Improving Web Services Robustness

Nuno Laranjeiro, Marco Vieira, Henrique Madeira
CISUC, Department of Informatics Engineering
University of Coimbra – Portugal
cnl@dei.uc.pt, mvieira@dei.uc.pt, henrique@dei.uc.pt

Abstract

Developing robust web services is a difficult task. Field studies show that a large number of web services are deployed with robustness problems (i.e., presenting unexpected behaviors in the presence of invalid inputs). Several techniques for the identification of robustness problems have been proposed in the past. This paper proposes a mechanism that automatically fixes the problems detected. The approach consists of using robustness testing to detect robustness issues and then mitigate those issues by applying inputs verification based on well-defined parameter domains, including domain dependencies between different parameters. This integrated and fully automatable methodology has been used to improve three different implementations of the TPC-App web services. Results show that this tool can be easily used by developers to improve the robustness of web services implementations.

1. Introduction

Service Oriented Architectures (SOA) are now widely used to support many businesses, linking suppliers and clients in sectors such as banking, transportation, and automotive manufacturing, just to name a few. Web services are the key element of modern SOA and consist of self-describing components that can be used by other software across the web in a platform-independent manner. This technology is supported by standard protocols such as SOAP (Simple Object Access Protocol) for message exchange, WSDL (Web Services Description Language) for interface description, and UDDI (Universal Description, Discovery, and Integration) for service discovery [7].

Web services provide a simple interface between a provider and a consumer, where the first offers a set of services that are used by the second. An important aspect is services composition, which typically consists of a collection of services working together to achieve an objective. This composition is normally a "business process" that describes the sequencing and coordination of calls to the component services.

Software faults (i.e., program defects or bugs) are recognized as the major cause of computer system failures [9], [11]. Interface faults, related to problems in the interaction among software components/modules, are particularly relevant in service-oriented environments. In fact, web services must provide a robust interface to the client applications, even in the presence of invalid inputs, which may occur due to bugs in the client applications, corruptions caused by silent network failures, or even security attacks.

Robustness testing is an effective technique to characterize the behavior of a system in presence of erroneous input conditions. It has been successfully used to assess the robustness of operating systems [16], and recently has been extended to the evaluation of web services [15], [13]. Robustness tests stimulate the system under testing through its interfaces ("black-box" testing) by submitting erroneous input conditions that may trigger internal errors.

Although web services are increasingly being used in complex business-critical systems, current development tools do not provide a practical way to automatically identify and correct robustness flaws in the web services code. This is just the goal of this paper.

The approach integrates automatable workload generation and robustness testing to identify robustness problems. The former is applied to create a set of invocations capable of exercising the web service implementation and the latter to test the behavior of the service in the presence of invalid inputs (including invalid responses from external services) and, thus, identify existing robustness problems.

To eliminate robustness problems we defined a language (designated as Extended Domain Expression Language – EDEL) that enables the inclusion of a detailed domain description into the XML Schema document (XSD) [26] typically referenced by the web services’ WSDL file. The proposed language allows full expression of domains, including complex parameter dependencies. Using EDEL as a starting point, we propose a technique that endows web services with the capability of performing input verification based on EDEL-announced input domains.

Our approach does not require any access to the source code of the application. Instead, we propose a bytecode instrumentation approach that in compile-time is able to weave the necessary modifications into the target service. In our opinion, this integrated tool is of utmost importance to help web services developers...
improving their code and system administrators improving the robustness of services already deployed.

We have applied our approach to three implementations (coded by different developers) of the services specified by the standard TPC-App benchmark [23]. Several problems have been disclosed and fixed, showing the effectiveness of the methodology.

The structure of the paper is as follows. Next section presents some background and related work. Section 3 presents the techniques for fixing robustness problems. The experimental evaluation is presented in Section 4 and Section 5 concludes the paper.

2. Background and related work

In an environment based in web services a provider supplies a set of services for consumers [7]. Web services are supported by an infrastructure that typically includes an application server, the operating system and a set of external systems (e.g., databases, payment gateways, etc) [7]. Robustness testing of web services consists of the generalization of traditional robustness testing [16], [12] (typically applied to the test of robustness of operating systems and microkernels) to the web service environments. A first approach to assess the behavior of web services in the presence of tampered SOAP messages was proposed in [13]. It consists of a set of robustness tests based on invalid call parameters. The services are then classified using an adaptation of the CRASH robustness scale, proposed by Koopman and DeVale for operating system [16]. The work presented in [13] has, however, a few limitations: it does not consider robustness testing in external web services responses and the identification of robustness problems is semi-manual. The present work extends that approach by handling these issues.

