# Fault Injection for Web-services

Marek Rychlý and Martin Žouželka
*Department of Information Systems, Faculty of Information Technology, Brno University of Technology, IT4Innovations Centre of Excellence, Božetěchova 2, 612 66 Brno, Czech Republic*

Keywords: Software Testing, Fault Injection, Fault Model, Service-oriented Architecture, Web-service, Tool.

Abstract: Recently, web-services have become a popular technology for implementing information systems as service-oriented applications. The service orientation allows to decompose complex software systems into collections of cooperating and autonomous components which communicate asynchronously via messages of appropriate formats. Nevertheless, the assurance of reliability and robustness of these systems is becoming more and more complicated matter. For this reason, the new testing methods such as fault injection and specialised tools for automated validation of web-services appear. This paper discusses the design, implementation, and evaluation of a tool for software implemented fault injection into web-services. This tool allows to monitor and to test the most common types of web-services according to given setup criteria.



## 1 INTRODUCTION

Currently, information systems are often designed as component-based systems with service-oriented architecture and web-service technology. The service orientation allows to decompose a complex software system into a collection of cooperating and autonomous components known as services. These services communicate asynchronously via messages of appropriate formats and through various protocols. They cooperate with each other to carry out a business process representing a particular functionality of the implemented software system.

Design of an information system as service-oriented architecture (SOA), i.e. its decomposition into a collection of cooperating services, increase reusability of its components and usually reduces its development time and effort. The system designed as SOA can adopt existing services and integrate them together with new services implementing missing functionality. The services can be adopted from other information systems developed in-house including their full development documentation and their source code, or they can be purchased as commercial third-party services available as "black-boxes" only, i.e. with limited or without any insight into their extra-functional or safety-critical properties. In the second case, it is necessary to perform thorough reliability testing, i.e. unit tests of the adopted services as well as integration tests of their compositions.

Although the intensive reliability testing may increase trustworthiness of the services and decrease probability of undetected failures, it is often impossible to test the services in all configurations and all possible integrations and guarantee robustness (fault tolerance) of the resulting system (Crnkovic, 2002, pp. 197–198). For this reason, new testing methods such as fault injection and specialised tools for automated validation of the services appear. The fault injection, contrary to the previously mentioned reliability testing, tries to reveal how robust (tolerant) a software system is to potential failures caused by the unreliable services.

This paper discusses the design, implementation, and evaluation of a tool for software implemented fault injection into web-services. The tool allows to monitor and to test robustness of the most common types of web-services compositions according to given setup criteria, i.e. to reveal the consequences of failures of the web-services to the rest of a system.

The rest of this paper is organised as the follows. Section 2 describes software implemented fault injection (SWIFI) techniques, their applications in fault injection into web-services, and existing SWIFI tools and their usability for robustness testing of web-services of different technologies. In Section 3, we introduce the architecture and the implementation of our SWIFI tool for SOAP and RESTful web-services. Its evaluation is described in Section 4. Finally, Section 5 sums up the results, provides a conclusion, and outlines a future work.

Rychlý M. and Žouželka M..
Fault Injection for Web-services.
DOI: 10.5220/0004153003770383
In *Proceedings of the 14th International Conference on Enterprise Information Systems* (NTMIST-2012), pages 377-383
ISBN: 978-989-8565-11-2

# 2 SOFTWARE IMPLEMENTED FAULT INJECTION INTO WEB-SERVICES

The goal of *software implemented fault injection* (SWIFI) is to ensure robustness testing, as it has been mentioned in the introduction of this paper. According to (Crnkovic, 2002, p. 198), the fault injection into a software component means to artificially corrupt a function of this component, input data consumed by the component, or output data produced by the component and sent to its successors, to observe how the component, the successors, or the whole system behaves. To perform this tasks, a fault injection tool has to inject one or more faults into a software component under test and to help to analyse the consequences of potential failures. By simulating the faults in various software components, we determine whether or not their failures can be tolerated.

