2012 IEEE Fifth International Conference on Software Testing, Verification and Validation

Generating String Test Data for Code Coverage

Michael Beyene, James H. Andrews
Department of Computer Science
University of Western Ontario
London, Ontario, Canada
mbeyene,andrews@csd.uwo.ca

***Abstract*—String data has traditionally been difficult for test data generation tools to generate. Of particular concern are strings that conform to a given grammar, since coverage of grammar productions does not guarantee high code coverage or fault-finding ability.**

We address this problem by deriving Java classes from the grammatical categories in the grammar, and then using a collection of deterministic and metaheuristic techniques to generate strings from them. We compare the code coverage of these techniques and the standard conformance test suites for the grammars. We conclude that the various techniques have complementary strengths, and that they can be usefully used in combination.

***Keywords*-Software testing; string data generation; metaheuristic algorithms; structural code coverage;**

I. Introduction

Test data generation in program testing is the process of identifying a set of test data that satisfies a set of given criteria. There are a number of different automated software test data generation approaches, such as randomized generation and symbolic execution. Many published approaches, however, deal only with the generation of numeric data, and do not deal sufficiently with generating character string data. This presents a problem since much software not only processes strings, but processes strings in a particular format.

Our research deals with how to automatically generate character string test data. The goal is to thoroughly test software that accepts and processes strings that conform to a particular grammar. To approach that goal, we first show how the problem can be transformed into a more general form which is amenable to program analysis, by converting the grammar to a set of Java classes. We then compare various deterministic and metaheuristic ways of using these classes to generate the required strings. We also compare these approaches to using conformance test suites.

Our results indicate that these techniques, in particular the conformance test suite and the metaheuristic approaches, are strongly complementary: they achieve similar thoroughness separately, but much more when used together. The techniques together achieved high code coverage, and revealed one previously-unknown fault in a heavily-studied subject program.

In Section II, we review related work. In Section III, we describe the generation of Java classes, referred to as Grammatical Category Objects (GCOs), from a context-free grammar, and discuss their application to the grammars of HTML and XML. In Section IV, we describe in detail the various deterministic and metaheuristic approaches that we compared in our empirical study. In Section V, we give the details of that study. We conclude in Section VI.

II. Background and Related Work

The work most closely related to our work has to do with software testing, generating string input, and randomized unit testing.

A. Software Testing

Software test data selection can be divided into black-box approaches, which use information about the requirements, and white-box approaches, which use information about the code such as how many lines of code were executed by the test cases. This paper takes a combined approach, since we start with a specification of a context-free grammar of a character string (black-box), but also attempt to optimize test data generation to achieve high code coverage (white-box).

Software testing tasks can be divided into groups according to the "level" of the code they test. Unit testing tests the correctness of functions, methods and groups of methods; system testing tests the entire system after all the units have been assembled. This paper focuses on unit testing tasks.

Coverage means different things for specifications and code. A specification element such as a rule in a grammar is said to have been covered if the rule has been used in the generation of some test data. A program element such as a line of code is said to have been covered if there is some test case that causes it to be executed. We take as our primary goal the coverage of program elements, and we use coverage of grammar rules as one means to achieve that.

In this paper, we concentrate on the Java programming language, and use the open-source tool Cobertura [1] to measure line coverage. However, our results generalize to any object-oriented language and any numerical coverage measure.

B. Generating String Input

Purdom [2] was perhaps the first to explore thoroughly the concept of generating data that is acceptable by a context-free grammar. The central problems are not simply the

978-0-7695-4670-4/12 $26.00 © 2012 IEEE
DOI 10.1109/ICST.2012.39

problem of generating any string that conforms, or even the problem of generating a set of strings that cover all productions in the grammar; these problems have been solved. Some of the central problems are:

- How deep should the syntax tree of the string be? This question is independent of production coverage: even a simple grammar, such as for arithmetic expressions, can produce strings with deep syntax trees, and it may be necessary to generate such strings in order to thoroughly test the software that processes them.
- Should all the strings with a syntax tree of a certain depth be generated? If not, how do we select the strings? Beyond a certain depth, many strings will be uninteresting, since they will cause the software under test to repeatedly re-run sections of code without finding new bugs.
- For repeated elements (as, for instance, noted by a Kleene star), how many repetitions are appropriate? Similar considerations to the above are at work here.
- Should strings that do *not* conform to the grammar be generated? If so, how close should they be to conformant strings, and how should they be generated?

Hennessy and Power [3] explored the effect of minimizing a test suite on coverage and effectiveness. They took a large, standard test suite for C++ compilers and removed as many test cases as possible without losing production coverage on the programming language grammar. They found that they could cut the test suite down to a small fraction of its original size, but that this reduced test suite was much less effective at finding faults. Their work dramatically illustrates the need for something beyond rule coverage.

