J Internet Serv Appl (2012) 3:215–232
DOI 10.1007/s13174-012-0062-2

ORIGINAL PAPER

A robustness testing approach for SOAP Web services

Nuno Laranjeiro · Marco Vieira · Henrique Madeira

Received: 23 March 2011 / Accepted: 28 May 2012 / Published online: 14 July 2012

Abstract The use of Web services in enterprise applications is quickly increasing. In a Web services environment, providers supply a set of services for consumers. However, although Web services are being used in business-critical environments, there are no practical means to test or compare their robustness to invalid and malicious inputs. In fact, client applications are typically developed with the assumption that the services being used are robust, which is not always the case. Robustness failures in such environments are particularly dangerous, as they may originate vulnerabilities that can be maliciously exploited, with severe consequences for the systems under attack. This paper addresses the problem of robustness testing in Web services environments. The proposed approach is based on a set of robustness tests (including both malicious and non-malicious invalid call parameters) that is used to discover programming and design errors. This approach, useful for both service providers and consumers, is demonstrated by two sets of experiments, showing, respectively, the use of Web services Robustness testing from the consumer and the provider points of view. The experiments comprise the robustness testing of 1,204 Web service operations publicly available in the Internet and of 29 home-implemented services, including two different implementations of the Web services specified by the standard TPC-App performance benchmark. Results show that many Web services are deployed with critical robustness problems and that robustness testing is an effective approach to improve services quality.

Keywords Testing · Reliability and robustness · Web services · Benchmarking

N. Laranjeiro · M. Vieira (✉) · H. Madeira
DEI/CISUC, University of Coimbra, 3030-290 Coimbra, Portugal
e-mail: mvieira@dei.uc.pt

N. Laranjeiro
e-mail: cnl@dei.uc.pt

H. Madeira
e-mail: henrique@dei.uc.pt

1 Introduction

Service-oriented architectures (SOA) are now widely used to support business infrastructures, linking suppliers and clients in sectors such as banking and financial services, transportation, automotive manufacturing, healthcare, just to name a few. Web services are a key element of SOA [1], and consist of self-describing components that can be used by other software across the Web in platform-independent manner, supported by standard protocols such as Web Services Description Language (WSDL) [2] and Universal Description, Discovery, and Integration (UDDI) [3].

Web services provide a simple interface between a provider and a consumer, where the former offers a set of services that are used by the latter. An important aspect is Web services composition, which is based on a collection of services working together to achieve a business objective [4]. This composition is normally a "business process" that describes the sequencing and coordination of calls to the component services. Thus, if one component fails then the composite Web service may suffer an outage.

Several solutions implementing the same service are frequently available for consumers as the same or different providers may offer alternative implementations. This way, it is a responsibility of the consumer to make the best choice by comparing the alternatives available and an obligation of the provider to deploy highly robust Web services. We believe

that a tool that helps providers and consumers assessing the robustness of Web services is of utmost importance.

Although Web services are increasingly being used in complex business-critical systems, assessing the robustness of services implementations (and comparing the robustness of alternative solutions) is still a difficult task. This paper proposes a practical approach to test the robustness of Web services code. Our approach is based on a set of robustness tests (including malicious inputs) that allows discovering programming and design problems. Robustness is characterized according to the following failure modes: correct (behavior is robust), crash (abnormal termination of the execution), and error (an unexpected error code is returned by the server). These failure modes are complemented with a detailed classification of the observed service behavior that may help understanding the source of the failures (i.e., defects).

The need for practical means to assess Web services robustness is corroborated by several studies that show a clear predominance of software faults (i.e., program defects or bugs) [5–7] as the root cause of computer failures and, given the huge complexity of today's software, the weight of software faults will tend to increase. Web services are no exception, as they are normally intricate software components that frequently implement a compound task, in some cases using compositions of several services, which makes them even more complex.

Interface faults, related to problems in the interaction among software components/modules [8], are particularly relevant in service-oriented environments. In fact, services must provide a robust interface to the client applications, even in the presence of invalid inputs that may occur due to bugs in the client applications, corruptions caused by silent network failures, or even security attacks.

Classical robustness testing, in which the approach proposed in this paper is inspired, is an effective approach to characterize the behavior of a system in presence of erroneous input conditions. It has been used mainly to assess robustness of operating systems (OS) and OS kernels [9–11], but the concept of robustness testing can be applied to any kind of interface. Robustness tests stimulate the system under testing through its interfaces submitting erroneous input conditions that may trigger internal errors or may reveal vulnerabilities.

The approach proposed in this paper consists of a set of robustness tests (i.e., invalid malicious and non-malicious service call parameters) that are applied during execution in order to observe robustness problems of the Web service itself, including security problems resulting from robustness failures. This is useful in three scenarios: (1) help providers to evaluate the robustness of their Web services code; (2) help consumers to pick the services that better fits their requirements; and (3) help system integrators to choose the best Web services for a given composition. In short, the main contributions of this work are:

- *Proposal of a robustness testing approach for Web services* This approach is based and largely extends a previous work [12]. In fact, the present paper consolidates the concepts and extends the approach by including robustness testing for security vulnerabilities and by proposing a new approach for classifying not only failure modes but also the observed service behavior. This is in fact a key contribution of the paper as it can be used as basis for the future definition of other robustness benchmarks in service-based applications, or service-oriented architectures.
- *Evaluation of 250 public Web services* (comprising 1,204 operations) publicly available in the Internet. Several robustness problems have been disclosed, including major security vulnerabilities and critical robustness failures, showing that many Web services are deployed without being properly tested for robustness.
- *Evaluation of 29 home-implemented Web services* (i.e., services developed in the context of this work, but by independent developers), including two different implementations of the Web services specified by the standard TPC-App performance benchmark. The robustness problems identified show that the proposed approach can be easily used in a development environment to improve Web services robustness.
- *An online tool* (**wsrbench**) *that implements the robustness testing approach proposed for Web services robustness testing* This tool fills a gap in current development tools, providing an easy interface for robustness testing of Web services.

The structure of the paper is as follows. The next section presents background and related work. Section 3 introduces the robustness testing approach and components. Section 4 presents the experimental evaluation and discusses the results. Section 5 concludes the paper.

2 Background and related work

Service-oriented architecture is an architectural style that steers all aspects of creating and using services throughout their lifecycle, as well as defining and providing the infrastructure that allows heterogeneous applications to exchange data. This communication usually involves the participation in business processes, which are loosely coupled to their underlying implementations. SOA represents a model in which functionality is decomposed into distinct units (services) that can be distributed over a network and can be combined together and reused to create business applications [13].

Web services provide a simple interface between a provider and a consumer and are a strategic mean for data exchange and content distribution [1]. In fact, ranging from

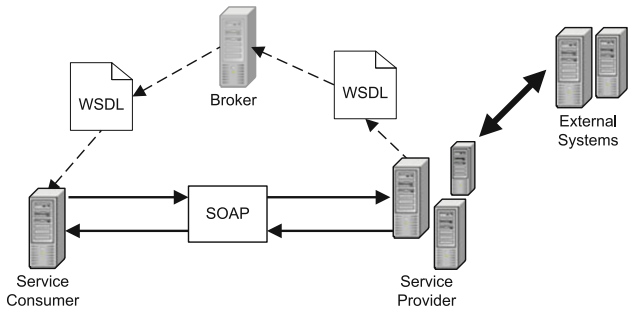

Fig. 1 Typical Web services infrastructure

on-line stores to media corporations, Web services are becoming a key component within the organizations information infrastructure and are a key technology in SOA domains.

In these environments the Simple Object Access Protocol (SOAP) [14] is used for exchanging XML-based messages between the consumer and the provider over the network (using, for example, HTTP or HTTPS protocols). In each interaction the consumer (client) sends a SOAP request message to the provider (the server). After processing the request, the server sends a response message to the client with the results. A Web service may include several operations (in practice, each operation is a method with one or several input parameters) and is described using Web-Services Definition Language (WSDL) [2], which is a XML format used to generate server and client code. A broker enables applications to find Web services. Figure 1 depicts a typical Web services infrastructure.

The goal of robustness testing is to characterize the behavior of a system in presence of erroneous input conditions. Although it is not directly related to benchmarking (as there is no standard procedure meant to compare different systems/components concerning robustness), authors usually refer to robustness testing as robustness benchmarking. This way, as proposed by [9], a robustness benchmark is essentially a suite of robustness tests or stimuli. A robustness benchmark stimulates the system in a way that triggers internal errors, and in that way exposes both programming and design errors in the error detection or recovery mechanisms. Systems can be differentiated according to the number of errors uncovered.

Many relevant studies [15–19] evaluate the robustness of software systems, nevertheless, [10,11] are the ones that present the most popular robustness testing tools, respectively, Ballista and MAFALDA.

