Evaluating Program Analysis and Testing Tools with the RUGRAT Random Benchmark Application Generator

Ishtiaque Hussain, Christoph Csallner
University of Texas at Arlington
Arlington, TX 76019, USA
ishtiaque.hussain@mavs.uta.edu, csallner@uta.edu

Mark Grechanik
Accenture Technology Labs and University of Illinois
Chicago, IL 60601, USA
drmark@uic.edu

Chen Fu, Qing Xie
Accenture Technology Labs
Chicago, IL 60601, USA
{chen.fu, qing.xie}@accenture.com

Sangmin Park
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
sangminp@cc.gatech.edu

Kunal Taneja
Accenture Technology Labs and North Carolina State University
Raleigh, NC 27606, USA
ktaneja@ncsu.edu

B. M. Mainul Hossain
University of Illinois at Chicago
Chicago, IL 60607, USA
bhossa2@uic.edu

ABSTRACT

Benchmarks are heavily used in different areas of computer science to evaluate algorithms and tools. In program analysis and testing, open-source and commercial programs are routinely used as benchmarks to evaluate different aspects of algorithms and tools. Unfortunately, many of these programs are written by programmers who introduce different biases, not to mention that it is very difficult to find programs that can serve as benchmarks with high reproducibility of results.

We propose a novel approach for generating random benchmarks for evaluating program analysis and testing tools. Our approach uses stochastic parse trees, where language grammar production rules are assigned probabilities that specify the frequencies with which instantiations of these rules will appear in the generated programs. We implemented our tool for Java and applied it to generate benchmarks with which we evaluated different program analysis and testing tools. Our tool was also implemented by a major software company for C++ and used by a team of developers to generate benchmarks that enabled them to reproduce a bug in less than four hours.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*testing tools*; D.1.2 [**Programming Techniques**]: Automatic Programming; K.6.2 [**Management of Computing and Information Systems**]: Installation Management—*benchmarks*

WODA '12, July 15, 2012, Minneapolis, MN, USA

General Terms

Management, Measurement, Performance, Reliability

Keywords

Benchmark applications, program analysis, benchmark application generator, stochastic grammar, stochastic parse tree

1. INTRODUCTION

A benchmark is a point of reference from which measurements can be made in order to evaluate the performance of hardware or software or both [19]. Benchmarks are important, since organizations and companies use different benchmarks to evaluate and choose mission-critical software for business operation [14]. Businesses are often confronted with a limited budget and stringent performance requirements while developing and deploying enterprise applications, and benchmarking is often the only way to choose proper infrastructures from a variety of different technologies for these applications. For example, application benchmarks play a crucial role in the U.S. Department of Defense acquisition process [27]. Given that corporations spend between 3.4% and 10.5% of their revenues on technologies, biased or poorly suitable benchmarks lead to wrong software and hardware architecture decisions that result in billions of dollars of losses every year [20].

Benchmarks are very important for evaluating *pRogram Analysis and Testing (RAT)* algorithms and tools [2, 3, 9, 23]. Different benchmarks exist to evaluate different RAT aspects, such as how scalable RAT tools are, how fast they can reach high test coverage, how thorough they handle different language extensions, how well they translate and refactor code, how effective RAT tools are in executing applications symbolically or concolically, and how efficient these tools are in optimizing, linking, and loading code in compiler-related technologies, as well as profiling. For example, out of the 29 papers that described controlled experiments in software testing published in TOSEM/TSE/ICSE/ISSTA from 1994 to 2003, 17 papers utilize the *Siemens* benchmark which includes a set of seven C programs with only several hundreds lines of code [8]. Currently, a strong preference is towards selecting benchmarks that have much richer code complexity (e.g., nested `if-then-else` statements),

class structures, and class hierarchies [2, 3]. Unfortunately, complex benchmark applications are very costly to develop [14, page 3], and it is equally difficult to find real-world applications that can serve as unbiased benchmarks for evaluating RAT approaches.

