SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2016; **XXX**:1–28
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

RUGRAT: Evaluating Program Analysis and Testing Tools and Compilers with Large Generated Random Benchmark Applications†

Ishtiaque Hussain[1], Christoph Csallner[1*], Mark Grechanik[2], Qing Xie[3], Sangmin Park[4], Kunal Taneja[3], and B. M. Mainul Hossain[2]

[1]*University of Texas at Arlington, Arlington, TX 76019, USA*
[2]*University of Illinois, Chicago, IL 60607, USA*
[3]*Accenture Technology Labs, San Jose, CA 95113, USA*
[4]*Georgia Institute of Technology, Atlanta, Georgia 30332, USA*

SUMMARY

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 and compilers. 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 a set of large benchmark programs of up to 5M LOC each with which we evaluated different program analysis and testing tools and compilers. The generated benchmarks let us independently rediscover several issues in the evaluated tools.
Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

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 [2]. Benchmarks are important, since organizations and companies use different benchmarks to evaluate and choose mission-critical software for business operation [3]. 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 [4]. 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 [5].

†This article is an extended version of our previous work presented at WODA 2012 [1].
*Correspondence to: University of Texas at Arlington, Computer Science and Engineering Department, Arlington, TX 76019-0015, USA. E-mail: csallner@uta.edu

Copyright © 2016 John Wiley & Sons, Ltd.

Benchmarks are very important for evaluating *pRogram Analysis and Testing (RAT)* algorithms and tools [6, 7, 8, 9, 10]. Different benchmarks exist to evaluate different RAT aspects, such as how scalable RAT tools are, how fast they can achieve 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 [11]. 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 [8, 9]. Unfortunately, complex benchmark applications are very costly to develop [3, 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 [12]. 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 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 [14], since commercial benchmarks cannot be easily shared among organizations and companies for legal reasons and trade-secret protection. For example, Accenture Confidentiality Policy (item 69) [‡] states that source code, which is generated by the company and relates to its business, research and development activities, clients or other business partners, or employees are considered confidential information. Other companies have similar policies. Finally, it is often required to have more than one benchmark 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 [15]. Suppose there is a claim that a relational database engine performs better at certain aspects of SQL optimization than some other engines. 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 article, we propose a *Random Utility Generator for pRogram Analysis and Testing (RUGRAT)* for generating application benchmarks within the specified constraints and within a range of predefined properties. Our goal is to complement "real-world" application benchmark with synthetic ones, so that researchers and industry professionals have diverse options for evaluating program analysis and testing tools. RUGRAT is implemented for Java and it is used to evaluate different Java compilers and open-source RAT tools.

[‡] https://policies.accenture.com/Pages/0001-0100/0069.aspx

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

This article makes the following contributions:

- We apply stochastic parse trees for generating random application benchmarks. In stochastic parse trees, 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 RUGRAT for Java and used it to generate dozens of applications, ranging from 300 LOC to 5M LOC, to benchmark several versions of a popular Java source to bytecode compiler as well as popular program analysis and testing tools. This version of RUGRAT is open source software and available for download from the RUGRAT tool web site[§].

2. OUR APPROACH: LEVERAGING STOCHASTIC GRAMMARS TO GENERATE LARGE RANDOM BENCHMARK APPLICATIONS

In this section we present a model for RUGRAT and discuss our goals and approach for generating random object-oriented benchmark applications. Specifically, we review the stochastic grammar model [16, 17] that is at the core of our technique and discuss how we can apply the stochastic model to generating large-scale benchmark applications that resemble real-world applications. Then, we list the benefits of our approach over handwritten programs.

2.1. Background: 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 [18]. 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 [19], and also in generating SQL statements for testing database engines [15]. We use a *stochastic grammar model* to generate large random object-oriented programs.

Random programs are constructed 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 the rules' probabilities. 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.

In addition to the rules that are found in a typical context-free grammar of a programming language, our approach takes into account additional rules and constraints that are imposed by the programming language specification. For example, a variable has to be defined before it can be used and a non-abstract class in an object-oriented program has to implement all abstract methods it inherits from its super-types. With such an enhanced 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 for evaluating RAT approaches and tools. We do not see

[§]https://sites.google.com/site/rugratproject/

Copyright © 2016 John Wiley & Sons, Ltd.

RUGRAT as a replacement of real-world applications for evaluating RAT approaches and tools. We rather see RUGRAT as a tool that enables experimenters to quickly generate a large number of nontrivial application benchmarks that have desired properties. The goal of RUGRAT is thus to supplement evaluations of RAT tools using not only real-world application benchmarks but also synthetic ones that are generated on demand using a set of constraints. 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 achieve our goal, we should address several issues. First, generated programs must have a wide variety of language constructs that are important for evaluating RAT approaches and tools. Sample constructs include recursion, dynamic dispatch, and array manipulations using expressions that compute array indices that test the boundaries of RAT algorithms. Existing program generators often do not take into consideration such language constructs and do not add them to generated programs.

In addition, we adhere to a requirement that generated programs and handwritten programs should trigger similar behaviors in RAT tools. This requirement is motivated by the needs of the potential RUGRAT users. That is, we expect that RAT tool developers and RAT tool users care most about the performance of RAT tools on real-world programs, since processing real-world programs is likely going to be the main use case of these RAT tools.

While stress-testing a RAT tool with unusual programs is also important, we expect RUGRAT users to care most about RAT tool performance on programs that are more mainstream. In this regard we expect RUGRAT users to be similar to RAT tool users. RAT tool users have frequently complained about RAT tools generating input values for the analyzed program that appear exotic and do not represent expected program behavior as well as about warnings on bugs that cannot occur or can only occur in very rare situations [20, 21, 22, 23].

As a consequence, we enable experimenters to tune the default parameters of RUGRAT such that generated programs are as similar as possible to what one would consider a normal handwritten program. We implement this issue by varying the probabilities that are assigned to different production rules of the language grammar. RUGRAT users can diverge from our default parameters to produce more exotic kinds of programs. In our evaluation we explore an example use of RUGRAT with non-standard parameters.

2.3. *Benefits*

Our approach scales to generating programs that are large, have complex properties, and can trigger similar RAT tool behaviors as handwritten programs (see the comparison results in Section 6). While RUGRAT-generated programs are similar to handwritten programs, our approach provides multiple benefits over benchmarking with handwritten applications.

First, using RUGRAT one can easily generate a large variety of random programs. Such a large set of programs can complement existing suites of handwritten benchmark programs, which are often relatively small sets of programs. For example, the well-known Siemens suite [11] consists of a few small programs and could benefit from a large set of complimentary applications that cover a range of program configuration points.

Second, RUGRAT-generated programs have a designated entry point or main function. In contrast, handwritten programs such as libraries typically lack such a clear entry point, which forces many RAT tool developers to write test harnesses for RAT tool evaluation [24]. Each RUGRAT-generated program has a dedicated main method that can be used to start the program directly and does not require a test harness.

Third, our approach scales down from realistic applications to toy applications that only contain a specified set of language features. This down-scaling is useful during RAT tool development. That is, at an early RAT tool development stage, a RAT tool may only be able to handle a few programming language features. At this point a RAT tool developer may still want to test her RAT tool on large applications. However handwritten applications often use multiple language features and it may be hard to find handwritten applications that only use a given set of language features, especially when looking for a variety of larger applications.

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

A good example is the task of benchmarking compilers of different versions of the compiled programming language. For this task we clearly need benchmark programs that can be compiled by all participating compilers. Since programming languages tend to support programs coded in prior versions of the language, this task limits generated programs to the subset of language features supported by the oldest compiler. However it is not easy to find in open source project repositories many large handwritten applications that only use features of, say, Java 1.2.1. (This may be due to project developers incorporating new Java language features into their Java applications as these features become available in new versions of the Java language.) On the other hand, it is trivial to generate many such applications with RUGRAT, in various sizes and using various subsets of the language.

Fourth, as another special case of down-scaling, RUGRAT is useful for generating programs that have no external dependencies. In contrast, most realistic handwritten applications have external dependencies such as on external libraries. Such external dependencies often complicate RAT tool operation, even for industrial-strength RAT tools. For example, in a recent study on why the industrial-strength Pex dynamic symbolic execution (DSE) tool achieved less than perfect branch coverage, it is discovered that more than a quarter of the missed branches were due to calls to external libraries such as native code [25]. To analyze such applications, DSE tool developers sometimes have to resort to writing mock versions or models of the external dependencies, which is tedious and laborious [26]. In contrast, programs generated by RUGRAT do not have external dependencies, so DSE tool developers can easily compare the scalability of their tools on benchmark applications.

Fifth, since RUGRAT-generated applications do not have external dependencies it is very easy to compile, install, execute, and test RUGRAT-generated applications. In contrast, handwritten programs are often difficult to install and execute. For example, before a realistic handwritten application can be tested, external dependencies have to be resolved. For example, additional systems such as databases, servers, and communication infrastructure have to be installed and configured. A survey on evaluating static analysis tools and benchmarks showed that most user-reported failures in software repositories are false failures, i.e., failures that will not be fixed as they do not concern the code [27]. Indeed, the false failures are mostly installation failures, which may be caused by poor documentation and difficult deployment procedures. RUGRAT users avoid this potential pitfall as RUGRAT-generated applications do not require any installation or configuration and can be compiled and executed immediately.

Given the difficulties inherent in using handwritten programs for benchmarking, it is maybe not surprising that existing comparisons of RAT tools have mainly focused on using small to medium-sized subject programs for benchmarking. That is, the empirical comparisons we are aware of are limited to subject sizes of small test cases [28, 29, 30], less than 150k LOC [21, 23, 27], or less than 500k LOC [31]. With RUGRAT it is easy to generate subject applications that contain several million lines of code (see Section 4.2).

Finally, our approach enables more experiments at lower cost by providing, on demand, many high-quality programs in short time. When evaluating a RAT tool with handwritten programs, the RAT developer needs to explore code repositories with specific requirements if more programs are desired, which can be time-consuming. However, with RUGRAT the developer can generate such programs automatically in a short amount of time (see the RUGRAT resource consumption evaluation in Section 4.2), by specifying such requirements as parameters to RUGRAT.

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.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

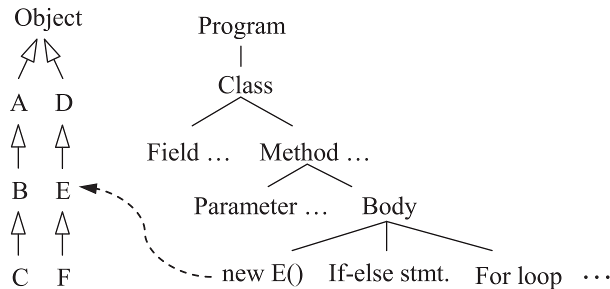


Figure 1. Left: Example RUGRAT-generated Java subtype hierarchy, rooted at the Java base type Object. Right: Parts of an example abstract syntax tree RUGRAT generated from a stochastic grammar of the Java programming language. RUGRAT picks types (such as class E) from such a subtype hierarchy randomly to create instances and call methods on these instances.

