Stateful web service robustness

Sébastien Salva, Issam Rabhi

Statefull web service robustness

Sebastien Salva [1] Issam Rabhi [2]

Research Report LIMOS/RR-09-09

14 décembre 2009

1. sebastien.salva@u-clermont1.fr LIMOS - Campus des cézeaux - 63173 Aubière
2. rissam@isima.fr LIMOS - Campus des cézeaux - 63173 Aubière

Abstract

Web Services fall under the so-called emerging technologies category and are getting more and more used for Internet applications or business transactions. Since web services are often the foundation of large applications, they need to be reliable and robust. So, we propose in this paper, a robustness testing method of statefull web services, modeled with STS (Symbolic Transition Systems). We analyze the web service observability and the hazard effectiveness in a SOAP environment. Then, we propose a test case generation method based on the two hazards "Using unusual values" and "Replacing /Adding operation names", which are the only ones which can be applied. The Amazon E-commerce web service is taken as example.

Keywords: Robustness testing, Statefull web services, STS, test architecture

Résumé

Les services web font partis des techniques de développement dites émergentes et sont de plus en plus utilisés aujourd'hui pour construire des applications. De part leurs natures, ces web services doivent être fiables et robustes. C'est pourquoi nous proposons, dans cet article, une méthode de test de robustesse de services web persistants, modélisés par des spécifications symboliques. Une partie de cet article est réservée à l'analyse de la robustesse en se focalisant sur l'environnement SOAP, qui réduit le contrôle et l'observation des messages transmis. Nous en déduisons notamment les aléas qui sont pertinents pour le test et ceux qui sont ignorés et bloqués par les processeurs SOAP. Nous générons les cas de test en complétant la spécification et en injectant, dans les cas de test, des valeurs inhabituelles prédéfinies.

Mots clés : test de robustesse, services web persistants, STS, architecture de test

Acknowledgement / Remerciements

Cette étude a été effectuée dans le cadre du projet Webmov http ://webmov.lri.fr.

1 Introduction

The web service paradigm is now well established in companies for developing business applications. These self-contained components offer many advantages such as, providing interoperability between heterogeneous applications, externalizing functional code in a standardized way, or composing choreography and orchestration processes. Interoperability is ensured by standards proposed by the W3C and the WS-I consortiums. Especially, the WS-I basic profile gathers the SOAP protocol, which models how invoking a web service with XML messages, and the WSDL language, which is used to describe web service interfaces.

Web services constitute often the foundation of large and complex applications. Consequently, quality and so trustability are essential criteria which must be taken into account while developing them. Trustability can be achieved only by following quality processes, like the CMMI process (Capability Maturity Model Integration). These processes are based on testing activities performed during the whole software life cycle. Among them, the robustness testing is a required step when dealing with web services. Indeed, the latter are distributed in nature, and can be used by different and heterogeneous client applications. So, they need to behave correctly despite the receipt of unspecified events, called *hazards*.

The purpose of this paper is to tackle the statefull web service robustness. Unlike stateless ones, such services are persistent through a session and have an internal state which evolves over the sequences of operation calls. For instance, all the web services using shopping carts or beginning with a login step are statefull. The Amazon E-commerce web service (AWSECommerceService), which is taken as example in this paper, is one of them. We consider web services as *black boxes*, from which only the SOAP messages (requests and responses) are observable (we don't have access to the code). So, we model the web service state with a symbolic specification describing the different states, the called operations and the associated data.

Black box web service testing faces a challenging issue which concerns the lack of observability, involved by SOAP. A message which is usually directly observable, like a classical response or a fault, is either no more observable or is encapsulated and spread to the client over the SOAP protocol. For instance, according to the SOAP 1.2 specification, exceptions, in object oriented programming, ought to be translated into XML elements called SOAP faults. But, to be observed, this feature needs to be implemented by hand in web services.

Consequently, we begin to analyze the web service observability in the presence of hazards to determine those which are relevant for testing. And

we show that only few hazards are really interesting, because most of them are blocked by SOAP processors and are not given to the web service itself. We also analyze the responses obtained. These analysis lead to the test case generation method. This one can be summarized by two main steps. First, the specification is completed to add all the operations which can be called and to model the unexpected behaviors. Then, test cases are constructed from the completed specification, by injecting into paths predefined values, well known for relieving bugs. Test cases are then executed with a specific test platform, which has been implemented in an academic tool.

This paper is structured as follows: section 2 provides an overview of the web service paradigm. We give some related works about web service testing and the motivations of our approach. Section 3 analyzes the web service robustness over the SOAP layer. Section 4 describes the testing method: we detail the test case generation and a testing framework. We also describe the results obtained from the Amazon E-commerce web service. Finally, section 5 gives some perspectives and conclusions.

2 Web Service Overview

2.1 Web service

Web services are "self contained, self-describing modular applications that can be published, located, and invoked across the web" [1]. To ensure and improve web service interoperability, the WS-I organization has proposed profiles, and especially the WS-I basic profile [2], composed of four major axes:

- the *web service description*, expressed by WSDL (Web Services Description Language [3]), defines the web service interface by giving the available operations, their parameter/response types and the message structures by describing the complex types used within. WSDL is often used in combination with SOAP,
- the *definition and the construction of XML messages*, based on the Simple Object Access Protocol (SOAP) [4]. SOAP aims to invoke service operations (object methods) over a network by serializing/deserializing data (parameter operation and responses). SOAP takes place over different transport layers: HTTP is which mainly used for synchronous web service calls, or SMTP which is often used for asynchronous calls,
- the *discovery of the service* in UDDI registries. Web service descriptions are gathered into UDDI (Universal Description, Discovery Integration [5]) registries, which can be consulted manually or automatically by

using dedicated APIs to find dynamically specific web services,
– the *service security*, which is obtained by using the HTTPS protocol or XML encryption.

