XMLMate: Evolutionary XML Test Generation

Nikolas Havrikov · Matthias Höschele · Juan Pablo Galeotti · Andreas Zeller
Saarland University, Saarbrücken, Germany
{havrikov, hoeschele, galeotti, zeller}@st.cs.uni-saarland.de

ABSTRACT

Generating system inputs satisfying complex constraints is still a challenge for modern test generators. We present XMLMATE, a *search-based* test generator specially aimed at XML-based systems. XMLMATE leverages program structure, existing XML schemas, and XML inputs to generate, mutate, recombine, and evolve valid XML inputs. Over a set of seven XML-based systems, XMLMATE detected 31 new unique failures in production code, all triggered by system inputs and thus true alarms.

Video: `http://youtu.be/-yKom5mbft0`

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*

General Terms

Reliability, Verification

Keywords

Test generator, search-based testing, evolutionary algorithms, XML

1. INTRODUCTION

One of the major advances in testing of the past decade is the concept of *automatic test generation:* Not only do we want to *execute* tests automatically, but also to *generate* them automatically. Given an arbitrary program, the idea is to generate and execute tests to automatically reveal failures. The assumption is that executing a program is cheap, even if we execute it thousands of times.

Automatic test generation is not without challenges, though. Its main problem is long-standing: How can we obtain a sufficient *coverage* of the program and its behavior? Let us consider the following example. The *FreeDots*[1] tool renders musical scores as *Braille music* suitable for blind users. Its input comes in MusicXML format, an XML representation of musical scores.

[1] `http://delysid.org/freedots.html`

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.
FSE '2014 Hong Kong, China
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$15.00.

```
private void transcribeSection(final Part part, final Section section,
                               final int sectionNumber,
                               final boolean numbering) {
    final int staffCount = section.getStaffCount();
    for (int staffIndex = 0; staffIndex < staffCount; staffIndex++) {
      final Staff staff = section.getStaff(staffIndex);

      if (transcriber.getCurrentColumn() > 0) transcriber.newLine();

      if (numbering && staffIndex == 0) {
#       transcriber.printString(new UpperNumber(sectionNumber));
#       transcriber.spaceOrNewLine();
#       transcriber.printString(
#            new LowerRange(section.getFirstMeasureNumber(),
#                           section.getLastMeasureNumber()));
#       transcriber.spaceOrNewLine();
      } else {
        transcriber.indentTo(2);
      }
      // more code...
}
```

Figure 1: Random XML generators struggle to cover central parts of the *FreeDots* code (marked with "#")

If we wanted to test the *FreeDots* system, a simple and straightforward strategy would be to use a black-box random XML generator. Unfortunately, functions such as `transcribeSection()` shown in Figure 1 have very specific *conditions* that are not easily satisfied by a random input, resulting in uncovered alternatives.

To cover the *FreeDots* code, we thus (a) need inputs that would be expected by the method—that is, XML trees that satisfy the MusicXML rules; and (b) generator techniques that are aware of code structure to reach testing goals.

In this paper, we present *XMLMATE,* a search-based test generator that generates, mutates, recombines, and evolves XML inputs. As shown in Figure 2, XMLMATE draws on three sources of information:

1. XMLMATE is *search-based*, systematically evolving an input population to maximize coverage; it thus is driven by code and embedded conditions.

2. XMLMATE uses XML schemas, which help ensuring *structural and syntactic validity* of the generated inputs.

3. XMLMATE can use existing XML inputs (from tests or real executions), mutate and recombine them to obtain both realistic and novel inputs.