According to (Hsueh et al., 1997), SWIFI techniques can be classified into two types, compile-time injection and run-time injection. In the first technique, a source code is modified to inject simulated faults into a software component in its compile-time. In the second technique, the software component's functionality or data are modified in its run-time.

The *compile-time injection* requires availability of the source code of a software component provided by its developers or obtained by its disassembling. This technique does not require any modifications of the component's run-time environment, and therefore, it is applicable especially in the case of in-house developed software components deployed into different uncontrollable environments. The compile-time injection into web-services with available source codes can be done by common SWIFI tools without any specialisation to web-service technologies.

During the *run-time injection*, a software component is executed as a "black-box" in a special run-time environment without need of any modifications or recompilation of the component's source code. The run-time environment enables a tester to modify the component's control-flow and internal memory (invasive testing) as well as input and output data transmitted via the component's interfaces (non-invasive testing). In the case of the run-time injection into a web-service, a run-time modification of the service's control-flow or its internal memory does not require any knowledge of a web-service technology contrary to a modification of data transmitted via the service's interfaces, which are technology dependent. For the first type of modifications, common SWIFI tools can be used while the second type of modifications require SWIFI tools specialised for web-services.

The SWIFI *tools for the run-time injection into web-services* usually act as HTTP proxies encapsulating web-services under test, or they can be integrated into web-servers providing run-time environments for the web-services (i.e. service containers).

## 2.1 SWIFI Tools for Web-services

This section describes the state of the art for the SWIFI tools specialised for (non-invasive) run-time fault injection into web-services. In addition to the specialised tools, also SWIFI tools providing network-level fault injection, such as DOCTOR (Han et al., 1993) or ORCHESTRA (Dawson et al., 1996), can be used for the fault injection into web-services to corrupt the services' messages without understanding their content, i.e. a particular web-service technology.

*WS-FIT* (Looker et al., 2007) is a tool implementing targeted perturbation of web-services' RPC parameters as well as generic network-level fault injection, which can be applied to a range of other RPC-based middlewares, such as DCE, DCOM, and CORBA. To enable the run-time fault injection into web-services, WS-FIT requires to insert a hook code into a SOAP stack of web-servers providing run-time environments for the web-services to capture SOAP messages. A reference implementation of WS-FIT includes its integration into service containers of Apache Jakarta Tomcat web-servers versions 4 and 5. The implementation of WS-FIT tool is not publicly available.

*wsrbench* (Laranjeiro et al., 2008) is a publicly available on-line tool for robustness testing of SOAP-based web-services represented by their WSDL descriptions. The tool acts as a web-service client which automatically generates SOAP requests according to a given test-case specification (e.g. to call a web-service with parameters outside a given domain, etc.). This approach allows to test only single services, but not service compositions. Faults also cannot be injected outside parameters of a web-service's operations, e.g. to test a corruption of message headers.

*WSInject* (Valenti et al., 2010) is a fault injection tool for testing single and composed web-services. It allows to inject both communication and interface faults, i.e. to simulate a network-level malfunction as well as wrong service calls which do not match their descriptions. The tool acts as a HTTP proxy injecting faults into passing SOAP messages while transmitting non-SOAP HTTP messages without modification. WSInject is able to inject faults into service compositions, but it is not applicable to web-services using non-SOAP technologies (e.g. RESTful web-services).

All the mentioned tools are capable of run-time injection into SOAP-based web-services. However, they are not useful for testing of RESTful web-services which are recently becoming popular on the Internet.

# 3 FAULT INJECTION TOOL FOR WEB-SERVICES

As it has been mentioned in the previous sections, the demand for SWIFI tools which enable a tester to create test specifications for fault injection and to apply them automatically at the specified parts of web-services is still growing.

This section describes the development of the *FIWS tool* (*Fault Injector for Web Services*), which can be classified as a tool for software implemented run-time fault injection and non-invasive robustness testing of SOAP-based, XML-RPC, and RESTful web-services and their compositions. It allows to inject faults in run-time of tested web-services, into communication between the web-services and their partners. After the injection, the resulting abnormal states and erroneous behaviour of the web-services can be observed.

