2014 IEEE International Conference on Software Maintenance and Evolution

# A Web Service Test Generator

Patrick Vanderveen, Michael Janzen, Andrew F. Tappenden
Department of Computing Science
The King's University
Edmonton, Canada
{patrick.vanderveen, michael.janzen, andrew.tappenden}@kingsu.ca

***Abstract*—An automated process for generating test inputs for web services from a WSDL is presented. A grammatical representation of the web service is extracted from the WSDL and used to produce test cases. A context-free grammar (CFG) is generated from the XSD that is stored in the WSDL. The CFG is provided as input into a constraint-satisfaction problem solver to automatically generate a diverse set of structurally correct XML documents. Testing data is then inserted into the XML templates in accordance with any constraints specified in the XSD. Web service-specific testing can be performed with the inclusion of external datasets and service-specific configurations.**

***Index Terms*—Software Testing, Web Services, Web Service testing, WSDL, XML Generation, XSD, XML.**

## I. Introduction

Automatic generation of XML data can reduce the time and effort needed to test and maintain a Web service (WS). The Web Service Definition Language (WSDL) contains a set of definitions that describe a web service. An XML Schema Document (XSD) can be extracted from the WSDL and used to facilitate the automation of web service testing. Currently there are few tools available that can automatically generate test inputs from XSDs for the testing of WS, the most notable being ToXGene [1] and TAXI/WS-TAXI [2, 3]. Our proposed solution is an automated tool for the generation of WS test inputs from an XSD or a WSDL. Although the tool can be automated, it maintains meaningful control for the tester.

The tool takes a WSDL document as input. The XSD is then extracted from the WSDL and a context-free grammar is created. Using the grammar we create XML template documents using a string constraint solver, each document having a unique structure. The data values are inserted after the XML structure has been generated. The output of the system is a variable sized set of XML documents that validate against the XSD.

The remainder of the paper is organized as follows: Section II provides a summary of related works, Section III provides an overview of our approach starting from the WSDL document to the production of the final test inputs, Section IV outlines the WS-specific modifications present in the tool, in Section V we compare our tool with other similar tools and we conclude with future work in Section VI.

## II. Relevant Work

### A. XML Generators

There are many other tools that can be utilized for the generation of XML Documents, but few are useful for the purpose of automated test generation. All of the following tools require manual work in order to generate XML documents and most cover a small subset of the entire XSD. Tools currently available include: ToXGene [1], TAXI [2], XMLXIG [4], SunXMLGenerator [5], EJB Source Generator [6], and WSDLTest [7]. There is a plethora of other tools that generate a single XML document from an XSD but they are not useful in a software testing context [8]. The two most immediate candidates for test generation are TAXI [2, 3] and ToXGene [3]. Both these tools generate XML data from an XSD but cannot be deployed as a fully automated tool.

ToXGene requires a Template Specification Language (TSL) document as input and will output a set of XML documents that conform to the TSL. ToXGene has some very useful characteristics for the generation of representative data values. ToXGene uses probability distributions for element occurrences and data values. It can cache values for inter-XML instance value sharing and it can read external XML documents to retrieve values. It is difficult, however, to control the structure of the data [8]. ToXGene requires the manual creation of a TSL document. The TSL is a hybrid document containing both XSD and ToX notations. Currently there is no formalized method for the automated generation of TSL documents from an XSD— hence ToXGene requires a large degree of manual interaction.

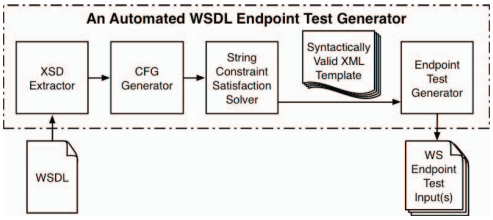
TAXI uses a category partition method in order to generate XML Documents [9]. It creates a separate XML sub-schema for each possible structure. Since the number of combinations can become intractable, they attempt to reduce the number of combinations using weighted elements. TAXI automates the structural generation of the XML but poorly represents diversity in the leaf nodes, *i.e.* the data fields.

