A pragmatic approach for testing stateless and stateful Web Service Robustness

Sébastien Salva [*] — Antoine Rollet [**]

[*] *LIMOS - CNRS UMR 6158*
Université d'Auvergne, Campus des Cézeaux,
Aubière, France

sebastien.salva@u-clermont1.fr

[**] *LABRI CNRS UMR 5800*
University of Bordeaux
33405 Talence cedex, France

rollet@labri.fr

RÉSUMÉ. *L'intérêt dans les méthodes de test dédiées aux Services Web croît autant que l'utilisation massive de ces composants. Du fait que les Services Web sont hétérogènes par nature et prennent place dans l'élaboration de processus complexes de type Business, le test de robustesse, qui correspond au sujet principal de cet article, est une étape majeure pour développer des services fiables. Dans une première étape, nous nous focalisons sur l'environnement SOAP qui est utilisé pour appeler des opérations dans un format XML. Nous montrons que le protocole SOAP doit être pris en compte lors du test car il modifie de façon substantielle le comportement observable d'un Service Web mais aussi bloque certains tests. Ensuite, nous proposons deux approches: la première a pour objectif de test des Services Web* sans état *(stateless) formalisés par des modèles relationnels. La seconde est dédiée aux services* à états *(stateful) modélisés par des systèmes à transitions symboliques (STS). Pour ces deux méthodes, l'environnement SOAP est pris en compte en filtrant les réponses reçues ou en complétant la spécification. Ces méthodes ont été expérimentées au moyen d'un outil académique sur de nombreux Services Web déployés sur Internet. Cette expérimentation montre que plusieurs services comportent des failles de robustesse et que notre méthode est capable de les détecter.*

ABSTRACT. *The interest in testing methodologies dedicated to Web Services is soaring as much as the massive use of these components. Since Web Services are heterogeneous in nature and take part in complex Business processes, robustness testing which is the topic of this paper,* *is an important step to build them with confidence. Firstly, we focus on the SOAP environment which is used to call Web Service operations in an XML format. We show that SOAP must be taken into account in testing methods because it substantially modifies the Web Service observable behaviour and blocks many classical hazards used for testing. Then, we propose two approaches: the first one aims to test stateless Web Services, represented by relational models. The second approach is dedicated to stateful ones modelled with Symbolic Transition Systems. For both methods, the SOAP environment is taken into account by filtering the messages or by completing the specification. These methods have been experimented with an academic tool on many Web Services deployed on the Internet. This experimentation shows that several ones have robustness issues and that our methods are able to detect them.*

MOTS-CLÉS : *Test de robustesse; Services Web stateless et stateful; plateforme de test*

KEYWORDS: *Robustness testing; stateless, stateful Web Services; testing framework*

1. Introduction

The last three years mark a peak in Web Service development with almost all well-known companies are tending to model or to implement Web Service-based applications. These self-contained components offer many advantages such as, developing and deploying Business processes or externalizing functional code in a standardized way. Web Services can also be the foundation stones of large and complex Business applications such as BPEL processes [OC07].

Besides the Web Service tendency, there is currently a significant interest in the study of Web Service testing methodologies. Indeed, the standards proposed by the W3C and the WS-I consortiums, such as the WS-I basic profile [WI06], ensure interoperability but not reliability or robustness, which is the topic of this paper. A robust system is defined in [IEE90] as *able to function properly in the presence of faults or stressful environments*. Web Services are distributed in nature and heterogeneous. So, to trust them in an environment like the Internet, they need to behave correctly even they receive unspecified events. So, in other words, they need to be robust.

This paper tackles the robustness of stateless and stateful Web Services. Unlike stateless ones, stateful Web Services are persistent through a session and have an internal state which evolves over operation call sequences. For instance, all the Web Services using shopping cart software or beginning with a login step are stateful. Our approach is *black box*-based, so we have no knowledge of the Web Service internal structure. The benefit is to test deployed Web Services in the SOAP environment [Wc03], by calling operations and by observing the resulting reactions. We also show, in the paper, that taking into consideration the SOAP environment while testing is necessary, since SOAP modifies the messages by serializing them, and adds new messages which change the Web Service observable behaviour. For instance, when a Web Service is not available, the SOAP layer usually responds instead with a specific message. This one is received by testers but not generated by the Web service under test. In the same way, if a Web Service crashes, we may observe a SOAP response or not, according to the chosen SOAP framework.

Consequently, the Web Service behaviour is firstly analyzed in the presence of the SOAP environment. We filter the SOAP messages to recognize only those which are constructed by the Web Service under test. The other messages may falsify the test verdict and must be ignored. Then, we check if the classical *hazards*, used in software testing, can be still applied through the SOAP environment. In robustness testing, a hazard denotes an unusual and severe event e.g., an abnormal action or environmental influence [SKRC07]. Hazards are not faults but are actions used to stimulate the implementation under test. Hazards are not chosen while modeling but are usually combined with the specification to produce test cases which can be experimented on the implementation. These ones contain unexpected actions according to the specification. We study well-known hazards based on parameter and interface modifications. Network based hazards (random message modification etc.) are not considered in the paper. We show that only few hazards provide significant observable reactions while the testing process, because most of them are blocked by a component called *SOAP processor*.

This analysis leads to two testing methods : the first method tests stateless Web Services and gives a verdict by filtering the SOAP messages. The second one, dedicated to stateful Web Services, completes the specification according to the SOAP environment and the relevant hazard set before generating test cases. These methods have been implemented in an academic tool called WSAT (Web Service Automatic Testing). We experimented both methods, thanks to WSAT, on real implementations such as the Amazon E-commerce Web Service (AWSECommerceService) [Ama09]. Our results reveal that many of them have robustness issues.

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. Stateless and stateful Web Service modelling is described in Section 3. Section 4 analyzes the Web Service robustness over the SOAP layer. This leads to Section 5, which presents the testing methods and details the test case generation. We describe the testing framework and some experimentation results in Section 6. Finally, Section 7 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* [Tid00]. To ensure and to improve their interoperability, the WS-I organization has proposed profiles, and especially the WS-I basic profile [WI06], composed of four major axes :

– the *Web Service description* models how to invoke a Service set, called enpoints, and defines their interfaces and their parameter/response types. This description, called WSDL (Web Services Description Language) file [Wc07], also shows how messages must be structured by giving the type and message structures. WSDL is often used in combination with SOAP,

– the *definition and the construction of XML messages*, based on the Simple Object Access Protocol (SOAP) [Wc03]. SOAP is used to invoke Service operations (object methods) over a network by serializing/deserializing data (parameter operation and responses). SOAP takes place over different transport layers such as HTTP or SMTP,

– the *discovery of the Service* in UDDI registers. Web Service descriptions are gathered into UDDI (Universal Description, Discovery Integration [Oc02]) registers, which can be consulted manually or automatically by using dedicated APIs to find dynamically specific Web Services,

– the *security of the Service*, which is obtained by using the HTTPS protocol or XML encryption based protocols.

2.2. *Related work on Web Service testing*

There are many research papers concerning Web Service testing, and some of them are summarized below.

Some works consider compositions, where the components are Web Services. System specifications, often expressed by the UML or BPEL languages, describe the global system functioning by showing the possible interactions between the Services. In [GFTdlR06], the BPEL specification is translated into PROMELA in order to be used by the SPIN model checking tool. In [DYZ06], the authors use BPEL specifications which are translated into Petri nets. Then, standard Petri net tools are applied to study verification, testing coverage and test case generation. In [TFM05], the system is represented by a Task Precedence Graph and the behaviour of the composed components is represented by a Timed Labelled Transition System. Test cases are generated from these graphs and are executed by using a specific framework over SOAP. In [LZCH08], the BPEL specification is translated into an IF model, which enables modelling of timing constraints. Test case generation is based on simulation where the exploration is guided by test purposes.

Other works focus on Web Services seen as black boxes. In [FTdV06], the specification describes some successive calls of different operations which belong to the same Web Service. The specification is translated into the LTS model and test cases are generated according to the *ioco* implementation relation [Tre96]. In [BDTC05], Web Services are automatically tested by using only the WSDL description. Test cases are generated for two perspectives : test data generation (analysis of the message data types) and test operation generation (operation dependency analysis). In [BFPT06], the authors test the interoperability between Web Services. They propose to augment the WSDL description with a UML2.0 Protocol State Machine (PSM) diagram which models the possible interactions between the Service and a client. Test cases are then generated from the PSM. A framework, called the Audition framework, is proposed for executing these test cases in [BP05]. The tool WS-TAXI is proposed in [BBMP09] : it integrates the tool SOA-PIU which performs manual operation testing on Web Services, and integrates the tool TAXI which allows to define XML instances from an XSD file. The WS-TAXI tool generates test cases to automatically test all operations referred in the WSDL description.

There are few papers dealing with Web Service robustness testing. Most of them consider stateless ones. Robustness is then tested by handling WSDL descriptions [MX06, HM09, SR09] or by injecting hazards into SOAP messages [OX04, VLM07]. In [MX06], 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. If fact, this does not test directly Web Services but rather client methods which call them. Another method is proposed in [HM09], which is based on the WSDL analysis to identify which faults could affect the robustness attribute and then test cases were designed to detect those faults. In [OX04], the Web Service robustness is tested by applying some mutations on the request messages and by analyzing the obtained responses. In [VLM07], fault injection techniques are employed to create diverse fault-trigging test cases in order to display some possible robustness problems in the Web-Service code. We also proposed in [SR09] a Web Service robustness testing method, which automatically assesses the stateless Web Service robustness by using WSDL descriptions. The method performs random testing, improved by the use of the hazard *Using unusual values*. A tool has been developed and tested on real Web Services. By comparison to the previous works, we take into consideration the notion of SOAP processor. These components filter the requests and sometimes create responses instead of the service itself. As a consequence, our method detects more robustness issues.

