Fault-Based Combinatorial Testing of Web Services

Bellanov Apilli[1], Lydia Richardson[2], and Cory Alexander[2]
[1]Department of Computer Science, North Carolina State University, USA
[2]Department of Information Technology, Georgia Southern University, USA
bsapilli@ncsu.edu, {lr00018, calexan3}@georgiasouthern.edu

***Abstract*—The Internet houses diverse applications (i.e., banking, networking, etc.), commonly implemented as web services. Web services are flexible but can become complex, making it difficult to assure their quality. We propose fault-based combinatorial testing and compare its fault-detection capability to existing web service testing techniques.**

I. Introduction

The Internet houses diverse applications, ranging from online auctioning, banking, networking, to many other applications. These applications contain multiple software components that interact with one another. This interaction is an example of a web service, where the components send and receive messages to or from other components. Having multiple components allows web services to be flexible as opposed to having one large application. Flexibility comes at the cost of complexity. Web services are constantly being modified. For instance, updating current components or adding new ones. These changes can bring an unprecedented level of complexity, making it difficult to ensure certain levels of quality. One solution to this problem is to test web services as these changes are being introduced. There currently exist intelligent techniques to test web services but such techniques may be in their infancy [2], [3], [5], [4]. We propose a fault-based combinatorial testing approach, where we combine fault-based and combinatorial testing techniques. We purposefully introduce combinatorial faults to a web service, and invoke the tests on the web service. We then determine if the tests exposed the fault that we introduced. This helps fortify web services by assuring that certain faults cannot exist within them.

II. Web Services & Applications

Web services are defined as being the server component in a client-server relationship where exchanged messages are written in `XML` [8]. A client-server relationship could be described using a web based mail service such as Yahoo. For instance, a user, the client, communicates with Yahoo, the server. Web applications are applications that are accessed via a web browser over a network. They receive input from a client and produce output. Multiple web applications are integrated in a web service, allowing web applications from different sources to interact on a network. For instance, an application executing on a Linux machine can interact with an application executing on a Windows machine, allowing web services to be portable. Communication between applications is best defined by a `Sender`, the server, and `Receiver`, the client, relationship.

A. Defacto Standards of Web Services

Web services are built on `SOAP` (Simple Object Access Protocol) [1]. `SOAP` is a communication protocol that allows the transfer of data in `XML` over the Internet, enabling different applications on different operating systems with different languages to communicate with each other [1]. The inputs and outputs of each application within a web service are wrapped through the `SOAP` protocol into input and output messages, illustrated in Figure 1. These messages are in a format reconizable by other web services.

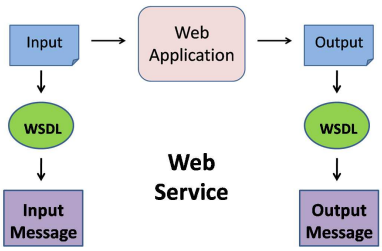

Fig. 1. Input & Output Messages

Clients for web services are specified in a `WSDL` (Web Service Description Language) file. This file defines the necessary operands required for communication with another web service. Tied with `WSDL` is `UDDI` (Universal Description Discovery and Integration). `UDDI` specifications store information about web services, such as location and requirements, in a format any web service can recognize, enabling them to interact [7].

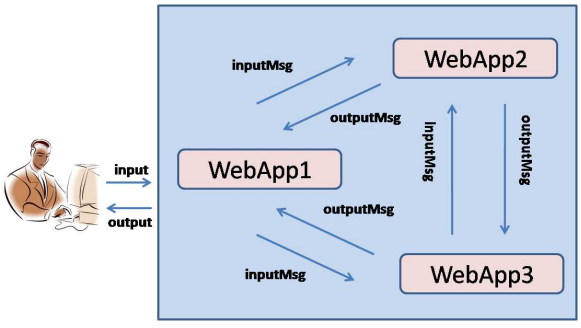

Fig. 2. Client accessing with a web service.