In this paper, we consider black box web services, from which we can only observe SOAP messages. Other messages, such as database connections and the web service internal code are unknown. So, the web service definition, given below, describes the available operations, the parameter and response types. We also use the notion of SOAP fault. As defined in the SOAP v1.2 protocol [4], a SOAP fault is used to warn the client that an error has occurred. A SOAP fault is composed of a fault code, of a message, of a cause, and of XML elements gathering the parameters and more details about the error. Typically, a SOAP fault is obtained, in object-oriented programming, after the raise of an exception by the web service.

Definition 2.1 *A web service WS is a component which can be called by a set of operations $OP(WS) = \{op_1, ..., op_k\}$, with op_i defined by $(resp_1, ..., resp_n) = op_i(param_1, ..., param_m)$, where $(param_1, ..., param_m)$ is the parameter type list and $(resp_1, ..., resp_n)$ is the response type list.*

For an operation op, we define $P(op)$ the set of parameter value lists that op can handle, $P(op) = \{(p_1, ..., p_m) \mid p_i$ is a value whose type is $param_i\}$. The set of response lists, denoted $R(op)$, is expressed with $R(op) = \{(r_1, ..., r_n) \mid r_j$ is a value whose the type is $resp_j\} \cup \{r \mid r$ is a SOAP fault$\} \cup \{\epsilon\}$. ϵ models an empty response (or no response).

The operation op corresponds to a Relation $op : P(op) \rightarrow R(op)$. We denote an invocation of this operation with $r = op(p)$ with $r \in R(op)$ and $p \in P(op)$.

The parameter/response types may be simple (integer, float, String...) or complex (trees, tabular, objects composed of simple and complex types...) and each one is either finite (integer...) or infinite (String...).

We model the web service state with STS (Symbolic Transition Systems [6]). STS offer a large formal background (definitions of implementation relations, test case generation algorithms,...) and are semantically close to UML state machines. Besides, some tools are proposed to translate an UML state machine to STS [7]. An STS is a tuple $< L, l_0, Var, var_0, I, S, \rightarrow >$, composed of symbols S: inputs, beginning with "?" are provided to the system, while outputs, (beginning with "!") are observed from it. An STS is composed of a variable set Var, initialized by var_0. Each transition $(l_i, l_j, s, \varphi, \varrho) \in \rightarrow$ from the state l_i to l_j, labeled by the symbol s, may update variables with ϱ and may have a guard φ on Var, which must be satisfied to fire the transition.

Furthermore, we denote $(x_1, ..., x_n) \models \varphi$, the satisfaction of the guard φ according to the values $(x_1, ..., x_n)$.

The specification example, given in figure 1, describes a part of the Amazon web service devoted for E-commerce (AWSECommerceService), which offers many features such as searching items, looking for item details, creating carts, purchasing... Note that we do not include all the parameters for readability reasons. A symbol table is given in figure 2. This specification uses a database whose a part, extracted from the Amazon documentation, is illustrated in figure 3.

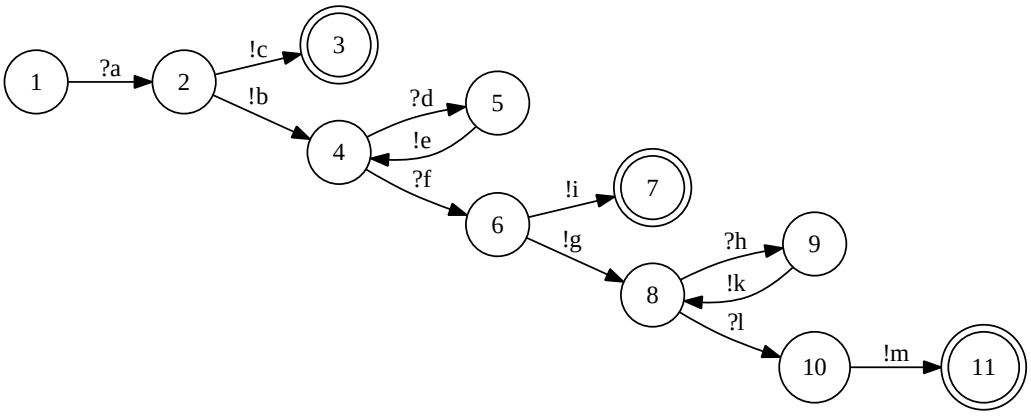

Figure 1: The Amazon AWSECommerceService specification

2.2 Related work on web service robustness testing

There are many research papers concerning web services testing, and some of them, focusing on robustness, are summarized below. Most of them consider stateless web services. Robustness is then tested by handling WSDL descriptions [8, 9, 10] or by injecting hazards into SOAP messages [11, 12].

In [8], web service robustness is automatically tested from WSDL descriptions. The method uses the Axis 2 framework to generate a class composed of methods allowing to call service operations. Then, test cases are generated with the tool "Jcrasher", from the previous class. Finally, the tool Junit is used to execute test cases. Another method is proposed in [9], which is based on the WSDL analysis to identify what faults could affect the robustness attribute and then test cases were designed to detect those faults. In [10], we also proposed a web service robustness testing method, which automatically assesses the stateless web service robustness by using WSDL description.