We describe briefly this testing method in this paper and we go further by proposing a robustness testing method for stateful Web Services. To the best of our knowledge, testing the stateful Web Service robustness has not been studied before. On the one hand, some works have been proposed on control software system robustness [MS09], modelled with symbolic specifications. However, robustness is tested, in this work, with small input perturbations. Web Services are composed of operations, of variables with different type and depend of the SOAP environment. Testing their robustness raises new issues. On the other hand some works focus on WS-BPEL robustness [BCM09, ES06]. BPEL processes aim to orchestrate different Web Service partners. The partner internal states are not tested here. Moreover, BPEL robustness methods are currently passive i.e. faults are injected in SOAP messages passing through the network and results are monitored during a long period of time.

Unlike these works, we begin to evaluate which hazards are relevant for testing. We show that many are blocked by SOAP processors and are useless. We complete the specification to consider the observable actions spawned by SOAP processors and the SOAP protocol. Our method is also active and offers a better test coverage by testing all the operation call instances of the specification.

Furthermore, we use our own low level test framework to increase the message observability. Our platform directly calls Web Service operations and analyzes the SOAP responses, and especially the SOAP faults. So, it can detect if the exception management (error recovery) is not implemented by the Web Service under test but managed by the SOAP processor which responds with SOAP faults instead of the Web Service.

3. Web Service modelling

3.1. *Stateless Web Service*

We consider in this paper 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 Web service interface only i.e., the available operations, the parameter and response types. We also use the notion of SOAP fault. As defined in the SOAP v1.2 specification [Wc03], a SOAP fault is used to warn a client that an error has occurred. A SOAP fault is composed of : a fault code, a message, a cause, and 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. SOAP faults are not consistently described in WSDL files.

Definition 3.1 *Let X be a variable set and T be a set of types (Integer, String, Object, SOAPfault, etc.). A variable set is associated to a type set with the function $type\colon X^n \to T^n$.*

A Web Service WS is a component which can be called over an operation set $OP(WS) = \{op_1, ..., op_k\}$.

Each op_i is defined by the relation $op_i\colon\ from(op_i) \to to(op_i)$ with $from(op_i) = (x_1, ..., x_n) \in X^n$ and $to(op_i) = (x'_1, ..., x'_m) \in X^m$. The parameter types (the response types) are $type(from(op_i))$ ($type(to(op_i))$ resp.).

Let U be a value set. A valuation val is a relation val: $X^n \rightarrow U^n$ giving a value set from a variable set. A valuation of a variable x with $type(x) = SOAPfault$, is denoted $val(x) = (c, soapfault)$ with c the fault cause message. For simplicity, we also denote the type of a value set v, $type(v)$ with $v = val(x)$ and $type(v) = type(x)$.

We define the restriction $E_{|X^n} = \{val \in E \mid val\colon X^n \rightarrow E\}$. The latter leads to the operation invocation definition with the mapping inv_{op_i} such that inv_{op_i}: $U^n_{|from(op_i)} \mapsto U^m_{|to(op_i)} \cup \{(c, soapfault)\} \cup \{\epsilon\}$. ϵ models an empty response. To ease notation, inv_{op_i} is also denoted op_i.

The parameter types are either simple (integer, float, String, etc.) or complex (trees, tabular, objects composed of simple and complex types, etc.). Similarly, rhe response types are simple (integer, float, String, etc.), or complex or may be a SOAP fault. We denote a SOAP fault with (c,soapfault) to differentiate the SOAP cause in the testing methods. For instance (”Client”,soapfault) illustrates a SOAP fault composed of the ”Client” cause which means that the client request form is incorrect.

Some operations may be called without parameter and/or do not return any response. With or without parameter, an operation is always called with a SOAP message. However, if an operation does not produce any response, no SOAP message is sent to the client. We define that an operation which always returns a response is said observable :

Definition 3.2 *Let WS be a Web Service. An operation $op_i \in OP(WS)$ is observable if $\forall (p_1, ..., p_n) \in U^n_{|from(op_i)}, op_i(p_1, ..., p_n) \neq \epsilon$.*

In the remainder of the paper, we assume that Web Service operations are observable only. Without response that is without observable event, it becomes difficult to conclude whether an operation is robust or crashes and is faulty :

Web Service observable operation hypothesis : We suppose that each Web Service operation, given in WSDL descriptions, is observable.

Figure 1 illustrates, with UML sequence diagrams, a Web Service example which has two available operations : *getPerson* takes a String value and returns a Person object and *divide* returns the integer result of a division. The WSDL description of the *getPerson* operation is given in Figure 2. It provides the exchanged message form. For a request, the message is composed of two elements ”getPerson” and ”String”. The response message is composed of two elements ”getPersonResponse” and ”Person”.

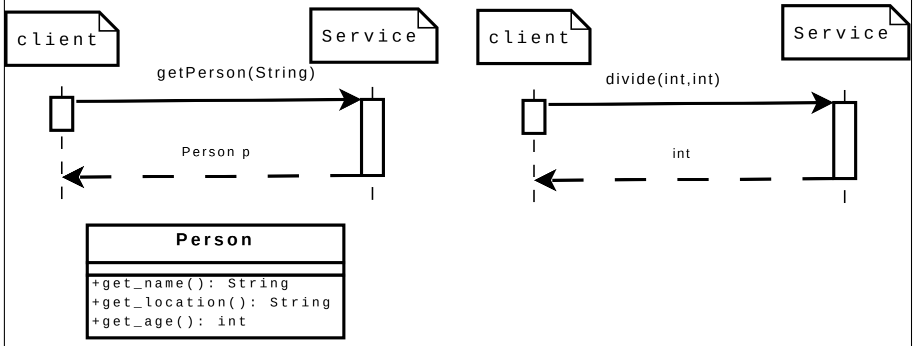

Figure 1 – Web Service UML specification

3.2. *Stateful Web Service modelling*

A stateful Web Service WS has an internal state which evolves over the operation call sequences. Its interface, which describes the available operations and parameters, is still formulated with Definition 3.1. We propose to also formalize its internal state with deterministic STS (Symbolic Transition Systems [FTW05]). This model corresponds to a kind of extended automaton model composed of transitions labelled by symbols combined with communication parameters, and of two sets of internal and external variables which may be used to evaluate guards and which may be updated. We briefly recall the STS definition below.

Definition 3.3 *A Symbolic Transition System STS is a tuple* $< L, l_0, Var, var_0, I, S, \rightarrow >$*, where :*

– L *is the finite set of locations, with* l_0 *the initial one,*

– Var *is the finite set of internal variables, while* I *is the finite set of external or interaction ones. We denote* D_v *the domain in which a*

```
<types> <schema>
    <element name="Person">
    ...
    </element>
    <element name="getPerson">
    <complexType>
    <sequence>
        <element name="x" type="xsd:string"/>
    </sequence>
    </complexType>
    </element>
    <element name="getPersonResponse">
    <complexType>
    <sequence>
        <element name="y" type="Person"/>
    </sequence>
    </complexType>
    </element>
    </element>
</schema> </types> <message name=
    "getPersonRequest">
    <part name="parameters" element=
    "getPerson"/>
</message> <message name=
    "getPersonResponse">
    <part name="parameters" element=
    "getPersonResponse"/>
</message>
```

Figure 2 – WSDL description of the *getPerson* operation

variable v takes values. The internal variables are initialized with the assignment var_0, which is assumed to take an unique value in D_{Var},

– S is the finite set of action labels, partitioned by $S = S_I \cup S_O$: inputs, beginning with ?, are provided to the system, while outputs (beginning with !) are observed from it,

– $\rightarrow$ 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 $a(p) \in S \times I^n_{n>0}$, $\varphi \subseteq D_{Var} \times D_p$ is a guard which restricts the firing of the transition. Internal variables are updated with the assignment $\varrho\colon D_{Var} \times D_p \rightarrow D_{Var}$ once the transition is fired.

The STS model is not specifically restricted to Web services. This is why we assume, in accordance with the Web service interface definition, that an action a represents either the invocation of an operation op which is denoted $?op$ or the return of an operation with $!op\_return$. An operation call is denoted by the transition $(l, l', ?op_i(from(op_i)), \varphi, \varrho)$, with ϱ the assignment of the parameter values to interval variables in Var. A response sending is modelled by a transition $(l, l', !op\_return(to(op_i)), \varphi, \varrho)$ composed of the guard φ describing e.g., a choice or a restriction depending on the parameter values used to call op. The observable operation assumption involves that each STS operation call (input) is followed by an operation response (output).

The STS example, given in Figure 3, describes a specification part of the Amazon Web Service devoted for E-commerce (AWSECommerceService). We begin to search for items (ItemSearch operation), to look for item details (ItemLookUp operation). Then we create a cart (CartCreate operation), which may be filled in with items (CartAdd operation) and purchased (Purchase operation). Note that we do not include all the parameters for readability reasons. A symbol table is given in Figure 4. Some parameters (BD.item.ASIN, BD.item.Quantity, BD.account.KeyID), given in the Amazon documentation, refer to Amazon database items. We only illustrate in Figure 5 the database part used in our example.

