GRT at the SBST 2015 Tool Competition

Lei Ma*††, Cyrille Artho†, Cheng Zhang‡, Hiroyuki Sato*,
Masami Hagiya*, Yoshinori Tanabe§, Mitsuharu Yamamoto††
*University of Tokyo, Japan {malei, schuko}@satolab.itc.u-tokyo.ac.jp hagiya@is.s.u-tokyo.ac.jp
†RISEC, AIST, Japan c.artho@aist.go.jp
‡University of Waterloo, Canada c16zhang@uwaterloo.ca
§National Institute of Informatics, Japan y-tanabe@nii.ac.jp
††Chiba University, Japan mituharu@math.s.chiba-u.ac.jp

***Abstract*—GRT (Guided Random Testing) is an automatic test generation tool for Java code, which leverages static and dynamic program analysis to guide run-time test generation. In this paper, we summarize competition results and experiences of GRT in participating in SBST 2015, where GRT ranked first with a score of 203.73 points over 63 Java classes from 10 packages of 9 open-source software projects.**

***Index Terms*—Software testing, automatic test case generation, program analysis, random testing**

I. Introduction

In 2015, the 8th International Workshop on Search-based Software Testing (SBST) organized the third testing tool competition for unit testing of Java programs. The objective of the competition is to promote testing research and tool development. Each participating tool follows a protocol for given by the competition committee and generates tests for each class in the SBST benchmark [1], [2]. These generated tests are evaluated to compare the efficiency and effectiveness of each testing tool by using a *score-function* based on (1) time to prepare, generate and execute test cases, (2) achieved code coverage (i. e., instruction coverage and branch coverage), and (3) mutation score (to measure fault detection ability). Seven tools have participated and been evaluated in the competition, including Evosuite [3], Evosuite-Mosa [4], GRT [5], jTexPert [6], T3 [7], Randoop [8], and a commercial tool (referred as CT). Randoop and manual test creation are used as the two baselines for all tools, where each represents an extreme end of the level of intelligence.

GRT (Guided Random Testing) is a fully automatic test generation tool for Java programs. Table I summarizes the basic feature of GRT. GRT accepts the class under test (CUT) as Java bytecode and generates test cases, without requiring the source code. Source code of a CUT can be optionally provided as the input of GRT to generate code coverage report automatically. GRT generates test cases in both JUnit 3 and JUnit 4 format. It leverages both static analysis and dynamic analysis to guide each step of test generation, while random selection is adopted when multiple choices exist.

In this paper, we summarize the results of GRT over the benchmarks in the SBST 2015 tool competition. Following a manual analysis of those benchmarks where GRT encounters low coverage, we discuss the reasons for low code coverage and propose some potential future enhancements.

TABLE I
Brief Description and Summarization of GRT

Prerequisites	
Static or dynamic	Dynamic testing at class and system level
Software Type	Java classes
Lifecycle phase	Unit testing for Java programs
Environment	Java virtual machine / Java bytecode
Knowledge required	Unit testing and Java programming
Experience required	Basic software testing and unit (Junit) testing knowledge
Input and Output of the tool	
Input	The bytecode of target classes and their dependencies
Output	JUnit test cases (version 3 or 4)
Operation	
Interaction	Command line interface
User guidance	Through command line and manuals
Source of information	https://sites.google.com/site/grtprojectut
Maturity	Research tool
Technology behind the tool	Program analysis guided random testing
Obtaining the tool and information	
License	To be determined
Cost	Free
Support	Contact the developer
Empirical evidence	
Data is going to be published [5].	

II. The techniques of GRT

A. The Workflow of GRT

GRT works in two phases (see Figure 1). Given a System Under Test (SUT), GRT first performs static analysis on the SUT to extract domain knowledge of the given SUT (bytecode) such as the possible primitive constants, method side effects, and dependencies between methods. Combined with the static information from the first phase, GRT further performs dynamic analysis on method sequence execution feedback. The runtime phase generalizes the idea of feedback-directed random testing [8] (that only uses the execution results of a

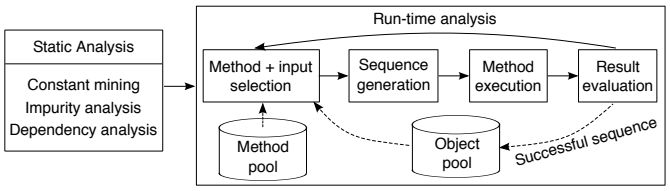

Fig. 1. Simplified overall workflow of GRT.

test sequence to decide whether to use it as inputs for further test generation), by analyzing dynamic information like the exact type information and code coverage information to guide method selection to methods with currently low coverage.