Figure 2 illustrates a user accessing a web service. Input and output messages are denoted as `inputMsg` and `OutputMsg`,

respectively. All that is visible to the user is their provided input and the output they receive from the web service. The number of components interacting within a web service is unknown to the user and may also be unknown to testers. It is difficult to assure the quality of a web service based on simple inputs and outputs. Thus intelligent testing techniques are required.

B. Existing Testing Techniques

Exhaustive testing suggests testing for every possible combination of inputs [2]. It may be difficult, even impossible, or too costly to test for every possible input. For instance, an input parameter defined as a person's first name. Unless a set containing every possible configuration of characters exists, it is impossible to test for every combination associated with this input. Considering the complexity in web services, an exhaustive approach may be infeasible. A feasible approach that is in its infancy with web services is combinatorial testing. In this approach, test inputs are generated considering interactions between input parameters [3]. This mechanism minimizes the number of tests by focusing on, for example, 2-way interactions. As opposed to the exhaustive approach of testing every possible combination, all 2-way combinations are tested. Combinatorial testing minimizes the number of tests, but may not be effective on systems with few inputs.

A validation framework, `iTac-Qos` (iTac Tests and Certifies Quality of Services), has proved to be an effective technique to testing web services [5]. This tool requires a model of the web services in order to generate tests. It utilizes a `UDDI` server, where web services are registered, deployed, and evaluated [5]. Although this tool proved to be effective, it is heavily depenedent on models of web services, which may not always be available.

Fault-based testing techniques have been applied as means of testing web services. In the data pertubation technique, messages are modified by predefined rules [4]. This technique alters request messages and sends the modified requests as tests to determine correct behavior. It requires access to existing message data and `WSDL` file information prior to pertubation. This technique helps fortify web services, assuring that certain faults cannot exist, but it is limited to known faults. Fault-based testing includes testing for robustness, where a white-box testing situation is assumed and error-recovery code coverage is emphasized. Faults are injected directly into source code, paying attention to the altered pieces of code. The error-recovery code coverage is measured when faulty source code is executed. Although this technique may help reduce crashes in a service, it is limited to white-box cases.

III. Web Service Emulation: iTrust

`iTrust` is a medical application that provides patients with a means to keep up with their medical history and records [6]. It handles very sensitive information that could physically affect the lives of its users which may include, but not limited to the following: allergies, overdose, and underdose. Therefore, the quality of this application needs to be ensured before releasing it to the public. Through the `SOAP`, `WSDL`, and `UDDI` specifications, we wrap iTrust, enabling it to emulate a web service.

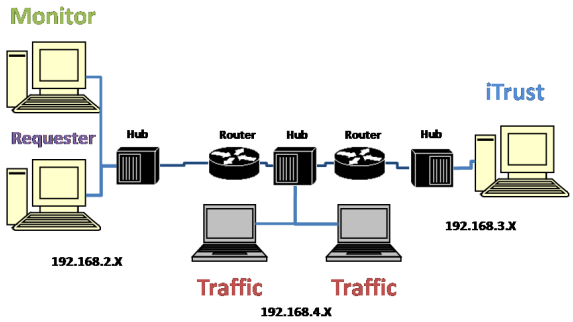

Fig. 3. Testing Framework

Our testing framework (Figure 3) is a network, where we emulate the Internet. In the center hub, the `Traffic` machines generate random network traffic, mimicking the Internet. The `Requester` is the client that will be accessing `iTrust`. The `Monitor` observes and collects information on the traffic coming to and from the `Requester`. The `iTrust` machine is the location where `iTrust` is deployed as a web service.

IV. Fault-Based Combinatorial Testing

Our approach combines both fault-based and combinatorial testing techniques, focusing on known fault types specific to web services, not the applications that they integrate. The fault types in this experiment include `Version Mismatch`, `Sender`, and `Receiver` faults [1]. They are as follows:

- **VersionMismatch** - The faulting node found an invalid element information item instead of the expected Envelop element information item. The namespace, local name or both did not match the Envelope element information item required by [the WC307] recommendation [1].
- **Receiver** - The message could not be processed for reasons attributable to the processing of the message rather than to the contents of the message itself. For example, processing could include communicating with an upstream SOAP node, which did not respond. The message could succeed if resent at a later point in time [1].
- **Sender** - The message was incorrectly formed or did not contain the appropriate information in order to succeed. For example, the message could lack the proper authentication or payment information. It is generally an indication that the message is not to be resent without change [1].