Consider a situation where different test input generation techniques are evaluated to determine which one achieves higher test coverage in a shorter period of time. Typically, test input generators use different algorithms to generate input data for each application run, and the cumulative statement coverage is reported for all runs as well as the elapsed time for these runs. On one extreme, "real-world" applications of low complexity are poor candidate benchmarks, since most test input data generation approaches will perform very well by achieving close to 100% statement test coverage in very few runs. On the other extreme, it may take significant effort to adjust these approaches to work with a real-world distributed application whose components are written in different languages and run on different platforms. Ideally, a large number of different benchmark applications are required with different levels of code complexity to appropriately evaluate test input data generation tools.

Writing benchmark application from scratch is laborious and requires a lot of manual effort, not to mention that a significant bias and human error can be introduced [13]. In addition, selecting commercial applications as benchmarks negatively affects reproducibility of results, which is a cornerstone of the scientific method [24], since commercial benchmarks cannot be easily shared among organizations and companies for legal reasons and trade-secret protection. For example, Accenture Human Resource Policy item 69 states that source code constitutes confidential information, and other companies have similar policies. Finally, more than one benchmark is often required to determine the sensitivity of the RAT approaches based on the variability of results for applications that have different properties.

Ideally, users should be able to easily generate benchmark applications with desired properties. This idea has already been used successfully in testing relational database engines, where complex *Structured Query Language (SQL)* statements are generated using a random SQL statement generator [26]. Suppose that a claim is made that a relational database engine performs better at certain aspects of SQL optimization than some other engine. The best way to evaluate this claim is to create complex SQL statements as benchmarks for this evaluation in a way that these statements have desired properties that are specific to these aspects of SQL optimization, for example, complicated nested SQL statements that contain multiple joins. Since the meaning of SQL statements does not matter for performance evaluation, this generator creates semantically meaningless but syntactically correct SQL statements thereby enabling users to automatically create low-cost benchmarks with significantly reduced bias.

In this paper, we propose a *Random Utility Generator for pRogram Analysis and Testing (RUGRAT)*, which is a novel language-independent approach and a tool for generating application benchmarks within specified constraints and within the range of predefined properties. RUGRAT is implemented for Java and it is used to evaluate different open-source RAT tools. This paper makes the following contributions:

- We created a novel approach for random generation of application benchmarks that is based on stochastic parse trees, where language grammar production rules are assigned probabilities that specify the frequencies with which instantiations of these rules will appear in the generated programs. We show that random programs are similar to real-world complex programs using different software metrics.
- RUGRAT is implemented and used at a large corporation where a bug was found in a loader in less than four hours, in contrast to a team of five engineers who spent close to three weeks to find this bug before using RUGRAT.
- Finally, we implemented RUGRAT and used it to generate dozens of Java applications, ranging from 300 LOC to 5 MLOC, to evaluate popular program analysis and testing tools. This version of RUGRAT is available for public use[1].

2. OUR APPROACH

In this section, we present a model for the random application generator, and discuss the goal of our work along with the approaches to address the issues to achieve the goal.

2.1 Stochastic Grammar Model

Consider that every program is an instance of the grammar of the language in which this program is written. Typically, grammars are used in compiler construction to write parsers that check the syntactic validity of a program and transform its source code into a parse tree [1]. An opposite use of the grammar is to generate branches of a parse tree for different production rules, where each rule is assigned the probability with which it is instantiated in a program. These grammars and parse trees are called *stochastic*, and they are widely used in natural language processing, speech recognition, information retrieval [4], and also in generating SQL statements for testing database engines [26]. We use a *stochastic grammar model* to generate random object-oriented programs.

We obtain random programs by construction that is based on the stochastic grammar model, and the construction process can be described as follows. Starting with the top production rules of the grammar, each nonterminal is recursively replaced with its corresponding production rule. When more than one production rule is available to replace a nonterminal, a rule is randomly chosen based on equal probability. Terminals are replaced with randomly generated identifiers and values that preserve syntax rules of the given language. Termination conditions for this process of generating programs include the limit on the size of the program or selected complexity metrics. With the stochastic grammar model it is ensured that the generated program is syntactically correct and compiles. The construction process can be fine tuned by varying the ranges of different configuration parameter values and limiting the grammar to a subset of the production rules that are important for evaluating specific RAT tools (e.g., recursion, use of arrays, or use of different data types can be turned off if a RAT approach does not address these).