A technique to test web services using parameter mutation is presented in [25]. The WSDL is parsed and mutation operators are applied to it resulting in several mutated documents that will be used to test the service. These operators are however somewhat limited.

A key aspect in robustness testing of web services is the generation of representative workloads, which are capable of exercising the web service code in a complete way, especially in the case of random workload generation approaches. Code coverage analyzers, such as Cobertura [10] and Clover [2], can be used to validate and improve workloads through the automatic identification of the areas of a program that are not exercised by a set of test cases, helping in the definition of additional test cases needed to increase coverage.

Several approaches have been proposed to improve the robustness by the use of wrappers. The HEALERS [4] toolkit can be used to enhance both the robustness and security of applications. It protects applications from errors related to C library functions by intercepting calls to the C library. In [17] authors also propose a wrapper-based approach for protecting off-the-shelf components. In [1] robustness wrappers are also provided for off-the-shelf software in order to handle operating systems failures gracefully.

The Service Modeling Language (SML) provides a powerful way for creating models of complex services [3]. These models may include information such as configuration, deployment, service level agreements, etc. SML uses Schematron [5] and XML Schema. Schematron and XPath functions [27] are used for the language’s rules, which are Boolean expressions that constrain the structure and content of documents. SML scenarios require, however, several features that either do not exist or are not fully supported in XML Schema. Our proposal includes a language that conforms to XML Schema and is not as cumbersome as Schematron. In fact it is specifically designed to reuse XML Schema validation aspects as much as possible, hence keeping its footprint to a minimum.

The Bean Validation Java Specification Request (draft) [19], is an effort to specify class level constraint declaration and validation facilities for the Java developer. Our current work introduces dependency constraints and connects validation aspects to the web service technology. Ideally, in the future, a developer will be able to use its favorite programming language to automatically generate EDEL code for domain expression and announcement.

3. Robustness improvement approach

Our proposal for disclosing and removing robustness problems in web services is based on the following phases:

1. Gather required information and prepare the web service
   1.1. Obtain a list of operations, parameters, and data types.
   1.2. Obtain parameters domains.

2. Workload generation and execution
   2.1. Generate a service workload.
   2.2. Run the workload and verify service answers.
   2.3. Measure the workload coverage. Return to 2.1 until an acceptable percentage is achieved.
   2.4. Add domain information about any existing external service calls.

3. Execute a set of robustness tests
   3.1. Identify robustness issues.
   3.2. Return to earlier steps as needed (e.g., when a parameter’s domain must be redefined).
4. Fix disclosed robustness problems and verify the service behavior
   4.1. Generate a protective service wrapper.
   4.2. Re-execute robustness tests (as in Phase 3).
   4.3. Re-execute the workload and verify service answers to identify potential deviations.

3.1. Web service description

The first step of our approach consists of obtaining relevant definitions about the web service. 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 [26], [7].

The second 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 mainly is due to:

- Lack of integrated tools (and programming language support) that could be easily used to add the domain values to the service’s WSDL descriptor.
- Currently WSDL or XSD have no support for expressing dependencies between two (or more) parameters of a given service operation. This absent feature impairs the full definition of a domain and, in what concerns robustness, a partial domain definition is ultimately useless.

To fill the gap that hinders current web service applications from fully expressing and disclosing business logic domains, we propose an extension to XML Schema (XSD) that not only allows service developers to provide information on the valid domains for each parameter, but also provides strong semantics for expressing domain dependencies between multiple parameters. This is particularly important when a client application is using an unknown or undocumented service, but can also be used to prevent robustness problems. An important aspect is that we are using existing XSD features to provide these semantics, hence maintaining retro-compatibility with existing service implementations and supporting stack tools.

Basic domain information is expressed by means of standard XSD restriction elements [26]. In our case, these elements must have an id attribute (it is an optional attribute in XSD) so that they can be referenced when defining parameters inter-dependencies. To state these parameters inter-dependencies, while maintaining retro-compatibility with any XML Schema reader, we make use of the XSD element appinfo, in which we include our extended domain representation. The syntax for this representation must follow a particular language, which we have designated as ‘Extended Domain Expression Language – EDEL’. Figure 1 represents the relation between EDEL and the main web service descriptors – WSDL and XSD.