Alshraideh and Bottaci [4] tackled the issue of generating test data for string predicates by using a genetic algorithm. Their work focuses on assembling and comparing various search operators and cost functions, to cover predicates that include string equality, string ordering and regular expressions.

Approaches to test data generation such as symbolic execution [5], [6] attempt to generate test data that achieve high code coverage by following control paths and resolving the constraints that lead to given sections of code. However, the constraints arising from string processing are complex and cannot always be resolved by constraint handlers. Zhao and Liu [7] extend the general symbolic-execution approach of Korel [8] to string data. They define a string predicate as boolean expression that contains at least one character string variable and one string comparison function. Then they define a branch function with respect to the character string predicate that has not taken a particular branch.

Lämmel and Schulte [9] describe a system in which not only grammars, but a set of properties of grammars, nonterminals, and rules can be defined. They show that with a good choice of values of properties, good coverage can be obtained. This work is the most similar to ours, but we use a general-purpose metaheuristic framework to find good choices of values for similar properties, rather than relying on the programmer to find them by hand.

Kieżun et al.'s system Hampi [10] was designed to combine off-the-shelf components to generate string data. A Hampi user defines a context-free grammar (CFG) for a set of strings, and a regular expression (RE) expressing a constraint on a string; Hampi then generates a string that conforms to both the CFG and the RE. To obtain multiple strings – e.g., to cover multiple grammar productions or program paths – the user identifies constraint REs that lead to those productions or paths being covered.

In their study [10], Kieżun et al. used Hampi to generate strings corresponding to security faults, and to generate string constraints that Klee [11] can use for generating inputs to C programs. The longest strings they generated for C program input were 28 bytes. Our study adapts the Nighthawk tool for testing Java programs through Java reflection [12], so to compare our techniques directly to Hampi would require a Java equivalent of Klee. We are currently exploring other ways to use Hampi to generate high-coverage string inputs for Java programs.

C. Randomized Unit Testing

In this paper, randomized unit testing is defined as testing a group of target methods by introducing randomization on the number of method calls to make to a particular method, the sequence of method calls and the values of the parameters of the methods we would like to test. Randomized testing has been shown to be very effective in forcing failures even in well-tested units [13], [14], [15]. However, the thoroughness one can achieve using this testing strategy is heavily dependent on selection of good values for properties controlling the testing. Some of these properties include the number of operations, frequencies of choosing one operation relative to others, and range of arguments that are used during testing [16].

Pacheco et al. [14]'s system Randoop improves the efficiency of randomized unit testing by incorporating a feedback mechanism that will safeguard against the generation of "meaningless" or redundant test input(s). Andrews et al.'s system Nighthawk [12] seeks to guide random testing by using a genetic algorithm to find values of the properties controlling randomized testing that optimize code coverage. In this paper, we use Nighthawk as the basis for our metaheuristic approaches, but we believe that other randomized testing frameworks, such as Randoop, can also be used.

III. GCO Generation

Here we describe how we translate a grammar into a set of Java object classes. Each class represents one category (terminal or nonterminal) in the grammar, so the objects are called Grammatical Category Objects or GCOs. The advantage of translating a grammar into GCO classes is that

```
Expr  →  Number
Expr  →  Expr Op Expr
Expr  →  "(" Expr ")"
  Op  →  "+"
  Op  →  "-"
```

Figure 1. Example grammar. Expr and Op are non-terminals, Number is a terminal, and the double-quoted strings are literals.

we can then use the Java reflection mechanism and general-purpose software testing tools such as Nighthawk to generate instances of them.

A. Definitions

Consider the example grammar of arithmetic expressions in Figure 1. It is based on the terminal symbol Number, the non-terminals Expr and Op, and a number of literals including the left parenthesis `"("` and right parenthesis `")"`. We will use this grammar to illustrate our method of creating GCO classes.

A set of GCO classes for a grammar consists of the following.

1) A class for every terminal and non-terminal symbol. The class should be a subclass of the generic class `GCOElement`, which takes a string as the argument to its constructor, and contains a public method `getString` which returns that string.
2) For every rule in the grammar whose left-hand side is a nonterminal N, a public static method in the class corresponding to N. This method should return a new instance of the class; the parameters should be instances of the terminals and non-terminals appearing on the right-hand side of the rule. The string in the new object should be composed from the parameters and literals in the way that is indicated by the rule.
3) For every terminal symbol T, some public static methods in the class corresponding to T. These methods should return representative examples of T.

Figure 2 shows a GCO class for the nonterminal Expr of the example grammar. The methods corresponding to the three rules are named `valid_01`, `valid_02`, and `valid_03`. Note that the new instance is valid if each of its parameters is valid. Note also that literals in the rules appear as string literals in the methods, as in method `valid_03`. The class `GCO_Number` for the Number terminal is not shown, but can consist of methods that return `GCO_Number` objects with representative strings, e.g. `0` and `42`.

