2013 IEEE Ninth World Congress on Services

Using Rules for Web Service Client Side Testing

Nabil El Ioini, Alberto Sillitti, Giancarlo Succi
Faculty of Computer Science, Free University of Bolzano
Piazza Domenicani 3, 39100, Bolzano, Italy
{nelioini, asillitti, gsucci}@unibz.it

***Abstract*— Web Services (WS) are software components accessible over the Internet through a well-defined set of standards. When consumers invoke a service, they expect to receive a valid response. However, the problem is to determine the structure of a valid request [21]. WS specifications are used to solve this problem since they are considered the primary piece of information for building service requests. Unfortunately, existing specifications do not provide enough support for this type information (e.g., WSDL) or there is little support on the client side (e.g., OWL-S). In this paper we address this issue by implementing a technique to reduce the number of faulty requests. We specifically propose an approach for extending WSDL with service input parameters rules that help consumers and integrators to verify their calls on the client side.**

Keywords- Web Services; Testing; Client Side; Annotation; WSDL

I. INTRODUCTION

The recent urge in providing fast and standard way for integrating heterogeneous systems has motivated leading companies and the research community to increase their interest in improving Web Services technologies [4, 14]. The focus is mainly on the infrastructures that manage the integration [23], with less attention on other aspects such as security and testability [7, 19]. Web services are self-contained software components that are integrated at runtime to accomplish specific tasks [6]. The main difference with traditional software components relates to the unpredictability of the system behavior at runtime since they are executed over the Internet on different execution environments [17]. This is one of the main challenges in WS development [8, 22]. To tackle this issue, different studies propose the use of testing [7]. While testing is generally performed in controlled environments before releasing software products, WS development require testers to additionally test services at runtime to validate their behavior in the production environment. Moreover, this behavior can even change from one call to another since WS rely on distributed execution [8]. To this end, the WS community is working on developing new testing strategies, which support the fact that we need to perform testing on the production environment as well [8].

To test WS, developers need to address new challenges, such as who is going to test them (provider, customer, third party), when, and how much it costs in terms of money and effort. Moreover, the evolution of services increases the level of complexity, since consumers might face unexpected behavior when a new version of the service is released [9, 25, 26].

In this paper, we focus on two problems: 1) how service providers can take advantage of the historical data of services invocation to build a model for the services' input parameters; 2) how service consumers can use such a model to check the services requests preventing faulty requests and reducing the time and the money related to wrong requests.

The reminder of the paper is structured as follows. Section 2 discusses the related work; Section 3 provides an overview of the proposed approach; Section 4 describes the implementation details; Section 5 presents the experimental evaluation; finally, Section 6 draws the conclusions and outlines future work.

II. RELATED WORK

Extending existing specifications is a challenging task since it may affect the existing infrastructures that already use them. Extensions generally present new and/or enhanced features that were not considered at the time of the original development of a specification [24]. In the context of web services, specifications are developed to guarantee the highest level of interoperability between service providers and their consumers. The specifications need to provide the necessary means for a consumer to invoke a service without any further information [13, 16].

As new requirements emerge, new extensions are needed to address them. One such requirement is testing. While some studies rely entirely on the existing specifications [3, 4], other researches considered them to be lacking useful information for testing services [3]. Consequently, they have proposed a variety of extensions to enrich them.

One of the first extensions reported in the literature was applied to the Universal Description Discovery and Integration (UDDI) registry. The extension consists of adding a verification step to test services before adding them to the registry [2]. Before a service is accepted into the registry a test bed is generated and executed to assess the functional compliance of a service. Service providers have to develop such test bed to be allowed to publish their services on the registry.

978-0-7695-5024-4/13 $26.00 © 2013 IEEE
DOI 10.1109/SERVICES.2013.63

G. Dai *et al.* [12] expressed the rights and obligations that consumers need to respect to interact with a service using OWL-S. In this way, the process for checking the validity of the consumer request and the provider response can be automated.

