Abstract

Testing web services for robustness is a difficult task. In fact, existing development support tools do not provide any practical mean to assess web services robustness in the presence of erroneous inputs. Previous works proposed that web services robustness testing should be based on a set of robustness tests (i.e., invalid web services call parameters) that are applied in order to discover both programming and design errors. Web services can be classified based on the failure modes observed. In this paper we present and discuss the architecture and use of an on-line tool that provides an easy interface for web services robustness testing. This tool is publicly available and can be used by both web services providers (to assess the robustness of their web services code) and consumers (to select the services that best fit their requirements). The tool is demonstrated by testing several web services available in the Internet.

1. Introduction

Web Services are a key technology in Service Oriented Architecture (SOA) environments, which are increasingly being used in business critical applications. A web service is a software component that exposes a given functionality that can be assessed by service consumers in an interoperable manner. In fact, from the client standpoint there is no need to know the details of the implementation of the service (including the underlying technology). Web service clients only need to know the format of the message to be exchanged with the provider. This makes web services the lingua franca for systems integration and communication.

Web services are supported by several key standards, including:

- **WSDL** (Web service Description Language): describes network services as a set of endpoints that operate on messages, in a cross-platform way [5].
- **SOAP** (Simple Object Access Protocol): a XML based message protocol that, in simple terms, describes message content and how that content is processed [11].
- **UDDI** (Universal Description Discovery and Integration): defines a standard for dynamic discovery of web services [19].

Software faults (i.e., program defects or bugs) are recognized as the major cause of computer system failures [7], [9], [10]. Interface faults are particularly relevant in web services as providers have to provide a robust interface to client applications, even in the presence of invalid inputs (e.g., invalid SOAP messages). These invalid inputs may occur due to bugs in the client applications, corruptions caused by silent communication failures, or even security attacks.

Robustness testing allows the characterization of the behavior of a system or component in presence of erroneous input conditions [1], [16], [12]. Web services development environments do not provide any support for web services robustness testing. This way, as web services are usually complex software components developed against typically aggressive schedule constraints, they are frequently deployed before being properly tested and carrying residual software (interface) defects. This may expose real applications to severe problems, including critical vulnerabilities that can be maliciously exploited with serious consequences, such as denial-of-service (DoS), or data loss.

In [13] it is proposed an approach for the evaluation of the robustness of web services. The approach consists of a set of robustness tests (i.e., invalid web services call parameters) that is applied during execution in order to disclose both programming and design problems. The robustness of the web services is characterized according to the following failure modes: Catastrophic, Restart, Abort, Silent, and Hindering (adapted from the CRASH scale [16]).

This paper presents wsrbench, an online tool that can be used to perform robustness tests on web services. This tool, publicly available at http://wsrbench.dei.uc.pt, is based on the work presented in [13] and, in our opinion, is a powerful instrument that can easily be used in three key scenarios:

1) Help vendors in evaluating and improving the robustness of their web services implementations before deployment.
2) Help client application developers to select the web services that best fit their requirements by testing different alternatives.
3) Help vendors and client application developers to identify the need for wrappers to perform the required validations before execution.

This paper demonstrates the usefulness of wsrbench by testing 100 publicly available web services provided by different relevant parties (including Microsoft). Several problems were exposed in the tested services, which illustrate the utility of the testing tool.

The structure of the paper is as follows. The next section presents background on web services robustness testing. Section 3 briefly describes the application. Section 4 presents some experimental results and their discussion. Finally, Section 6 concludes the paper.

2. Web Services Robustness Testing

As mentioned before, the present work is based in the web services robustness testing approach previously proposed in [13]. For the sake of completeness, in this section we briefly describe the approach, for more details see [13].

Web services robustness testing is based on erroneous call parameters. The robustness tests consist of exceptional and acceptable input values of parameters of web services operations that can be generated by applying a set of predefined rules according to the data types of each parameter.