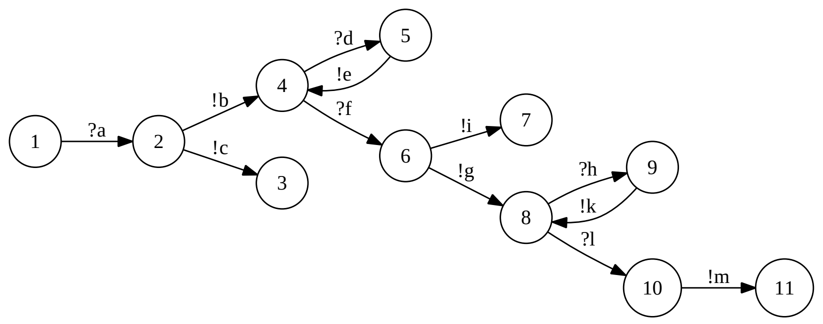

Figure 3 – The Amazon AWSECommerceService specification

Symbol	Message	Guard	Update
?a	ItemSearch<String AWSAccess-KeyID,String SearchIndex,String KeyWords>		id :=AWSAccessKeyID
?d	ItemLookUp<String AWSAccess-KeyID,String RequestID>		id :=AWSAccessKeyID
?f	CartCreate<String AWSAccess-KeyID,String ItemASIN,Integer Quantity>		id :=AWSAccessKeyID, q :=Quantity, item :=ItemA-SIN
?h	CartAdd<String AWSAccess-KeyID,String ItemASIN,Integer Quantity>		id :=AWSAccessKeyID, q :=Quantity, item :=ItemA-SIN
?l	Purchase<String CartId,String CustomerInfos>	[CartId==cart]	
!c	ItemSearchResponse< String Er-rors,String IsValid>	[IsValid=="false" $\wedge$ id $\neq$ BD.account.KeyID]	
!b	ItemSearchResponse< String Items,String IsValid>	[IsValid=="true" $\wedge$ id== BD.account.KeyID]	
!e	ItemLookUpResponse< String Items, String IsValid>	[IsValid=="true" $\wedge$ id== BD.account.KeyID]	
!i	CartCreateResponse< String Er-rors, String IsValid>	[IsValid=="false" $\wedge$ (q$\geq$ BD.Item.Quantity $\vee$item $\neq$BD.Item.ASIN)]	
!g	CartCreateResponse< String Car-tId, String IsValid>	[IsValid=="true" $\wedge$ q<BD.Item.Quantity $\wedge$ item== BD.Item.ASIN]	cart :=CartId
!k	CartAddResponse<String IsValid>	[IsValid=="true" $\wedge$ Quantity< BD.Item.Quantity $\wedge$ item== BD.Item.ASIN]	
!m	PurchaseResponse<String IsValid>	[IsValid=="true"]	

Figure 4 – Specification symbol table

account
KeyID
”ID”

ItemASIN	
ASIN	Quantity
”43451”	”5”
”66405”	”30”

Figure 5 – The Amazon AWSECommerceService Database

4. 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* [IEE90]. Although this sentence sounds usual, Web services are specific components which can be invoked through a SOAP environment only. The latter may modify messages and may also block some invocations. So, we analyze below the kind of SOAP messages which may be received and the hazards which can be used with Web service invocations to improve the robustness issue detection and to reduce the test cost by filtering out the useless hazards.

4.1. *Web Service observable event analysis and robustness*

We consider, in this paper, black box Web Services from which only SOAP requests and responses can be sent and received respectively. The SOAP layer, mainly used to serialize the invocations, substantially modifies the observed behaviour too. It is manifest that data, requests and responses are XML encoded, but the SOAP layer can also construct messages instead of Web Services. This analysis aims to describe these messages and to take them into account later in the testing steps.

As in the WS-I basic profile [WI06], we consider that a receiver, in a Web server, is a 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, which receives all the requests, instantiates the corresponding operations and returns a SOAP response to Client applications. This is specifically this component which modifies the observable event set. Below, we summarize the significant modification involved by SOAP processors that we have collected :

– **Calling an unavailable operation :** this action produces the receipt of a SOAP fault, constructed by the SOAP processor, composed of the cause ”the endpoint reference is not found”,

– **Calling an available operation with incorrect parameter types :** this action produces also the receipt of a SOAP fault, composed of the cause ”Client”. This latter means that the Client request does not match the Web Service WSDL description,

– **Exception management :** by referring to the WS-I basic profile, when an exception is triggered by an operation, it should to be translated into a SOAP fault and sent to the Client application. However, this feature needs to be implemented by hands in the operation code. So, when the exception management is implemented, the SOAP fault cause is usually equal to SoapFaultException (in Java or C# implementations). Otherwise, the operation crashes and the SOAP processor may construct itself a SOAP fault (or does nothing, depending on the chosen Web Service framework). In this case, the SOAP fault cause is different from SoapFaultException. Figures 6 and 7 depict two *divide* operations written in Java, which illustrate this exception management difference. When the first *divide* operation is called to divide an integer by 0, the operation crashes and some SOAP processors return a SOAP fault composed of the cause ”java.lang.ArithmeticException” since exceptions are not managed in the code. With other SOAP processors, no SOAP message is returned. For the second *divide* operation, the exception is spread until the Client application thanks to the piece of code ”throw new SoapFaultException(”error divide”+x+” by ”+y) which produces a SOAP fault, composed of the cause ”SoapFaultException”. In this case, the operation manages itself the exception.

```
Class Service { public int divide(int x, int y) {
 return (x/y); }
            }
```

Figure 6 – Example I

```
Class Service { public int divide (int x, int y)
   throws SOAPFaultException {
 try{
    int result=x/y; return result;}
 catch (Exception e) {
    throw new SOAPFaultException(
    "error divide"+x+" by "+y); }
}
```

Figure 7 – Example II

This analysis helps to propose a Web Service operation robustness definition which removes the events constructed by SOAP processors. This definition implies that a robust operation can catch any exception and constructs itself SOAP faults composed of the SOAPFaultException cause only.

Definition 4.1 *Let WS be a Web Service. An operation op_i: $from(op_i) \to to(op_i) \in OP(WS)$ is robust, iff* $\forall v \in U^n_{|from(op_i)}$, $r = op_i(v) \in U^m_{|to(op_i)} \cup \{(SOAPFaultException, soapfault)\}$.

4.2. *Hazard acceptance assessment for stateless Web Services*

We analyzed in [SR09] the Web Service operation behaviour in the presence of hazards. The obtained results are summarized below. This analysis helps to separate the hazards which can test operations, to those which are blocked by SOAP processors. For an operation op: $from(op) \to to(op)$, we focused on the following hazards based on the parameter modification [KKS98, CS04]. Note that the WS-I basic profile does not permit operation overloading. So, overloading is not considered here.

– **Replacing parameter types :** one or more variable types in $from(op)$ are replaced by other types. With this hazard, we always obtain a SOAP fault composed of the cause Client. This means that the given parameter values are incorrect and that the invocation is blocked by the SOAP processor. So, this hazard is not relevant for testing the Web Service robustness,

– **Adding/injecting parameters :** adding parameters in the beginning of the request or between existing parameters is equivalent to *replacing parameter types* and we have seen previously that this hazard is useless. When, we call an operation by adding parameters at the end of the existing ones with $op(p_1, ..., p_n, p_{n+1}, ..., p_{n+k})$, the values $p_{n+1}, ..., p_{n+k}$ are not read by the SOAP processor and not given to the Web Service. Therefore, this hazard is not relevant,

– **Deleting parameters :** as previously, deleting parameters in the beginning or between existing parameters is comparable to the hazard *replacing parameter types* and is not relevant. When, we call an operation with less parameter values than required with $op(p_1, ..., p_k), (k < n)$, we obtain two kind of responses according to the WSDL description. If the option ”nillable=true” is used in the WSDL file, this is equivalent to calling *op* with null values $(p_1, ..., p_k, null, ..., null)$. We consider that ”null” is an unusual value (see below). Otherwise, the SOAP processor returns a SOAP fault composed of the cause Client which means that the invocation is blocked. Thereby, this hazard is either comparable to the hazard *Using unusual values* or is unnecessary,

– **Inverting parameters :** this hazard is comparable to *Replacing parameter types*,

– **Using unusual values :** this hazard, well-known in software testing [KKS98], aims to call *op* with values $(p_1, ..., p_n) \in U^n_{|from(op_i)}$, satisfying its WSDL description (so, $p_i = val(x_i)$, and $type(x_i)$ equals to the type given in the WSDL description). But these predefined values are assumed to have a high bug-revealing rate when used as inputs. For instance, null ; ”” ; ”$” ; ”*” are some unusual String values. The requests composed of such values are not rejected by SOAP processors since the WSDL description is respected by these. Therefore, this hazard can be used for Web Service robustness testing.

Consequently, only the last hazard *Using unusual values* is not blocked in the SOAP environment and can be used for testing.

