Information Sciences 181 (2011) 600–619

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Improving data perturbation testing techniques for Web services

Ana C.V. de Melo *, Paulo Silveira

Department of Computer Science, University of São Paulo (USP), Rua do Matão, 1010, Cidade Universitária, 05508 090, São Paulo – SP, Brazil

ARTICLE INFO

Article history:
Received 23 November 2009
Received in revised form 24 September 2010
Accepted 28 September 2010

Keywords:
Data perturbation
Software testing
Web services

ABSTRACT

The widespread use of service-oriented architectures (SOAs) and Web services in commercial software requires the adoption of development techniques to ensure the quality of Web services. Testing techniques and tools concern quality and play a critical role in accomplishing quality of SOA based systems. Existing techniques and tools for traditional systems are not appropriate to these new systems, making the development of Web services testing techniques and tools required. This article presents new testing techniques to automatically generate a set of test cases and data for Web services. The techniques presented here explore data perturbation of Web services messages upon data types, integrity and consistency. To support these techniques, a tool (*GenAutoWS*) was developed and applied to real problems.

1. Introduction

Most organizations today rely on information systems as part of their business process. The need to exchange data between different applications requires them to be more flexible and interoperable. Web services emerged to support such requirements: services can communicate with each other by passing data from one service to another or by coordinating an activity between two or more services.

Due to the fact that service-oriented architecture (SOA) and Web services are used in heterogeneous contexts, Web services are required to satisfy high quality standards, and automated test tools are necessary to help improve such a quality from a practical point-of-view. However, SOA based systems differ from traditional ones and the existing traditional testing techniques and tools are no longer appropriate to test them. The widespread use of Web services and the need of ensuring quality of services have driven the academic community to research testing techniques and tools devoted to Web services.

The testing strategies applied to Web services depend on the test level addressed and testers perspectives [9,10]: service developer, service provider, service integrator, third-party certifier and end-user. Unit and integration testing can be applied using functional and non-functional techniques adapted from components techniques. For the unit testing, some adaptations are necessary to overcome the Web services features [40,10]:

- *Abnormal behaviors:* since hypertext transfer protocol (HTTP) is stateless, the system is responsible for tracking the services transactions. The Web services transactions may stop in the middle of the operation, not finishing the entire transaction. The system must treat such abnormal behaviors and the testing strategies must ensure they are treated accordingly.
- *Unexpected usages:* in principle, Web services can be used by requesters in different ways. The testing strategies must cover a variety of scenarios which might not be available at the design and implementation time, making the test activity even harder.

* Corresponding author. Tel.: +55 1130919689; fax: +55 1130916134.
E-mail addresses: acvm@ime.usp.br (A.C.V. de Melo), psilveirap@gmail.com (P. Silveira).

0020-0255/$ - see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2010.09.030

- *Incomplete systems:* certain services rely on other services to run. Providing a unit testing to cover the behavior of such dependent service can be very cost and time consuming, making the adequate test to be delayed until the integration testing. This situation imposes some extra management to decide when actually test the services in order to provide the required quality.

Besides the extra care on the unit testing, the integration testing is affected by the particular features of Web services [40,10]:

- Source code is not available if services are provided by a third-party. Tests for statically bound services can be written based on the standards and published descriptions. This could, however, be very costly and error-prone since generating services stubs becomes very hard with no access to code. On the other hand, for the dynamically bound services, testing becomes very hard if the services functionalities are not well-specified by providers. Overall, having no access to the internal behavior of services prevents an adequate combination of the black with the white-box techniques. To overcome these problems, the internal service behavior must be exposed by providers, despite not being adopted, in general, due to confidentiality and commercial issues.
- In general, testing a Web service does not cover the various combinations of possible applications. For instance, the situations in which many requesters access the service concurrently. The real situations are hard to predict and mechanisms to control test executions must be defined by services providers. However, no standards for these mechanisms have been established so far by the World Wide Web Consortium (W3C), making them not available in practice.

Due to these features and the widespread use of Web services in industrial software, testing Web services has recently received more attention [31]. Canfora and Di Penta [9] summarized the main challenges of testing service-oriented systems and surveyed some of the testing techniques for the service-oriented architectures [10]. Huang et al. [19] pointed out two major approaches to address the Web services testing problem: automatic testing and model checking. New techniques have been proposed undertaking these approaches since then. Some of these researches provide a general framework for testing, focusing on testing process and management, while others are devoted to developing techniques to address Web services testing at different description levels.

Zhu [40] presented a framework for testing Web services using *testing services*, based on a service-oriented testing ontology [39], and pointed out the main differences between testing traditional and Web services oriented systems (summarized above). Tsai et al.[31] proposed a hierarchical testing framework to generate test scenarios based on Web Service Description Language (WSDL) specifications, together with some WSDL improvements [29]. Also at the WSDL level, Bai et al. [1] proposed a framework to automatically generate Web services test cases from the Web services description, which embeds the basic information of a service (interface operations and the data transmitted). Based on the service WSDL information, test data are generated for simple, aggregate and user-defined types. Operation sequences to be tested are also generated based on operation dependencies from the service description. Bartolini et al. [4] developed a tool, WS-TAXI, to automatically generate test suite using TAXI [7,6], based on the category partition strategy to generate functional tests from XML schema.

For the integration testing, Tsai et al. [30] described a testing framework, Coyote, that comprises a test master and a test engine. The test master allows testers specifying test scenarios and cases, and performing a set of analysis such as services dependency, completeness and consistency. The test engine interacts with Web services providing traces information. Following a formal approach, Huang et al. [19] presented a model checking process for Web Ontology Language for Web Services (OWL-S) in which the model checker BLAST [17] is extended to cope with concurrency in OWL-S. Some OWL-S extensions were also proposed.

Regarding the automation of non-functional testing, Offutt and Xu [25] presented a Web services testing technique based on data perturbation. Existing Extensible Markup Language (XML) messages are modified based on message grammars rules and data perturbation on values and interactions. The set of these modified messages are then used as test suites. This article extends the data perturbation testing technique by Offutt and Xu by adding mutation operators, boundary values considering XML Schema *Facets*, test cases using relationships defined in the WSDL, Universal Description, Discovery and Integration (UDDI), internal database to collect and use values previously captured from messages. As a proof-of-concept, a tool was developed, *GenAutoWS*, embedding the previous and the new techniques presented here.

The forthcoming sections present: some fundamental concepts on Web services and SOA; the new testing techniques based on data perturbation complementary to existing techniques; some experimental results regarding the new techniques; and, finally, some concluding remarks on the presented techniques.

2. Web services and SOA preliminaries

Service-oriented architecture (SOA) is essentially an architectural style to allow a collection of loosely coupled software agents interacting with each other [16]. The most common way of implementing SOA is by the use of Web services.

There are today many definitions for Web services. According to W3C [35], Web services are software systems designed to support machine-to-machine interaction over a network via well-defined interfaces. A Web service is specified in a standard way by a *service descriptor* using a service description language, Web Service Description Language (WSDL, [34]) for example.

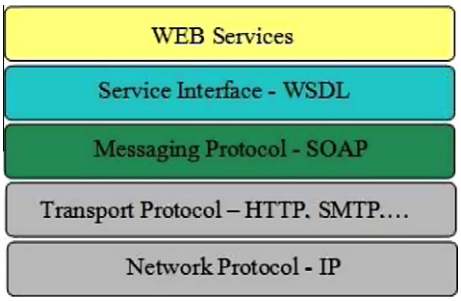

Fig. 1. Web Services Structure.

	Document	RPC
Literal	<soap:Body> <car xmlns="http://..."> <model...> <year...> </car> </soap:Body>	<soap:Body> <purchase> <car xmlns="http://..."> <model...> <year...> </car> </purchase> </soap:Body>
Encoded	<soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> <car> <model xsi:type="soapenc:string"...> <year xsi:type="soapenc:int"...> </car> </soap:Body>	<soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"> <purchase> <car> <model xsi:type="soapenc:string"...> <year xsi:type="soapenc:int"...> </car> </purchase> </soap:Body>

Fig. 2. Messages type and codification styles.

Each service descriptor must contain all the information needed to make the service interaction possible, including message format, transportation protocol and binding information. Fig. 1 shows Web services architecture.

Web services can interact with other systems, in the way described by the service descriptors, using Simple Object Access Protocol (SOAP) to receive and send information. SOAP exchanges XML-based messages over another application layer protocol, usually Hypertext Transfer Protocol (HTTP) or Multipurpose Internet Mail Extensions (MIME). Those messages can differ in type and style. The two most common messages types are Remote Procedure Call (RPC) and *Document*. The RPC messages wrap program methods into the message, allowing them to be remotely invoked. The body and all parameters are sub-elements. By contrast, in the Document type, the message content is placed directly into the body element, making Document-based Web services loosely coupled and document driven.