2.2 Our Goal and Approach

We address one main goal—to allow experimenters to automatically generate benchmark applications that have desired properties that these experimenters need to evaluate RAT approaches and tools. It is not a purpose of RUGRAT to replace real-world applications for evaluation of different RAT approaches and tools; instead we see RUGRAT as a tool that enables experimenters to quickly generate a large number of application benchmark that have desired properties to supplement evaluations of RAT tools using real-world application benchmarks and to achieve statistical significance of these evaluations. In a way, we see RUGRAT as a rapid prototyping tool for producing a set of benchmark applications for initial evaluation of RAT approaches and tools.

To address our goal, we should address several issues. First, generated program must have a wide variety of language constructs

[1]http://www.rugrat.ws

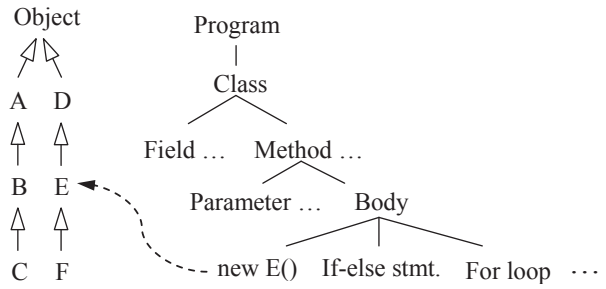


Figure 1: RUGRAT can generate deep subtype hierarchies. It then picks classes (such as E) from such a hierarchy randomly to create instances and call methods on these instances.

that are important for evaluating RAT approaches and tools. For example, recursion, dynamic dispatch, and array manipulations using expressions that compute array indices test the boundaries of RAT algorithms. Multithreading is also a prominent feature of real-world applications. Existing program generators that are based on the stochastic grammar model do not take into consideration these specific language constructs to add them to generated programs.

In addition, there is an issue that generated programs should represent real-world programs using software metrics. We address this issue by varying the probabilities that are assigned to different production rules of the language grammar. We compare the metrics of generated programs with metrics of different open-source Java programs as well as commercial ones as part of our experimental evaluation of RUGRAT (see Section 5).

3. IMPLEMENTATION

We describe the implementation of our RUGRAT approach in Java. Figure 1 shows an example snapshot of RUGRAT's program generation process. Starting from the root of the abstract syntax tree, RUGRAT keeps instantiating syntax rules. When there are multiple rules available for a non-terminal, we randomly choose one that satisfies the overall program configuration. For example, if we have reached the configured maximum depth of nested conditions in the current method, we skip the *if-else* rule. Similarly, if we have reached the configured total LOC, we choose only terminals.

At first glance such a blind random generation process may seem simplistic. However, modern object-oriented languages such as Java, C++, or C# contain many complex features that impose additional well-formedness rules on generated programs. It is therefore more challenging to generate correct programs, especially if we want the generated programs to use a wide variety of complex language features. Our goal is to let the user choose the size of the generated programs as well as the mix of language features the generated programs should be using. In the following, we briefly describe how we solve some of the key challenges.

3.1 Language Features

Many language features cannot be generated correctly by blind random program generation, because they have associated well-formedness rules that any legal program must satisfy. For example, a method can only be called if it and its defining type have a visibility that permits the call from the specific call-site, a final field defined by class C must be initialized directly or by each constructor of C, and generated non-abstract classes have to provide implementations for all inherited abstract methods. Special care has to be given to avoid generation of loops that may not terminate or non-terminating recursive calls, if desired.