We first inject a fault into the web service. We accomplish this by satisfying one of the conditions required for a specific fault type. For instance, for a `Version Mismatch` fault, either the `NameSpace` or `LocalName` is invalid. Thus we would purposefully alter these values, creating a faulty web service. Test cases are generated by a combinatorial algorithm, where `t-way` interactions are stressed. A value of $t = 2$, a *2-way* interaction, implies that every combination of 2 values will

be covered. For example, in a `Version Mismatch` fault, an invalid `Namespace` and `LocalName` make up one *2-way* interaction, whereas a valid `Namespace` and an invalid `LocalName` make up another *2-way* interaction. Thus most types of faults can be recreated by different comibinations of invalid conditions. In each test case, all *2-way* interactions are covered, although there may be constraints. For instance, for a `Version Mismatch` fault to exist, a message has to be sent or received. Thus `Sender` and `Receiver` faults cannot exist. Outside constraints, all *t-way* interactions are covered, and depending on the combination of invalid values generated, multiple faults may be introduced to the web service.

After faults are introduced to a web service, test cases are invoked on it. The test cases involve simply using the applications integrated in the web service. For instance, `iTrust` requires a user to login in order to access their health information. An example test case could be inputting invalid login information, whereas another case could involve inputting valid information. A testing oracle, containing expected outputs for specific inputs, is utilized. After a fault is introduced to the web service, a test case from the oracle is invoked on it. The output produced is compared to the expected value in the oracle. If the responses differ, the fault is exposed. If the web service detected an introduced fault successfully, it means that fault cannot exist in the web service.

V. Anticipated Contributions

Web services can grow and be very complex throughout their lifetime, making it difficult to assure their quality. Quality may be very important, depending on the web service. For instance, `iTrust`, a hospital management system, handles very sensitive information that could affect the lives of its users. Quality is must. Thus web service testing is required. Combinatorial testing techniques have proven to be efficient in testing software, but as far as web services, they may be in their infancy. By combining both fault-based and combinatorial testing techniques, and by introducing diverse faults into a web service and monitoring for correct behavior, we are more confident about the quality of the web service. With proper testing, we believe this fault-based combinatorial testing of web services may be able to better assess and evaluate web services.

References

[1] M. Gudgin, M. Hadley, N. Mendelsohn, J. Moreau, H. Nielsen, A. Karmarkar, and Y. Lafon. Soap version 1.2. http://www.w3.org/TR/soap12-part1/, 2007.

[2] D. Kuhn, D. Wallace, and A. G. Jr. Software fault interactions and implications for software testing. *IEEE Transactions on Software Engineering*, 30:418–421, 2004.

[3] C. Mao. Performing combinatorial testing on web service-based software. In *IEEE International Conference on Computer Science and Software Engineering*, pages 755–758, Nanchang, China, 2008. IEEE Computer Society.

[4] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *SIGSOFT Softw. Eng. Notes*, 29(5):1–10, 2004.

[5] V. Pretre, F. Bouquet, and C. Lang. Automating uml models merge for web services testing. In *iiWAS '08: Proceedings of the 10th International Conference on Information Integration and Web-based Applications & Services*, pages 55–62, New York, NY, USA, 2008. ACM.

[6] N. C. S. U. RealSearch Research Group. itrust: Role-based healthcare v7.0.1n. http://agile.csc.ncsu.edu/iTrust/wiki/doku.php, 2008.

[7] O. U. specification TC. Uddi version 3.0.2. http://uddi.org/pubs/uddi_v3.htm, 2005.

[8] W. Vogels. Web services are not distributed objects. *Web services are not distributed objects*, 7:59–66, 2003.