Similar to Randoop [8], GRT adopts a bottom-up style to construct test sequences incrementally. A method with fewer input dependencies is tested first. At each step of the run-time test generation, GRT picks a method from the *method pool* and uses objects from *object pool* as inputs to construct a new test sequence (see Figure 1). Upon successful execution (where no failures and errors occur), new objects returned from the test sequence are stored in the object pool and are used as inputs for other methods, to construct new test sequences. GRT repeats the same procedure to generate more test sequences iteratively until the resource budget (such as the time limit or test sequence size) is exhausted.

B. Major GRT Components

The core of GRT is consisted of six collaborative program analysis components that extract domain knowledge of the SUT to guide each step of test generation to improve the overall testing efficiency and effectiveness:

1) *Constant mining* (static): Constants are extracted from the SUT to seed the object pool (see Figure 1). The mined constants are also used as input for specific methods.
2) *Impurity* (static + run-time): Impurity performs static analysis on all Methods Under Test (MUT) to identify whether they have side effect on program state upon executed. At the run-time phase, it further fuzzes input objects based on method purity analysis before using them as inputs to test an MUT.
3) *Exact type analysis* (run-time): Each generated object from run-time sequence execution uses its exact (dynamic) type to decide whether it is used on a specific MUT, instead of simply using its declared static type.
4) *Detective* (static + run-time): Detective analyzes the method input and output type dependency statically, and constructs missing input data (i. e., non-primitive data) on demand at run-time.
5) *Orienteering* (run-time): The execution cost of each method sequence is measured, and method sequences that require lower cost are favored as inputs for further test sequence construction.
6) *Coverage Guidance* (run-time): All CUTs are instrumented during dynamic class loading, and the coverage of each MUT is analyzed to intelligently guide the method selection for sequence generation at run-time.

Although each of the six GRT components functions independently, they are designed to work collaboratively and can enhance each other when being used together [5]. For example, *exact type analysis* enables *detective* to further diversify the input object types so that a seemingly unobtainable input object can still be found with the exact type information.

C. Use Case of GRT and Adaptations to the Contest

In software development, the time budget for testing is often limited. GRT is mainly designed to automatically generate high-quality test cases for a target software or library under a given time constraint. Although GRT supports to test a target SUT either as a whole or class by class, its current design works better if all classes or a whole package of a given SUT are tested together, because classes of an SUT usually have dependencies. In this contest, GRT follows the SBST protocol [2], [9] and tests a target SUT class by class.

The test execution of a method sequence may cause the undesirable side effect such as deleting files or even the entire working directory. To prevent this, we equip GRT with a security manager that prohibits potentially dangerous operations. Evosuite uses a similar solution [3], [10].

The *Impurity* component of GRT uses the type inference approach to analyze whether a method has side effects (purity analysis). However, the current implementation sometimes exhibits slow performance, and may fail in some circumstances such as when memory is constrained. We therefore disable the Impurity component of GRT during the contest to comply with the SBST protocol that requires full automation.

III. Benchmark results

The SBST 2015 competition benchmark consists of 63 classes from 10 packages of 9 open source projects. Each participant tool is required to be previously installed in its home directory of the contest server running Ubuntu 12.04 on an Intel(R) Core(TM)2 Quad CPU Q9550 @ 2.83 Hz with 8 GB of main memory. The contest is fully automated. When a tool finishes all its evaluation, its author will be notified of its score, calculated by the scoring function [2]:

$$\sum_{class}(1\cdot cov_i + 2\cdot cov_b + 4\cdot cov_m) - (t_{prep} + \sum(t_{gen} + t_{exec}))$$

where cov_i, cov_b, cov_m represent the instruction, branch and mutation coverage; and t_{prep}, t_{gen}, t_{exec} are the time (in hours) for tool preparation, test generation, and execution of generated tests, respectively.

GRT scores 203.73 points over the benchmark, and ranks first among all seven participants. Compared with the two baseline approaches, GRT scores more than twice than Randoop, which scores 93.45 points, and is very close to human tester score of 210.45 points. For the test generation time, GRT uses 4.58 hours in total, which is longer than Randoop with 1.77 hours but far less than the time a human tester took, which was more than 22 hours [2].