To enforce these restrictions, RUGRAT utilizes internal tables and sets. I.e., RUGRAT implements a symbol table to ensure that only variables from correct scopes are used, it maintains type compatibility, and it makes a type cast for every assignment expression. It allows primitive and reference types in method parameters and method bodies. To avoid runtime exceptions such as divide-by-zero, RUGRAT enforces that only non-zero valued expressions occur in the denominator of a division operation. For iteration statements, RUGRAT only uses *for* loops with literals in the loop condition to avoid infinite loops. It uses special configuration parameters to enable and control recursion and indirect recursion. It also ensures that all abstract methods of all (transitive) super-types are implemented and no *non-static* field is referenced in a *static* method.

For instance fields our prototype currently only supports primitive types. Not generated are calls to Java library methods. Several advanced language features such as *generics* are also not yet implemented. All of these are subject to future work.

3.2 Configuration Options

RUGRAT is highly configurable. Some of the important parameters include number of classes, number of methods per class, number of interfaces, number of methods per interface, maximum depth of the inheritance hierarchy, number of class fields, number of parameters per method, and recursion depth (if recursion is enabled). Most of the parameters have a lower and an upper limit. Moreover, many parameters are inter-dependent (e.g., there should be enough classes to populate an inheritance tree of a desired depth). Once these limits are defined, RUGRAT randomly chooses values from each range.

For each of these configuration parameters, we define a default range that seems reasonable based on empirical data [11, 30, 5]. For example, to determine the number of classes, we follow Zhang et al.'s [30] observation that LOC is roughly 114 times the number of classes in a program and set $classes = LOC/114$. To define the number of interfaces, we follow the observation of Collberg et al. [5], that each package in a program has roughly 12 classes and there is 1 interface per package. Hence we set $interfaces = LOC/(114*12) = LOC/1368$. Grechanik et al. [11] found that the average value for the 'maximum number of methods per interface' is 3.4, we took ten times the average value and set the upper limit of the range to 34. Collberg et al. found that 96% of the programs have less than 20 class fields, and 99% of the programs have less than 60 methods per class. We conformed to these observations and used these values as the upper bound for respective parameters. We used similar heuristics for other parameters, such as number of parameters per method and maximum inheritance depth. The tool website, given in Footnote 1, has a complete list of the configuration parameters and their default values.

4. CASE STUDY ON A LOADER

RUGRAT was implemented for generating C++ programs by a Fortune 100 company, and it was used to reproduce a bug in a loader, which is a utility that copies a program from a secondary storage into main memory so it can be run [16]. We cannot reveal the company name for confidentiality reasons. (Revealing the company name would not change the nature or validity of our results.) A loader is one of many products of this company, and RUGRAT was evaluated by the team that maintains and evolves the loader code. The loader was written in the middle of the 1970s and since then has been ported to different platforms and is maintained regularly. In 2010, a large customer complained that it takes an unusually long to load a large C++ program with more than three million LOC. Repeated attempts failed to reproduce this problem on different subject applications. The client could not share its source code

to reproduce the bug, for trade-secret and other reasons.

After a costly search, the loader maintainers found the problem in a hash function used in the symbol table. The objective of making a simple and efficient hash function was possibly the main reason why the loader developers decided to compute a hash value by applying the binary operation `XOR` to every third character in an access path starting with the first character up to a certain limit, which was 128 characters. The identifier was inserted in the list of the bucket in the hash, and the bucket number was determined by dividing modulo the hash value by the number of buckets in the hash. As long as identifiers were spread among different buckets, retrieving them was fast. This was probably a good choice for programs decades ago, as back then storage space was tight, which motivated programmers to use short identifier names that fit easily into the 128 character range used in the loader's hash function.

The efficient hash table approach worked fine for many years. However over time programmers started using longer names, deeper inheritance hierarchies and nested namespaces, and more access paths had the same 128 character prefix. At some point, tens of thousands of identifiers were put in a single bucket, reducing the hash table to a linked list. This situation was aggravated by the fact that this application dynamically linked to many external libraries, and the loader performed extensive relocations, which involved searching and retrieving different identifiers. With the hash structure reduced to a linked list, the complexity of searching increased to $O(n)$ and caused a significant performance penalty.