In [3], W. T. Tsai *et al.* proposed the extension of the Web Service Definition Language (WSDL). They have considered four extensions: input-output dependency, invocation sequence, hierarchical functional description, and sequence specifications. Input-output dependency adds information regarding any connections between input and output messages (e.g., if the output message of an operation is used as input for another one). Invocation sequence, defines the order of services invocation in case the service needs to invoke other external services. Hierarchical functional description defines relations between operations and messages and other services. Sequence specification defines the order of operations inside a WSDL file.

III. PROPOSED APPROACH

To better understand the idea of the proposed approach, let us consider the following scenario. A travel agency application uses a web service for booking hotels for the clients. The service provider charges the travel agency based on the number of requests sent. Consequently, the travel agency needs to make sure that the requests sent respect the service call constraints specified by the service provider to avoid paying for faulty requests. For instance, if the travel agency sends the wrong check-in date (e.g., check-in date after the check-out date) the service will return an error. However, the agency still needs to pay for the call. A second case is where the whole booking process takes a considerable amount of time (let us assume that it takes 2 minutes). If the travel agency makes a faulty call, it does not want to wait 2 minutes to discover that the call was faulty and it needs to send another request. For both cases, the issue is how the travel agency can be sure that only valid requests are sent. The current approach to this scenario is by conforming to the provider specifications. This is done by reading the documentation provided by the service provider, then implementing the necessary mechanisms to verify that the constrains hold before sending the request. However, this approach requires a lot of work from the service consumer, since it needs to be done manually and for each new service.

Our approach automatically validates the consumers' requests on the client side to avoid the problems described above. It is a type of conformance testing that checks if the client input data could generate a faulty request.

Our design is based on the idea of collaboration between providers and consumers to reduce the number of faulty requests. On the provider side, the architecture requires a number of components that interact to generate the necessary information, which is made available to consumers to annotate their service calls implementation.

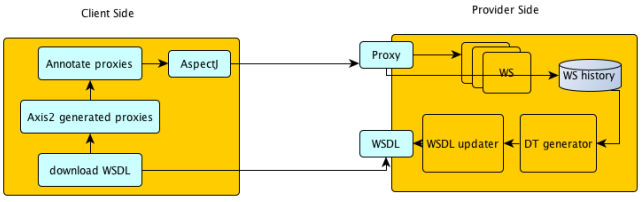

Figure 1. Overall Architecture

Figure 1 shows the general architecture of the approach. The provider side performs most of the tasks. The interaction starts at the provider side where the service provider collects service invocations data, namely service requests input parameters and responses messages. Once collected, service requests and responses are stored in WS history database, which is fed as input to a Decision Tree (DT) generator. The DT generator builds a decision tree, which represents the different classes that the combination of the input parameters leads to. Afterwards, it generates the parameters rules from the DT and writes them in the WSDL file.

Consequently, it generates a new version of the WSDL file the consumers can use to annotate their service calls. The annotation part is done manually in the current implementation but we are working on automating it.

In the following sub-sections we go through the different components and show how they fit in the overall architecture.

A. Data Collection

Data collection represents the first step of the entire process. Data is collected at the provider side for two reasons: 1) providers can easily collect requests and responses; 2) providers can collect data from different consumers [15].

The data of interest at this stage is the requests and the related responses for each service [20]. Input parameters and their values represent the service request and response messages represent the service response.

For new services that have no real data yet, providers could execute a test bed that generates an initial data set to be used by the other components.

B. Decision tree generation

Decision trees are used for creating a mapping between the values of different attributes and output classes [25]. The example shown in figure 2 shows a decision tree constructed using a service that takes three parameters (name, age, and salary) with their data types (string, integer, and double respectively). In this example, based on the combination of the parameters, there are 6 different classes. In a real scenario we could have much more classes. For example specific combinations of parameter values can generate some distinct errors, which may be defined as separate classes.

In a similar fashion our approach takes the data collected and builds a decision tree that classifies the different requests based on their input values.
Once the data is collected, it is passed to the decision tree generator as input. The decision tree generator component reads all the log data and builds a decision tree. The initial branching points of the tree are the input parameters (e.g. age, salary), and depending on parameters type and values more branches are created (e.g. 15<age<100). The leaf of the tree on the other hand is the decision that the combination of parameters leads to. As a result the component is able to classify the requests collected by associating them to one of the generated classes.