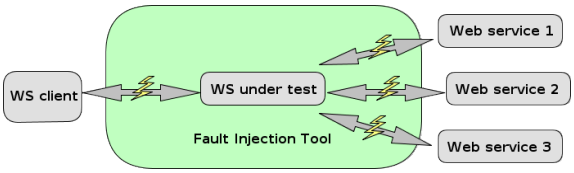


Figure 1: The communication scheme.

Figure 1 describes an application style of the tool in the communication scheme between a tested service (i.e. “WS under test”) and its surroundings. All communication of the tested service with its clients and another web-services required by the tested service (i.e. “WS client” and “Web service 1–3”, respectively) is routed via a proxy server where the FIWS tool monitors incoming and outgoing messages. When the communication meets particular criteria given by a test specification, the corresponding messages are corrupted with faults of defined types.

The FIWS tool does not require a specific type of the message routing method – the routing can be done by an arbitrary network setting (e.g. network routing or firewall rules), by a transparent HTTP proxy, by a modification of a web-server providing web-services’ run-time environment (i.e. service containers), etc.

The tool’s internal architecture is described in Figure 2. The three-layered architecture divides program

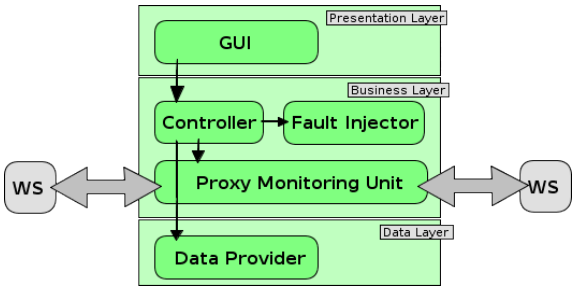


Figure 2: The three-layered architecture model.

components into three groups of different responsibilities for different tasks. The top-level layer is the presentation layer that contains modules and classes responsible for a (graphical) user interface of the tool (it consists of classes representing windows and dialogs). The next one is the business layer implementing the application logic, which is composed of three units:

- *Proxy Monitoring Unit* – Its goal is to act as a HTTP proxy, which listens at a given port for incoming connections and mediates the communication between services. Behaviour of this component is a bit different from common proxy servers. It waits until a whole HTTP message is read, and then, it analyses important parts of the message and forwards them to the Fault Injector.
- *Fault Injector* – It receives specific parts of service-call messages from the Proxy Monitoring Unit and processes them by classes representing different types of conditions and faults. The processing is defined by a particular test specification in the tool’s user interface and received by the Controller.
- *Controller* – It is a unit controlling the business layer according to events from the tool’s user interface. It also implements the persistence of tests and results which are stored in a XML database.

Finally, the bottom-level layer is the data layer, which consists of classes accessing the mentioned *XML* database with test specifications and results.

## 3.1 Conditions and Faults

The fault injection mechanism in the FIWS tool is based on a model of conditions and faults and with respect to the general fault model by (Looker et al., 2005). In the tool, the fault injection mechanism is controlled by user-defined test specifications. Every test specification contains a list of named rules and every rule consists of a set of conditions and a set of faults. When a service message meets all conditions

of a particular rule, each of the corresponding faults is applied.

A rule in a test specification can include a combination of the following conditions:

- *Content linked Condition* – fulfilled iff a particular message contains a specified string;
- *URI linked Condition* – fulfilled iff a particular message contains a specified string in URI address;
- *Destination linked Condition* – fulfilled iff a particular message is a HTTP request (i.e. a service call) or a HTTP response (i.e. a service response).

Moreover, the rule can include a set of the following faults, which will be applied to the communication between services or between a service and its clients. Some faults have been adopted from existing SWIFI tools (see Section 2.1) and the others have been designed for our needs solely.