### B. Constraint Satisfaction Problem Solvers

Constraint satisfaction problem (CSP) solvers provide a method to create an XML document that satisfies a grammar. There are many different solvers available from the artificial intelligence community. The tools we have explored include: Z3-str [10], KALUZA [11], HAMPI [12], STRSOLVE [13], SCSP [14]. Our WS test input generation tool is designed to separate grammar extraction from CSP-specific input format, *i.e.* the tool is not tied to any specific CSP solver. We currently employ HAMPI [12] to generate valid XML documents because of its simplicity and the understandability of its input grammar. Furthermore HAMPI is well-understood within the software testing community and has been shown effective for test generation [12, 15].

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.85

Fig. 1. The Test Generation Process




## III. The Proposed Test Input Generator

The proposed tool takes a WSDL document as its input. The output is a set of XML documents that validate against the XSD alongside all pertinent information to test the endpoints of a WS. The automated test generation process consists of the modules presented in Fig. 1. Each module identified in Fig. 1 is explained in the following subsections.

### *A. XSD Extractor*

The XSD Extractor takes a WSDL as input and extracts the XSD for processing by the CFG Generator. The XSD is extracted from the WSDL through the use of an Extensible Stylesheet Language Transformation (XSLT) preserving all XML namespace definitions. The validity of the XSD is also ensured. An example of an XSD extracted from a WSDL is shown in Fig. 2. This XSD will be used for all subsequent examples.

### *B. CFG Generator*

The CFG Generator accepts any XSD and generates an equivalent context-free grammar (CFG). The grammar represents the set of all possible inputs to the WS endpoint. The CFG is formatted for use in a string CSP solver.

Fig. 2. A Sample XSD Document

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.java2s.com"
  xmlns="http://www.java2s.com" elementFormDefault="qualified">
  <xsd:element name="Books">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Book" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Title" type="xsd:string"/>
              <xsd:element name="Author" type="xsd:string"
               maxOccurs="unbounded"/>
              <xsd:element name="Date" type="xsd:date"/>
              <xsd:element name="ISBN" type="ISBNRestriction" />
              <xsd:element name="Publisher" type="xsd:string"/>
            </xsd:sequence>
            <xsd:attributeGroup ref="BookAttributes"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:attributeGroup name="BookAttributes">
    <xsd:attribute name="Category" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="autobiography"/>
          <xsd:enumeration value="non-fiction"/>
          <xsd:enumeration value="fiction"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="InStock" type="xsd:boolean"
     default="false"/>
    <xsd:attribute name="Reviewer" type="xsd:string"
     default="none"/>
  </xsd:attributeGroup>
  <xsd:simpleType name="ISBNRestriction">
    <xsd:restriction base="xsd:integer">
      <xsd:totalDigits value="13"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

The CFG Generator can process a large subset of the XSD definition and automatically creates an equivalent grammar. The CFG generator processes the entire XSD with the exception of key references and the inclusion of external elements or attributes. The generated grammar is able to model the constraints specified in the XSD for simple type elements. Facets (XSD constraints) on elements or attributes are included in the grammar as annotated terminals that forward the facets to the Endpoint Test Generator. For example, the *totalDigits=13* facet in the simple type *ISBNRestriction* is encoded in the generated grammar. The constraint is realized by the *ISBNRestriction* production rule with the type, *integer*, followed by the facet *TOTDIGITS*, and the value, *13*, as shown in Fig. 3. All XSD complex types can be modeled in the grammar including *sequence*, *all* and *choice*. All complex content structures: *extension* and *restriction* can also be modeled. Fig. 3 shows the output grammar resulting from the CFG Generator applied to the XSD in Fig. 2. The grammar also defines the overarching structure required for the creation of an XML document that will validate against the given XSD.

The CFG Generator has self-imposed limitations that reduce the size of the grammar. For example, in cases where the *maxOccurs* facet is above a predefined constant, *k*, it is substituted with a Kleene operator. The *all* aggregation provides elements in any permutation but the CFG Generator limits *all* aggregations to a fixed ordering. With knowledge of the grammar, a tester can make simple changes without impacting the validity of the generated XML documents, giving some control over the structure and the data present in the final set of XML tests, as explored further in Section IV.