The detailed results of GRT are summarized in Table II, where the data are the averaged results over six runs. Among all benchmark classes, GRT achieves 61.0 % instruction coverage in average (per class), and 44.6 % and 45.2 % for branch and mutation coverage, respectively. As discussed in [2], each participating tool adopts different techniques and performs differently on a CUT of the benchmark. When all combined (by taking the best result for each CUT among all tools), an even higher score 266.7 points could be obtained, with the averaged instruction coverage 78.0 %, branch coverage 64.7 %, mutation score 60.3 %. This shows a large potential for improvement.

In line with the GRT evaluation from competition organizers, we also repeat the evaluation of GRT and perform manual analysis over the benchmark by ourselves. In the rest of this section, we summarize some of the issues we have found that cause the low score of GRT on specific CUTs, and discuss some potential solutions.

A. Low Code Coverage

Our manual analysis on the competition benchmark shows that the low code coverage is mainly caused by two reasons. The first reason is that GRT is designed to test an SUT as a whole or in the package level. GRT is not well tested after being adapted to follow the SBST contest protocol to test class by class as we had little time to tune this new feature. This caused GRT to obtain very low coverage (zero or close to zero) on more than 8 out of the 63 CUTs. Another reason is that some branch condition is difficult to satisfy, where a non-trivial method combination with specific order and inputs is required to set its precondition to cover it. Although the Impurity component of GRT enables to create sequences with diverse states, which helps on this issue partially, we disable this option to ensure GRT works correctly with full automation.

1) Zero code coverage: GRT encounters zero coverage on several classes like $\mathrm{net.sf.javaml.core.Fold}$, $\mathrm{org.scribe.model.Request}$. This is because that GRT fails to create several critical input objects of a target CUT (i. e., the inputs of the constructor to create an object) so that GRT simply skips testing all of its MUTs.

As we have discussed in Section II, GRT is designed to test all classes or a whole package of an SUT, where all MUTs are loaded into the fixed method pool (see Figure 1) before testing starts. To test an MUT $m(T_1\,t_1, T_2\,t_2, \ldots, T_n\,t_n)$, where $T_1 \ldots T_n$ are the input type dependencies of m, GRT selects the previously generated objects with desired types from the *object pool* as inputs. If an input object of an MUT is not available at run-time, such an MUT would be skipped for testing until its input objects are available. This issue is diminished if multiple CUTs of an SUT are tested together as they usually have input and output dependencies, where an object returned from testing a CUT can be used as the input to test an MUT of another CUT.

However, if we test an SUT class by class, we lose the ability of reusing objects obtained when testing other CUTs. In the worst case, no input objects can be constructed for all MUTs of a CUT, resulting in zero coverage. Although it would be tempting to include all dependent libraries when testing a CUT, this would waste much effort in testing unrelated classes.Therefore, GRT adopts a lightweight demand-driven approach (the detective component) to construct missing object types online. When a missed input object with type T is required, GRT loads necessary constructors and static methods of T and its methods' dependent types to construct an object typed T. The effectiveness of this approach is shown in our study over 30 benchmarks containing more than 7000 classes, when testing an SUT as a whole [5].

With our adaptation to the competition protocol, the detective component of GRT seems limited in constructing some complicated objects at run-time. We plan to enhance this component by selecting a suitable subset of dependency methods of a CUT to perform demand-driven construction so that more methods could be covered.

2) Code that is difficult to cover for GRT: The first category of difficult code to cover is comprised of the non-abstract methods of abstract classes. As there is no way to initialize an abstract class, to test its non-static methods, subclassing the target abstract class is required. However, it is not trivial to create such a subclass, especially for the abstract methods that have to be overridden. GRT partially solves this issue by collecting subclasses of the target abstract class. There remains the issue of testing non-abstract methods in the abstract parent class: These methods are only tested by the subclass if they are either not overridden, or explicitly called by the overriding method.

The second category of difficult cases consists of methods that require a complex object state, such as methods in $\mathrm{de.tudarmstadt.ukp.wikipedia.api.PageQueryIterable}$. Such methods contain branches that usually require a tester to understand the specific structure and condition of an object to set it to the specific state to satisfy the branch conditions. Such code is still challenging to be covered by automatic tools. We think symbolic execution [11] can be useful for covering some non-trivial branches. Its effectiveness and a possible integration with GRT still requires further investigation.