4.3. *Hazard acceptance assessment for stateful Web Services*

We extend here the previous study on stateful Web Services. We focus on the following hazards modifying the Web Service interface

[KKS98, CS04] : **Changing operation name, Replacing /Adding operation name** in addition to the previous ones. Let WS be a Web Service and STS be its specification :

– **Changing the operation name** : this hazard aims to randomly modify an operation name $op \in OP(WS)$ to op_modif such that 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 the WSDL description is not satisfied by these. So, because the test cannot be performed, we consider that this hazard is useless,

– **Replacing /Adding operation calls** : Let l be a location with the outgoing transitions $(l, l_1, ?op_1(p_{11}, ..., p_{1n}), \varphi_1, \varrho_1)$, ..., $(l, l_k, ?op_k(p_{k1}, ..., p_{kn}), \varphi_k, \varrho_k)$ modelling operations calls. If it exists an operation $op\colon from(op) \rightarrow to(op)$ such that $op \notin \{op_1, ..., op_k\}$, this hazard aims to replace/add the call of an operation $op_i \in \{op_1, ..., op_k\}$ by op. Since this hazard satisfies the Web Service WSDL description, it is not blocked by SOAP processors. However, when op is invoked, we can only receive a response from op. We cannot just replace/add a name in the specification. For instance, if we replace the operation ItemSearch by AddCard, in our example of Figure 15, we do not receive a response from ItemSearch but from AddCard. So, If the operation op is robust, according to Definition 4.1, the expected response from any invocation $op(p_1, ..., p_n)$ is either a response $(r_1, ..., r_n)$ with $type((r_1, ..., r_n)) = type(to(op))$, or a SOAP fault (SOAPFaultException,soapfault).

This hazard involves to complete the specification on the operation calls for each location. This modification is detailed in Section 5.2.

There exist of course other hazards based on the SOAP message modification, such as replacing the port name or modifying SOAP messages randomly. These hazards are usually used for testing Web Service compositions in order to observe partner behaviours. Concerning stateful Web Services, either the message random modification is equivalent to a previous hazard (parameter, operation modification) or gives an inconsistent SOAP message which is always blocked by SOAP processors since it does not satisfy the Web Service WSDL description.

Consequently, the two hazards *using unusual values* and *replacing /adding operation names* can be used for testing stateful Web Services since these are not blocked by SOAP processors.

5. Web Service robustness testing methods

Prior to describing the two robustness testing methods, we define the test case modelling, with STS :

Definition 5.1 *A test case* $T = < L, l_0, Var, var_0, I, S, \rightarrow >$ *is an STS tree where each final location is labelled by a verdict in* $\{pass, fail/available, fail\}$.

For a Web Service WS, branches are labelled either by $(?op_i(from(op_i)), \varphi, \varrho, \lambda)$ or by $(!op_i\_return((to(op_i)), \varphi, \varrho, \emptyset)$ or by δ where op_i is an operation in $OP(WS)$ and δ represents quiescence. Intuitively, the *pass* verdict means that the test is successful, *fail/available* means that the WS operations can be called as it is described in the WSDL file (the operations exist and take the correct parameter types) and that WS is not robust. With *fail*, at least one operation is unavailable and WS is not robust. The two last verdicts refine the classic fail case by distinguishing the case where the implemented operation do not match the WSDL description.

For example, $l_0 \xrightarrow[s:=\text{"12345"}]{?getPerson(Strings)} l_1 \xrightarrow{!getperson\_return(Strings2)} pass$ is a test case which invokes the $getperson$ operation with the parameter value ”12345”. The response must be a String value.

5.1. *Stateless Web Service automatic robustness testing*

For a stateless Web Service WS, our method aims at testing these two features :

– *Existence of all Service operations* : for each operation op_i: $from(op_i) \rightarrow to(op_i) \in OP(WS)$, we construct test cases to check whether the implemented operation corresponds to its description in the WSDL file. So, test cases call the operation op_i with several values $(p_1, ..., p_n) \in U^n_{|from(op_i)}$. op_i exists if $op_i(p_1, ..., p_n)$ returns a response r such that r is either a specified response ($r \in U^m_{|to(op_i)}$), or $r = (c, soapfault)$ is a SOAP fault where the cause c is different from Client and the endpoint reference not found. The first cause means the operation is called with bad parameter types. The second cause means that the operation name does not exist (see Section 4.1). Otherwise, op_i does not exist as described in the WSDL file,

– *Web Service operation robustness* : for each operation op_i: $from(op_i) \rightarrow to(op_i) \in OP(WS)$, we construct test cases to check if op_i does not crash or hang by calling it with hazards. As stated in Section 4.2, we use the hazard *Using unusual values* which means that op_i is called with unusual parameter values $(p_1, ..., p_n) \in U^n_{|from(op_i)}$. According to the operation robustness definition (Definition 4.1), either $op_i(p_1, ..., p_n)$ should return a specified response $r \in U^m_{|to(op_i)}$ or a SOAP fault $(SOAPFaultException, soapfault)$. We consider that a Web Service is not robust if quiescence (no response received after a timeout) is observed or if any other response is received.

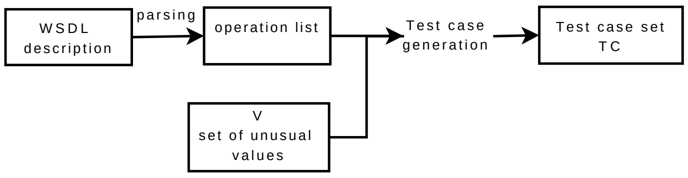

Figure 8 – Stateless Web Service test case generation

The test case generation is illustrated in Figure 8. We parse the Web Service WSDL file to list the operation set. Then, we use a predefined set of values V to generate test cases. This set contains for each type, an XML value list used for calling each operation. Theses values have been chosen after the Web Service response analysis of Section 4.2 in order to check that operations exist, and to construct requests with unusual values.

We denote $V(t)$ the set of specific values for the type t which can be a simple type or a complex one. Figures 9, 10 and 11 show some values used for the type Int, 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 $V(String)$.

```
<type id="Int">
        <val value=null />
        <val value="0" />
        <val value="-1" />
        <val value="1" />
        <val value="MIN" />
        <val value="MAX" />
        <val value=RANDOM" /> <!-- a random Int-->
</type>
```

Figure 9 – V(Int)

```
<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>
```

Figure 10 – V(String)

For a Web Service WS, the test case generation is composed of the following steps :

1) The WSDL description is parsed to produce the operation list $L = \{op_1, ..., op_l\}$,

2) for each operation $op_i \in L$: $from(op_i) = (p_1, ..., p_n) \rightarrow to(op_i) = (r_1, ..., r_m)$, we construct the tuple set $Value(op_i) = \{(v_1, ..., v_n) \in V(p_1) \times ... \times V(p_n)\}$. If the parameter types are complex (tabular, object, etc.), we compose them with other ones to obtain the final values. Since $Value(op_i)$ is constructed by means of a cartesian product, the number of values used for testing may manifestly explode. So,

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

Figure 11 – V(tabular)

we use a heuristic to estimate and eventually to reduce the $Value(op_i)$ cardinality : if $card(Value(op_i)) > Max$, $Value(op_i))$ is reduced by removing successively one value of $V(p_1)$, then one of value of $V(p_2)$, and so on up to $card(Value(op_i)) <= Max$,

3) for each operation $op_i \in L$, we construct the test case set $TC(op_i)$ with :

$TC(op_i) = \bigcup_{v \in Value(op_i)} \{(l_0, l_1, ?op_i(X), [X = v], \emptyset),$
$(l_1, pass, !op_i\_return(r), \varphi_1, \emptyset), (l_1, pass, !op_i\_return(r), \varphi_2, \emptyset),$
$(l_1, fail/available, !op_i\_return(r), \varphi_3, \emptyset), (l_1, fail, \delta, \emptyset, \emptyset),$
$(l_1, fail, !op_i\_return(r), \neg(\varphi_1 \vee \varphi_2 \vee \varphi_3), \emptyset)\}$
where $\varphi_1 = [type(r) = type(to(op_i))]$, $\varphi_2 = [val(r) = (SOAPFaultException, soapfault)]$, $\varphi_3 = [val(r) = (c, soapfault), c \notin$ {Client, SOAPFaultException, the endpoint reference not found],

4) and finally, the test case set $TC = \bigcup_{op_i \in L} \{TC(op_i)\}$.

For more readability, we illustrate two test case schema : one for the operation existence testing (Figure 12) and one for the robustness testing (Figure 13). In TC, each test case calls one operation. If the response is not a SOAP fault and if the response type matches the one described in the WSDL file, the local verdict is pass. If the response is a SOAP fault with the cause SOAPFaultException, then the operation manages itself exceptions and the local verdict is pass. If the response is a SOAP fault whose cause is not in {SOAPFaultException,client, the endpoint reference not found} then the operation exists but does not manage exceptions. The operation crashed and the SOAP fault is returned by the SOAP processor. Thus, this operation is not robust and the local verdict is fail/available. Otherwise, the local verdict is fail.

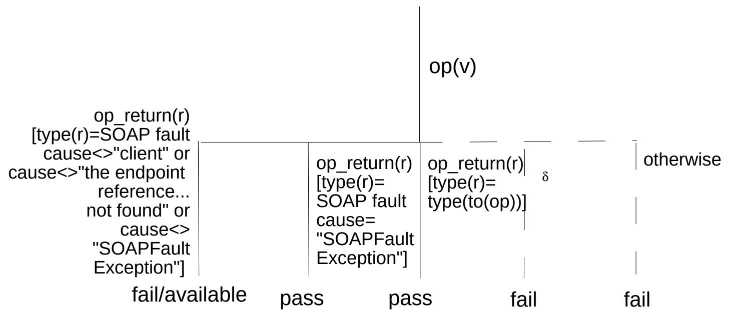
op(v)
op_return(r) [type(r)=SOAP fault cause<>"client" or cause<>"the endpoint reference... not found" or cause<> "SOAPFault Exception"]
op_return(r) [type(r)= SOAP fault cause= "SOAPFault Exception"]
op_return(r) [type(r)= type(to(op))]
δ
otherwise
fail/available
pass
pass
fail
fail

Figure 12 – Test case schema for testing the operation existence

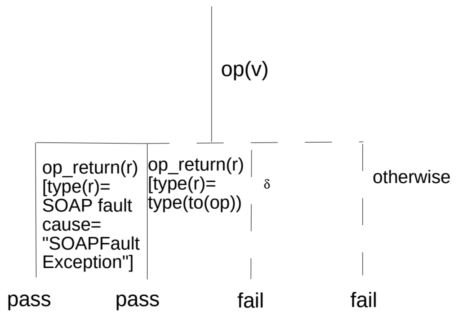
op(v)
op_return(r) [type(r)= SOAP fault cause= "SOAPFault Exception"]
op_return(r) [type(r)= type(to(op))
δ
otherwise
pass
pass
fail
fail

Figure 13 – Test case schema for testing robustness

5.2. *Stateful Web Service robustness testing method*

This method checks whether a stateful Web Service behaves correctly despite the use of the hazards *Using unusual values* and *Replacing /Adding operation names*, previously described in Section 4.