Given these grammatical category classes, we can build up any string in any grammatical category by a sequence of method calls to the GCO classes. For instance, the sequence of method calls shown in Figure 3 should result in `e4` being an instance of `GCO_Expr` whose string is `"(0 + 42)"`. A random series of calls to the methods in the GCO classes should result in a set of random representatives of the grammatical categories being created.

```
GCO_Number n1 = GCO_Number.valid_01();
GCO_Number n2 = GCO_Number.valid_02();
GCO_Expr e1 = GCO_Expr.valid_01(n1);
GCO_Expr e2 = GCO_Expr.valid_01(n2);
GCO_Op o = GCO_Op.valid_01();
GCO_Expr e3 = GCO_Expr.valid_03(e1,o,e2);
GCO_Expr e4 = GCO_Expr.valid_04(e3);
```

Figure 3. An example sequence of method calls to GCO methods, creating the expression `(0 + 42)`.

B. Subject Grammars and Software

Here we describe how we selected and prepared the grammars used in the research, and applied GCO generation to them.

We wanted to study widely-used grammars that had a number and complexity of rules that realistically represented grammars used in industry. We therefore chose HTML and XML as our subjects, since both are non-trivial grammars that together account for billions of existing documents.

For HTML, we obtained a BNF grammar created by Marek Gryber[1]. We checked this grammar for correctness with respect to the W3C definition of HTML. For XML, we used the BNF grammar published by W3C[2]. Because we were not interested in lexical-level parsing issues, we introduced rules giving valid instances of strings for some grammatical categories, for instance the text between HTML tags or the name of an XML tag. (Kieżun et al. [10] followed a similar procedure.) We programmed a simple Java tool to read a BNF grammar and produce the Java source code for the GCO classes corresponding to it. For the HTML grammar, the tool generated 169 GCO classes with a total of 441 methods. For the XML grammar, the tool generated 105 GCO classes with a total of 228 methods.

We are often interested in seeing whether we can generate strings that are invalid, i.e. do *not* conform to a given grammar. This is so that we can test the robustness of the software under test to erroneous input. Completely random strings are likely to be invalid, but they are also unlikely to exercise very much of the capabilities of the software under test, since they are likely to be rejected very quickly. Strings that are invalid but close to being valid are more likely to cause more paths in the code to be followed, and more likely to force failures.

We therefore added methods by hand to each generated GCO class, that returned objects representing invalid instances of the grammatical categories. We added these invalid methods in such a way that it was easy to not include

[1] http://www.angelfire.com/ar/CompiladoresUCSE/COMPILERS.html
[2] `http://www.w3.org/TR/xml11/`

```
public class GCO_Expr extends GCOElement {
 private GCO_Expr(String s)
  { super(s); }
 public static GCO_Expr valid_01(GCO_Number n)
  { return new GCO_Expr(n.getString()); }
 public static GCO_Expr valid_02(GCO_Expr e1, GCO_Op o, GCO_Expr e2)
  { return new GCO_Expr(e1.getString() + o.getString() + e2.getString()); }
 public static GCO_Expr valid_03(GCO_Expr e)
  { return new GCO_Expr("(" + e.getString() + ")"); }
}
```

Figure 2. GCO class for the Expr grammatical category.

them if we wanted to see the effect of using only valid instances. In the future, we intend to explore automated ways of generating these methods.

IV. String Generation Approaches

We here describe the approaches that we compared for generating strings for the grammars. These fall into three broad categories: conformance test suites, deterministic approaches and metaheuristic approaches.

A. Conformance Test Suites

For both grammars, we obtained the standard conformance test suite published by W3C. The advantage of these test suites is that they are the product of many hours of expert work, and presumably thoroughly exercise the elements of the grammar.

The disadvantage of the conformance test suites is that they are intended primarily to test parsers for the grammars. Most subject software that we want to test consists not only of a parser, but also code that manipulates the abstract syntax tree that is the result of the parsing. Thus, the documents in the conformance test suite may contain features that are irrelevant for the software under test, and may miss specific instances that are handled in a special way by the software under test.

B. Deterministic Approaches

We studied two deterministic approaches to test data generation: depth-first search and rule coverage.

1) Depth-First Search: In depth-first search, we set a depth bound d and generate all possible strings that can be generated to that depth (which correspond to all possible instances of the top-level GCO class). In order to generate all possible objects of GCO class c to depth d, we used a backtracking equivalent of the following algorithm:

1) If $d \leq 0$, then return the empty set of objects.
2) Otherwise, initialize set S to the empty set, and for each method m of c returning a valid instance of c:
   a) For each parameter of m, generate all possible objects of the parameter's class to depth $d-1$.
   b) Call m using every possible combination of the resulting parameter values, and place the resulting objects in the set S.
3) Return S.

This algorithm has the advantage of generating an extremely thorough set of inputs, but the disadvantage of quickly running into a combinatorial explosion problem.