The robustness benchmark includes the following key components: workload (represents the work that the service must perform during the benchmark run); robustness tests (set of invalid call parameters that is applied to expose robustness problems); and failure modes classification (characterize the behavior of the web service while executing the workload in the presence of the robustness tests). The testing procedure is based in the following generic set of steps:

1. Tests preparation
   1.1 Analysis of the WSDL of the web service under testing in order to gather information about the relevant operations, call parameters and data types.
   1.2 Workload generation.

2. Tests execution
   2.1 Execution of the workload generated in step 1.2. The goal is to understand the expected correct behavior for the service.
   2.2 Execution of the robustness tests in order to trigger faulty behaviors, and in that way disclose robustness problems.

3. Web service characterization, including failure modes identification (using the data collected in step 2).

Before generating and executing the robustness tests we need to obtain some definitions about the web service operations, parameters, data types and domains. The web service interface is described as a WSDL file that can be processed to obtain the list of operations, parameters (including return values) and associated data types. However, the valid values for each parameter (i.e., the domain of the parameter) cannot be deduced from the WSDL description. Thus, information on the valid domains for each parameter must be provided by the tester.

A workload (set of valid web service calls) is needed to exercise each operation of the web service under testing. As it is not possible to propose a generic workload that fits all web services, we need to generate a specific workload for the web service under testing. Two options can be considered for the generation of the workload: user defined workload and random workload.

The execution of the benchmark includes two steps. In the first step, the workload is run without considering invalid call parameters. The goal is to understand the typical behavior expected from the web service. In the second step of the benchmark, the workload is run in the presence of invalid call parameters (robustness tests). This step includes several tests, where each test focuses a given operation of the web service and includes a set of slots. Each slot targets a specific parameter of the operation, and comprises several injection periods. In each injection period several faults (from a single type) are applied to the parameter under testing.

The set of robustness tests performed is automatically generated by applying a set of predefined rules (see detailed list in [13]) to the parameters of each operation of the web service during the workload execution. An important aspect is that rules focus difficult input validation aspects, such as: null and empty values, valid values with special characteristics, invalid values with special characteristics, maximum and minimum valid values in the domain, values exceeding the maximum and minimum valid values in the domain, and values that cause data type overflow.

The robustness of the web services is classified according to an adapted version of the CRASH scale [16] (the wsCRASH scale) that distinguishes the following failure modes: Catastrophic (the application server used to run the web service under testing becomes corrupted or the machine crashes or reboots), Restart (the web service execution hangs and must be terminated by force), Abort (abnormal termination of the web service execution), Silent (no error is indicated by the application server), and Hindering (the error code returned is not correct or the response is delayed).

The use of the wsCRASH scale is not possible when tests are performed from the consumer point of view. In fact, as the tests are run remotely it is not possible to distinguish between a catastrophic and a restart failure mode as the tool does not have access to the application server where the web service is running. To face this problem, a simplified classification scale is proposed (the wsAS scale). This scale is based on two easily observable failure modes: Abort (an unexpected exception is raised by the
application server and sent to the web service) and Silent (no response from the server). Although it is a simplified characterization approach it still is quite useful.

3. The wsrbench Application

This section briefly introduces the main features provided by wsrbench. This tool, publicly available at http://wsrbench.dei.uc.pt, provides a web based interface that allows users to perform configurations and visualize the results of tests. Note that, anyone can use wsrbench as it is free and very easy to use. Only a very simple registration and posterior authentication is required.

After registration and authentication three key options are available for users: Configuration; Add WSDL; and My Tests. The Configuration option allows several configuration aspects to be defined, such as the user’s email, number of finished tests to show on screen, etc.

The Add WSDL option allows users to add the WSDL file describing a web service to be tested for robustness. After submitting the WSDL file the user can visualize the set of operations and parameters provided by the service. This is represented in Figure 1.

