

Towards Efficient Measuring of Web Services API Coverage

Waldemar Hummer¹, Orna Raz², and Schahram Dustdar¹

¹ Distributed Systems Group
Vienna University of Technology, Austria
{hummer,dustdar}@infosys.tuwien.ac.at

² IBM Haifa Research Lab
Haifa University Campus, Israel
ornar@il.ibm.com

ABSTRACT

We address the problem of interface-based test coverage for Web services. We suggest an approach to analyze the Application Programming Interface (API) of Web services, calculate the number of possible input combinations and compare it to the number of actual historical invocations. Such API coverage metrics are an indicator to which extent the service has been used. Measuring API coverage is a key concern for assessing the significance of Validation and Verification (V&V) techniques; on the other hand, API coverage metrics can also yield interesting usage reports for a service-based system in production use. The coverage metrics rely on the exact specification of service interfaces, and we provide a mechanism to specify restrictions for data types in the Java Web services framework (JAX-WS). As full enumeration of all possible inputs is often unfeasible, we allow the definition of custom coverage metrics by means of domain partitioning: the user divides domain ranges into subsets, and a coverage of 100% means that the logged invocations contain at least one sample for each subset. Based on a prototype implementation, we evaluate different aspects of our approach.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification

General Terms

Design, Management, Reliability, Verification

Keywords

API Coverage Metrics, Web Services, TeCoS Framework

1. INTRODUCTION

In recent years, the Service-Oriented Architecture [15] (SOA) has become a widely adopted paradigm to create loosely coupled distributed systems, and Web services¹ are the most commonly used technology to build SOA. One of the defining characteristics of SOA is that the API (application programming interface)

¹<http://www.w3.org/2002/ws/>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PESOS '11 Hawaii, USA

Copyright 2011 ACM X-XXXXXX-XX-X/XX/XX ...\$10.00.

of the available services is published to service consumers using standard, machine-readable description languages. In the case of Web services, this is achieved using the Web Services Description Language (WSDL), paired with XML Schema Definitions (XSD) of the service operations' input and output messages.

Now that the WS-* stack builds a solid technological foundation, traditional software engineering disciplines are being applied to Web services. Among these disciplines is software testing in terms of verification and validation (V&V) [6, 16]. An important field in software testing is concerned with test coverage, i.e., the extent to which an implemented system has been tested or used. Classical code-based coverage metrics such as function, statement or branch coverage require access to the source code, which is not always possible for Web services. Therefore, testing methods for Web services can usually only rely on the service's API definition, and hence focus on black-box testing of single services or testing the composition of services [9]. A variety of professional commercial services have evolved around Web services API testing^{2 3 4}, which clearly shows the practical relevance of this field.

API coverage is commonly expressed as the ratio of previously performed, distinct invocations to the number of (theoretically) possible invocations as defined by the API [12]. In the simplest case, single parameter values are varied and tested, e.g., with extreme values, to verify functionality and detect failures. Since an isolated analysis of parameters is often not sufficient, input combinations need also be considered. Achieving an API coverage of (near) 100% in this case is generally subject to the problem of *combinatorial explosion* [16], because all input parameter combinations need to be considered. Hence, it is desirable to restrict the value domain of each parameter to its smallest possible size, in order to minimize the total number of possible combinations. We see two possibilities to achieve that. Firstly, Web service developers need to provide a more precise specification of the valid operation parameters, e.g., in terms of string patterns or allowed numeric ranges. Secondly, service testers may analyze these parameters to identify similarities or combinations that seem less important to be tested. XSD already provides a solution for the first point in the form of *facets*, but expressing such schema-based restrictions is surprisingly hard to achieve in major Web service frameworks such as JAX-WS⁵ (Java API for XML Web Services) or Microsoft's .NET platform. Essentially, developers cannot rely on the automated XSD generation, but have to manually define the XML Schema that uses facet restrictions. Concerning the second point, there is still an evident lack for API coverage frameworks with customizable coverage metrics that

²<http://qualitylogic.com/tuneup/uploads/docfiles/web-api-testing.pdf>

³<http://www.crosschecknet.com/>

⁴http://www.stylusstudio.com/ws_tester.html

⁵<http://jcp.org/en/jsr/detail?id=224>

can be easily plugged into (existing) service-based systems.