In retrospect, this bug should be easy to find. But since the program that exposed this problem could not be shared with engineers who worked on the loader, they had to first reproduce the bug. To expose the bug, it was important to have a subject program with several key characteristics including deep levels of inheritance, deeply nested namespaces, and long identifier names. Since loader engineers thought of at least two dozen possible causes of this bug, it took over three weeks for a team of five engineers to finally find the cause of this bug (over 600 man hours with the cost of over $35,000). Fixing the bug then took just a couple of hours. The main reason for this high cost of the bug is the inability to quickly reproduce it and locate the fault in the hash data structure.

With RUGRAT, it took less than four hours to generate and compile ten programs with 10MLOC with over 5,000 classes and 200 inheritance hierarchies, whose length on average was five classes. Once ran, the bug was immediately reproduced and the fault was found very quickly, since a profile showed that most time was spent searching for identifiers in the hash table. The team now uses RUGRAT routinely to generate subject programs for testing linkers and loaders.

5. EXPERIMENTATION ON RATS

In this section, we describe our initial experimentation with our RUGRAT prototype implementation for Java. The goal of the experimentation is to determine whether RUGRAT-generated applications can be useful for finding bugs or shortcomings in program analysis and testing (RAT) tools. In the context of RAT tools we also refer to RUGRAT-generated applications as applications under test (AUTs). Specifically, we explore the following concrete research questions.

- **RQ1.** How similar are RUGRAT-generated applications to third-party applications?
- **RQ2.** How do program analysis tools behave while analyzing RUGRAT-generated applications?
- **RQ3.** Can RUGRAT-generated applications find defects in program analysis tools?

To explore these research questions, we ran two experiments. In the first experiment, we generated AUTs using RUGRAT's default parameter ranges, which model the properties of typical third-party applications. In the second experiment, we widened the parameter ranges to also allow for values that are only found rarely in third-party applications (but are still possible according to the empirical data described in Section 3.2).

For both experiments we used RUGRAT to generate AUTs of various sizes, ranging from some 10kLOC to 5MLOC. Specifically, we picked 7 LOC sizes (10k, 50k, 100k, 500k, 1M, 2.5M, and 5M) and generated several AUTs for each LOC value. (Due to implementation limitations the actual LOC of an AUT may deviate slightly from the target value.) For the first experiment we used RUGRAT to generate 10 random AUTs per LOC value, yielding 70 AUTs. For the second experiment we just generated a single AUT per LOC value, yielding 7 AUTs. In this initial experimentation, we only generated single-threaded applications. We ran all experiments on a HotSpot 1.6.0_24 JVM on Windows XP on a 2.33GHz 64-bit Xeon processor with 32GB RAM.

5.1 Program Analysis and Testing (RAT) Tools

On each of the 77 generated AUTs, we applied four Java program analysis tools: three static analysis tools, FindBugs, PMD, and JLint, and one dynamic analysis tool, Randoop. These tools apply different techniques in analyzing programs and produce various kinds of warnings. Such program analysis tools are typically highly configurable. To approximate the behavior of the tools under different configurations, for each tool we set a minimum and a maximum configuration. In the minimum configuration, we try to evoke a minimum amount of tool features; in the maximum configuration, we try to invoke all tool features.

FindBugs[2] applies syntactic bug patterns and dataflow-analysis on AUT bytecode to find bugs. It supports custom patterns and is easily expandable. For the configurations, we used two flags ('effort' and 'reportLevel'). For the maximum configuration, we set 'effort' to maximum and 'reportLevel' to 'low', which reports all the found bugs. In the minimum configuration, we set 'effort' to minimum and 'reportLevel' to 'high', to restrict reporting to high priority bugs.

PMD[3] applies syntactic bug patterns on AUT source code. It supports custom bug patterns (called ruleset) and is easily expandable. For the minimum configuration, we enabled only ruleset 'basic'. For the maximum configuration, we also enabled rulesets braces, clone, codesize, controversial, coupling, design, imports, naming, strictexception, strings, typersolution and unusedcode. Descriptions of these ruleset are in the PMD manual.