The test case generation method is illustrated in Figure 14. First, the specification locations are completed on the input set to apply the hazard *Replacing /Adding operation names*. The specification is also completed to model the incorrect behaviour (incorrect responses) and

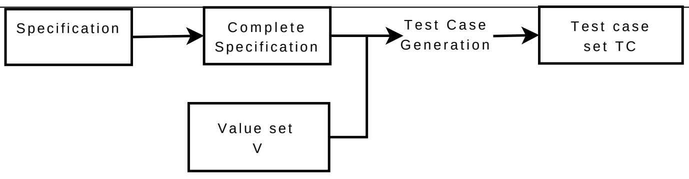

Figure 14 – Stateful Web Service test case generation

quiescence. So, the test cases generated from the complete specification, will describe both the correct behaviour and the incorrect behaviour of a Web Service.

These steps are detailed below.

5.2.1. *Specification completeness*

As stated previously, we complete the specification to apply the *Replacing /Adding operation name* hazard and to append the incorrect Web Service behaviour.

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 with the following steps :

– **Replacing the terms referring to database items** with database values,

– **Operation call completion** : locations preceding an operation request and end locations are completed to be input enabled and to take into account the hazard *Replacing /Adding operation names*.

So, $\forall l \in L$ such that l is a end location or has the outgoing transitions $(l, l_1, ?op_1(p_{11}, ..., p_{1n}), \varphi_1, \varrho_1)$, $..., (l, l_k, ?op_k(p_{k1}, ..., p_{kn}, \varphi_k, \varrho_k)$, we add :
$\forall op_i : from(op_i) \rightarrow to(op_i) \in OP(WS)/\{op_1, ..., op_k\}$, $(l, l_i, ?op_i(p_{i1}, ..., p_{in}), \emptyset, \emptyset)$, $(l_i, l, !op_i\_return(r_1), \varphi_1, \emptyset)$, $(l_i, l, !op_i\_return(r_2), \varphi2, \emptyset)$, with
$\varphi_1 = [type(r_1) = type(to(op_i))]$, $\varphi_2 = [val(r_2) = (SOAPFault Exception, soapfault)]$. From l, any operation op_i can be invoked. We also add the possible responses which can be received if op_i is robust as described in Definition 4.1,

– **pass verdict addition**, each specification location is labelled by *pass*. These labels will be transferred into the test cases and mean that to reach a pass location while testing, a specification behaviour has been executed.

– **Incorrect behaviour completion** : the specification is completed on the incorrect response set with the following transitions. As previously, the states labelled by *fail*, will be transferred into the final test cases. $\forall l \in L$ such that l has the outgoing transitions $(l, l_1, !op_i\_return(r_1), \varphi_1, \varrho_1)$, ..., $(l, l_k, !op_i\_return(r_k)\ , \varphi_k, \varrho_k)$, we add :

- $(l, fail, \delta, \emptyset, \emptyset)$, with δ expressing quiescence,

- $(l, fail, !op_i\_return(r), \varphi_1, \emptyset)$ with $\varphi_1 = [r = (Client, soapfault) \vee r = (the\ endpoint...\ found, soapfault)]$. The called operation is not robust and not available if it returns a SOAP fault composed of the cause Client or the endpoint reference is not found, (see Section 4.1),

- $(l, fail/available, !op_i\_return(r), \varphi', \emptyset)$, $\varphi' = [\neg(\varphi_1 \vee ... \vee \varphi_k \vee \varphi)]$. If the operation returns any other response, then the Web Service is not robust in the presence of hazards but the operation is available. Thus a fail/available verdict is reached.

The STS of Figure 15 illustrates a completed specification, obtained from the STS of Figure 3. The dotted transitions represent the operation call completion while those illustrated with dashed lines model the incorrect behaviour addition. The new symbol table is given in Figure 16. For instance, from location 1 any operation can now be invoked. When an operation different from ItemSearch is called with ?v1, the service is robust if a specified response or a SOAP fault with the cause SOAPFaultException is received.

5.2.2. *Test case generation*

Test cases are constructed with Algorithm 1. Firstly, Algorithm 1 refers to the $PathFinding$ procedure given in Algorithm 2. This latter is classically based on the transition exploration with backtracking. But it also solves the constraints of the current path p on the fly to ensure that p can be completely executed (line 4). It calls the *Solving* proce-

