Automatic web service testing from WSDL descriptions

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

Antoine Rollet
LABRI - UMR CNRS 5800
Université de Bordeaux, Talence cedex, France
Email: rollet@labri.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. Currently, there is an important need for validation techniques of web service based architectures. Web services, that are currently proposed into UDDI registries, are not always tested. And for most of them, no specification is provided. So, we propose in this paper, a testing method which can generate test cases only from WSDL descriptions. This method is able to check the following aspects: operation existence, exception management, and session management. We express how to generate test cases and we describe a testing framework, composed of a web service tester, which executes test cases and gives the final test verdict.**

Key-words: conformance testing, web services, exception management, session management

I. Introduction

SOA (Service Oriented Architecture) is emerging as the standard paradigm to develop business applications over Internet, like Business to Business (B2B) and Business to Consumer (B2C) applications which involve the transaction of goods or services. Such applications are mainly based on web service interactions. Web services can be seen as components or objects whose methods, called operations, can be called over a network like Internet. A web service standard, composed of profiles, has been proposed by the WS-I consortium to promote web service interoperability. 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.

The web service paradigm is now well established in companies for developing business applications like banking systems, for externalizing functional code in a standardized way, for statistics, or for new web site generation, composed of dynamic services. These companies foremost want to use trustable web services, so process improvement approaches, like the CMMI (Capability Maturity Model Integration) are usually followed. Especially, the testing activity is a key step to obtain reliable and trustable web services.

However, testing web services brings new challenging issues, especially when we consider the WS-I profiles. Indeed, web services must be invoked by using SOAP messages. And these invocations are build by using WSDL descriptions. As a consequence, every event which is usually directly observable, like a classical response or a failure, may be translated (or not) according the WSDL descriptions and spread to the client over SOAP. For example, exceptions, in object oriented programming, are not directly observed but need to be translated into SOAP faults.

This paper proposes a method to test automatically the web service conformance from WSDL descriptions. We consider not having any specification of the web service to test, but a WSDL file which describes the web service interface. Without specification, a classical conformance testing method which constructs test cases from a specification, cannot be used. As a consequence, our proposal tests several web services properties. First, it checks if every operation described in the WSDL file exists and handles the correct value types. Then, our method tests the exception management: the WS-I basic profile indicates that exceptions are correctly managed when each raised exception produces a SOAP fault which is sent to the client. The testing method constructs test cases trying to force the web service to raise exceptions and checks if SOAP faults are sent. Finally, our method tests session management: sessions are introduced when different operations are interrelated. A good example would be banking applications where you log into the system, withdraw money, and log out. During the session time, data are stored by the web service. The method constructs test cases for setting specific data in the web service session and then for retrieving session data in order to check if the data read and written in the session are identical.

This paper is structured as follows: section II provides an overview of the web service paradigm and on some related works about web service testing. In section III we give our motivations and a general description of our approach. section IV describes the testing method: we detail how are generated the test cases and propose a testing framework. Finally, section V gives some perspectives and conclusions.

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" [Tid00]. To ensure and improve web service interoperability, the WS-I organization has proposed profiles, and especially the WS-I basic profile [org06], 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 [Con01], 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) [Con03]. 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 registries. Web service descriptions are gathered into UDDI (Universal Description, Discovery Integration [Spe02]) registries, which can be consulted manually or automatically by using dedicated APIs to find dynamically specific web services.
- the *service security*, which is obtained by using the HTTPS protocol or XML encryption.

In this paper, we consider black box web services, from which we can only observe SOAP messages. Other messages, 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 bellow, 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 [Con03], 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 $OP = \{op_1, ..., op_k\}$, 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 lists that op can handle, $P(op) = \{(p_1, ..., p_m) \mid p_i \text{ is a value whose the type is } param_i\} \cup \{\epsilon\}$. ϵ models an empty parameter (or no parameter). The set of response lists, denoted $R(op)$, is expressed with $R(op) = \{(r_1, ..., r_n) \mid r_j \text{ is a value whose the type is } resp_j\} \cup \{r \mid r \text{ is a SOAP fault}\} \cup \{\epsilon\}$.

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 parameters and/or do not return any response. With or without parameter, an operation is always called with a SOAP message. However, when no response must be given to the client, no SOAP message is received.

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. We consider, in this paper, that it does not exist any relation between the parameter or response types.