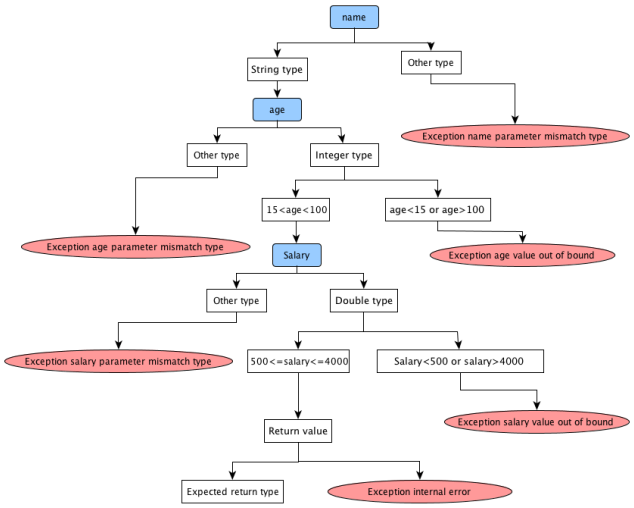

Figure 2. Decision Tree Example

C. *Rules extraction*

The decision tree tells us what are the possible classes based on the input values. However, consumers are not interested in all the classes. They are interested in what makes their requests valid requests. For this purpose, the next step is the extraction of all the rules that help consumers build valid requests.
For instance, in the tree in Figure 2,we can extract the following rule:

```
Name  = String AND 15 < age < 100  AND
        500 <= salary <=  4000
```

This rule represents the values of the parameters that generate a valid response based on historical data. By having this combination of parameters, the client should receive the expected value.
The two cases to consider while building the rules are:

a. The attribute is numeric: the tree generates different intervals for which the attribute leads to an acceptable class. For this case all the intervals found will be connected with the logical operator AND.
For instance: 100 < salary < 300 and salary != 200.

b. The attribute is non numerical: the attribute values will be connected with the logical operator OR.
For instance: Country = USA OR Country= ITALY.

c. All the attributes are connected to each other with the logical operator AND.

D. *Update WSDL*

WSDL is the standard interface used to publish the WS information, so instead of using a separated method for publishing the rules extracted from the decision tree, we make it easier for consumers by having that rules as part of the WSDL. WSDL provides a predefined structure that can be extended with custom data. Our approach augments WSDL by adding a new attribute to each operation. The new attribute represents the validation expression associated with every operation (Figure 3).

```
<wsdl>
<operation
          name="operation1"
          expression="EXPRESSION1">
         …
</wsdl:operation>

<operation
          name="operation2"
          expression="EXPRESSION2">
         …
</wsdl:operation>
```

Figure 3. Expression attribute added to operations

E. *Annotation*

Annotation is the final phase, where the consumer uses the rules published in the WSDL. The purpose of this phase is to use the information published by the provider to annotate the WS services calls. We are using Aspect Oriented Programming (AOP) syntax since we can use existing tools to check the validity of the constraints before calling the service call. This means that for each service call, the parameters passed will be checked against the constraint published in the WSDL. The service call will be executed only if they do not violate the constraints.

IV. IMPLEMENTATION

The system is implemented using a combination of existing open source components. Figure 4 presents the components used and how they are connected.

