(p),\varphi,\phi(\varrho)} l_i\}$;
6      else
7          $p' := \emptyset$; $Cover(l_i, p')$;
           $\to := \to \cup \{l_0 \xrightarrow{a(p),\varphi,\phi(\varrho)} l_i.p'\}$;
8  $Cover$(location $l$, path $p$);
9  if $\exists l \xrightarrow{a(p),\varphi,\varrho} l' \in \to_{S^\uparrow}$ with $a(p) \in \Lambda^O_{S^\uparrow}$ then
10     $p := p.l \xrightarrow{a,\varphi,\varrho} l'$; $l := null$;
11 else
12     foreach $t_i = l \xrightarrow{a(p),\varphi,\varrho} l_i \in \to_{S^\uparrow}$ labelled by "UNEXPLORED" do
13         $label(t_i)$ :="VISITED"; $Cover(l_i, p.t_i)$;
14         if $l == null$ then
15             break;
16         $label(t_i)$ :="UNEXPLORED";
```

vice operation at runtime. SOAP processors also enable to differentiate the exceptions resulting of unexpected Web service crashes from those which are thrown in Web service operations (with try...catch blocks for instance). In the last case only, we obtain SOAP faults composed of the "SoapFaultException" cause.

With the specification completion described in Section III, we can construct test purposes to test whether the exception handling is correctly implemented and not managed by SOAP processors. However, to trigger exceptions, test purposes must be formulated over predefined value sets, that we denote $U(t)$. These ones are composed of unusual values well known for relieving bugs, for any simple or complex type t. For instance, $U(string)$ is composed of the values &", "\$", null or "_", which usually trigger exceptions. For a specification $S^\uparrow =< L_{S^\uparrow}, l0_{S^\uparrow}, V_{S^\uparrow}, V0_{S^\uparrow}, I_{S^\uparrow}, \Lambda_{S^\uparrow}, \to_{S^\uparrow}>$, the test purpose set is given by:

$$TP = \bigwedge_{l \xrightarrow{?opReq(p),\varphi,\varrho}_{S^\uparrow} l'} \{tp =< L, l_0, V, V0, I_{S^\uparrow}, \Lambda_{S^\uparrow}, \to> \text{ where } \to = \{l_0 \xrightarrow{?opReq(p),\varphi',\phi(\varrho)} l_1, l_1 \xrightarrow{(!soapfault,"SOAPFaultException"),\emptyset,\emptyset} l_2 \text{ with } \varphi' = \varphi \wedge p = (p_1, ..., p_n) \in U(type(p_1)) \times ... \times U(type(p_n))\}$$

The specification of Figure 1(a) contains two operation requests. If we suppose that $card(U(type(p_1)) \times ... \times U(type(p_n))) = n$, we obtain at most $2n$ test purposes. It is observed that the larger the unusual values sets, the larger the test purpose set will be. To limit it, instead of using a cartesian product, we have chosen to use pairwise testing [8] which helps to reduce the coverage of the variable domain by constructing discrete combinations for pair of parameters only.

V. Testing methodology

Each test purpose is synchronized with the specification to produce products, formalized with STSs, which combine the specification behaviour with the test purpose properties. We extract from these synchronous products complete paths (from their initial location until a final one) with an algorithm which performs a reachability analysis to check whether the guards of each path can be satisfied to guarantee its execution. These paths are also completed to express the incorrect (unspecified) behaviour. We obtain test cases ended by locations labelled by $pass, fail, inconclusive$ which represent the test case local verdict. These ones are finally translated into an XML format to be used with the Soapui tool. Each of these steps are described in detail below.

A. Synchronous product

A test purpose represents a test requirement which should be met in the implementation. To test this statement, both the specification and the test purpose are synchronized to produce paths which model test purpose runs with respect to the specification.