2) Rule Coverage: The rule coverage algorithm that we followed is actually rule coverage to a given depth d as well. It is the same as the depth-first search algorithm, but with one adaptation. As soon as it has generated one object of a given class, it caches that object, and reuses it whenever it needs another object of that class. It does, however, continue to explore all methods for a given class as in step (2) of the depth-first search algorithm, in order to guarantee that all rules are covered.

For every grammar, if any strings can be generated that use a given rule, then there must be a d at which rule coverage to depth d generates a string that uses that rule. We used rule coverage in order to find the lowest depth d for which all rules are covered. This number was 14 for HTML, and 20 for XML.

Rule coverage has the obvious advantage of guaranteeing that all the rules of the grammar correspond to code in the application that correctly handles them. However, because it reuses only one representative of a GCO class once one has been generated, it is unlikely to detect interactions between features of the grammar that exercise code or reveal bugs.

C. Metaheuristic Approaches

Metaheuristic algorithms are those which search in a space of possible solutions to a problem in order to find optimal solutions. They are usually based on a notion of *fitness* of a solution to a particular problem. By translating a grammar into Java object classes, we are able to leverage an existing tool, Nighthawk [12], that uses metaheuristic search to create unit tests that aim to achieve high code coverage.

Nighthawk is made up of two levels: a lower level that performs randomized unit testing, and an upper level that uses a genetic algorithm (GA) to find optimal properties

to be used in the randomized testing. We first describe the randomized level of Nighthawk, then the upper level. We then discuss how we adapted Nighthawk for this work to use other metaheuristic algorithms.

1) Randomized Level: The randomized unit testing level performs the following general algorithm:

1) For each type of interest, set up one or more arrays (referred to as a *value pool*) to contain objects of that type.
2) Initialize the primitive-type value pools to contain values within chosen ranges.
3) Repeat n times, where n is the desired length of the test case in number of method calls:
    a) Choose a method or constructor to call.
    b) Choose parameters to the method or constructor, and the target (if any), from the appropriate value pool.
    c) Call the method or constructor. If it throws an uncaught exception, consider this a failure.
    d) For a constructor, or a method that returns a non-null value, choose a location in a value pool to contain the returned object.

The methods and constructors called are not only the ones that we wish to test, but also those that will construct and manipulate arguments to those methods. Programmers can ensure that an uncaught exception really does correspond to a failure by enclosing the unit under test in a "wrapper" class that handles expected exceptions. The word "choose" or "chosen" above represents a random choice according to given parameters.

For the example grammar in Figure 2 and its GCO classes, the random testing level of Nighthawk would be given `GCO_Number`, `GCO_Expr`, and `GCO_Op` as types of interest, and would construct value pools for each type. It would then make method calls to methods such as the `valid_XX` methods of `GCO_Expr`, meaning that it could potentially make a sequence of method calls having the same effect as that shown in Figure 3 – along with many other possible sequences.

2) Metaheuristic Level: The upper, metaheuristic level encodes the parameters of the randomized testing in a "chromosome", including the test case length n. The original implementation of Nighthawk finds good settings for the parameters by the usual genetic-algorithm process of mutation and recombination of successful chromosomes [17]. To evaluate the fitness of a chromosome (i.e., a set of parameter settings), the metaheuristic level runs the randomized level given the settings in the chromosome, and observes the number of lines of code covered. The fitness function that drives chromosomes to be better is based on a balance between the line coverage achieved on the software under test (more is better) and the length of the test case (less is better, because a long test case is expensive to run). Chromosomes that are more fit are permitted to recombine to form the next generation of chromosomes to be evaluated.

An important class of parameters encoded in the GA chromosome is the relative weight of a method; there is one such gene in the chromosome for every method. The relative weight determines how often the method is called by the randomized testing level. Generally speaking, if calling method A tends to lead to higher code coverage than calling method B, then the GA is likely to find an optimal weight for A which is higher than that of B. Other parameters include the size of the value pools for the individual types.

For the example grammar in Figure 2 and its GCO classes, the metaheuristic level of Nighthawk will tend to find relative weights for the `valid_XX` methods that optimize code coverage. For instance, if the software contains a lot of code for manipulating expressions with many parentheses, we would expect Nighthawk to tend to assign a higher weight to the method `GCO_expr.valid_03()`, which creates parenthesized expressions from simpler ones. We would also expect Nighthawk to converge on values for the test case length (number of calls) and value pool sizes which cause the randomized testing level to create large enough strings to test the program thoroughly.

D. Adaptation

For this work, in addition to the default GA implementation of the upper level, we implemented two further metaheuristic algorithms.

In the *hill-climbing (HC)* approach, we start with a default chromosome[3], and then perform a set number of steps. At each step, the current chromosome is randomly mutated; if the new chromosome encodes a better solution, it becomes the current chromosome, used for the next mutation, and otherwise we discard it.