In order to generate a grammar from an XSD, a root element is needed. Since the output XML can be valid against

Fig. 3. The context free grammar created by the CFG Generator

```
var v:1000;
cfg Target := " xmlns= 'http://www.java2s.com' ";
cfg Books :=  "<Books" Target ">" SEQUENCE0 "</Books>" ;
cfg SEQUENCE0 := Book ;
cfg Book := (BookNUM)+ ;
cfg BookNUM :=  "<Book" BookAttributes ">" SEQUENCE1 "</Book>" ;
cfg SEQUENCE1 := Title NEXT0;
cfg NEXT0:=Author NEXT1;
cfg Title :=  "<Title>" string "</Title>" ;
cfg NEXT1:=Date NEXT2;
cfg Author := (AuthorNUM)+ ;
cfg AuthorNUM :=  "<Author>" string "</Author>" ;
cfg NEXT2:=ISBN Publisher ;
cfg Date :=  "<Date>" date "</Date>" ;
cfg ISBN :=  "<ISBN>" ISBNRestriction "</ISBN>" ;
cfg ISBNRestriction := integer "{[TOTDIGITS 13]}";
cfg Publisher :=  "<Publisher>" string "</Publisher>" ;
cfg BookAttributes :=  Category NEXT3;
cfg NEXT3:= InStock NEXT4;
cfg Category := " Category =" "'"SIMPLE0"'";
cfg SIMPLE0 := string "{&autobiography&&non-fiction&&fiction&}";
cfg NEXT4:= Reviewer;
cfg InStock :=  | " InStock =" "'"boolean"'";
cfg Reviewer :=  | " Reviewer =" "'"string"'";

cfg string := " STRING " ;
cfg date := " DATE " ;
cfg integer := " INTEGER " ;
cfg boolean := " BOOLEAN " ;

assert v in Books;
```

the XSD using different root elements a target root element must be specified. The WSDL contains the root elements for the WS endpoints. In many cases multiple WS endpoints are present in the same WSDL and all endpoints have an explicitly defined input type. The CFG Generator will be executed once for each endpoint present in the WSDL resulting in the generation of one or more grammars.

### C. CSP Solver

The CSP Solver accepts the grammar produced by the CFG Generator as an input. The output of the CSP Solver is a set of structurally valid XML templates with additional annotations. The annotations specify the type of data, size restrictions, or other facets defined in the XSD. Each XML template has a unique structure. In some cases there may be only one possible structure; others have an infinite set of possible structures. For example, elements with unbounded *maxOccurs* result in an infinite set of templates.

Fixed sized (number of characters) XML templates are generated by the string CSP solver. The template size is included in the grammar (*var v: 1000* in Fig. 3); The CSP Solver modifies the grammar to produce XML templates for a predefined range of sizes. The annotated XML templates are generated up to a fixed size limit, *S*; this solves the problem of the infinite number of different structures.

Some drawbacks to using HAMPI are that there can only be two non-terminals in each production rule so longer rules are split into multiple rules. Because of these limitations extra grammars must be added for complex types that include many elements. For example, the element *book* in Fig. 2 contains a sequence with 5 elements. In the resulting grammar, Fig. 3, the sequence labeled *SEQUENCE1* is divided into 4 different production rules, with the *NEXT0, ..., NEXT3* rules continuing the sequence. This technique is used whenever necessary, for example, the *all* aggregation.

### D. XML Templates

Anecdotally we found that HAMPI was effective at generating the structure of an XML document but was ill suited for data generation. For example, HAMPI generated data exhibited repetitious values and poor coverage of the input space. To overcome this limitation we propose that the final generation of tests be conducted using established software testing methodologies such as, Adaptive Random Testing [16], Quasi-Random Testing [17], Combinatorial Testing [18], etc. The current implementation provides random generation of data or data retrieved from files. Fig. 4 presents an example of an annotated XML template. All the necessary XML structure is present along with the appropriate tags. A large set of these templates is generated from each grammar and they are used as input into the Endpoint Test Generator.

### E. Endpoint Test Generator

The Endpoint Test Generator accepts the annotated XML templates as input and produces tests for each WS Endpoint present in the WSDL. The current implementation of this module employs random generation for data values. Data can also be retrieved from files when non-random data is desired. All 44 atomic data types can be randomly generated and validated against the XSD although data integrity is not enforced. Our current prototype cannot generate *key references, ID, IDREF* and *ENTITY* that validate against the XSD. The data is generated according to all facets in the XSD aside from the *pattern* and *date* type constraints.