In our previous work, we developed coverage metrics for data-centric dynamic service compositions [11]. Under the same umbrella project named *TeCoS* (Test Coverage for Service-based systems) we now investigate coverage of service APIs. In this paper we apply software testing concepts to Web services and present a solution for measuring API coverage based on historical invocations. Our contribution is threefold: 1) we define API coverage metrics and their instantiation for Web services, 2) we suggest the implementation of XSD facets in the JAX-WS framework, and 3) we present and assess our prototype in an experimental evaluation.

The remainder of this paper is structured as follows. In Section 2 we briefly introduce a scenario Web service to which the API coverage metrics will be applied. Section 3 illustrates our approach for exact Web service interface definition using Java annotations. Section 4 discusses how different API coverage metrics can be defined by SOA testers and developers, and in Section 5 we detail the technique for measuring and calculating these metrics. Section 6 presents our prototype implementation, and the overall approach is evaluated in Section 7. We then discuss related work in the field of API coverage for service-oriented systems in Section 8, and Section 9 concludes the paper with an outlook for future work.

2. SCENARIO

As an illustrative scenario for this paper we assume a *Chart Web service* (CWS) that is capable of generating chart images from numeric input values, similar to the *Google Chart API*⁶. For the sake of brevity we consider only a simplified version of a service with 1 operation (*generateChart*) and 4 parameters (see Table 1). However, our approach is also applicable to more comprehensive APIs.

Service <i>ChartService</i>		
Operation <i>generateChart</i>		
Parameters:		
Name	XSD Type	Constraints
<i>type</i>	string	$type \in \{ 'line', 'bar', 'pie' \}$
<i>values</i>	list of integers	$values \in \{ -100, \dots, 100 \}^n$, $1 \leq n \leq 10$
<i>name</i>	string	pattern "[a-z][a-z0-9]{0,4}"
<i>config</i>	list of Configs	-

Table 1: API of Scenario Service

The *generateChart* operation has a parameter *type*, whose domain is an enumeration of 3 values. The parameter *values* is a list of integers, with a length between 1 and 10, and an integer range of $\{-100, \dots, 100\}$. A short alphanumeric identifier (maximum length 5, starting with a letter) is provided using the parameter *name*. Finally, various configuration settings in the form of key-value pairs are passed to the operation using the parameter *config*. The XSD complex type *Config* contains a sequence with a key element and a list of one or more *value* elements. More details about the XML schema are given in Section 3.

3. EXACT DEFINITION OF WEB SERVICE API WITH JAX-WS AND JAXB

To provide meaningful data for API coverage metrics, it is desirable to limit the domain range of the parameters of service operations. For example, the numeric values in our scenario are of type *integer*. Considering the service is implemented in Java, each

value has a range of 32 bits (4.294.967.296 possibilities), while the API requires the values to be between -100 and +100 (201 possibilities). Note that constraining the domain range does not eliminate the combinatorial explosion problem per se, but can lead to a significantly smaller size of the parameters' combined domain range.

```

1  @XmlRootElement
2  public class GenerateChart {
3      public static enum ChartType { line, bar, pie }
4      public static class Config {
5          public String key;
6          public List<String> value;
7      }
8      @XmlElement(required=true)
9      public ChartType type;
10     @MinOccurs(1) @MaxOccurs(10)
11     @Facets(minInclusive=-100, maxInclusive=100)
12     public List<Integer> value;
13     @Facets(pattern="[a-z][a-z0-9]{0,4}")
14     public String name;
15     public List<Config> config;
16 }
17
18 @WebService
19 public class ChartService {
20     public String generateChart(GenerateChart request) {
21         // generate chart, return in Base64 format
22     }
23 }