Ballista is a tool that combines software testing and fault injection techniques. The main goal is to test software components for robustness [11], focusing specially on operating systems. Tests are made using combinations of exceptional and acceptable input values of parameters of kernel system calls. The parameter values are extracted randomly from a database of predefined tests and a set of values of a certain data type is associated to each parameter. The robustness of the target OS is classified according to the CRASH scale that distinguishes the following failure modes: *Catastrophic* (OS becomes corrupted or the machine crashes or reboots), *Restart* (application hangs and must be terminated by force), *Abort* (abnormal termination of the application), *Silent* (no error is indicated by the OS on an operation that cannot be performed), and *Hindering* (the error code returned is not correct).

Ballista was initially developed for POSIX APIs (including real time extensions). More recent work has been developed to adapt it to Windows operating systems [20]. In this study, the authors present the results of executing Ballista-generated exception handling tests over several functions and system calls in Windows 95, 98, CE, NT, 2000, and Linux. The tests were able to trigger system crashes in Windows 95, 98, and CE. The other systems also revealed robustness problems, but not complete system crashes. Ballista was also adapted to various CORBA ORB implementations [21]. For the CORBA ORB implementations the failure modes were adapted to better characterize the CORBA context.

Microkernel Assessment by Fault injection AnaLysis and Design Aid (MAFALDA) is a tool that allows the characterization of the behavior of microkernels in the presence of faults [10]. MAFALDA supports fault injection both into the parameters of system calls and into the memory segments implementing the target microkernel. However, in what concerns to robustness testing, only the fault injection into the parameters of system calls is relevant. MAFALDA has been improved afterwards (MAFALDA-RT) [22] to extend the analysis of the faulty behaviors in order to include the measurement of response times, deadline misses, etc. These measurements are quite important for real-time systems and are possible due to a technique used to reduce the intrusiveness related to the fault injection and monitoring events [23]. Another study [24] has been carried out to extend MAFALDA in order to allow the characterization of CORBA-based middleware.

The success of robustness testing on operating systems was so great that studies have appeared on particular components of OSs. In [25] a study on the robustness properties of Windows device drivers is presented. Recent OS kernels tend to become thinner by delegating capacities on device drivers (which currently represent a substantial part of the OS code), and a large part of system crashes can be attributed to these device drivers because of residual software bugs. The evaluation presented in [25], focusing on the robustness properties of Windows (XP, Server 2003, and Vista), concluded that, in general, the tested OS versions appear to be vulnerable, as some of the injected faults caused system crashes or hangs, which highlights the importance of robustness testing.

One of the first examples of robustness testing applied to Web services is [26]. This paper proposes a technique to test Web services using parameter mutation analysis. The Web services description file (defined using WSDL) is parsed initially and mutation operators are applied to it, resulting in several mutated documents that will be used to test the service. The set of mutations that can be applied to the interface document apply to operation calls, in particular, the input messages, output messages and their data types. Authors tried to focus on errors that developers make when defining, implementing and using these interfaces. In a robustness testing approach this is not typically taken into account, as the goals of the approach are related with assessing the robustness of the system in any conditions, and not with the representativeness of the injected faults. In spite of the effort set on this approach, the parameter mutation operators are very limited and consist basically on switching, adding, deleting elements, or setting complex types to null.

In [27] an approach based on XML perturbation is presented. In this case, a formal model for XML schema is defined and a method to create virtual schema when XML schemas do not exist is presented. Based on the formal model, perturbation operators are designed and these operators are used to modify virtual or real schema, with the goal of satisfying defined test coverage criteria. Test cases, as XML messages, are generated to satisfy the perturbed XML schema. Although it represents a detailed study, the coupling that is done to the XML (eXtensible Markup Language) technology invalidates any kind of test generalization (i.e., it does not apply to other technologies since it is tightly coupled to XML).

An approach enabling a testing methodology for white-box coverage testing of error handling code is presented in [28]. The approach uses a compile-time analysis that allows compiler-generated instrumentation to control a fault injection process and keep track of the exercised recovery code. The approach is designed for Java applications and it was tested with applications that can be used within a network; namely, an FTP server (FTPD), a File Server (JNFS), Haboob (a Web server), and Muffin (a proxy server). The methodology used leaves, on average, approximately 16 % of the exception-catch links uncovered, which translates into the need of further examination by a human tester. Our approach can be used without code access and does not require that the Web services tested be written in Java (or any other language).

Considering that the quality of a composed service may depend on the ability of its component services to react to unforeseen situations, such as data quality problems and service coordination problems, an approach is proposed in [29] to analyze the quality of composed services using fault injection techniques. The technique is based in two main aspects: the reactions of the composed service to data faults and the effect of delays on composed services. The data faults include substituting digits in numbers, reversing characters in strings and dates, representing dates in different formats. However, some limit conditions could also have been used in this study (for instance like the ones used in our work that consider data domains and explore their limits).

A framework for the robustness testing of service compositions built with WS-BPEL is presented in [30]. The composition of multiple services may result in a compound service that carries more robustness problems than a single service, due to the complexity usually involved in this kind of environments. The execution of tests using a number of faulty conditions or exceptional cases is crucial to build robust compositions of services. However, testing the service composition for robustness is more difficult than testing regular applications because the number of test scenarios and the cost of testing greatly increase with the increasing number of component services. The testing framework presented introduces a virtual service, which corresponds to a real component service. It simulates abnormal situations within the real service and allows the verification of the robustness of Web services compositions against various errors and/or exceptional cases. The approach described in this paper does not directly deal with all aspects involved with testing services compositions. However, it can be used as basis to test individual components and, in this way, characterize the overall robustness of the composition.

3 Robustness testing approach

Our proposal for Web services robustness testing is based on erroneous call parameters, including both malicious and non-malicious inputs. The robustness tests consist of combinations 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 type of each parameter. In practice, the approach for Web services robustness benchmarking includes the following key components:

- *Workload* represents the work that the Web service must perform during the benchmark execution.
- *Robustness tests* faultload consisting of a set of invalid call parameters that are applied to the target service to expose robustness problems.
- *Failure modes classification* characterize the behavior of the Web service while executing the workload in presence of the robustness tests. This includes also the classification of the behavior of the service using a set of generic but descriptive tags. The goal is to enable a more consistent understanding of the robustness level of the service under test.

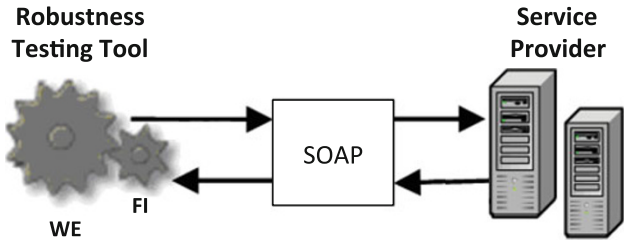

Fig. 2 Test configuration required

In addition to these components the benchmark also specifies the testing setup and procedure. The first defines the setup required to run the tests and the second describe the procedures that must be followed to implement and run the benchmark.

Figure 2 presents the proposed generic setup to benchmark the robustness of Web services. The main element is a robustness testing tool that includes two main components: a workload emulator (WE), which acts as a Web service consumer by submitting Web service calls (it emulates the workload), and a fault injector (FI) that automatically injects erroneous input parameters. An important aspect is that the source code of the Web service under testing is not required for the robustness tests generation and execution. This is true for both the provider and the consumer sides.

The testing procedure is a description of the steps and rules that must be followed during the benchmark implementation and execution. For Web services robustness testing, we propose the following set of steps:

1. *Tests preparation* analysis of the Web service under test in order to gather relevant information.
2. *Workload generation and execution* execution of the workload in order to understand the expected correct behavior of the Web service.
3. *Robustness tests generation and execution* execution of the robustness tests in order to trigger faulty behaviors that may disclose robustness problems.
4. *Web service characterization* failure modes and service behavior identification based on the data collected in steps 2 and 3.

These steps can be implemented by any robustness testing tool that follows the approach presented here and detailed in the following sections. As referred, wsrbench is a tool that can be used for testing the robustness of Web services (see [31] for details on practical use of the tool).

3.1 Tests preparation

Before generating and executing the workload and the robustness tests we need to obtain some definitions about the Web service operations, parameters, and data types. As mentioned before, a Web service interface is described as a WSDL file. This XML file is automatically processed to obtain the list of operations, parameters and associated data types. The information describing the structure and type of all inputs and outputs of each operation is usually found in a XML Schema file (a XSD file that describes the structure of an XML object), which is referenced by the original WSDL [2,32].

The next step consists of gathering information on the valid domains for all input and output objects. For this purpose, the XSD file, that describes all parameters, is searched. This file may also include information about valid values of each parameter, provided that XSD schema restrictions are defined. It is rare, however, to find the valid values for each parameter expressed in a WSDL/XSD pair. This is mainly due to the lack of integrated tools that could be used to define domain values and the inexistence of support for expressing dependencies between two (or more) parameters of a given service operation. This way, the benchmark user is allowed to provide information about the valid domains for each parameter. This includes values for parameters based on complex data types (which are decomposed in a set of individual parameters) and domain dependencies between different parameters.

Table 1 shows an example of how the benchmark user can specify the domains for each parameter. This is the information needed to support the workload generation and tests execution. Figure 3 presents an excerpt of a WSDL file for a Web service named *ValidateService*. The figure illustrates how the information introduced by the user maps to the WSDL definitions. The service provides the following operation to clients: *ValidateObject (String name, int number)*.