Two message styles are used: *Encoded* or *Literal*. They define how data will be transmitted. In the *Encoded* style, apart from the WSDL service descriptors, there is an additional set of rules that specifies both: how the original data is encoded/serialized to XML and how the XML is decoded/de-serialized back to the original data. The attribute `encodingStyle` identifies this set of rules. The most common encoding style is the *SOAP Data Model*.[1] In the *Literal* style, data is serialized based on a schema, usually *W3C XML Schema*, and no rules are predefined for serializing objects or structures.

Message types, *RPC* and *Document*, can be combined with *Encoded* or *Literal* styles. Fig. 2 illustrates those combinations, although RPC/Literal and Document/Encoded are rarely adopted in practice.

A simple example of a Web service message, using *Document* type, for a *Movies Rental Store* is shown in Listing 1. In this example, the `drivingLicense` identifies the customer and the message contains a list of movies, each one with `id`, `media` and `price` elements.

SOAP messages depend on XML standards, such as *XML Schemas* and *XML Namespaces*. XML Schemas are used to describe messages exchanged between Web services. As such, Schemas define content, structure and semantics of XML documents that can be shared between applications. The datatypes defined in XML Schemas are of types: Simple or Complex elements. The Simple types are, in turn, defined over built-in XML primitive (e.g. `string`, `boolean` and `decimal`) and derived (e.g. `name`, and `integer`) datatypes. Besides that, some constraints on the range of datatypes can be applied using *Facets*. Table 1 shows the XML *Facets* available to define restrictions on datatypes.

[1] The *SOAP Data Model* – http://schemas.xmlsoap.org/soap/encoding.

```
<id>
    <drivingLicense>S1234 -123456-12</drivingLicense>
</id>
     <moviesList>
          <movie>
               <id>12</id>
               <media>DVD</media>
               <price>3.25</price>
          </movie>
          <movie>
               <id>130</id>
               <media>DVD</media>
               <price>3.25</price>
          </movie>
     </moviesList>
...
```

Listing 1. XML document-Movies Rental store.

Table 1
XML *Facets*.

Facet	Description
`enumeration`	A list of acceptable values
`fractionDigits`	The maximum number of decimal places allowed
`length`	The exact number of characters or list items
`maxExclusive`	The upper bounds values – less than this value
`maxInclusive`	The upper bounds values – less than or equal to this value
`maxLength`	The maximum number of characters or list items
`minExclusive`	The lower bounds values – greater than this value
`minInclusive`	The lower bounds values – greater than or equal to this value
`minLength`	The minimum number of characters or list items
`pattern`	The exact sequence of characters that are acceptable
`totalDigits`	The exact number of digits allowed
`whiteSpace`	Specifies how white space is handled

Table 2
XML indicators.

Indicator	Description
Order	
`all`	All sub-elements must appear – in any order
`choice`	Only one of the sub-elements can appear
`sequence`	All sub-elements must appear in an specific order
Occurrence	
`maxOccurs`	The maximum number of times an element can appear
`minOccurs`	The minimum number of times an element must appear
Group	
`name`	Gives a name to a group of elements to be further referred
`attributeGroup`	Defines an attribute to a group of elements to be further referred

The XML Schemas can also define how the XML elements are to be used in documents with indicators. These indicators can define: the order in which sub-elements (child elements) must appear, the *order indicators*; how often elements can be used, the *occurrence indicators*; and how sets of elements are related, the *group indicators*. Table 2 summarizes the XML indicators used.