```

Listing 1: Implementation of Chart Web Service

We argue that an exact definition of the interfaces should be an integral part of engineering service-oriented systems to provide for reasonable coverage analysis. The Web services framework is based on XML messaging and allows to limit domain ranges of simple types using XSD *facets*, and the range of elements with multiple occurrences using the XSD attributes *minOccurs* and *maxOccurs*. An exact specification of parameters is also useful to enforce the integrity of invocations at runtime. Instead of validating parameters manually in the implementation (e.g., checking for *null* values to avoid a *NullPointerException*), the service execution engine can automatically filter invalid parameters by matching incoming messages against the XSD using XML Schema validation.

```

1  <complexType name="generateChart">
2      <sequence>
3          <element name="type">
4              <simpleType>
5                  <restriction base="xs:string">
6                      <enumeration value="line" />
7                      <enumeration value="bar" />
8                      <enumeration value="pie" />
9                  </restriction>
10             </simpleType>
11         </element>
12         <element name="value" minOccurs="1" maxOccurs="10">
13             <simpleType>
14                 <restriction base="int">
15                     <minInclusive value="-100" />
16                     <maxInclusive value="100" />
17                 </restriction>
18             </simpleType>
19         </element>
20         <element name="name">
21             <simpleType>
22                 <restriction base="string">
23                     <pattern value="[a-z][a-z0-9]{0,4}" />
24                 </restriction>
25             </simpleType>
26         </element>
27         <element name="config" type="Config" maxOccurs="99"/>
28     </sequence>
29 </complexType>
30 <complexType name="Config">...</complexType>

```

Listing 2: Generated XML Schema for *GenerateChart*

⁶<http://code.google.com/apis/chart/>

In the world of Java Web services (JAX-WS), JAXB (Java Architecture for XML Binding) is used to create a mapping between XML elements and Java objects. Because JAXB currently supports neither XSD facets nor occurrence ranges, we suggest to extend JAXB with 3 new annotations: `@Facets`, `@MinOccurs` and `@MaxOccurs`. The Java implementation of the Chart Web service using the extended JAXB annotations is illustrated in Listing 1.

When deploying the Web service, the WSDL file is automatically generated and contains the according XSD type for the class `GenerateChart`, which is printed in Listing 2. Details concerning the implementation are presented in Section 6. Based on this exact description of the service API, we are now able to define metrics to measure the coverage of the Chart Web service in Section 4.

4. CONFIGURABLE API COVERAGE METRICS FOR WEB SERVICES

API coverage can be measured in various ways, and the ability to define coverage metrics is vital for SOA testers and developers. For instance, we could request that the Chart Web service needs to be invoked at least with the extreme values for the parameter `values` (i.e., -100 and +100) and with all possible chart `types`. Furthermore, it could be of interest to see a service invocation fail which uses values beyond the range (e.g., -200). Apart from testing the functionality of services, coverage metrics can also be used to generate usage reports of the running system, e.g., how often service consumers have requested different chart types, or which configuration parameters were frequently used.

Determining the aforementioned coverage metrics is supported in our approach by means of customizable *domain partitioning*. The basic idea is that the user specifies rules which divide the range of a parameter domain d into (usually disjoint) subsets $P_d = \{d_1, \dots, d_n\}$, $d_1 \cup \dots \cup d_n = d$. At runtime, service invocation messages are logged and matched against the specified rules to find out which subset the message belongs to. A 100% coverage would then indicate that at least one message has been logged for each subset.

We distinguish two methods to define domain partitioning rules:

1. *Membership Test (MT)*: For each subset s in the partition P_d of domain d , a membership function $m_s : d \rightarrow \text{boolean}$ determines for every element of d whether it is a member of s . That is, the user has to define n different membership functions, and all n expressions need to be evaluated to determine which subset(s) an element is member of.
2. *Membership Identifier (MI)*: The user specifies 1) the total number of subsets in the partition, $|P_d|$, and 2) a single membership function $i : d \rightarrow ID$, which returns for each element in the domain d a unique numeric identifier ($ID \subset \mathbb{R}$, $|ID| = |P_d|$) of the subset it is member of. Evidently, with this approach the subsets d_1, \dots, d_n are automatically disjoint.

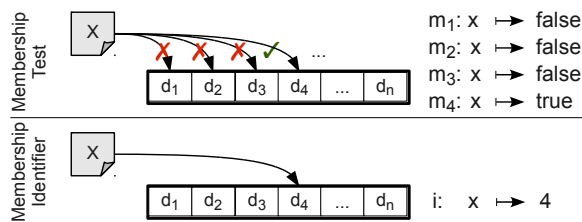


Figure 1: Domain Partitioning Methods

The two methods are illustrated in Figure 1. The MT method requires more function evaluations, but it is more expressive and