- *String Corruption* is a simple fault which replaces all specified sub-strings in a message for another sub-string. Since this is done on the string level regardless the message's format, it can be used with various data interchange formats (e.g. XML, JSON, YAML, HDF, etc.).
- *XPath Corruption* is similar to the previous fault except that the replaced parts are specified by an XPath expression (the message has to be in XML). This kind of fault is useful when we want to modify a value of a particular element or an attribute representing parameters of a service's operation.
- *XPath Multiplication* also uses XPath expressions to find a particular part of a message in XML, however, the targeted part is not changed, but multiplied (which is a specific type of the corruption).
- *WSDL Operation Corruption* uses a WSDL service description to identify service operation calls or responses inside a SOAP message and to modify the message's body. By means of these faults, we are able to test robustness of service operations.
- *Header Corruption* allows to modify parameters of an URI inside a message's HTTP header. This is useful for testing of RESTful services containing data (or meta-data) also in the HTTP header.
- *Emptying Fault* empties the content of a HTTP message (just its header remains and the rest of the message is discarded). This can be used for testing a service's response to empty HTTP messages.

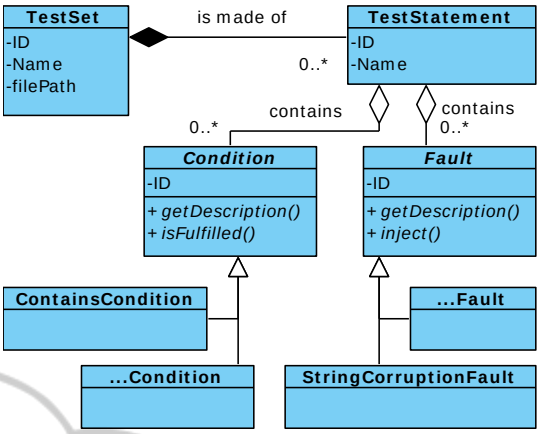


Figure 3: The class diagram of test specifications.

- *Delay Fault* delays a targeted message for a specified number of milliseconds. This fault allows to simulate some network anomalies caused by message loss or problems with network latency.

Figure 3 outlines the class diagram of test specifications[1]. A test specification is represented by an instance of class TestSet and consists of several statements represented by TestStatement. Each statement contains several test conditions and injected faults represented by classes Condition and Fault, respectively.

The pre-defined list of test conditions and injected faults can be extended by new specialisations of classes Condition and Fault, respectively. Each specialisation of class Condition has to implement method getDescription to obtain its informative description and method isFulfilled, which decides, applied to a HTTP message, whether the condition is meet or not. Each specialisation of class Fault has to implement method getDescription analogously to the previous case and method inject performing desired modifications of the HTTP message (i.e. the fault injection).

## 4 EXPERIMENTS

We performed several experiments with robustness testing of various web-services by the FIWS tool. At first, we tested sample web-service projects included in the *Netbeans IDE* as demonstrations of JAX-WS and JAX-RS standard Java EE frameworks

[1]Please note that for space reasons the class diagram is not complete in the sense that it contains classes of all the mentioned conditions and faults and full definitions of the mentioned methods of the classes.

for SOAP-based and RESTful web-services, respectively. The web-services have been deployed into service containers of the *Oracle GlassFish* web-server in version 3.2 providing a run-time environment for the services.

Our goal has been to analyse fault tolerance of web-services in the JAX-WS and JAX-RS frameworks and also compatibility of the tool with the web-server. The results are in Sections 4.1 and 4.2. Finally, we also focused on several publicly available web-services with results described in Section 4.3.

## 4.1 SOAP-based Calculator (JAX-WS)

The first tested service is called *Calculator*. It is a common SOAP-based web-service implemented as a composition of web-services and providing a few methods to perform basic arithmetic operations. The composition has been slightly modified to allow testing interception and corruption of messages sent between services inside the composition (i.e. not only between the main web-service and its client).

The testing scenario consists of two SOAP-based web-services in the composition and one client. The client sends its requests to the first service, which serves as a mediator to access operations provided by the second service. This arrangement creates three HTTP interactions to monitor, which means six HTTP messages in one service call by the client.