Listing 2 shows an XML Schema for the message in Listing 1. In this XML Schema, we can see *Facets* applied to constrain the elements: `drivingLicense`, `media` and `price`. Besides that, some indicators are used: the `choice` indicator on the element `id`; the `sequence` indicator on `movieRental`, `moviesList` and `movie` elements; and the occurrence indicators `minOccurs` and `maxOccurs` restricting the number of movies in this Web service call.

```
version="1.0" encoding="UTF-8"?> <xs:schema
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
 <xs:element name="movieRental">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="id">
        <xs:complexType>
          <xs:choice>
            <xs:element name="drivingLicense">
              <xs:simpleType>
                <xs:restriction base="xs:string">
                 <xs:pattern value="[A-Z
                     ][0-9]{4}-[0-9]{6}-[0-9]{2}"/>
                </xs:restriction>
              </xs:simpleType>
            </xs:element>
            <xs:element name="memberNumber" type="xs:decimal"/>
          </xs:choice>
        </xs:complexType>
      </xs:element>
      <xs:element name="moviesList">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="movie" minOccurs="1" maxOccurs="5">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="id" type="xs:int"/>
                  <xs:element name="media">
                    <xs:simpleType>
                     <xs:restriction base="xs:string">
                       <xs:enumeration value="BLURAY"/>
                       <xs:enumeration value="DVD"/>
                       <xs:enumeration value="VHS"/>
                     </xs:restriction>
                    </xs:simpleType>
                  </xs:element>
                  <xs:element name="price">
                   <xs:simpleType>
                    <xs:restriction base="xs:decimal">
                      <xs:fractionDigits value="2"/>
                      <xs:maxInclusive value="10"/>
                      <xs:minInclusive value="1"/>
                    </xs:restriction>
                   </xs:simpleType>
                  </xs:element>
                </xs:sequence>
              </xs:complexType>
            </xs:element>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Listing 2. XML Schema for the Movies Rental Store.

The Universal Description, Discovery and Integration (UDDI) [33] specification is used to catalog the Web services. The implementation of this specification is called UDDI registry, representing data and metadata on Web services. UDDI registry includes a set of Web services to allow services to be published and found [18,14].

3. Testing Web services

As with traditional systems, Web services must be tested at the unit and integration levels. Since they are used in very distributed and heterogenous contexts, they require dynamic integration testing. Web services applications interact in three different ways: *publishing*, the service provider makes a service interface available to other services; *finding*, other services (requesters) must be able to discover the service interface; *binding*, addresses the ability to connect and invoke services. Then, the communication aspects [25,31] of Web services must be tested accordingly: discovering, publishing and finding services, and checking the data format exchanged and the request/response mechanisms.

Testing SOAP messages addresses request/response mechanisms and data format aspects of Web services. WSDL is used to expose interfaces as services available on the Internet. Testing WSDL files can be used to generate test plans to validate services. Testing UDDI registries provides the capabilities of publishing, finding and binding of SOA, giving the way software is integrated. The present work focus mainly on data perturbation testing techniques for SOAP messages.

Data perturbation testing technique consists of changing (perturbing) existing data to create new test sets. To provide testing techniques for Web services, the different types of messages must be considered: *RPC* or *data communications* (Document-based). Offutt and Xu [25] presented a data perturbation technique based on data value and interaction perturbations for both *RPC* and *data communications*.

Data value perturbation modifies values in SOAP messages according to their datatypes while the interaction perturbation may consider the data values and data relationships. For most Document-based data communication messages, the XML Schema is available and testing focuses on data use, format and relationships. For RPC messages, however, testing is confined to data uses. This research presents new testing data perturbation techniques based on [25,15]. The extensions to the previous works aim to increase the test coverage, creating new messages with information not explored in the original works:

1. The boundary analysis is enlarged with values immediately above and below the datatype domain, as defined by Pressman [26] and Myers [21].
2. XML *Facets* are also considered in the boundary analysis.
3. New relationship rules are added to data communication perturbation, including the sequence indicators `choice` and `all`, the occurrence indicator `minOccurs` and the `any` element.
4. New mutation operators are defined for Document-based messages.

Invalid test cases are considered for both data value and interaction perturbation. This means that the Web service should return an error when test suites corresponding to these test cases are executed. Sections 4–6 present the new techniques based on data value perturbation, data communication perturbation and data mutation respectively.

4. Data value perturbation

Data value perturbation modifies values using datatype information based on the boundary value testing approach [5]. To implement it, a set of rules for XML datatypes, corresponding to the primitive types in most programming languages, were created. Table 3 shows the datatypes with the corresponding data value perturbations to be applied presented in [25] (all 19 primitive datatypes are defined by the authors, but only a subset of them is shown). Then, for a given test data, new ones are created based on the boundary values.

Web services using *Literal* messages can be defined by XML Schema and the legal values for each datatype can be constrained using XML Schema *Facets*. They are used to define and validate data upon constraints on datatypes value space. We improved the testing technique on data value perturbation to comprise invalid values for all datatypes and valid and invalid values for the XML *Facets*. Here, the *Facets* addressed and the corresponding test cases are presented:

`pattern`: defines the valid content for a datatype, specified by a regular expression. We use pattern expressions to generated valid and invalid messages. For the "`drivingLicense`" type definition shown in Listing 2, the new messages in Table 4 correspond to the test suite generated for the Pattern test cases:

Table 3
Data value perturbation.

Datatype	Boundary values
String	Maximum length, minimum length, upper case, lower case
Numeric	Maximum value, minimum value, zero
Boolean	True, false
...	...

Table 4
Test suites for pattern in Listing 2.

<drivingLicense>Z9999-999999-99</drivingLicense>	Valid
<drivingLicense>A0000-000000-00</drivingLicense>	Valid
<drivingLicense>9ZZZZ-ZZZZZZ-ZZ</drivingLicense>	Invalid

Table 5
Test suites for enumeration.

<media>DVD</media>	Valid
<media>VHS</media>	Valid
<media>ZZZZZZZZ</media>	Invalid

Table 6
Data value perturbation.

	Data perturbation		Technique	
	Boundary	Invalid values	Off & Xu	New techniques
Primitive datatypes				
String	•		√	⊕
		•		√
Numeric	•		√	⊕
		•		√
Boolean	•		√	⊕
		•		√
Datatypes restrictions – Facets				
totalDigits	•	•		√
maxInclusive	•	•		√
minInclusive	•	•		√
maxExclusive	•	•		√
minExlusive	•	•		√
maxLength	•	•		√
minLength	•	•		√
Pattern	•	•		√
Enumeration	•	•		√
fractionDigits	•	•		√
Length	•	•		√
WhiteSpace	•	•		√

`enumeration`: Constrains the valid values of a data type to a specified set. A message is generated for each value in the given enumeration set. An invalid message is also generated with a value out of this set. For the "`media`" type definition presented in Listing 2, for example, the technique generates the messages showed in Table 5:

`fractionDigits`: Specifies the maximum number of decimal digits that are allowed in the fractional part of the value. The value must be equal or greater than zero. Three messages are generated: one with the maximum number of digits, the second with one digit and an invalid message with oversized fractional digits.

`length`: Specifies the number of characters or list items that are allowed. Valid and invalid messages are generated. The invalid message is generated by adding an extra character to a valid message.

`totalDigits`: Defines the maximum number of values by only allowing numbers to be expressed as $i \times 10^{-n}$, where i and n are integers such that $|i| < 10^{totalDigits}$ and $0 \leqslant n \leqslant totalDigits$. For example, if $totalDigits = 4$, value 55.51 is valid because it can be expressed as 5551×10^{-2}, $i = 5551$ and $n = 2$. A valid message is generated using the maximum number of digits allowed and an invalid message is created using a value over this maximum value. An extra message is generated with fractional digits if the *FractionDigits* is also specified for this element.

`whiteSpace`: Specifies how spaces, line feeds, tabs, and carriage returns will be handled. Depending on the `whiteSpace` value (preserve, replace, collapse), messages are generated including line feeds, tabs and carriage returns.

For all other datatype *Facets* ("`maxInclusive`", "`minInclusive`", "`maxExclusive`", "`minExclusive`", "`maxLength`", "`minLength`"), boundary data value perturbation is applied to generate the test cases. Besides that, values immediately above and below the value defined by the *Facets* are applied, generating invalid messages. For example, if `maxInclusive` is `10`, a message containing the invalid value `11` is generated.

Table 6 summarizes the data value perturbation applied by the original work and by the new techniques developed: √ represents the techniques developed by each work, while ⊕ represents what has been included from other techniques (these symbols are used in the forthcoming tables with the same meaning). In this table, the term *Boundary* represents either the

Table 7
Test data derived from data value perturbation.

Seed	Test data input	Test case
<drivingLicense>S1234-123456-12</drivingLicense>	<drivingLicense>AAAAAAAAAAAAAAAAAAA</drivingLicense>	String maximum length
	<drivingLicense></drivingLicense>	String minimum length
	<drivingLicense>A0000-000000-00</drivingLicense>	Pattern
	<drivingLicense>Z9999-999999-99</drivingLicense>	Pattern
	<drivingLicense>9ZZZZ-ZZZZZZ-ZZ</drivingLicense>	Pattern – invalid value
<id>45879</id>	<id>2^{32}-1</id>	Maximum value
	<id>-2^{32}</id>	Minimum value
	<id>0</id>	Zero
<media>Blu-ray</media>	<media>AAAAAAAAAAAAAAAAAAA</media>	String maximum length
	<media></media>	String minimum length
	<media>BLU-RAY</media>	Upper case
	<media>blu-ray</media>	Lower case
	<media>DVD</media>	Enumeration
	<media>VHS</media>	Enumeration
	<media>ZZZZZZZZ</media>	Invalid enumeration
<price>3</price>	<price>10</price>	Maximum value
	<price>10</price>	Maximum value
	<price>11</price>	Above maximum value
	<price>9</price>	Below maximum value
	<price>1</price>	Minimum value
	<price>1</price>	Minimum value
	<price>2</price>	Above minimum value
	<price>0</price>	Zero and below minimum value
	<price>3.99</price>	FractionDigit
	<price>3.999</price>	FractionDigit exceeded

boundary data perturbation over primitive datatypes, as presented by Offutt and Xu, or the ones on *Facets* as presented above.

Since data perturbation on constrained datatypes are created with the new techniques, a set of new test cases can be applied generating new test data. Table 7 shows a set of test data input derived from test cases either defined by the previous or by the new techniques for the example in Listing 2. The shaded rows show data generated from test cases exclusively defined by the new techniques while the white rows represent the ones from previous works. Some identical data are generated from both techniques independently, such as <price>10</price>, however most of them are complementary. Due to this, both sets of techniques, the previous and the new ones, have been implemented into the tool *GenAutoWS* (Section 7).

5. Data communication perturbation

In Document-based Web services, service consumer and provider interact using complete documents. These documents are typically XML files, defined in a common way, agreed upon schema. Data communications perturbations (DCP) aim at testing Document-based Web service and focus on testing relationship and constraints over message data (defined by the XML Schemas Indicators). To precisely define these, XML Schemas are defined using Regular Tree Grammars (RTG) [11], a formal model for XML schemas.

Definition 1. A regular tree grammar is a 6-tuple $\langle E, D, N, A, P, n_s \rangle$, where:

1. E is a finite set of element types.
2. D is a finite set of datatypes.
3. N is a finite set of non-terminals.
4. A is a finite set of attribute types.
5. P is a finite set of production rules with two forms:
   (a) $n \to a\langle d \rangle$, where n is non-terminal in N; a is either an attribute type in A or an element type in E, and d is a datatype in D.
   (b) $n \to e\langle r \rangle$, where n is non-terminal in N; e is an element in E, and r is a regular expression made up of non-terminals.
6. n_s is the starting non-terminal, $n_s \in N$.

The relationships and constraints are the finite set of production rules P as in Definition 1 (5a) and (5b), respectively.

Based on the `maxOccurs` indicator of XML schemas (see Listing 2, for example), the parent–child associations are acquired and a regular expression for the relationship is created. In the regular expressions, operators '?', '+', and '*' denote

Table 8
Regular expressions used to represent relationships in RTGs.

XML Schema indicator	Regexp	Description
minOccurs, maxOccurs	$\{x,y\}$	At least x and no more than y times
choice	|	One child element or another can occur
all	$\{x_1,\ldots,x_n\}$	The child elements can appear in any order but each must occur only once
Any element	.	Element not specified in the XML Schema

Table 9
Data communication perturbation.

Regular expression $n \to e\langle r\rangle$	Test case	Off & Xu	New techniques
α? in r	α	√	⊕
	Empty	√	⊕
α+ in r	α	√	⊕
	α Instances	√	⊕
	No instances		√
α^* in r	$\alpha^*\alpha$	√	⊕
	$\alpha^{*^{-1}}$	√	⊕
	Deleting all instances of α		√
	k Instances of α		√
'.' in r	β – Arbitrary		√
$\alpha\{x,y\}$ in r	x Instances of α		√
	y Instances of α		√
$\{x_1,\ldots,x_n\}$ in r	Random permutation of $\{x_1,\ldots,x_n\}$		√
	$\{x_1,\ldots,x_{n-1}\}$		√
$x_1\|\ldots\|x_n$ in r	x_i Such that $1 \leqslant i \leqslant n$		√
	All n elements		√

zero-or-one, at least one, and any number of element occurrences, respectively. These operators denote the cardinality constraints in an XML Schema. For these relationships, some testing strategies were defined [25]:

- Given a relationship $n \to e\langle r\rangle$, if there is an expression α? in r, there will be two test cases: one contains one α instance and the other contains an empty instance.
- Given a relationship $n \to e\langle r\rangle$, if there is an expression α+ in r, there will be two test cases: one contains one α instance and the other one contains an allowable number of α instances.
- Given a relationship $n \to e\langle r\rangle$, if there is an expression α^* in r, there will be two test cases. One contains $\alpha^*\alpha$ and the other contains $\alpha^* - 1$, where $\alpha^*\alpha$ duplicates one element instance and $\alpha^* - 1$ deletes one element instance.

Apart from the testing strategies over `maxOccurs`, we extended the technique for testing data integrity and consistency with the use of the occurrence indicator `minOccurs`, and the order indicators: `all` and `choice`, and the element `any`. Table 8 describes each of the XML Schema indicator used and the corresponding regular expression.

The following testing strategies are part of the new technique:

- Given a relationship $n \to e\langle r\rangle$, if there is an expression α+ in r, there will be one extra test case that contains no instances of α. This test case must result an error when executed.
- Given a relationship $n \to e\langle r\rangle$, if there is an expression α^* in r, there will be two extra test cases. One deleting all instances of α, and the other one containing k instances of α, where k is a predefined number representing *unbounded*.
- Given a relationship $n \to e\langle r\rangle$, if there is an expression containing '.' in r, there will be one test case. It contains one instance of β, where β represents any element.
- Given a relationship $n \to e\langle r\rangle$, if there is an expression $\alpha\{x,y\}$ in r, there will be two test cases. One contains x instances of α and the other one contains y instances of α. If y has the value *unbounded*, y will have the value of k, where k is a predefined number.
- Given a relationship $n \to e\langle r\rangle$, if there is an expression $\{x_1,\ldots,x_n\}$ in r, there will be two test cases. One contains a random permutation of $\{x_1,\ldots,x_n\}$, and the other one contains $\{x_1,\ldots,x_{n-1}\}$.
- Given a relationship $n \to e\langle r\rangle$, if there is an expression $x_1|\ldots|x_n$ in r, there will be $n + 1$ different test cases. The first n test case contains x_i where i is an integer and $1 \leqslant i \leqslant n$. The other test case contains all n elements (an error is expected when executed).

The differences between the test cases generated by the Offutt and Xu techniques and the new technique are summarized in Table 9. Besides these differences base on the regular expressions, Offutt and Xu only applied their techniques to the `maxOccurs`, while we applied all to `minOccurs`, `all` and `choice` indicators and the `any` element.