supports non-disjoint subsets (i.e., each element may be member in more than one subset). For both methods, the expressions are defined using the syntax of the *Groovy*⁷ scripting language for Java. Seven exemplary domain partitions are illustrated in Table 2. The predefined variable x references the target value for which the function expression should be evaluated (e.g., the parameter `values` introduced in Section 2), and `MIN`/`MAX` are predefined variables for the minimum/maximum value of the domain of x . It is important to mention that domain partitions may apply to different targets: 1) the numerical occurrence count (e.g., $x \in \{1, \dots, 10\}$ for `values`), and 2) the value (text content) of XML elements (e.g., $x \in \{-100, \dots, 100\}$ for `values`). Hence, the variable x refers to either of the two, depending on the target for which the partition has been specified. The examples in Table 2 contain numerical partitions for extreme values (t), negative/zero/positive values (n , z), blocks of values (b) or values that are out of the valid range (o), as well as a partition to select the type `pie` (p) and an “ignore” partition (returns `true` on all membership tests). Additionally, we provide the predefined *default* partition (d) which defines no custom subsets but reflects the value domains as specified in the XSD.

ID	Name	Type	P_d	$m_1(x)$	$m_2(x)$	$m_3(x)$	$i(x)$
i	<code>ignore</code>	MT	1	<code>true</code>	-	-	-
n	<code>negZeroPos</code>	MT	3	<code>x < 0</code>	<code>x == 0</code>	<code>x > 0</code>	-
z	<code>zero</code>	MT	1	<code>x == 0</code>	-	-	-
t	<code>extreme</code>	MT	2	<code>x == MIN</code>	<code>x == MAX</code>	-	-
b	<code>blocksOf10</code>	MI	<code>int (abs (MAX - MIN) / 10) + 1</code>	-	-	-	<code>(int) x / 10</code>
o	<code>outOfRange</code>	MT	1	<code>x < MIN x > MAX</code>	-	-	-
p	<code>pieChart</code>	MT	1	<code>x == 'pie'</code>	-	-	-
d	<code>default</code>	predefined, based on XSD of the Web service					

Table 2: User-Defined Domain Partitions

5. EFFICIENT COVERAGE COMPUTATION

In the following we discuss how Web service API coverage is computed taking into consideration the configurable coverage metrics described in Section 4. A prerequisite is that all invocations performed in the service-based system are accessible. Details on how our implementation intercepts and logs the invocations messages are given later in Section 6. For now, we assume that the XML messages are stored in tables of a relational database management system (DBMS). To enable detailed queries on their structure and content, the XML messages are parsed and stored in a structured format as depicted in Figure 2. This figure uses UML notation and the model maps directly to the persistence (database) level.

The base class is `XMLNode`, which models nodes in an XML structure. The type (simple/complex element, attribute, text, ...) is distinguished by the attribute `type`, and the `value` (text content) may be empty for complex type elements. Additionally, the complete XML source of the root elements is stored in `xml` for performance optimizations (see Section 7). The actual XSD type of elements is stored separately, and is left out in the figure. The combination of `name` and `value` is unique, i.e., every encountered XML is only stored once and there are no redundancies. In turn this means that the parent-child relationship has many-to-many cardinality: an element may have multiple children and each element may be the child of more than one parent. In addition to the parent-child association, we store the list of all descendants of an invocation input message m using the class `XMLDescendant`.

⁷<http://groovy.codehaus.org/>

For point 2, PI_v can be inferred from the domain partition DP_v and is generally equal to the number of subsets in this partition. PI_o expresses the number of possible occurrences of an element, and is subject to partitioning with DP_o . Besides custom domain partitions, our approach also supports computing the number of possible invocations directly from the XSD definitions (illustrated in Figure 4). Usually this number is much higher, e.g., the regular expression of `name` specifies 44917730 ($\approx 45M$) possibilities (see Section 6 for details about how this is determined). The total number of possibilities PI_a is a combination of PI_v and PI_o . Consider the `value` element and its assigned partition $n.X$ (negative-zero-positive) which has 3 subsets ($PI_v=3$). As specified in the XSD, this element may occur between 1 and 10 times, which means that the total number of possible instantiations is $3^1 + 3^2 + \dots + 3^{10}$. We observe that this term can be calculated by means of a geometric series, as printed in Equation 1.

$$\sum_{k=m}^n r^k = \frac{r^{n+1} - r^m}{r - 1} \quad (1)$$

In our example, $m = 1$, $n = 10$, $r = 3$, the value of PI_a hence amounts to 88572 possible combinations. The calculation of PI_a for complex XSD types (e.g., `generateChart`, `config`) is different, because all combinations of the contained sub-elements need to be taken into account. PI_a of a complex type is therefore defined as the product of the PI_a values of all its direct child elements. For instance, `generateChart` has a total of 265716 possibilities ($3 \cdot 88572 \cdot 1 \cdot 1$) when we apply our example domain partitioning. Note that it is in fact possible that a complex type also contains a text content which is not embraced by another element (e.g., `<a>text`), for instance resulting from an XSD `any` element. Our approach supports such constructs, since in terms of API coverage we can treat the text node like a simple type element.