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 syntactically 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. As we want to generate benchmark applications, an important additional constraint is that RUGRAT-generated programs should resemble real-world programs relatively closely. In the following we briefly discuss how we solve these key challenges.

3.1. Supporting Complex 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 Foo must be initialized directly or by each constructor of Foo; 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. That is, 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 can enforce that only non-zero valued expressions occur in the denominator of a division operation. To avoid infinite loops RUGRAT only uses *for* loops with literals in the loop condition. RUGRAT 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.

There are certain limitations in our current RUGRAT implementation. For instance, RUGRAT currently only supports primitive types for fields. Our prototype also does not generate calls to Java library methods. Finally, several advanced language features such as *generics* have not been implemented yet. All of these are subject to future work.

3.2. Configuration Options

To satisfy different requirements for generated programs, 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

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

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 [32, 33, 34]. For example, to determine the number of classes, we follow Zhang et al.'s observation that LOC is roughly 114 times the number of classes in a program [33] and set $classes = LOC/114$. To define the number of interfaces, we follow the observation of Collberg et al. [34], that each package in a program has roughly 12 classes and there is one interface per package. Hence we set $interfaces = LOC/(114 * 12) = LOC/1,368$. Grechanik et al. [32] found that the average value for the 'maximum number of methods per interface' is 3.4, and thus we took ten times the average value and set the upper limit of the range to 34. Collberg et al. [34] 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 RUGRAT tool website has a complete list of the configuration parameters and their default values.

3.3. Choice of RUGRAT Target Language: Java Source Code

The Java language is different from many earlier languages such as C and C++ in that Java has an explicit well-defined intermediate language, the Java bytecode language. While every legal Java source code program can (by definition) be compiled to a Java bytecode program, there are many legal Java bytecode programs that do not have a direct correspondence in Java source code. When generating random Java programs, there is thus a trade-off between generating Java source code and generating Java bytecode programs.

While other program generators for Java have targeted the Java bytecode language [35, 36], RUGRAT generates programs as Java source code, for the following reasons. First, when generating Java source code it is easier from an engineering perspective to write a generator that only generates legal Java programs. Since we already have a stochastic grammar, by construction every generated program satisfies the grammar. However, when basing the generator on the Java bytecode language, we would need additional functionality to ensure that each generated bytecode program is also a legal Java program.

Second, generating Java source code makes it easier to benchmark RAT tools that operate on Java source code. While many Java RAT tools take as input Java bytecode programs, there are also RAT tools that work on source code, such as style checkers, bug pattern detectors, and source to bytecode compilers. Concrete example tools for Java are Checkstyle, PMD, and various compilers. On the other hand, benchmarking such tools on generated bytecode programs seems cumbersome as such generated bytecode programs would first have to be translated back to source form, for example, via a bytecode disassembler.

Finally, we expect that to a lot of Java developers generated applications in Java source code format will be more attractive and useful than generated Java bytecode programs. The key reason is that many Java developers are more familiar with the Java source language, as many developers do not normally interact with the Java bytecode language. We expect that this higher level of familiarity will be important when developers attempt to investigate the cause of a given benchmark result, attempt to change generated applications, and use generated code for testing and debugging.

4. OVERVIEW OF EXPERIMENTS WITH RUGRAT-GENERATED APPLICATIONS AS RAT TOOL BENCHMARKS

In this section and the following two sections we describe our experience with our RUGRAT prototype implementation for Java. That is, we conducted several experiments to explore the usefulness of RUGRAT for the evaluation and benchmarking of Java RAT tools.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

At a high level, a prospective Java RAT tool user is interested in comparing and benchmarking RAT tools and may be asking several questions. For example, how do Java RAT tools behave when running on applications of different sizes? This question is especially relevant if experiments published to date do not cover the kind of input application sizes that are relevant for the prospective RAT tool user [21, 23, 28, 27, 37, 29, 30, 31].

Specifically, before acquiring a particular Java RAT tool, a prospective RAT tool user may be wondering if a given Java RAT tool will break down for large input applications, while a competitor RAT tool may scale to such applications. A secondary question is how for different input application size categories the time and memory requirements of various RAT tools compare against each other.

Before using a new tool such as RUGRAT, a potential user may want to know how expensive RUGRAT is in terms of computational resources. Part of the appeal of RUGRAT is that RUGRAT can save the user time, as installing handwritten programs can be very time intensive. Then a natural question is how much time it takes RUGRAT to generate programs.

When using a random benchmark program generator such as RUGRAT to explore such questions, it is important to determine if the RUGRAT-generated applications and handwritten applications trigger similar behaviors in RAT tools. To answer this question we compare RAT tool behavior on RUGRAT-generated programs with a baseline of handwritten programs.

Another interesting question is whether RUGRAT-generated applications provide benefits that handwritten applications do not provide. Section 2.3 lists such benefits and Section 5.2 describes a concrete case in which RUGRAT can generate benchmark programs when handwritten programs may not be easily available.

We thus explore the following concrete research questions.

- **RQ1.** How many computational resources (i.e., execution time, main memory and disk space) does the RUGRAT random benchmark program generator require?
- **RQ2.** Can RUGRAT-generated applications be used for focused benchmarking of existing Java RAT tools—i.e., compilers and static and dynamic program analysis tools?
  - **RQ2a.** Can RUGRAT-generated applications be used for benchmarking the execution time and memory requirements of existing Java source to bytecode compilers?
  - **RQ2b.** Can RUGRAT-generated applications be used for benchmarking the execution time and amount of output of existing static and dynamic Java program analysis tools?
- **RQ3.** Can RUGRAT-generated applications find defects in program analysis tools?

Besides the difference between compilers and RAT tools, the two benchmarking research questions RQ2a and RQ2b differ in another aspect. The reason for this difference is that it is hard to establish a good baseline of large handwritten programs that can be compiled with many versions of a standard Java source to bytecode compiler (see Section 5.2). Such a baseline is much easier to establish for current RAT tools (RQ2b), as these RAT tools work on recent versions of the Java language and therefore support many modern language features (see Section 6.2).

Recall that the goal of RUGRAT is not to supplant all other ways of RAT tool benchmarking (Section 2.2). Instead, RUGRAT aims at enabling a focused benchmarking of specific RAT tool features. A complete benchmarking of existing Java RAT tools is therefore outside the scope of this article. A complete benchmarking would also address important issues such as RAT tool installation and maintenance requirements, the precision and recall of the RAT tool outputs, and ease of use. We leave such issues for future work. Instead, in the following we focus on selected features of RAT tools that are easy to measure, such as the quantity of RAT tool outputs as well as RAT tool memory consumption and execution time.

4.1. Experimental Setup

We used RUGRAT to generate applications of various sizes, ranging from some 10k LOC to 5M LOC. Specifically, we picked 7 target application sizes given in non-comment, non-blank lines of code (LOC), i.e., 10k, 50k, 100k, 500k, 1M, 2.5M, and 5M and generated several applications for each target LOC size. (Due to implementation limitations the actual LOC of an AUT may deviate from the target value.)

Copyright © 2016 John Wiley & Sons, Ltd.

Prepared using speauth.cls
DOI: 10.1002/spe

Table I. RUGRAT execution time, main memory, and disk space consumption when generating programs of various sizes on a standard desktop computer. Later experiments in Section 6 were performed on a more powerful machine. Program sizes (10k, 50k, etc.) are given in LOC. All aborts are due to RUGRAT running out of main memory.

Resource		10k	50k	100k	500k	1M	2.5M	5M
Time [s]	Avg.	7	26	73	440	794	5,934	3,259
	Best	2	6	19	113	113	383	227
	Worst	12	49	230	798	1,867	20,696	6,256
Mem [MB]	Avg.	20	48	105	415	575	847	1,327
	Best	16	23	42	174	286	361	866
	Worst	25	81	341	826	997	1,060	1,498
Disk [MB]	Avg.	1	5	12	55	72	173	262
	Best	<.4	2	4	23	40	110	137
	Worst	2	12	47	109	130	315	315
Abort		0	0	0	0	0	1	5

In the context of benchmarking RAT tools, we refer to RUGRAT-generated applications and handwritten applications used as benchmarks as *applications under test* or just AUTs. For these experiments we generated only single-threaded applications. Extending RUGRAT to generate multi-threaded applications is a subject of future work.

To capture RUGRAT's behavior on a standard desktop computer, we performed most experiments on a 32-bit Windows 7 OS running on a 2.5GHz AMD dual core processor with 4GB RAM. This machine is a few years old, but it is well suited to explore the scalability limitations of the current RUGRAT implementation. With these limitations established we switched to a more powerful machine for subsequent experiments that benchmark existing RAT tools on RUGRAT-generated applications. The latter machine has more computation power with 32GB RAM and is running a 64-bit Windows XP OS on a 2.33GHz Xeon processor (see Section 6).

4.2. RQ1: RUGRAT Resource Consumption: The Current RUGRAT Tool Implementation Can Be Used On A Standard Desktop Computer

Table I summarizes the resource consumption of our RUGRAT prototype. For each column or LOC category (10k, 50k, etc.), we used RUGRAT and its default parameter ranges to generate 10 random programs. For 2.5M LOC we generated 9 programs, as one attempt aborted with an out of memory exception. Similarly, for 5M LOC five attempts were aborted with out of memory exceptions. This data indicates that the current RUGRAT implementation scales to about 2.5M LOC on a standard desktop computer (a 2.5 GHz machine with 4 GB RAM).

For each column or group of 10 RUGRAT executions, the table shows the average, best, and worst consumption of time, main memory[¶], and disk space of that group. Lower numbers are better as they indicate lower resource consumption. Disk space consumption is the space that is required to store a generated application.

In our experiments the size of the generated applications grew about linearly with the target LOC size. The data further suggests that the average RUGRAT execution time currently does not scale linearly with respect to LOC. In general, the RUGRAT tool implementation is currently not optimized for either speed or (main) memory consumption and we expect that these aspects can be improved with more engineering work.

[¶]We used the Windows 7 default performance monitor PerfMon to log memory usage.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

5. EXPERIENCE WITH RUGRAT-GENERATED APPLICATIONS AS BENCHMARKS FOR JAVA SOURCE TO BYTECODE COMPILERS

In this section we describe our experience with compiling both handwritten and RUGRAT-generated programs with several versions of the standard JDK Java source to bytecode compiler (RQ2a).