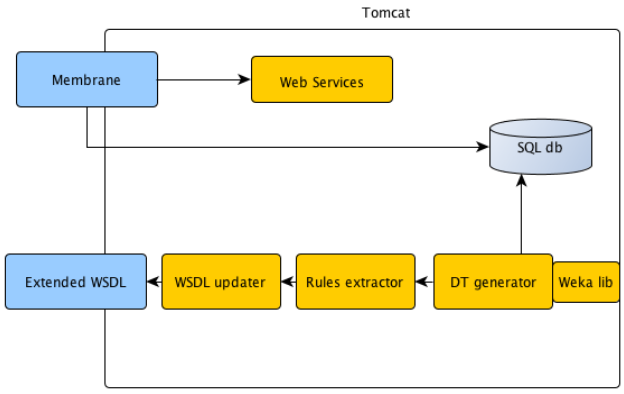

Figure 4. Implementation Overview

To generate an initial dataset for building the decision tree, an ad-hoc WSDL testing tool is used. The tool uses the XML schema defined inside the WSDL and generates test data for each data type. The rules for the data generation can be configured. For instance, we can configure it to generate boundary values for each primitive type, or generate random values, or specify manually the values to assign to each data type. Once the data is generated, test cases for each service call are generated with the different combinations of the data values. For instance, if we specify the values manually we can have a configuration file that looks as Figure 5. In this figure we specify that whenever a Boolean type is used at least two cases need to be generated for example.

```
<Rules>
   <Datatype name="Boolean">
      <Case value="True" />
      <Case value="False" />
   </Datatype>
<Datatype name="String">
      <Case value="null" />
      <Case value="" />
      <Case value="(*&^%$#@!@#$%^&^$#%^&^%$#@@*(*&" />
      <Case value="SPACE" />
   </Datatype>
</Rules>
```

Figure 5. Configuration file for test data generation

A. Data collection

To collect the data of the service calls we used a proxy to intercept the requests/responses going through the provider infrastructure. We store the data on the provider side, then we use it to build the decision tree. In the current implementation, we are using membrane router [1] as proxy. Membrane router is composed of three main parts: 1) the EndpointListner, which waits for the incoming messages, 2) the EndpointSender, which sends the messages to their destination, and 3) a set of interceptors in between the two end points. In the current implementation, we have two interceptors to capture the request parameters and the response messages.

B. Java Code

For the three components decision tree generator, rules extractor, and WSDL updater, we developed a Java project to handle the three operations.

1) DT generation

The decision tree generator uses the Weka library J48 [10]. It takes a dataset from the web services execution logs as input. The dataset is pre-processed by extracting input parameters and response messages and formatting them to be compatible with Weka. Weka, generates a tree model with the classes as leafs. We used the fault attribute in the SOAP responses as errors' classes. If the response does not contain a fault attribute we consider it to be a correct response. It is important to mention that if the service has some problem with the business logic, it cannot be found by this approach. Our approach relies strongly on the exceptions thrown by the services.

2) Rules extraction

The next step consists of extracting the relevant information, which is given to the consumers to annotate their service calls. The rule extraction component does it by executing the following algorithm:

```
select all the OK rules
Foreach argument a in the argumentList
        foreach rule r in the ruleList
            extract condition c related to a in r
            conditionList += c
        done
done
```

This algorithm first selects only the rules that have positive responses (their arguments are acceptable by the WS). The outer loop iterates over the arguments list, while the inner loop iterates over the rules. For each argument the condition is extracted and it is concatenated to the list of conditions. We have selected the positive rules to provide consumers with the values that are most likely to be accepted by the services.

3) WSDL updater

The WSDL updater component is responsible of parsing the WSDL operations and adding the rules expressions to each operation. The extension is added in the form of an attribute to the operation tag. Each operation is extended with an expression. For instance, in the example bellow we have extended the operation *CalculateTax* with the expression generated from the DT in Figure 2. In this example, the operation *CalculateTax* expresses the type and values acceptable by the operation. The logical operator AND (&&) is used to concatenate the different terms of the expression.

```
<wsdl:operation name="CalculateTax"
   expression="name=String &&
   age > 15 &&
   age < 100 &&
   salary >= 500 &&
   salary <= 4000">
  <wsdl:documentation>Calculate Tax for a
          specific person</wsdl:documentation>
                .....
                .....
    </wsdl:operation>
```

4) Client side annotation

On the client side, consumers need to annotate their java methods that invoke service calls. The annotation is done manually for the current implementation; however, it is possible to autormate it and we are working on that. The annotation consists of taking expressions published in the WSDL and adding them as constraints to the method calls, so that the method will not be executed unless it satisfies the constraints. We have used Aspect Oriented Programming (AOP) [11] to do that. More specifically, we have used the AspectJ framework [11] to implement our annotation approach.