Similarly to the specification $S^\uparrow =< L_{S^\uparrow}, l0_{S^\uparrow}, V_{S^\uparrow}, V0_{S^\uparrow}, I_{S^\uparrow}, \Lambda_{S^\uparrow}, \to_{S^\uparrow}>$, a synchronous product $SP = S^\uparrow \times TP$, with $TP =< L_{TP}, l0_{TP}, V_{TP}, V0_{TP}, I_{S^\uparrow}, \Lambda_{S^\uparrow}, \to_{TP}>$ a test purpose, is defined as an STS $SP =< L_{S^\uparrow} \times L_{TP}, l0_{S^\uparrow} \times l0_{TP}, V_{S^\uparrow} \cup V_{TP}, V0_{S^\uparrow} \cup V0_{TP}, \Lambda_{S^\uparrow}, \to_{SP}>$, where the transition relation $\to_{SP}$ is defined with the following rules. The R_2 and R_3 rules perform the product of one specification transition with one test purpose one by synchronizing actions, variables updates and guards. We have written the specific rule R_3 for output actions to add locations labelled by *inconclusive*. This rule yields two transitions: the first one is composed of a guard satisfying both the specification and the test purpose ones ($\varphi_i \wedge \varphi'_i$). The second transition, ended by an *inconclusive* location, is composed of the guard $\varphi_i \wedge \neg\varphi'_i$ which satisfies the specification transition but not the test purpose one. So, reaching such an *inconclusive* location during the tests means that the test purpose transition is not satisfied although the specification is not faulty.

$$R_1 : \quad \frac{(l_i,l_{i'})\in L_{SP}, l_i \xrightarrow{a(p),\varphi_i,\varrho_i}_{S\uparrow}, l_{i'} \xrightarrow{b(p),\varphi_{i'},\varrho_{i'}}_{TP} l_{j'}}{(l_i l_{i'}) \xrightarrow{a(p),\varphi_i,\varrho_i}_{SP} (l_j l_{i'})}$$

$$R_2 : \quad \frac{\dfrac{(l_i,l_{i'})\in L_{SP}, l_i \xrightarrow{?a(p),\varphi_i,\varrho_i}_{S\uparrow} l_j, l_{i'} \xrightarrow{?a(p),\varphi_{i'},\varrho_{i'}}_{TP} l_{j'}}{(\exists x \in D_{(I_{S\uparrow}\cup V_{S\uparrow}\cup V_{TP})} \varphi_i \wedge \varphi_{i'}(x) \models \text{true})}}{(l_i l_{i'}) \xrightarrow{?a(p),\varphi_i\wedge\varphi_{i'},\varrho_i\wedge\varrho_{i'}}_{SP} (l_j l_{j'})}$$

$$R_3 : \quad \frac{\dfrac{(l_i,l_{i'})\in L_{SP}, l_i \xrightarrow{!a(p),\varphi_i,\varrho_i}_{S\uparrow} l_j, l_{i'} \xrightarrow{!a(p),\varphi_{i'},\varrho_{i'}}_{TP} l_{j'}}{(l_i l_{i'}) \xrightarrow{!a(p),\varphi_i\wedge\varphi_{i'},\varrho_i\wedge\varrho_{i'}}_{SP} (l_j l_{j'})} \quad (\exists x \in D_{(I_{S\uparrow}\cup V_{S\uparrow}\cup V_{TP})} \varphi_i \wedge \varphi_{i'}(x) \models \text{true})}{(l_i l_{i'}) \xrightarrow{!a(p),\varphi_i\wedge\neg\varphi_{i'},\varrho_i\wedge\varrho_{i'}}_{SP} (l_j inconlusive)}$$

The synchronous product of the test purpose given in Figure 1(b) and the completed specification is depicted in Figure 3. The yellow transition, which reaches an inconclusive location, models a response which does not contradict the specification but does not satisfy the test purpose. Transitions labelled by ?f1 !f2 still belong to the product. They represent a *CartAdd* operation request which may be called before the *CartCreate* operation given in the test purpose.