5.1. Experimental Setup

We obtained 8 versions of the Java development kit (JDK) from the Oracle Java Archive[‖], i.e., versions 1.2.1, 1.2.2, 1.3.0, 1.3.1, 1.4.0, 1.5.0, 1.6.0, and 1.7.0. Each downloaded development kit contains a default Java source to bytecode compiler. These 8 compilers are listed in Tables II and III by their JDK version jY.Z, i.e., as j2.1, j2.2, etc., omitting the common top-level version 1 identifier.

Since some of the older Java development kits were only available in 32-bit versions, we conducted all experiments on our standard desktop computer, a 32-bit Windows 7 OS running on a 2.5GHz AMD dual core processor with 4GB RAM. As on this machine our current RUGRAT random benchmark program generation prototype does not scale to generating 5M LOC programs (Section 4.2) we used a more powerful machine to run RUGRAT and supply us with a total of 70 subject programs, 10 in each LOC size category.

For the experiments we configured each compiler to use the maximum amount of memory (heap space) that was possible on our machine for that particular compiler. As a side note, the compiler options for setting this maximum amount of memory changed between compiler versions and the corresponding maximum amount that could be set on the machine also fluctuated between compiler versions, i.e., between 1.15 GB (for j3.0) and 2 GB (for j2.1 and j2.2). The remaining compilers accepted a maximum of either 1.6 GB (for j4.0) or 1.5 GB (j5.0, j6.0 and j7.0).

For creating a baseline of compiling handwritten programs, we are guided by the following four goals. (1) First, we want to keep our experiments as reproducible as possible. We therefore use programs from a major open-source program repository. (2) Second, since one of the goals of RUGRAT is to generate programs of a wide variety of program sizes (i.e., ranging from a few thousand to a few million LOC), the baseline would ideally use a similar variety of program sizes. (3) Third, we want to minimize selection bias and therefore use random sampling where possible. If random sampling does not cover an important criterion, we add other selection strategies. (4) Finally, we have to keep the experiments feasible and therefore limit the number of programs in our experiments.

To create a baseline of compiling handwritten programs, we used the third-party SF100 and SF110 random samples of all Java projects on SourceForge [38]. SourceForge is a large open-source program repository. SF100 was created by randomly sampling 100 projects from the 48,109 projects on SourceForge tagged as "Java". For each selected project the latest version was checked out of the repository. SF110 contains SF100 plus the latest versions of the 10 most popular Java programs on SourceForge.

To compare RUGRAT-generated programs with handwritten programs, we treated all non-JDK libraries required by a handwritten program as a part of the program. This allows a fair comparison as RUGRAT-generated programs contain all required code and do not depend on external libraries. As a result, the LOC counts we reported for handwritten programs may differ substantially from the literature, which typically does not include required libraries in LOC counts.

5.2. Baseline: Comparing Java Source to Bytecode Compilers on Handwritten Applications

To compare different compiler versions, we need subject programs that all compilers can compile. This fixes the set of allowed language features to the intersection of the features supported by all compilers. A baseline for research question RQ2a thus requires handwritten Java programs that can be compiled with all compilers listed in the experimental setup of Section 5.1.

[‖] http://www.oracle.com/technetwork/java/archive-139210.html

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table II. Failed (×) and successful (✓) compilation of three groups of handwritten programs with standard JDK Java source to bytecode compilers: A random sample of all SourceForge projects (top), a random sample of the 10 most popular SourceForge projects (middle), and early releases of major open-source projects (bottom). Some programs did not contain all libraries required for compilation (-).

Subject	LOC	j2.2	j3.0	j4.0	j5.0	j6.0	j7.0
templatedetails	282	×	×	×	✓	✓	✓
omjstate	387	×	×	×	×	✓	✓
imsmart	1,060	×	×	×	×	✓	✓
bpmail	1,354	×	✓	✓	✓	✓	✓
saxpath	1,919	✓	✓	✓	✓	✓	✓
jni-inchi	2,108	×	×	×	✓	✓	✓
a4j	2,787	×	✓	✓	✓	✓	✓
dsachat	2,993	×	×	×	✓	✓	✓
javaviewcontrol	3,844	×	×	×	✓	✓	✓
beanbin	4,332	×	×	×	✓	✓	✓
water-simulator	5,433	×	×	×	✓	✓	✓
javabullboard	7,520	×	✓	✓	✓	✓	✓
schemaspy	8,038	×	×	×	✓	✓	✓
gangup	8,607	×	×	×	✓	✓	✓
jdbacl	14,520	×	×	×	×	✓	✓
netweaver	19,316	×	×	×	✓	✓	✓
firebird	43,159	×	×	×	×	✓	✓
sweethome3d	50,837	×	×	×	✓	✓	✓
JMeter 1.0.2	765	✓	✓	✓	✓	✓	✓
Ant 1.1	5,749	-	-	×	×	×	×
Log4j 1.0.4	7,175	-	-	×	×	×	×
Tomcat 3.0	17,120	-	-	-	×	×	×

To obtain handwritten programs for such a baseline, we explored the following three avenues. (1) First, we took a random sample of all Java projects from a large open-source project repository. That is, we took a random sample of the SF100 random SourceForge sample. However, a large open-source repository such as SourceForge may contain many low-quality toy projects that are not representative of industry-grade programs. (2) To address this issue, we also examined a random sample of the most popular Java projects on SourceForge. That is, we used a random sample of the 10 most popular Java SourceForge projects included in SF110. The first two avenues only capture the latest (current) version of open-source projects. One may suspect that these latest versions use recent Java language features such as parametric polymorphism (generics) and therefore cannot be compiled by the older compilers listed in Section 5.1. To address this issue we also looked for programs that have been developed before these newer language features became available. (3) Third, we therefore examined the oldest still-available releases of several well-known early major open-source Java projects, i.e., Ant, JMeter, Log4J, and Tomcat.

Table II summarizes the results. In summary, we were not able to locate a good set of handwritten programs that would allow us to benchmark the compilers listed in Section 5.1. From our subjects only two programs (saxpath and JMeter) worked on all compilers. Three programs worked on j3.0 and later (a4j, bpmail, and javabullboard). The remaining programs required j5.0 or later, or they did not compile at all.

This result may be explained by Java programmers using modern language features in their code. Such new features typically cannot be compiled by older compilers. However, there are three programs that did not work with new compilers either (Ant, Tomcat, and Log4J). These programs use identifiers that in later Java versions became keywords, i.e., *enum* and *assert*. These programs also do not include in their distribution all libraries that are required to compile the program.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

The two handwritten programs that compile with all compilers (saxpath and JMeter) are very small, with under 2k LOC each. In the RUGRAT experiments of Section 5.3, benchmark programs range from some 10k LOC to some 5M LOC. So within the scope of this article we could not establish a good baseline of handwritten programs. On a side note, none of these two programs (saxpath and JMeter) compiled with a Java 1.1 compiler.

With various amounts of effort, better-suited handwritten programs may be found and non-compiling programs may be salvaged. For example, identifiers named *enum* or *assert* could be systematically renamed. The required old versions of missing libraries may be located somewhere on the Web. Program source code could be rewritten to reduce its reliance on modern language features and libraries. In the extreme, benchmark applications could be manually implemented from scratch. We consider the need for such costs a motivation for an automated program generator such as RUGRAT. To compare RUGRAT-generated programs with a baseline, Section 6 compares RAT tools by how they behave on RUGRAT-generated and handwritten programs.

5.3. RQ2a: Comparing Java Source to Bytecode Compilers on RUGRAT-Generated Programs

Table III shows the absolute execution time and main memory consumption of the subject compilers when compiling our 70 RUGRAT-generated subject programs. For both execution time and memory consumption the table shows average ($\ominus$), maximum ($\top$), and minimum ($\bot$) measurements. Lower values are better, as they indicate lower resource consumption.

Figure 2 plots these values to show the trends in these data. Each plot in Figure 2 shows the compiler execution time and memory consumption for all compilers on the 10 subject programs of an AUT size category. The plot uses box and whiskers to show minimum, lower quartile, median, upper quartile, and maximum values.

From the results we can make several observations. First, maybe expected, the newer the compiler the more likely it can compile more of the generated applications, including the very large ones. Each case in which a compiler failed to compile a subject is noted in Table III as “did not compile” (dnc). In our experiments each dnc case was caused by the compiler running out of available (heap) memory. In other words, the older the compiler, the more likely it ran out of memory when attempting to compile some of the largest applications in our sample.

Specifically, the newest compilers, j5.0, j6.0, and j7.0, could compile the most applications and, on the sample size of 70 generated applications, failed to compile only 13 applications. The next older compiler, j4.0, failed to compile 14 applications; the next older compiler, j3.1, failed to compile 15; j3.0 failed to compile 22; j2.2 failed to compile 32; and j2.1 failed to compile 34. Table III shows that the applications that could not be compiled are mostly in the 2.5M and 5M LOC categories.

While for this experiment we ran RUGRAT with its default parameters, the results of the experiment, somewhat surprisingly, resemble the results of stress-testing. That is, although all generated programs use common combinations of language features, the tested compilers could not compile all programs. This effect increased with the size of the generated programs and was most pronounced for the largest generated programs, i.e., in the 5M LOC category.

Second, on the subjects up to and including 2.5M LOC that each compiler could compile, the newer compilers had the highest average memory consumption. Specifically, j7.0, had the highest average memory consumption, followed closely by the next older compiler, j6.0. On individual subjects the older compilers j4.0 and j3.1 had higher memory consumption than their newer peers, but on average these older compilers consumed less memory.

Third, j2.2 was the slowest compiler with the highest average compile time for most AUT size categories. The exceptions are the smallest and possibly the largest AUTs, since for the largest AUTs (2.5M) this compiler did not compile a single subject. A close second slowest was the predecessor compiler j2.1, with the same caveats for the smallest and largest AUT sizes.

Finally and maybe somewhat surprising, for the small and medium sized AUTs of up to 100k LOC, the fastest compiler was j3.0. For these AUT size categories, the average compile speed of j3.0 was between 50 and 60% of the newer j7.0 baseline compiler. This trend continues to larger AUTs of up to 1M LOC if we only consider the applications the respective compilers could compile.

Copyright © 2016 John Wiley & Sons, Ltd.