Point 3 of the above list concerns the number of distinct invocations, DI_a , the calculation of which has been described earlier in this section. Additionally, we calculate DI_v and DI_o with two modified versions of the SQL query in Figure 3. In our example for the element `value`, DI_v is 3 because values from all 3 domain subsets (negative, zero, positive) have been logged, and DI_o is 2 because we logged invocations with 2 different occurrence counts of the `value` element (7 and 4).

Having computed all required values, the API coverage is calculated as DI_a/PI_a . The example coverage of 0.0008% is very low, which is hardly surprising considering the fact that we only logged 2 invocations for illustrative purposes. The overall coverage increases considerably when either more invocations are stored in the DB or other (more restricting) domain partitions are chosen.

6. IMPLEMENTATION

In this section we discuss the implementation of the presented concepts. The overview of the TeCoS framework architecture in Figure 5 shows a SOA under test consisting of Web services, which are invoked by a business process and end users. The services are deployed in a container, whose responsibility is to transparently log incoming invocations to the *Tracing Service* (TS). To that end, we provide an implementation of *SOAPHandler* which can be easily plugged into the JAX-WS handler chain. The TS makes use of a *Service Registry* and logs all messages to the *Invocation Database* (InvDB), which also incorporates a simplified *XML Schema Database*. MySQL⁸ (version 5.1) is used as the DBMS. The *Coverage Calculator* (CC) operates on the InvDB and contains

⁸<http://www.mysql.com/>

the core business logic for computing API coverage metrics. Furthermore, the CC manages the user-defined domain partitions. The TeCoS framework also defines coverage metrics for WS-BPEL service compositions, which we have presented in earlier work [11]. We further plan to extend our notion of API coverage from parameter combinations of a single operation to invocation sequences.

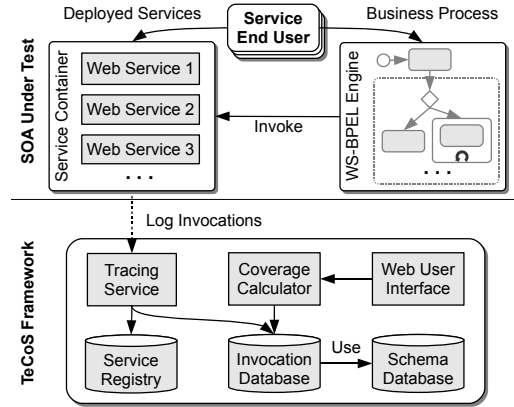


Figure 5: TeCoS Framework Architecture

To allow for a graphical feedback and experimental exploration of coverage metrics, we have implemented a graphical Web user interface (UI) using Java Server Faces (JSF) technology. The UI allows the configuration of user-defined partitions, visualizes the information stored in the Service Registry, and depicts invocations and API coverage data. A screenshot of the Web UI, showing the coverage results of our test scenario, is depicted in Figure 6. The leftmost table column lists the element structure together with the applicable facets. By clicking on the name of an element, the user receives additional information such as the actual occurrences and values that were logged for this element. We are currently also working on extended coverage reports with graphical charts.

XML Element	Domain Partition	Poss. Occ.	Dist. Occ.	Poss. Val.	Dist. Val.	Poss. Inv.	Dist. Inv.	Fault %	Cvg. %
generateChart		1	1	-	-	265716	2	0	0.0008
<code>type: chartType (enum, 3 values)</code>	Val.: XSD-based	1	1	3	2	3	2	0	66.6667
<code>value: int [1..10] (-100,...,100)</code>	Occ.: XSD-based Val.: negZeroPos	10	2	3	3	88572	2	0	0.0023
<code>name: string /[a-z][a-z0-9]{1-4}/</code>	Val.: ignore	1	1	1	1	1	1	0	100
<code>config [0..99]</code>	Occ.: ignore	1	1	-	-	1	1	0	100
<code>key: string</code>	Val.: ignore	1	1	1	1	1	1	0	100
<code>value: string [1..*]</code>	Occ.: ignore Val.: ignore	1	1	1	1	1	1	0	100

