Author manuscript, published in "12th European Workshop on Dependable Computing, EWDC 2009, Toulouse : France (2009)"

hal-00380717, version 1 - 4 May 2009

Automatic web service robustness testing from WSDL descriptions*

Sébastien Salva
LIMOS - UMR CNRS 6158
Université d'Auvergne, Campus des Cézeaux,
Aubière, France
Email: salva@iut.u-clermont1.fr

Issam Rabhi
LIMOS - UMR CNRS 6158
Université de Blaise Pascal, Campus des Cézeaux,
Aubière, France
Email: rissam@isima.fr

***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 used in large and heterogeneous applications, they need to be reliable. So, we propose in this paper, a robustness testing method which generates and executes test cases automatically from WSDL descriptions. We analyze the web service observability to find the relevant hazards which may be used for testing and those which are always blocked by SOAP processors. We show that few hazards can be really handled. By reducing them, we reduce the test cost too. We improve the robustness issue detection by separating the SOAP processor behavior from the web service one. With an academic tool, we show that many web services have robustness issues and that our method is able to detect them.**

Key-words: robustness testing, web services, test framework

I. Introduction

The web service paradigm is now well established in companies for developing business applications. These self-contained components may be used to provide interoperability between heterogeneous applications, to externalize functional code in a standardized way, or to compose 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 are often the foundation of large and complex applications. So, to finally produce reliable ones, software development companies follow software quality processes like the CMMI process (Capability Maturity Model Integration). And, of course, to ensure reliability, such quality processes are composed of testing activities. Testing web service conformity is required, but we believe that it is not sufficient. Indeed, web services 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*. In other words, they need to be robust.

Many web services are currently proposed in UDDI registers. For most of them, specifications or any information about their internal structures, are not given. So, keeping in mind these features, the aim of this paper is to test the web service robustness without specification, i.e by using only their WSDL descriptions. Web services are seen here like black boxes from which only requests and responses are observable.

The crucial issue, with black box web service testing, concerns the lack of observability. To respect the WS-I standards, they are inserted into several layers (HTTP, SOAP, client layers,...). Any message which is usually directly observable, like a classical response or a failure, is encapsulated (or not) and spread to the client over SOAP. For instance, according to the SOAP 1.2 protocol, exceptions, in object oriented programming, ought to translated into XML elements called SOAP faults. But this feature needs to be specified and implemented by hand in web services.

Consequently, we begin to analyze the web service observability over hazards to determine what kind of hazards 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. This analysis reduces the number of hazards used for testing, thus decreases the test case number and thus the test cost too. From this analysis, we describe an automatic robustness testing method whose the main purposes are to be mere, rapid and automatic. First, it checks if each operation described in the WSDL file exists and handles the correct value types. Then, the method tests the robustness of each operation by using a set of predefined values as hazards. We have implemented this method in an academic tool which has been used randomly on some web services deployed on Internet. Our results reveal that most of them have robustness issues.

This paper is structured as follows: section II provides an overview of the web service paradigm. We give some related works about web service testing and the motivations of our approach. Section III analyzes the web service robustness over the SOAP layer. Section IV describes the testing method: we detail the test case generation and a testing framework. Finally, section V gives some results, some perspectives and conclusions.

*This Research is supported in part by the French National Agency of Research within the WebMov Project http://webmov.lri.fr.

II. Web Service Overview

A. 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* models how to invoke a service set, called endpoints, and defines their interfaces and their parameter/response types. This description, called WSDL (Web Services Description Language) file [3], shows how messages must be structured 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 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: 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 registers. Web service descriptions are gathered into UDDI (Universal Description, Discovery Integration [5]) registers, 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, as database connections and the web service internal code are unknown. The only available details are the web service interfaces, given in WSDL files. 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. SOAP faults are not described in WSDL files.