Tests are generated according to constraints annotated by the XML templates. The generator is aware of all implicit data constraints for all 44 atomic types, such as *int*, which is a signed 32-bit value. Each atomic type, shown in Fig. 4 (denoted by all capital letters), is replaced with a constrained random value by the Endpoint Test Generator. Fig. 5 presents the resulting XML document produced by the Endpoint Test Generator applied to the XML template in Fig. 4. The Endpoint Test Generator outputs a set of XMLs that validate against the XSD and are valid for one of WS endpoints. A variable number of tests can be generated from each XML Template and for each WS endpoint.

Fig. 4. Example of a XML template document generated by HAMPI

```
<Books xmlns= 'http://www.java2s.com' >
  <Book
   Category =' STRING {&autobiography&&non-fiction&&fiction&}'>
    <Title> STRING </Title>
    <Author> STRING </Author>
    <Date> DATE </Date>
    <ISBN> INTEGER {[TOTDIGITS 13]}</ISBN>
    <Publisher> STRING </Publisher>
  </Book>

  <Book InStock =' BOOLEAN '
   Category =' STRING {&autobiography&&non-fiction&&fiction&}'>
    <Title> STRING </Title>
    <Author> STRING </Author>
    <Author> STRING </Author>
    <Date> DATE </Date>
    <ISBN> INTEGER {[TOTDIGITS 13]}</ISBN>
    <Publisher> STRING </Publisher>
  </Book>

  <Book InStock =' BOOLEAN ' Reviewer =' STRING '
   Category =' STRING {&autobiography&&non-fiction&&fiction&}' >
    <Title> STRING </Title>
    <Author> STRING </Author>
    <Date> DATE </Date>
    <ISBN> INTEGER {[TOTDIGITS 13]}</ISBN>
    <Publisher> STRING </Publisher>
  </Book>
</Books>
```

Fig. 5. Example of a test case generated by the Endpoint Test Generator

```
<Books xmlns= 'http://www.java2s.com' >
  <Book Category ='autobiography'>
    <Title>7:y#B_5uJ </Title>
    <Author>sFVVGR2(g0# </Author>
    <Date>7616-01-22 </Date>
    <ISBN>1176157885 </ISBN>
    <Publisher>@u^afjX6OeBMpZkuT </Publisher>
  </Book>

  <Book Category ='autobiography' InStock ='1 '>
    <Title>JWP1z31 </Title>
    <Author>k1B </Author>
    <Author>#)zlygI2.9Cfy0O </Author>
    <Date>2066-02-04 </Date>
    <ISBN>129820007175 </ISBN>
    <Publisher>Sfny_BO-EzK$1jv@1u </Publisher>
  </Book>

  <Book Category ='fiction' InStock ='false '
   Reviewer ='aEOrOCWKC7Dr '>
    <Title>*q8RoGuno </Title>
    <Author>mX1yY </Author>
    <Date>3925-09-19 </Date>
    <ISBN>1520358272 </ISBN>
    <Publisher>sWR:WIgx1^;9T#tj22 </Publisher>
  </Book>
</Books>
```

## IV. WS-Specific Testing Manipulations

Due to the nature of automated testing, which relies on the random generation of testing data, finer-grain test generation control is needed. Generic forms of control are needed to support testing activities throughout the generation process. The proposed tool supports tester controlled data generation through external file constraints, additional data value constraints, and structural manipulations. All of the changes discussed do not affect the final XML validity.

### *A. External File Constraints*

Although the chance of generating test values meaningful to the WS is non-zero, the likelihood of occurrence can render the test data ineffective. WS-specific data can be provided through the manual inclusion of external file constraints allowing for the selection of meaningful data during the Endpoint Test Generation step.

An external file constraint can be added to the grammar produced by the CFG generator in the same way other XSD facets are modeled (see *TOTDIGITS* in Fig. 3). File constraints are denoted by the *{[FILENAME filename]}* terminal, where *filename* is the uri of the external file. The file contains line-separated values that are chosen randomly during the Endpoint Test Generation step. The additional constraint is defined once in the grammar and propagates through the rest of the system: XML templates, generated XML tests, and finally the WS endpoint tests. Furthermore, grammar substitutions can be defined before the process begins, allowing for the automated generation of values post WS-specific configuration.

### *B. Additional Data Value Constraints*

Often the constraints contained within an XSD are insufficient for representing the underlying application. There is a need to allow for additional restrictions to be placed on input values. The proposed tool supports the inclusion of additional tester-defined constraints through grammar manipulations mid-process and/or through predefined WS-specific configurations.