Using AspectJ, the user can take the extended WSDL and add the constraints to the Java methods as follows:

1. Define a pointcut for each method: in an AOP a pointcut means that we define the behavior to be executed when a matching expression is found. For instance the following pointcut states that when a method with the a matching signature *.*sayHello(..)* is executed, call *method1* before executing *.*sayHello(..)*

```
pointcut method1() : execution(*
        *.sayHello(..)) ;
```

2. Validating the method parameters against the expression from the WSDL: in AspectJ this is done using *around*, which defines a custom behavior to perform before executing a method. For instance, in the following code snippet the parameters of method1 are checked. The *exectractValue* method is a user method responsible for extracting the value passed to the *sayHello* method, which is then checked against the string *"value"*. The expressions from the WSDL need be validated inside this method. If the method parameter satisfies the condition the *around* returns *proceed()*, which means execute the *sayHello* method, otherwise return FALSE which blocks the execution.

```
Object around() : method1() {
  Object[] paramValues= hisJoinPoint.getArgs();
  String[] paramNames=
    ((CodeSignature)thisJoinPointStaticPart.
          getSignature()).getParameterNames();
  if(extractValue( paramNames,
              paramValues).equals("value"))
      return proceed();
  else
      System.out.println("The method 1 does not
                    satisfy the condition");
      return "FALSE";

}
```

V. EXPERIMENTAL EVALUATION

To evaluate our approach we have devised an experiment that puts in action all the pieces together.

A. Subject Web Services

We tested our approach on two web services. The first one called HotelBooking, which is a simple service developed to handle hotels booking. It uses mock objects to simulate the booking process. The second service is a freely available service called HolidayWebService [1] that provides information about holidays in the US and the UK. More than a thousand requests have been generated and executed for each service.

1) HotelBooking Service

In the case of the HotelBooking service, we generated random data values for each data type, however, later we made some changes to make sure that we cover some special cases. For instance, the check-in date and check-out date need to be checked in different situations such as checkin > checkout, checkin = checkout, and checkin < checkout.

2) Holiday Service

In the case of HolidayWebService, we combined random values with manually generated values. The reason for this is that the service has a set of pre-defined expected values that need to be used.

B. Test cases

The process starts by generating the test cases. For this purpose we used an ad-hoc tool which takes a WSDL file as parameter and generates a set of test cases for each operation. The tool works as shown in Figure 6.

In Figure 6 the WSDL is the input to our tool, which extracts all the operations and their parameters. For each parameter, we defined a set of rules. They are basically the possible values for each primitive data type. The last step consists of generating service calls that use different combinations of the data values generated. In a real scenario this tool will be used in the initial stage when a new service is deployed and the provider does not have any historical data to use to build the decision tree model.

---

1 http://www.holidaywebservice.com/Holidays/HolidayService.asmx?WSDL

Otherwise, in case the WS is already in use, the historical data will be used.

Once the test cases are ready, the tool executes the service calls and logs the service's responses. Each log entry contains the parameters used to send the request and the response message received from the server.

To increase the test coverage of the two services we added some test cases manually based on the services documentation. For instance, in the Holiday Web Service, some parameters have a set of predefined values, and if we rely only on the random values, most of the test cases will be rejected by the service.