Definition II.1 *A web service WS is a component which can be called with 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, $P(op) = \{(p_1, ..., p_m) \mid p_i \text{ is a value whose type is } param_i\}$. The set of response lists, denoted $R(op)$, is expressed with $R(op) = \{(r_1, ..., r_n) \mid r_j \text{ is a value whose the type is } resp_j\} \cup \{r \mid r \text{ is a SOAP fault}\} \cup \{\epsilon\}$. 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 $p \in P(op)$.

Note that 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, when there is no response, no SOAP message is sent to the client.

The parameter types are 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...). The response types are either simple, or complex or may be a SOAP fault.

A web service example is illustrated in figure 1 with UML sequence diagrams. This one has two available operations: "getPerson" which returns a Person object by giving a "String" and the operation "divide" which returns the integer result of a division. The WSDL description of the "getPerson" operation is given in figure 3. This one provides the exchanged message format. For a request, the message is composed of two elements "getPerson" and a "String". The response message is composed of two elements "getPersonResponse" and a Person objet. The Java code of the "getPerson" operation (figure 2), shows that two exceptions can be raised (ClassNotFoundException and SQLException), so that two different SOAP faults can be received after a "getPerson" invocation.

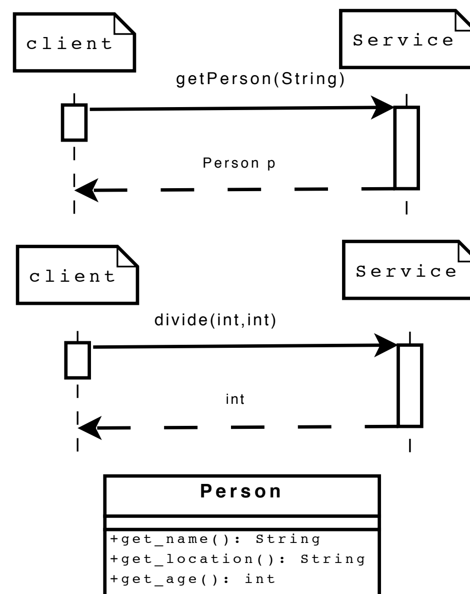

Fig. 1. Web service UML specification

B. Related work on web service testing

Some papers on web service testing have been proposed in [6-14]. Some of them consider compositions, where components are web services. System specifications, often expressed

hal-00380717, version 1 - 4 May 2009

```
Person getPerson(String name) {
try{
    p=new Persistent_Layer();
    Person pers=p.getperson(name);
    }
catch (ClassNotFoundException e)
    {throw new RemoteException("no
     Database driver found");}
catch (SQLException e)
    {throw new RemoteException("SQL
     error");}
return pers;
```

Fig. 2. The "getPerson" operation code

```
<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>
```

Fig. 3. WSDL description of the "getPerson" operation

by the UML or the BPEL languages describe the global system functioning by showing the possible interactions between the services. In [6], the BPEL specification is translated into the PROMELA language in order to be used by the SPIN model checking tool. In [7], the authors use BPEL specifications, describing web service compositions. Specifications are translated into Petri nets, then classical Petri net tools are used to study verification, testing coverage and test case generation. In [8], the system is represented by a Task Precedence Graph and the behavior of the composed components is represented by a Timed Labeled Transition System. Test cases are generated from these graphs and are executed by using a specific framework over SOAP. In [9], the BPEL specification is translated into an IF model, which enables modeling of timing constraints. Test case generation is based on simulation where the exploration is guided by test purposes.

Other works, about conformance, robustness and interoperability tests, focus on web services seen as black boxes. In [10], web service robustness is tested by performing mutations on the request messages and by analyzing the obtained responses. In [11], 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 [12]. In [13], web services are automatically tested by using only the WDSL 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 [14], 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 [15]. The authors of [16] propose a method to test automatically web service robustness. From a WSDL description, 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. In [17], fault injection techniques are employed to create diverse fault-trigging test cases in order to display possible robustness problems in the web-services code. The method in [18] discuss a technique of Web Services performance assessment taking out of the network delays. The main focus in [19], is on analyzing exception propagation and performance as the major factors affecting fault tolerance. In [20], the authors show how to use WS-FIT package to detect a problem in a web service based system. Their methods are used into, Compile-Time Injection (is an injection into the source code) and runtime injection.