2. SEARCH-BASED XML TESTING

XMLMATE is a *search-based* test generator. As sketched in Figure 2, it either takes a population of sample XML inputs or randomly generates one, and then systematically *evolves* this population over several generations to reach the desired coverage goal. To

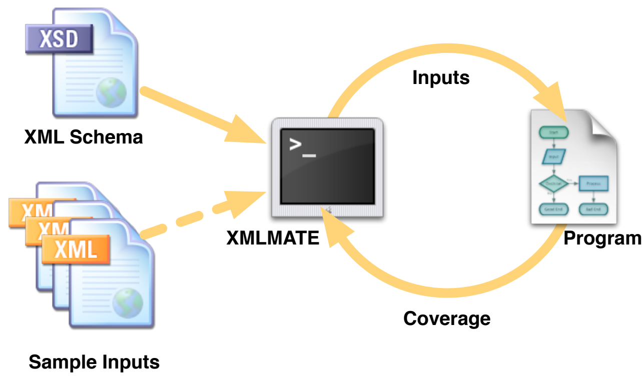

Figure 2: How XMLMATE works. From an XML schema and, optionally, sample XML inputs, XMLMATE feeds the program under test with generated inputs, which XMLMATE then evolves in order to maximize coverage.

evolve inputs it applies mutation and crossover operators to individuals and favors the procreation and survival of the fittest according to a fitness function geared towards obtaining coverage and triggering exceptions.

XMLMATE is built on top of the EVOSUITE search-based testing framework [2]. In contrast to EVOSUITE, XMLMATE does not generate unit tests, but XML inputs; hence, it uses XML inputs as chromosomes and XML manipulations as mutation operators. However, the genetic algorithm itself, as well as the fitness function, are adapted directly from EVOSUITE.

XML Trees In XMLMATE, the central data structure is an *XML tree,* the standard representation of XML inputs. In Figure 3, an XML-processing program takes a textual XML input and parses it into an XML tree representation—in this case, MathML, an XML representation of mathematical notations, capturing both structure and content. A MathML interpreter would process the tree recursively to render it as a mathematical formula, as shown.

Mutation and Instantiation The schema instantiation engine of XMLMATE is responsible for the random generation of initial populations and the basis for all mutation operators. We implemented a mutation operator that randomly mutates elements in a way which guarantees that the resulting value is schema-valid for its type. As an example, consider Figure 4, where we mutate an operator in a MathML formula. Here, the `divide` and `power` operators both are math operators, and thus of a matching type; this is a standard mutation that XMLMATE could generate.

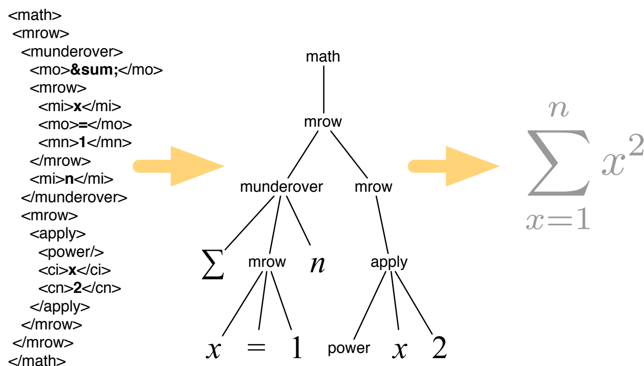

Figure 3: A MathML formula as XML source, as XML tree, and the rendered presentation