Constraints on simple type data can be added into the grammar to control the values that are generated. For example, in Fig. 3 the *ISBNRestriction* simple type must have 13 or less digits as defined in the XSD, if the tester wants less digits then it can be changed in the grammar. Additionally, enumeration constraints can be added or removed in the grammar. The *Category* element in the XSD (Fig. 2) has a local simple type so it is given the name *SIMPLE0* in the grammar (Fig. 3). This simple type has 3 possible enumerations: *fiction, non-fiction,* and *autobiography*. For example, if the tester wants to exclude the *fiction* category in the final tests then they can simply remove that enumeration. Also, for any elements that do not have enumerations, they can be added to the grammar to ensure a type has the desired set of values.

Tester defined WS-specific constraints can be defined including: *enumerations*, *maxLength*, *minLength*, *length*, *maxInclusive*, *maxExclusive*, *minInclusive*, *minExlusive*, *fractionDigits* and *totalDigits*. The constraints can be applied to all atomic data types. All of these constraints are present in the XSD domain, but can be added to provide fine-grain control over the XML data generated.

### *C. Structural Manipulations*

The structure of generated XML documents can be controlled through grammar manipulations. With the use of these changes a tester can limit the number of occurrences of an element, limit choices between elements and ensure an elements/attributes presence in all template documents, and control permutations of elements.

For example, anywhere a Kleene star or Kleene plus is used there is the possibility of repetitive elements in the generated XML. The number of elements can be limited by changing the grammar to a finite number of elements rather than the Kleene operator. The tester could define the exact number of desired occurrences of the element or define a set of desired number of occurrences using the *or* operator.

Anywhere the *or* operator is used a change can be made in the grammar if desired. The *choice* complex type is modeled in the grammar using the *or* operator. A tester can limit these choices by removing some from the grammar. An attributes presence in the final XML can also be ensured through simple grammar modification. Both the attributes *inStock* and *Reviewer* are not required, so the *or* operator is used in the grammar to specify that the attribute can be *null*, or it can be present in the element. If an attribute is desired to be present by the tester, this *or* operator can be removed to ensure that attribute is always in the final XML.

The *all* aggregation is modeled in the grammar as a fixed ordering even though any possible ordering is allowed. Modeling all possible orderings can quickly become intractable depending on the number of elements in the all aggregation. A single ordering is used to limit the size of the grammar. If a desired ordering is required then a simple change can be made in the grammar to specify the desired ordering.

## V. Comparison To Existing XML Generation Tools

A comparison between our tool, TAXI and ToXGene is presented in Table I. The comparison details the most important parts of automatic test case generation with regards to automation, control, and functionality. In Table I, our tool exhibits the highest level of automation through the use of the WSDL document to automatically generate the WS endpoint test inputs. Test inputs contain the testing data alongside the WS endpoint: URL, port, message, operation, namespace, etc. The generated test documents are self-contained packages that can be executed against a WS without additional manual packaging. Even with this level of automation, meaningful control is retained. Our tool works implicitly with XSD documents while ToXGene requires a manual conversion from XSD to TSL. Our tool exhibits a higher level of XSD coverage than both TAXI and ToXGene allowing for a much larger set of all XSDs to be analyzed and used to facilitate automatic test input generation.

## VI. Conclusion & Future Work

In conclusion, the proposed test generation tool can provide an automated method to generate test inputs for a generic WS from a WSDL document. A set of inputs can be

TABLE I. COMPARISON OF THE PROPOSED TOOL VS. TAXI VS. TOXGENE

| | The Proposed Tool | TAXI | ToXGene |
|---|---|---|---|
| **Ease of Automation** | Fully automated with random data | Semi-Automated | Semi-Automated |
| **Manual Interactions Required** | *Optional:*<br>WS-specific configuration available | XSD Extraction, Root Element, and Generation Options Required<br>No Post Generation Packaging | Automated when processing TSL<br>XSD Extraction and Manual XSD to TSL Required<br>No Post Generation Packaging |
| **Tester Control** | Tester supplied data and structure customizations | Control over structure but not data. | Extensive control over data, limited control over structure. |
| **Namespace Handling** | Automatically maintains all namespaces | Maintains namespaces, but requires manual configuration | Does not maintain namespaces |
| **XSD Coverage** | Cannot generate *key references, ID, IDREF* and *ENTITY* that validate against the XSD.<br>The *pattern* constraint and *date* type constraints are not currently implemented. | Lack of support for nested *choices*, *attribute groups*, *any*, *union*, *extension*, *IDREF*, *ID*, *groups*, *enumerations*.<br>Unaware of implicit data constraints for all 44 atomic data types. | None, requires manual conversion from XSD to TSL. |

generated with no manual interaction. Although the tool can be fully automated, control for the tester is still available through external datasets and pre-defined WS-specific configurations. In comparison with existing solutions, our tool matches or exceeds in automation, XSD coverage, and tester control.