?a	ItemSearch<String AWSAccessKeyID,String SearchIndex,String KeyWords>
?d	ItemLookUp<String AWSAccessKeyID,String RequestID>
?f	CartCreate<String AWSAccessKeyID,String ItemASIN,Integer Quantity>
?h	CartAdd<String AWSAccessKeyID,String ItemASIN,Integer Quantity>
?l	Purchase<String CartId,String CustomerInfos>
!c	ItemSearchResponse<String Errors,String IsValid> [IsValid==”false” && AWSAccessKeyID ≠ BD.account.KeyID]
!b	ItemSearchResponse<String Items,String IsValid> [IsValid==”true” && AWSAccessKeyID==BD.account.KeyID]
!e	ItemLookUpResponse<String Items,String IsValid> [IsValid==”true” && AWSAccessKeyID==BD.account.KeyID]
!i	CartCreateResponse<String Errors,String IsValid> [IsValid==”false” && (Quantity≥ BD.ItemASIN.Quantity || ItemASIN≠BD.Item.ASIN)]
!g	CartCreateResponse<String CartId,String IsValid> [IsValid==”true” && Quantity¡BD.ItemASIN.Quantity && ItemASIN==BD.Item.ASIN]
!k	CartAddResponse<String IsValid> [IsValid==”true” && Quantity¡BD.ItemASIN.Quantity && ItemASIN==BD.Item.ASIN]
!m	PurchaseResponse<String IsValid> [IsValid==”true”]

Figure 2: Specification symbol table

account	
KeyID	
”ID”	
ItemASIN	
ASIN	Quantity
”43451”	”5”
”66405”	”30”

Figure 3: The Amazon AWSECommerceService Database

The method performs random testing improved with the use of the hazard "Using unusual values". A tool has been developed and tested on real web services.

In [11], web service robustness is tested by applying mutations on the request messages and by analyzing the obtained responses. In [12], fault injection techniques are employed to create diverse fault-trigging test cases in order to display possible robustness problems in the web-services code.

Comparing to the previous works, we focus on statefull web services, which have an internal state modeled with a symbolic specification. In the past, some works have been proposed on reactive system robustness, modeled with symbolic specifications too. However, the web service paradigm leads to new issues: many hazards are blocked by SOAP processors and message observability is reduced on account of SOAP. So, we begin to analyze the web service robustness in the presence of hazards and we analyze the SOAP responses to separate the web service behavior to the SOAP processor one. As a consequence, we propose a specific specification completion and a new test case generation algorithm.

3 Web service robustness study

As many works, referring to robustness testing, we consider that a web service is robust "if it is able to function properly in the presence of faults or stressful environments" [13]. This implies that a robust web service must satisfy its specification despite the presence of hazards. This sentence sounds classical, however, to give a verdict, we need to focus on the operation observability, and this is specific to web services.

As in the WS-I basic profile, we consider that a *receiver* in a web server is software that consumes a message (SOAP processor + web service). The SOAP processor is often a part of a more complete framework like Apache Axis or Sun Metro JAXWS.

3.1 Web service operation observability

We analyzed, in [10], the responses received from stateless web service operations, in the presence of hazards, to separate the responses generated by the web service itself to those produced by SOAP processors. An operation is called through the SOAP protocol, and this is concretely the couple (SOAP processor, web service) which is really requested. We have observed that SOAP processors affect the web service observability by generating responses instead of the web service itself, when it crashes. The same issue is raised

with statefull services. So, we need to separate the SOAP processor behavior to the web service one.

The WS-I basic profile enables to differentiate the SOAP faults generated by SOAP processors from those constructed by web services. When a exception is raised, the web service ought to generate itself a SOAP fault. In this case, the SOAP fault cause is always equal to "Remote Exception". Otherwise, SOAP faults are generated by SOAP processors.

We deduced from this analysis that a web service operation is robust if it returns a classical response or a SOAP fault whose the cause is equal to "RemoteException". This is formalized in following definition:

Definition 3.1 *Let WS be a web service. An operation $(resp_1, ..., resp_n) = op(param_1, ..., param_m) \in OP(WS)$, op is robust in the presence of the hazard "Using unusual values", if $\forall v \in P(op), r = op(v)$ with:*
- *$r = (r_1, ..., r_n)$ such as $r_i = resp_i$,*
- *or r is a SOAP fault composed of the cause "RemoteException".*

3.2 Statefull web service robustness analysis

We analyzed in [10], the hazards based on the operation parameter handling. We studied the following hazards: **Replacing parameter types, Inverting parameter types, Adding/injecting parameter types, Deleting parameter types** and **Using unusual values**. We deduced that the latter is the only one accepted by SOAP processors and really given to web services. The other ones are always blocked by SOAP processors and become unnecessary. This hazard, well-known in software testing [14], aims to call an operation $op(param_1, ..., param_m)$ with values $(p_1, ..., p_m) \in P(op)$, such as each parameter p_i has the type $param_i$. But these predefined values are unusual and may potentially reveal bugs. For instance, null, "", "$", "*" are some unusual "String" values.

We extend here this study for statefull web services. As previously, some hazards are unnecessary because they are blocked by SOAP processors. We have studied the following hazards: **Changing operation name, Replacing /Adding operation name**. Let WS be a web service and STS its specification:

- **Changing operation name**: this hazard aims to randomly modify an operation name $op \in OP(WS)$ to op_modif such as op_modif is not an existing operation ($op\_modif \notin OP(WS)$). When this hazard is put into practice, we always receive a SOAP fault composed of the cause "Client", which means that the client request is faulty. This hazard produces requests which are always blocked by SOAP processors since

these ones do not conform the WSDL description. So, because the test cannot be performed, we consider that this hazard is useless,

– **Replacing /Adding operation names**: Let l be a state of the STS specification with outgoing transitions $(l, l_1, "op_1(p_{11}, ..., p_{m1})", \varphi_1, \varrho_1)$, ..., $(l, l_n, "op_n(p_{1n}, ..., p_{mn})", \varphi_n, \varrho_n)$ modeling operations calls. If it exists an operation $(resp_1, ..., resp_n) = op(param_1, ..., param_m)$ which is not called from l ($op \notin \{op_1, ..., op_n\}$), this hazard aims to replace/add the call of an operation $op_i \in \{op_1, ..., op_n\}$ by op. Since this hazard satisfy the web service WSDL description, it is not blocked by SOAP processors. However, when calling op, we can only receive a response from op. We cannot just replace/add a name. For instance, if we replace the operation "itemSearch" to "AddCard", in our example of figure 6, we don't receive a response from "itemSearch" but from "AddCard". If the operation op is robust, as defined in the section 3.1, the expected response from op is either a classical response $(r_1, ..., r_n)$ where r_i has the type $resp_i$, or a SOAP fault whose the cause is "RemoteException". This hazard involves to complete the specification on the operation calls for each state. This modification is detailed in the test case generation section (see 4.2).