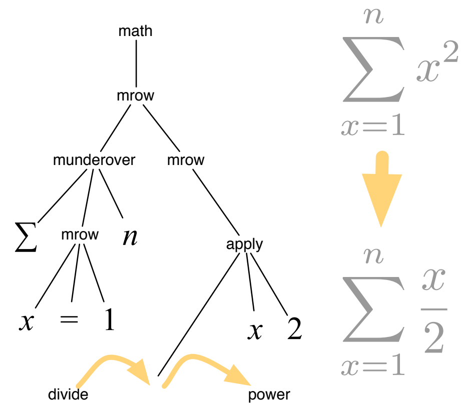

Figure 4: Mutating a MathML formula from `power` to `divide`

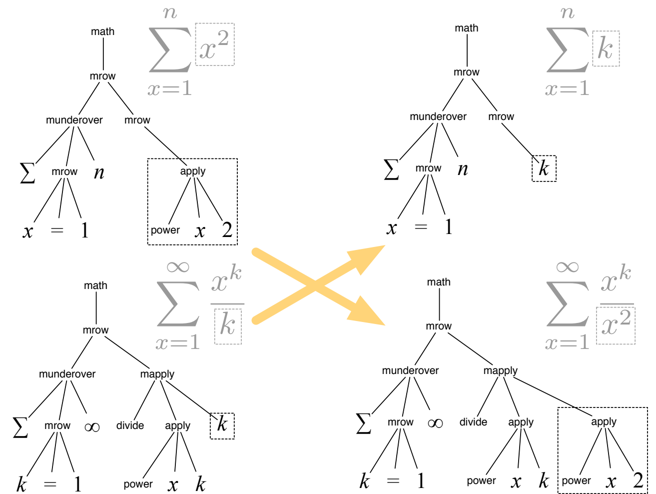

Figure 5: Crossing over x^2 and k in two MathML formulas

Besides mutation, a test generator also needs *instantiation:* that is, the creation of entirely new inputs. This is implemented as the mutation of an empty dummy element of the root type defined in the schema—in other words, a mutation from nothing to something.

XMLMATE mutates XML elements according to its type. For example, if the given XML element is of a a simple type, and its schema definition contains a pattern restriction (i.e., a regular expression which defines all valid values), XMLMATE uses the *Automaton* library [7] to instantiate these regular expressions.

Crossover Like many genetic algorithms, XMLMATE not only *mutates* inputs, but also *recombines* them. This *crossover* operation is implemented on two levels. On the *chromosome* level, the crossover operator can swap entire XML inputs within a population. This does not change the individual XML inputs, and is a standard crossover operation to evolve test suites. The more interesting crossover operation takes place on the *XML tree* level. Here, the crossover operator swaps entire XML subtrees, as shown in Figure 5. This is not only direct and intuitive; XMLMATE can also preserve validity of the inputs according to the schema, something that is hard to achieve in general.

Fitness Function XMLMATE uses the extended fitness function implemented in EVOSUITE, formally defined in [1]. This extended fitness function optimises towards achieving code coverage and

Table 1: XML-based systems under test

Subject	Version	#Classes	#SLOC	Format	Schema Size	Source
ROME	1.0	117	10,624	Atom	11k	https://rometools.jira.com/
JEuclid	3.1.9	256	15,291	MathML	124k	http://jeuclid.sourceforge.net/
Freedots	20101116	158	8,865	MusicXML	303k	http://delysid.org/freedots.html
Apache Batik	1.7	1,446	178,447	SVG	132k	http://xmlgraphics.apache.org/batik/
SVG Salamander	1.0	176	17,688	SVG	132k	https://svgsalamander.java.net/
SVG Image	1.0	1	469	SVG	132k	http://goo.gl/F4Ivaw
FlyingSaucer	R8pre2	294	41,619	XHTML	68k	http://code.google.com/p/flying-saucer/

triggering a high number of unique exceptions. XMLMATE leverages this definition by targeting the entire system instead of a single compilation unit.

3. THE XMLMATE PROTOTYPE

Let us briefly describe some of our experience from executing the XMLMATE prototype on seven publicly available XML-based systems listed in Table 1: *ROME* is an Atom and RSS processing and generation library. *JEuclid* is a rendering and conversion library for MathML. *Freedots* allows blind users to view MusicXML as braille notation and offers MIDI playback functionality. *Apache Batik* and *SVG Salamander* are libraries for SVG modification, rendering and conversion. *SVG Image* is a simple Java application that renders SVG images. *FlyingSaucer* is a XHTML rendering library that also supports SVG.

3.1 Sample and Random Seeding

By default, XMLMATE evolves a randomly generated initial population (i.e., no sample inputs are used). For each subject, we manually selected a population size (i.e., number of chromosomes) and a maximum number of individuals per chromosome. Overall, the size of the obtained random initial populations ranged from 15 to 25 chromosomes, and the number of XML inputs per chromosome ranged from 10 to 20 depending on subject.