3.2 Workload generation and execution

The workload defines the work that has to be done by the Web service during the benchmark execution. Three different types of workload can be considered for robustness benchmarking purposes: real workloads, realistic workloads, and synthetic workloads.

Real workloads are made of applications used in real environments. Results of benchmarks using real workloads are normally quite representative. However, several applications are needed to achieve good representativeness and those applications frequently require some adaptation. Additionally, the workload portability is dependent on the portability requirements of all the applications used in the workload.

Realistic workloads are artificial workloads that are based on a subset of representative operations performed by real systems. Although artificial, realistic workloads still reflect real situations. It is important to note that realistic workloads are quite representative and are more portable than real workloads.

Table 1 Example of the specification of the parameters of Web services operations

Parameter	Data type	Domain specification	Description
name	String	[a-z|A-Z]{2,16}	The valid domain includes all the strings with a minimum of 2 and a maximum of 16 characters. The valid characters are only the letters from A to Z (both uppercase and lowercase)
number	Int	[0-199]	The valid values are integer numbers from 0 to 199
return	String	[OK|NOK]	The method return value admits only two strings: OK and NOK

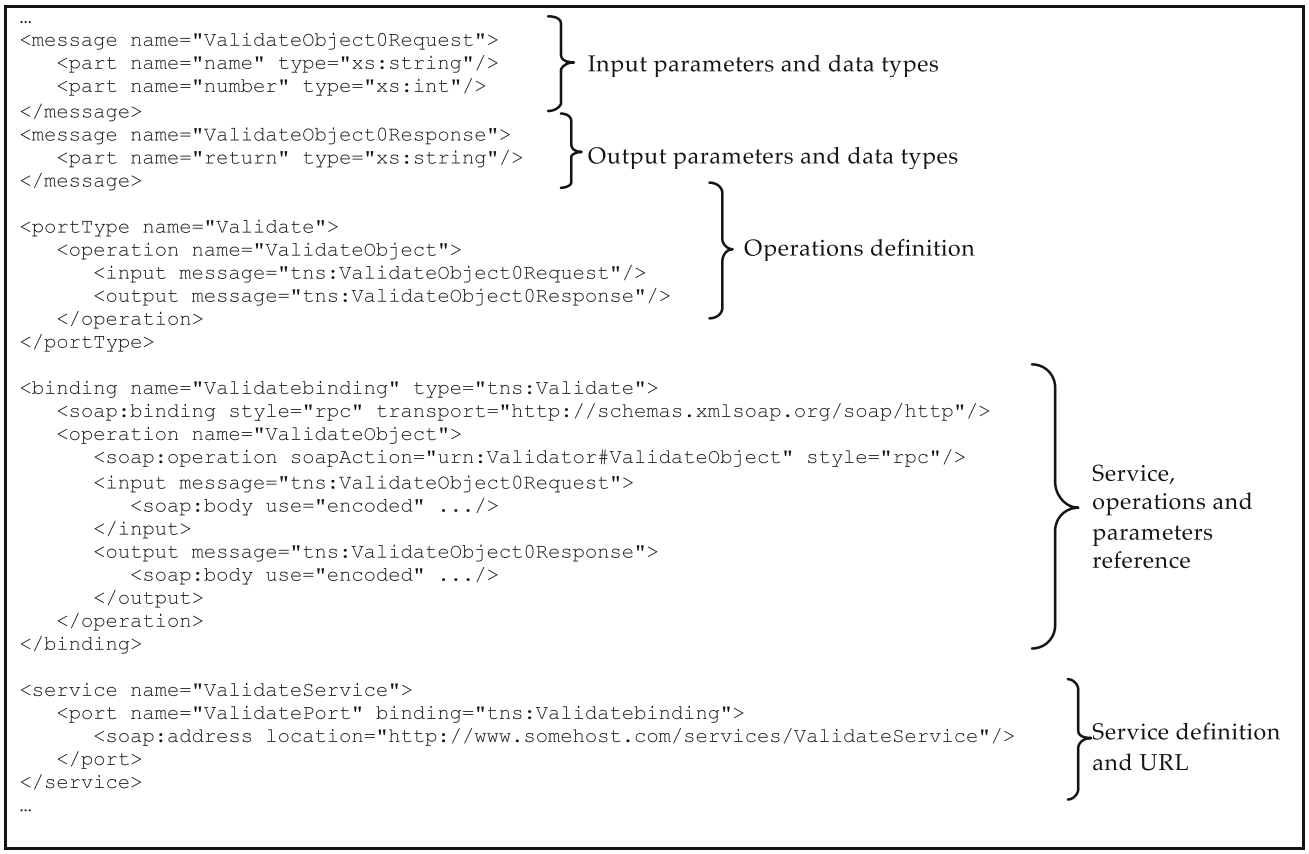

Fig. 3 Example of a WSDL file

In a Web services environment, a *synthetic workload* can be a set of randomly selected service calls. Synthetic workloads are easier to use and may provide better repeatability and portability when comparing to realistic or real workloads. However, the representativeness of these workloads is lower.

As it is not possible to propose a generic workload that fits all Web services (because different Web services have different interfaces and behaviors), we need to generate a specific workload for the Web service under testing. In our approach the following options can be used for the generation of the workload:

- *User defined workload* the benchmark user implements a workload emulation tool based on the knowledge he has about the service being tested. This emulator can be integrated in our robustness-testing setup (by using the wsr-bench API [31] and performing some simple configurations). To simplify the workload emulator generation there are several easy to use client emulation tools like soapUI [33] and WS-TAXI [34] that can be applied. In theory any similar tool can be used, although in practice the amount of technical effort required can vary, depending on the specificities of the workload generation tool being used.
- *Random workload* this workload can be generated automatically using the Web service definitions mentioned above. For every parameter of each operation, a set of valid input values (values in the parameter domain specified by the benchmark user) is generated randomly. The number of generated values is configurable by the user if using

wsrbench [31] as testing tool. Those values are combined in such a way that guarantees a large number of valid execution calls, improving the coverage of the workload. The benchmark user must specify the intended total number of executions for each operation.

One of the problems related to the random generation of the workload is that the representativeness of the Web service calls is not guaranteed. For instance, some services may require highly diverse inputs to achieve satisfactory code coverage (specifically, statement coverage). Depending on the service under test, a user can define a more extensive workload in order to achieve higher code coverage (when the code coverage depends on the input values). This definition can be further tuned if the user has some specific knowledge of the services being tested (e.g., if he is the service provider). However, randomly generated values are appropriate in most cases [9].

An alternative is to generate the workload based on the automated analysis of the Web service source code and on some simple definitions provided by the benchmark user (obviously, as the source code is needed, this alternative can only be used by service providers). The work from Santiago et al. [35] proposes the use of state charts for the automated test case generation (our goal is to use a similar approach to generate the workload). Another approach, based on de Barros et al. [36], is to generate the workload using the characterization of real load patterns through the application of Markov chains.

The execution of the workload (without considering invalid call parameters) includes several tests where each test corresponds to several executions of a given operation of the Web service. The number of times the operation is executed during each test depends on the size of the generated workload (which can be specified by the benchmark user). The goal of this step is simply to understand the typical behavior expected from the Web service (e.g., typical response format). This information is not used in the definition of the tests, but can be used in the classification of the tests results, either by individually comparing the tests results with the regular workload output, or by using an automatic robustness classification procedure based on machine learning techniques [37].

3.3 Robustness tests generation and execution

Robustness testing involves parameter tampering at some interface level. Thus, a set of rules must be defined for parameter mutation. An important aspect is that these rules must be focused on limit conditions that typically represent difficult validation aspects (which are typically the source of robustness problems), such as:

- Null and empty values (e.g., null string, empty string).
- Valid values with special characteristics (e.g., non-printable characters in strings, valid dates by the end of the millennium).
- Invalid values with special characteristics (e.g., invalid dates using different formats).
- Maximum and minimum valid values in the domain (e.g., maximum value valid for the parameter, minimum value valid for the parameter).
- Values exceeding the maximum and minimum valid values in the domain (e.g., maximum value valid for the parameter plus one).
- Values that cause data type overflow (e.g., add characters to overflow string max size, duplicate the elements of a list, and replace by maximum number valid for the type plus one).
- Malicious values that try to explore potential security vulnerabilities [38–40], including SQL Injection and XPath Injection, two key vulnerabilities frequently found in Web services code [41].

The proposed generic rules, summarized in Table 2, are defined for each data type and are based on the validation issues presented above and on previous work on service robustness testing [10,11]. Regarding the detection of robustness problems that originate security vulnerabilities, the set of attack types considered is based on the compilation of the types used by a large set of commercial and open-source scanners, namely the latest versions of, HP WebInspect, IBM Rational AppScan, Acunetix Web Vulnerability Scanner, Foundstone WSDigger, and wsfuzzer. This list was analyzed and complemented based on practical experience and based on information on Web application attack methods available in the literature (e.g., [38–40]). For the time being, the list includes 146 attack types (see Table 3 for examples on SQL Injection).