As in [13], we check if each web service operation described in the WSDL file, exists, that is if each can be called with the parameter types given in the WSDL file and returns the good response types. We also analyze the web service behavior to determine that the only hazard which is not blocked by SOAP processors is the operation call with unusual values. The methods in [10], [16] use other hazards which are finally not relevant. The use of Axis in [16] adds a client layer with reduces the web service observability, this is why we have made our own testing platform which generates test cases and executes them. If fact, the method in [16] does not test directly web services but rather client methods which call web services. Our platform directly calls web service operations and analyzes the SOAP responses, and especially the SOAP faults. Indeed, it can detect another robustness problem which occurs when the exception management (error recovery) is not implemented by the web service itself but managed by the SOAP processor (the SOAP fault cause is different to "RemoteException").

hal-00380717, version 1 - 4 May 2009

hal-00380717, version 1 - 4 May 2009

III. WEB SERVICE ROBUSTNESS STUDY

Web services can be invoked only through a SOAP layer. This one reduces observability. So, we study, in this section, the web service robustness and the kind of hazards which can be really used for testing.

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 the a more complete framework like Apache Axis or Sun Metro JAXWS.

A. Web service Robustness and exception management

We suppose that a black box web service is robust if and only if its operations are robust. An operation is robust when this one does not hang or crash when it is invoked with hazards. Since web service are in a SOAP environment, when we call an operation, we call in fact the couple (SOAP processor, web service) and we obtain a response from it. We have observed that SOAP processors may affect the test result by improving the robustness. This improvement is observed when it returns a SOAP fault if the operation crashes. By observing this response, we conclude that the operation is robust despite it crashed. Some SOAP processors generate SOAP faults in some cases, other don't. So, we need to separate the SOAP processor behavior to the web service one.

The WS-I basic profile gives the required informations to differentiate the SOAP faults generated by SOAP processors from the SOAP faults constructed by web services. Indeed, 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. So, we can define a robust web service by:

Definition III.1 *A web service WS is robust if for each $(resp_1, ..., resp_n) = op(param_1, ..., param_m) \in OP(WS)$, op is robust, i.e $\forall v \in P(op), r = op(v)$ with:*

- $r = (r_1, ..., r_n)$ *such as* $r_i = resp_i$,
- *or r is a SOAP fault composed of the cause "RemoteException".*

To express this issue, consider the "divide" operation codes of figures 4, 5. When, we wish to divide an integer by 0, we observe different responses.

In figure 4, there is no exception in the "divide" operation. When a division by 0 occurs, the web service behavior may differ according to the web service framework used (Axis 1, Axis 2, JAXRPC or JAXWS libraries). On the one hand, the web service crashes without returning any result. A robustness issue is then detected. On the other hand, with other frameworks, the web service crashes but the SOAP processor returns a SOAP fault composed of the "divide / 0" message and of the cause "java.lang.ArithmeticException", which corresponds to the raised exception in the server side. In this case, the client receives a SOAP fault, but this is not thanks to the web service. Since the cause is not "Remote Exception" we detect that the operation is not robust.

The web service code of figure 5 describes a good exception management. When the exception is raised in the web service, this one spreads until the client thanks to the piece of code "throw new RemoteException("error divide"+x+" by "+y). This one produces one SOAP fault, composed of the message *"error divide"+x+" by "+y* and of the cause *java.rmi.RemoteException*. Here, the operation has itself managed the exception and is robust.