It exists of course other hazards based on the SOAP message modification, such as replacing the port name or modifying randomly the SOAP message. These hazards are usually used for testing web service compositions in order to observe partner behaviors. Concerning statefull web services, either the message random modification is equivalent to a previous hazard (parameter, operation modification) or gives an inconsistent SOAP message which is blocked by SOAP processors. Note that the WS-I basic profile does not allow overloading of operation so this hazard is not considered.

Consequently, the most relevant hazards for statefull web services are, calling operations with "using unusual values" and "replacing /adding operation names".

4 Statefull web service robustness testing method

Like many testing methods, we first set an assumption on the web service observability to improve the test efficiency. We suppose that web service operations do not return empty responses. Indeed, without response that is without observable data, we cannot conclude whether the operation is correct or crashes and is faulty.

Web service observable operation hypothesis: We suppose that each web service operation, described in WSDL files, returns a non empty response.

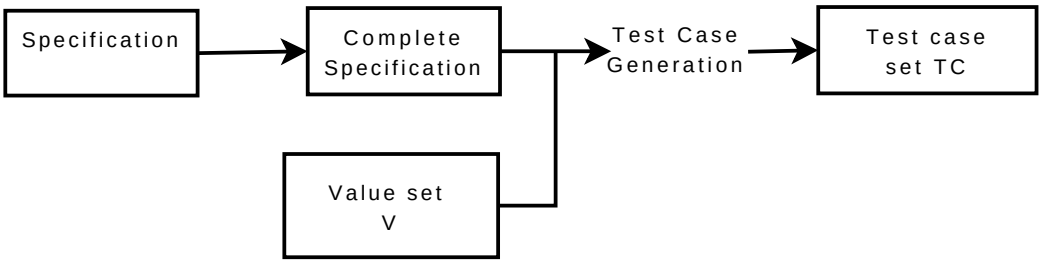

Figure 4: Test case generation

The test case generation method is illustrated in figure 4. This one is mainly based on the two hazards "Using unusual values" and "Replacing /Adding operation names" previously described. First, the specification states are completed on the operation calls to apply the hazard "Replacing /Adding operation names". It is also completed to model the incorrect behavior (incorrect responses) and the state quiescence (states blocked after a timeout). So, test cases, generated from the complete specification, will describe both the correct behavior which will lead to a pass verdict and also the incorrect behavior which will lead to a fail verdict (faulty implementation). Taking into account quiescence in web application testing is required since these ones may hang or crash without returning a response. According to the observability hypothesis previously given, we consider that web services are faulty when they are observed as blocked.

Test cases are generated with the "Using unusual values" hazard which is expressed by a set of values V. This set contains for each type, an XML list of values that we use for calling operations. We use predefined values well known in software testing for relieving bugs (section 3.1).

We denote $V(t)$ the set of specific values for the type t which can be a simple type or a complex one. Figures 4, and 4 show some values used for the types "String" and for "tabular of "simple-type". For a tabular composed of String elements, we use the empty tabular, tabulars with empty elements and tabulars of String constructed with $V(String)$.

In the following, we develop the specification completeness, the test case generation and the elaboration of the final verdict.

4.1 Specification Completeness

As stated previously, we complete the specification to apply the "Replacing /Adding operation name" hazard and to add the incorrect web service behavior.

```
<type id="String">
        <val value=null />
        <val value="" />
        <val value=" " />
        <val value="$" />
        <val value="*" />
        <val value="&" />
      <val value="hello" />
        <val value=RANDOM" />
        <!-- a random String-->
        <val value=RANDOM(8096)" />
</type>
```

(a) V(String)

```
<type id="tabular">
        <val value=null />
        <!-- an empty tabular-->
        <val value= null null />
        <!--tabular
        composed of two empty elts-->
        <val value= simple-type />
</type>
```

(b) V(tabular)

Figure 5:

Let WS be a web service and STS be its specification, with $STS =< L, l_0, Var, var_0, I, S, \rightarrow >$. STS is completed by using the following steps:

1. **Replacing conditions on database** with values,
2. **Adding "pass" verdict into the final states**, which means that to reach this final state, a correct behavior has to be executed,
3. **Operation call completion**: each state is completed to take into account the hazard "Replacing /Adding operation names". $\forall l \in L$ such as l has the outgoing transitions $(l, l_1, "op_1(p_{11}, ..., p_{m1})", \varphi_1, \varrho_1)$, $..., (l, l_n, "op_n(p_{1n}, ..., p_{mn}) ", \varphi_n, \varrho_n)$, we add: $\forall (resp_1, ..., resp_k) = op_i$ $(param_1, ..., param_m) \in OP(WS) \neq \{op_1, ..., op_n\}$, $(l, l_i, op_i(p_{i1}, ..., p_{im})$, $\emptyset, \emptyset)$, $(l_i, l, op_i\_return(r), \varphi_i, \emptyset)$, with $\varphi_i = [(r == (c, soapfault), c = RemoteException)||(r = (r_1, ..., r_k)$ has the type $(resp_1, ..., resp_k))]$. If, from a state, an unspecified operation is called and if either the obtained response has the expected type or if the response is a SOAP fault whose the cause is "RemoteException", then the web service has accepted the request and is robust (see Section 3.2).Then, it returns to the state preceding the request,
4. **Incorrect behavior completion**: for each state, the specification is completed on the incorrect response set: $\forall l \in L$ such as l has the outgoing transitions $(l, l_1, "op\_return(r_1)", \varphi_1, \varrho_1), ..., (l, l_n, "op\_return(r_n)$ $", \varphi_n, \varrho_n)$, we add: (1) $(l, fail, \delta, \emptyset, \emptyset)$, (2) $(l, fail, "op\_return(r)", \varphi, \emptyset)$,