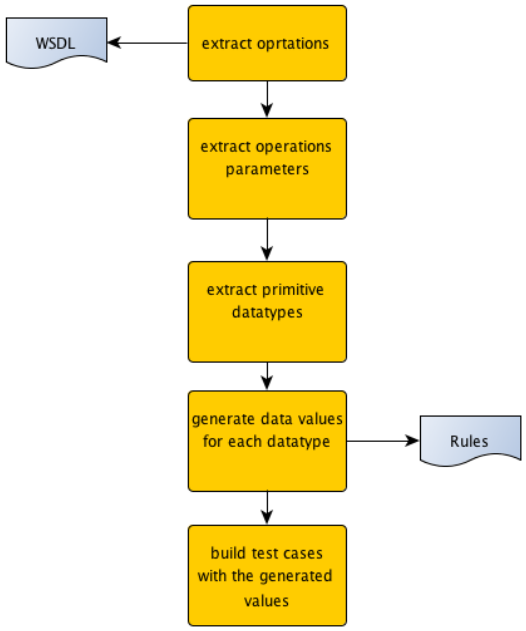

Figure 6. Test Cases Generation Tool

We divided the experiment into three phases:

1. Generation of the test data and call the service to build an initial decision tree.
2. Generation of a second set of test cases to check if further refinement to the generated decision tree could be added.
3. Application of the annotations to the Java methods and generation of a set of test cases, which would violate the annotations. Then, execution of those tests cases directly against the services to check the annotations coverage.

C. Experiment Phases

1) Phase 1

In the first phase we generated random and manual test cases for each of the two services (Table I).

TABLE I. RANDOM AND MANUAL TEST CASES (PHASE 1)

Service Name	Number of Parameters	Random Tests	Manual Tests
HotelBooking	3	900	50
HolidayWebService	3	900	100

The result obtained is a the following expressions (Table II).

TABLE II. RESULTS OF TEST CASES EXECUTION (PHASE 1)

Service	Expression
HotelBooking	P1 is String AND P2 is Date AND P3 is Date AND P2 < P3
HolidayWebService	(P1 = US OR P1 = GBEAW) AND (P2 = NEW_YEAR OR MLK) AND P3 > 0

The initial test bed allowed us to have a set of rules to start with. Later we need to check if this set of rules can be improved.

2) Phase 2

In the second phase, we generated more test cases to cover additional scenarios to check if the tool is able to discover them (Table III).

TABLE III. RANDOM AND MANUAL TEST CASES (PHASE 2)

Service Name	Number of Parameters	Random Tests	Manual Tests
HotelBooking	3	500	30
HolidayWebService	3	500	50

After executing the test cases and rebuilding the decision tree, we extracted the following expressions (Table IV).

TABLE IV. RESULTS OF TEST CASES EXECUTION (PHASE 2)

Service Name	Expression
HotelBooking	P1 is String AND P2 is Date AND P3 is Date AND P2 < P3
HolidayWebService	(P1 = US OR P1 = GBEAW) AND (P2 = NEW_YEAR OR P2 = MLK OR P2 = MEMORIAL OR P2 = MOTHERS) AND P3 > 0

In the second phase, we noticed that the expressions have been updated with new terms added to the expression.

3) Phase 3

In this phase we applied the generated expressions to the java methods for calling this services as follows:

```
Object around() : method1() {
 Object[] paramValues= thisJoinPoint.getArgs();
 String[] paramNames=
    ((CodeSignature)thisJoinPointStaticPart.
          getSignature()).getParameterNames();

 if((paramsValues[0] instanceof String) &&
   paramsValues[1] instanceof   DateTime &&
   paramsValues[2] instanceof DateTime &&
   paramsValues[1].compareTo(paramsValues[1]) < 0)
       return proceed();
 else {
        System.out.println("The method 1 does not
                    satisfy the condition");
        return "FALSE";
       }
```

}

The expression in the rectangle represents the part of the code that checks the parameter values.
We generated again random and manual test cases to test the java calls with and without the annotations to check the effectiveness of our approach (Table V).

TABLE V. RANDOM AND MANUAL TEST CASES (PHASE 3)

Service Name	Number of parameters	Random Tests	Manual Tests
HotelBooking	3	500	30
HolidayWebService	3	500	50

The results of executing the test cases with and without the annotated methods are presented in the following (Tables VI and VII).

TABLE VI. RESULTS OF TEST CASES EXECUTION WITH ANNOTATIONS

Service Name	Number of tests	Percentage of tests violating the expressions
HotelBooking	530	70%
HolidayWebService	550	78%

The next step, we re-executed the test cases that have violated the expressions directly against the web services. The results we have obtained are the following (Table VII).