The *simulated annealing (SA)* approach is the same as hill-climbing, except that every chromosome has a chance of becoming the current chromosome, even if its fitness is lower. The chance that a less fit chromosome is chosen as the current is dependent on how much worse it is, and how close we are to the end of the sequence of steps (the "temperature" of the simulated annealing). In the SA approach, as in the original GA, we always remember the best chromosome found so far.

V. Empirical Comparison

In this section, we describe our empirical comparison of the approaches on the two subject grammars. We first describe the subject software and our procedure, and then our qualitative observations about the application of the techniques to the grammars, including the discovery of a long-standing bug in NanoXML. We then present the quantitative results of the study, and end with a discussion.

[3]For consistency, we will refer to the solution as a chromosome even when referring to the non-GA algorithms.

A. Subject Software

Because we were interested in testing actual software that used the grammars in question, we chose two subject programs that have been used in many software testing experiments in the past. For HTML, we selected JTidy, an open source Java port of HTML Tidy, a HTML syntax checker and pretty printer. JTidy can be used to fix a wide range of HTML problems as long as it is sure of how to handle them. Some of the problems JTidy is able to fix are missing or mismatched end tags, end tags in the wrong order, missing quotes around attribute values, and missing backslashes on end tag anchors. It was used as a subject program by Patrick et al. [18] and Masri et al. [19], among others.

For XML, we selected NanoXML, an open source XML parser which is relatively small and fairly fast for small XML documents. The latest version of NanoXML is NanoXML2. It has three branches: namely, NanoXML/Lite, NanoXML/Java and NanoXML/SAX. In our experiment, we used NanoXML/Lite. NanoXML has been used as a subject program by Masri et al. [20] and Pavan et al [21], among others, and is available in the SIR (Subject Infrastructure Repository) at the University of Nebraska-Lincoln.

B. Procedure

For each of the subject programs, we used the open-source coverage tool Cobertura [1] to measure the line coverage achieved on the program. However, our general approach can incorporate any coverage measure.

For the conformance test suite approach, we ran all of the conformance tests for the two grammars on the two applications: the conformance test suite for HTML on JTidy, and the conformance test suite for XML on NanoXML. (NanoXML is strictly speaking a library; what we ran was a default application that simply reads an XML document and then writes it out again.)

For the deterministic approaches (depth-first and rule coverage), we increased the depth bound d until the generation process ran for an infeasibly long time (in the case of depth-first), or until no more coverage was achieved on the subject program (in the case of rule coverage).

For the metaheuristic approaches (GA, HC and SA), we first semi-automatically created "jackets" for the subject program. These jackets were special wrappers that took GCO objects as arguments, and extracted the strings in them for processing by the subject program (either as strings, or as input streams based on strings). The generation of the jackets was automatic except for the change of parameter from string to GCO object. For each run of the Nighthawk tool, we gave the GCO classes and the jacket as the classes whose methods the tool was to call; however, as with the other approaches, we measured coverage only on the subject program itself.

An exploratory analysis showed that a good tradeoff in time and coverage could be achieved by running Nighthawk three times, getting a different solution chromosome every time, and then generating 1000 test cases based on each chromosome. We refer to this as a "grand run" of Nighthawk. Because of the randomness involved in the search and test case generation, we performed 10 grand runs of each metaheuristic approach.

With the metaheuristic approaches, we also ran two different settings: one in which only the "valid" methods generated by the GCO generator were called, and one in which both the valid methods and the "invalid" methods that we added by hand were called. We did this in order to see if one or the other setting achieved higher coverage.

C. Observations

The application of the tools to HTML and JTidy was relatively straightforward, and involved no adaptation.

For NanoXML, however, we at first achieved very low coverage with all but the conformance test suite. Analysis revealed that there were two main reasons for this. First, the context-free grammar for XML was unable to express the constraint that the starting and ending tags of an XML element must match. This is in fact a constraint that is impossible to express in a context-free grammar. We therefore adapted the generated code so as to ensure that starting and ending tags matched. A more general solution would be to use some form of context-sensitive grammar as the basis of the GCO classes.

The second reason for the initial low coverage was that many of the conformance tests referred to declarations held in files external to the main XML file being parsed. Rather than generate these external declarations and write them to disk, which would have been computationally expensive, we instead adapted NanoXML so that whenever it was going to read from an external file, it would read instead from the string in a GCO object containing valid external-file declarations.

In the course of testing NanoXML, we realized that the optimal solutions that Nighthawk was reaching were avoiding a particular feature of XML. This is the NDATA feature, which allows binary data to be included in an XML file, tagged with the type of data it is (e.g., a JPEG image). Further investigation revealed that the NanoXML code did not mention NDATA at all, and rejected as invalid any XML string that used the feature. NDATA has been a feature of XML since the first draft of XML 1.0, and has persisted in every draft of XML 1.0 and 1.1 since, but has never been implemented in NanoXML, whose latest release was in 2003.

Our testing therefore revealed a bug in NanoXML that had apparently been unnoticed for over 8 years, despite its extensive use in software testing experiments. We reported the bug to the developers.