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

Fig. 4. Example I

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

Fig. 5. Example II

B. Analysis of black box web service behavior with hazards

Analyzing the web service behavior with the presence of hazards offers the advantage to know the hazards which are really given to the service and those which are blocked by SOAP processors. The blocked hazards are unnecessary for robustness testing since the service is not tested. Using only the hazards which test the service enables to reduce the test case number and the test cost.

For a web service operation $(resp_1, ..., resp_n) = op(param_1, ..., param_m)$, we have analyzed the following hazards. Note that the WS-I basic profile does not permit operation overloading. So, overloading is not dealt with here.

- **Replacing parameter types:** one or more parameter types in $(param_1, ..., param_m)$ 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 parameter types:** adding parameter types in the beginning of the request or between existing parameters is equivalent to *replacing parameter types*. So, as we have seen previously, this hazard is blocked. When, we call an operation by adding parameters at the end of the existing ones with $op(p_1, ..., p_m, p_{m+1}, ..., p_{m+k})$, the values $p_{m+1}, ..., p_{m+k}$ are not read by the SOAP processor. So, these ones are not given to the web service and are useless too. Therefore, this hazard is not relevant,

hal-00380717, version 1 - 4 May 2009

- **Deleting parameter types:** as previously, deleting parameters in the beginning or between existing parameters is comparable to *replacing parameter types* and is not relevant. When, we call an operation while deleting parameters at the end of the existing ones with $op(p_1, ..., p_k), (k < m)$, we obtain two kind of responses according the WSDL description. If the option "nillable=true" is used in the WSDL file, this is equivalent to call op with null values $(p_1, ..., p_k, null, ...null)$. We consider that a "null" value is an unusual one (see below). Otherwise, the SOAP processor returns a SOAP fault composed of the cause "Client" which means that the invocation is blocked. Thereby, either this hazard is comparable to *Calling with unusual values* or is unnecessary,
- **Inverting parameter types:** this hazard is comparable to *Replacing parameter types* and is unnecessary for testing,
- **Calling with unusual values:** this hazard, well-known in software testing [21], aims to call op with a type of 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. For instance, null, "", "$", "*" are some unusual "string" values. These ones are accepted by the SOAP processor and given to the web service since they satisfy the WSDL description. Thus, these unusual values can be used for web service robustness testing.

It exists of course other hazards on the SOAP messages, such as replacing the operation name, the port, modifying randomly the SOAP message. These hazards are usually used for testing web service compositions in order to observe partner behaviors. These ones are not interesting for testing only one web service. Indeed, when these hazards are injected in SOAP messages, either the test is then performed on another operation, or the SOAP processor returns that the operation or the service does not exist. Since we test all the operations and not the interoperability between several components, we do not need these hazards.

Consequently, the most relevant hazard for testing black box web services is "Calling web service operations with unusual values" since this hazard is the only one which is really given to operations. And this is the one which will be used in our approach.

IV. Automatic web service robustness testing

For a web service WS, our method aims at testing these two features:

- *Existence of all service operations*: for each operation $resp = op(param_1, ..., param_m) \in OP(WS)$, 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 with several values $(p_1, ..., p_m) \in P(op)$. op exists if op returns a response r. On the one hand, r may be a classical response $(r_1, ..., r_n)$ such as the type of each value r_i corresponds to $resp_i$ with $resp = (resp_1, ..., resp_n)$. On the other hand, r may be a SOAP fault where the cause is different from "Client" and "the endpoint reference not found". This first cause means the operation is called with bad parameter types. The second cause means that the operation name does not exist. Otherwise, op does not exist as described in the WSDL file,
- *robustness of all web service operations*: for each operation $resp = op(param_1, ..., param_m) \in OP(WS)$, we construct test cases to check if op does not crash or hang by calling it with hazards. According to our analysis of section III-B, the most relevant hazard corresponds to the call of op with unusual values $(p_1, ..., p_m) \in P(op)$ where p_i has the type $param_i$. So, we construct test cases with such unusual values. By executing them, either op should return a "classical" response, or op should return a SOAP fault whose the cause is equal to "RemoteException". We consider that the web service is not robust if no response is observed or if another kind of SOAP fault, constructed by the SOAP processor, is received.