Web service interactions may be specified with some languages like UML or BPEL. A web service example is illustrated in figure 1 with two UML sequence diagrams. This one has four available operations: "getPerson" which returns a Person object by giving a "String", the operation "divide" which returns the integer result of a division and two setter/getter operations "set-PersonName" and "get-PersonName". 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 (ClassNotFoundEception and SQLException), so two different SOAP faults can be received after a "getPerson" invocation.

B. Related work on web service testing

Some papers on web service testing have been proposed in [GFTdlR06], [TFM05], [DYZ06], [OX04], [BDTC05], [BFPT06], [BP05], [MX06]. Some of them consider distributed systems, where components are web services. System specifications, often expressed by the UML or the BPEL languages describe the global system functioning by showing the possible interactions between the services. In [GFTdlR06], the BPEL specification is translated into the PROMELA language in order to be used by the SPIN model checking tool. In [DYZ06], the authors use BPEL specifications, describing web service compositions. Specification are translated into Petri nets, then classical Petri net tools are used to study verification, testing coverage and test

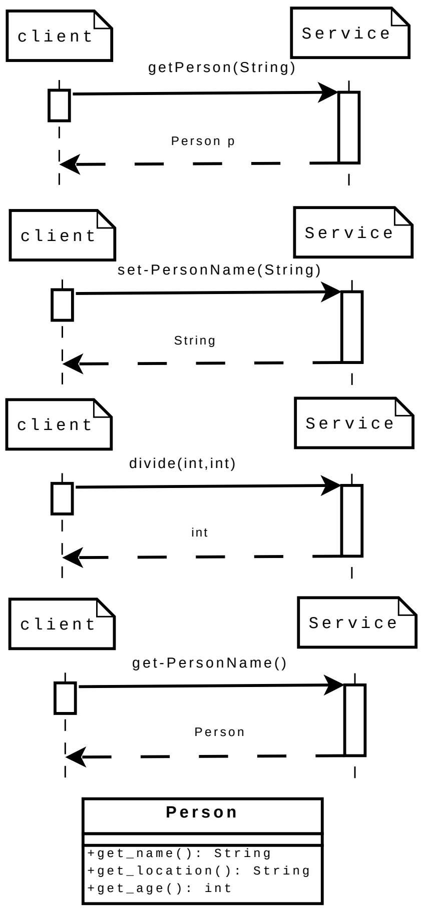

Fig. 1. Web service UML specification

```
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

case generation. In [TFM05], 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.

Other works, about conformance, robustness and interoperability tests, focus on web services seen as black boxes. In [OX04], web service robustness is tested by performing mutations on the request messages and by analyzing the obtained responses. In [FTdV06],

```
<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

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 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 [BFPT06], the authors proposes to 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 authors of [MX06] propose a method to test automatically web service robustness. From a WSDL description, the method use the Axis 2 framework to generate a class composed of methods allowing to call any service operation. Then, test cases are generated with tool Jcrasher, from the previous class. Finally, the tool Junit is used to execute test cases.

III. General description and motivations

The testing method in [BDTC05] already tests automatically web services from WSDL descriptions. Without specification, a classical conformance testing method, which constructs test cases from a specification, cannot be used. As a consequence, the method in [BDTC05] tests some web service properties. Our proposal aims to test others properties which are essential in web service development.

As in [BDTC05], 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. Then, we test the web service session management as well as the exception management. This is a major contribution of this paper.

- *web service session management testing*: sessions are introduced with web services when different operations are interrelated. During the session time, for a single client, data are stored by the web service. Usually, an operation is used to provide data to the web service and another operation is used to retrieve these data. We consider that the two operation names have the form "set-*operation-name*" and "get-*operation-name*. So, this part checks that, during a session and after giving some specific data to the web service with "set-*operation-name*", these data are persistent in the web service and can be retrieved with "get-*operation-name*".
- *exception management testing*: this part tests if exceptions are correctly controlled in web service codes. As described in the WS-I basic profile, when an exception is raised by an operation, a SOAP fault should be returned to the client. Despite the recent efforts of the web service frameworks, exceptions are not always correctly managed. So, when an exception is raised (Database connection error, thread creation error,...) the client is not always warned. To express this issue, consider the "divide" operation codes of figures 4, 5, 6. When, we wish to divide an integer by 0, we observe different responses.
  In the piece of code of figure 4, exceptions are not managed correctly: when the exception is raised, the web service simply returns the integer "1". This result does not meant an error for the client. The exception does not spread over the network because no SOAP fault is constructed and sent. So, the exception management is not correct.
  In figure 5, 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). The web service may crash without returning any result. This is the worst situation: the client has no result and is not warned. With other frameworks, the web service is stopped but the web server 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, so the exception is detected by the client. However, this is not thanks to the web service. Usually, when exceptions are not managed, the client may receive a SOAP fault composed of at least two elements, a message which equals to the raised exception message and a cause which equals to the raised exception name. In the worst case, it does not receive anything.
  The web service code of figure 6 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*. Usually, when exception are managed, the client receives a SOAP fault which is constructed by the web service. This one is composed of the raised exception message and the cause *java.rmi.RemoteException*. So, our testing method tries to raise exceptions in a web service by calling its operations with test cases composed of specific parameter values. Then, the method checks whether SOAP faults, constructed by the web service, are received.