It is important to emphasize that the proposed mutation rules are quite extensive and try to focus on testing limit conditions that are typically the source of robustness problems. The list of mutation rules can be easily extended to accommodate more types of robustness problems and vulnerabilities and to address new techniques for robustness problems exploitation.

To improve the coverage and representativeness of the benchmark, the robustness tests must be performed in such way that fulfills a set of key goals/rules:

- All the operations provided by the Web service must be tested.
- For each operation all the parameters must be tested.
- For each parameter all the applicable tests must be considered. For instance, when considering a numeric parameter, the fifteen tests related to numbers must be performed (see Table 2).

Table 2 Parameter mutation rules

Data type	Test name	Parameter mutation
String	StrNull	Replace by null value
	StrEmpty	Replace by empty string
	StrPredefined	Replace by predefined string
	StrNonPrintable	Replace by string with nonprintable characters
	StrAddNonPrintable	Add nonprintable characters to the string
	StrAlphaNumeric	Replace by alphanumeric string
	StrOverflow	Add characters to overflow max size
	StrMalicious	Malicious predefined string (see examples in Table 3)
Number	NumNull	Replace by null value
	NumEmpty	Replace by empty value
	NumAbsoluteMinusOne	Replace by −1
	NumAbsoluteOne	Replace by 1
	NumAbsoluteZero	Replace by 0
	NumAddOne	Add one
	NumSubtractOne	Subtract 1
	NumMax	Replace by maximum number valid for the type
	NumMin	Replace by minimum number valid for the type
	NumMaxPlusOne	Replace by maximum number valid for the type +1
	MumMinMinusOne	Replace by minimum number valid for the type −1
	NumMaxRange	Replace by maximum value valid for the parameter
	NumMinRange	Replace by minimum value valid for the parameter
	NumMaxRangePlusOne	Replace by maximum value valid for the parameter +1
	NumMinRangeMinusOne	Replace by minimum value valid for the parameter −1
List	ListNull	Replace by null value
	ListRemove	Remove element from the list
	ListAdd	Add element to the list
	ListDuplicate	Duplicate elements of the list
	ListRemoveAllButFirst	Remove all elements from the list except the first one
	ListRemoveAll	Remove all elements from the list
Date	DateNull	Replace by null value
	DateEmpty	Replace by empty date
	DateMaxRange	Replace by maximum date valid for the parameter
	DateMinRange	Replace by minimum date valid for the parameter
	DateMaxRangePlusOne	Replace by maximum date valid for the parameter plus one day
	DateMinRangeMinusOne	Replace by minimum date valid for the parameter minus one day
	DateAdd100	Add 100 years to the date
	DateSubtract100	Subtract 100 years to the date
	Date2-29-1984	Replace by the following invalid date: 2/29/1984
	Date4-31-1998	Replace by the following invalid date: 4/31/1998
	Date13-1-1997	Replace by the following invalid date: 13/1/1997
	Date12-0-1994	Replace by the following invalid date: 12/0/1994
	Date8-31-1992	Replace by the following invalid date: 8/31/1992
	Date8-32-1993	Replace by the following invalid date: 8/32/1993
	Date31-12-1999	Replace by the last day of the previous millennium
	Date1-1-2000	Replace by the first day of this millennium
Boolean	BooleanNull	Replace by null value
	BooleanEmpty	Replace by empty value
	BooleanPredefined	Replace by predefined value
	BooleanOverflow	Add characters to overflow max size

Table 3 Examples of SQL Injection attack types

Parameter mutations
”or 1=0 –
”or 1=1 –
”or 1=1 or ””=”
’ or (EXISTS)
’ or uname like ’%
’ or userid like ’%
’ or username like ’%
’ UNION ALL SELECT
’ UNION SELECT
char%2839%29%2b%28SELECT
char%4039%41%2b%40SELECT
" or 1=1 or ""="
' or ''='

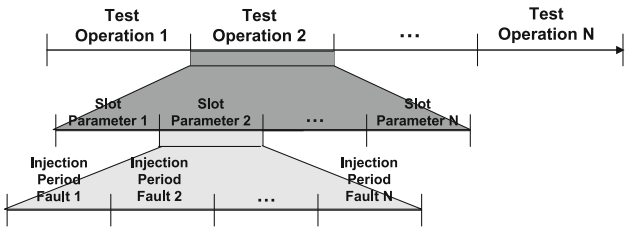

Fig. 4 Execution profile of the step 2 of the benchmark

The number of tests that are generated can vary according to the knowledge the tester has about the service. When in presence of a service whose execution coverage largely depends on the input parameters, it may be helpful to generate more tests (i.e., repeat the tests with new randomly generated values) in order to have an adequate coverage for the complete set of tests. Typically, service providers are in a better position to decide the extension of the tests; however, this is a configuration parameter that can also be set without requiring extra knowledge about the service.

The execution of the robustness tests consists of running the workload in presence of the invalid call parameters. As shown in Fig. 4, this 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 system state is explicitly restored at the beginning of each injection period and the effects of the faults do not accumulate across different periods. However, this is required only when the benchmark is used by the Web service provider as in many cases the consumers are not able to restore the Web service state (since it is being tested remotely). In fact, a potential problem related to the use of the benchmark by consumers is that they cannot control the state of the Web service under testing and, in some cases, are not allowed to test the services. In addition, some tests may change the state of the service (e.g., change the state of the data used), which is also a problem (obviously not relevant for stateless Web services). These problems are minimized when the providers supply a parallel infrastructure that can be used by consumers only for testing purposes. This parallel infrastructure is frequently required by consumers before start using a paid service. Note that, these problems do not affect the ability of providers to test stateful Web services as they can test the code in isolated testing environments (which is typically the testing approach for any software artifact).

During the execution of the robustness tests, the fault injector intercepts all the SOAP messages sent by the emulated clients (generated by the workload emulator component) to the server (see Fig. 2). The XML is modified according to the robustness test being performed and then forwarded to the server. The server response is logged by the robustness-testing tool and used later on to analyze the behavior of the Web service in the presence of the invalid call parameters injected.

An important aspect is that the benchmark user may be interested in repeating the robustness tests. Although repeating the robustness tests using the same workload typically leads to the same results, performing additional tests with a different workload may be a simple way of disclosing more robustness issues.

Notice that, the procedure described above can be executed at runtime, i.e., during real service operation. Obviously, this scenario is applicable when the execution of these tests do not impact: (a) the real clients’ needs (for instance, by degrading the quality of service to unacceptable levels); (b) the service itself (for instance, by changing its internal state, when this has impact on the future operation of the service); (c) external services (i.e., in other domains or owned by distinct organizations) used by the service being tested. It is the responsibility of the tester to identify these cases and take proper measures (for instance by isolating the system, when possible). In fact, in a deployed system, this testing activity usually implies some kind of impact and, frequently, the solution is to test the system offline. In this case, if the service under test includes the invocation of external services that are not available (these external services may be unavailable for invocations by isolated installations of the service under test), the tester can use mock objects that respond with values within well-known domains. Anyway, it is the responsibility of the tester to provide an accurate image of the service that is going to be tested for robustness.

3.4 Web services characterization

Web services robustness is characterized based on a set of *failure modes*, complemented with a detailed *tag-based categorization of the service responses* that helps understanding the source of the observed failures. Robustness characterization includes an automated analysis of the responses obtained in order to distinguish valid replies from replies that reveal robustness problems in the service being tested. wsr-bench (the tool used in our experimental evaluation) is able to perform a preliminary verification of the tests results. However, in some cases, automated identification is not enough to decide if a given response is due to a robustness problem or not (e.g., in many cases it is difficult to automatically decide whether a given response represents an expected or unexpected behavior). We have recently studied the applicability of machine learning algorithms, typically used in text classification tasks, in the classification Web services robustness [37]. We found out that it is possible to obtain good results with these algorithms, i.e., a tool based on these algorithms can be used to automate the process effectively. Although the algorithms revealed that they do not perform perfectly, they can still be used to reduce the amount of manual effort required, specially, when a large amount of tests is executed. However, manual validation can still be an option for small to medium-sized services. In fact, it is normally straightforward for developers and testers to classify a given response as expected or not expected.

The robustness of services can be classified according to the severity of the exposed failures. A potential approach can be to use a well-known classification, such as the CRASH scale [11], as basis for services characterization and tailor this scale according to the specificities of the class of services targeted. The following points show how the CRASH failure modes can be adapted to the Web services environment:

- *Catastrophic* The Web service supplier (i.e., the underlying middleware) becomes corrupted, or the server or operating system crashes or reboots.
- *Restart* The Web service supplier becomes unresponsive and must be terminated by force.
- *Abort* Abnormal termination when executing the Web service. For instance, abnormal behavior occurs when an unexpected exception is thrown by the service implementation.
- *Silent* No error is indicated by the Web service implementation on an operation that cannot be concluded or is concluded in an abnormal way.
- *Hindering* The returned error code is incorrect.