```
...
<id>
    <memberNumber>1234</memberNumber>
</id>
...
```

Listing 3. Test data for choice –2nd element.

```
...
<id>
 <drivingLicense>S1234-123456-12</drivingLicense>
 <memberNumber>1234</memberNumber>
</id>
...
```

Listing 4. Test data for all choice elements -An error is expected.

```
...
<id>
  <memberNumber>S1234-123456-12</memberNumber>
</id>
<moviesList>
    <movie>
        <id>12</id>
        <media>DVD</media>
        <price>3.25</price>
    </movie>
    <movie>
        <id>130</id>
        <media>DVD</media>
        <price>3.25</price>
    </movie>
    <movie>
        <id>12</id>
        <media>DVD</media>
        <price>3.25</price>
    </movie>
    <movie>
        <id>12</id>
        <media>DVD</media>
        <price>3.25</price>
    </movie>
    <movie>
        <id>12</id>
        <media>DVD</media>
        <price>3.25</price>
    </movie>
</moviesList>
```

Listing 5. Test data for the maximum number allowed for sequence's relationships.

The RTG for the XML Schema in the Listing 2 contains two relationships:

$n_{id} \rightarrow id\langle n_{drivingLicense} \mid n_{memberNumber}\rangle$
$n_{movieList} \rightarrow movie\langle(n_{movieId}, n_{media}, n_{price})\{1,5\}\rangle$

Four test data for both relationships are shown in Listings 3–6.