![Figure 1 – Relation between EDEL, XSD, WSDL.](image)

Figure 2 shows an example of the proposed language syntax. A schema describing the complete language can be found at [14].

```
<dependencies>
  <dependency id="1">
    <function id="f1" name="aggregator" strategy="and">
      <param index="0" name="r1" />
    </function>
  </dependency>
  <dependency id="2">
    <function id="f2" name="starts-with" strategy="or">
      <param index="0" name="r4" />
      <param index="0" name="r5" />
    </function>
  </dependency>
</dependencies>
```

![Figure 2 –EDEL example.](image)

As we can see, the extended domain representation consists of a set of dependencies, each expressing a relation between two or more parameters. Keep in mind that, at this point, parameters already have their individual domains defined in standard XSD restrictions that are identified by a unique id attribute.

Each relation (the function element in the figure) is a function that uses individual restrictions to ultimately produce a Boolean output and is composed of three attributes: an id that uniquely identifies it; a name that indicates the behavior to apply to each restriction argument (each sub-element of the function element); and a strategy that specifies how the named function should be applied. The sub-elements of the function element (param elements) are basically arguments for its parent. Each param element specifies the name of a restriction (by referring the restriction’s id) and, optionally, an index attribute for the cases where a service accepts multiple complex objects of the same type.

For instance, in Figure 2, the aggregator function with the and strategy indicates that a logical AND
must be applied to the values of parameters associated with restrictions $r1$ and $r3$. $r1$ could, for instance, define that some numeric parameter must be greater than zero and $r3$ could define that a string parameter must match a given regular expression. In this two-element case, the strategy attribute has no effect and can be removed (it is and by default). However, the strategy element makes a significant difference in the next dependency ($id$ 2 in the figure). Here, we are defining that at least one of the parameters associated with $r4$ and $r5$ must start with the value obtained at runtime for the parameter associated with $r2$. The first param element in all functions is the one that serves as reference for applying the function through all of the remaining children.

In order to reuse developers’ knowledge we aim to provide a comprehensive set of functions based on the XPath 2.0 and XQuery 1.0 function reference list [27]. In this way, developers do not need to learn another new language when specifying domain dependencies. Virtually any XPath reference function that returns a result is a candidate function that can be used in EDEL. However, the final result of a dependency element must be a Boolean so that our EDEL engine can logically evaluate it. Thus, all non-Boolean results provided by XPath functions that are used to express the domains of a web service must also be used as parameters to functions that can compute a Boolean result.

### 3.2. Workload generation and execution

A workload (set of valid web service calls) is needed to exercise each operation of the web service. As it is not possible to propose a generic workload that fits all web services, we need to generate a workload specifically for the web service under improvement. Two options are available:

- **User defined workload**: the developer implements a workload emulation tool based on the knowledge he/she has about the service (e.g., using a client emulation tools like soapUI [6]).
- **Synthetic workload**: this workload can be automatically generated using the web service definitions mentioned above. For every parameter of each operation a set of valid input values is randomly generated. Those values are adequately combined to guarantee a large number of valid execution calls.

For the generation of a synthetic workload we integrated a set of well-known tools in a workload generation process depicted in Figure 3. Some of these tools are specific to Java, but similar ones exist for all major languages, as explained ahead.

Using the XSD file as starting point, we are able to automatically generate a synthetic workload by using appropriate tools, such as benerator [24] (stage 1). This tool is able to read XML Schema files and, using the domain information present in each schema, it is able to generate a set of XML files containing values that will be used later on to exercise our target service.

![Figure 3 – Workload generation and execution.](image)

In order to use the generated values, we need to create programming language level objects that accurately represent the structures found in the XSD file (stage 2). JAXB’s binding compiler (xjc – https://jaxb.dev.java.net/) can be used for this purpose.

At this point, we are able to use XStream (http://xstream.codehaus.org/) to deserialize the produced XML into the corresponding generated Java objects, creating this way a list of objects that form our final workload (stage 3). This is a process that uses reflection to load classes by name and builds a list of objects that are integrated into one unit test case per each service operation (stage 4).

Most tools like benerator are, up to this date, unable to consider multiple domain relations for the input parameters. In fact, to generate the input values this tool only allows the definition of a single domain restriction. Although this restriction can also be a union of restrictions, inter-parameter restrictions are not taken into account, hence not usable. This way, the generated workload may include invalid web service calls that have to be identified and discarded.

