Practical Extensions of a Randomized Testing Tool

Hojun Jaygarl, Carl K. Chang
Department of Computer Science
Iowa State University
Ames, USA
{*jaygarl,chang*}*@cs.iastate.edu*

Sunghun Kim
Department of Computer Science
The Hong Kong University of Science and Technology
Hong Kong, China
hunkim@cse.ust.hk

***Abstract*—Many efficient random testing algorithms for object-oriented software have been proposed due to their simplicity and reasonable code coverage; however, even the state-of-the-art random test algorithms yield very low code coverage (around 22%) on large-scale software. We propose four testing techniques to improve test coverage. The proposed techniques are pluggable to any existing random testing techniques for object-oriented software. We incorporated our techniques to a state-of-the-art random testing tool and tested large-scale software, including Java Collections, Apache Ant, and ASM. Our experimental study shows that the proposed techniques increase at most 21% of branch coverage – a significant improvement.**

I. Introduction

Writing software tests is difficult and labor intensive work. In particular, developers under release deadline pressure often do not have enough resources to write and update test cases. To make matters worse, writing tests for objected-oriented (OO) code requires complex data inputs and valid call sequences of each instance to properly test software.

Automated testing has the potential to address testing problems, or at least to assist in addressing the problems. Recently, many random testing algorithms for OO software have been proposed and spotlighted due to their simplicity and reasonable test coverage [1], [2], [3]; however, most random testing algorithms are evaluated with small or toy software. To be widely useful, testing algorithms should scale and yield high test coverage for large-scale software. We ran two state-of-the-art testing algorithms [1], [3] on large-scale software including Java Collections, ASM and Apache Ant. Unfortunately, the test coverage[1] of exiting random testing algorithms is only 22.4%, which is too low (see section 4).

Constructing unit tests for OO codes involves a number of choices: what methods to call, what arguments to give, or what order for invocations. The search space of possible method sequences is very large, and different method/input selection strategies can have a big impact on the effectiveness of test generation algorithms. Therefore, finding better method/input selection strategy is desirable.

We propose and evaluate four techniques that together can improve test coverage. Our four techniques use different selection points in the construction of test inputs. *Distance-based selection* collects the furthest input data from the used values. *Type-based selection* gives equal chances between different data types to be chosen. *Open-access selection* allows the testing of all protected and private classes and methods. Finally, the *array generation* technique generates arrays by gathering values that have a requested element type. These four techniques are pluggable in existing random testing algorithms.

We implemented our techniques on top of the Random Tester for OO Programs (RANDOOP) [3]. Then, we generated test cases and measured test coverage of them.

In this paper, we make the following contributions:

- We present three selection techniques, distance-based, type-based, and open-access random testing approaches. Our techniques give a direction to select input test data. These approaches show better test coverage compared to the existing state-of-the-art approach.
- We provide one technique to generate array inputs and array distance calculation.
- Our proposed four techniques together can greatly improve the test coverage achieved by a test generator.
- To validate scalability, we evaluate our approach on large-scale software including Java Collections, ASM, and Apache Ant.

Our results show that the test coverage for these large-scale software systems are increased from 43.2% to 64.2%, 22.4% to 35.7%, and 41.9% to 52.9% respectively. This experiment shows that the proposed techniques significantly increase the test coverage and make random testing techniques more useful in practice.

The remainder of the paper is organized as follows: Section 2 describes the background and related works of this research. Section 3 proposes our method and input selection techniques. Our empirical evaluation is presented in Section 4. We conclude in Section 5.

[1] "Test coverage" is defined as the percentage of test requirements in this paper. If a test suite achieves 100% of coverage rate, it completes all test requirements. Note that we use "branch-coverage" as test coverage in this paper. Branch coverage measures which possible branches in selection structures are followed.

II. Background and Related Work

A. Random Test Generation

In order to automatically generate test input data, *random testing* (RT) has been proposed [4], which selects test cases randomly from the input domain of a program. The concept and implementation are relatively simple compared to other existing techniques, and offers several benefits as an effective black box testing technique, such as reliability. Random testing has been widely used since it is simple to be implemented and does not have human intervention.

Nevertheless, it has been argued that random testing lacks a systematic approach to learn from previous executing results. *Adaptive random testing* (ART) [5] offers an opportunity to choose evenly distributed inputs over the range of possible input values. Thus, it became one of the most effective approaches in the automatic test generation area. One study shows ART to be at least 50% more efficient than RT [5].

There are many ART approaches [6], [7], [5]. Among them, the simplest adaptive random selection techniques is Distance-based Adaptive Random Testing (D-ART), also called Fixed-Sized-Candidate-Set ART (FSCS-ART) [5]. D-ART projects input values into an input space and measure distances between values in the input space. D-ART uses two input data sets; a candidate set has several input data and an executed set records used inputs. D-ART checks a sum of distances from all elements of the executed set to each candidate input in the candidate set, and then chooses the furthest candidate as an input value.

Our approaches incorporate with *feedback-directed random test generation* [3] and enhance *adaptive random testing for OO software* [1]. We describe these two techniques in the following sections.