Furthermore, to implement this method, we have made our own testing framework which generates test cases and executes them. Unlike [MX06], this one is able to directly call web service operations and to analyze SOAP responses. Indeed, the use of a framework like Axis2 [tasf04] to call operations, adds a layer which hides the receipt of messages like SOAP faults. We prefer calling the operations directly. We also use our own tool to generate test cases. The use of test tools like Jcrasher [SC07] is a good solution to test classical objects. But web services can be rather seen as several objects inside of a SOAP environment. So, we prefer using our own tool permitting to handle our own parameter values for testing. We can also change these values easily. This is not possible with Jcrasher.

```
Class Service {
public int divide (int x, int y) {
 int result=1;
 try {
    result=x/y; return result;}
 catch (Exception e) {
    return result; }
}
```

Fig. 4. Example I

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

Fig. 5. Example II

IV. Automatic web service testing

Many web services are currently proposed in UDDI registries. For most of them, specifications or any

```
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. 6.   Example III

information about their internal structures, are not given.

Without specification, it is not possible to test the complete web service conformance. Indeed, WSDL descriptions give only information about operations, parameter types and response types. These do not give any information about the web service behavior. The single solution would be to test with all the parameter values of the operations. For the "integer" type, the number of values is very large and for the "String" type, it is infinite. So, we cannot test all these values. Instead, for a web service WS, our method tests these different web service properties:

- *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 $(p_1, ..., p_m) \in P(op)$. If op returns a response r, we may have two cases. On the one hand, r is a correct response and equals to $(r_1, ..., r_n)$ such as the type of each value r_i corresponds to $resp_i$ with $resp = (resp_1, ..., resp_n)$. On the other hand, r is a SOAP fault where its own message is not composed of "Illegal Argument". Otherwise, op does not exist as described in the WSDL file. An "Illegal Argument" message is given when an IllegalArgument exception is raised in the web service. This one means that the parameters does not have the good types or that the number of the given parameters is not correct.
- *Exception management*: for each operation $resp = op(param_1, ..., param_m) \in OP(WS)$, we generate test cases for trying to raise exceptions in op by calling it with specific values $(p_1, ..., p_m) \in P(op)$. Without specification, we do not know the values which force the web service to raise exceptions. But, we use unusual values which should probably raise exceptions. For example, for the "String" type we use "", null, "*", "$". The *Exception management* test is performed while the *Existence of all service operations* test. If a SOAP fault is received we can check the exception management. If a "classical" response is received, we can check the operation existence.

  More precisely, after calling the operation, if no response is received, the web service has crashed without returning any SOAP fault. Otherwise, a SOAP fault should be received, giving some details about the exception (causes, values,...). If each SOAP fault is composed of the "RemoteException" cause, the web service manages exception correctly (see example 6). Otherwise, if some SOAP faults are not composed of the "RemoteException" cause, some exceptions are not well spread to the client. So, the exception management is faulty. Finally, if a correct response is received (not a SOAP fault), we cannot conclude anything about the exception management. But we can check that the operation interface is as described in the WSDL file.