As previously stated, XMLMATE could also evolve a user-selected initial population. In order to execute XMLMATE using sample initial populations, we composed a set of sample inputs for each evaluation subject. These sets were obtained directly from the sample documents and test suites included in the source packages of the individual test subjects. The samples seeds for MathML, XHTML and SVG were taken from official test suites. For Atom we crawled the web for available feeds; for MusicXML, we converted publicly available sets of classical music MIDI files to MusicXML using *Musescore*[2]. Each sample initial population consisted of arbitrarily chosen input sets matching the size of their randomly seeded counterpart.

3.2 Achieved Coverage

We executed each configuration of XMLMATE with a time bound of 6 hours. Due to the stochastic nature of the genetic algorithm, we repeated each execution 10 times.

Table 2 exhibits the achieved branch coverage for each configuration on the selected XML-based systems. When comparing the two configurations of XMLMATE (i.e., random seeds and sample seeds), results are mixed: On two subjects (*JEuclid* and Batik), coverage increased; on two (Freedots and ROME), coverage decreased; and on the other three, coverage is basically unchanged. A further analysis of the effect size on each subject indicates that, in terms of coverage, there is no difference between any of the XMLMATE configurations.

[2] http://musescore.org

3.3 Failures Found

At the end of the day, any testing and analysis tool will be valued by the defects it finds. Exceptions or crashes at the system level can serve as partial oracles. In contrast to unit-level, undeclared exceptions at the system-level rarely denote an improper usage of the system, but a valid interaction or input that is not correctly handled. In order to measure failures on our selected subjects, we count the number of unique pairs composed by the exception type and the program line where the exception was thrown (i.e., the topmost element in the stack trace). We do not consider an exception a failure if the exception type is explicitly declared in the test driver. For example, for *JEuclid*, `IOException` and `SAXException` types are both ignored since they are declared exceptions in `parseFile()`, `render()` and `convert()`.

Table 2 also summarises the failures found by XMLMATE along all executions. The tests generated by XMLMATE triggered more failures than any of the other representatives of the state-of-the-art, exposing problems in six out of our seven subjects. With random seeds, XMLMATE triggered a total of 21 unique failures—that is, failures that differ by the topmost element of the stack trace and exception type, and thus imply a similar number of fixes. Starting with sample inputs and evolving these further raises the number of unique failures to 31, highlighting the potential value of sample inputs for finding failures at system-level.

XMLMATE triggered 31 unique failures across six out of seven subjects.

The most frequent Java exception types found by XMLMATE were `NullPointer` (5), `IllegalArgument` (3), `Runtime` (2), `IndexOutOfBounds` (2), followed by `ArrayIndexOutOfBounds`, `NoSuchElement`, `StringIndexOutOfBounds`, `StackOverflow`, `NumberFormat`, and `IllegalPathState`, each occurring once; the remaining exception types are specific to the respective program.

It is important to note that each of these failures can be reproduced via the respective XML input; consequently, all of them imply a potential denial of service attack on a system that processes such XML inputs. As long as we assume that the "external" input actually is external, that is, under control of a third party, all alarms generated by XMLMATE are true, by construction.

Incidentally, our investigations also led to the discovery of defects in other programs. While investigating the structure of XML files generated by our tests, we found one small file which causes the *Firefox* web browser to consume the entire available main memory and then shut down. Another file caused the *Opera* web browser to crash with an `AccessViolation` error. Finally, some *JEuclid* tests caused fatal crashes of the Java 1.6 virtual machine both on Linux and Windows machines, which could indicate a potential security vulnerability of the Java virtual machine. All issues have been reported to the respective software authors.

All failures reported by the system-level tools could be triggered in deployed code.