A key difficulty related to the workload generation is that the coverage of the web service calls is not easy to guarantee (e.g., it is extremely difficult, or even impossible, to generate a workload that exercises all the web service code). Our proposal includes executing the workload and using a test coverage analysis tool to get a metric of the code coverage, such as Cobertura [10] (stage 5). If the developer is not satisfied with the coverage then more web service calls are required. Calls must be added to the workload until the code coverage...
reaches the level the developer desires (stage 6).

The next step is to **execute the workload** in order to understand the behavior of the service in the presence of valid input values. The results of this execution will be used later to verify if the robustness improvement code harmed the normal behavior of the service. In addition, these results can be used to identify possible deviations from the expected output domain.

A software defect probably exists when the response of the web service in the presence of legal inputs is outside of the valid output domain that was previously provided by the developer. If a defect is identified then the developer must fix it (this has to be done manually as the correction of these defects is outside the scope of this work). The workload should then be re-executed to try to identify more issues.

As mentioned, composite web services use external web services to execute a given process. The results of the execution of these external components can be seen as inputs for the composite service and are thus a potential source of robustness problems. When we are testing a compound service, the last step of this phase consists of **gathering information on the response domains of the external web services used**. While executing the generated workload, all calls to external services are intercepted and logged. The relevant information on those services responses is afterwards extracted from the WSDL descriptions. If this description already specifies the valid domains (using EDEL) then these valid domains are used. Otherwise, the developer is asked to provide the valid response domain for each component service, which is converted to EDEL and appended to a local copy of the XSD file.

### 3.3. Robustness tests execution

Our proposal for robustness problems identification consists of performing a set of tests on web service inputs, considering both call parameters and external web service responses. The robustness tests are based on combinations of exceptional and acceptable input values that are generated and applied by a fault injection tool. This tool applies a set of predefined rules to the incoming workload objects, according to the data type of each parameter or external web service response. Figure 4 presents a view of the fault injection process including the relation between a simple web service, a client, and the fault injection tool.

In the present work we have used a widely known AOP framework (AspectJ [22]) to create a fault injection tool (represented by the fault injector in the AOP layer in Figure 4) that is able to inject faults into any web service that uses the reference JAX-WS model.

AOP is a programming paradigm that allows injecting crosscutting concerns into any application in a non-intrusive way [8]. Essentially, fault injection capabilities are transparently integrated into the bytecode of the web service during the build process. Fault injection can hence be included or excluded from the process with a simple change in the build file in compile-time, or with the change of a value in a properties file at runtime, if necessary. Note that this technique easily extends to other programming languages and web service stacks, as the concepts used here are quite generic and also present in major languages.

In practice, the test runner invokes the service by cyclically using the generated workload (each workload member is \(w^*\) in Figure 4). The fault injector looks at the EDEL definition (and any possible information about external service invocations) at the beginning of the injection campaign and generates a set of faults to be applied (\(f_1\) to \(f_n\) in Figure 4). Then repeatedly applies them, as explained in the following paragraphs. The test ends when all faults have been applied, or when the developer decides to stop the test.

To **trigger robustness problems in the web service call parameters**, invalid input values are injected during the execution of the workload. This step includes several tests, each one focusing on a given operation of the web service and including a set of slots. Each slot targets a specific parameter of the operation and comprises several injection periods. In each injection period several faults (of a single type) are applied to the parameter under testing. A set of predefined rules (see detailed list in [13]) is applied to automatically generate the faults.

To **trigger robustness problems in the responses of external web services** we again inject invalid input values, but we extend this concept to the injection of exceptions. In this case, we explicitly throw exceptions at particular joinpoints (invocations of external web services) with the goal of exercising any existing error recovery code. The set of possible exceptions to be thrown includes all exceptions declared by the external service (i.e., checked exceptions) and a set of runtime
exceptions (Java’s RuntimeException, and all direct subclasses known to the Java environment). These exceptions are created by reflection and this enables us to maintain the same fault injection code independent of the application being tested.

A detailed log is created during the execution of the robustness tests. Each entry in this log (represented by r* in Figure 4) corresponds to a single execution of a web service operation and includes the input values, the fault injected, and the web service response. Identifying robustness problems consists then in running a tool that using the EDEL description is able to detect responses that violate the valid output domain of the operation, including unexpected exceptions (i.e., any checked exception not declared at the service’s interface or any runtime exception). This tool’s output is a set of invalid responses (if any) and all useful information to debug the problem (injected fault type, input value, parameter name and type, etc.), which is used later to fix detected problems.