with $\varphi = [\neg(\varphi_1||...||\varphi_n)]$. (1) δ models the state quiescence (blocking state). (2) If the called operation does not return a specified response, then the web service behavior is incorrect in the presence of hazards and the operation is not robust.

The STS of figure 6 illustrates a complete specification, obtained from the one given in figure 1. The dotted transitions represent the call of unspecified operations. Those illustrated with dashed lines model incorrect responses. The new symbols are given in figure 7.

4.2 Test case generation

Prior to describe the test case generation, we define a test case with:

Definition 4.1 *Let WS be a web service modeled by $STS =< L, l_0, Var, var_0, I, S, \rightarrow >$. A test case T is an STS tree where each final state is labeled by a verdict in $\{pass, fail\}$. Branches are labeled either by $op(v), \varphi, \varrho$ or by $op\_return(r), \varphi, \varrho$ or by δ where:*

- *$v \in P(op)$, is a list of parameter values used to invoke op,*
- *$r = (c, soap\_fault)$ is a SOAP fault composed of the cause c,*
- *$r = (r_1, ...r_m)$ is a response*
- *φ is a guard and ϱ an update on Var,*
- *δ represents the state quiescence.*

Test cases are constructed by using the following algorithm. This one generates traces in which hazards are injected. For a transition modelling the call of the operation *op*, the algorithm constructs a preamble to reach this transition (lines 4-5). A value set over V is constructed according the parameter types. If these ones are complex (tabular, object,...), we compose them with other types to obtain the final values. We also use an heuristic to estimate and eventually to reduce the number of tests according the number of tuples in $Value(op)$. Constraint solvers [15, 16] are also used to generate the values allowing the full preamble execution. We add $op(v_1, ..., v_m)$ after the preamble to call the operation *op* with unusual values $(v_1, ..., v_n)$ (line 9). Then, we concatenate all the executable traces reaching a final state labeled by a verdict (lines 10-11) (traces whose the conditions can be satisfied).

The constraint solvers allow to construct values satisfying the guards of a specification path and hence satisfying its execution. We use the solvers [15] and [16] which works as external servers that can be called by the test cases generation algorithm. The solver [16] manages "String" types, and the solver [15] manages most of the other simple types. Where the data type is