After defining the domain of the parameters the tests can start (by clicking the Start Test button). The user will be informed by email when the tests conclude and the results can be analyzed later on.

The My Tests option allows the user to visualize the tests previously performed along with information on currently ongoing tests. Detailed results for the concluded tests are available and are presented as shown in Figure 2.

![Figure 1 - Web service parameters domain definition.](image1)

Operations not conformant with the WS-I Basic Profile [3] will be grayed out and not tested. This standard is an industry effort to clarify the ambiguous parts of the WSDL specification and is accepted by the main service providers, including Microsoft’s .NET framework version 3 [14] and Sun Microsystems’s Java 6 Web Services stack (JAX-WS) [18]. By standardizing the specification, service interoperability is assured and the development of service-based tools (such as wsrbench) is simplified. Nevertheless, external components that perform WSDL analysis are currently being tested so that we can provide increased compatibility with older versions of WSDL documents, or documents that are not compliant with the WS-I Basic Profile standard.

As shown in Figure 1, for each testable operation the user may define the valid values for each parameter (i.e., the domain of the parameter). When these are not defined, the tool considers that the parameter domain is the domain of the corresponding data type. After defining the

![Figure 2 - Detailed test results for a web service.](image2)

As we can see, for each operation of a web service, the results for the individual faults applied to each parameter are shown (Figure 2 shows only a partial set). The first row for each parameter (identified by the word none) represents a regular request where no robustness testing is performed (i.e., no faults are injected). Clicking the ‘XML’ link opens a popup where more details are provided (see Figure 3). These include the list of requests sent and the service responses received.

The user can then use the three available buttons to mark the service interaction as a robustness problem, a correct interaction (no problem detected), or simply leave it unmarked. This is then reflected on the previous window (shown in Figure 2) by respectively placing a red, green or gray square in the right side. For instance, in the example shown in Figure 2, the StringNull fault applied to the name parameter of the registerUser operation disclosed a robustness problem. On the other hand, the StringNonPrintable fault did not disclose any problem.
It is important to emphasize that, after testing a given web service, the tool performs an automatic analysis of the responses obtained in order to distinguish regular replies from replies that reveal robustness problems in the service being tested. However, in some cases the tool is not able to decide if a given response is due to a robustness problem or not. That is why the tool also allows users to perform this analysis manually.

4. wsrbench Architecture

The wsrbench application is based in a multi-tier model. In practice, wsrbench is distributed over three tiers: client tier, application server tier, and data persistence tier. Tests execution also involves a fourth tier that corresponds to the external web service being tested, however this is not considered as part of the application itself. Figure 4 represents this scenario, including a high level view of the application modules.

As we can see, wsrbench is composed of three modules that correspond to a fully layered architecture:

- **Common Module**: includes several domain objects in POJO (Plain Old Java Object) form. It also includes abstraction for functionalities implemented by other modules. The goal was to allow development to take place as independently as possible.
- **Core Module**: provides basic core functionality, which includes SOAP request creation, fault generation, response analysis, etc.
- **Web Module**: includes advanced functionalities such as a complete persistence manager to enable database access, thread creation control (since a web application is virtually accessed by multiple users), and presentation layer code related to the web environment.

Solid arrows in Figure 4 represent visibility relations. The core module is aware of the common module and the web module is aware of the other two. In practice, this means that the common is a dependency of the core, and the common and core are dependencies of the web module.

It is important to emphasize that the wsrbench project uses the latest standards and technologies for Java development, namely:

- The Spring framework [17] for dependency injection, database access and connection management.
- Spring MVC for model-view-controller web layer implementation, including HTTP request validation and domain objects binding.
- Acegi Security [8] for fine grained control over the security requirements of the application. It also enables a complete isolation of all security related aspects from the application logic itself.
- Maven 2 [2] for project management tasks. These include the build process, report and documentation creation, and dependency management.