3.4. Robustness problems removal

The protective scheme proposed basically consists of validating all domains as expressed by EDEL, prohibiting any execution with invalid values. To provide input validation, we instrument the bytecode of the target service using AOP to transparently inject our crosscutting validation concern. Hence, each arriving request undergoes a complete validation process before it is delivered for business logic processing, as depicted in Figure 5.

![Figure 5 – The input validation process at runtime.](image)

In some cases it may be necessary to fully protect the service. However, for other services (e.g., legacy services that already perform partial input validation), it may be unnecessary to double validate parameters. In this case we use the robustness tests’ results to identify which parameters must be protected. With this information we perform a domain reduction, where all domain information that exclusively refers to non-problematic parameters is removed.

As mentioned before, at runtime all parameters are analyzed and validated before being delivered to the target web service. This also applies to the responses coming from external services invocations in the same manner. In other words, besides instrumenting the main service, we also instrument any existing external service call to perform domain validation over the received responses. Additionally, any exception thrown by the external service is caught and analyzed. If it is one of the declared exceptions, then it is re-thrown. For all other cases, it is wrapped in our custom exception (which exists both in checked and unchecked versions — ValidationException and ValidationRuntimeException, respectively). From our service point of view these responses are nothing more than input data.

The next step consists of repeating the robustness tests to check if the robustness problems identified before were fixed and if new problems appeared (fixing a problem may disclose another). Phase 3 should be repeated if new robustness problems are identified.

Automatically adding functionality to a web service is obviously a risky operation. This way, to confirm that the added functionality did not modify the behavior of the service and only corrected robustness problems, we need to re-execute the workload, and check the results execution for responses outside the expected domains, or that differ from the original workload execution step (depending on the web service implementation). If problems are identified then the developer should review the web service definitions.

4. Experimental evaluation

Here we present and discuss the experimental evaluation performed over an initial prototype tool (available at [14])) that was created to demonstrate the feasibility of the proposed approach. All implementation efforts for the experimental evaluation used Java.

4.1. Experiments preparation

To demonstrate our approach we have used the TPC-App benchmark [23]. TPC-App is a performance benchmark for web services and application servers widely accepted as representative of real environments. A subset of the services specified by this benchmark (Change Payment Method, New Customer, New Product, and Product Detail) was used to reduce the effort of performing the experiments. To verify if the approach is applicable in multiple scenarios, we created 3 different versions of each service (versions A, B, and C). These versions were implemented by distinct developers with more than 2 years of experience.

Regarding the container for deploying the TPC-App services, we decided to use JBoss 4.2.2.GA and the reference implementation for the Java API for XML Web Services (JAX-WS) due to their relevance in in-
duty [18], [20]. The experimental setup consisted of two major nodes (client and server) that were deployed on two machines on an isolated Fast Ethernet network.

The WSDL and XML schema (XSD) of each web service were analyzed and, for each input and output parameter, we manually extended the XSD file to include domain restrictions using the standard XSD restriction element and fully respecting the TPC-App specification. EDEL was applied to express the final domains. The full schemas can be found at [14].

For our study we manually defined a workload based on a set of web service requests (a total of 5 requests for the 4 web services). However, as proposed in Section 3.1, this task can be fully automated with the help of an EDEL-aware workload generation tool. Before proceeding to the tests we analyzed the coverage of the workload using Cobertura [10]. As we can see in Table 1, the coverage is in general above 80% (except in one case), a value defined as acceptable by the developers. Thus, we decided to analyze the source code of all versions to understand what code was not being covered. In all cases it corresponded to unused exception catch blocks. This simple workload was able to exercise the useful source code perfectly, but was not expected to trigger any error-handling blocks. Thus, we considered the workload adequate for all services.

The workload was then run and the web services responses analyzed in order to identify deviations from the expected output domain that could indicate software defects, which was not the case.

4.2. Identifying and fixing robustness problems

Table 2 presents a summary of the identified robustness problems in each version (A, B, and C) of the tested TPC-App web services. Each robustness problem represents either a regular result that falls outside of the initially specified valid response domain, or an exceptional result that is not one of the declared web service exceptions.