Like FindBugs, **JLint**[4] applies syntactic bug patterns and dataflow analysis on AUT bytecode, but it is not easy to expand [22]. JLint has patterns for detecting thread synchronization bugs, which we disabled in the minimum configuration. For the maximum configuration, we enable all patterns.

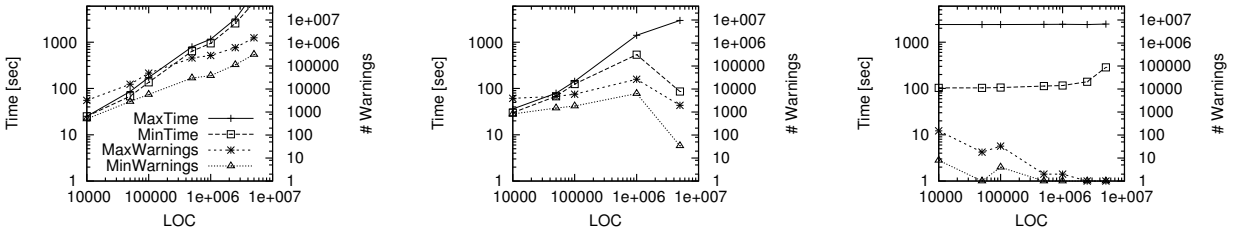
Randoop[5] applies feedback-directed test generation on AUT bytecode to deduce program behavior and create assertions to detect bugs. Randoop does not have any flags or configuration options we could set for our configurations. By default, it runs either for 100 seconds or until 100,000,000 tests are generated. We limit the timing to 100 seconds and 2,400 seconds (40 minutes) for the minimum and the maximum configurations, respectively.

[2]Version 1.3.9, http://findbugs.sourceforge.net/

[3]Version 4.2.5, http://pmd.sourceforge.net

[4]Version 2.3, http://artho.com/jlint

[5]Version 1.3.2, http://code.google.com/p/randoop



(a) FindBugs. (b) FindBugs skipping some classes. (c) Randoop.

Figure 2: Experimental results for RUGRAT-generated programs. The x-axis shows LOC on a logarithmic scale, the y-axis shows a RAT tool's runtime and the number of warnings it produced. 'Max' and 'Min' refer to our maximum and minimum RAT tool configurations. Each data point in 2(a) and 2(c) is the average of 10 AUTs from RUGRAT's default parameter range (experiment 1), 2(b) shows a single data point each from a wider range of configuration parameters (experiment 2).

5.2 RQ1: RUGRAT-Generated AUTs are Similar to Open-Source Applications

We collected 78 different software metrics for the generated programs and for 33 open-source applications that we selected from SourceForge. These open-source applications are downloaded many times, they are nontrivial, and they are actively evolving (the applications are listed on the tool website given in Footnote 1). Our goal was to determine if the generated applications differ from these open-source applications by software metrics. To do that, we used ANOVA to determine if there are significant differences w.r.t. the collected software metrics. The result shows with strong statistical significance that these applications differ from one another not only between different categories (i.e., open-source and generated), but also intra-categorically. That is, when applying pairwise t-tests, we determine that some applications from these two categories are highly similar to each other, while significant differences exist between applications within the same categories. To summarize this result, it is statistically impossible to tell whether an application is generated or written by programmers using such software metrics.

5.3 RQ2: Comparing RAT Tools

We performed 616 experiments by invoking 4 RAT tools in two configurations each on 77 generated AUTs. Figure 2 plots average tool runtime and warnings for each target LOC value. For space reasons we omit JLint and PMD and show the results of the second experiment (wider parameter ranges) only for FindBugs. In general we found that, for static analysis tools, execution time and the number of warnings increase with the program size (LOC). Three other observations are (1) JLint had the smallest execution time, followed by PMD and FindBugs. (2) JLint also produced the fewest warnings, followed by FindBugs and PMD. (3) PMD had almost the same runtime for both configurations.