- *Session management*: if WS has two operations named by the expressions $set - [opname]$ and $get - [opname]$, where $[opname]$ is an operation name, we suppose that WS uses a session. $set - [opname]$ is called to give data to the web service and $get - [opname]$ is called to retrieve it. We generate test cases to check if data are persistent, by calling $set - [opname]$ with a parameter list $(p_1, ..., p_n) \in P(set - [opname])$ and by checking whether the retrieved data equals to $(p_1, ..., p_n)$. If the retrieved data are different, the session management is faulty.

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. Note we don't make any hypothesis on the operation determinism.

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,*
- *$r = (m, soap\_fault)$ is a SOAP fault composed of the message m 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 example, $n_0 \xrightarrow{getperson\_call("12345")} n_1 \xrightarrow{getperson\_return(("*",String))} pass$ is a test case which invokes the $getperson$ operation with the parameter "12345". The response must be a String value.

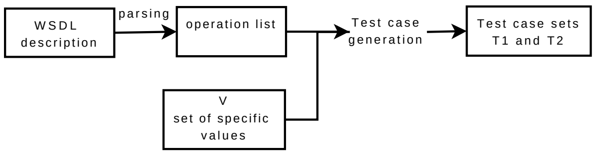

Fig. 7. Test case generation

Test case generation is illustrated in figure 7. We parse the web service WSDL file to list the available operations. Then, we use an existing set of values V to generate test cases. This set contains for each type, an XML list of values that we use for calling the operation. Theses values, permitted in the WSDL description, have been chosen to:

- obtain responses, whose the types are described in the WSDL file, for checking that the operations are available.
- try to provoke exceptions in the web service in order to check whether SOAP faults are returned.

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, 9 show the specific values used for the type "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="String">
        <val value=null />
        <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 by 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 ones (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 a first test case set T_1 to test the *operation existence* and the *exception management*.
$T_1 = \bigcup_{v \in Value(op)} \{n_0.op\_call(v).n_1.op\_return(r_1).pass, n_0.op\_call(v).n_1.op\_return(r_2).pass\}$
where $r_1 = (*, t)$ with $t = (resp_1, ..., resp_n)$, $r_2 = (m, soap\_fault)$
with "IllegalAgumentException" not in the SOAP fault message and cause composed of "RemoteException".
Any other branch corresponds to a fail case and is finished by "fail". The test case schema is illustrated in figure 10.
4) For each $(op_i, op_j) \in L^2$, if the name of op_i has the form $get-[op-name]$ and op_j has the form $get-[op-name]$, we construct a second test case set T_2 to test the session management.
$T_2 = \bigcup_{v \in Value(op)} \{n_0.set-[op-name]\_call(v).n_1.set-[op-name]\_return(r_1).inconclusive,$
$n_0.set-[op-name]\_call(v).n_1.set-[op-name]\_return(r_2).n_2.get-[op-name]\_call().n_3.get-[op-name]\_return(r_1).inconclusive,$
$n_0.set-[op-name]\_call(v).n_1.set-[op-name]\_return(r_2).n_2.get-[op-name]\_call().n_3.get-[op-name]\_return(r_3).pass\}$ where
   - $r_1 = (m, soap\_fault)$ with "IllegalAgumentException" not in the SOAP fault message and cause composed of "RemoteException",
   - $r_2 = (*, (resp_1, ..., resp_n))$,
   - $r_3 = (u, t)$ with $u = v$.

   As previously, any other branch corresponds to a fail case and is finished by "fail". The test case schema of T_2 is depicted in figure 11.

For more readability, we express the fail cases (the test case discovers a failure) with a dashed line in the schemas of figures 10 and 11. In T_1, each tree calls the operation with authorized parameters. 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 message is not composed by "IllegalArgument" and whose the cause expresses a "RemoteException" then the operation manages exceptions correctly and the local verdict is "pass". Otherwise, the local verdict is "fail".