The problem is that, in most cases, the use of the CRASH scale is very difficult or even not possible. For example, when the benchmark is executed by service consumers, it is not possible to distinguish between a catastrophic and a restart failure mode, as the consumer does not have access to the server where the service is running. Additionally, the CRASH scale does not allow a detailed categorization of the sources of the observed failures, not allowing, for example, distinguishing robustness problems that lead to security vulnerabilities. To face this issue, we propose a more adequate classification scale based on two elements: failure modes classification and service behavior classification.

The *failure modes classification* includes only those failures that can be effectively observed while conducting robustness testing of Web services code from both providers and consumers point-of-view. This way, based on our observations during two large set of Web services robustness testing experiments (see Sect. 4), we propose the use of the following failure modes:

- *Correct* The Web service response in the presence of an invalid input is correct (i.e., the Web service responds with an expected exception or error code). Although this is not really a failure mode, it allows characterizing a correct service behavior in the presence of invalid inputs.
- *Crash* An unexpected exception is raised by the Web service and sent to the client application.
- *Error* The service replies with an expected object that, however, encapsulates an error message that indicates the occurrence of an internal problem.

Figure 5 summarizes the procedure for the analysis of Web service responses and identification of robustness problems. The termination tags containing 0, 1, and 2 indicate, respectively, a Correct response, a Crash, and an Error.

The analysis starts with the identification of a SOAP envelope (1) in the service response (i.e., we verify if the response follows the general format of the SOAP protocol). When not present (e.g., a plain HTTP reply is received instead) it may indicate that there is some kind of problem (e.g., a server incorrectly configured) that is preventing the Web service to process the SOAP request, but, as a solid conclusion cannot be drawn, we discard the response from the analysis. The next step is to verify if a SOAP Fault (which holds errors and status information) is present (2). If not, another type of message indicating an error (3) may be present (e.g., an application specific error message). If it is a regular response it is marked as Correct. On the other hand, if it is an error message that indicates an unexpected error (4) (e.g., database error when a database-related fault has been injected) the response is marked as an Error. If the error is expectable, the response is marked as Correct.

When in presence of a SOAP Fault, the presence of a soap:Server tag is an indication that the fault cause is at the server-side (5). When these tags are not present it is highly likely that the fault has its origin in the SOAP Stack. For instance, a soap:Client tag is observed when a numeric value is rejected at the stack because it is larger than its datatype and

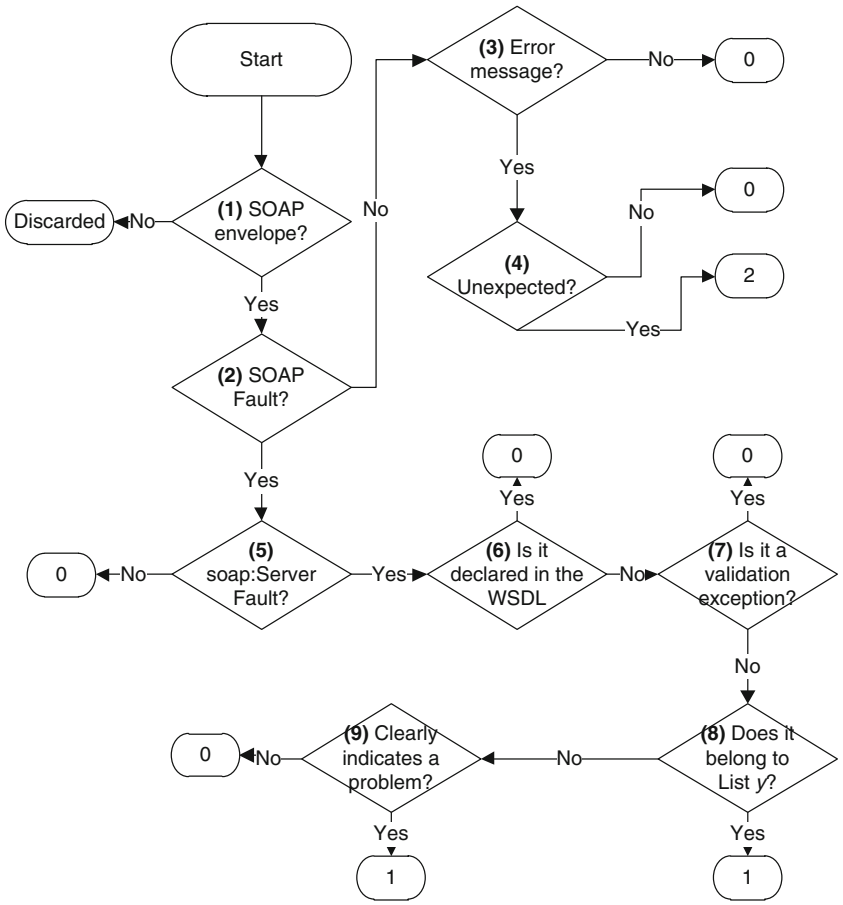

Fig. 5 Flowchart for the identification of robustness problems

cannot be deserialized for delivery to the service implementation. Issues that are handled at stack level are marked as Correct. The next step (6) is to check if the response represents an Exception that is declared in the WSDL file (declared exceptions are marked as Correct). When the exception type is not present in the WSDL and does not represent an input validation exception (7) (validation exceptions are marked as Correct), it is compared against List *y*—a list of exceptions that represent typical robustness problems (8) (e.g, SQLException, NullPointerException, etc.). Responses that hold these exceptions are marked as Crash. If a different exception is observed and it clearly indicates the existence of an internal unexpected problem (9) (e.g., a username, database vendor, or filesystem structure disclosure), the response is marked as a Crash. Responses that do not indicate the presence of a problem are marked as Correct.

To gather the relevant information needed to correct or wrap the identified robustness problems, the failure modes characterization needs to be complemented with a detailed analysis of the observed service behavior in order to understand the source of the failures (i.e., defects). Obviously, the source of the failures depends on several specificities of the services being tested (e.g., programming language, external interfaces, operational environment), which complicates the definition of a generic classification scale. However, based on our experiments (see Sect. 4) and on the results from previous work [31], we propose the use of a *tag-based behavior classification* system. This serves as a support for a detailed classification of the observed service behaviors.

As presented in Table 4, the tags included in our classification system were designed to be as comprehensive and generic as possible. On one hand there was an effort to minimize the number of tags, but, on the other hand, we wanted our tags to be as descriptive as possible. In fact, for every problem, any tag or tag combination can be decisive in helping a developer to produce an adequate fix. For instance, when the problem is marked with the data access operations tag, this is a strong indicator that the developer should focus his attention on the persistence modules of his application. Due to the importance and difficulty of creating clear and generic, but also descriptive tags, the tag-based classification was created iteratively while analyzing the 420,375 distinct service responses obtained during the experimental evaluation presented in Sect. 4.

Table 4 Classification for failure symptoms

Tag	Description
Server resource disclosure	Information about the servers filesystem or a physical resource is disclosed
Conversion issues	A conversion problem exists in the service
Wrong type definition	The service operation expects a value whose type is not consistent with what is announced in the services WSDL file
Data access operations	A problem exists related with data access operations
Specific server failure message	An exception is thrown and application or development specific information is revealed. This information is, however, generally too vague or too context-specific to allow us an association with another tag
Persistence error	An exception is thrown indicating a persistence-related problem. This is typically an SQL exception that is thrown as a consequence of improper parameter handling
Argument out of format	The service operation requires a restriction on a parameters format. However, no restriction is specified in the WSDL file, allowing clients to invoke the operation with an out-of-format parameter
Wrapped error information	An error response is wrapped in an expected object. The response indicates the occurrence of an internal error
Array out of bounds	Occurrence of an array access with an index that exceeds the limits of the array (upper or lower)
Null references	A null pointer or reference exception is thrown by the server application
Command or schema disclosure	An internal command is totally or partially disclosed (e.g., an SQL statement is revealed), or the data schema is revealed (e.g., the table names in a relational database are revealed)
Arithmetic operations	An indication of an arithmetic error is returned by the service operation
Division by zero	The service operation indicates that a division by zero has been attempted
Internal function name disclosure	The name of an internal or system procedure is disclosed (e.g., a database stored procedure)
System vendor disclosure	System vendor information is disclosed (e.g., database or operating system vendor)
Overflow	The service operation is unable to properly handle a value that is larger than the capacity of its container, indicating the occurrence of an overflow error
System instance name disclosure	The name of a system instance is revealed to the client (e.g., a database instance name)
System user disclosure	A system username or password is exposed to the client (e.g., the username used to connect to a database or the operating system username)
Other	Any other service response that does not fit into any of the previous categories

We are aware that the proposed tag classification may be incomplete. However, it can easily be extended based on additional test results from other Web services. Also, developers can adapt the tagging system to their specific scenario, taking into account factors like the programming language, the Web services stack, the application server, etc. An important aspect is that, using the proposed tags does not require access to the Web service internals. In fact, tags identification is based on the exceptional behavior of the tested service as reported to the client applications. Obviously, having access to the service code (e.g., while testing the Web service before deployment) allows improving the classification accuracy.