Figure 6: Screenshot of Web User Interface

As part of its extension mechanism, the JAX-WS reference implementation⁹ (RI) employs a means to implement special-purpose

⁹<http://jax-ws.java.net/>

interceptors for custom WSDL generation (e.g., WS-Addressing headers). However, JAX-WS RI does not allow custom schema generation based on JAXB. In 2007, a JIRA entry¹⁰ with priority “Major” was created for this issue, but as of January 2011 the status of this feature request is still “Open”. We therefore investigated the source code of JAXB RI to hook into the schema generation process. Our modification requires 3 new annotation interfaces (used in Listing 1), 1 new class (XsdFacets) and 3 additional lines of code in the class `com.sun.xml.bind.v2.XmlSchemaGenerator` (lines with comments on right margin in Listing 3). We have initiated an open discussion on the JAXB *dev* mailing list about whether our solution will be merged into JAXB RI.

```

1 private Tree handleElementProp(
2     final ElementPropertyInfo<T,C> ep) {
3     ...
4     if(!XsdFacets.hasFacets(t, e)) //1
5         writeTypeRef(e,t, "type");
6     ...
7     if(!XsdFacets.writeOccurs(t,e,isOptional,repeated)) //2
8         writeOccurs(e,isOptional,repeated);
9     XsdFacetsGenerator.addFacets(t, e); //3
10    ...
11 }

```

Listing 3: Modifications of *XmlSchemaGenerator* in JAXB RI

One of the key prerequisites for determining the number of possible invocations is the analysis of regular expressions (regex). Usually, regex engines allow only to match a given string against a regex string, but have no support for generating all strings that match the regex. Our current implementation is hence based on a small (yet powerful) Python script¹¹ by Paul McGuire, which can determine the number of all possible matching strings for a regex.

7. DISCUSSION AND EVALUATION

To evaluate different aspects of our approach, we have implemented the test service presented in Section 2 and generated 10000 invocations with randomized parameters (concerning both the length of sequences and the values of simple types). Figure 7 depicts the measured results when applying the domain partitions of Figure 4: number of distinct invocations (DI), time required to calculate this number (CT), and XML Elements (XE) stored in the database. As described in Section 5, only unique XML elements are stored in the database; e.g., if two subsequent invocations i_1 and i_2 use a `value` parameter with value 5, a new database entry is inserted for i_1 , but the stored invocation for i_2 then points to the existing row. This means that more queries are required when we store an invocation message, with the advantage that the DB is free of duplicates. We can observe the memory effect in the figure: the minimum number of XML elements per invocation is 3 (1 `type`, 1 `value`, 1 `name`), but for 10000 invocations only 10209 distinct elements were saved.

Another interesting aspect is the trend of DI with increasing number of total random invocations. We calculate DI with the partition settings and the query depicted in Figure 3. Whereas DI rises quite sharply in the initial phase (635 out of 1000, 1074 out of 2000), the curve flattens out when more invocations are recorded (Δ DI between 9000 and 10000 was 160). The reason for this is clearly that an increasing number of stored messages decreases the possibility that a new distinct invocation arrives. Finally, the figure indicates the time required to calculate the coverage metrics with the aid of the DBMS. The time for 10000 stored invocations is still feasible (< 6 seconds) and shows that the approach generally scales

¹⁰<http://java.net/jira/browse/JAXB-392>

¹¹<http://pyparsing.wikispaces.com/file/view/invRegex.py>

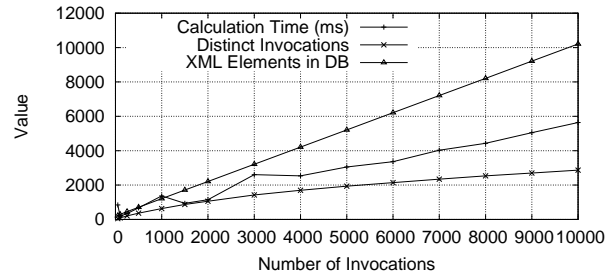


Figure 7: Performance Evaluation Results

well. However, for very large systems with millions of records the current solution poses limitations for real-time usage. In our future work, we are therefore investigating techniques for caching and storing/calculating coverage data in a distributed manner.