Figure 5 represents the internal architectural design of the application. The main components and existing dependencies are described in the following sections.

4.1. Web Module Design

The web module uses the MVC design pattern implementation offered by Spring [6] as a basis to provide multi-user support in a web environment. In this sense, the Controller component (see Figure 5) serves as an entry point for HTTP requests. This entry point is represented by the flow arrow between the web client and this component. Arrows in Figure 5 represent the main application flows that exist during execution. There are other relations and dependencies that are not included for sake of simplicity. For instance, bidirectional flows exist between the Driver Request Handler and all the other components presented at its right in the figure, but representing these would clutter the illustration unnecessarily.

The MVC element is divided into 3 key sub-elements: Binder, Validator and Controller. The Binder is the facil-

---

1 The source of wsrbench project is available at [11].
ity provided by Spring MVC that performs automatic binding to domain objects (note that more complex binding logic is defined by the programmer). The **Validator** is a set of classes that perform business logic validation over the HTTP request inputs. The **Controller** is responsible for request processing, building a domain model that is rendered by an adequate view, in our case a Java Server Page (JSP) view. This corresponds to the typical behavior of a controller in a MVC implementation.

The **Core Caller** which is the component responsible for the interaction with the Core Module. The **Executor Service** is used to maintain a fixed pool of threads that are used whenever an asynchronous call is required. In practice, there are two ways of internally serving user requests in wsrbench:

- **Synchronous**: used when, at the client interface, a user action requires immediate response from the Core Module (e.g., adding a WSDL file and proceed to the definition of the domain values of the available parameter: see Section 3 for details).
- **Asynchronous**: used when a user action does not depend on an immediate answer from the Core Module. For instance, the user can submit a new test and does not have to wait for an answer before defining another test on a different web service.

Two different objects representing the Core Module can be created by the **Core Caller**, one for synchronous invocation and another for asynchronous invocation. Both are obtained without direct knowledge of the Core Module implementation details by using a Factory class that resides in the Abstraction Layer and creates objects by reflection. The **Executor Service** is only used if the operation is asynchronous.

Note that for asynchronous operations, the Core Module needs a way to contact the Web Module when the operation concludes. A reference to the caller object (that implements a well known interface defined in the **Abstraction Layer**) is provided so that callbacks to the origin can occur in a simple way.

Transversely to the whole application there is a **Security** element which is basically a configuration of the Acegi Security framework [8]. This powerful framework provides declarative security for Spring-based applications and enables a complete separation of all security aspects into XML files. Although we could write security functionality directly into the application code, it is generally a better option to keep security concerns separate from application implementation specificities.

Another key component is the **Persistence Manager**. This is based on Spring’s Java Database Connectivity (JDBC) support that eliminates the need for programmatically handling database connections (which are a frequent cause for problems in web applications) and greatly simplifies information insertion, update and retrieval.

A **Configurator** object is created for each user session and contains user specific configuration parameters such as the number of tests to present on screen, the user’s email address, etc. Finally, the **SMTP Transport** component is used to send emails to users during registration and whenever a test is concluded.

### 4.2. Core Module Design

There are two ways of contacting the Core Module. One is via the **Synch Handler** and the other is through the **Asynch Handler**. These serve the synchronous and asynchronous operation modes mentioned before. Both make use of a **Driver Request Handler** that is responsi-
Testing the robustness of a given web service involves the \textbf{Wsdl Analyzer} that is responsible for analyzing a submitted WSDL (Web Service Description Language) document and returns operation and parameter information to the user. After this analysis step, the Driver Request Handler calls the \textbf{Request Generator} that generates a list of requests (i.e., the workload) for the specific service being tested. It considers the maximum and minimum domain values provided by the user to generate acceptable random requests for the service.

The \textbf{Fault Generator} receives the list of generated requests as input, creates a list of faults and injects those faults into the requests workload. For each parameter of a given type, all of the applicable faults are considered and injected into the request list. Several faulty requests are generated for a single type of fault for a given parameter (but only one fault per request), following the testing procedure described in Section 2.