5.4 RQ3: RUGRAT Found RAT Bugs/Issues

RUGRAT-generated programs let us independently rediscover several issues in RAT tools. While not dramatic, these results demonstrate the potential usefulness of RUGRAT.

FindBugs may skip classes and miss bugs. I.e., in the second experiment, which used wider parameter ranges, we encountered the situation depicted in Figure 2(b), where FindBugs did not show its usual execution time and warning behavior. Instead, it terminated quickly and reported only few warnings. Further investigation revealed that FindBugs has two limitations, which cause it to skip some code; i.e., if a class has more than 1,000 methods or is larger than 1MB, FindBugs declares it to be too large and skips it. In the generated AUT, the majorities of classfiles were larger than 1MB. FindBugs thus skipped almost the entire AUT and terminated quickly, reporting few warnings. FindBugs has no configuration option to prevent such skipping. We confirmed with the tool authors that the recommend solution is to instead modify the FindBugs source code.

One may argue that such limitations only affect analysis of generated programs. However, we have found real (manually written as well as generated but then manually edited) applications on SourceForge that have such large classes, including Apache Derby, DoctorJ, Drools, and OpenJDK. Reducing the number of methods for some of the applications caused FindBugs to report warnings where it was previously skipping the analysis.

While the other analysis tools generated more warnings for larger programs, **Randoop**, surprisingly, does the opposite; i.e., the larger the programs the fewer warnings Randoop generated (Figure 2(c)). We verified this behavior in a separate experiment, in which we increased the time allotted to Randoop's execution from 40 minutes to up to 8 hours, which would mirror an overnight run as part of an automated build and integration system. Doing so did not change the average number of warnings produced by Randoop.

Increasing the runtime to up to 8 hours also lead us to independently discover another issue with Randoop. This issue has been reported previously as Issue 14 in Randoop's issue tracking system[6]; i.e., in the test generation phase, if no test is generated after 10 seconds of the last generated test, Randoop terminates without writing any tests, not even the last generated test.

A third issue we discovered is that for larger programs, Randoop does not terminate after 100 seconds as it was supposed to in the default setting (our minimum configuration).

6. RELATED WORK

In this section, we focus on related random program generation techniques and tools, as we have already discussed the most closely related non-generated RAT benchmarks in Section 1. Grammar-based test input generation, pioneered by Hanford and Purdom [12, 21] in the 1970s, can be divided into two broad categories, random [18, 25] and systematic [10, 15, 17, 7]. In the following, we discuss pieces of related work in more detail that are either representative or closely related.

Csmith constructs legal C programs randomly using a subset of the C language production rules [28]. Specifically, Csmith consults a probability table, similar to our stochastic selection. Csmith

[6]http://code.google.com/p/randoop/issues/detail?id=14

systematically avoids generating programs that use language features classified as undefined or unspecified by the C language. To achieve the goal, CSmith employs selective construction and analysis of the generated programs. Csmith has been used to test compilers [28] and static analyzers [6]. Unlike RUGRAT, Csmith does not support object-oriented language features.

In the domain of object-oriented programs, a random program generator has been used to test Java just-in-time compilers [29]. This generator takes the number of desired classes and branches as input. Then, it generates branches and fills them randomly with bytecode instructions. In contrast to RUGRAT, this generator does not allow features such as recursive calls. Moreover, it was evaluated only on small programs with up to ten classes, ten methods per class, and less than 100 bytecode instructions per method. We were unable to obtain the tool to compare it with RUGRAT.

ASTGen [7] systematically generates small Java programs. However, it requires the user to combine several generators. More importantly, many generated programs have compile errors, and they do not have complex structures (e.g., only '==' is supported in conditions and no deep 'if-else' nesting is possible).