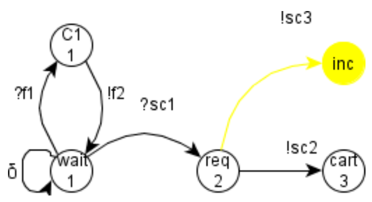


Figure 3. A synchronous product

B. Test case extraction

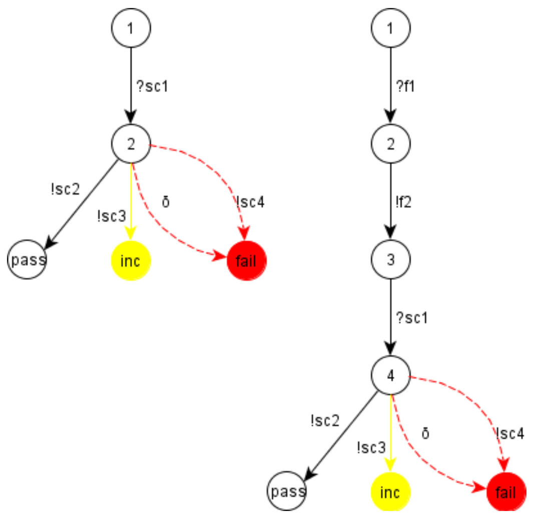


Figure 4. The final test cases

Final test cases are constructed with the following steps from the previous synchronous product SP.

- **Synchronous product path extraction with reachability analysis:** acyclic paths are extracted from SP with Algorithm 3. This one computes a set P of SP paths p. With the *Cover* subroutine (lines 4-14), it explores SP with backtracking and solves the constraints of p with the *solving* subroutine to ensure that p may be completely executed. *solving* takes a path p and returns a variable assignment ϱ_0 which satisfies the complete execution of p. If the constraint solvers [9], [10] cannot compute a value set allowing to execute p, then *solving* subroutine returns an empty set (lines 19-20). We use the solvers in [9] and [10] which work as external servers that can be called by the test case generation algorithm. The solver [10] manages "String" types, and the solver [9] manages most of the other simple types. In practice, to reduce the time required for solving guards and to prevent from an explosion of values possibilities, we assume that the variable domains are limited by using value sets extracted from database for instance.
- **"pass" verdict addition:** for each path $p \in P$, the final locations not already labelled by "inconclusive" are labelled by "pass" which means that some behaviours modelled by test purposes has been reached,
- **Incorrect behaviour completion:** each path $p \in P$ is completed on the incorrect response set: $\forall l \in L$ such that l has the outgoing transitions $l \xrightarrow{!opResp(r_1),\varphi_1,\varrho_1} l_1, ..., l \xrightarrow{!opResp(r_k),\varphi_k,\varrho_k} l_k$, we add: (1) $l \xrightarrow{\delta,\emptyset,\emptyset} fail$, (2) $l \xrightarrow{!opResp(r),\varphi,\emptyset} fail$, $\varphi = [\neg(\varphi_1 \vee ... \vee \varphi_k)]$. (1) δ models the location quiescence i.e., the lack of observation. We suppose that if no response is observed after a defined timeout Tmax, then the Web service under test is faulty. (2) If the called operation does not return an expected response, then the implementation does not satisfy both the test purpose and the specification. Thus, a fail verdict is reached too. Note that when an operation is called, we cannot observe the response provided by another operation. So, this case is not considered in this completion.

Final test cases are given in Figure 4. We obtain two acyclic paths from the previous synchronous product enhanced with the possible verdicts.