The \textbf{HTTP Invoker} component is responsible for sending each web service request to the provider. This unit also collects the corresponding responses that are forwarded to the \textbf{Response Analyzer} component through the Driver Request Handler. Response Analyzer is in charge of response interpretation. Each response is analyzed in order to distinguish regular replies from replies that reveal robustness problems in the service being tested. Currently, this module is in a preliminary phase and improvements are being studied in order to improve the automatic response analysis. As mentioned before, the automatic responses classification can be complemented by the user with a manual analysis of the results.

\section*{5. wsrbenchmark Demo}

In this section we illustrate the robustness testing capabilities of wsrbenchmark by testing 100 web services (from distinct providers) publicly available on the internet. The majority is listed at [20].

The tested services are implemented in several technologies including .NET (83%), Visual Dataflex (6%), Axis (5%), MS Soap (2%), SOAP/AM (1%), Visual FoxPro (1%), 4D (1%) and Visual Basic (1%). The service list available at [20] has a high predominance of .NET web services which explains the high percentage of .NET services present in our tests.

The testing procedure consisted in using wsrbenchmark to generate all necessary web service requests and collecting and interpreting responses. Although most responses were automatically interpreted by the tool, service replies were also manually analyzed in order to confirm the robustness properties of each tested service.

Table 1 presents a summary of the results obtained during the experiments. Due to space constraints, the table only presents information about services that revealed robustness problems (parameters and operations that did not disclose any robustness problems are also omitted). Description includes the service name, publisher, operation and parameters and the total of distinct observed failures (grouped into major problem causes). The major causes for robustness problems observed were:

\begin{itemize}
  \item \textbf{Null references}: related to null pointers that reflect none or poor input validation in the service provider.
  \item \textbf{Database access operations}: includes exceptional behavior caused by invalid SQL commands.
  \item \textbf{Arithmetic operations}: typically data type overflow in numeric operations.
  \item \textbf{Conversion problems}: typically class cast exceptions or numeric conversion problems.
  \item \textbf{Other causes}: arguments out of range, invalid characters, etc. These were not very frequent and are thus classified in this generic group.
\end{itemize}

A total of 61 Abort failures (not counting similar errors triggered by different faults for the same parameter in a given operation) were detected by wsrbenchmark. 30% were marked as null references, 30% as SQL problems, 13% conversion problems, 7% as arithmetic problems, and 21% as others. No Silent failures (which correspond to an absence of error indication by the server) were observed.

About 35% of the web services tested presented robustness problems and all of these were developed in .NET. However, no assumption can be made in the sense that developing .NET services is a much more error prone task, since the original service list includes a larger number of .NET web services.

A huge percentage of web services revealed robustness problems related to null references. It was observed that most of the services assume that their input parameters are not null. However, this may not always be the case and it may lead to robustness problems that can have even more serious consequences, such as security issues.

The database related failures detected were mostly associated to setting one of the service input parameters to null. This reveals more than a simple SQL construction error. In fact, it shows that the provider does not validate the SQL inputs that open a door for SQL injection attacks that can compromise the security of the web service (or of the whole service infrastructure).
The typical cause for failures related to arithmetic problems was the use of the maximum value for a numeric type which resulted in several overflows in arithmetic operations.