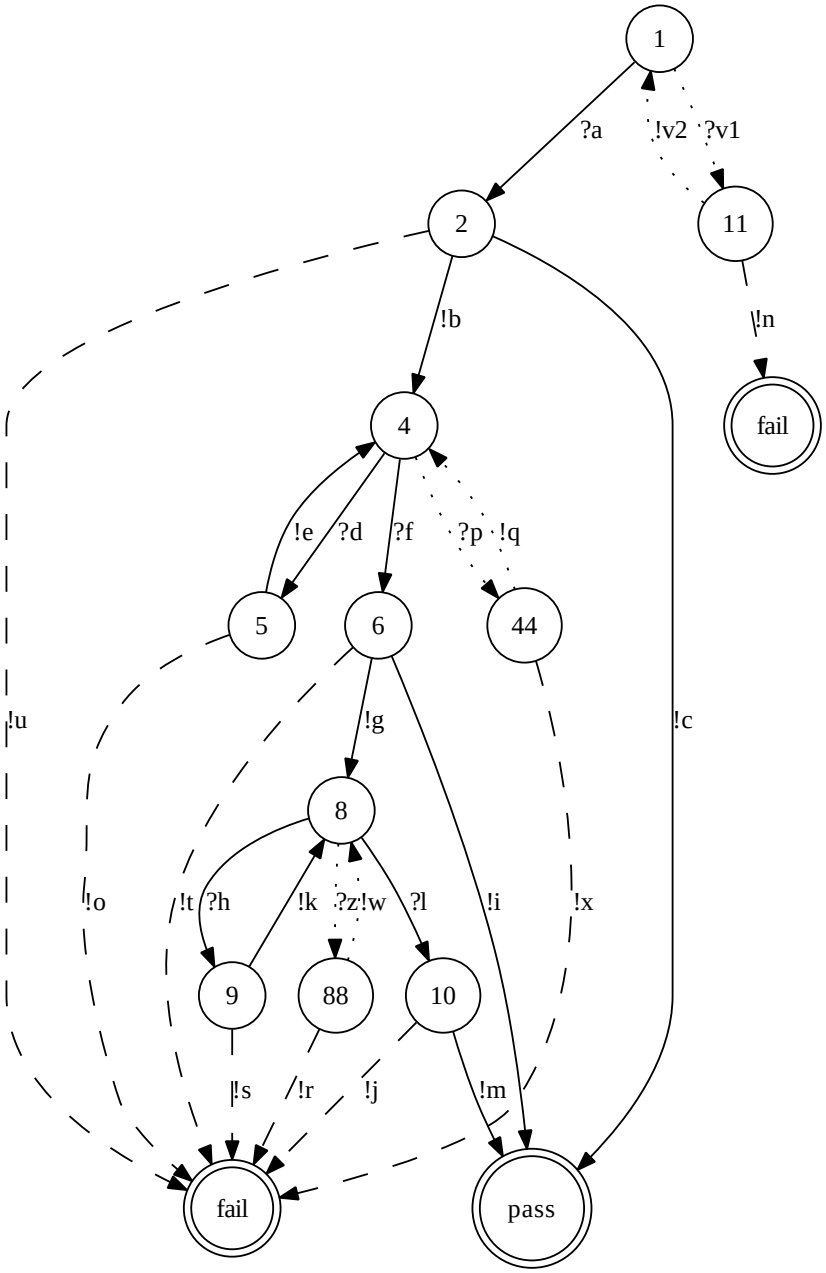

Figure 6: The Amazon AWSECommerceService specification

!c	ItemSearchResponse<String Errors,String IsValid> [IsValid==”false” && AWSAccessKeyID ≠ ”ID”]
!b	ItemSearchResponse<String Items,String IsValid> [IsValid==”true” && AWSAccessKeyID==”ID”]
!e	ItemLookUpResponse<String Items,String IsValid> [IsValid==”true” && AWSAccessKeyID==”ID”]
!i	CartCreateResponse<String Errors,String IsValid> [IsValid==”false” && (Quantity≥ 5 || ItemASIN≠”43451”)]
!g	CartCreateResponse<String CartId,String IsValid> [IsValid==”true” && Quantity¡5 && ItemASIN==”43451”]
!k	CartAddResponse<String IsValid> [IsValid==”true” && Quantity¡30 && ItemASIN==”66405”]
!m	PurchaseResponse<String IsValid> [IsValid==”true”]
?v1	ItemLookUp<String AWSAccessKeyID, String RequestID> |CartCreate<String AWSAccessKeyID,String ItemASIN, Integer Quantity> |CartAdd<String AWSAccessKeyID,String ItemASIN,Integer Quantity> |Purchase<String CartId,String CustomerInfos>
!v2	ItemLookUpResponse<R> |CartCreateResponse<R> |CartAddResponse<R> |PurchaseResponse<R> <R>=String| <R>=(c,soapFault) c=”RemoteException”
!n	δ|ItemLookUpResponse<R> |CartCreateResponse<R> |CartAddResponse<R> |PurchaseResponse<R> <R>=(c,soapFault) c≠”RemoteException”
!u	δ|ItemSearchResponse<R> [R≠String || (R≠”false” || AWSAccessKeyID==”ID”) && (IsValid ≠ ”true” || AWSAccessKeyID≠”ID”)]
?p	ItemSearch<String AWSAccessKeyID,String SearchIndex,String KeyWords> |CartAdd<String AWSAccessKeyID,String ItemASIN,Integer Quantity> |Purchase<String CartId,String CustomerInfos>
!q	ItemSearch<R> |CartAddResponse<R> |PurchaseResponse<Resp> R=String|R=(c,soapFault) c=”RemoteException”
!x	δ|ItemSearch<R> |CartAddResponse<R> |PurchaseResponse<R> R=(c,soapFault) c≠”RemoteException”
!o	δ|ItemLookUpResponse<R> [R≠ String || R≠”true” || AWSAccessKeyID≠”ID”)]
?z	ItemSearch<String AWSAccessKeyID,String SearchIndex,String KeyWords> |ItemLookUp<String AWSAccessKeyID,String RequestID> |CartCreate<String AWSAccessKeyID,String ItemASIN,Integer Quantity>
!w	ItemLookUpResponse<R> |ItemSearchResponse<R> |CartCreateResponse<R> R=String|*R*=(c,soapFault) c=”RemoteException”
!r	δ|ItemLookUpResponse<R> |ItemSearchResponse<R> |CartCreateResponse<R> R=(c,soapFault) c≠”RemoteException”
!t	δ|CartCreateResponse<R> [R≠String || (R≠”false” || (Quantity¡5 && ItemASIN == ”43451”)) && (R≠”true” || Quantity≥5) || ItemASIN≠”43451”)]
!s	δ|CartAddResponse< *R* > [R≠String || R≠”true” || Quantity≥30 || ItemASIN≠”66405”)]
!j	δ|PurchaseResponse<R>) [R≠String || R≠”true”]

Figure 7: Complete specification symbol table

complex (object, ...), we need to transform them into a set of simple types before using constraint solvers.

Figures 8 and 9 illustrate examples of test cases obtained from the complete specification of figure 6. In the first one, the operation "itemSearch" is called with the hazards "&", "168.150.13.1" and "$". If the response is equal to "false" (only possible response with the given hazards) then the verdict is "pass", otherwise we get the "fail" one. In the second test case, the operation "ItemLookUp" is called instead of "ItemSearch". The service is robust if it responds with an expected response type or with a SOAP fault composed of the cause "RemoteException" and if it is always possible to use the operation "ItemSearch" to reach a "pass" state (correct behavior). Otherwise, the verdict is "fail" and the web service is not robust.