```
...
<id>
  <memberNumber>S1234 −123456−12</memberNumber>
</id>
<moviesList>
    <movie>
        <id>12</id>
        <media>DVD</media>
        <price>3.25</price>
    </movie>
</moviesList>
...
```

Listing 6. Test data for the minimum number allowed for sequence's relationships.

Table 10
Mutation operators – data.

Divide (n)	Change value n to $1 \div n$, where n is double datatype
Multiply (n)	Change value n to $n \times n$
Negative (n)	Change value n to $-n$
Absolute (n)	Change value n to |n|
Exchange (n_1, n_2)	Substitute value n_1 for n_2 and vice versa, where n_1 and n_2 have the same type
Unauthorized (str)	Change string value str to str' OR '1' = '1

Table 11
Mutation operators – data/relations.

Operator name	Brief description
Null (n)	Set to *null* the value assigned to a node n in the SOAP message
Incomplete (n)	Delete a node n and its child nodes from the SOAP message
Inversion (n)	Inverts the order of nodes within node n in the given SOAP message
ValueInversion (n)	Inverts the order of the values assigned to the child nodes of node n in the given XML message
Mod_Len (n)	Modifies the length of the value assigned to node n in the given XML message
Space (n)	Set to ' ' the value assigned to node n

6. Data mutation

Mutation testing is a technique originally defined to analyze the adequacy of test cases sets. It basically consists of creating modified versions of programs, called mutants, and running test inputs to cause mutant programs to fail. In fact, mutation is advocated in [22] as an instantiation of the grammar-based testing technique. It has recently been used to generate test cases and data for Web services [25,38,15,27] by application of the same principles to mutate data or XML Schemas, instead of programs.

For Web services, mutation of data or XML Schemas are given by operators perturbation. The data mutation relies on the idea of RPC communication perturbation [25,15], while the schema mutation relies on perturbation operators over XML Schemas [38]. For data mutation, perturbation is applied to a *seed message*,[2] perturbing the data directly instead of using datatypes or relationships defined in the XML Schemas. Traditional mutation operators were redefined to generate data mutation of Web services messages [25], shown in Table 10.

Besides the strategy based on relationships presented in Section 5, a method to generate tests for XML-based communication by modification and further instantiation of XML Schemas was proposed in [38]. There, Schemas are modified based on predefined perturbation operators. The goal is to perturb XML Schemas to create invalid messages. With this aim, seven perturbation operators for XML Schema were defined; some are applied to nodes and others to sub-trees. For nodes, the operators are: insert and delete a new node between two other nodes, and insert and delete a new node with a datatype under an existing node. The sub-tree operators are: insert and delete a sub-tree below a node and modify an existing edge by inserting different constraints. Based on these ideas, Almeida and Vergilio [15] created six new mutation operators for SOAP messages. In some sense, they are redefinitions of the existing operators for Schemas [38] to be applied to SOAP messages. This means that these new operators are directly applied to messages and XML Schemas do not need to be provided (RPC messages, for example). Table 11 shows the operators defined by them.

[2] A message of service request. It can be created by a tester, when the system is being tested, or by an actual system/user of the service.

Table 12
Data mutation.

Operator	Message type		Technique		
	RPC	Doc	Off & Xu	Alm & Ver	New techniques
Data					
Divide (n)	•		√		⊕
		•			√
Multiply (n)	•		√		⊕
		•			√
Negative (n)	•		√		⊕
		•			√
Absolute (n)	•		√		⊕
		•			√
Exchange (n_1)	•		√		⊕
		•			√
Unauthorized (str)	•		√		⊕
		•			√
Null (n)	•			√	⊕
		•		√	⊕
Mod_Len (n)	•			√	⊕
		•		√	⊕
Space (n)	•			√	⊕
		•		√	⊕
Data Relationship					
Incomplete (n)	•			√	⊕
		•		√	⊕
Inversion (n)	•			√	⊕
		•		√	⊕
ValueInversion (n)	•			√	⊕
		•		√	⊕

All mutation operators in Table 10 were defined for RPC messages while the ones in Table 11 were defined for RPC and Document-based messages. Here, the mutation operators in Table 10 are redefined for Document-based Web services and implemented in *GenAutoWS*. Table 12 summarizes the mutation operators from the original works and the new ones presented here.

7. *GenAutoWS* – Automatically generating test suites for Web-services

GenAutoWS is a tool, based on the open source project soapUI [28], that implements all testing techniques presented in this article. The testing techniques presented by Offutt and Xu [25] and Almeida and Vergilio [15] were also implemented to make the tool more widely used (the new techniques are complementary to the previous works). The implementation of all those different techniques for Web services allowed us to compare the precision and efficiency of each technique when applied to real problems - presented in Section 8. Apart from the testing techniques already presented, the tool also includes:

- HTTP proxy to capture Web services traffic between client and server.
- Internal database – elements values are saved to be further used.
- UDDI integration.

The tool has the concept of project in which a set of WSDL interfaces are defined. For each interface, all the Web services operations are included. Embedded in each operation are the test messages either manually created by the user or automatically generated by the tool.

To illustrate the tool use, let us consider the example presented in Section 2 – the *Movie Rental Store*. This system uses *Document* type messages in the *Literal* style and its XML Schema is defined in Listing 2. Fig. 3 presents the *GenAutoWS* interface in which the project structure for the system example is shown. The project `Movie Rental` includes a single WSDL interface, `MovieRentalSOAP12Binding`, with a single message `Request 1` for the `movieRental` operation, all presented on the left-hand side of Fig. 3. The request message, `Request 1`, is shown on the right-hand side.

The test messages automatically generated by the tool are based on *seed messages* which are further perturbed using one, or many, of the testing techniques. A seed message can be

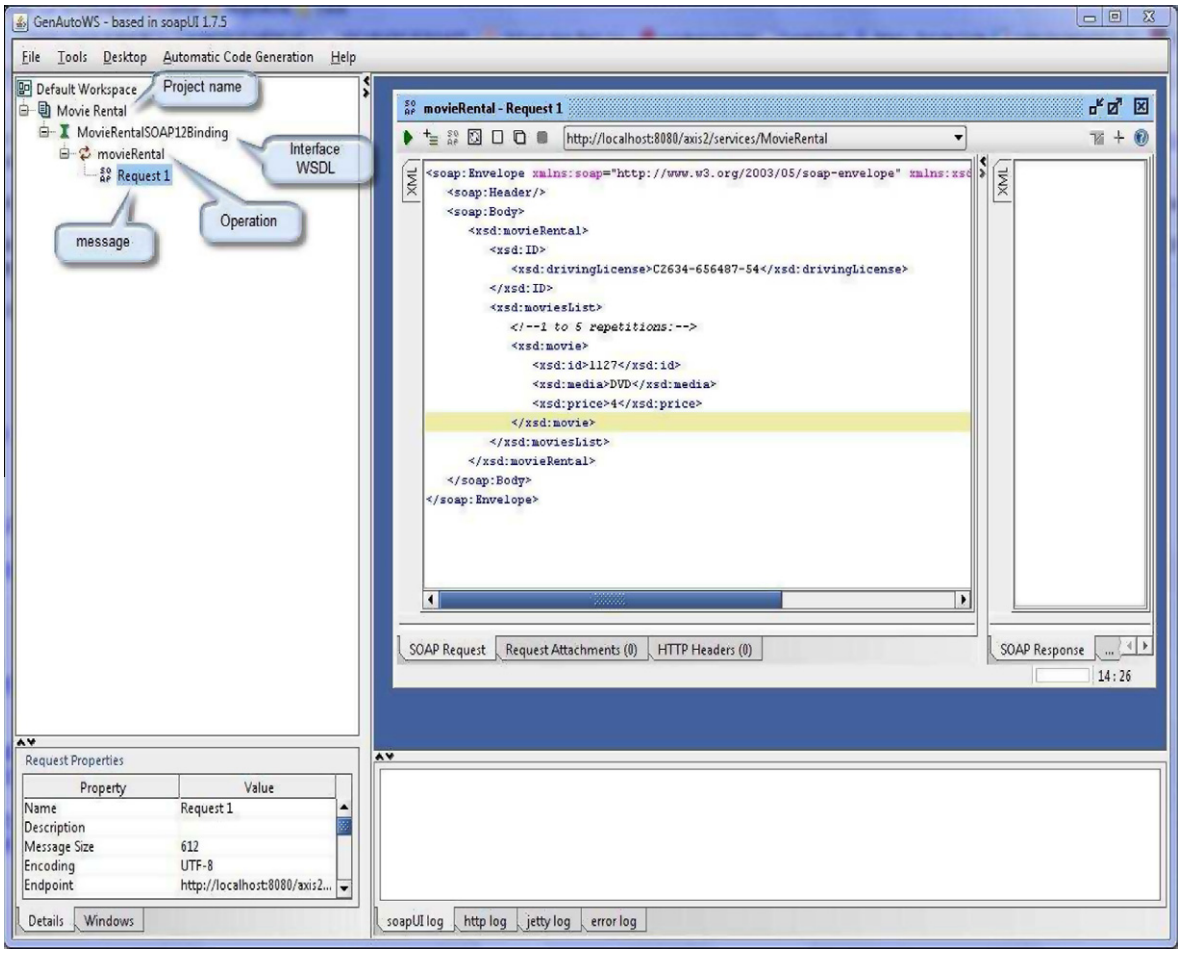

Fig. 3. *GenAutoWS*-Web Services Structure.

- manually created, corresponding to the tests created by users; or
- automatically created by the tool using the HTTP proxy feature. It works as a proxy between the client application and the Web service and is implemented at the transportation layer of Web services. All messages exchanged between them are captured and stored in the project's database, making them available to be used as seed messages.

Users are in charge of choosing the messages to be used as seeds, either from the project's database or created by her/himself. For the example shown in Fig. 3, Request 1 is used as *seed message* to automatically generate the test suite for this system.

Fig. 4 shows the options available to automatically generate test messages from a seed message. On the left-hand side are the possible techniques to be applied (presented in Sections 4–6). On the right-hand side, the test cases for the system example based on the boundary values for datatypes and XML *Facets* (DVP technique) are shown. For example, the `media` value can be perturbed with one of the string boundary values or the enumeration values, according to its definition in the XML Schema (see Listing 2).