A key aspect is that it is not possible to design a generic classification system that is able to describe all existing service behaviors. In fact, because Web services generally encompass the use of highly diverse systems and are based in technology that suffers continuous advance and transformations, a tag-based system like the one we propose will always need to be extended or adapted in the future. In this sense, we do not provide this as a closed classification. Instead, the benchmark user is free to use this classification, extend it, or devise any other classification tailored for the specific Web services being tested.

4 Experimental results and discussion

In this section we demonstrate the Web services robustness approach. Besides demonstrating the use of the benchmark, the experiments presented in this section try to give answer to the following questions:

- Can providers and consumers use robustness benchmarking to test Web services?
- Can robustness benchmarking be used to improve the robustness and security of Web services code?
- Can the benchmark be used to compare different implementations of a given Web service?

wsrbench [31], an online tool that can be used to perform robustness tests on Web services, has been used to support the experimental evaluation. This tool, publicly available at http://wsrbench.dei.uc.pt, implements the Web services testing approach proposed in this paper and provides a Web-based interface that allows users to configure, execute tests, and also visualize and analyze the results of tests. wsrbench is free, open-source, and easy to use, requiring only a very simple registration and posterior authentication process. Details on its architecture can be found at [31].

Two experimental scenarios were considered. In the first, we evaluated the robustness of 250 public Web services, comprising 1,204 operations and 4,085 parameters, deployed over 44 different country domains, and provided by 150 different relevant parties. These parties include several well-known companies like Microsoft, Volvo, Nissan, and Amazon; multiple governmental services; banking services; payment gateways; software development companies; internet providers; cable television and telephone providers, among many others. The complete list of tested services includes Web services deployed on 17 distinct server platforms and 7 different Web service stacks. A complete and detailed list can be found at [42].

In the second scenario, we have applied the benchmark to test 29 home-implemented service operations. These included 5 open-source services adapted from code publicly available on the Internet [43], and two different implementations (4 operations each) of the Web services specified by the standard TPC-App performance benchmark [44]. In both cases, the Web services have been coded by two experienced Java programmers with more than 2 years of experience.

TPC-App is a performance benchmark for Web services infrastructures widely accepted as representative of real environments. The business represented by TPC-App is a distributor that supports user online ordering and browsing activity. The two implementations tested in this work have been developed using N-Version programming [45].

As application server we used JBoss [46], which is one of the most complete J2EE application servers in the field and is used in many real scenarios. Oracle 10g was used to support the persistence requirements of the services.

The testing procedure consisted of using the default configuration of wsrbench to generate all necessary Web service requests and collecting and interpreting responses. This includes generating a workload following the random approach (10 requests per operation parameter, see Sect. 3.4). In general, and following the terms used in Sect. 3, one fault was applied in each injection period. For each operation tested, several slots were considered (one per each parameter) and in each slot all possible faults were used.

As referred, more (or less) requests can be used. Notice that the core of our proposal is not a workload generation approach or tool, it is instead the whole process for assessing the robustness of Web services. Multiple configurations, or even other strategies in components of the approach may be used with better or worse results. Our core contribution is indeed the approach and, as such, in some cases, it may not even be possible to define concrete aspects of the testing approach that have influence on the outcome of the tests. For example, in the case of public Web services, no input domains were specified as we did not possess information regarding the services being tested (thus, by default, the input domain of a given parameter corresponds to the parameter data type). On the other hand, for the home-implemented Web services those domains were defined based on the Web services specification (for the two TPC-App services [44]) and on a previous analysis of the source code (for the five Web services adapted from code publicly available on the Internet [43]).

Although the wsrbench tool automatically interprets the Web services responses and classifies them in terms of failure modes, data was also manually analyzed in order to confirm the robustness properties of each tested service and to tag the Web services behavior. The reason is that, for the time being, the wsrbench tool is not able to perform automatic tagging (using the tags in Table 4). We are currently researching the use of Conventional Machine Learning algorithms [47] to automate this task, but this is out of the scope of the present paper and therefore has not been used in the experimental evaluation.

4.1 Public Web services

Two hundred and fifty publicly available Web services, including more than one 1,200 operations, were tested for robustness. These services were obtained using Seekda (http://webservices.seekda.com/)—a Web service search engine. This is, to our best knowledge, the largest Web service search engine currently available on the Internet. The service selection process consisted in introducing technology-related keywords (e.g., Web, service, xml, etc) in the search engine and randomly selecting some services from the search results. The services were then tested using wsrbench according to the previously described procedure. As said, we also double-checked each test result (a total of 420,375 responses) to confirm the failure mode and to build our tag-based classification system. This process was done by two distinct and independent software developers with more than 4 years experience in developing Java-based Web applications and took about 2 weeks.

Results indicate that a large number of services are currently being deployed and made available to the general public with robustness problems. In fact, 49 % of the tested services presented some kind of robustness issue. This is a very large percentage of problematic services and the problem gains a larger dimension if we consider that a large part of these problems also represent security issues. Figure 6 presents the global results for our public services evaluation in three different granularity perspectives. The service execution granularity presents the analysis from the service perspective, being the service the unit of analysis. Similarly the operation and parameter granularity consider these two items the analysis unit.

If we consider the service execution granularity, we can see that the Correct failure mode was observed at least once for 96 % of the services. In fact, the correct behavior is present

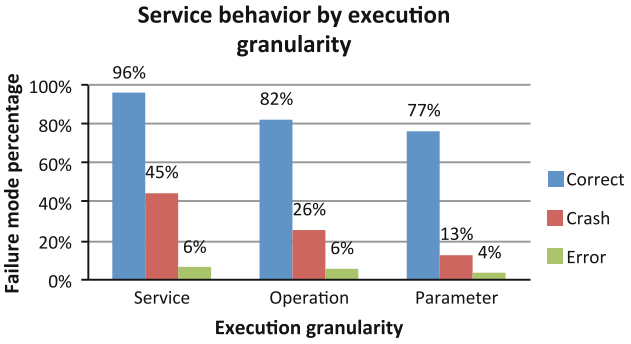

Fig. 6 Global robustness results

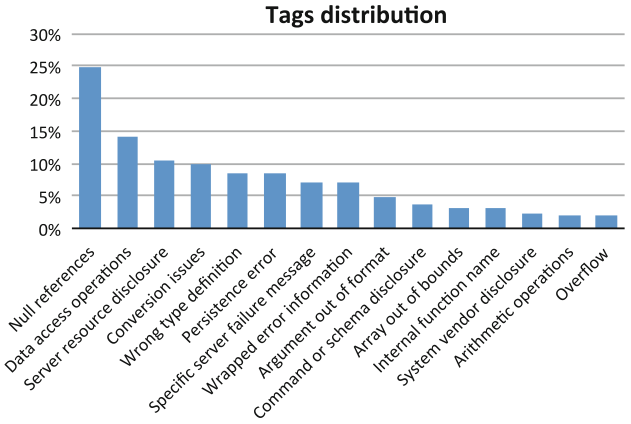

Fig. 7 Distribution of the most frequently observed tags

in almost all the services tested, which indicates that in some point services are able to display an adequate and expectable behavior. However, there are still relatively high percentages of the Crash and Error failure modes, respectively, 45 and 6 %. These are obviously non-additive results as the same service can display multiple failure modes.

We can also observe that the percentages of the different failure modes generally decrease as we increase the analysis granularity. Note that each failure mode only needs to be observed once (in a given parameter), so that we mark the whole parameter, operation, and service with that failure mode. This justifies the fact that higher execution granularities generally display higher percentages for each failure mode. Despite this, the global image is maintained being the Correct and Error failure modes, respectively, the most and least observed modes, whereas the Crash failure mode consistently maintains its middle position.

Besides this global analysis, each response generally represents a rich resource that can enable us to understand the source of failures and obtain a global view about the relative frequency of each observed behavior. This way, the next step is to categorize each received answer using the tags described in Sect. 3.

As the set of responses to be analyzed was very large, we analyzed and tagged each response in multiple iterations. Although we started from a base set of tags, during the analysis process more tags where created whenever a new (previously unseen) problem appeared and no existing tag could be used to describe accurately that problem. These iterations were also necessary to generalize a few tags (e.g., merge two tags into a more generic one).

Figure 7 presents the distribution of the most frequently observed tags (tags representing less than 1 % of the total observed problems are not represented). As we can see, Null references was the issue most frequently seen in all services. This issue was present in 25 % of the tested Web services and is related with the fact that services typically assume that clients will invoke their operations with non-null input parameters. Services tend to expect a correct, non-malicious client and thus provide themselves with no protection, resulting, at its best, in an unexpected exception at the client-side. Clients built on the assumption that the service is robust, or executes accordingly to some specification, can then easily collapse when in presence of an unexpected answer.

Among the most relevant issues are persistence-related problems (observed in 14 % of the services), which, in our experiments, were associated essentially with the use of SQL statements to access a database. This reveals more than a simple SQL construction error. In fact, it shows that the provider does not validate SQL inputs, which may open a door for SQL injection attacks that can compromise the security of the Web service (or of the whole service infrastructure).