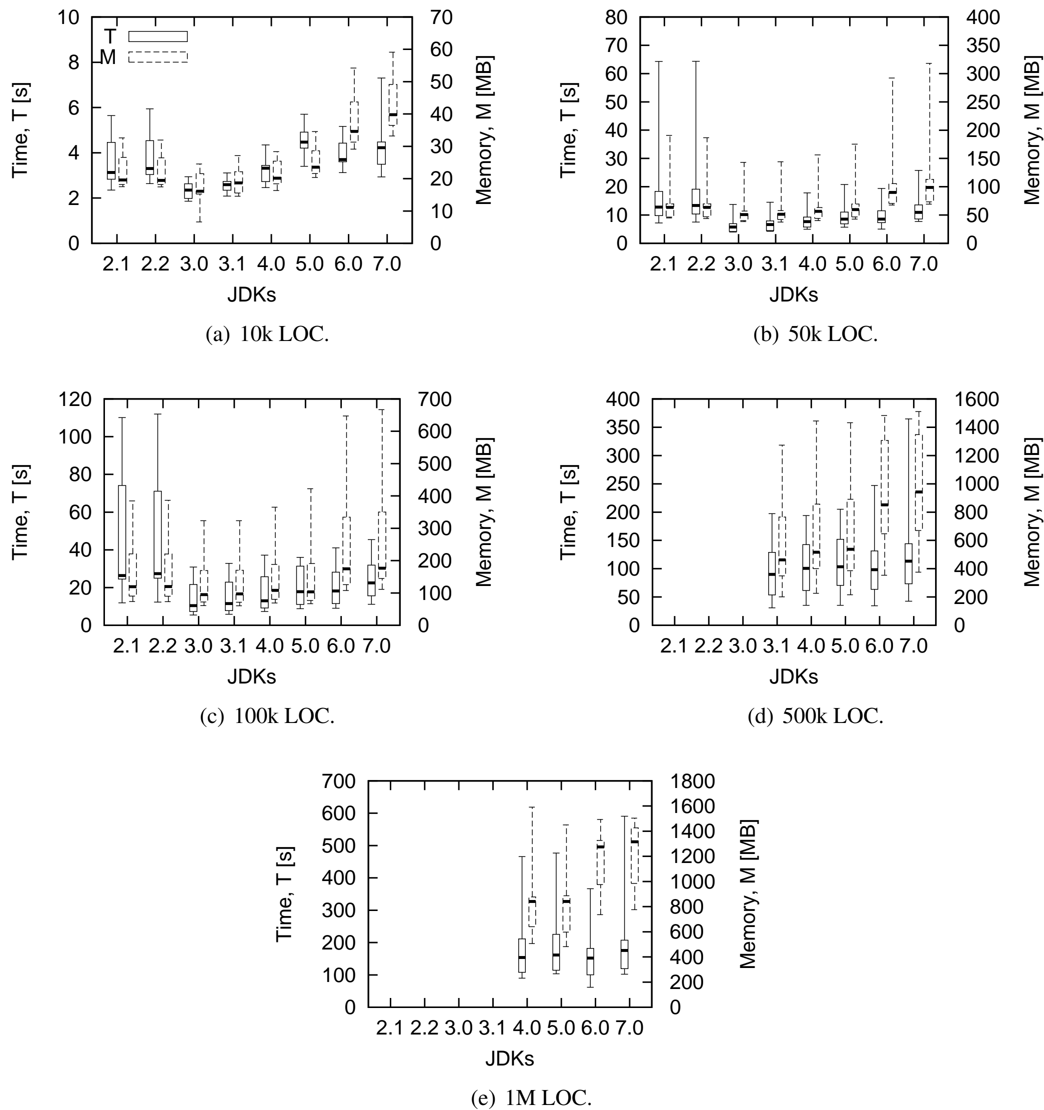
DOI: 10.1002/spe

Table III. Average ($\ominus$), maximum ($\top$), and minimum ($\bot$) absolute execution time (t) and main memory consumption (m) of standard JDK Java source to bytecode compilers compiling RUGRAT-generated programs. Lower values are better, as they indicate lower resource consumption. Did not compile (dnc) is the number of programs a compiler could not compile.

LOC			j2.1 [s,MB]	j2.2 [s,MB]	j3.0 [s,MB]	j3.1 [s,MB]	j4.0 [s,MB]	j5.0 [s,MB]	j6.0 [s,MB]	j7.0 [s,MB]
10k	t	$\top$	6	6	3	3	4	6	5	7
		$\ominus$	4	4	2	3	3	4	4	4
		$\bot$	2	3	2	2	2	3	3	3
	m	$\top$	33	32	25	27	28	35	54	59
		$\ominus$	22	22	17	19	21	25	38	43
		$\bot$	18	17	7	15	16	20	29	33
50k	t	$\top$	64	64	14	14	18	21	19	26
		$\ominus$	18	19	6	7	8	10	10	12
		$\bot$	7	8	4	4	5	6	5	8
	m	$\top$	191	187	143	144	156	175	292	318
		$\ominus$	76	76	61	62	68	72	114	121
		$\bot$	45	44	38	38	40	43	68	69
100k	t	$\top$	110	112	31	33	37	36	41	45
		$\ominus$	46	47	15	16	18	20	21	25
		$\bot$	12	12	5	6	7	9	9	11
	m	$\top$	385	386	323	324	365	423	648	667
		$\ominus$	166	166	131	131	147	152	247	258
		$\bot$	73	73	61	60	69	67	107	111
500k	t	$\top$	125	350	166	197	194	205	247	365
		$\ominus$	108	179	81	98	106	112	110	138
		$\bot$	76	82	31	31	35	35	34	43
	m	$\top$	398	519	969	1,273	1,444	1,432	1,483	1,510
		$\ominus$	333	406	510	593	665	676	948	987
		$\bot$	249	249	201	201	226	216	353	376
		dnc	7	5	1	0	0	0	0	0
1M	t	$\top$	237	236	204	481	466	477	367	591
		$\ominus$	218	220	129	177	183	194	159	208
		$\bot$	188	195	80	81	90	104	62	102
	m	$\top$	519	519	953	1,430	1,591	1,450	1,493	1,504
		$\ominus$	519	518	668	761	856	813	1,182	1,212
		$\bot$	518	518	437	445	507	482	737	776
		dnc	7	7	2	1	0	0	0	0
2.5M	t	$\top$	n/a	n/a	n/a	615	729	802	583	1,021
		$\ominus$	n/a	n/a	587	461	566	567	302	562
		$\bot$	n/a	n/a	n/a	288	302	302	125	181
	m	$\top$	n/a	n/a	n/a	1,532	1,631	1,522	1,524	1,505
		$\ominus$	n/a	n/a	930	1,355	1,484	1,374	1,487	1,486
		$\bot$	n/a	n/a	n/a	961	1,077	1,027	1,364	1,392
		dnc	10	10	9	5	5	4	4	4
5M	t	$\top$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		$\ominus$	n/a	n/a	n/a	919	1,064	1,031	214	462
		$\bot$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
	m	$\top$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		$\ominus$	n/a	n/a	n/a	1,492	1,569	1,469	1,496	1,405
		$\bot$	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
		dnc	10	10	10	9	9	9	9	9

Copyright © 2016 John Wiley & Sons, Ltd.

Softw. Pract. Exper. (2016)
DOI: 10.1002/spe



(a) 10k LOC. (b) 50k LOC.

(c) 100k LOC. (d) 500k LOC.

(e) 1M LOC.

Figure 2. Time (left y-axis) and memory (right y-axis) consumption of standard JDK source to bytecode compilers compiling RUGRAT-generated programs. A compiler is not plotted in a program size category if the compiler could not compile all programs in that category. No compiler could compile all 2.5M or 5M LOC programs.

For these larger AUTs the average compile time of j3.0 fluctuated between 70 and 80% of the j7.0 baseline compiler. However, for the largest AUTs of 2.5M LOC, j3.0 failed to compile 9 of 10 AUTs and for the one AUT it did compile j3.0 needs more than three times the compile time than the j7.0 baseline compiler.

6. EXPERIENCE WITH RUGRAT-GENERATED APPLICATIONS AS BENCHMARKS FOR STATIC AND DYNAMIC JAVA PROGRAM ANALYSIS TOOLS

To explore the two RAT tool research questions (RQ2b and RQ3), we conducted two RUGRAT experiments. In the first experiment, we used RUGRAT to generate applications under test (AUTs)

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

using RUGRAT's default parameter ranges, which model the properties of typical handwritten applications.

In the second experiment we widened the parameter ranges to allow for values that are rare in handwritten applications. While rare, these values are still possible according to the empirical data described in Section 3.2. This second experiment simulates a stress-testing of RAT tools.

In addition to the two RUGRAT experiments, we developed a baseline of how a subset of the RAT tools behave on handwritten applications. This allows us to compare the behaviors handwritten and RUGRAT-generated programs trigger in the selected RAT tools.

6.1. Experimental Setup

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. 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.

On each of the 77 generated AUTs, we applied five Java program analysis tools: four static analysis tools, Checkstyle, FindBugs, JLint, and PMD and one dynamic analysis tool, Randoop[**]. These tools apply different techniques in analyzing programs and produce output in the form of 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.

In the following we briefly summarize the key features of the five Java RAT tools and describe how we configured them for our minimum and maximum configurations.

Checkstyle Checkstyle [39] works on Java source code, is easy to expand, and supports custom bug patterns called ‘checks’. Checkstyle provides a standard ‘check’ that has 64 modules to check the Sun coding conventions which we used for the minimum-effort level experiments. For the maximum effort level experiments, we enabled 128 checking modules (which include the 64 in the standard check). Example modules are FileLength, MethodName, ConstantName, Indentation, and ParameterNumber.

FindBugs FindBugs [40] 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 yields all bugs found during analysis. For the minimum configuration, we set ‘effort’ to minimum and ‘reportLevel’ to ‘high’, to restrict reporting to high priority bugs.

JLint Like FindBugs, JLint [41] applies syntactic bug patterns and dataflow analysis on AUT bytecode, but it is not easy to expand [21]. JLint has patterns for detecting thread synchronization bugs, which we disabled in the minimum configuration. For the maximum configuration, we enable all patterns.

PMD PMD [42] 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.

[**]Checkstyle version 5.4: http://checkstyle.sourceforge.net/
FindBugs version 1.3.9: http://findbugs.sourceforge.net/
JLint version 2.3: http://artho.com/jlint
PMD version 4.2.5: http://pmd.sourceforge.net
Randoop version 1.3.2: http://code.google.com/p/randoop

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

Randoop Randoop [43] applies feedback-directed random test case generation [44] to 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 for the minimum configuration and 2,400 seconds (40 minutes) for the maximum configuration.

6.2. Baseline: Comparing RAT Tools on Handwritten Programs

Locating handwritten programs for a baseline was much easier for RAT tools than for JDK Java source to bytecode compilers (Section 5.2). The RAT tools listed in Section 6.1 support recent versions of the Java language, so a wide variety of suitable handwritten programs is readily available.

For creating a baseline of analyzing handwritten programs, we are guided by the same goals we followed for our compiler baseline in Section 5.1. That is, we keep our experiments reproducible by using open-source programs from SourceForge, aim for a wide range of program sizes, minimize selection bias via random sampling plus other selection strategies, and we keep the experiments feasible by limiting the number of programs in the experiments.