To test these properties, we need to set an hypothesis on web services. We suppose that web service operations return no empty responses. Indeed, without response that is without observable data, we cannot conclude whether the operation is faulty or correct. So, if an operation does not return a response, we consider that it is faulty.

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

In the following, we present the test case generation in section IV-A, our testing framework and the test case execution in section IV-B.

A. Test case generation

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

Definition IV.1 *Let WS be a web service and $(resp_1, ..., resp_n) = op(param_1, ..., param_m) \in OP(WS)$ an operation of WS. A test case T is a tree composed of nodes $n_0, ..., n_m$ where n_0 is the root node and each end node is labeled by a local verdict in $\{pass, inconclusive, fail\}$. The branch tree are labeled either by $op_call(v)$ or by $op_return(r)$ 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 or $r = (r_1, ...r_m)$ is a list of responses where $r_j = (v_j, t_j)$ with v_j a value and t_j the type of v_j. We also denote $*$ any response value. $(*, t)$ is a response whose the type is t.*

For instance, $n_0 \xrightarrow{getperson_call("12345")} n_1 \xrightarrow{getperson_return(("*", String))} pass$ is a test case which invokes the $getperson$ operation with the parameter "12345". The response must be a String value.

Test case generation is illustrated in figure 6. We parse the web service WSDL file to list the available operations. Then, we use a predefined set of values V to generate test cases. This set contains for each type, an XML list of values that we use

hal-00380717, version 1 - 4 May 2009

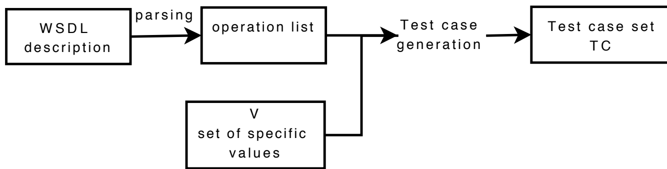

Fig. 6. Test case generation

for calling operations. Theses values have been chosen after the web service response analysis of section III-B in order to:

- obtain responses, whose types are described in the WSDL file, for checking that the operations exist,
- send an hazard. We have chosen, as hazards, the unusual values used in software robustness testing [21] which are assumed to have a high bug-revealing rate when used as inputs.

We denote $V(t)$ the set of specific values for the type t which can be a simple type or a complex one. Figures 8, 7 and 9 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>
```

Fig. 7. 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>
```