In the same way, each test case of T_2 tries to set a value v, with the $set - [opname]$ operation. Then, it calls the $get - [opname]$ operation. If the response value equals to v then the web service session works perfectly. So, the the local verdict is "pass". If after calling the operation $set - [opname]$ or $get - [opname]$, the response is a SOAP fault where either the message is not composed of "Illegal Argument" or the cause is composed of "RemoteException", the web service does not seem faulty but the session cannot be tested. So the local verdict is "inconclusive". For any other response, either these two operations are not implemented as described in the WSDL file or the session management is faulty.

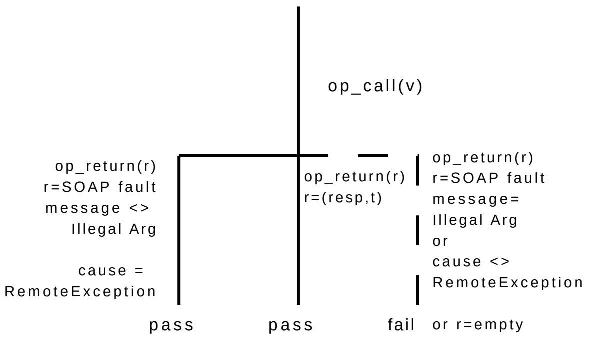

Fig. 10. Test case schema of T_1

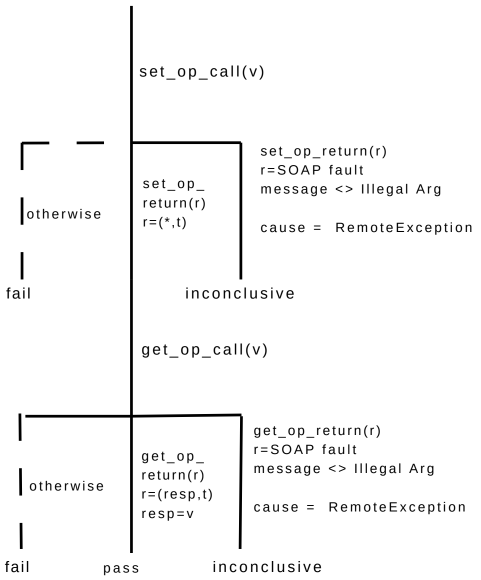

Fig. 11. Test case schema of T_2

Consider our web service example in figure 1. For the operation *Person getPerson(String)*, the testing method handles the value set $V(String")$ and generates the test cases illustrated in figure 12. For the two operations *String set-PersonName(String)* and *Person get-PersonName()*, the method generates a T_2 test case set to test the session management. Test cases are illustrated in figure 13.

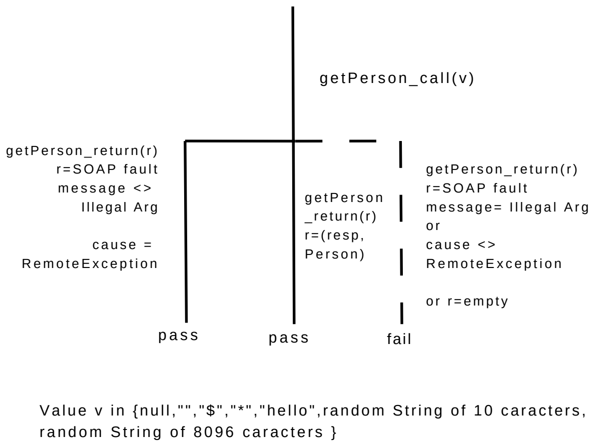

Fig. 12. Test cases generated from the "getPerson" operation

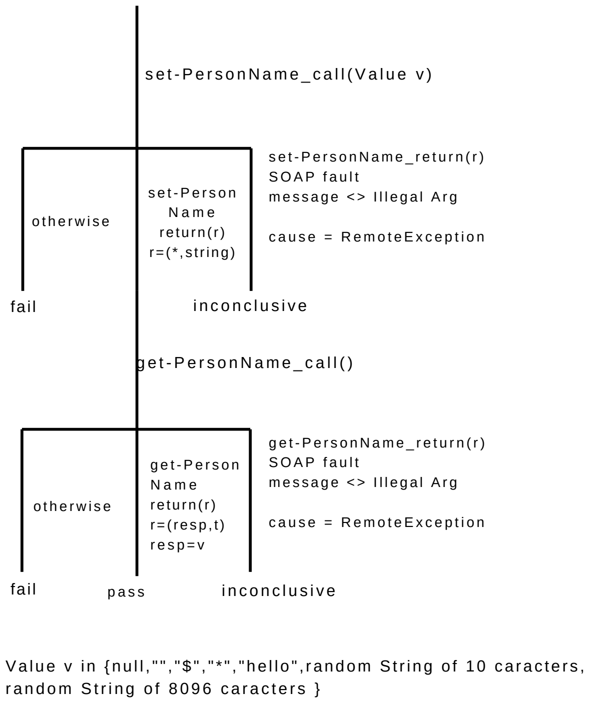

Fig. 13. Test cases generated from the "set-PersonName" and "get-PersonName" operations

B. Test case execution

Test cases are generated and executed with the testing framework, illustrated in figure 14, 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 test cases by constructing or analyzing SOAP messages. 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.

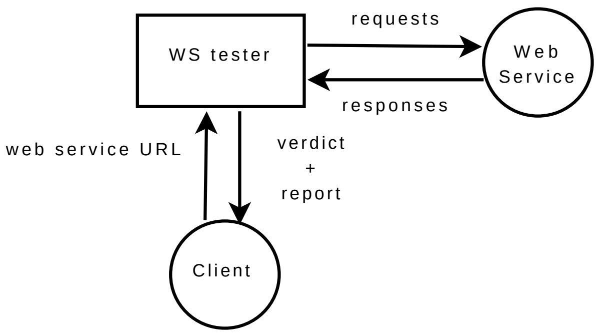

Fig. 14. Test architecture

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 T be a test case set. The verdict of the test by using T, over the set of values V, denoted $Verdict(T)/V$ equals*

- *pass, if for all $t \in T, trace(t) = pass$,*
- *inconclusive, if it exists $t \in T$ such as $trace(t) = inconclusive$, and it does not exists $t' \in T$ such as $trace(t') = fail$,*
- *fail, if it exists $t \in T$ such as $trace(t) = fail$.*

When an inconclusive verdict is given, an expert would analyze the SOAP faults inside the report given by the tester. He could determine the causes of the raised exceptions and eventually could conclude whether the web service is faulty. He could also modify the V set and run another test.

Suppose that we wish to test the *getPerson* operation with the test cases of figure 12. The values null and "", which can be used to call *getPerson* according the WSDL description, should raise an exception while querying the database. So, a SOAP fault should be received. By calling *getPerson* with the other values, we should receive either a Person objet or a SOAP fault. If no response is received, the web service has probably crashed and is faulty. If the response is a SOAP fault composed of the "Illegal Argument" message, the parameter values are refused by the operation. In this case, the web service interface does not correspond to the WSDL file. This one is still faulty. If the SOAP fault cause is not composed of the "RemoteException" message, the exception management is incorrect.

V. Conclusion