Several other issues were observed, such as:

- Unexpected cast exceptions in the Microsoft service whenever an overflowed Boolean was used as input.
- The 'Recipes' web service were the access to a node of a Red-Black tree object is affected, fortunately for the

| Table 1. Public web services robustness tests results |
|-----------------|-----------------|-----------------|
| **Web service** | **Parameter** | **Abort Failures** |
| webservices.net  | getAirportInforma- | cityOrAirportName 1 |
| Code39 BarCode   | tionByCityOrAirportName(String cityOrAirportName) |  |
| Global Weather   | GetWeather(String CityName, String CountryName) | 1 |
| ICD9Drug         | GetICD9Drug(String Substance) | 1 |
| Microsoft        | ConvertLonLatPtToNearestPlace(double Lon, double Lat) | 2 |
| TerraService     | ConvertLonLatPtToUtmPt(double Lon, double Lat) | 2 |
| textdisguise.com | GetNewWord(String Guid, String Password) | 1 |
| dls27seconds     | GetAslWeds(int year) | 1 |
| vba66a           | ICD-9Codes(String Code, String CodeLength) | 1 |
| webservices.net  | IsValidEmail(String text) | 2 |
| seanco.com       | Search(String text) | 1 |
| esynaps          | GetSpeech(String Request) | 1 |
| jcono            | Search ASMX/WSDL files on the Web service | |
| vb906           | PhoneVerify(String PhoneNumber) | 1 |
| dls27seconds     | GetEasterMonday(int year) | 1 |
| walterjones      | GetHTML(String text) | 2 |
| icuisine         | SearchRecipes(int serviceID, String email String criteria, int pageNumbere) | 1 |
| vba66a           | GetICD9Codes(String Code, String CodeLength) | 1 |
| Synaptic         | Search(String KeyWord) | 1 |
| tilsiso          | IsValid(String UserInput, String VeriCodeHash) | 1 |
| ingig            | LetterApproved(String letter, boolean boys, boolean girls, boolean middle) | 1 |
| walterjones      | GetICD10Code(String text) | 1 |
| Synaptic         | regionToAbbrev(String regionName) | 1 |
| walterjones      | GetICD9ToICD10(String icd9code) | 1 |
| Scientio LLC     | RenameRuleset(String key, String common, String oldname, String common) | 2 |
| esynaps          | GetHostInfoByIP(String IPAddress) | 1 |
| ingig            | GetICD9Cod(String text) | 1 |
| walterjones      | GetICD9Level3(String text) | 1 |

The typical cause for failures related to arithmetic problems was the use of the maximum value for a numeric type which resulted in several overflows in arithmetic operations.

Several other issues were observed, such as:

- Unexpected cast exceptions in the Microsoft service whenever an overflowed Boolean was used as input.
- The 'Recipes' web service were the access to a node of a Red-Black tree object is affected, fortunately for the
provider the operation appears to carry no further consequences.

• Arguments out of range (clearly visible in the ‘Search for Icelandic Person’ service) and other exceptional behavior not properly handled by the service implementations under testing.

The observed problems reveal that the tested web services were deployed without being properly tested, and that our approach can expose the insufficiencies of the tests performed by providers. It is important to emphasize that, robustness problems can gain a higher importance if we consider that some can also represent security problems that can be used to exploit or attack the problematic web services (e.g., using SQL Injection).

6. Conclusion

This paper presented a tool for web services robustness testing. The presented work builds on solid scientific concepts proposed in previous works, and seals the space between research and industry practice. wsrbench fills a gap in current development tools, providing an easy interface for robustness testing of web services. The tool is available online requiring no installation and little configuration effort. Its effectiveness was demonstrated by testing 100 publicly available web services, in which we have disclosed several robustness problems. The results show that faulty services were deployed on the internet, which is unacceptable not only for their providers, but also for clients that are naturally expecting to use a robust service. These robustness issues can also represent security issues. For example, some of the problems uncovered show that some services may be vulnerable to SQL injection attacks, which highlights the importance of testing services for robustness. This is something that could be easily avoided if these services were tested for robustness by wsrbench.

In our opinion, wsrbench is quite useful for providers to evaluate and improve their solutions, and helpful for clients that now have an effortless way to compare (and select) different web services in terms robustness. In fact, both clients and providers can benefit from using our tool since it can be applied to identify missing, inconsistent, or poor parameter validation.

6. References