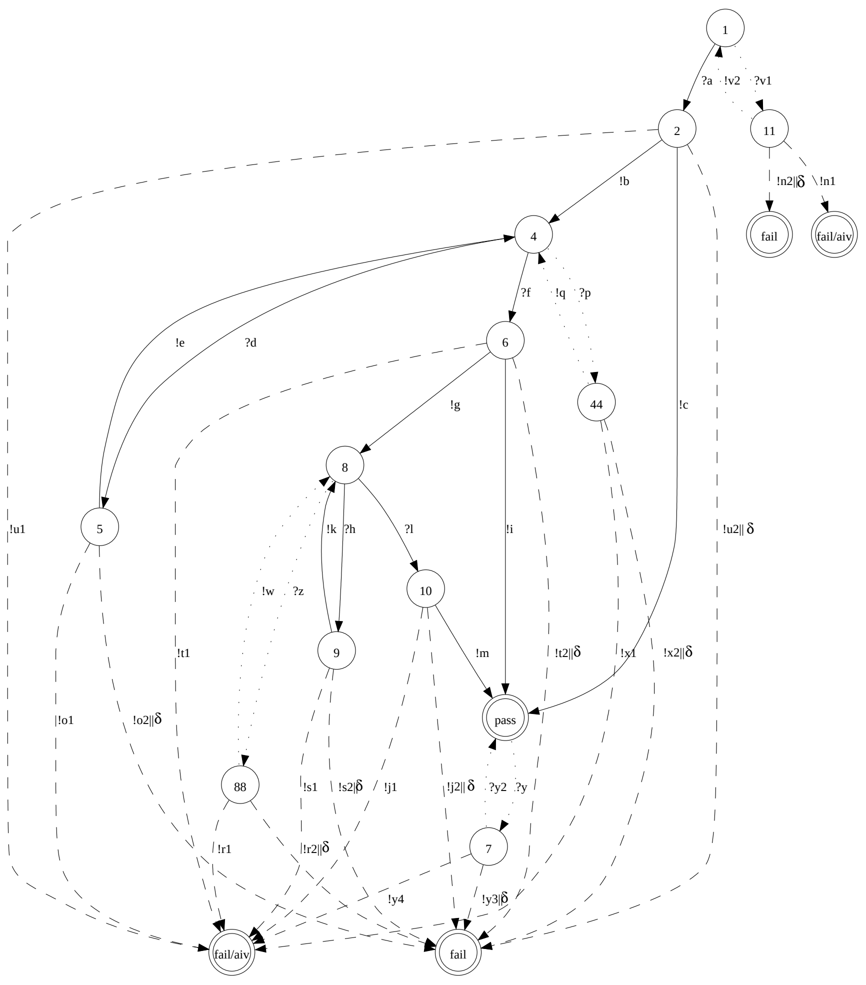

Figure 15 – The Amazon AWSECommerceService specification

Symbol	Message	Guard/Update
?v1	ItemLookUp||CartCreate|| CartAdd||Purchase	
!v2	ItemLookUpResponse<R> ||CartCreateResponse<R> ||CartAddResponse<R> ||PurchaseResponse<R>	[R=String∨R=(SOAPFaultException, soapFault)]
!n1		[$\neg(\varphi(!v2) \vee \varphi(!n2)$]
!n2		[R== (Client,soapFault)∨ R==(The endpoint...,soapFault)]
!u1	ItemSearchResponse<R>	[$\neg(\varphi(!b) \vee \varphi(!c) \vee \varphi(!u2)$]
!u2		[R== (Client,soapFault)∨ R==(The endpoint...,soapFault)]
?p	ItemSearch||CartAdd|| Purchase	
!q	ItemSearch<R> ||CartAddResponse<R> ||PurchaseResponse<R>	[R=String∨(R=(SOAPFaultException,soapFault)]
!x1		[$\neg(\varphi(!q) \vee \varphi(!x2)$]
!x2		[R== (Client,soapFault)∨ R==(The endpoint...,soapFault)]
!o1	ItemLookUpResponse<R>	[$\neg(\varphi(!e) \vee \varphi(!o2)$]
!o2		[R== (Client,soapFault)∨ R==(The endpoint...,soapFault)]
?z	ItemSearch||ItemLookUp|| CartCreate	
!w	ItemLookUpResponse<R> ||ItemSearchResponse<R> ||CartCreateResponse<R>	[R=String∨(R=(SOAPFaultException, soapFault)]
!r1		[$\neg(\varphi(!w) \vee \varphi(!r2)$]
!r2		[R== (Client,soapFault)∨ R==(The endpoint...,soapFault)]
!t1	CartCreateResponse<R>	[$\neg(\varphi(!g) \vee \varphi(!i) \vee \varphi(!t2)$]
!t2		[R== (Client,soapFault)∨ R==(The endpoint...,soapFault)]
!s1	CartAddResponse$< R >$	[$\neg(\varphi(!k) \vee \varphi(!s2)$]
!s2		[R==(Client,soapFault)∨ R==(The endpoint...,soapFault)]
!j1	PurchaseResponse<R>	[$\neg(\varphi(!m) \vee \varphi(!j2)$]
!j2		[R== (Client,soapFault)∨ R==(The endpoint...,soapFault)]
?y	ItemSearch||ItemLookUp|| CartCreate||CartAdd|| Purchase	
!y2	ItemLookUpResponse<R> ||ItemSearchResponse<R> ||CartCreateResponse<R> ||CartAdd<R> ||Purchase<R>	[R=String∨(R=(SOAPFaultException, soapFault)]
!y3		[R== (Client,soapFault)|| R==(The endpoint...,soapFault)]
!y4		[$\neg(\varphi(!y3) \vee \varphi(!y2)$]

Figure 16 – Complete specification symbol table

dure, which takes a test case path p and returns an external variable assignment λ which satisfies the firing of the transitions of p. If the constraint solvers [ES04, KGG^{+}09] cannot compute a value set allowing to execute p, then *Solving* returns an empty set (lines 18-19). The constraint solvers construct values satisfying the guards of a specification path and hence satisfying its execution. We use the solvers [ES04] and [KGG^{+}09] which work as external servers that can be called by the test case generation algorithm. The solver [KGG^{+}09] manages String types, and the solver [ES04] manages most of the other simple types. The time complexity of these solvers is polynomial.

Algorithm 1 constructs test cases with the following steps. For a transition t modelling the call of the operation *op*, a preamble p is constructed with the $PathFinding$ procedure (lines 3-5). Then, this preamble is completed with its incorrect behaviour (lines 6-8). A value set over V is constructed according to the *op* parameter types (line 9). As in Section 5.1, an heuristic is used to estimate and eventually to reduce the number of tests according to the cardinality of $Value(op)$. The test case tc' is reset with tc, its variables are initialized with ϱ_0, and the transition t id added to the transition set (lines 11-13). The external variable assignment $\lambda_{tc'}$ is used to test *op* with unusual values. Then, each next transition t_f from the location l_{k+1}, labelled by an output, is added to the test case (lines 14-15). tc' is finally added to the test case set (line 16).

The time complexity of Algorithm 1 is $O(n(n + m + card(Value(op))n))$. This time complexity is obviously not polynomial when considering $card(Value(op))$ which represents the tuple of values used for testing. If we suppose that any operation has at most p parameters and that we have k values for each variable type, the number of tuples becomes k^p. To solve this issue, we currently bound $card(Value(op))$ in our algorithm implementation. Nevertheless, there are more elegant solutions which help to reduce this number. For instance, *pairwize* testing is a testing criterion which requires that for each pair of input parameters every combination of values of these two parameters are covered by a test case. It has been show in [CSD^{+}97] that the number of tuples grows at most logarithmically in p and quadratically in k. But, such a method is still in debate because it supposes that bugs

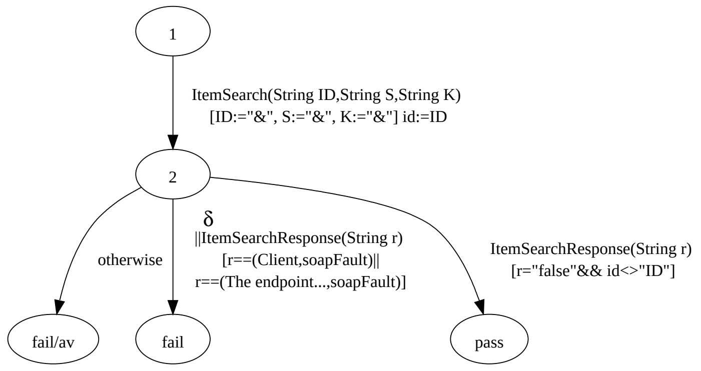
1
ItemSearch(String ID,String S,String K)
[ID:="&", S:="&", K:="&"] id:=ID
2
otherwise
δ
||ItemSearchResponse(String r)
[r==(Client,soapFault)||
r==(The endpoint...,soapFault)]
ItemSearchResponse(String r)
[r="false"&& id<>"ID"]
fail/av
fail
pass

Figure 17 – Test case 1

can be triggered by either one parameter or by the interactions between pairs of parameters. Moreover, the value tuple set requires a significant time to construct.

Figures 17 and 18 illustrate two test case obtained from the complete specification of Figure 15. In the first example, the operation ItemSearch is called with the values "&" ;"&" ;"&". If the response is equal to "false" (only possible response with the given values) then the verdict is pass. If either a SOAP fault composed of the cause Client or the endpoint... is received or if quiescence is observed, the operation is not available (verdict fail). If another response is produced, the operation is available but not robust (verdict fail/available). 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 SOAPFaultException and if it is always possible to call the operation ItemSearch to reach a pass location (correct behaviour). Otherwise, the verdict is fail or fail/available and the Web Service is not robust.

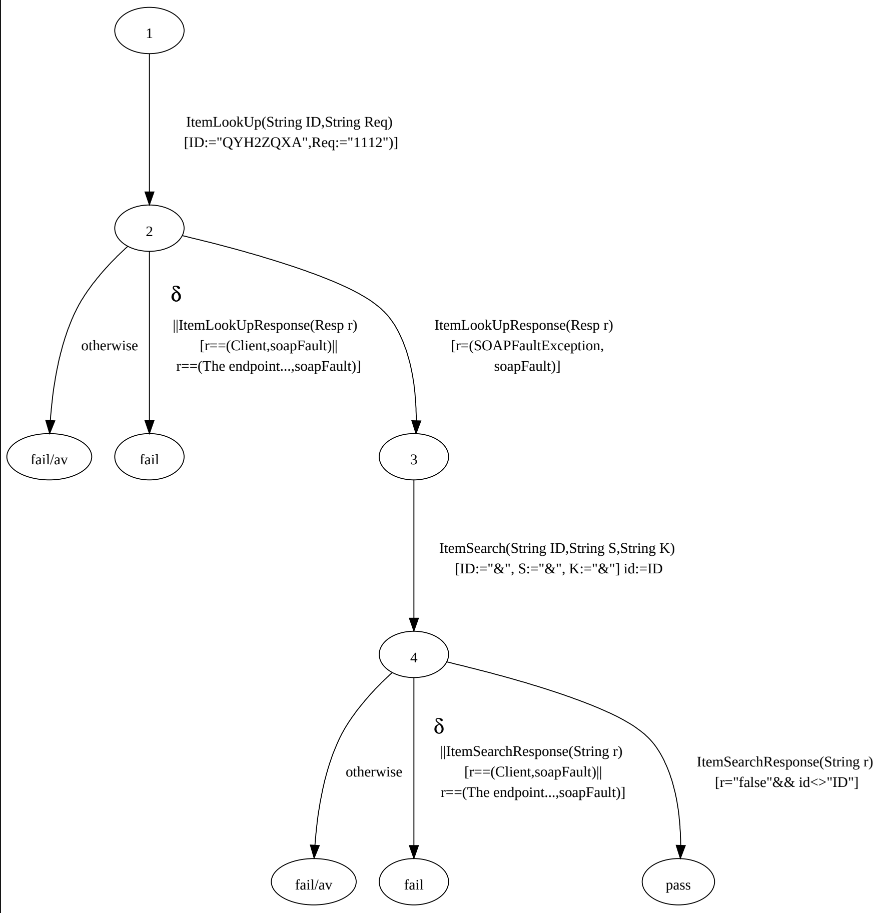
1
ItemLookUp(String ID,String Req)
[ID:="QYH2ZQXA",Req:="1112")]
2
otherwise
δ
||ItemLookUpResponse(Resp r)
[r==(Client,soapFault)||
r==(The endpoint...,soapFault)]
ItemLookUpResponse(Resp r)
[r=(SOAPFaultException,
soapFault)]
fail/av
fail
3
ItemSearch(String ID,String S,String K)
[ID:="&", S:="&", K:="&"] id:=ID
4
otherwise
δ
||ItemSearchResponse(String r)
[r==(Client,soapFault)||
r==(The endpoint...,soapFault)]
ItemSearchResponse(String r)
[r="false"&& id<>"ID"]
fail/av
fail
pass