Fig. 8. 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>
```

Fig. 9. V(tabular)

For a web service WS, this method generates test cases with the following steps:

1) We parse the WSDL description to obtain the list of operations $L = \{op_1, ...op_l\}$,
2) for each operation $(resp_1, ..., resp_n) = op(param_1, ... param_m) \in L$, we construct, from the set V, the tuple set $Value(op) = \{(v_1, ..., v_m) \in V(param_1) \times ... \times V(param_m)\}$. If the parameter types are complex (tabular, objet,...), we compose these complex types with other ones 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 $Value(op)$,
3) for each operation $(resp_1, ..., resp_n) = op(param_1, ... param_m) \in L$, we construct the test case set $TC(op)$: $TC(op) = \bigcup_{v \in Value(op)} \{n_0.op_call(v).n_1.op_return(r_1).pass, n_0.op_call(v).n_1.op_return(r_2).pass, n_0.op_call(v).n_1.op_return(r_3).inconclusive\}$ where $r_1 = (*, t)$ with $t = (resp_1, ..., resp_n)$, $r_2 = (c, soap_fault)$ cause="RemoteException", $r_3 = (c, soap_fault)$ cause $\notin$ {"client", "RemoteException", "the endpoint reference not found"}. Any other branch corresponds to a fail case and is finished by "fail",
4) and finally, the test case set $TC = \bigcup_{op \in L} \{TC(op)\}$.

For more readability, we express the fail cases (the test case discovers a failure) with a dashed line in two separated figures, one for the operation existence testing (figure 10) and one for the robustness testing (figure 11). In TC, each tree calls an operation with authorized parameter values according to the WSDL description. If the response is not a SOAP fault and if its type is the one described in the WSDL file, the local verdict is "pass". If the response is a SOAP fault whose the cause is equal to "RemoteException" then the operation manages itself exceptions and the local verdict is "pass". If the response is a SOAP fault whose the cause is not in {"RemoteException","client", "the endpoint reference not found"} then the operation exists but does not manage exceptions. The operation crashes and a SOAP fault is returned by the SOAP processor. So this operation is not robust. In this case, the local verdict is "inconclusive". Otherwise, the local verdict is "fail".

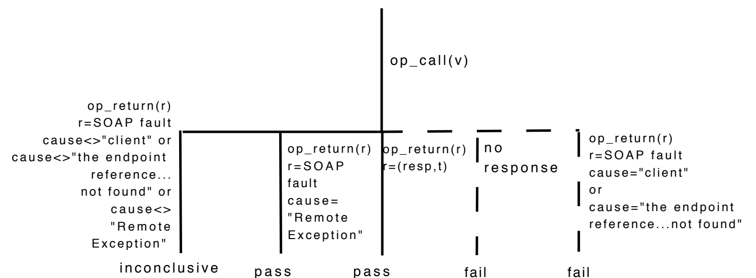

Fig. 10. Test case schema for testing the operation existence

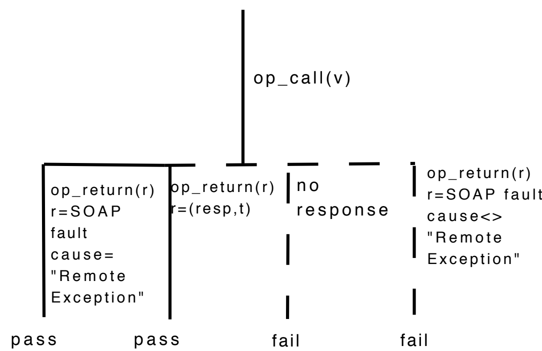

Fig. 11. Test case schema for testing robustness

B. Test case execution

Test cases are generated and executed with the testing framework, illustrated in figure 12, 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. It constructs test cases as described previously, and then calls successively the web service operations to execute them. Once test cases are executed, it analyzes the obtained responses and finally gives a test verdict. A more complete report is also produced to show the responses obtained after each call.

With this framework, we do not need of a specific test platform where web services should be deployed. The web service tester can call them on any accessible server.

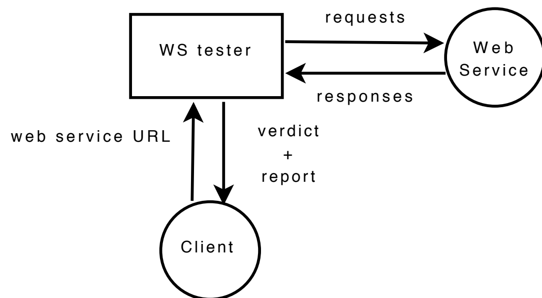

Fig. 12. Test architecture

To give the final 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 completely executed, a local verdict is obtained. Otherwise, the fail local verdict is given. For a test case t, we denote the local verdict $trace(t) \in \{\text{pass}, \text{fail}, \text{inconclusive}\}$.

The final verdict is given by:

Definition IV.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 $t \in TC, trace(t) = pass$. The pass verdict means that all the WS operation exit and are robust,*
- *inconclusive, if it exists $t \in TC$ such as $trace(t) = inconclusive$, and it does not exists $t' \in TC$ such as $trace(t') = fail$. This verdict means that the web service is not robust but all its operations exist.*
- *fail, if it exists $t \in TC$ such as $trace(t) = fail$.*

For instance, suppose that we wish to test the web service of figure 1:

- for the *getPerson* operation whose the code is illustrated in figure 2, the values of $V(String)$ will produce either a "String" response or a SOAP fault composed of the "RemoteException" cause since exception are correctly managed. So, this operation is robust,
- for the *divide* operation, the values used for testing are in the set $\{(0,0),(0,-1),(0,1),...,(RANDOM, RANDOM)\}$. For the tuple (0,0), the code of figure 4 will return either no response or a SOAP fault composed of the cause "Server" and of the message "divide/0". This SOAP fault is constructed by the SOAP processor, so this operation is not robust. And for the code of figure 5, we obtain the SOAP fault composed by the "RemoteException" cause which is constructed by the operation itself. So, no robustness issue is detected in this case.

V. Conclusion

The WS-I basic profile, which gathers the SOAP protocol and the WSDL language among others, reduces the web service observability. Few hazards are finally relevant for testing their robustness since most of them (replacing a type, inverting, adding deleting parameters) are blocked or deleted by SOAP processors. Only unusual values whose type is allowed by the WSDL description can be used. So, we have proposed to test the web service robustness by using this hazard only. This helps to reduce the test case number and the test cost as well. We have also improved the robustness issue detection by separating the SOAP processor behavior to the web service one. This method can be used to test black box web services or web services used in compositions.

We have successfully experimented this method randomly on some already deployed web services and have detected robustness issues for most of them. The obtained results are given in figure 13. In most cases, operations do not catch the triggered exceptions, and crash. This experimentation has confirmed the following advantages:

- *effectiveness:* the use of the tool is quite easy since the tester can test automatically most of web services deployed over Internet with only the WSDL description URL (of course those which does not use authentification). Formally, the test coverage of our method is quite simple: we check that after calling an operation with hazards, this one does not "hang or crash". However, the method can detect many problems, like operation accessibility (operation does not exist, does not handle the good parameters), exception management problems (lack of "try...catch" code, SOAP faults not sent), and

hal-00380717, version 1 - 4 May 2009

ws	operation number	parameter number	number of tests	fail
s1	1	1	10	7
s2	1	1	10	9
s3	2	2,4	22	0
s4	2	1,1	20	4
s5	1	1	10	6
s6	1	1	10	0
s7	2	2,2	22	20
s8	2	1,2	22	11
s9	1	3	12	0
s10	1	3	12	12
s11	1	5	12	0
s12	1	1	9	9
s13	2	1,2	19	0
s14	2	3,3	20	13
s15	2	3,3	20	0
s16	4	1,6,5,2	40	40
s17	5	2,3,1,4,1	50	0

Fig. 13. Robustness testing results

observability problems (operations do not respond). This method is also scalable since the predefined set of values can be upgraded easily,

- *test cost:* the method is mere and does not perform sequences of calls. So, the test cost depends only on the number of predefined values used for testing. To reduce the test case number, we have actually implemented an heuristic while their generation. Consequently, testing one web service with our tool takes some minutes.

However this experimentation has also revealed some drawbacks:

- the set V of parameters used for testing has been improved to detect more failures. But, it would be more interesting to propose dynamic analyzes to construct the most appropriate parameter list for each web service,
- to avoid the test case explosion, the list of parameters on V are chosen randomly. A better solution would be to choose these parameters according to the operation description,
- we have supposed that the operations of the same web service are independent. With dependent operations, the web service creates sessions with clients, becomes persistent in the web server and has different states during the session. Our tool is still able to detect robustness problems, but not all of them. For instance, if an operation checks that a session exists before using parameter values, the method cannot test it. So, we believe that a specification like a UML state diagram is required.