TABLE VII. RESULTS OF TEST CASES EXECUTION WITHOUT ANNOTATIONS

Service Name	Number of tests	Percentage of tests violating the expressions
HotelBooking	371	0%
HolidayWebService	429	6%

From the Table VII, we can see that 6% of the tests that have violated the expression were still be able to pass the tests when executed directly against the web services. This 6% represents the false positives, which are a result of some gaps in the training data used to generate the decision tree. When the training data is not uniformly distributed throughout the parameters domain, which means having requests that represent all the different combinations of parameters, there is a chance that some requests will be misclassified. For instance, if we consider a request with the parameter age and in the training data we do not use any requests with age > 100, the resulting expression will consider the requests with age > 100 to be valid although they will be rejected by the web service.

D. Limitations

To use this approach effectively, some limitations need to be understood and addressed to reduce the misclassification of incoming requests. The main limitation concerns the quality of the collected data and the decision tree generated [18]. As mentioned earlier, the data collected consists of service requests issued by the service consumers or generated by the service provider. However, it might happen that in some cases the data does not represent all the classes of the possible parameters combinations, which leads to misclassifying some requests. For instance, in Figure 2 if we do not have any requests with the parameter salary < 500, the generated tree will also consider requests with salary < 500 to be valid requests, even if this will lead to an exception.
One way of overcoming this limitation is by generating more robust test cases at the beginning or using the service without annotation for a period of time to be able to collect realistic data to use for the DT generation.

VI. CONCLUSIONS

Reducing the number of faulty requests executed by service consumers consists a major factor for reducing consumers' costs and services traffic [27, 28]. However, the level of detail missing from the existing specifications require much more work from the consumer to understand which requests are acceptable by a service. Extending the existing specifications with the necessary information for consumers benefits both the consumer and the provider. In this paper, we have presented an extension to the WSDL specification, which adds a model extracted from the historical data of a service call to help consumers avoid making the same faulty requests. The experiments we devised show the benefit of using the extra information. As a future work, we intend to automat the part for the code annotation to have a fully automated process, as well as experimenting the approach with larger scale web services.

REFERENCES

[1] A. Bertolino, L. Frantzen, A. Polini, J. Tretmans, "Audition of Web Services for Testing Conformance to Open Specified Protocols," In Proceedings of the 2004 international conference on Architecting Systems with Trustworthy Components, Ralf H. Reussner, Judith A. Stafford, and Clemens A. Szyperski (Eds.). Springer-Verlag, Berlin, Heidelberg, 2004.

[2] A. Bertolino, A. Polini, "The audition framework for testing Web services interoperability," Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on , vol., no., pp. 134- 142, 30 Aug.-3 Sept. 2005.

[3] W.T. Tsai, P. Ray, Y. Wang, C. Fan, D. Wang, "Extending WSDL to Facilitate Web Services Testing," High-Assurance Systems Engineering, IEEE International Symposium on, p. 171, 7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02), 2002.

[4] E. Martin, S. Basu, and T. Xie, "§," Web Services, IEEE International Conference on, Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 647-654.

[5] E. Martin, S. Basu, and T. Xie, "WebSob: A Tool for Robustness Testing of Web Services," International Conference on Software Engineering Companion, Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 65-66.

[6] W3C. Web Services Architecture. Nov. 2002. W3C Working Draft 14, available at: http://www.w3.org/TR/2002/WD-ws-arch- 20021114/.

[7] M. Bozkurt, M. Harman, Y. Hassoun, Testing web services: a survey, Technical Report TR-10-01, Department of Computer Science, King's College London, 2010.

[8] G. Canfora, M. Di Penta, “Testing services and service centric systems: Challenges and opportunities”. IT Professional, 8(2):10-17, March/April 2006.

[9] V. Borovskiy, A. Zeier, "Evolution Management of Enterprise Web Services," Advanced Management of Information for Globalized Enterprises, 2008. AMIGE 2008. IEEE Symposium on , vol., no., pp.1-5, 28-29 Sept. 2008.