One can choose any of those test cases available to automatically generate test suites based on the seed message. These test suites can, in turn, be executed by the tool – it is responsible for sending messages to the Web services and getting the responses back. Response messages are, in general, analyzed by testers, and in certain cases of invalid messages, automatically analyzed by the tool (when invalid messages test cases receive successful messages as response, instead of a SOAP fault). If *GenAutoWS* does not receive a SOAP fault during the execution of a test suite that contains this kind of message, the tool will notify that this test suite has failed. Fig. 5 illustrates the execution of a test suite in which one of the messages got a successful response when the test case should receive an error (the tool automatically advertises that the message has failed).

All messages created in the tool (by the user, captured using the HTTP proxy feature or automatically created by application of the testing techniques) are saved in an internal database. *GenAutoWS* has a feature, namely "internal

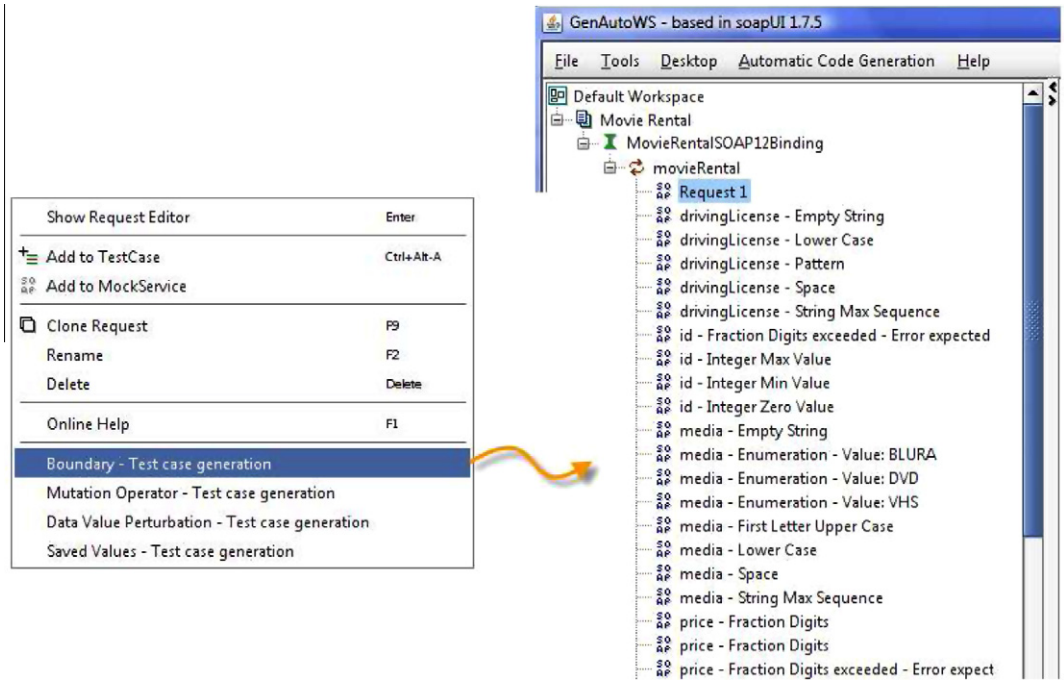

Fig. 4. *GenAutoWS*-Test Cases Inputs.

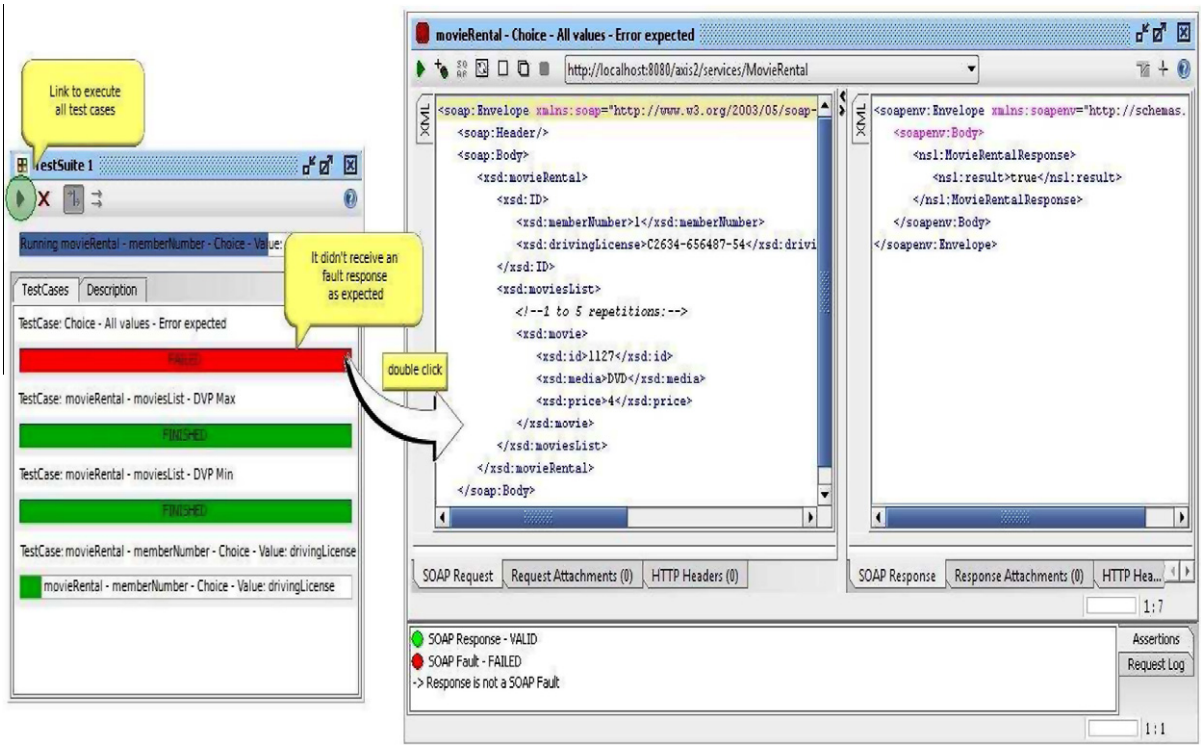

Fig. 5. *GenAutoWS*-Test suite execution.

data-perturbation", used to create new messages by swapping values from a seed message with values loaded from the internal database.

As important as creating and testing Web service is how to publish and discover information about them. Bloomberg [8] pointed out some issues on testing Web services related to publishing, finding and binding SOA capabilities. Regarding

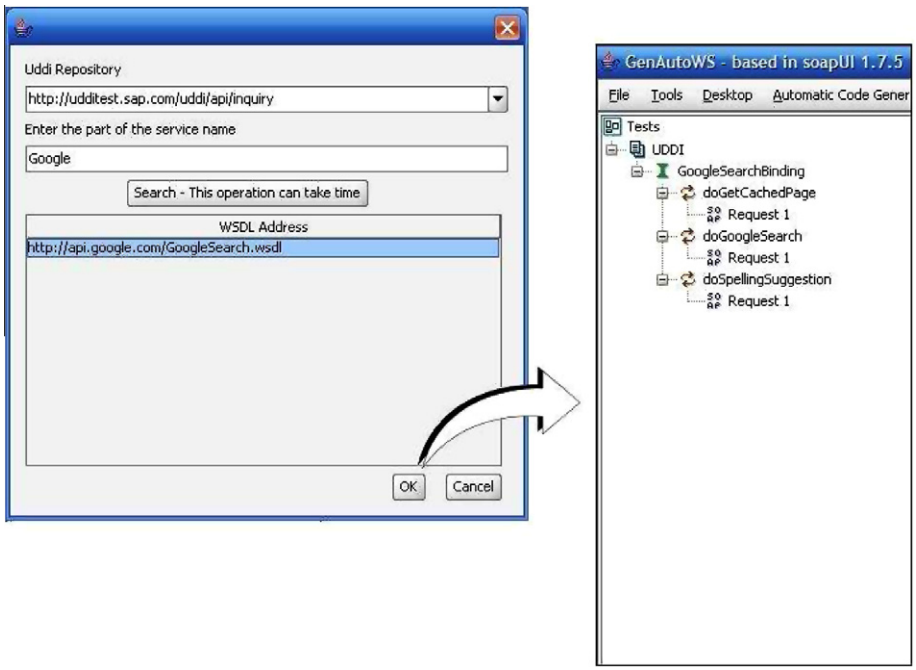

Fig. 6. *GenAutoWS* UDDI integration.

connectivity, *GenAutoWS* supports UDDI registry inquiries. The WSDL file returned from an inquiry is included in the current project and used to create new messages automatically. Fig. 6 shows UDDI integration dialog and the WSDL included in the project.

8. Empirical results

GenAutoWS was applied, as a first proof-of-concept, to generate test data for five Web services from two different Web-based systems currently used by a financial organization. The first three Web services (WS1, WS2 and W3) are part of an enterprise electronic-mail application. This application is classified as a critical system because it allows other applications to exchange e-mails in a uniform manner. It is responsible for: managing e-mail messages, including the attached files; interactively editing messages and attached files, including digital signature and cryptographed files; and managing e-mail distribution. Its efficiency is also critical because it is used to exchange at least fifty thousand e-mails per day. The other two Web services (WS4 and WS5) are used by the second system to verify credit information for e-government.[3]

All Web services used as case studies are specified using WSDL and communicate via SOAP over HTTP. The first four use document/literal while the WS5 is RPC/encoded messages. Although they are now largely used in a commercial environment, pre-released versions of these systems were used for test. During the tests, the faults found were classified as low, medium and high accordingly to their critical level to the system. All of them were tested using a seed message automatically captured by *GenAutoWS*. The messages were then perturbed using the techniques developed in the previous works and the ones proposed in this article (*GenAutoWS* implements all of them). Since projects are created for each technique, the results could be analyzed separately. The tests were automatically analyzed by the tool for the invalid messages cases and analyzed by testers for the valid messages cases.

Using the techniques of boundary values [25] and all XML Schema *Facets* presented in Section 4, we generated 162 different tests. Table 13 shows the results of each test case applied to the Web services. The results are presented with the number of faults found and the number of tests applied, giving the efficiency of each test case applied. Some of the test cases may not be applicable to certain Web services. For example, the `enumeration` *Facet* test cases could not be applied to the Web services (WS1–5) because it is not defined in. For the Web services analyzed, about 38% of the test suite generated with the new DPV techniques could detect a fault. A single fault, however, could be detected by more than one test data. Analyzing the efficiency of the techniques for detecting different faults, 20% of the test data run for these case studies could detect different faults. Note that not finding a fault in a system when the test data is applied does not dismiss the technique. This may mean that the system does not have the kind of fault addressed by the technique. Table 14 summarizes the results for this approach. The majority of the observed faults were classified of low level – e.g., no error message or incomplete message.

[3] Since these are real systems actually used by companies, no more details about the systems can be provided nor the companies involved can be cited due to a confidentiality agreement.

Table 13
DVP test cases.

	Test cases-FaultsFound/tests											
	Extensions for string type perturbation	FractionDigits exceeded	MaxLength *Facet*	Above maxLength *Facet*	Above maxInclusive	Bellow maxInclusive	Above minInclusive	Bellow minInclusive	TotalDigits exceeded	TotalDigits and fractionDigits exceeded	Pattern	Previous Techs
WS1	7/18	–	0/1	1/1	–	–	–	–	–	–	–	9/14
WS2	2/5	–	0/1	1/1	–	–	–	–	–	–	–	2/4
WS3	5/13	–	1/2	2/2	–	–	–	–	–	–	6/6	5/29
WS4	–	2/2	–	–	2/2	0/2	0/2	2/2	2/2	2/2	–	0/12
WS5	2/6	–	1/2	2/2	–	–	–	–	–	–	–	0/4
Efficiency	38%	100%	33%	100%	100%	0%	0%	100%	100%	100%	100%	25%

Table 14
Test summary – DVP.

Number of tests	162
Generated using the new techniques	99
Generated using the previous techniques	63
Total number of detected faults	32
Medium and high level faults	7
Tests that detect faults	62
From new techniques	48
From previous techniques	14
Efficiency: tests/detected faults	
New techniques	48%
Previous techniques	22%

Table 15
DCP test cases.

	Test cases – FaultsFound/tests						
	x Instances of α	y Instances of α	No instances	$\{x_1,\ldots,x_{n-1}\}$	x_i Such that $1 \leqslant i \leqslant n$	Deleting all instances of α	Previous Techs
WS1	–	–	1/1	0/1	0/2	1/1	–
WS2	–	–	1/1	0/1	–	1/1	–
WS3	–	4/6	3/7	3/9	–	1/9	4/9
WS4	1/1	–	–	–	–	–	–
WS5	–	–	–	–	–	–	–
Efficiency	100%	67%	55%	27%	0%	27%	44%

Table 16
Test summary – DCP.

Number of tests	49
Generated using the new techniques	40
Generated using the previous techniques	9
Total number of detected faults	12
Medium and high level faults	4
Tests that detect faults	20
From new techniques	16
From previous techniques	4
Efficiency: tests/detected faults	
New techniques	40%
Previous techniques	44%

Table 17
Test summary – mutation operators.

Number of tests	116
Total number of detected faults	16
Medium and high level faults	4
Tests that detect faults	34
Efficiency: tests/detected faults	29%

The relationship strategies for DCP (Section 5) generated 49 tests. Table 15 presents the test cases used to generate the test suite, together with the number of faults found. For this technique, about 41% of the test data could actually find a fault, while 25% of them could find different faults. For the Web service WS5, no test data could be generated using this technique since no XML Schema is provided.

A summary of the results for this technique is presented in Table 16. Despite the overall efficiency of the previous techniques is slightly better than the new techniques for these case studies, it is important to note that the new technique could detect 12 faults against 4 faults detected by the previous techniques (3 times the number of faults that could be previously detected).

The mutation operators presented in [25,15] and the adaptations presented in Section 6 generated 116 tests. The results are summarized in Table 17. For 34 test data, out of 116, a fault could be found, giving an efficiency of about 29% for this technique applied to these case studies.