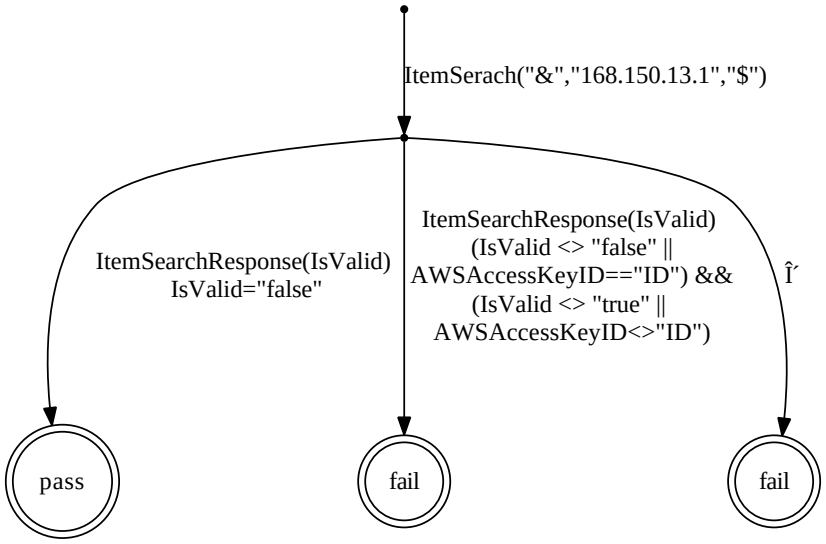

Figure 8: Test case 1

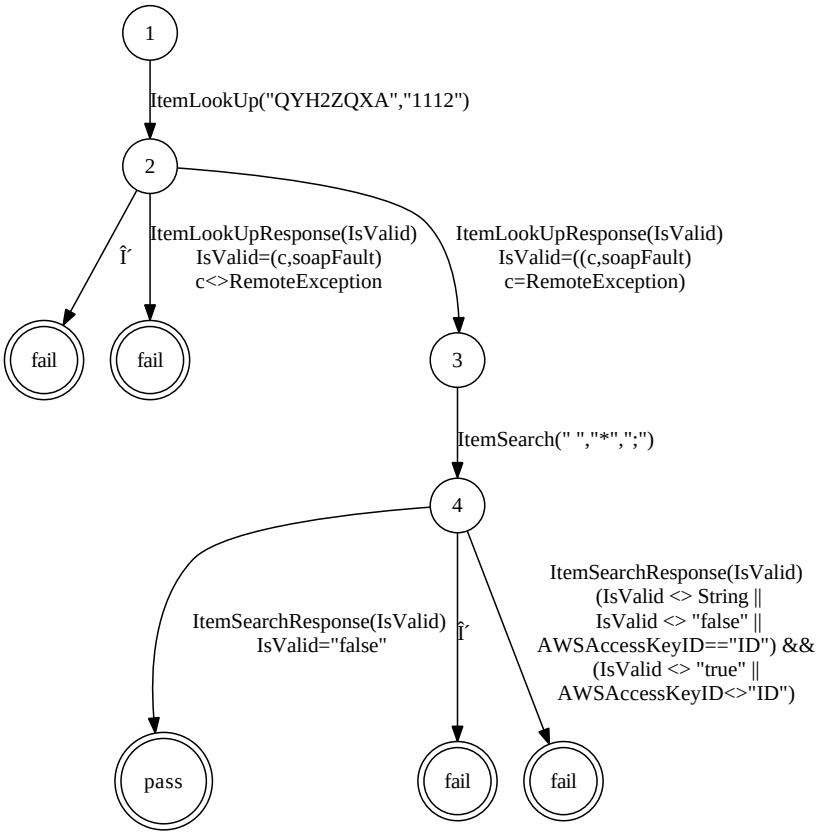

Figure 9: Test case 2

Algorithm: Test cases generation
Testcase(STS): TC
foreach *transition* $t = (l_k, l_{k+1}, ?e_k, \varphi_k, \varrho_k)$ *labeled by an input* $?e_k = op(param_1, ..., param_m)$ **do**
    path $p = DFS(l_0, l_k)$
    Solving(p)
    $Value(op) = \{(v_1, ..., v_m) \in V(param_1) \times ... \times V(param_m)\}$
    foreach $(v_1, ..., v_m) \in Value(op)$ **do**
        $TC = TC \cup tc$ with
        $tc = p; (l_k, l_{k+1}, op(v_1, ..., v_m), \varphi_k, \varrho_k); t$
        with $t = t'_j; postambule; verdict$ such as
        $\forall t' = (l_{k+1}, l_j, a_j, \varphi_j, \varrho_j) \in \rightarrow, (v_1, ..., v_n) \models \varphi_j$
        and $postambule = DFS(l_j, l_t)$ is a path between l_j and a final state $l_t \in L$
        and $verdict$ labeled in l_t
        Solving(postambule)
    end
end
Solving(path p):p
$p = (l_0, l_1, a_0, \varphi_0, \varrho_0)...(l_{k-1}, l_k, a_{k-1}, \varphi_{k-1}, \varrho_{k-1})$
foreach $(l_i, l_{i+1}, a_i, \varphi_i, \varrho_i)$ *with* $i > 0$ **do**
    $(x_1, ..., x_n) = solver(\varphi_i)$ //solving of the guard φ_i composed of the variables $(X_1, ..., X_n)$ such as $(x_1, ..., x_n) \models \varphi_i$
    $\varrho_{i-1} = \varrho_{i-1} \cup \{X_1 = x_1, ..., X_n = x_n\}$
end

Algorithm 1: Test case generation

4.3 Test case execution

Test cases are generated and then executed with the testing platform, illustrated in figure 10, which as been implemented in an academic tool. The tester corresponds to a web service which receives the URL of the web service to test and its specification. It constructs test cases as described in section 4.2, and then executes them successively. Once test cases are executed, it analyzes the obtained responses and finally gives a test verdict.