[10] Ian H. Witten, F. Eibe, “Data Mining: Practical Machine Learning Tools and Techniques”, Second Edition (Morgan Kaufmann Series in Data Management Systems). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[11] W.L. Dong, H. Yu, Y.B. Zhang, "Testing BPEL-based Web Service Composition Using High-level Petri Nets," Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International, vol., no., pp.441-444, Oct. 2006.

[12] G. Dai; X. Bai; Y. Wang; F. Dai, "Contract-Based Testing for Web Services," Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International, vol.1, no., pp.517-526, 24-27 July 2007.

[13] WSDL [http://www.w3.org/TR/wsdl]

[14] F. Maurer, G. Succi, H. Holz, B. Kötting, S. Goldmann, and B. Dellen. “Software process support over the Internet”. In Proceedings of the 21st international conference on Software engineering, 1999.

[15] N. El Ioini, A. Garibbo, A. Sillitti, G. Succi, "An Open Source Monitoring Framework for Enterprise SOA", 9th International Conference on Open Source Systems (OSS 2013), Koper, Slovenia, 25 - 28 June 2013.

[16] N. El Ioini, A. Sillitti, "Open Web Services Testing", 2011 IEEE International Workshop on Web Services / Cloud Services Testing (WSCS-Testing 2011), Washington DC, USA, 4 - 9 July 2011.

[17] E. Damiani, N. El Ioini, A. Sillitti, G. Succi, "WS-Certificate", 2009 IEEE International Workshop on Web Services Security Management (WSSM 2009), Los Angeles, CA, USA, 6 - 10 July 2009.

[18] H.G. Gross, M. Melideo, A. Sillitti, "Self Certification and Trust in Component Procurement", Journal of Science of Computer Programming, Elsevier, Vol. 56, pp. 141 - 156, April 2005.

[19] P. Predonzani, A. Sillitti, T. Vernazza, "Components and Data-Flow Applied to the Integration of Web Services", The 27th Annual Conference of the IEEE Industrial Electronics Society (IECON'01), Denver, CO, USA, 29 November - 2 December 2001.

[20] M. Scotto, A. Sillitti, G. Succi, T. Vernazza, “A non-invasive approach to product metrics collection”, Journal of Systems Architecture, Volume 52, Issue 11, November 2006.

[21] J. Clark, C. Clarke, S. De Panfilis, G. Granatella, P. Predonzani, A. Sillitti, G. Succi, T. Vernazza, “Selecting components in large COTS repositories”, Journal of Systems and Software, Volume 73, Issue 2, October 2004.

[22] A. Valerio, G. Succi, M. Fenaroli, “Domain analysis and framework-based software development”. *SIGAPP Appl. Comput. Rev.* 5, 2, September 1997.

[23] GL Kovács, S Drozdik, G Succi, P Zuliani, “Open source software for the public administration”, Proceedings of the 6th International Workshop on Computer Science and Information Technologies, 2004.

[24] L. Corral, A. Sillitti, G. Succi, “Mobile Multiplatform Development: An Experiment for Performance Analysis”, Procedia Computer Science, Volume 10, 2012

[25] Andrea A. Janes and Giancarlo Succi, “The dark side of agile software development”. In Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software, 2012.

[26] E, di Bella, I. Fronza, N. Phaphoom, A. Sillitti, G. , J, Vlasenko, "Pair Programming and Software Defects - A Large, Industrial Case Study," *Software Engineering, IEEE Transactions on* , vol.PP, no.99, pp.1,1, 0.

[27] I. Fronza, A. Sillitti, G. Succi, J. Vlasenko, “Does Pair Programming Increase Developers' Attention”, Industrial Track of ESEC/FSE2011, Szeged, Hungary, 2011.

[28] A. Sillitti, G. Succi, J. Vlasenko, "Toward a better understanding of tool usage: NIER track," *Software Engineering (ICSE), 2011 33rd International Conference on* , vol., no., pp.832,835, 21-28 May 2011.