Certain faults were detected by multiple tests, including different techniques. The new techniques presented in this article generated more test cases/data than the original ones for the DVP and DCP tests. These new tests could reveal faults not detected by the original techniques. The new testing strategies defined for DCP took advantage of testing relationships described in XML Schema not considered before. For instance, the order indicators `choice` and `all`. The generation of test cases/data that should cause errors in the application allowed us to validate error messages returned by the service, either for incorrect or nonexistent messages.

The use of datatype constraints (XML Schema *Facets*) in the DVP technique provided test cases/data within the element domain. For example, for the string element that contains `pattern`: [A-Z]{2,3}-[0-9]{2,3}, apart from the string maximum length from the previous techniques, two new ones were created: ZZZ-999 and ZZZZ-9999. The latter is an invalid message and can reveal system's faults if no message is returned advertising its invalidity.

9. Final considerations

This article proposed extensions to testing techniques based on Data Perturbation for Web services together with a tool to generate the test suites based on the existing and new techniques. For that, new boundary values considering all WSDL *Facets*, test cases using relationship defined in the WSDL, UDDI integration, internal database to collect and use values previously captured from messages were proposed.

The data value perturbation (DVP) techniques basically explore the universe of datatypes involved in existing messages to generate new messages. The new techniques presented here consider datatypes constraints defined by *Facets* and invalid messages as a complementary technique to boundary values for datatypes presented by Offutt and Xu [25]. Test cases using XML Schema *Facets* test boundary values based on datatypes and the defined constraints. Also, extensions were created for data value perturbation based on values immediately above and below the datatype domain to test invalid messages. Xu et al. [38] explored XML *Facets* under definition of a new RTG and some test cases were created based on them. The technique there, however, depends on creating RTGs for XML Schemas instead of directly applying DVP based on datatypes *Facets*, which is more efficient because no intermediate models are necessary to be built.

Regarding data communication perturbation (DCP), the data relationships are explored based on the XML Schemas. The data relationships are represented in XML Schemas by the occurrence and order indicators. In this article, apart from the testing strategies over `maxOccurs` previously defined [25], we included test data integrity and consistency with the use of the occurrence indicator `minOccurs`, and the order indicators: `all` and `choice`, and the element `any`. These new integrity and consistency tests together with the previous techniques can better cover the test space. Again, the new rules created here do not cover the ones defined in the previous works, they are used as complementary tests instead.

DVP and DCP techniques can explore datatypes and data consistency and integrity based on information in the XML Schemas. To provide data perturbation for SOAP messages, which can either have a corresponding XML Schema or not, some data mutation were created for Document-based messages as complementary cases of the previous techniques [25,15]. Note that data mutations applied directly to Document-based messages, instead of XML Schemas, are more efficient because no RTGs need to be built.

GenAutoWS is a tool built to support the extensions proposed in this article and can be used by both roles in Web services architecture: service consumers and service providers. For the former, the test suites are generated based on the service interfaces to certify particular uses of services. Service providers can also be benefited by the use of *GenAutoWS* as a development tool in which messages are automatically generated based on the presented techniques. Using *GenAutoWS* as tool support, the testing techniques can be applied to existing Web services without modifying or rewriting code, or adopting a specific framework. A first proof-of-concepts using five Web services from a financial institution was carried out. In this proof-of-concepts, DVP tests were shown more efficient compared to the other techniques. Such results are similar to the ones described by Offutt and Xu [25] in their proof-of-concepts. The Web services to which the techniques were applied so far are relatively small and single services. Studies on systematically applying the techniques to a large number of Web services, including composed services from a diversity of providers, are underway. Since the application of each technique are chosen by testers using *GenAutoWS* (all at once or one-by-one), we have the expectation they can be applied to large systems: one technique at a time to very large systems, or a set of techniques at once to smaller Web services.

There are today a set of tools to help testing Web services, but none of the existing tools support all techniques presented here. WSUnit [37] is an open source tool in which consumers can create a set of services requests to be repeatedly applied, similar to JUnit. JXWeb [20] is a scripting tool that expresses the test scenario in the form of script. Consumers and providers can create scripts to test Web services in which data and test code are separated to facilitate the analysis. Both tools are aimed to automate testing but not to automatically creating test suites. Bartolini et al. [4] developed a tool, WS-TAXI, to automatically generate test suite using TAXI [7,6], based on the category partition strategy to generate functional tests from XML schema. soapUI [28] is also an open source tool with which consumers and providers can test Web services regarding their performance and inspect WSDLs. It also allow the creation of parameterized models and has been used as basic model to implement *GenAutoWS*. We have added to soapUI all techniques presented in this article, together with the ones presented by Offut and Xu [25] and by Almeida and Vergilio [15].

The test cases added to the DVP and DCP previous techniques could generate more messages and reveal more faults than their original counterparts. The new rules inserted to DCP and the improvements for DVP were able to generate a reasonable

set of messages for WS1, WS2 and WS3. These Web services have many constraints specified by XML Schema *Facets* and occurrence and order indicators, allowing the testing techniques explore these features. Although the data mutation presented in Section 6 have a limited scope in the case studies applied so far, they might produce better results if applied to systems in which security issues are more relevant. New extensions for SQL mutations [32] are currently being investigated to be incorporated to the Web-services testing techniques and *GenAutoWS* to better explore security problems. Also, the *bypass testing* techniques [23,24] explore violations of data (invalid data) at the Web applications level to test systems robustness. These techniques can be adapted to better explore invalid messages and SQL code injection at the Web services level.