We have also supposed that the messages sent and received by web services are only SOAP messages. So, we have only considered their interfaces provided by the WSDL descriptions. This is true from the client side point of view. However, services can be connected to other servers, like database ones. These other messages are not currently considered in most of web service testing methods and in this work. So, in future works, we intend to consider web services not only as black boxes but rather as grey boxes from which any kind of messages could be observed.

References

[1] D. Tidwell, "Web services, the web's next revolution," in *IBM developerWorks*, November 2000.
[2] W.-I. organization, "Ws-i basic profile," 2006, http://www.ws-i.org/docs/charters/WSBasic_Profile_Charter2-1.pdf.
[3] W. W. W. Consortium, "Web services description language (wsdl)," 2001.
[4] ——, "Simple object access protocol v1.2 (soap)," June 2003.
[5] O. U. Specification, "Universal description, discovery and integration," 2002, http://www.oasisopen.org/cover/uddi.html.
[6] J. García-Fanjul, J. Tuya, and C. de la Riva, "Generating test cases specifications for compositions of web services," in *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, A. Bertolino and A. Polini, Eds., Palermo, Sicily, ITALY, June 9th 2006, pp. 83–94.
[7] W.-L. Dong, H. Yu, and Y.-B. Zhang, "Testing bpel-based web service composition using high-level petri nets," *edoc*, vol. 0, pp. 441–444, 2006.
[8] A. Tarhini, H. Fouchal, and N. Mansour, "A simple approach for testing web service based applications," in *5th International Workshop IICS, Paris, France*, june 2005, pp. 134–146.
[9] M. Lallali, F. Zaidi, A. Cavalli, and I. Hwang, "Automatic timed test case generation for web services composition," *Web Services, European Conference on*, vol. 0, pp. 53–62, 2008.
[10] 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.
[11] L. Frantzen, J. Tretmans, and R. de Vries, "Towards model-based testing of web services," in *in Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, A. Bertolino and A. Polini, Eds., Palermo, Sicily, ITALY, June 9th 2006, pp. 67–82.
[12] J. Tretmans, "Test generation with input, outputs, and repetitive quiescence," *Software - Concepts and Tools*, vol. 17, pp. 103–120, 1996.
[13] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen, "Wsdl-based automatic test case generation for web services testing," in *SOSE '05: Proceedings of the IEEE International Workshop*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 215–220.
[14] A. Bertolino, L. Frantzen, A. Polini, and J. Tretmans, "Audition of web services for testing conformance to open specified protocols," in *Architecting Systems with Trustworthy Components*, ser. LNCS, R. Reussner, J. Stafford, and C. Szyperski, Eds., no. 3938. Springer-Verlag, 2006, pp. 1–25. [Online]. Available: http://www.cs.ru.nl/ lf/publications/BFPT06.pdf
[15] A. Bertolino and A. Polini, "The audition framework for testing web services interoperability." in *EUROMICRO-SEAA*, 2005, pp. 134–142.
[16] 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)*, November 2006. [Online]. Available: http://www.csc.ncsu.edu/faculty/xie/publications/issre06-policytest.pdf
[17] 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)*.
[18] A. Gorbenko and al., "The threat of uncertainty in service-oriented architecture," in *SERENE '08: Proceedings of the 2008 RISE/EFTS Joint International Workshop on Software Engineering for Resilient Systems. ACM*, 2008.
[19] A. Gorbenko and al, "Experimenting with exception propagation mechanisms in service-oriented architecture," in *WEH '08: Proceedings of the 4th international workshop on Exception handling. ACM*, 2008.
[20] Looker, N., Munro, M., Xu, and J., "Ws-fit: A tool for dependability analysis of web services," in *in Proceedings of the 28th Annual International Computer Software and Applications Conference - Workshops and Fast Abstracts - (COMPSAC'04). IEEE Computer Society Press, Vol. 02.*, 2004.
[21] 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.

hal-00380717, version 1 - 4 May 2009