Server resource disclosure was also a frequently observed issue (in 10 % of the services). In fact it was frequently observed that, when in presence of an invalid input, some services disclose not only development information (e.g., a tacktrace wrapped in an exception thrown at an unexpected point), but also more critical information (e.g., the partial directory structure of a hard-drive which represents a security issue).

During our analysis we also observed that the Conversion issues and Wrong type definition tags are very frequently associated with each other. In fact, in about 96 % of the observed conversion issues, the problem was caused by an incorrect definition of the datatype. This indicates that many robustness problems are related with the fact that developers often do not choose the adequate datatypes for the parameters of their services. For instance, a given service expects a Number, however, the WSDL document announces that a String is required (that is later handled as a number). In these cases, besides not using the adequate datatype, the service provides no adequate protection against an incorrect or possibly malicious client. Note also that the announcement of an incorrect datatype in a WSDL document can result in severe interoperability issues.

Table 5 Relative distribution of tags per total tag count (public services)

Tag	% Distribution
Null references	22
Data access operations	12
Server resource disclosure	9
Conversion issues	9
Wrong type definition	8
Persistence error	7
Specific server failure message	6
Wrapped error information	6
Other	21

Table 5 presents an analysis of the tags distribution with respect to the total tag count (and considering service granularity). For readability, the table presents the eight most observed tags and aggregates the remaining in the Other group. As shown, the top eight tags presented in Fig. 7 (where results also represent a service granularity but with respect to the total service count) are again displayed as top issues in Table 5. Furthermore, they also maintain their relative positions, with small fluctuations.

The previous paragraphs presented the top issues disclosed in our complete set of experiences. However, all other types of issues can be relevant even if they occur less frequently (more detailed results can be found at [42]). From the analysis of the results, and supported by previous work [12,31], it is clear that the following actions are urgently needed for robust Web services development and deployment:

- Integrate robustness testing in the development cycle. Nowadays, any developer can freely use wsrbench to test its services for robustness. Additionally, we are currently adapting it so that it can be integrated in popular project build and management tools like Maven [48] (which can then be used by developers in any Integrated Development Environment).
- System administrators should also use robustness testing, even when they have legacy services. Testing will enable them to assess services in terms of robustness, and in many cases, security. Frequently service or server configuration is sufficient to hide or correct major issues.
- Support for complete domain expression and announcement in WSDL documents needs to be included in the Web services technology. Such support can help clients to execute services with adequate inputs, preventing accidental robustness problems.
- Easy support for domain validation must be available in Web services development frameworks. Providing developers with easy ways to validate inputs would certainly reduce many of the observed issues.

4.2 Home-implemented Web services

In addition to the extensive study presented before, which was conducted from the point of view of the consumer, we also executed our approach as service providers. For this purpose, we tested two implementations of a subset of the *TPC-App Web services* and a set of 5 *open-source Web services* adapted from code publicly available on the Internet [43] (a total of 29 service operations).

Table 6 includes a list of the tested services and summarizes the results of the experiments, presenting the classification of issues found in each operation tested. The last two columns of the table show the number of problematic parameters (P) with respect to the total number of parameters of the operation (T). In each row under the tags column, the number of problematic parameters per tag is also indicated between parentheses (and whenever a problem is observed).

As we can see, several robustness problems were found in the tested services. Although these tests represent a smaller set than the one used in the public services tests, all failure modes were again observed. For the 29 service operations we observed the Correct failure mode 14 times; the Crash failure mode 15 times; and the Error failure mode only once (in the modify operation of the *PhoneDir* service).

It is important to emphasize that the most frequently observed behaviors in this set of experiments is similar to the ones observed in the tests of the public services. In fact, Null references was the most observed issue, followed by Data access operations and Conversion issues (in roughly similar frequencies).

Concerning the *TPC-App services*, several robustness problems classified as Crash failures were observed for versions A and B. Six out of eight service operations were marked with the Crash failure mode (2 and 4 for implementation A and B, respectively).

Most of the problems found in these services were related with non-existent validation of null parameters, a relevant and important source of failures, as we have seen. However, a more appealing robustness problem was observed for the *newCustomer* service in implementation A. Although the code targeting the validation of a *contactEmail* parameter was in place, too large email addresses caused the Web service to throw a *StackOverflowException*. After some analysis of the code we concluded that the problem resided in the external API that was being used to validate email addresses (Jakarta Commons Validator 1.3.0 [49]). This shows that robustness problems may occur even when programmers pay a great attention to the code correctness. In fact, the use of third party software (as is the case in this example) may raise problems that are not obvious for programmers. Furthermore,

Table 6 Robustness problems observed for the home-implemented services

Service	Operation	Tags	P	T
TPC-App A	changePaymentMethod	Null references (4)	4	4
	newCustomer	Null references (14); Overflow (1)	15	16
	newProducts	–	0	3
	productDetail	–	0	1
TPC-App B	changePaymentMethod	Null references (2)	2	4
	newCustomer	Null references (6)	6	16
	newProducts	Null references (2)	2	3
	productDetail	Null references (1)	1	1
JamesSmith	login	–	0	2
	add	–	0	11
	update	–	0	12
	delete	–	0	1
	search	Null references (1); Data access operations (3)	3	9
Bank3	deleteAcc	Null references (1); Conversion issues (1); Wrong type definition (1)	1	1
	deposit	Null references (4); Conversion issues (1); Wrong type definition (1)	4	4
	displayDeposit	Data access operations (1); Persistence error (1); System vendor disclosure (1)	1	2
	displayInfo	Null references (1)	1	2
	newAccount	–	0	4
	withdrawal	–	0	4
AddComponent	addComponent	Data access operations (2); Persistence error (2)	2	7
Bank	balance	–	0	1
	create	–	0	2
	deposit	Null references (2); Conversion issues (2); Wrong type definition (2)	2	2
	sign	–	0	1
	withdraw	Null references (2); Conversion issues (2); Wrong type definition (2)	2	2
PhoneDir	addNewRecord	–	0	2
	deleteInput	–	0	1
	firstNameWithIt	–	0	1
	modify	Wrapped Error Info (2); Data access operations (2); Persistence error (2); System vendor disclosure (2)	2	2

this type of errors can easily appear or disappear when an apparently harmless update is done to the external libraries commonly required by projects. However, they can be easily detected with the help of robustness testing, which once more gives emphasis to the importance of using robustness testing.

Regarding the *open-source services*, we were able to identify Crash and Error failure modes in, respectively, 8 and 1 operations (out of 16 operations). We can see that in these services we were able to uncover problems related with data persistence operations. More than robustness issues, these usually represent security issues, as typically we observe this behavior in services that do not use parameterized data access queries (which was in fact the case of these services). This is a major concern as it can be an entry point for SQL or XPath Injection attacks, two of the most frequent attack types in the Web environment [38].

Besides data access-related issues, we also detected conversion issues mostly caused by incorrect definition of data types. This had already been a frequently observed issue during the public services tests and a major source of robustness issues.

Table 7 presents the tag distribution considering the service granularity and with respect to the total count of tags found. We can see that the persistence error, has now more weight than what was observed for the public services (see Table 5). Furthermore, the System vendor disclosure tag was not present in the top 8 tags analyzed in Table 5. On the other hand, the Server resource disclosure tag, observed for the public services, was not detected in the home-implemented ones. Obviously, these changes simply reflect the specificities of this smaller set of services and cannot be generalized.

Table 7 Relative distribution of tags per total tag count (home-implemented services)

Tag	% Distribution
Null references	29
Data access operations	19
Persistence error	14
System vendor disclosure	10
Conversion issues	9
Wrong type definition	9
Wrapped error information	5
Overflow	5

Also important is the consumers view of robustness results. When several options for a given service are available, the client can opt for the better one, the one that can give answer to the clients robustness requirements. For instance, from the analysis of the TPC-App results in Table 6, the best option would be to choose the *newProducts* and *productDetail* services from implementation A, and the *changePaymentMethod* and *newCostumer* services from implementation B.

From the provider point-of-view, it is clear that some software improvements are needed in order to solve the robustness problems detected. In fact, after performing the robustness tests we forwarded the results to the programmers in order to get the implementations improved (this is what is expected when robustness problems are detected). Through a detailed analysis of the results the programmers were able to identify solutions for the existing software faults and new versions of the services were developed. The robustness tests were then executed for these new versions and no robustness failures were observed. This shows that this type of testing can be an important tool for developers to improve the robustness of their solutions.

5 Conclusion

This paper proposes a robustness testing approach for Web services. Given the central role that Web services play today, the existence of such benchmark is a valuable tool when providing or consuming Web services. The approach is especially useful to evaluate the robustness of Web services code before deployment (relevant for providers) and to evaluate the robustness of alternative Web services (relevant for consumers).