JTidy			NanoXML		
T	LC	Groups	T	LC	Groups
GA/VO	3642	A	Conf	863	A
SA/VO	3627	A	HC/VO	831	A
HC/VO	3622	A	SA/VO	813	A
GA/VI	3545	B	Rule	757	B
SA/VI	3511	B	GA/VO	748	B,C
HC/VI	3486	B	SA/VI	719	B,C
Conf	2591	C	GA/VI	710	B,C
Rule	2535	C	HC/VI	696	C
Depth	2000	D	Depth	420	D

Figure 4. Average coverage and Tukey HSD groups for testing techniques. T: Technique. LC: Lines covered. VO: valid only. VI: valid and invalid.

D. Results

1) Average Coverage: Figure 4 shows the number of lines covered for the techniques on the subject programs, ranked by number of lines covered. For the metaheuristic techniques, each number represents the average over 10 runs. The best technique for JTidy (the GA with valid-only methods) covered 3642 out of the 7299 lines of code, or about 50% of the code; the best technique for NanoXML (the conformance test suite) covered 863 out of the 1137 lines of code, or about 76% of the code. The lines of code not covered mostly deal with features of the two programs that are not accessed by parsing files, and so do not concern us. Each grand run of the metaheuristic techniques (3 runs of Nighthawk + 1000 test cases generated for each run) took between 20 and 87 minutes of CPU time, run on a Sun UltraSPARC-IIIi running SunOS 5.10 and Java 1.5.0_11.

To measure statistical significance, we performed a Tukey Honest Significant Differences (HSD) test on the data, using the R statistical package; this test shows which pairs of data sets exhibit statistically significant differences to a 95% confidence level. We then grouped the techniques into groups, such that every group contains only members that are not statistically significantly different from each other, and we assigned each group a letter of the alphabet. Figure 4 also shows these groups. Some techniques are in more than one group because the groups only partially overlap.

The results show that for JTidy, the metaheuristic approaches were the best, although somewhat to our surprise the valid-only setting was superior to the valid-and-invalid setting. Within each setting, there was no significant difference among the three metaheuristic techniques. The conformance test suite achieved about the same as rule coverage, and depth-first search achieved the least coverage. We interpret these results as indicating that Nighthawk could find optimal weights for the different rules of the HTML grammar that maximized coverage of the combinations of features handled by JTidy, while the conformance test suite covered each feature only once.

The XML conformance test suite fared better for NanoXML, achieving the highest average coverage, though still not significantly different from simulated annealing or hill climbing. The rest of the metaheuristic techniques and rule coverage formed two overlapping groups of statistically indistinguishable techniques, while depth-first was again at the bottom.

2) Complementarity of Techniques: The different techniques achieved different levels of coverage, but did they cover the same lines of code? To answer this question, we performed several other visualizations.

Figures 5 and 6 show stacked bar graphs representing the coverage of each technique and the extent to which it overlaps the others. The top bar in each graph is the union of the lines of code covered by all techniques. In the rest of the bars, the darkest grey represents the coverage of the 1/3 of the lines covered by the union that were covered most commonly. The lightest grey represents the lines covered only by this technique. The intermediate grey represents the rest of the lines covered by this technique. Again, all results show the average over the 10 runs of the metaheuristic techniques.

We can see that for JTidy, all the techniques were able to cover a core set of lines of code, and that although the HTML conformance test suite achieved lower coverage, it did cover a sizable number of lines of code not covered by the other techniques. For NanoXML, the XML conformance test suite covered a large number of lines of code not covered by the other techniques, but when those lines were subtracted, the conformance test suite achieved less coverage than the best metaheuristic techniques.

Figures 7 and 8 show tables which illuminate the complementarity further. Each entry in the table shows the average size of the union of the lines covered by two of the techniques. The coverage is expressed as a percentage of the coverage of the conformance test suite. Thus, Figure 7 shows that for JTidy, although the best technique (GA/VO) achieved 41% more coverage than the conformance test suite (indicated by the number 141 in the entry for GA/VO union GA/VO), combining the conformance test suite with the best metaheuristic techniques achieved 51% more coverage than the conformance test suite alone. For NanoXML, Figure 8 shows that adding any of the techniques to the conformance test suite allows it to achieve an additional 17-19% code coverage.

E. Threats to Validity

The main threat to internal validity of the study was the correctness of the software we used. We used extensive cross-checks and testing to assure the quality of the software.

Although we used several different techniques for generating string data in the study, we used only two subject programs. This is the main threat to external validity. The subject programs were of a reasonable size and processed

	SA/VI	HC/VI	GA/VI	SA/VO	HC/VO	GA/VO	Rule	Depth	Conf
SA/VI	136	138	139	145	145	146	142	140	148
HC/VI	138	135	138	146	145	145	141	139	147
GA/VI	139	138	137	146	145	146	143	141	149
SA/VO	145	146	146	140	143	143	145	144	151
HC/VO	145	145	145	143	140	143	145	144	150
GA/VO	146	145	146	143	143	141	146	145	151
Rule	142	141	143	145	145	146	98	99	107
Depth	140	139	141	144	144	145	99	77	103
Conf	148	147	149	151	150	151	107	103	100

Figure 7. Coverage of unions of pairs of approaches for JTidy. Each number represents the percentage coverage, where the conformance test suite is the baseline 100%.