Figure 18 – Test case 2

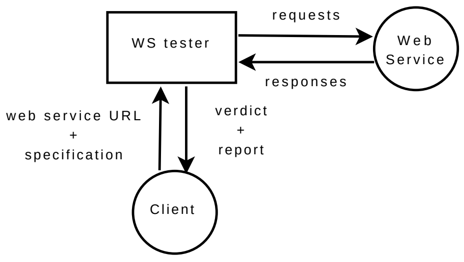

Figure 19 – Test architecture

Algorithm 1: Test case generation

```
1  Testcase(STS) : TC;
   input  : An STS $s =< L^s, l_0^s, Var^s, var_0^s, I^s, S^s, \rightarrow^s >$
   output : A Test case suite $TC$
2  foreach transition $t = (l_k, l_{k+1}, ?a_k(p), \varphi_k, \varrho_k) \in \rightarrow^s$ labelled by an input $?a_k(p) = ?op(p_1, ..., p_n)$ do
       // in search of an executable preamble which reaches $l_k$
3      $tc$ is a new test case $< L^{tc}, l_0^{tc}, Var^{tc}, var_0^{tc}, I^{tc}, S^{tc}, \rightarrow^{tc} >$ with
       $var_0^{tc} := var_0^s$;
4      $p := \emptyset$ is a $tc$ path;
5      $p := PathFinding(l_0, l_k, p$,s);
6      foreach location $l$ of $p$ do
           // the incorrect behaviour of the preamble is added
7          if it exists a transition $(l, l', a(p), \varphi, \varrho) \in \rightarrow^s$ from $l$ to a location $l'$ labelled
           by fail then
8              $\rightarrow^{tc} := \rightarrow^{tc} \cup (l, l', a(p), \varphi, \varrho)$;
9      $Value(op) = \{(v_1, ..., v_n) \in V(type(p_1)) \times ... \times V(type(p_n)\}$;
       // test case construction for each value in $Value(op)$
10     foreach $(v_1, ..., v_n) \in Value(op)$ do
11         $tc' := tc$;
12         $\lambda_{tc'} := [p_1 := v_1, ..., p_n := v_n]$;
13         $\rightarrow^{tc'} := \rightarrow^{tc} \cup (l_k, l_{k+1}, ?a_k(p), \varphi_k \wedge \lambda_{tc'}, \varrho_k)$ is the transition set of $tc'$;
           // addition of all executable transitions labelled by an
              output
14         foreach transition $(l_{k+1}, l_f, !a(p), \varphi_{k+1}, \varrho_{k+1}) \in \rightarrow^s$ with $!a \in S_O^s$ do
15             $\rightarrow^{tc'} = \rightarrow^{tc'} \cup (l_{k+1}, l_f, !a(p), \varphi_{k+1}, \varrho_{k+1})$;
16         $TC := TC \cup tc'$;
```

Algorithm 2: PathFinding algorithm

```
1  $PathFinding$(location $l$, location $l_f$, test case path $p$, STS $s$) : $p$;
2  if $l \neq l_f$ then
3      foreach $t_i = (l, l_i, a_i(p), \varrho_i, \varphi_i) \in \rightarrow^s$ do
4          if $Solving(p.(l, l_i, a_i(p), \varrho_i, \varphi_i)) := \lambda \neq \emptyset$ then
5              $label(t_i) := visited$;
6              if $a_i(p) \in S_I^s$ then
7                  $t := (l, l_i, a_i(p), \varphi_i \wedge \lambda, \varrho_i)$
8              else
9                  $t := (l, l_i, a_i(p), \varphi_i, \varrho_i)$;
10             $PathFinding(l_i, l_f, p.t, s)$;
11             $label(t_i) := unexplored$;
12 else
13     return ($p$);
14     $Solving$(test case path $p$) : $\lambda$;
15     $p = (l_1, l_2, a_1(p), \varphi_1, \varrho_1)...(l_k, l_{k+1}, a_k(p), \varphi_k, \varrho_k)$;
16     $c = \varphi_1(var_0^s) \wedge \varphi_2(\varrho_1) \wedge ... \wedge \varphi_k(\varrho_{k-1})$;
17     $(x_1, ..., x_n) = solver(c)$ //solving of the guard $c$ composed of the external
       variables $(X_1, ..., X_n)$ such that $c(x_1, ..., x_n)$ true;
18     if $(x_1, ..., x_n) == \emptyset$ then
19         $\lambda := \emptyset$
20     else
21         $\lambda := \{X_1 := x_1, ..., X_n := x_n\}$
```

6. Test Verdict and experimentation

6.1. *Test case execution*

Test cases are generated and executed with an academic tool which is available in [Sal11]. The tester, illustrated in Figure 19, corresponds to a classical Java application or a Web Service which receives the URL of the Web Service to test and eventually its STS specification. The tester constructs test cases, as described previously, by means of an XML file modelling the unusual value set V. This file can be updated easily. While the test cases are executed, it analyzes the SOAP responses and finally gives a test verdict. A more complete report is also produced to show the obtained responses after each call. With this framework, we do not need a specific test platform where Web Services should be deployed. The tester can call them on any accessible server.

The tester yields the final verdict after having executed each test case : it successively calls an operation with parameters and waits for a response while following the corresponding test case branch. When a branch is completely executed, a local verdict is reached. We also set that quiescence (blocking state after a timeout) is observed after a timeout of 60s. For a test case t, we denote the local verdict $trace(t) \in \{\text{pass}, \text{fail}, \text{fail/available}\}$. The final verdict is given by :

Definition 6.1 *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 $t \in TC, trace(t) = pass$. The pass verdict means that the WS operations exit and are robust over TC,

– fail/available, if it exists $t \in TC$ such that $trace(t) = fail/available$, and it does not exists $t' \in TC$ such that $trace(t') = fail$. This verdict means that the Web Service is not robust but all its operations are available,

– fail, if it exists $t \in TC$ such that $trace(t) = fail$.

Web Service	operations	parameters	tests	faults
primeNumbersGen	1	1	10	7
youtubeDownloader	1	1	10	9
sendSMS	2	2,4	22	0
mapIPtoCountry	2	1,1	20	4
numberToWords	1	1	10	6
localTimeByZipCode	1	1	10	0
textToBraille	2	2,2	22	20
tConvertions	2	1,2	22	11
strikeIron	1	3	12	0
svideoWs	1	3	12	12
yellowPagesLookup	1	5	12	0
codeLookup(BLZ)	1	1	9	9
textCasing	2	1,2	19	0
dateFunctions	2	3,3	20	13
koVidya	2	3,3	20	0
postML	4	1,6,5,2	40	40
ServiceObjects	5	2,3,1,4,1	50	0

Figure 20 – Stateless Web Service robustness testing results

6.2. *Results and discussion*

We experimented the first testing method on several stateless Web Services proposed by the provider *Xmethods* [Xme10]. Most of them have revealed robustness issues (results given in Figure 20). In most cases, operations do not catch the triggered exceptions, and crash without constructing and returning any SOAP fault.

We applied the second method on two versions of the AWSECommerceService Service (see Figures 21, 22). Roughly 30 percent of the tests provided unexpected responses. With the hazard *Using unusual values*, and despite that all the requests satisfy the WSDL description, we obtained some SOAP faults composed of the cause *Client*, meaning that the request is incoherent. The receipt of this cause may be due to security rules (firewalls, etc.). We also received unspecified messages corresponding to errors composed of a wrong cause. For instance, we received 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. We also ob-

	09/03	09/10
Number of tests	100	100
Using unusual values	75	75
Replacing/Adding operation names	25	25
Fails	34	34
Unspecified messages	28	28
Unspecified SOAP faults	6	6

Figure 21 – Test results on the Amazon AWSECommerceService Services

Operation	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 22 – Detailed Test results

served the same situation with the hazard *Replacing /Adding operation names* only when the operation CreateCard is replaced by another one. So, by referring to the test results and by supposing there is no security rule modifying the tests, the AWSECommerceService service seems to be not robust.

These results have confirmed the following advantages :

– *effectiveness :* using the tester is quite easy since it can test automatically most of Web Services deployed over the Internet with only the WSDL description URL (of course those which do not use security layers). Formally, the test coverage of both methods is quite simple : for the first one, we check that after calling an operation with hazards, this latter does not hang or crash. The second method tests whether a Web Service behaves correctly despite the use of hazards. But, both methods can detect many other problems, like operation accessibility (operation does not exist, does not take the good parameter types), exception management problems (lack of ”try...catch” code, SOAP faults not sent), and observability problems (operations which do not respond). The methods are also scalable since the predefined set of values V can be upgraded easily,

– *test cost :* the test case number depends mainly on the parameter number, the operation number and the specification state set. So, the test case set may manifestly explode, especially when operations are called with a large variable set. This is why we implemented a heuristic which limits the test case set, by limiting the value number used for testing (see Section 5.1. When the number of test cases is limited to 200, testing one Web Service with our tool takes at most some minutes. The execution of 1500 tests require less than one hour. The whole test cost naturally depends on the test case number, but also on the time required to observe quiescence. We have set arbitrarily this time to 60s but it may be necessary to augment it.

– *test coverage :* the test coverage of the stateless Web service testing method depends on the Max parameter which represents the test number per operation. The higher the number of parameters is, the more difficult it will be to cover the variable space domain. This corresponds to a well-known issue in software testing. So, we have chosen a straightforward solution by bounding the test case number per operation. The Max value limits must be chosen according to the available time for test execution and also according to the number of parameters used with the Web service operations so that each parameter ought to be covered by a sufficient value set. For instance, for one operation composed of 4 parameters, each covered with at least 6 values, the Max parameter must be set to 1300 tests. Nevertheless, as it is illustrated in our results, a lower test case number (100 tests) is sufficient to discover robustness issues. However, it sounds more interesting to combine this approach with another solution such as *pairwize* testing which requires that for each pair of input parameters every combination of values of these two parameters are covered by a test case. With the stateful Web service testing method, according to Algorithm 1, all the operation invocations found in the initial Web service specification are covered by the tests. Nevertheless, the parameter coverage raises the same issue as previously.