The techniques presented in this article are mainly devoted to help predicting the internal reliability of Web services, as DVP is applied, and the service usage reliability, as DCP is applied, serving for both atomic and component services [12]. Apart from reliability, security is a key issue for Web services to be applied in the large. Trust computing must ensure messages are protected against malicious parties, preventing they are accessed or modified by unauthorized parties. The new extensions for SQL mutations mentioned above can address some of the vulnerabilities of Web services related to SQL and XPath [2,13]. Web Services Security (WS-Security) [36], published by Advancing Open Standards for the Information Society (OASIS), is an initiative to establish standards to apply security to Web services. However, these standards have not been addressed by the techniques presented here and are subject of further studies.

The techniques for data perturbation based on rules for XML Schema can be easily adapted to different kinds of applications that exchange messages using the XML format. One approach is using the same techniques presented here to explore the generation of test cases for Representational State Transfer (REST) with Web Application Description Language (WADL), a message descriptor for REST services.

Acknowledgements

This project has been co-funded by the National Council for Scientific and Technological Development (CNPq - Brazil) – Proc:551038/ 2007-1 and the Ministry of Education and Research Agency (CAPES- Brazil) – Proc:0671-08-8. The author Ana C.V. de Melo also thanks the Oxford University Computing Laboratory for providing research facilities during her stay on sabbatical leave at the Oxford University, and Mario Jino by his comments on the first draft of this article.

References

[1] X. Bai, W. Dong, W.T. Tsai, Y. Chen, WSDL-based automatic test case generation for Web services testing, in: Proceedings of the IEEE International Workshop, IEEE Computer Society, 2005, p. 220.
[2] Abbie Barbir, Chris Hobbs, Elisa Bertino, Frederick Hirsch, Lorenzo Martino, Challenges of testing Web services and security in SOA implementations. In Baresi and Nitto [3]. pp. 395–440.
[3] Luciano Baresi, Elisabetta Di Nitto (Eds.), Test and Analysis of Web Services, Springer, 2007.
[4] C. Bartolini, A. Bertolino, E. Marchetti, A. Polini, WS-TAXI: a WSDL-based testing tool for Web services, in: Proceedings of the 2009 International Conference on Software Testing Verification and Validation, IEEE Computer Society, Washington, DC, USA, 2009, pp. 326–335.
[5] B. Beizer, Software Testing Techniques, second ed., International Thomson Computer Press, 1990.
[6] A. Bertolino, J. Gao, E. Marchetti, A. Polini, Automatic test data generation for XML schema-based partition testing, in: Proceedings of the Second International Workshop on Automation of Software Test, IEEE Computer Society, 2007, p. 4.
[7] A. Bertolino, J. Gao, E. Marchetti, A. Polini, TAXI – a tool for XML-based testing, in: Companion to the Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, 2007, pp. 53–54.
[8] J. Bloomberg, Testing Web Services Today and Tomorrow, The Rational Edge E-zine for the Rational Community, 2002.
[9] G. Canfora, M. Di Penta, Testing services and service-centric systems: challenges and opportunities, IT Professional, 2006, pp. 10–17.
[10] G. Canfora, M. Di Penta, Service-oriented architectures testing: a survey, in: Andrea De Lucia, Filomena Ferrucci (Eds.), ISSSE, Lecture Notes in Computer Science, vol. 5413, Springer, 2009, pp. 78–105.
[11] Boris Chidlovskii, Using regular tree automata as XML Schemas, in: Seventh IEEE Advances in Digital Libraries Conference (ADL'00), IEEE Computer Society, Los Alamitos, CA, USA, 2000, pp. 89–99.
[12] Vittorio Cortellessa, Vincenzo Grassi, Reliability modeling and analysis of service-oriented architectures. In Baresi and Nitto [3], pp. 339–362.
[13] Marco Cova, Viktoria Felmetsger, Giovanni Vigna, Vulnerability analysis of web-based applications. In Baresi and Nitto [3], pp. 363–394.
[14] Marco Crasso, Cristian Mateos, Alejandro Zunino, Marcelo Campo, Easysoc: making Web service outsourcing easier, Information Sciences (2010), doi:10.1016/j.ins.2010.01.013.
[15] Lourival F. Junior de Almeida, Silvia R. Vergilio, Exploring perturbation based testing for Web services, in: ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 717–726.
[16] A. Harrison, I.J. Taylor, Wspeer - an interface to web service hosting and invocation, in: IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4, IEEE Computer Society, Washington, DC, USA, 2005, p. 175.1.
[17] Tom Henzinger, Ranjit Jhala, Rupak Majumdar, Dirk Beyer, Blast (berkeley lazy abstraction software verification tool) model checker. <http://embedded.eecs.berkeley.edu/blast/> (last access May 2010).
[18] Angus F.M. Huang, Ci-Wei Lan, Stephen J.H. Yang, An optimal QoS-based web service selection scheme, Information Sciences 179 (19) (2009) 3309–3322.
[19] H. Huang, W. Tsai, R. Paul, Y. Chen, Automated model checking and testing for composite Web services, in: ISORC '05: Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05), IEEE Computer Society, Washington, DC, USA, 2005, pp. 300–307.
[20] JXWeb, sourceforge.net. <http://qare.sourceforge.net/web/2001-12/products/jxweb/> (last access May 2010).
[21] G.J. Myers, The Art of Software Testing, second ed., Wiley, New York, 2004.
[22] J. Offutt, P. Ammann, L.L. Liu, Mutation testing implements grammar-based testing, in: Proceedings of the Second Workshop on Mutation Analysis, IEEE Computer Society, Washington, DC, USA, 2006, pp. 12–22.
[23] J. Offutt, Q. Wang, J. Ordille, An industrial case study of bypass testing on Web applications, in: First International Conference on Software Testing, Verification, and Validation, 2008, pp. 465–474.
[24] J. Offutt, Y. Wu, X. Du, H. Huang, Bypass testing of Web applications, in: Fifteenth International Symposium on Software Reliability Engineering (ISSRE 2004), 2004, pp. 187–197.
[25] J. Offutt, W. Xu, Generating test cases for Web services using data perturbation, ACM SIGSOFT Software Engineering Notes 29 (5) (2004) 1–10.

[26] S.R. Pressman, Software Engineering: A Practitioner's Approach, sixth ed., McGraw-Hill, 2004.
[27] L. Shan, H. Zhu, Generating structurally complex test cases by data mutation: a case study of testing an automated modelling tool, The Computer Journal, 2007.
[28] soapUI, sourceforge.net. <http://www.soapui.org/> (last access May 2010).
[29] W.T. Tsai, Ray Paul, Yamin Wang, Chun Fan, Dong Wang, Extending WSDL to facilitate Web services testing, in: IEEE International Symposium on High-Assurance Systems Engineering, IEEE Computer Society, Los Alamitos, CA, USA, 2002, p. 171.
[30] W.T. Tsai, R. Paul, W. Song, Z. Cao, Coyote: an XML-based framework for Web services testing, in: Proceedings of the Seventh IEEE International Symposium on High Assurance Systems Engineering, 2002, pp. 173–174.
[31] W.T. Tsai, R. Paul, L. Yu, A. Saimi, Z. Cao, Scenario-based Web services testing with distributed agents, IEICE Transactions on Information and Systems 86 (10) (2003) 2130–2144.
[32] J. Tuya, M.J. Suárez-Cabal, C. la Riva, Mutating database queries, Information and Software Technology 49 (4) (2007) 398–417.
[33] UDDI Specification, OASIS UDDI. <http://uddi.xml.org/> (last access May 2010).
[34] W3C, Web services description language (WSDL) version 2 part 1: Core language. <http://www.w3.org/TR/wsdl20/> (last access May 2010).
[35] W3C, Web services glossary (last access May 2010).
[36] WS-Security, OASIS. <http://www.oasis-open.org/specs/> (last access May 2010).
[37] WSUnit, java.net. <https://wsunit.dev.java.net/> (last access May 2010).
[38] W. Xu, J. Offutt, J. Luo, Testing Web services by XML perturbation, in: 16th IEEE International Symposium on Software Reliability Engineering, 2005, ISSRE 2005, p. 10.
[39] Yufeng Zhang, Hong Zhu, Ontology for service oriented testing of Web services, in: IEEE International Workshop on Service-Oriented System Engineering, IEEE Computer Society, Los Alamitos, CA, USA, 2008, pp. 129–134.
[40] H. Zhu, A framework for service-oriented testing of Web services, in: COMPSAC, vol. 6, 2006, pp. 145–150.