Web service are often defined as objects which can be accessed over a network like Internet, and testing them does not seem difficult. However the WS-I basic profile, which gathers the SOAP protocol and the WSDL language, brings some new issues. For example, the raised exceptions, which represent a part of the web service behavior, should be observed by the client with a SOAP fault. However, we have shown in this paper that exceptions are not always observed from the client when they are not correctly managed in the web service. So, our testing method can be used to test this exception management automatically, but also the session management and the web service operation existence.

We have successfully experimented this method on some real web services (*Amazon Associates* web service and some on *webservicex.net*) and we have detected an incorrect exception management on one of them (test cases have revealed that soap faults are not constructed by the service itself). The use of the tool is quite easy since only the WSDL description URL is required for testing. However the experimentation has revealed some drawbacks:

- the set R of parameters used for testing has been improved to detect more failures. But, it would be more interesting to propose dynamic analyzes, as in software testing, to construct the more appropriate parameter list for each web service,
- to avoid the test case explosion, the list of parameters on R are chosen randomly. A better solution would be to choose these parameters according the operation description,
- except for the session management test, we have supposed that the operations of the same web service are independent. Indeed, testing dependent operations without any specification is a big challenge because we do not have the order of the successive calls. A basic solution would be to find the operation dependence graph, while testing, by calling a list of successive operation and by analyzing the observed response. We need to investigate this possibility.

It could be also interesting to analyze the web service paradigm to determine if other properties could be tested. For example, the observability and the controllability are two essential properties required to improve the fault detection during the testing process.

A preliminary step could check automatically if web services are observable and controllable.

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, or to other web services (service composition). These other messages are not currently considered in most of web service testing methods and in this work. In the same way, the web server internal messages are not considered too. 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

[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*, number 3938 in LNCS, pages 1–25. Springer-Verlag, 2006.

[BP05] Antonia Bertolino and Andrea Polini. The audition framework for testing web services interoperability. In *EUROMICRO-SEAA*, pages 134–142, 2005.

[Con01] World Wide Web Consortium. Web services description language (wsdl). 2001.

[Con03] World Wide Web Consortium. Simple object access protocol v1.2 (soap). June 2003.

[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.

[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.

[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.

[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)*, November 2006.

[org06] WS-I organization. Ws-i basic profile. 2006. http://www.ws-i.org/docs/charters/WSBasic_Profile_Charter2-1.pdf.

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

[SC07] Yannis Smaragdakis and Christoph Csallner. Combining static and dynamic reasoning for bug detection. In *Proc. International Conference on Tests And Proofs (TAP)*, volume 4454 of *LNCS*, pages 1–16. Springer, February 2007.

[Spe02] OASIS UDDI Specification. Universal description, discovery and integration. 2002. http://www.oasisopen.org/cover/uddi.html.

[tasf04] the apache software foundation. Axis. 2004. http://ws..apache.org/axis/.

[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*, November 2000.

[Tre96] J. Tretmans. Test generation with input, outputs, and repetitive quiescence. *Software - Concepts and Tools*, 17:103–120, 1996.