We selected programs via the following four strategies. (1) First, we started with the random SourceForge sample SF100. Since this sample contains many small programs, we picked the largest program of a random sample of 15 SF100 programs (jdbacl). (2) Second, to get additional large programs and since program size is often correlated with the number of classes [33], we also selected two programs from the set of 10 SF100 programs that have the most classes (corina and lilith).(3) Third, to prevent a possible bias towards low-quality programs we also included a random sample of the ten most popular SourceForge programs from SF110 (sweethome, firebird, and netweaver). (4) Finally, to obtain additional large programs, we used Boa [45] to locate the three SourceForge programs with the most AST nodes (clarion2java, JFire, and jEdit). Table IV lists our subject programs and their respective size, ranging from 15k LOC to over 5M LOC.

For the baseline we used a subset of three static RAT tools—Checkstyle, FindBugs, and PMD. We omitted the dynamic Randoop tool because several of the selected AUTs did not include all required classes. We omitted JLint because JLint aborted unexpectedly for several AUTs, likely because we were using an older version of JLint.

Table IV summarize the results of 54 RAT tool runs—3 RAT tools ran in the 2 effort levels of Section 6.1 on 9 handwritten programs. For each run, Table IV lists the tool runtime and the number of tool warnings. One run exceeded our 12 hour time budget and was terminated. From the results we can make the following observations.

- First, as expected, both runtime and number of warnings roughly increased with program size (LOC). The biggest exception are the values for FindBugs on the largest program. This irregularity is likely caused by FindBugs complaining about missing classes for this program.
- Second, each tool produced more warnings in the higher effort level. For Checkstyle this difference was within a factor of 10. For FindBugs and PMD the difference was mostly one order of magnitude.
- Third, Checkstyle produced by far the most warnings among all the RAT tools, followed by PMD and FindBugs. The difference in the number of warnings between Checkstyle and PMD were mostly two orders of magnitude for minimum effort and one order of magnitude for maximum effort. The number of warnings of PMD and FindBugs were mostly within a factor of 10.
- Fourth, higher effort mostly had a higher runtime than lower effort. The difference was mostly within a factor of 2 for FindBugs and PMD. For Checkstyle this difference roughly increased with program size, from one to two orders of magnitude.
- Fifth, in many runs, FindBugs had the highest runtime, followed by PMD and Checkstyle. The runtime difference between FindBugs and PMD was within a factor of 5, between PMD and Checkstyle this difference increased with program size for minimum effort but decreased for maximum effort. For maximum effort, with increasing program size Checkstyle takes relatively longer and takes the most time for the two largest programs. This may point to some elements of the maximum Checkstyle configuration not scaling well with program size.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table IV. Produced warnings (W) and runtime (T) of popular static Java RAT tools running on handwritten programs. Both T and W are given for the two effort levels defined in Section 6.1. Tool execution was canceled (-) after 12 hours.

AUT	LOC	Tool	W_{Ef-Min}	T_{Ef-Min}	W_{Ef-Max}	T_{Ef-Max}
jdbacl	15k	Checkstyle	21,758	5	54,142	11
		FindBugs	17	23	479	30
		PMD	216	9	4,787	14
netweaver	19k	Checkstyle	21,691	5	42,374	9
		FindBugs	3	24	1,042	30
		PMD	7	9	3,067	11
lilith	32k	Checkstyle	34,014	7	124,376	20
		FindBugs	32	35	594	43
		PMD	189	15	-	-
corina	33k	Checkstyle	28,546	5	82,924	14
		FindBugs	83	42	2,376	51
		PMD	110	11	11,160	12
firebird	43k	Checkstyle	55,245	10	120,155	27
		FindBugs	29	32	1,515	44
		PMD	294	23	13,344	27
sweethome3d	51k	Checkstyle	51,243	10	156,098	27
		FindBugs	126	63	2,372	84
		PMD	173	16	10,425	22
jedit	75k	Checkstyle	90,954	11	289,309	42
		FindBugs	157	149	2,879	213
		PMD	463	44	21,099	47
jfire	145k	Checkstyle	177,276	20	847,604	198
		FindBugs	417	105	11,236	139
		PMD	517	64	38,879	59
clarion2java	5,689k	Checkstyle	8,929,007	512	64,058,138	24,747
		FindBugs	772	129	6,096	165
		PMD	20,654	2,434	1,229,716	2,384

- Finally, FindBugs produced the fewest warnings per second (W/s), followed by PMD and Checkstyle. FindBugs had between 0.1 and 10 W/s for minimum effort and between 10 and 100 W/s for maximum effort. PMD had between 1 and 25 W/s for minimum effort and between 250 and 1k W/s for maximum effort. Checkstyle had between 4k and 18k W/s, increasing with LOC, for minimum effort and between 2.5k and 7k W/s for maximum effort.

6.3. RQ2b: Comparing RAT Tools on RUGRAT-Generated Programs

We performed 770 experiments by invoking 5 RAT tools in 2 configurations each on 77 generated AUTs. The two configurations are the minimum and the maximum RAT tool effort configurations described in Section 6.1. For each experiment we captured each tool's execution time and the number of warnings generated by the tool.

Tables V and VI summarize the experimental results for the bulk of the experiments, i.e., the 700 experiments on the default parameter ranges. For each RAT tool, program size category, and both the minimum and the maximum RAT tool effort configurations, these tables give the minimum, maximum, and average RAT tool runtime and number of warnings produced by a RAT tool.

For space limitation we omit the results of the remaining 70 experiments and instead plot highlights of both sets of experiments in Figure 3. Figure 3 shows the average execution time and average number of warnings for each program size category for both the minimum and the maximum RAT tool effort configuration of the experiments on the default parameter

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table V. Produced warnings (W) and runtime (T) of Java RAT tools running on RUGRAT-generated AUTs of various size categories. Both T and W are given for the two effort levels (Ef) defined in Section 6.1.

AUT	Ef	Tool	W_{min}	W_{max}	W_{mean}	T_{min}	T_{max}	T_{mean}
10k	Min	Checkstyle	37,506	144,799	80,008	2	6	3
		FindBugs	111	1,218	499	16	64	25
		JLint	76	651	269	1	4	2
		PMD	192	1,636	776	1	9	4
		Randoop	0	28	8	101	107	103
	Max	Checkstyle	58,354	216,221	120,970	3	9	5
		FindBugs	778	7,889	3,217	17	35	25
		JLint	366	1,101	694	1	5	2
		PMD	3,485	17,901	9,550	4	11	7
		Randoop	0	795	152	2,401	2,404	2,403
50k	Min	Checkstyle	276,984	1,351,638	513,644	10	47	19
		FindBugs	1,080	7,102	2,866	45	146	68
		JLint	211	11,241	2,589	1	5	2
		PMD	3,149	7,631	4,277	5	60	19
		Randoop	0	4	1	102	105	103
	Max	Checkstyle	417,065	1,946,076	749,994	14	71	28
		FindBugs	4,681	51,335	16,097	54	189	86
		JLint	2,468	14,047	4,606	2	11	6
		PMD	23,243	156,035	53,334	9	61	21
		Randoop	0	75	18	2,402	2,410	2,406
100k	Min	Checkstyle	507,352	3,317,995	1,229,880	21	117	43
		FindBugs	1,771	10,026	5,823	77	243	136
		JLint	310	13,737	5,023	1	11	4
		PMD	5,767	14,530	9,749	11	112	43
		Randoop	0	25	4	102	109	105
	Max	Checkstyle	757,294	4,637,270	1,809,312	30	158	63
		FindBugs	9,067	130,142	48,889	98	326	173
		JLint	881	17,486	8,936	3	19	13
		PMD	44,214	403,141	144,826	16	112	46
		Randoop	0	204	33	2,404	2,411	2,406
500k	Min	Checkstyle	1,798,495	13,252,753	5,998,586	62	486	209
		FindBugs	5,172	70,566	30,251	255	1,101	640
		JLint	2,522	123,133	30,967	4	59	19
		PMD	19,105	85,612	46,662	32	393	188
		Randoop	0	0	0	104	134	113
	Max	Checkstyle	2,797,354	19,093,653	8,740,053	97	779	325
		FindBugs	35,965	622,225	223,451	311	1,535	794
		JLint	13,467	145,244	50,548	12	131	69
		PMD	171,052	1,713,544	685,026	41	399	194
		Randoop	0	17	2	2,406	2,438	2,415

ranges. Figure 3(f) shows these measurements for the relaxed parameter range for the FindBugs experiments.

From the results we can make several observations. First, as one would expect, for static analysis tools both the average execution time and the average number of warnings roughly increased with the program size (LOC). This was true for both the minimum and the maximum effort category. The one exception to this observation is the data for JLint in the minimum effort configuration. There the average number of warnings decreases from 1M LOC to 2.5M LOC, while the average number of

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Table VI. Continued from Table V.

AUT	Ef	Tool	W_{min}	W_{max}	W_{mean}	T_{min}	T_{max}	T_{mean}
1M	Min	Checkst.	4,652,104	14,226,538	7,553,695	144	445	248
		FindBugs	9,920	72,593	38,263	597	1,794	945
		JLint	1,880	153,682	60,802	7	171	35
		PMD	25,860	165,267	88,084	42	504	226
		Randoop	0	2	0	105	132	116
	Max	Checkst.	6,663,438	20,924,135	11,369,848	249	879	449
		FindBugs	41,185	560,864	287,923	733	2,009	1,166
		JLint	27,850	191,993	104,567	72	217	121
		PMD	305,925	1,738,556	863,091	90	504	241
		Randoop	0	17	2	2,406	2,543	2,429
2.5M	Min	Checkst.	8,980,257	43,034,144	20,266,502	394	1,304	695
		FindBugs	20,455	299,848	113,640	1,221	5,114	2,540
		JLint	2,596	163,099	42,259	14	167	64
		PMD	25,754	370,177	226,312	93	1,569	593
		Randoop	0	0	0	107	219	140
	Max	Checkst.	13,188,717	59,743,379	29,621,014	716	3,237	1,506
		FindBugs	71,731	1,417,767	628,937	2,006	5,834	3,173
		JLint	52,112	261,345	146,593	147	607	342
		PMD	268,086	4,689,904	2,056,934	213	1,560	631
		Randoop	0	0	0	2,187	2,497	2,413
5M	Min	Checkst.	13,252,180	84,472,865	42,495,951	997	3,108	1,649
		FindBugs	8,686	767,965	319,103	1,795	13,837	7,184
		JLint	4,611	732,496	175,002	27	1,029	364
		PMD	30,281	913,082	391,262	116	3,921	1,577
		Randoop	0	0	0	116	816	286
	Max	Checkst.	20,717,406	129,182,401	64,471,844	1,976	12,401	5,061
		FindBugs	55,583	4,809,618	1,691,975	4,259	18,340	10,087
		JLint	87,128	1,114,570	371,335	238	1,398	642
		PMD	785,208	8,079,371	4,095,392	335	3,881	1,618
		Randoop	0	13	1	1,774	2,984	2,456

warnings for 5M LOC is again higher than for both 1 and 2.5M LOC. This overall trend was largely in line with the experiments on handwritten programs, for which we similarly observed runtime and warnings roughly increasing with LOC.