The approach consists of a set of robustness tests that are applied during Web services execution in order to observe robustness problems. Systems are classified according to the failure modes and the observed behavior. The effectiveness of the proposed approach was shown in the robustness testing of 250 Web services (1,204 operations) publicly available in the Internet and 29 home-implemented service operations, including two different implementations of the services specified by the TPC-App performance benchmark implemented by experienced programmers. Clearly, the approach is useful for consumers and providers. Consumers can use the results of robustness tests to select the most robust services (by comparison, when alternative options are available), while providers can make use of the tests results to assess the robustness of already deployed services. Furthermore, the results obtained in the experimental evaluation (which disclosed numerous issues, including severe security problems) can be used by developers to either correct the problems or create wrappers to protect services against the issues uncovered by the tests. In summary, these results show that it is possible to use robustness testing to identify robustness problems and clearly show that robustness testing can be successfully applied to Web services environments.

A tool implementing the proposed Web services robustness testing approach was implemented. The wsrbench tool 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, so the interested reader can easily use the proposed Web service benchmarking approach in real world Web services scenarios.

As future work, we plan extending our approach to test the robustness of Web service compositions, with focus on their basic service units. In such complex environments, and particularly with the participation of distinct service providers, the application of robustness testing can be crucial to ensure that the behavior of the system meets the composition provider or users expectations.

References

1. Chappel DA, Jewell T (2002) Java Web services: using java in service-oriented architectures, O'Reilly
2. Curbera F et al (2002) Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI. IEEE Internet Comput 6:86–93
3. Bellwood T (ed) (2002) UDDI Version 2.04 API Specification. http://uddi.org/pubs/ProgrammersAPI_v2.htm
4. Andrews T et al. (2003) Business process execution language for Web services, v. 1.1
5. Lee I, Iyer RK (1995) Software dependability in the tandem GUARDIAN system. IEEE Trans Softw Eng 21(5):455–467
6. Kalyanakrishnam M, Kalbarczyk Z, Iyer R (1999) Failure data analysis of a LAN of windows NT based computers. In: Symposium on reliable distributed database systems, SRDS18, Switzerland
7. Sullivan M, Chillarege R (1991) Software defects and their impact on systems availability. A study of field failures on operating systems. In: Proceedings of the 21st Fault Tolerant Computing, symposium, FTCS-21, pp 2–9
8. Weyuker E (1998) Testing component-based software: a cautionary tale. IEEE Softw 15:54–59

9. Mukherjee A, Siewiorek DP (1997) Measuring software dependability by robustness benchmarking. IEEE Trans Softw Eng 23(6):366–378
10. Rodrguez M, Salles F, Fabre J-C, Arlat J (1999) MAFALDA: microkernel assessment by fault injection and design aid. In: 3rd European dependable computing conference, EDCC-3
11. Koopman P, DeVale J (1999) Comparing the robustness of POSIX operating systems. In: Twenty-Ninth annual international symposium on fault-tolerant computing, 1999. Digest of Papers, pp 30–37
12. Vieira M, Laranjeiro N, Madeira H (2007) Benchmarking the robustness of Web-services. In: Proceedings of the The 13th IEEE Pacific Rim dependable computing conference, PRDC07. Melbourne, Victoria, Australia
13. Erl T (2005) Service-oriented architecture: concepts, technology, and design. Prentice Hall, Upper Saddle River
14. Gudgin M et al. (2007) SOAP Version 1.2 Part 1: Messaging framework, 2nd edn, Web Services Activity: XML Protocol Working Group. http://www.w3.org/TR/soap/
15. Miller BP, Koski D, Lee CP, Maganty V, Murthy R, Natarajan A, Steidl J (1995) Fuzz revisited: a re-examination of the reliability of UNIX utilities and services, University of Wisconsin, USA, Research, Report, CS-TR-95-1268
16. Siewiorek DP, Hudak JJ, Suh B-H, Segall Z (1993) Development of a benchmark to measure system robustness. In: 23rd International symposium on fault-tolerant computing, FTCS-23. Toulouse, France, pp 88–97
17. Carrette GJ (1996) CRASHME: random input testing. http://people.delphi.com/gjc/crashme.html
18. Fabre J-C, Salles F, Rodr'guez Moreno M, Arlat J (1999) Assessment of COTS microkernels by fault injection. In: 7th IFIP working conference on dependable computing for critical applications: DCCA-7. CA, USA, San Jose
19. Koopman P et al (1997) Comparing operating systems using robustness benchmarks. The sixteenth symposium on reliable distributed systems, In, pp 72–79
20. Shelton C, Koopman P, Vale KD (2000) Robustness testing of the microsoft Win32 API. In: International conference on dependable systems and networks, DSN2000. NY, USA, New York
21. Pan J, Koopman PJ, Siewiorek DP, Huang Y, Gruber R, Jiang ML (2001) Robustness testing and hardening of CORBA ORB implementations. In: Proceedings of the 2001 international conference on dependable systems and networks, DSN-2001. Gothenburg, Sweden, pp 141–50
22. Rodrguez M, Albinet A, Arlat J (2002) MAFALDA-RT: a tool for dependability assessment of real-time systems. In: IEEE/IFIP international conference on dependable systems and networks, DSN (2002) Bethesda MD , USA
23. Rodrguez M, Fabre J-C, Arlat J (2001) Dependability assessment of real-time systems, LAAS-CNRS, Research, Report, N01–189
24. Marsden E, Fabre J-C (2001) Failure mode analysis of CORBA service implementations. In: Proceedings of the IFIP/ACM international conference on distributed systems platforms, Middleware'2001. Germany, Heidelberg
25. Mendona M, Neves N (2007) Robustness testing of the windows DDK. In: 37th Annual IEEE/IFIP International conference on dependable systems and, networks, pp 554–564
26. Siblini R, Mansour N (2005) Testing Web services. In: The 3rd ACS/IEEE international conference on computer systems and applications, p 135
27. Xu W et al. (2005) Testing Web services by XML perturbation. In: 16th IEEE international symposium on software reliability engineering
28. Fu C, Ryder BG, Milanova A, Wonnacott D (2004) Testing of java web services for robustness. In: Proceedings of the 2004 ACM SIGSOFT international symposium on software testing and, analysis. 2334
29. Fugini MG, Pernici B, Ramoni F (2009) Quality analysis of composed services through fault injection. Inf Syst Front 11:227239
30. Seung HK, Hyeon SK (2009) Robustness testing framework for Web services composition. In: Services computing conference, 2009. APSCC 2009. IEEE Asia-Pacific, pp 319–324
31. Laranjeiro N, Canelas S, Vieira M, (2008) wsrbench: an on-line tool for robustness benchmarking. In: 2008 IEEE international conference on services computing, SCC 2008. Honolulu, Hawaii, USA
32. W3C, W3C XML Schema (2008). http://www.w3.org/XML/Schema
33. Eviware, soapUI (2007). http://www.soapui.org/
34. Bartolini C, Bertolino A, Marchetti E, Polini A (2009) WS-TAXI: A WSDL-based testing tool for Web services. In: International conference on software testing verification and validation, ICST (2009) Denver. CL, USA
35. Santiago V, Amaral A, Vijaykumar NL, Mattiello-Francisco M, Martins E, Lopes O (2006) A practical approach for automated test case generation using statecharts. COMPSAC 2006
36. de Barros M, Shiau J, Gidewall K, Shang C, Forsmann J, Shi H (2007) Web services wind tunnel: on performance testing large-scale stateful Web services. In: IEEE/IFIP international conference on dependable systems and networks, DSN 2007. Edinburgh, UK
37. Laranjeiro N, Oliveira R, Vieira M (2010) Applying text classification algorithms in Web services robustness testing. In: 29th IEEE international symposium on reliable distributed systems (SRDS (2010) IEEE Computer Society. New Delhi, India
38. Stock A, Williams J, Wichers D (2007) OWASP top 10, OWASP Foundation
39. Stuttard D, Pinto M (2007) The Web application Hacker's handbook: discovering and exploiting security Flaws, Wiley. New York. ISBN- 10:0470170778
40. Web Application Security Consortium, Classes of Attack (2008). http://www.webappsec.org/projects/threat/classes_of_attack.shtml
41. Antunes N, Vieira M, Madeira H (2009) Using Web security scanners to detect vulnerabilities in Web services. In: IEEE/IFIP international conference on dependable systems and networks, DSN 2009, Lisbon, Portugal
42. Laranjeiro N, Vieira M, Madeira H (2010) Web services robustness testing results summary. http://eden.dei.uc.pt/~cnl/papers/2010-tsc-robustness.zip
43. Planet Source Code (2010). http://www.planet-source-code.com/
44. Transaction Processing Performance Council, TPC BenchmarkTM App (Application Server) Standard Specification, Version 1.1 (2005). http://www.tpc.org/tpc_app/
45. Avizienis A (1995) The methodology of N-version programming. In: Lyu MR (ed) Software fault tolerance, Chap 2. Wiley, New York, pp 23–46
46. JBoss, JBoss Application Server Documentation Library. http://labs.jboss.com/portal/jbossas/docs. Accessed 12 June 2012
47. Sebastiani F (2002) Machine learning in automated text categorization. ACM Comput Surv 34:1–47
48. Apache Software Foundation, Maven (2010). http://maven.apache.org
49. Apache Software Foundation, Jakarta Commons Validator. http://jakarta.apache.org/commons/validator/. Accessed 12 June 2012