For simple processing of XML data, MySQL provides the function `ExtractValue` to access node values in XML markup via XPath expressions. To evaluate the use of this function, we have stored the complete XML source of invocation messages in the column `xml` of table `XMLNode` (cf. Figure 2). This allows to execute the coverage sub-queries (cf. Figure 3) directly on the XML source, e.g., `select ExtractValue(xml, '//type') from XMLNode ...` Such XPath-based queries show very good performance, even with several thousand rows in the table, and in some cases an improvement could be achieved over the alternative approach of joining `XMLDescendant` with the `XMLNode` table. The queries using `ExtractValue` are especially profitable when the stored invocations differ in content, but the tradeoff is that always the complete XML is stored, which results in redundant and duplicate data.

#	Example Metrics	gt_v	gv_o	gv_v	gn_v	gc_o	gck_v	gcv_o	gcv_v
		User-Specified Partitions							
1	Usage Counter	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>
2	Extreme Values	<i>i</i>	<i>t</i>	<i>t</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>
3	Chart Types	<i>d</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>
4	Blocks of 10	<i>i</i>	<i>i</i>	<i>b</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>
5	Pie Charts	<i>p</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>	<i>d</i>
6	Out of Range	<i>i</i>	<i>i</i>	<i>o</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>
7	Config. Settings	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>d</i>	<i>i</i>	<i>i</i>	<i>i</i>
8	Config. Keys	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>i</i>	<i>d</i>	<i>i</i>	<i>i</i>
9	Values of Default Pie Charts	<i>p</i>	<i>i</i>	<i>d</i>	<i>i</i>	<i>z</i>	<i>i</i>	<i>i</i>	<i>i</i>

Table 3: Partition Settings for Example Coverage Metrics

The introductory paragraph of section 4 mentions that TeCoS allows various coverage metrics and usage reports. To provide a complete taxonomy of supported metrics is out of the scope of this paper, but we evaluate this claim by giving a number of concrete examples in Table 3. The column titles are analogous to those used in Figure 3 and the partition IDs refer back to Table 2. In the default setting (#1) all historical (distinct) invocations are computed, which provides a general usage counter of the chart service. To analyze specific parameters, the domain partitions of all remaining elements are set to *i* (ignore). For instance, example #2 calculates the coverage of the extreme values (concerning both the occurrence and the numeric value) of the `value` parameter, and metric #3 determines which chart types are covered. Conversely, if a subset of parameters should be fixed, all remaining parameter partitions can be set to *d*: e.g., #5 represents the partition settings to determine all `pie charts` requests. Metric #9 is a more advanced example which shows the potential of combining different parameter partitions. It calculates

the chart values ($gv_v=d$) of all pie chart requests ($gt_v=p$) with default settings, i.e., no additional user configurations ($gc_o=z$).

8. RELATED WORK

In the following we discuss related work in the area of API test coverage and its application to service-oriented systems.

Xu et al. [19] present an approach for testing Web services based on the operations' XML schema. The paper defines a formal model for XSD, and defines operators for schema "perturbation", i.e., for creating XML messages that are *invalid* with respect to the schema. These operators, which include insertion, deletion and changing of nodes, are the basis for test coverage criteria. Their approach is complementary to ours, as it focuses on generating invalid test cases, whereas we perform logging and analysis of performed invocations, which serves both as a reporting tool for service providers and as a means to enforce coverage goals for service testers.

Apart from interface-based Web service testing, different methods have been proposed for, e.g., group testing [17], collaborative contract-based testing [2], and testing of data-centric service compositions [14, 11]. Further coverage criteria have been defined for operation/message combinations [5], and execution paths in processes defined with the Web Services Business Process Execution Language (WS-BPEL) [10, 13]. Baresi et al. [3] presented various techniques for Web services testing and verification, including monitoring, modeling, and reliability analysis.

A certain similarity can be seen with the approach of Bertoloni et al. [7] who also utilize XML Schema based partitioning. A major difference is that only the partitions defined by the schema are considered, whereas we support customizable partitioning of domain ranges. Furthermore, their approach aims at test data generation, which is not the primary concern in this paper. Under the term "whitening" of SOA testing [4], Bertoloni et al. have suggested that *testable* services should expose additional metadata in the form of coverage data. Similarly, we argue that providing an exact interface definition is a key requirement for meaningful API coverage testing and analysis. Conversely to Bertoloni et al., we do not require the service under test to measure the coverage itself, but merely to log its invocations to the TeCoS tracing service.