C. Test execution and verdict

The implementation under test I is assumed behaving like an LTS semantics, composed of valued transitions (Section II). We assume that there is no security constraint or firewall between the tester and the implementation, which modifies the SOAP messages and thus the implementation behaviour. To produce a verdict on the test purpose satisfaction, the tester executes each test case by traversing the test case tree: it successively calls an operation with parameters and waits for a response from I while following the corresponding

Algorithm 3: Testcase(STS): P;

```
input : An STS $SP =< L_{SP}, l_{0SP}, V_{SP}, V0_{SP}, I_{SP}, S_{SP}, \rightarrow_{SP} >$
output: A set $P$ of $STS$ paths
1  $\forall t \in \rightarrow_{SP}, label(t)$ :="UNEXPLORED";
2  $p := \emptyset$;
3  $Cover(l0_{SP}, p, 0)$;

4  $Cover$(location $l$, path $p$, int $n$);
5  if $\exists (l, l', a, \varrho, \varphi) \in \rightarrow_{SP}$ labelled by "UNEXPLORED" then
6      foreach $t_i = (l, l_i, a_i, \varrho_i, \varphi_i) \in \rightarrow_{SP}$ labelled by "UNEXPLORED" do
7          if $Solving(p.t_i) \neq \emptyset$ then
8              $label(t_i)$ :="VISITED";
9              $Cover(l_i, p.(l(n), l_i(n+1), a_i, \varrho_i, \varphi_i), n+1)$;
10             $label(t_i)$ :="UNEXPLORED";
11 else
12     $\varrho := Solving(p)$;
13     $V0_p = \varrho$ //$V0_p$ is the variable initialization of $p$;
14     $P := P \cup p$ ;

15 $Solving(path\ p) : \varrho$;
16 $p = (l_0, l_1, a_0, \varphi_0, \varrho_0)...(l_k, l_{k+1}, a_k, \varphi_k, \varrho_k)$;
17 $c = \varphi_0 \wedge \varphi_1(\varrho_0) \wedge ... \wedge \varphi_k(\varrho_{k-1})$;
18 $(x_1, ..., x_n) = solver(c)$ //solving of the guard $c$ composed of the variables $(X_1, ..., X_n)$ such that $c(x_1, ..., x_n)$ true;
19 if $(x_1, ..., x_n) == \emptyset$ then
20     $\varrho := \emptyset$
21 else
22     $\varrho := \{X_1 := x_1, ..., X_n := x_n\}$
```

branch. When a branch is completely executed, a local verdict is reached. For a test case t, we denote the local verdict $v(t) \in \{$pass, inconclusive, fail$\}$. The final verdict is given by:

Definition 2 *Let I be a Web service under test, P be a test purpose set and TC be a generated test case set. The verdict of the test over P, denoted $Verdict(I/P)$ is*

- *pass, if for all $t \in TC, v(t) = pass$. The pass verdict means that test purposes in P are satisfied,*
- *inconclusive, if it exists $t \in TC, v(t) = inconclusive$ and for all $t \in TC, v(t) \neq fail$. This verdict means that some test purposes in P are not satisfied although the implementation does not sound faulty,*
- *fail, otherwise. At least on test purpose in P is not satisfied and the implementation is faulty.*

VI. EXPERIMENTATION

	Existence	Critical locations	Exception handling
test purposes	22	2	22
test cases	44	22	210
fail verdicts	0	0	39

Figure 5. Test results on the Amazon AWSECommerceService Service

At the moment, we have implemented an incomplete tool which performs the test purpose generation from a completed STS and the synchronous products between the specification and test purposes only. So, test cases are not generated from synchronous products but are extracted manually. To experiment them on real Web services, test cases are extracted and written into the Soapui format. So, these ones can be executed with the Soapui tool [6] which aims to experiment Web services with unit test cases.