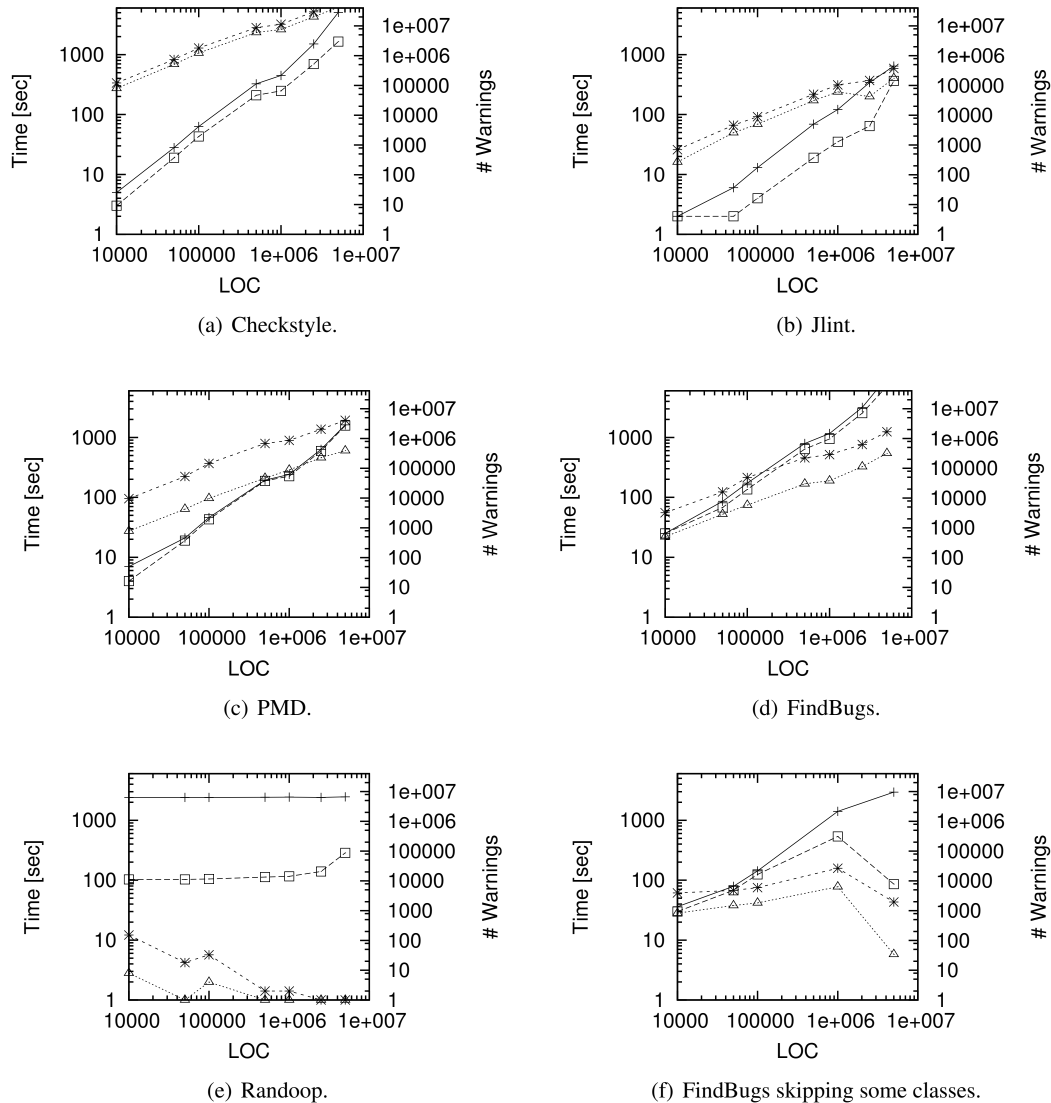
Second, each tool produced more warnings in the higher effort level than in the lower effort level. For Checkstyle this difference was within a factor of 2, for FindBugs within a factor of 20, and for the other tools within a factor of 50.

The high-level trend was in line with the experiments on handwritten programs, for which the tools also produced more warnings in the higher effort configuration. However on handwritten programs the difference between the number of warnings in the minimum and maximum effort levels was larger (by a factor of less than 10). The fact that handwritten programs use more programming language features and libraries than RUGRAT-generated programs may contribute to this larger difference in handwritten programs. The maximum configuration of especially the static tools analyze a larger set of language features than the minimum configuration.

Third, Checkstyle produced by far the most warnings among all the RAT tools (Figure 3(a)). This is true for both effort categories and across all AUT sizes. The difference with the second most producing RAT tool was one or two orders of magnitude, across both effort levels and all AUT sizes. The second highest average number of warnings was produced by the PMD tool, followed (in order) by FindBugs, JLint, and Randoop.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe



(a) Checkstyle. (b) Jlint.

(c) PMD. (d) FindBugs.

(e) Randoop. (f) FindBugs skipping some classes.

Max. config.-Time
Min. config.-Time
Max. config.-Warnings
Min. config.-Warnings

Figure 3. Comparing static and dynamic Java program analysis tools on RUGRAT-generated programs. Each data point is the average of 10 AUTs from RUGRAT's default parameter range (experiment 1), except for 3(f), which shows a single data point each from a wider range of configuration parameters (experiment 2); x-axis = LOC (log scale); left y-axis = RAT tool runtime; right y-axis = RAT tool warnings (log scale); MaxA/MinA = average A in maximum/minimum RAT tool configuration where A is either RAT tool time or number of RAT tool warnings. For static analysis tools both the average execution time and the average number of warnings mostly increased with program size (3(a)–3(d)). FindBugs was an exception when using a wider parameter range (3(f)). The dynamic analysis tool Randoop also behaved differently (3(e)).

The difference in the number of warnings between Checkstyle and PMD was up to two orders of magnitude for minimum effort and within a factor of 50 for the maximum effort category. The

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

number of warnings of PMD and FindBugs were (with one exception) all within a factor of 10. Similarly the number of warnings of FindBugs and JLint were (with few exceptions) within a factor of 10. This observation largely follows our observations on handwritten programs, i.e., both the order of tools by the number of warnings they produce and the relative differences between the number of warnings.

The dynamic program analysis tool Randoop consistently produced the lowest numbers of warnings across both effort categories and all AUT sizes (Figure 3(e)). Moreover, Randoop differed from the other RAT tools in that it produced fewer warnings with increasing AUT LOC sizes. Since this result is counter-intuitive we examine it more closely in Section 6.4.

Fourth, most program/tool combinations had a higher runtime in the higher effort level than in the lower effort level. The average difference was 2 for Checkstyle, 1.3 for FindBugs and PMD, and 5 for JLint. For Checkstyle this difference roughly increased with program size, from about 1.5 to about 4.

Most of this observation is similar to what we have seen for handwritten programs. The main difference is the effort difference of Checkstyle was larger for handwritten programs. However we observed the increase with LOC for both handwritten and generated programs.

Fifth, in both effort levels FindBugs had the highest runtime, followed by (in order) Checkstyle, PMD, and JLint. An exception were smaller programs up to about 30k LOC, for which PMD took more time than Checkstyle. The average runtime difference between FindBugs and Checkstyle was a factor of 4 in minimum effort and a factor of 3 in maximum effort. The runtime difference between Checkstyle and PMD was within a factor of 10. The average distance between PMD and JLint was a factor of 10 for minimum effort and a factor of 3.3 for maximum effort.

This observation is largely similar with our observation on handwritten programs, for which we observed the same runtime ordering of tools. We also observed that Checkstyle did not scale well with larger programs, starting around 30k LOC.

Finally, among the static tools, FindBugs produced the fewest warnings per second (W/s), followed by PMD, JLint, and Checkstyle. FindBugs had an average 40 W/s for minimum effort and 190 W/s for maximum effort. PMD had an average 374 W/s for minimum and 2.9k W/s for maximum effort. JLint had an average 1k W/s for minimum and 611 W/s for maximum effort. Checkstyle had an average 28k W/s for minimum and 24k W/s for maximum effort.

The main difference between these results and the handwritten programs was that for generated programs all tools generate more warnings per second. However for both handwritten and generated programs FindBugs produced the fewest warnings per second, followed by PMD and Checkstyle.

6.4. *RQ3: RUGRAT Found RAT Bugs/Issues*

As a by-product of benchmarking, the RUGRAT-generated programs let us independently rediscover several issues in RAT tools, i.e., in FindBugs and in Randoop. While not dramatic, these results demonstrate the potential usefulness of RUGRAT for testing and debugging.

FindBugs FindBugs may skip classes and miss bugs. For example, in the second experiment, which used wider parameter ranges simulating stress-testing, we encountered the situation depicted in Figure 3(f), 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. Specifically, 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.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Reducing the number of methods for some of the applications caused FindBugs to report warnings where it was previously skipping the analysis.

The FindBugs results are also an example of the influence of the RUGRAT parameters. While FindBugs skipped classes and thus produced few warnings in the second experiment with non-standard RUGRAT parameters (Figure 3(f)), FindBugs did not exhibit this erratic behavior in the first experiment, which uses RUGRAT with its default parameter settings (Figure 3(d)).

Randoop While the other analysis tools generated more warnings for larger programs, Randoop, surprisingly, did the opposite; i.e., the larger the programs the fewer warnings Randoop generated (Figure 3(e)). 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, and therefore yields the same plot as Figure 3(e).

Increasing the runtime to up to 8 hours also led us to independently discover another issue with Randoop. This issue has been reported previously as Issue 14 in Randoop's issue tracking system*. Specifically, 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).

7. RELATED WORK

While RUGRAT leverages the grammar of a programming language to generate programs, there are other program generation techniques that are not based on grammars. For example, Sreenivasan and Kleinman describe a technique for synthesizing programs that produce close-to-realistic workloads for hard drives [46]. The approach composes individual workloads to match certain probability distributions. Unlike this approach, RUGRAT's goal is to create programs that use a wide variety of complex object-oriented language features.

In the remainder of this section we focus on related grammar-based test input generation techniques. Grammar-based test input generation was pioneered by Hanford [47] and Purdom [48] in the 1970s and can be roughly divided into two broad categories, random and systematic.