Another kind of difficult code requires the MUTs to be tested in some specific order, like $\mathrm{org.asynchttpclient.SimpleAsyncHttpClient}$. This often occurs when an SUT implements a non-trivial API protocol, and is a challenge for automatic tools like GRT, as the API protocol specification is not available from the bytecode of an SUT. Automated inference of API protocols is future work.

B. Low Mutation Score

The current version of GRT only captures simple program regression behaviors on primitive values to generate test assertions. However, many mutants require non-trivial assertions to be killed, e. g., a collection that is expected to have a specific size, or an array that has to contain a specific element. High-quality assertion generation is important future work.

Non-deterministic behavior of certain tests is also responsible for a low mutation score. GRT currently removes all assertions in a test method if one of them is non-deterministic.

Although this does not influence the code coverage, it decreases the mutation score. This can be improved by only removing the assertions that fail upon replay.

In addition, we find that all participating tools (even the human testers) have a relative low score on the void method call (VMC) mutants compared to other kinds of mutants [2]. This might be caused by the program state change not being captured when executing a mutated (removed) method with void return type. This can be possibly improved by analyzing the class attributes that can be influenced by VMC mutations. In summary, we think the assertion enhancement is another major future work of GRT.

IV. Conclusion

We have summarized the techniques of GRT and discussed our experience in participating in the SBST 2015 testing competition. At current stage, GRT's score is close to the score obtained by manually created test cases. However, there is still a long way to go to for GRT to create high-quality test cases with both high code coverage and bug detection ability. GRT scores less than a half of the ideal full score $441 = (7*63-0)$ points, showing that there is still room for improvement. The participation in this competition helped us to uncover issues and new research directions for GRT.

Acknowledgment

We would like to thank the SBST testing competition organizers for holding such an attracting event and providing a stage for all participants to demonstrate their tools to facilitate automatic testing research and tool development.

References

[1] SBST Testing Contest, "http://sbstcontest.dsic.upv.es/," 2015.

[2] U. Rueda, T. Vos, and I. Prasetya, "Unit testing tool competitions - round three," in *The 8th Int. Workshop on Search-Based Softare Testing (SBST)*, 2015.

[3] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proc. of the 19th ACM SIGSOFT Symp. and the 13th Euro. Conf. on Found. of Softw. Eng. (ESEC/FSE)*, Szeged, Hungary, 2011, pp. 416–419.

[4] A. Panichella, M. Kifetew, Fitsum, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *Int. Conf. on Softw. Testing, Verification and Validation (ICST)*, 2015.

[5] L. Ma, C. Artho, H. Sato, J. Gmeiner, R. Rudolf, and C. Zhang, "Enhancing random testing via run-time guidance," in preparation.

[6] A. Sakti, G. Pesant, and Y.-G. Gueheneuc, "Instance generator and problem representation to improve object oriented code coverage," *to appear in IEEE Transactions on Software Engineering*, 2015.

[7] I. Prasetya, "T3, a combinator-based random testing tool for java: Benchmarking," in *Future Internet Testing*, 2014, pp. 101–110.

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. of the 29th Int. Conf. on Softw. Eng. (ICSE)*, Minneapolis, USA, 2007, pp. 75–84.

[9] S. Bauersfeld, T. Vos, and K. Lakhotia, "Unit testing tool competitions lessons learned," in *Future Internet Testing*. Springer International Publishing, 2013, pp. 75–94.

[10] G. Fraser and A. Arcuri, "Evosuite: On the challenges of test case generation in the real world," in *Proc. of the 2013 IEEE Sixth Int. Conf. on Softw. Testing, Verification and Validation (ICST)*, Washington, DC, USA, 2013, pp. 362–369.

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

TABLE II
Detailed Results of GRT over SBST Contest benchmarks.