In the first test set, we focused on a parameter corruption test, i.e. a fault injection into input parameters of the service's operations and into return values produced by the operations and consumed by another service in the service composition. Each test was performed two times to eliminate unexpected conditions.

In test statements, *WSDL Operation Corruption* and *XPath Corruption* types of faults were utilised (see Section 3.1). Based on these fault types, parameter mutation rules for numeric values were adopted from (Vieira et al., 2007) as it is described in Table 1. The table describes names of the tests and corresponding actions which are applied to the operation's parameters or to the return values by the mentioned types of faults.

During the testing, the *NumNull* corruption was interpreted by Calculator as a number zero in all performed cases, so the operation was completed without any errors or exceptions shown. The *NumMax* and *NumMin* tests resulted in data type overflow, while the service responded without any error messages and returned wrong values. For example, the additive operation with parameters 2147483647 and 4 returned $-2147483645$. On the other hand, during the corruption of the return values, the consuming service interpreted the overflowed value as a zero number.

Table 1: Numeric parameter curruption.

| Test name | Parameter corruption |
|---|---|
| NumNull | Replace by null val. |
| NumEmpty | Replace by empty val. |
| NumAbsoluteMinusOne | Replace by $-1$ |
| NumAbsoluteOne | Replace by 1 |
| NumAbsoluteZero | Replace by 0 |
| NumAddOne | Add one |
| NumSubtractOne | Subtract 1 |
| NumMax | Max. type valid |
| NumMin | Min. type valid |
| NumMaxPlusOne | NumMax + 1 |
| NumMinMinusOne | NumMin − 1 |

In the second test set, several tests were performed to verify robustness of interfaces of the tested services. The *XPath Multiplication* fault was injected into a SOAP message, however, it did not effect processing the message in a service, since SOAP parsers process only the first occurrence of an element within the message. The injection of *Emptying Fault* resulted into a SOAP error message response informing that an XML reader exception was thrown. Finally, we tried to inject *Delay Faults* with delays for 10, 20, and 60 seconds. The first two cases run correctly, however, the one-minute delay resulted in the unavailable consuming service (it did not close its connection).

## 4.2 RESTful CustomerDB (JAX-RS)

The second tested service is a RESTful web-service which is called *CustomerDB*. This service provides standard RESTful HTTP methods GET, PUT, POST, and DELETE, to query and update resources representing data in a *Java Derby* database.

In this test set, we created test statements using the *Header Corruption* fault to be able to modify the specific parts HTTP URIs containing service request parameters, i.e. describing a database query going towards the web-service. The parameters were corrupted as values of the string data-type by actions shown in Table 2. Each test was performed two times to eliminate unexpected conditions.

Table 2: String parameter corruption.

| Test name | Parameter corruption |
|---|---|
| StrNull | Replace by null val. |
| StrEmpty | Replace by empty str. |
| StrNonPrintable | Replace by nonprintable |
| StrAddNonPrintable | Add one nonprintable |
| StrAlphaNumeric | Apply alphanumeric string |
| StrOverflow | Overflow max size |

After the fault injection, some of the corrupted service calls resulted in inconsistencies in the queried database while no errors were indicated. The web-

service came into the "silent" failure mode according to the *wsCrash* scale introduced by (Vieira et al., 2007).

Contrary to the SOAP parsers in the case of SOAP-based web-services (see Section 4.1), the RESTful approach does not provide any implicit validation of formats of incoming messages. The messages of RESTful web-services can be in various formats (e.g. XML, JSON, YAML, HDF, etc.) and the fault tolerance of such web-services has to be ensured by their developers, e.g. by usage of a particular validator corresponding to a specific message format. Obviously, this was not the case of the *CustomerDB* web-service.