In the following, we discuss pieces of related work in more detail that are either representative or closely related. Additional related work can be found in a survey article on generating programs for compiler testing [49].

7.1. Probabilistic Grammar-Based Program Generation

Several earlier pieces of work have used probabilistic grammar-based random program generation before [16, 17, 35, 36, 50, 51, 52, 53]. However earlier work mostly focused on testing and debugging. These approaches thus tried to systematically cover corner cases and bugs that are otherwise hard to find. To simplify debugging, the focus was on triggering these corner cases with minimized, focused programs or program fragments. From our perspective, the earlier approaches could be described as generating a collection of maximally diverse micro-benchmarks of rare program shapes. We aim at end-to-end benchmarking and therefore generate large, complex benchmark applications that are close to realistic applications but satisfy specific user-defined constraints.

An early random or probabilistic program generator that is guided by a programming language grammar is presented by Murali and Shyamasundar [16]. The technique targets the PL compiler for a subset of Pascal, the canonical procedural programming language.

*http://code.google.com/p/randoop/issues/detail?id=14

Copyright © 2016 John Wiley & Sons, Ltd.

An early expressive language for grammar-based random program generation is presented by Maurer [17]. That is, the Data-Generation Language or DGL is more expressive than context-free languages, as it supports various actions. The approach generates test suites in the C programming language for functional testing of VLSI circuits.

Burgess describes a system for testing optimizing Fortran compilers [50]. The user specifies the Fortran syntax in an attribute grammar and uses the attributes to express complex correctness rules. The user can also assign probabilities or weights to individual production rules and thereby control how frequently they are utilized in program generation. The generated programs are relatively small, with a size of up to 4k LOC, compared to up to 5M LOC by RUGRAT.

Sirer and Bershad [35] describe probabilistic testing with production grammars. A production grammar is a context-free grammar that can be enhanced with probabilities and actions. The work also introduces the concrete domain specific language (DSL) *lava* for specifying production grammars. The lava language was used to generate Java bytecode programs for testing Java virtual machines. The generated programs ranged up to 60k bytecode instructions. On the other hand, in our experiments we generate large (up to 2.5M LOC) Java source code programs and compare source code to bytecode compilers.

In recent work, Csmith constructs legal C programs randomly using a subset of the C language production rules [52]. Specifically, Csmith consults a probability table, similar to our stochastic selection. Csmith 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. Unlike RUGRAT, Csmith does not support object-oriented language features.

Other than testing C compilers, Cuoq et al. used Csmith for testing static analyzers [53]. They tested Frama-C, a 300k LOC size framework for analysis and transformation of C programs and found 50 bugs.

In the domain of object-oriented programs, a random program generator has been used to test Java just-in-time compilers [36]. 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.

7.2. *Test Program Generation by Combinatorial Grammar Production Rule Coverage*

Combinatorial coverage of grammar production rules is an alternative to stochastic production rule coverage. In the following we briefly review representative and closely related papers.

Purdom has defined a pioneering algorithm for generating small test programs from a given programming language grammar. That is, Purdom's algorithm generates programs that cover each production rule of a given context-free grammar [48].

Celentano et al. describe an early implementation of Purdom's algorithm [54]. This work uses multi-level grammars to support complex correctness rules that cannot be expressed in a context-free grammar alone (such as "define before use"). However it is not clear how this approach scales to complex Java-like languages [55].

Boujarwah et al. implement Purdom's algorithm for a subset of Java [56]. However the implementation has not been applied to generate entire programs and no empirical results are available.

Lämmel and Schulte [57] describe the general-purpose syntax-driven test-data generator *Geno*. Geno works on grammars written in a hybrid of EBNF and algebraic signatures. Geno systematically achieves a user-defined combinatorial coverage of the grammar's production rules. Geno supports computations during test data generation, yielding expressiveness similar to attribute grammars. However Geno does not address the complex correctness rules of Java-like programming languages (such as "define before use", visibility, and inheritance). Geno is also not available for experimentation.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls

DOI: 10.1002/spe

Fischer, Lämmel, and Zaytsev use a grammar-guided test case generator to compare different concrete grammars of the same grammar specification [58]. For example, the work compares various ANTLR grammars of the Java 5 language specification. However, the program generation technique ignores semantic rules (such as "define before use") and removes all such rules from the input ANTLR grammars, yielding context-free grammars.

Harm and Lämmel extend test case generation from systematically covering context-free grammar production rules to systematically covering production rules of attribute grammars [59]. For generating benchmark programs, this technique may enable generating programs that satisfy semantic correctness rules (such as "define before use"). However the scalability of the technique is unclear [57] and the technique has not been applied to Java-like programming languages.

In recent work, Hoffman et al. present YouGen, a practical tool for combinatorial production rule coverage [60]. Similar to earlier work, YouGen takes as input a context-free grammar. YouGen has a wider range of configuration options than previous combinatorial production rule coverage generators.

7.3. Exhaustive Test Program Generation

Exhaustive test program generation aims at enumerating all possible test programs up to a given size. In the following we discuss three representative recent approaches.

Coppit and Lian describe *yagg*, a generator for test data generators that exhaustively enumerate all possible test data up to a given length [61]. The yagg tool supports context-free input grammars that can be enriched with semantic actions.

ASTGen by Daniel et al. systematically generates small Java programs [62]. However, ASTGen 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 value equality (==) is supported in conditions and no deep `if` nesting is possible).

Majumdar and Xu describe a directed test program generation technique that attempts to exhaust the execution paths of a particular compiler or program analysis tool under test [63]. The technique converts a given context-free grammar into a symbolic grammar, exhaustively derives all possible symbolic strings (programs) up to a certain size, and uses these strings in a dynamic symbolic or concolic execution as inputs to the program under test. This directed search yields a small set of representative test programs, as the symbolic reasoning prevents the generation of concrete input programs that cover the same path in the program under test. On the other hand, symbolic reasoning is very expensive, which limits the scalability of the technique. The corresponding tool, *CESE*, has been used to generate small test programs. RUGRAT on the other hand can quickly generate very large random test programs independent of any particular program under test.

7.4. Model-Based Test Program Generation

Beyond grammar production rules, other models of programming language specifications exist. Such models often encode rich semantic information and can be covered systematically by program generators. Given the richness of the information encoded in these models, test case generators are typically slower and focus on generating small programs that are focused on testing specific features.

For example, Zhao et al. capture the rules under which individual compiler optimizations can be applied in temporal logic [64]. The JTT tool then systematically generates focused test programs to test individual compiler optimizations. However it is not clear how this approach scales to entire applications and especially large-scale benchmark applications.

7.5. Random Test Data Generation for Different Domains

Random test data generators have been applied to domains related to object-oriented programming such as generating valid XML files and generating SQL queries. In the following we focus on SQL as an example domain.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

Probabilistic test data generation has been successfully used in testing relational database engines, where complex SQL statements are generated using a random SQL statement generator [15]. RUGRAT extends this idea by applying it to imperative languages such as C++ and Java in that RUGRAT generalizes the approach to generate applications with predefined properties while the SQL statement generator is designed only for a declarative language such as SQL.

A few other approaches are created for generating SQL statements and query sets. One of them is QGEN, a flexible, high-level query generator optimized for decision support system evaluation. QGEN generates arbitrary query sets, which conform to a selected statistical profile without requiring that the queries be statically defined or disclosed prior to testing [65]. QGEN links query syntax with abstracted data distributions, enabling users to parameterize their query workload to match an emerging access pattern or data set modification.

Another recent approach for random SQL generation is a work by Khurshid et al. that generates syntactically and semantically correct SQL queries as inputs for testing relational databases [66]. They leverage the SAT-based Alloy tool-set to reduce the problem of generating valid SQL queries into a SAT problem. With their approach, SQL query constraints are translated into Alloy models, which enable it to generate valid queries that cannot be automatically generated using conventional grammar-based generators. Both this approach and QGEN are complementary to RUGRAT, since the latter can use generated SQL statements to integrate in its generated Java and C++ programs to interact with backend databases. This is our work in progress that gives positive initial results.

Interestingly, generating random images is widely used to evaluate image processing and pattern recognition algorithms [67, 68]. Essentially, finding images with desired properties to evaluate specific algorithms is difficult and laborious; not always these images can be located on the Internet. Yet it is important to obtain images that have specific geometric figures that highlight certain properties of algorithms that use these images. Generating images with desired properties is a standard practice in image processing and pattern recognition [69, 70, 71, 72].

7.6. Other Non-Generated Benchmarks

Other benchmarks of test programs have been developed besides the already discussed widely used DaCapo Java benchmarks [8].

For example, Sewe et. al introduce a Scala benchmark based on the popular DaCapo benchmark for the JVM [73]. Several programming languages (e.g., Scala, Clojure, Groovy, JRuby, and Jython) are typically compiled to Java bytecode and target the JVM. But in JVM research, benchmarks written in these languages are not commonly in use. The authors address this issue by presenting a Scala benchmark and comparing it with the popular DaCapo benchmark on different bytecode metrics. The results show differences between Scala and Java code.

8. CONCLUSIONS

We propose a novel approach for generating random benchmarks for evaluating compilers and program analysis and testing tools using 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 RUGRAT tool for Java and applied it to generate a set of large benchmarks up to 5M LOC with which we evaluated different compilers as well as static and dynamic program analysis tools. The generated benchmarks let us independently rediscover several issues in the analysis tools.

9. 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

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe

supported by the National Science Foundation under Grants No. 0916139, 1017633, 1217928, 1017305, and 1117369, as well as Microsoft Research and Accenture.