The tester executes each test case by traversing the test case tree: it successively calls an operation with parameters and waits for a response while following the corresponding branch. If a branch is fully executed, a local verdict "pass" or "fail" is obtained. We also set that quiescence (blocking

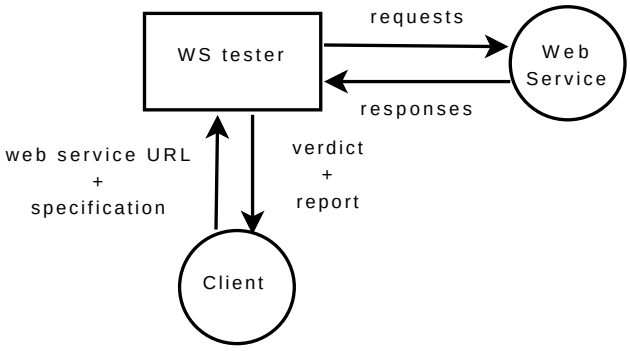

Figure 10: Test platform

state after a timeout) is observed after a timeout of 60s. For a test case t, we denote the local verdict $trace(t) \in \{\text{pass} , \text{fail}\}$. The final verdict is given by:

Definition 4.2 *Let WS be a web service and TC be a test case set. The verdict of the test over TC, denoted $Verdict(WS)/TC$ is*

- *"pass", if for all $t \in TC, trace(t) = pass.$,*
- *"fail", if it exists $t \in TC$ such as $trace(t) = fail.$*

We applied this method and executed test cases on two versions of the AWSECommerceService service (see figures 11, 12). Roughly 30 percent of the tests provide unexpected responses. With the hazard "Using unusual values", and despite that all the tests satisfy the WSDL description, we sometimes obtain SOAP faults composed the cause "Client", meaning that the request is incoherent. However, these results may be altered of account of security rules (firewalls,...). We have also received unspecified messages corresponding to errors composed of a wrong cause. For instance, we obtain the response "Your request should have at least 1 of the following parameters: AWSAccessKeyId, SubscriptionId." when we use "CartAdd" with a quantity equal to "-1", or when we search for a "Book" type instead of the "book" one, whereas these two parameters are correct. We have also observed the same situation with the hazard "Replacing /Adding operation names" only when we replace the operation "CreateCard" by another one. So, according to our robustness definition, this service seems to crash and to be not robust.

	09/03	09/10
Number of tests	100	100
Using unusual values	75	75
Replacing/Adding operation names	25	25
Fails	34	34
Unspecified response	28	28
Unspecified SOAP fault	6	6
SOAP fault with cause $\neq$ "RemoteException"	6	6

Figure 11: Test results on the Amazon AWSECommerceService services

	ItemSearch	ItemLookup	CartCreate	CartAdd
Number of tests	35	25	20	20
Using unusual values	29	19	14	13
Replacing/Adding operation names	6	6	6	7
Fails	29	1	2	2
Unspecified messages	28	0	0	0
Unspecified SOAP faults	1	1	2	2

Figure 12: Detailed Test results

5 Conclusion

The WS-I basic profile, which gathers the SOAP protocol and the WSDL language among others, reduces the web service observability. This lack of observability leads to new issues for robustness testing. We have shown, in this paper, that few hazards can be really used (only the hazards "Replacing name/Adding operation" and "Using unusual values") and that the web service behavior needs to be separated from the SOAP processor one to finally conclude on the web service robustness. We have proposed a robustness testing method which takes a symbolic specification and applies the previous hazards by completing the specification behavior and by injecting unusual values into the test cases.

Some perspectives can be considered, especially in relation to the set of unusual values V. This one can be manually modified but stays static during the test case generation. It could be more interesting to propose a dynamic analysis of the parameter types to build a list of the most adapted values for each web service. Furthermore, to avoid a test case explosion, the values of V are randomly chosen. A better solution would be to choose these parameters according to the operation description. It could also interesting to analyze the values raising the more errors while testing and to set a weighting at each

of them.

We have also assumed that the messages sent and received by web services are only SOAP messages. However, services can be connected to other servers, as databases. So, a web service could be considered as a gray box from which, any type of message could be observed.

Finally, this work can lead to the study of security testing which covers many aspects such as integrity, authentication or availability which depends on robustness. In the same way as this method, an observability analysis is required to determine the kind of security issues which can be detected and the responses from which a verdict can be established. A new specification completion should be proposed too.

References

[1] D. Tidwell, "Web services, the web's next revolution," in *IBM developerWorks*, Nov 2000.

[2] WS-I, "Ws-i basic profile." WS-I organization,, 2006, http://www.ws-i.org/docs/charters/WSBasic_Profile_Charter2-1.pdf.

[3] WSDL, "Web services description language (wsdl)." World Wide Web Consortium, 2001.

[4] SOAP, "Simple object access protocol v1.2 (soap)." World Wide Web Consortium, June 2003.

[5] O. U. Specification, "Universal description, discovery and integration," 2002, http://www.oasisopen.org/cover/uddi.html.

[6] 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

[7] "Magicdraw homepage," in *http://www.magicdraw.com*.

[8] E. Martin and T. Xie, "Automated test generation for access control policies," in *Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, Nov 2006.

[9] S. Hanna and M. Munro, "An approach for wsdl-based automated robustness testing of web services," in *Information Systems Development, Challenges in Practice, Theory, and Education*, vol. 2. Springer US, 2009, pp. 493–504.

[10] S. Salva and I. Rabhi, "Automatic web service robustness testing from wsdl descriptions," in *12th European Workshop on Dependable Computing*, June 2009.

[11] J. Offutt and W. Xu, "Generating test cases for web services using data perturbation," in *ACMSIGSOFT*, S. E. Notes, Ed., vol. 29(5), 2004, pp. 1–10.

[12] M. Vieira, N. Laranjeiro, and H. Madeira, "Assessing robustness of web-services infrastructures," in *In Proc. of the Int. Conf. On Dependable Systems and Networks (DSN'2007)*, 2007.

[13] "Ieee standard glossary of software engineering terminology," in *IEEE Standards Software Engineering 610.12-1990. Customer and terminology standards*, vol. 1. IEEE Press, 1999.

[14] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek, "Automated robustness testing of off-the-shelf software components," in *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*. Washington, DC, USA: IEEE Computer Society, 1998, p. 230.

[15] N. S. Niklas Een, "Minisat," 2003, http://minisat.se.

[16] 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.