	SA/VI	HC/VI	GA/VI	SA/VO	HC/VO	GA/VO	Rule	Depth	Conf
SA/VI	83	85	87	97	99	93	103	89	117
HC/VI	85	81	84	97	99	92	101	86	117
GA/VI	87	84	82	98	100	93	101	88	118
SA/VO	97	97	98	94	97	95	105	100	118
HC/VO	99	99	100	97	96	97	106	102	118
GA/VO	93	92	93	95	97	87	103	92	117
Rule	103	101	101	105	106	103	88	94	119
Depth	89	86	88	100	102	92	94	49	117
Conf	117	117	118	118	118	117	119	117	100

Figure 8. Coverage of unions of pairs of approaches for NanoXML.

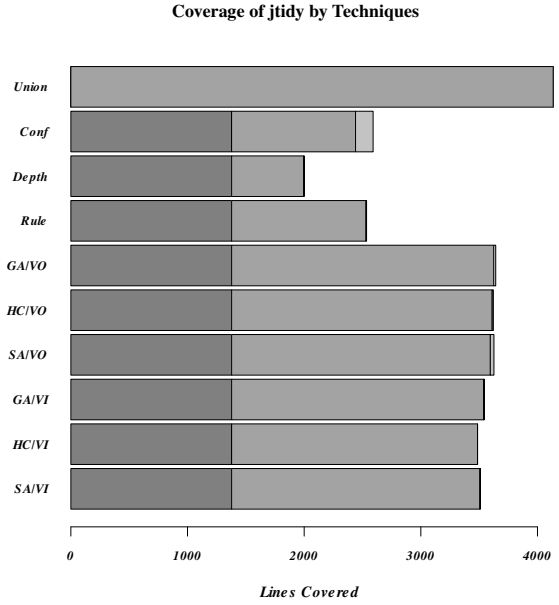

Figure 5. Coverage of JTidy code by testing techniques.

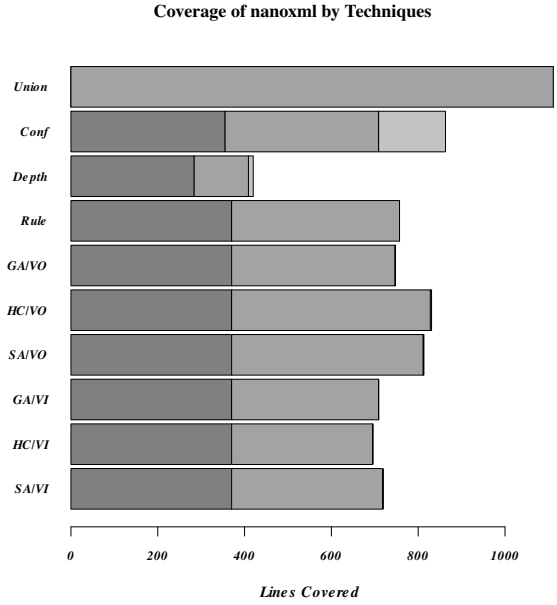

Figure 6. Coverage of NanoXML code by testing techniques.

strings in heavily-used languages (HTML and XML); however, the behaviour of the studied techniques on other programs might be different. In the future, we would like to extend the study to grammars for the input to other Java software, and also to the testing of non-Java programs through the use of JNI or system calls.

F. Discussion

We now discuss some implications of the empirical study.

The differing results for the two programs indicate a difference in the nature of the software. JTidy is a program for cleaning up HTML code, and has numerous switches that users can change to access individual features. This probably accounts for the poor overall coverage of all the techniques on the JTidy code. NanoXML is a lightweight library with a minimum of features, allowing the conformance test suite to achieve high coverage on it.

The metaheuristic techniques managed to achieve a reasonable amount of coverage even for NanoXML, especially since the amount of work needed for generating the strings was considerably less than the work done by the committee that developed the conformance test suite. The best results were achieved by augmenting the conformance test suite by one of the metaheuristic techniques. It should be noted, however, that for NanoXML even simple rule coverage added as much coverage as the metaheuristic techniques.

The use of invalid methods (GCO methods generating invalid strings) did not turn out to be very useful, even in the case of JTidy, which processes and tidies up invalid HTML. We assume that this is because the particular strings that we generated did not necessarily conform to the common types of errors that JTidy is designed to correct.