SBST Benchmark Package & Class	JUFiles	Time (min.) Gen.	Time (min.) Exec.	Instr. Cov (%)	Bran. Cov (%)	Mutat. Cov (%)
package: com.google.gdata.data						
AttributeHelper	2	0.4	0.01	2.54	0	0
DateTime	39	4.5	0.52	94.23	84.76	60.68
Kind	27	3.82	0.28	67.49	65.91	62.68
Link	16	4.27	0.27	74.13	70.05	55
OtherContent	22	4.39	0.21	42.48	38.64	30
OutOfLineContent	23	5.1	0.23	75.61	79.17	45.83
Source	13	4.81	0.15	35.01	23.48	25.65
package: net.sf.javaml.core						
AbstractInstance	24	3.78	0.38	92.31	71.43	79.17
Complex	18	4.17	0.16	100	0	100
DefaultDataset	19	3.85	0.17	48.81	37.5	41.94
DenseInstance	24	5.52	0.32	100	100	88.71
Fold	0	2.27	0	0	0	0
SparseInstance	22	6.3	0.36	98.37	87.5	73.53
package: net.sf.javaml.tools.data						
ARFFHandler	21	4.91	0.21	38.98	25	22.22
package: twitter4j						
ExceptionDiagnosis	100	2.75	0.92	0	0	0
GeoQuery	25	4.31	0.22	93.35	80.56	77.25
OEmbedRequest	22	6.83	0.23	97.33	84.88	67.34
Paging	57	5.5	0.52	99.48	97.22	79.11
TwitterBaseImpl	2	3.07	0.01	0	0	0
TwitterException	26	9.16	0.24	75.35	64.29	67.07
TwitterImpl	2	3.31	0.01	0.19	0	0
package: com.puppycrawl.tools.checkstyle.api						
AbstractLoader	53	2.07	0.71	43.5	8.33	1.67
AnnotationUtility	121	4.17	0.95	19.05	10	0
AutomaticBean	32	6.94	0.5	36.59	0	4.65
FileContents	26	6.09	0.3	92.53	80.77	51.33
FileText	18	9	0.23	90.18	87.82	64.18
ScopeUtils	18	10.57	0.14	26.58	11	29.51
Utils	121	10.21	1.17	93.93	96.15	82.54
package: com.google.common.base						
CharMatcher	17	3.02	0.2	84.55	82.02	61.5
Joiner	12	3.41	0.16	84.55	86.96	97.87
Objects	28	4.65	0.27	100	99.07	100
Predicates	23	5.25	0.22	54.65	44.44	46.61
SmallCharMatcher	16	4.3	0.34	98.16	96.79	74.31
Splitter	30	4.47	0.28	95.39	89.1	66.85
Suppliers	24	4.73	0.23	46.34	20.83	27.27
package: org.hibernate.search						
SearchException	28	3.03	0.21	100	0	0
Version	2	2.82	0.01	100	0	0
backend.BackendFactory	13	3.19	0.13	30.08	6.25	33.33
backend.FlushLuceneWork	19	3.5	0.18	89.74	100	50
backend.OptimizeLuceneWork	17	3.69	0.17	89.74	100	50
util.logging.impl.LoggerFactory	2	3.42	0.01	56.25	0	50
util.logging.impl.LoggerHelper	19	4.08	0.24	100	0	33.33
package: de.tudarmstadt.ukp.wikipedia.api						
CategoryDescendantsIterator	0	2.27	0	0	0	0
CycleHandler	118	2.79	1.28	18.36	0	0
Page	81	5.5	1.11	1.49	6	0
PageIterator	5	5.17	0.05	11.26	0	0
PageQueryIterable	0	2.32	0	0	0	0
Title	44	8.37	0.44	78.4	77.08	100
WikipediaInfo	2	5.64	0.01	8.58	7.33	1
package: org.asynchttpclient						
AsyncHttpClient	15	2.95	0.26	71.62	58.33	66.67
AsyncHttpClientConfig	17	3.12	0.22	95.17	54.44	84.55
FluentCaseInsensitiveStringsMap	11	3.26	0.1	95.23	90.41	90.57
FluentStringsMap	10	3.38	0.11	96.12	92.05	93.8
Realm	21	5.13	0.2	92.95	59.46	84.37
RequestBuilderBase	13	3.75	0.2	71.62	54.64	42.64
SimpleAsyncHttpClient	7	5.84	0.26	74.4	40.98	58.28
package: org.scribe.model						
OAuthConfig	33	3.12	0.28	68.35	50	88.89
OAuthRequest	0	0.31	0	0	0	0
ParameterList	14	3.29	0.18	91.55	94.44	95.65
Request	0	0.31	0	0	0	0
Response	2	1.5	0.01	1.77	0	0
Token	41	4.52	0.37	99.16	96.88	90.91
Verifier	106	6.66	0.83	100	0	50
Average	**27**	**4.36**	**0.29**	**61.01**	**44.63**	**45.21**