REFERENCES

1. Hussain I, Csallner C, Grechanik M, Fu C, Xie Q, Park S, Taneja K, Hossain BMM. Evaluating program analysis and testing tools with the RUGRAT random benchmark application generator. *Proc. 10th International Workshop on Dynamic Analysis (WODA)*, ACM, 2012; 1–6.
2. McDaniel G. *IBM Dictionary of Computing*. McGraw-Hill, 1994.
3. Kanoun K, Spainhower L. *Dependability Benchmarking for Computer Systems*. Wiley-IEEE, 2008.
4. William A Ward J. Role of application benchmarks in the DoD HPC acquisition process. U.S. Army Engineer Research and Development Center, ERDC MSRC Resource 2005.
5. Nash KS. Information technology budgets: Which industry spends the most? (URL:http://tinyurl.com/InformationTechnologyBudgets), Nov 2007.
6. Saavedra RH, Smith AJ. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems (TOCS)* Nov 1996; **14**(4):344–384.
7. Dufour B, Driesen K, Hendren L, Verbrugge C. Dynamic metrics for Java. *Proc. 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, ACM, 2003; 149–168.
8. Blackburn SM, *et al.*. The DaCapo benchmarks: Java benchmarking development and analysis. *Proc. 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM, 2006; 169–190.
9. Blackburn SM, *et al.*. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Communications of the ACM (CACM)* Aug 2008; **51**(8):83–89.
10. Schmeelk S. Towards a unified fault-detection benchmark. *Proc. 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, ACM, 2010; 61–64.
11. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering (ESE)* Oct 2005; **10**(4).
12. Park S, Hussain I, Csallner C, Taneja K, Hossain BM, Grechanik M, Fu C, Xie Q. CarFast: Achieving higher statement coverage faster. *Proc. 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, ACM, 2012; 35:1–35:11.
13. Joshi A, Eeckhout L, Bell RH Jr, John LK. Distilling the essence of proprietary workloads into miniature benchmarks. *ACM Transactions on Architecture and Code Optimization (TACO)* Sep 2008; **5**(2):10:1–10:33.
14. Schwab M, Karrenbach M, Claerbout J. Making scientific computations reproducible. *Computing in Science and Engineering* Nov 2000; **2**(6):61–67.
15. Slutz DR. Massive stochastic testing of SQL. *Proc. 24rd International Conference on Very Large Data Bases (VLDB)*, Morgan Kaufmann, 1998; 618–622.
16. Murali V, Shyamasundar RK. A sentence generator for a compiler for PT, a Pascal subset. *Software—Practice & Experience (SPE)* Sep 1983; **13**(9):857–869.
17. Maurer PM. Generating test data with enhanced context-free grammars. *IEEE Software* Jul 1990; **7**(4):50–55.
18. Aho AV, Sethi R, Ullman JD. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.
19. Cohen S, Kimelfeld B. Querying parse trees of stochastic context-free grammars. *Proc. 13th International Conference on Database Theory (ICDT)*, ACM, 2010; 62–75.
20. Musuvathi M, Engler D. Some lessons from using static analysis and software model checking for bug finding. *Proc. Workshop on Software Model Checking (SoftMC)*, Elsevier, 2003.
21. Rutar N, Almazan CB, Foster JS. A comparison of bug finding tools for Java. *Proc. 15th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2004; 245–256.
22. Zitser M, Lippmann R, Leek T. Testing static analysis tools using exploitable buffer overflows from open source code. *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, ACM, 2004; 97–106.
23. Wagner S, Jürjens J, Koller C, Trischberger P. Comparing bug finding tools with reviews and tests. *Proc. 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems (TestCom)*, Springer, 2005; 40–55.
24. Kannan Y, Sen K. Universal symbolic execution and its application to likely data structure invariant generation. *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ACM, 2008; 283–294.
25. Xiao X, Xie T, Tillmann N, de Halleux J. Precise identification of problems for structural test generation. *Proc. 33rd International Conference on Software Engineering (ICSE)*, ACM, 2011; 611–620.
26. Marri MR, Xie T, Tillmann N, de Halleux J, Schulte W. An empirical study of testing file-system-dependent software with mock objects. *Proc. 4th International Workshop on Automation of Software Test (AST)*, IEEE, 2009; 149–153.
27. Wedyan F, Alrmuny D, Bieman JM. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. *Proc. 2nd International Conference on Software Testing Verification and Validation (ICST)*, IEEE, 2009; 141–150.
28. d'Amorim M, Pacheco C, Xie T, Marinov D, Ernst MD. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2006; 59–68.
29. Ware MS, Fox CJ. Securing Java code: Heuristics and an evaluation of static analysis tools. *Proc. Workshop on Static Analysis (SAW)*, ACM, 2008; 12–21.
30. Wang S, Offutt J. Comparison of unit-level automated test generation tools. *Proc. 2nd International Conference on Software Testing Verification and Validation (ICST) Workshops*, IEEE, 2009; 210–219.

Copyright © 2016 John Wiley & Sons, Ltd.

31. Austin A, Williams L. One technique is not enough: A comparison of vulnerability discovery techniques. *Proc. 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, IEEE, 2011; 97–106.
32. Grechanik M, McMillan C, DeFerrari L, Comi M, Crespi-Reghizzi S, Poshyvanyk D, Fu C, Xie Q, Ghezzi C. An empirical investigation into a large-scale Java open source code repository. *Proc. 4th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, ACM, 2010.
33. Zhang H, Tan HBK. An empirical study of class sizes for large Java systems. *Proc. 14th Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2007; 230–237.
34. Collberg C, Myles G, Stepp M. An empirical study of Java bytecode programs. *Software—Practice & Experience* May 2007; **37**(6):581–641.
35. Sirer EG, Bershad B. Using production grammars in software testing. *Proc. 2nd Conference on Domain-Specific Languages (DSL)*, ACM, 1999; 1–13.
36. Yoshikawa T, Shimura K, Ozawa T. Random program generator for Java JIT compiler test system. *Proc. 3rd International Conference on Quality Software (QSIC)*, IEEE, 2003; 20–24.
37. Dallmeier V, Zimmermann T. Extraction of bug localization benchmarks from history. *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2007; 433–436.
38. Fraser G, Arcuri A. Sound empirical evidence in software testing. *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, IEEE, 2012; 178–188.
39. Evans BJ, Verburg M. *The Well-Grounded Java Developer: Vital techniques of Java 7 and polyglot programming*. Manning Publications, 2012.
40. Hovemeyer D, Pugh W. Finding bugs is easy. *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004*, 2004; 132–136.
41. Artho C, Havelund K. Applying jlint to space exploration software. *Proc. 5th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, Springer, 2004; 297–308.
42. Copeland T. *PMD Applied*. Centennial Books, 2005.
43. Pacheco C, Ernst MD. Randoop: Feedback-directed random testing for Java. *Companion to the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007*, 2007; 815–816.
44. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. *Proc. 29th ACM/IEEE International Conference on Software Engineering (ICSE)*, IEEE, 2007; 75–84.
45. Dyer R, Nguyen HA, Rajan H, Nguyen TN. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. *Proc. 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013; 422–431.
46. Sreenivasan K, Kleinman AJ. On the construction of a representative synthetic workload. *Communications of the ACM (CACM)* Mar 1974; **17**(3):127–133.
47. Hanford KV. Automatic generation of test cases. *IBM Systems Journal* 1970; .
48. Purdom P. A sentence generator for testing parsers. *BIT Numerical Mathematics* 1972; **12**(3):366–375.
49. Boujarwah AS, Saleh K. Compiler test case generation methods: A survey and assessment. *Information & Software Technology (IST)* 1997; **39**(9):617–625.
50. Burgess CJ. The automated generation of test cases for compilers. *Software Testing, Verification & Reliability (STVR)* Jun 1994; **4**(2):81–99.
51. Sirer EG. Testing Java virtual machines. *Proc. International Conference on Software Testing And Review (STAR WEST)*, 1999.
52. Yang X, Chen Y, Eide E, Regehr J. Finding and understanding bugs in C compilers. *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, ACM, 2011; 283–294.
53. Cuoq P, Monate B, Pacalet A, Prevosto V, Regehr J, Yakobowski B, Yang X. Testing static analyzers with randomly generated programs. *Proc. 4th NASA Formal Methods Symposium (NFM)*, Springer, 2012; 120–125.
54. Celentano A, Crespi-Reghizzi S, Vigna PD, Ghezzi C, Granata G, Savoretti F. Compiler testing using a sentence generator. *Software—Practice & Experience (SPE)* Nov 1980; **10**(11):897–918.
55. Hennessy M, Power JF. Analysing the effectiveness of rule-coverage as a reduction criterion for test suites of grammar-based software. *Empirical Software Engineering (ESE)* Aug 2008; **13**(4):343–368.
56. Boujarwah AS, Saleh K, Al-Dallal J. Testing syntax and semantic coverage of java language compilers. *Information & Software Technology (IST)* 1999; **41**(1):15–28.
57. Lämmel R, Schulte W. Controllable combinatorial coverage in grammar-based testing. *Proc. 18th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom)*, Springer, 2006; 19–38.
58. Fischer B, Lämmel R, Zaytsev V. Comparison of context-free grammars based on parsing generated test data. *Proc. 4th International Conference on Software Language Engineering (SLE)*, Springer, 2011; 324–343.
59. Harm J, Lämmel R. Two-dimensional approximation coverage. *Informatica (Slovenia)* Nov 2000; **24**(3).
60. Hoffman D, Ly-Gagnon D, Strooper PA, Wang HY. Grammar-based test generation with YouGen. *Software—Practice & Experience (SPE)* Apr 2011; **41**(4):427–447.
61. Coppit D, Lian J. yagg: An easy-to-use generator for structured test inputs. *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2005; 356–359.
62. Daniel B, Dig D, Garcia K, Marinov D. Automated testing of refactoring engines. *Proc. 15th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, ACM, 2007; 185–194.
63. Majumdar R, Xu RG. Directed test generation using symbolic grammars. *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2007; 134–143.
64. Zhao C, Xue Y, Tao Q, Guo L, Wang Z. Automated test program generation for an industrial optimizing compiler. *Proc. 4th International Workshop on Automation of Software Test (AST)*, IEEE, 2009; 36–43.
65. Poess M, Stephens JM Jr. Generating thousand benchmark queries in seconds. *Proc. 13th International Conference on Very Large Data Bases (VLDB)*, Morgan Kaufmann, 2004; 1045–1053.
66. Abdul Khalek S, Khurshid S. Automated SQL query generation for systematic testing of database engines. *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, 2010; 329–332.

Copyright © 2016 John Wiley & Sons, Ltd.
Prepared using speauth.cls

67. Mantere TJ, Alander JT. Automatic image generation by genetic algorithms for testing halftoning methods 2000; **4197**:297–308.
68. Reghbati HK, Lee AYC. *Computer graphics hardware - image generation and display: tutorial*. IEEE, 1988.
69. Buehler C, Matusik W, McMillan L, Gortler S. Creating and rendering image-based visual hulls. *Technical Report* 1999.
70. Jin Z, Yang D, Yong-jun L, Ying W, Ru-long W. Survey on simplified olfactory bionic model to generate texture images. *Proc. 19th International Conference on Neural Information Processing (ICONIP) - Volume Part II*, Springer, 2012; 316–323.
71. Culik K II, Dube S. Affine automata: A technique to generate complex images. *Proc. on Mathematical Foundations of Computer Science (MFCS)*, Springer, 1990; 224–231.
72. Martin B, Horton RM. A Java program to create simulated microarray images. *Proc. 2004 IEEE Computational Systems Bioinformatics Conference (CSB)*, IEEE Computer Society, 2004; 564–565.
73. Sewe A, Mezini M, Sarimbekov A, Binder W. Da Capo con Scala: Design and analysis of a Scala benchmark suite for the Java virtual machine. *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2011; 657–676.

Copyright © 2016 John Wiley & Sons, Ltd.

DOI: 10.1002/spe