Not discussed so far in this paper is the issue of checking test results. One advantage of using a conformance test suite is that the test cases are documented and labelled, allowing a tester to interpret more easily if their software does or does not conform to the desired grammar and why. Since the metaheuristic techniques all generated 3000 test cases with no documentation, they would seem to impose a heavy duty of test result checking on the user.

However, this perception is misleading. Given the complementarity of the testing techniques, it is possible to run a conformance test suite, and then run a metaheuristic technique, discarding each new test case that arises if it does not cover more lines of code. Each test case can then be minimized using Zeller-Hildebrandt minimization [22] so as to be the smallest possible test case that still achieves the extra coverage. This minimization should cut down on the amount of work a tester needs to do to check the correctness of the testing techniques. We leave the question of how much is saved in this way for future work.

VI. Conclusion

We studied six techniques for generating string data, falling into three groups: conformance testing, deterministic appraoches, and metaheuristic approaches. All the deterministic and metaheuristic approaches were facilitated by the automatic translation of a context-free grammar into a set of Java classes. We also studied the effect of adding methods that generate invalid strings to those classes.

We compared the techniques on two large, important grammars (HTML and XML) and two widely-studied pieces of software that used then (JTidy and NanoXML). We concluded that using the standard conformance test suite in combination with one of the metaheuristic techniques achieved, for both programs, higher code coverage than could be achieved by any technique in isolation. Our work also identified and diagnosed a long-standing bug in NanoXML.

Our overall conclusion is that automatically generating string data by metaheuristic techniques is a viable way to obtain data for testing applications that process strings conforming to a given grammar. However, it is not the only solution. The complementarity of the metaheuristic approaches with the conformance test suites indicates that such test suites still have a place in testing software. For grammars without an existing conformance test suite, however, the metaheuristic approach is a good starting point.

Acknowledgment

This work was supported by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada. Many thanks to Michael Ernst for useful observations and suggestions, to Vijay Ganesh and Adam Kieżun for help with Hampi, and to Marc De Scheemaecker for discussions about NanoXML.

References

[1] Cobertura Development Team, "Cobertura web site," accessed August 2010, `cobertura.sourceforge.net`.

[2] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366–375, September 1972.

[3] M. Hennessy and J. F. Power, "An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, Long Beach, CA, USA, November 2005, pp. 104–113.

[4] M. Alshraideh and L. Bottaci, "Search-based software test data generation for string data using program-specific search operators," *Software Testing, Verification and Reliability*, vol. 16, pp. 175 – 203, September 2006.

[5] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[6] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. SE-2, no. 3, pp. 215–222, September 1976.

[7] R. Zhao and M. R. Lyu, "Character string predicate based automatic software test data generation," in *3rd International Conference on Quality Software (QSIC 2003)*, Dallas, TX, November 2003, pp. 255–262.

[8] B. Korel, "Automated software test generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, August 1990.

[9] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *Testing of Communicating Systems (TestCom 2006)*, ser. LNCS, no. 3964, May 2006, pp. 19–38.

[10] A. Kieżun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimiejer, and M. Ernst, "Hampi: A solver for word equations over strings, regular expressions and context-free grammar," *ACM Transactions on Software Engineering and Methodology*, in press.

[11] C. Cadar, Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, 2008, pp. 209–224.

[12] J. H. Andrews, F. C. H. Li, and T. Menzies, "Nighthawk: A two-level genetic-random unit test data generator," in *IEEE ASE'07*, 2007.

[13] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of Reliability of UNIX Utilties," *Communications of the ACM*, vol. 33, no. 12, pp. 32–34, 1990.

[14] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed Random Test Generation," in *In Proceedings of the 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, MN, May 2007, pp. 75–84.

[15] A. Groce, G. J. Holzmann, and R. Joshi, "Randomized Differential Testing as a Prelude to Formal Verification," in *Proceedings of the 29th international conference on Software Engineering(ICSE 2007)*, Minneapolis, MN, May 2007, pp. 621–631.

[16] R.-K. Doong and P. G. Frankl, "The ASTOOT Approach to Testing Object-oriented Programs," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 3, no. 2, pp. 101–130, April 1994.

[17] J. H. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor: University of Michigan Press, 1975.

[18] P. Francis, D. Leon, M. Minch, and A. Podgurski, "Tree-Based Methods for Classifying Software Failures," in *Proceedings of the 15th International Symposium on Software Reliability Engineering*, Washington, DC, USA, 2004, pp. 451–462.

[19] W. Masri and A. Podgurski, "An Empirical Study of the Strength of Information Flows in Programs," in *Proceedings of the 2006 international workshop on Dynamic systems analysis*, Shanghai, China, 2006, pp. 73–80.

[20] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An Empirical Study of the Factors that Reduce the Effectiveness of Coverage-based Fault Localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, Chicago, Illinois, 2009, pp. 1–5.

[21] M. J. Harrold and P. K. Chittimalli, "Recomputing Coverage Information to Assist Regression Testing," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 452–469, July 2009.

[22] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, February 2002.