We applied this preliminary tool on several Web services to experiment the test purpose generation. Figure 5 describes the results obtained for the Amazon AWSECommerceService (09/10 version) which is a representative sample because it is composed of a large operation set (22 operations) and of many data structures. We limited the test purpose number to 10 per operation for the exception handling method. We obtained fail verdicts only for the exception handling tests. Indeed, we obtained some SOAP faults composed of the cause *Client*, meaning that the requests are incoherent although the test cases satisfy the specification. We also received unspecified messages corresponding to errors composed of a wrong cause. For instance, instead of receiving SOAP faults, we obtained the response *"Your request should have at least 1 of the following parameters: AWSAccessKeyId, SubscriptionId"* when we called the operation CartAdd with a quantity equal to "-1", or when we searched for a "Book" type instead of the "book" one, whereas the two parameters AWSAccessKeyId, SubscriptionId were right.

In comparison with the other test purpose-based methods for service-oriented applications [2], [3] or tools, our approach takes into account the SOAP environment for testing. This one generates messages which help to conclude if operations exist as it is stated in the specification and which help to identify the exceptions resulting of unexpected Web service crashes from those which are thrown in Web service operations. These features helped to detect the incorrect SOAP faults, composed of the cause *Client* in the previous experimentation. These errors cannot be detected by the previous methods. But the major benefit of this approach concerns the automatic generation of test purposes. Most of the test purpose-based method assume having an existing test purpose set, constructed manually. As stated earlier, this manual construction requires time and is difficult when the system is large.

VII. Conclusion

We have presented some automatic test purpose generation methods dedicated to Web services, which aim to test the operation existence, the critical locations, and the exception handling. Then, we have defined a synchronous product of the test purpose with the specification to construct test cases, which are finally translated into XML and then executed by means of SOAPUI.

An immediate line of future work is to take into consideration test purposes describing incorrect behaviours. Such test purposes may be composed of properties which do not belong to the specification. These ones can be used for testing behaviours which should be met in the implementation but also behaviours which should not. With such test purposes, we could propose new generation methods for robustness or security testing.

References

[1] J. García-Fanjul, J. Tuya, and C. de la Riva, "Generating test cases specifications for compositions of web services," in *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, A. Bertolino and A. Polini, Eds., Palermo, Sicily, ITALY, June 9th 2006, pp. 83–94.

[2] M. Lallali, F. Zaidi, A. Cavalli, and I. Hwang, "Automatic timed test case generation for web services composition," in *The 6th IEEE European Conference on Web Services (ECOWS'08)*, I. C. S. Press, Ed., Dublin, November 2008, 53–63.

[3] T.-D. Cao, P. Felix, and R. Castanet, "Wsotf: An automatic testing tool for web services composition," in *Proceedings of the 2010 Fifth International Conference on Internet and Web Applications and Services*. Washington, DC, USA: IEEE Computer Society, 2010, pp. 7–12. [Online]. Available: http://dx.doi.org/10.1109/ICIW.2010.9

[4] O. Henniger, M. Lu, and H. Ural, "Automatic generation of test purposes for testing distributed systems," in *FATES*, ser. Lecture Notes in Computer Science, A. Petrenko and A. Ulrich, Eds., vol. 2931. Springer, 2003, pp. 178–191.

[5] L. Frantzen, J. Tretmans, and T. Willemse, "Test Generation Based on Symbolic Specifications," in *Formal Approaches to Software Testing – FATES 2004*, ser. Lecture Notes in Computer Science, J. Grabowski and B. Nielsen, Eds., no. 3395. Springer, 2005, pp. 1–15. [Online]. Available: http://www.cs.ru.nl/~lf/publications/FTW05.pdf

[6] Eviware, "Soapui," 2011, http://www.soapui.org/.

[7] W.-I. organization, "Web services basic profile," 2006, http://www.ws-i.org/docs/charters/WSBasic_Profile_Charter2-1.pdf.

[8] M. B. Cohen, P. B. Gibbons, and W. B. Mugridge, "Constructing test suites for interaction testing," in *Proc. Intl. Conf. on Software Engineering (ICSE)*, 2003, pp. 38–48.

[9] N. Een and N. Sörensson, "Minisat," 2003, http://minisat.se.

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