Future work includes the incorporation of multiple CSP solvers allowing for an empirical evaluation of the use of various CSPs in the software testing context. Further work will focus on an empirical evaluation of the proposed testing tool against TAXI and ToXGene using WS-MAT [19] as the benchmark for comparison.

REFERENCES


[1] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. Lyons, "ToXgene: An extensible template-based data generator for XML," in *IN WEBDB*, 2002, pp. 49–54.

[2] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "TAXI–A Tool for XML-Based Testing," in *29th International Conference on Software Engineering - Companion, 2007. ICSE 2007 Companion*, 2007, pp. 53–54.

[3] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini, "WS-TAXI: A WSDL-based Testing Tool for Web Services," presented at the International Conference on Software Testing Verification and Validation, 2009. ICST '09, 2009, pp. 326–335.

[4] "XML-XIG: XML Instance Generator (XIG)." [Online]. Available: http://xml-xig.sourceforge.net/. [Accessed: 02-Jul-2014].

[5] "Sun Multi-Schema Validator," 02-Jul-2014. [Online]. Available: https://java.net/projects/msv/sources/svn/show/trunk?rev=1827. [Accessed: 02-Jul-2014].

[6] "EJBGen : EJB Source Generation Library." [Online]. Available: http://ejbgen.sourceforge.net/. [Accessed: 02-Jul-2014].

[7] H. M. Sneed and S. Huang, "The design and use of WSDL-Test: a tool for testing Web services," *J. Softw. Maint. Evol. Res. Pract.*, vol. 19, no. 5, pp. 297–314, Sep. 2007.

[8] S. Cohen, "Generating XML Structure Using Examples and Constraints," *Proc VLDB Endow*, vol. 1, no. 1, pp. 490–501, Aug. 2008.

[9] A. Bertolino, J. Gao, E. Marchetti, and A. Polini, "Automatic Test Data Generation for XML Schema-based Partition Testing," presented at the Second International Workshop on Automation of Software Test , 2007. AST '07, 2007, pp. 4–4.

[10] Y. Zheng, X. Zhang, and V. Ganesh, "Z3-str: A Z3-based String Solver for Web Application Analysis," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2013, pp. 114–124.

[11] V. Ganesh and A. Kiezun, "Kaluza String Solver." [Online]. Available: http://webblaze.cs.berkeley.edu/2010/kaluza/. [Accessed: 21-Oct-2013].

[12] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst, "HAMPI: A Solver for String Constraints," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, New York, NY, USA, 2009, pp. 105–116.

[13] P. Hooimeijer and W. Weimer, "StrSolve: solving string constraints lazily," *Autom. Softw. Eng.*, vol. 19, no. 4, pp. 531–559, Dec. 2012.

[14] M. Janzen, M. Horsch, and E. Neufeld, "Camera Selection Using SCSPs," in *Proceedings of the 2008 Conference on Future Play: Research, Play, Share*, New York, NY, USA, 2008, pp. 252–253.

[15] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *SIGPLAN Not.*, 2008, vol. 43, pp. 206–215.

[16] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal Voronoi Tessellations—A New Approach to Random Testing," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 163–183, Feb. 2013.

[17] T. Y. Chen and R. Merkel, "Quasi-Random Testing," *IEEE Trans. Reliab.*, vol. 56, no. 3, pp. 562–568, Sep. 2007.

[18] R. Kuhn, R. Kacker, Y. Lei, and J. Hunter, "Combinatorial Software Testing," *Computer*, vol. 42, no. 8, pp. 94–96, 2009.

[19] A. Martens and A. F. Tappenden, "A Benchmark for Automated Web Service Testing," presented at the 1st CCWSR Meeting, Edmonton, Canada, 2014.