Results show a lot of diversity, with implementation C presenting no robustness issues at all. In fact, a later analysis of this version’s source code confirmed that this that the programmer took into account expectable erroneous input data, and for all tested cases a check was in place to see if the incoming data was valid.

Version A presented problems in 3 of the services. The vast majority of the observed issues manifested as unexpected exceptions due to erroneous construction of SQL commands (78% of the total issues observed in this version). This is explained by the fact that this service directly uses client input to dynamically build SQL statements. The remaining issues included null responses (10%); runtime exceptions (10%); and null pointer exceptions (2%). The great number of issues uncovered in the NewCustomer service is related to the fact that this service has a large number of parameters, and no appropriate input validation was in place.

Version B presented a total number of issues lower than version A. The reason was that the services included extra error preventing code that did not exist in version A. The issues observed in B ranged from uncategorized SQL exceptions (43% of the total cases for this version) to data integrity violations (16%), runtime illegal argument exceptions (16%), bad SQL grammar (9%), null pointers (9%), and responses falling out of the expected domain (7%). We did not disclose any robustness issue related to external service invocations.

Based on the robustness results we then deployed three robust versions of the diverse benchmark implementations. This consisted in applying our robustness improvement approach, which uses AOP and the EDEL XSD extension to automatically protect each application against invalid inputs. The robustness tests were then repeated and no robustness issues were detected. This is an extremely important result as it demonstrates the strength of our approach in protecting web service application against invalid inputs, hence avoiding potentially serious robustness problems. For instance, while performing robustness testing on the NewCustomer service, a fault applied to an identifier field of the service’s input data triggered a data integrity violation exception. Note that it is unacceptable that a particular operation is able to trigger such exception, and in that sense our validation mechanism stopped any possible damage ahead of time.

4.3. Web services behavior verification

To verify if the robustness improvement mechanisms changed the web services’ behavior we re-ran the workload for all three versions. The web services responses were automatically analyzed by our tool to identify potential deviations from the valid output domains. As expected, no problem was identified (we also double-checked the results manually), providing a

<table>
<thead>
<tr>
<th>Web Service</th>
<th>Code Coverage</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>ChangePaymentMethod</td>
<td>92%</td>
<td>87%</td>
<td>91%</td>
<td></td>
</tr>
<tr>
<td>NewCustomer</td>
<td>80%</td>
<td>92%</td>
<td>93%</td>
<td></td>
</tr>
<tr>
<td>NewProducts</td>
<td>74%</td>
<td>80%</td>
<td>87%</td>
<td></td>
</tr>
<tr>
<td>ProductDetail</td>
<td>94%</td>
<td>80%</td>
<td>87%</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Web Service</th>
<th>Robustness Problems</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>ChangePaymentMethod</td>
<td>14</td>
<td>13</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>NewCustomer</td>
<td>44</td>
<td>26</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>NewProducts</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ProductDetail</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 – Workload coverage.

Table 2 – Identified robustness problems.
strong indicator that our framework did not change the application's normal behavior. For many cases this run's output must be exactly equal to the output of the first workload run. However, for services that do not deliver equal answers for distinct invocations (with the same input), the framework's only option is to check if each response fits the specified domain or not.

Finally, we executed a test to assess the performance impact related to the robustness improvement execution. We found that the best executing time was of 7,338 ms (± 0.746), being the worst execution time of 16,093 ms (± 0.961) for a total of 10000 executions for each service. To obtain these measurements we used a Java method that provides nanosecond precision (but does not guarantee nanosecond accuracy). We initially ran baseline performance tests for each service (30 minutes each test) and were able to verify that these values represent 6% and 48% of the average execution time of the original unmodified services.

5. Conclusion

This paper presents an approach for improving the robustness of web service applications. Previous work shows that robustness issues are extremely relevant in web services, and our solution closes the existing gaps by providing a low-effort integrated view of full domain expression, data generation and validation, protection integration, and application testing.

The proposed approach was applied to three different TPC-App implementations. Multiple robustness problems were disclosed and corrected, showing that the approach has a low performance impact, is quite effective, and is an extremely powerful instrument.

Our future goal is to provide a fully automated end-to-end tool, which, based on EDEL-defined input domains, is able to generate the adequate workload. Additionally, we are planning to develop a version of our robustness improvement tool that integrates with building tools to facilitate the development of service-based applications, contributing for more robust services.

6. References