Table 2: Results of executing XMLMATE on the XML-based systems

Subject	Average Branch Coverage		Total triggered unique failures	
	XMLMATE (sample seeds)	XMLMATE (random seeds)	XMLMATE (sample seeds)	XMLMATE (random seeds)
ROME	32%	36%	1	1
JEuclid	51%	48%	1	3
Freedots	36%	37%	10	12
Apache Batik	5%	3%	3	4
SVG Salamander	16%	16%	3	4
SVG Image	40%	40%	3	4
FlyingSaucer	24%	24%	0	0
Total	—	—	21	31

4. RELATED WORK

Search-based testing is a successful unit test generation technique. After the pioneering work of Tonella et al. [9], the EVOSUITE framework currently provides an industrial-strength implementation [2]. With EXSYST [5], this idea has been applied to systematically generate interaction sequences for GUI applications to exercise program behavior from a system interface.

Most related to our work is *white-box fuzzing,* as implemented in the SAGE tool [4], introduced as a way to efficiently explore behavior instead of solely relying on entropy. The advantage of XMLMATE over SAGE is that XMLMATE can make use of XML schemas to restrain XML inputs, while SAGE has to infer all constraints through symbolic analysis. Additionally, XMLMATE can also start its seed with entirely random inputs; having sample real inputs is helpful, but not necessary.

Godefroid et al. [3] present a white-box approach that extends SAGE [4] by efficiently guiding input generation using a constraint solver; this work also can use a grammar to constrain inputs. In contrast to XMLMATE (and SAGE proper), this work does not leverage sample inputs to seed test generation.

The mutation of XML instances in the context of testing web services has been implemented previously. Offut, Xu et al. [8, 10] describe a test generation approach using data perturbation in order to mutate XML requests. Lee, Offut et al. [6] use similar techniques to apply mutation analysis in the context of testing web components. These approaches require user interaction and provide no fully automated approach that aims to maximize coverage.

5. CONCLUSIONS

We have introduced XMLMATE, a white-box tool for evolving random or sample initial populations towards higher branch coverage by means of a genetic algorithm. By bringing search-based system testing to XML inputs, XMLMATE improves practical test generation in many ways. Deploying XMLMATE is straightforward: All one needs is a schema with the essential syntactic constraints. As XMLMATE generates inputs at the system level, every failure found can also occur during production; its absence of false alarms is an important advantage over test generators at the unit level.

In our experience using XMLMATE to generate test cases for a selection of XML-based systems, sample seeding seems to have no benefit over random seeding in terms of structural branch coverage, but more failures have been detected by evolving sample inputs instead of randomly generated populations.

The XMLMATE prototype, as well as all the drivers and instructions for reproducing our experiments, are publicly available for download at:

http://www.st.cs.uni-saarland.de/testing/xmlmate/

Acknowledgments. This work was funded by an European Research Council (ERC) Advanced Grant "SPECMATE – Specification Mining and Testing". Kevin Streit, Alessandra Gorla and Eva May provided helpful comments on earlier revisions of this paper. Special thanks go to Gordon Fraser for EVOSUITE support.

6. REFERENCES

[1] G. Fraser and A. Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 2013. To appear.

[2] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, Feb. 2013.

[3] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. *SIGPLAN Not.*, 43(6):206–215, June 2008.

[4] P. Godefroid, M. Y. Levin, D. A. Molnar, et al. Automated whitebox fuzz testing. In *Network Distributed Security Symposium (NDSS)*. Internet Society, 2008.

[5] F. Gross, G. Fraser, and A. Zeller. Search-based system testing: high coverage, no false alarms. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 67–77, New York, NY, USA, July 2012. ACM.

[6] S. C. Lee and J. Offutt. Generating test cases for XML-based web component interactions using mutation analysis. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 200–209. IEEE, 2001.

[7] A. Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. http://www.brics.dk/automaton/.

[8] J. Offutt and W. Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[9] P. Tonella. Evolutionary testing of classes. *SIGSOFT Softw. Eng. Notes*, 29(4):119–128, July 2004.

[10] W. Xu, J. Offutt, and J. Luo. Testing web services by XML perturbation. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10–pp. IEEE, 2005.