Also Bai et al. have studied automatic test case generation based on WSDL documents and the XML schema exposed therein [1]. Their approach mostly concentrates on random generation based on the interface and dependencies (message flows) of operations, but does not define in detail how coverage metrics are calculated.

Domain partitioning for API testing has been proposed previously, e.g., in the form of the Category-Partition test design pattern by Binder [8]. Jorgensen and Whittaker [12] do not differentiate between methods for defining partitions, whereas we provide two alternative ways (MT and MI) and additionally support default partitioning that is automatically derived from WSDL documents.

9. CONCLUSION

We presented an efficient, novel solution to measure API coverage of service-based systems implemented with Web services. User-specified domain partitioning allows for the definition of customizable and reusable API coverage metrics. Our end-to-end framework TeCoS can be easily plugged into existing service execution engines to log service invocations, calculate coverage data and render the results in a Web UI. Furthermore, we suggest to enhance JAX-WS and JAXB with support for XSD facets, and provide a solution that integrates seamlessly with the JAXB RI. This user-friendly extension greatly reduces the required development effort and is a step towards meaningful coverage data and schema-based

validation of invocation parameters. The performance and scalability has been successfully evaluated in our experimentation. As part of our ongoing work, we plan to extend the scope of API coverage to invocation sequences and semantic input description, as well as to enhance the TeCoS framework with distributed storage and computation of coverage data.

10. REFERENCES

- [1] X. Bai, W. Dong, W.-T. Tsai, and Y. Chen. WSDL-based automatic test case generation for Web services testing. In *Int. Workshop Service-Oriented Syst. Eng.*, pages 215–220, 2005.
- [2] X. Bai, Y. Wang, G. Dai, W.-T. Tsai, and Y. Chen. A framework for contract-based collaborative verification and validation of web services. In *10th Int. Conf. on Component-Based Software Engineering*, pages 258–273, 2007.
- [3] L. Baresi and E. D. Nitto. *Test and Analysis of Web Services*. Springer-Verlag New York, Inc., 2007.
- [4] C. Bartolini, A. Bertolino, S. Elbaum, and E. Marchetti. Whitening SOA testing. In *ESEC/SIGSOFT FSE '09*, pages 161–170, 2009.
- [5] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based Testing Tool for Web Services. In *ICST 2009*, pages 326–335, 2009.
- [6] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, USA, 1990.
- [7] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. Automatic Test Data Generation for XML Schema-based Partition Testing. In *Int. Workshop Automation of Software Test*, 2007.
- [8] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [9] G. Canfora and M. Di Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*, 8(2):10–17, 2006.
- [10] J. García-fanjul, J. Tuya, and C. D. L. Riva. Generating Test Cases Specifications for BPEL Compositions of Web Services Using SPIN. In *WS-MaTe 2006*, pages 83–94, 2006.
- [11] W. Hummer, O. Raz, O. Shehory, P. Leitner, and S. Dustdar. Test coverage of data-centric dynamic compositions in service-based systems. In *4th IEEE International Conference on Software Testing, Verification and Validation*, 2011.
- [12] A. Jorgensen and J. Whittaker. An API Testing Method. In *STAREAST Conf. on Softw. Testing Analysis & Review*, 2000.
- [13] D. Lübke, L. Singer, and A. Salnikow. Calculating BPEL Test Coverage Through Instrumentation. In *Int. Workshop on Automation of Software Test*, pages 115–122, 2009.
- [14] L. Mei, W. Chan, and T. Tse. Data flow testing of service-oriented workflow applications. In *ICSE*, 2008.
- [15] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, 2007.
- [16] J. O. Paul Ammann. *Introduction to Software Testing*. Cambridge University Press, 2008.
- [17] W. T. Tsai, Y. Chen, R. Paul, N. Liao, and H. Huang. Cooperative and Group Testing in Verification of Dynamic Composite Web Services. In *COMPSAC*, pages 170–173, 2004.
- [18] World Wide Web Consortium (W3C). XML Path Language (XPath). <http://www.w3.org/TR/xpath/>, 1999.
- [19] W. Xu, J. Offutt, and J. Luo. Testing Web Services by XML Perturbation. In *16th IEEE Int. Symposium on Software Reliability Engineering*, pages 257–266, 2005.