We also experimented the stateless Web Service method on stateful ones. As it might be expected, only a part of the specification is covered, and usually this corresponds to the first Web Service location.

Pragmatically, we can test some specification locations with random parameter values until specific values would be required to reach other locations. With the Amazon Web Service, the stateless Web Service testing method is able to cover the specification until the CartCreate action, which is based on an authentication process. So, with this Web Service, roughly half of the specification is tested.

7. Conclusion

The WS-I basic profile, which gathers the SOAP protocol and the WSDL language, reduces the Web Service observability and leads to new issues for robustness testing. We have shown, in this paper, that few hazards can be really used : for stateless Web Services, only the hazard *Using unusual values* is relevant, whereas stateful Web Services can be tested with the hazard *Replacing name/Adding operation* too. The other ones are blocked by SOAP processors. These latter also change the observable Web Service behaviour by constructing messages instead of Web Services. So it is essential to take into consideration the SOAP environment while testing. This is what we did, with the two previous robustness testing methods.

Several issues remain open, especially when considering the test case generation algorithm. Our implementation can only handle collections of simple types (tabular, object composed of String, double, etc.) with simple guards and variable updates. Currently, constraint solvers give results only on simple types. Thus, if the parameter/response variable types are complex objects, a solution would be to spread and transform them into simple type lists. But this requires to manually transform the specification and the test cases.

Other perspectives could be considered, especially in relation to the set of unusual values V. This set can be manually modified but stays static during the test case generation. Furthermore, to avoid a test case explosion, the cardinality of V is reduced independently of the Web Service under test. It could be more interesting to propose a dynamic analysis of the parameter types to build a list of the most adapted values. The use of Web Service ontology, which describes the service semantic, may help. It could be also interesting to analyze the values leading to more errors while testing and to set a weighting at each of them.

We have also supposed that the observable messages are the SOAP responses. However, Web Services depend often on other servers, like database ones. These other messages are not currently considered in most of testing methods and in this work. However, taking into account such messages augments the Web Service observability and could help to refine the testing process.

Références

[Ama09] Amazon. Amazon e-commerce service (ecs), 2009.

[BBMP09] Cesare Bartolini, Antonia Bertolino, Eda Marchetti, and Andrea Polini. Ws-taxi : A wsdl-based testing tool for web services. In *ICST '09 : Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 326–335, Washington, DC, USA, 2009. IEEE Computer Society.

[BCM09] Fayçal Bessayah, Ana Cavalli, and Eliane Martins. A formal approach for specification and verification of fault injection process. In *ICIS '09 : Proceedings of the 2nd International Conference on Interaction Sciences*, pages 883–890, New York, NY, USA, 2009. ACM.

[BDTC05] Xiaoying Bai, Wenli Dong, Wei-Tek Tsai, and Yinong Chen. Wsdl-based automatic test case generation for web services testing. In *SOSE '05 : Proceedings of the IEEE International Workshop*, pages 215–220, Washington, DC, USA, 2005. IEEE Computer Society.

[BFPT06] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans. Audition of web services for testing conformance to open specified protocols. In R. Reussner, J. Stafford, and C. Szyperski, editors, *Architecting Systems with Trustworthy Components*, volume 3938/2006 of *LNCS*, pages 1–25. Springer-Verlag, 2006.

[BP05] Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 134–142, 2005.

[CS04] Christoph Csallner and Yannis Smaragdakis. Jcrasher : An automatic robustness tester for java. In *Software – Practice & Experience*, volume 34, pages 1025–1050, 2004.

[CSD^{+}97] David Cohen, Ieee Computer Society, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system : An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23 :437–444, 1997.

[DYZ06] Wen-Li Dong, Hang Yu, and Yu-Bing Zhang. Testing bpel-based web service composition using high-level petri nets. *edoc*, 0 :441–444, 2006.

[ES04] Niklas Een and Niklas Sörensson. An extensible satsolver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004.

[ES06] Onyeka Ezenwoye and Seyed Masoud Sadjadi. Enabling robustness in existing bpel processes. In Yannis Manolopoulos, Joaquim Filipe, Panos Constantopoulos, and José Cordeiro, editors, *In Proceedings of the 8th International Conference on Enterprise Information Systems (ICEIS-06)*, pages 95–102, 2006.

[FTdV06] Lars Frantzen, Jan Tretmans, and René de Vries. Towards model-based testing of web services. In Antonia Bertolino and Andrea Polini, editors, *In Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 67–82, Palermo, Sicily, ITALY, June 9th 2006.

[FTW05] L. Frantzen, J. Tretmans, and T.A.C. Willemse. Test Generation Based on Symbolic Specifications. In J. Grabowski and B. Nielsen, editors, *Formal Approaches to Software Testing – FATES 2004*, number 3395 in Lecture Notes in Computer Science, pages 1–15. Springer-Verlag, 2005.

[GFTdlR06] José García-Fanjul, Javier Tuya, and Claudio de la Riva. Generating test cases specifications for compositions of web services. In Antonia Bertolino and Andrea Polini, editors, *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 83–94, Palermo, Sicily, ITALY, June 9th 2006.

[HM09] Samer Hanna and Malcolm Munro. An approach for wsdl-based automated robustness testing of web services. In *Information Systems Development, Challenges in Practice, Theory, and Education*, volume 2, pages 493–504. Springer-Verlag, 2009.

[IEE90] IEEE. Ieee standard glossary of software engineering terminology. In *IEEE Standards Software Engineering 610.12-1990. Customer and terminology standards*, volume 1. IEEE Press, 1990.

[KGG$^+$09] Adam Kiezun, Vijay Ganesh, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. Hampi : a solver for string constraints. In *ISSTA '09 : Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 105–116, New York, NY, USA, 2009. ACM.

[KKS98] 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 Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 230, Washington, DC, USA, 1998. IEEE Computer Society.

[LZCH08] Mounir Lallali, Fatiha Zaidi, Ana Cavalli, and Iksoon Hwang. Automatic timed test case generation for web services composition. *ECOWS08, European Conference on Web Services*, pages 53–62, 2008.

[MS09] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In *30th IEEE Real-Time Systems Symposium (RTSS 2009)*, pages 105–116. IEEE Computer Society, December 2009.

[MX06] Evan Martin and Tao Xie. Automated test generation for access control policies. In *Supplemental Proc. 17th IEEE International Conference on Software Reliability Engineering (ISSRE 2006)*, Nov 2006.

[Oc02] OASIS-consortium. Uddi (universal description, discovery and integration) specification. 2002.

[OC07] OASIS-Consortium. Ws-bpel v2.0 specification. apr 2007.

[OX04] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. In *ACM SIGSOFT Software Engineering Notes*, volume 29(5), pages 1–10. ACM, 2004.

[Sal11] Sébastien Salva. Wsat, web service automatic testing, 2011.

[SKRC07] F. Saad Khorchef, Antoine Rollet, and Richard Castanet. A framework and a tool for robustness testing of communicating software. In *ACM SAC 2007*, pages 1461 – 1466, Corée, République de, mar 2007.

[SR09] Sebastien Salva and Issam Rabhi. Automatic web service robustness testing from wsdl descriptions. In *12th European Workshop on Dependable Computing, EWDC 2009*, June 2009.

[TFM05] Abbas Tarhini, Hacène Fouchal, and Nashat Mansour. A simple approach for testing web service based applications. In *5th International Workshop IICS, Paris, France*, pages 134–146, June 2005.

[Tid00] D. Tidwell. Web services, the web's next revolution. In *IBM developerWorks*, Nov 2000.

[Tre96] Jan Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17(3) :103–120, 1996.

[VLM07] Marco Vieira, Nuno Laranjeiro, and Henrique Madeira. Assessing robustness of web-services infrastructures. *Dependable Systems and Networks, International Conference on*, 0 :131–136, 2007.

[Wc03] WWW-consortium. Simple object access protocol v1.2 (soap). World Wide Web Consortium, June 2003.

[Wc07] WWW-consortium. Web services description language (wsdl). World Wide Web Consortium, 2007.

[WI06] WS-I. Ws-i basic profile. WS-I organization,, 2006.

[Xme10] Xmethods. *The Xmethods provider*. http ://www.xmethods.net, 2010.

ANNEXE POUR LA FABRICATION
A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE

1. ARTICLE POUR LA REVUE :

   Studia Informatica Universalis.

2. AUTEURS :

   Sébastien Salva * *— Antoine Rollet* **

3. TITRE DE L'ARTICLE :

   A pragmatic approach for testing stateless and stateful Web Service Robustness

4. TITRE ABRG POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :

   A pragmatic approach for testing...

5. DATE DE CETTE VERSION :

   22 septembre 2011

6. COORDONNES DES AUTEURS :

   – adresse postale :
   * LIMOS - CNRS UMR 6158
   Université d'Auvergne, Campus des Cézeaux,
   Aubière, France
   sebastien.salva@u-clermont1.fr
   ** LABRI CNRS UMR 5800
   University of Bordeaux
   33405 Talence cedex, France
   rollet@labri.fr

   – tlphone : 01 44 10 84
   – tlcopie : 00 00 00 00
   – e-mail : ivan.lavallee@gmail.com

7. LOGICIEL UTILIS POUR LA PRPARATION DE CET ARTICLE :

   LaTeX, avec le fichier de style `studia-Hermann.cls`,
   version 1.2 du 03/12/2007.

SERVICE ÉDITORIAL – STUDIA UNIVERSALIS
41 rue Gay Lussac, 75005 Paris
Tl : 01-44-10-84/83
mel : contact@complexica.net
Serveur web :
http://studia.complexica.net