7. CONCLUSIONS

We propose a novel approach for generating random benchmarks for evaluating program analysis and testing tools using the concept of stochastic parse trees, where language grammar production rules are assigned probabilities that specify the frequencies with which instantiations of these rules will appear in the generated programs. We implemented our tool for Java and applied it to generate benchmarks with which we evaluated different program analysis and testing tools. Our tool was also implemented by a major software company for C++ and used by a team of developers to generate benchmarks that enabled them to reproduce a bug in less than four hours.

8. ACKNOWLEDGMENTS

We thank Balamurugan Prabakaran, Nischit Rangapan, and Arthi Vijayakumar from the University of Illinois for their contribution as part of their M.S. work. This material is based upon work supported by the National Science Foundation under Grants No. 0916139, 1017633, 1017305, and 1117369, as well as Accenture.

9. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Jan. 1986.

[2] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 169–190. ACM, Oct. 2006.

[3] S. M. Blackburn et al. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51(8):83–89, Aug. 2008.

[4] S. Cohen and B. Kimelfeld. Querying parse trees of stochastic context-free grammars. In *Proc. 13th International Conference on Database Theory (ICDT)*, pages 62–75. ACM, Mar. 2010.

[5] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Software—Practice & Experience*, 37(6):581–641, May 2007.

[6] P. Cuoq et al. Testing static analyzers with randomly generated programs. In *Proc. 4th NASA Formal Methods Symposium (NFM)*, pages 120–125. Springer, Apr. 2012.

[7] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proc. 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 185–194. ACM, Sept. 2007.

[8] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4), Oct. 2005.

[9] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, pages 149–168. ACM, Oct. 2003.

[10] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 206–215. ACM, June 2008.

[11] M. Grechanik et al. An empirical investigation into a large-scale Java open source code repository. In *Proc. 4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, Sept. 2010.

[12] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 1970.

[13] A. Joshi, L. Eeckhout, R. H. Bell, Jr., and L. K. John. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 5(2):10:1–10:33, Sept. 2008.

[14] K. Kanoun and L. Spainhower. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE, July 2008.

[15] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In *Proc. 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom)*, pages 19–38. Springer, May 2006.

[16] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann, Oct. 1999.

[17] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 134–143. ACM, Nov. 2007.

[18] P. M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990.

[19] G. McDaniel. *IBM Dictionary of Computing*. McGraw-Hill, Dec. 1994.

[20] K. S. Nash. Information technology budgets: Which industry spends the most?, Nov 2007.

[21] P. Purdom. A sentence generator for testing parsers. *BIT Numerical Mathematics*, 12(3):366–375, 1972.

[22] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *Proc. 15th International Symposium on Software Reliability Engineering (ISSRE)*, pages 245–256, Nov. 2004.

[23] R. H. Saavedra and A. J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Trans. Comput. Syst.*, 14(4):344–384, Nov. 1996.

[24] M. Schwab, M. Karrenbach, and J. Claerbout. Making scientific computations reproducible. *Computing in Science and Engineering*, 2(6):61–67, Nov. 2000.

[25] E. G. Sirer and B. Bershad. Using production grammars in software testing. In *Proc. 2nd Conference on Domain-Specific Languages (DSL)*, pages 1–13. ACM, Oct. 1999.

[26] D. R. Slutz. Massive stochastic testing of SQL. In *Proc. 24rd International Conference on Very Large Data Bases (VLDB)*, pages 618–622. Morgan Kaufmann, Aug. 1998.

[27] J. William A. Ward. Role of application benchmarks in the DoD HPC acquisition process. U.S. Army Engineer Research and Development Center, ERDC MSRC Resource, 2005.

[28] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294. ACM, June 2011.

[29] T. Yoshikawa, K. Shimura, and T. Ozawa. Random program generator for Java JIT compiler test system. In *Proc. 3rd International Conference on Quality Software (QSIC)*, pages 20–24. IEEE, Nov. 2003.

[30] H. Zhang and H. B. K. Tan. An empirical study of class sizes for large Java systems. In *Proc. 14th Asia-Pacific Software Engineering Conference (APSEC)*, pages 230–237. IEEE, Dec. 2007.