### 4.3 Publicly Available Web-services

Finally, we performed robustness testing of the following publicly available web-services:

- *Affiliate Web Service* by alpharooms.com[2]
- *WSCore Web Service* by altaircom.net[3]
- *ForeignExchangeService* by as.asp2.cz[4]
- *wvWAVKY* by cuni.cz[5]

These services have been tested as "black-boxes" without any knowledge of their internal implementations. Therefore, we were not able to inject faults into communication between services of potential service compositions (we cannot redirect the communication between the externally provided services via our proxy). We also did not have access to error logs of the web-services' containers, so could not detect internal errors of the services which were not externally visible.

We performed random fault injection into calls of the services by their clients and observed the services' external behaviour, specifically their responses and availability. The fault injection caused few failures resulting into unpredictable behaviour. The *Emptying Fault* corruption mostly ended with the "HTTP 400 Bad Request" notification, however, in some cases, the response was "HTTP 500 Internal Server Error" indicating that a failure of a web-service occurred. Moreover, the fault injection into XML element AffiliateAuthentication in a call of *Affiliate Web Service* resulted into the unexpected behaviour where non-printable characters in the element's content did not caused the "permission denied" message.

[2]http://xml.alpharooms.com/affiliate.asmx
[3]http://palehorse.altaircom.net/WS/WSCore.asmx
[4]http://as.asp2.cz/ForeignExchangeService.asmx
[5]http://is.cuni.cz/webapps/ws.php

## 5 SUMMARY AND CONCLUSIONS

In this paper, we introduced the FIWS tool for software implemented fault injection (SWIFI) into web-services. Contrary to the previously existing SWIFI tools, the FIWS tool is able to inject faults into service compositions and to perform robustness testing of common SOAP-based web-services as well as contemporary RESTful web-services. To evaluate applicability of the tool, several web-services have been tested using both web-service technologies. We performed fault injections into a SOAP-based web-service implemented in Java JAX-WS, a RESTful web-service implemented in Java JAX-RS, and several publicly available web-services.

The further research will focus on integration of the tool and the fault injection in general into integrated development environments (IDEs) to enable developers to define and automatically perform tests of web-services' robustness in desired service compositions.



## ACKNOWLEDGEMENTS

This work was supported by the research programme MSM 0021630528 "Security-Oriented Research in Information Technology", the BUT FIT grant FIT-S-11-2, and by the European Regional Development Fund in the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070).

## REFERENCES

Crnkovic, I. (2002). *Building Reliable Component-Based Software Systems*. Artech House, Inc., Norwood, MA, USA.

Dawson, S., Jahanian, F., and Mitton, T. (1996). ORCHESTRA: A fault injection environment for distributed systems. Technical Report CSE-TR-318-96, University of Michigan, Michigan.

Han, S., Rosenberg, H. A., and Shin, K. G. (1993). DOCTOR: An integrated software fault injection environment. Technical Report CSE-TR-192-93, University of Michigan, Michigan.

Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *Computer*, 30:75–82.

Laranjeiro, N., Canelas, S., and Vieira, M. (2008). wsrbench: An on-line tool for robustness benchmarking. In *Proceedings of the 2008 IEEE International Conference on Services Computing - Volume 2*, SCC '08, pages 187–194, Washington, DC, USA. IEEE Computer Society.

Looker, N., Burd, L., Drummond, S., Xu, J., and Munro, M. (2005). Pedagogic data as a basis for web service fault models. In *IEEE International Workshop on Service-Oriented System Engineering*, pages 125–136, Los Alamitos, CA, USA. IEEE Computer Society.

Looker, N., Xu, J., and Munro, M. (2007). Determining the dependability of service-oriented architectures. *International Journal of Simulation and Process Modelling*, 3:88–97.

Valenti, A. W., Maja, W. Y., Martins, E., Bessayah, F., and Cavalli, A. (2010). WSInject: A fault injection tool for web services. Technical Report IC-10-22, Institute of Computing, University of Campinas, Campinas.

Vieira, M., Laranjeiro, N., and Madeira, H. (2007). Benchmarking the robustness of web services. In *13th Pacific Rim International Symposium on Dependable Computing*, pages 322–329.