B. Feedback-directed Random Test Generation

Feedback-directed Random Test generation technique (F-RT) [8] incrementally generates method sequences by selecting a method to apply and then selecting inputs to the method of OO software. F-RT uses feedbacks from the execution of the previously generated sequences as an input for a currently selected method.

RANDOOP [9] is a tool that implements F-RT algorithm by taking a set of classes to test and generates method call sequences. RANDOOP creates JUnit [10] test cases based on the sequences.

The experimental results show that F-RT outperforms systematic and pure random test generation, according to the criteria of test coverage and error detection [9], [3]. F-RT suggests an innovative idea; however, an adaptive distribution of input selection (a directed random selection) may be desirable for F-RT, since F-RT uses pure random (a undirected random) selection strategy to select input data.

C. ARTOO

Ciupa *et al*. suggest adaptive random testing for OO software (ARTOO) [1] that applies the D-ART approach to choose input objects from a pool. Since D-ART for OO programs needs to calculate the distance between objects, they developed a new notion, *object distance* [11], and applied it in OO software testing. ARTOO found the first fault in a much smaller number of test cases, using only 20% of the number of test cases required by an undirected random testing approach [1].

ARTOO characterizes objects by *elementary distance* (the distance between the direct values of objects), *type distance* (the distance between types of objects, completely independent of the values of the object themselves), and *field distance* (the distance between individual fields of the objects). The object distance is calculated as summation of these three components with weights and normalization[2].

A major problem with D-ART and ARTOO is that a dimension of input domain increases calculation time exponentially. For example, integer type values are easier and faster for checking the distance; however, calculating an object distance takes a much longer time. In Ciupa's paper [1], ARTOO took a much longer time (160% from undirected RT) because of the calculation of object distance, although ARTOO generates a smaller number of test cases.

III. Extension of Method and Input Selection

We propose *Practical Extensions of Random Testing* (PE-RT) that combines F-RT with our several random selection techniques, such as distance, type, and open-access selection.

A. Distance-based input selection

ARTOO's object distance calculation has high time complexity. We suggest a *simplified object distance* that calculates object distance with lesser time complexity. We divided input data types into three categories - primitive types (including boxed types and a string type), array types and object types. This separation removes unnecessary calculation of the ARTOO's object distance algorithm.

1) Primitive type distance: We determine a distance of a primitive type as the following:

- Number type : $|p - q|$
- Character type: convert to a number type (e.g. based on ASCII code table) and calculate as a number type
- Boolean type: 0 if identical, otherwise a positive constance value which is greater than 0
- String type: the Levenshtein distance [12]

[2]We skip details of object distance algorithm in this paper. Please refer Ciupa *et al*'s paper [11], [1].

2) Array type distance: For an array type distance, we consider element type. If an array element type is a primitive, distance is calculated as the following:

- *identical*: an array size is identical, a type of an element is compatible[3], and all the element values are the same.
- *same_size*: an array size is identical, a type of an element is compatible, but some of element values are not identical. Calculate $\sum |a(i) - b(i)|$, where $0 < i <=$ the size of the array, and $a(i)$ and $b(i)$ are i_{th} elements of two arrays respectively.
- *different_size* : only a type of element is compatible, calculate $\sum |x - y|$, where x and y are a size of each array.
- *different*: they have different types of elements.

We normalize and give a weight for each category to make the following equation:

identical > *same_size* > *different_size* > *different* = 0

3) Object type distance: We categorize an object distance between two objects into four levels:

- *identical*: identical types and identical field values
- *similar*: identical types, but different field values
- *compatible*: compatible types
- *different*: different types

The degree of distance is the following:

identical > *similar* > *compatible* > *different* = 0

We test the equality of two objects by the `equal()` method and == operator to decide whether they are *identical*. If their types are identical, our algorithm compares its member variables to decide whether the distance of two objects is *identical* or *close*. If they do not have identical object types, our algorithm explores their common ancestors to decide whether their distance is *compatible* or *different*.

B. Type-based input selection

There are two possible type-matching techniques to choose input data for generating test cases: compatible type-matching and identical type-matching. Literally, compatible type-matching assumes compatible types as a matched type, while identical type-matching defines only the same types as a matched type. If objects have the common ancestor class or interface, their types are compatible. For instance, consider an interface I and two classes, C and D, that implement I. In this instance, C and D are compatible types.

An input selection technique can be wrongly directed if we use only one matching strategy among two type-matching techniques. If we keep only an identical type-matching technique, compatible type values never get a chance to be selected. On the other hand, if we use the compatible type-matching technique, identical types have lower probability to be selected than compatible types, because the number of compatible type values is greater, in general, than that of identical type values. Input data need to be selected with the equal probability among compatible and identical type values.

To solve this problem, we set the equal probability between two type-matching techniques instead of using only one matching technique. In order words, input data are selected with probability 0.5 for compatible type values and 0.5 for identical type values.

C. Open-access method selection

The *open-access method selection* technique can test non-public methods. In Java, private and protected methods are hard to be tested, since these non-public methods are not accessible from the outside of classes or packages. We found that 17% for Ant, 28% for Java containers and 34% for ASM methods are non-public methods. The *open-access method selection* increases test coverage by changing non-public methods to public.

To include non-public methods for testing, we modified method accessibility by using the ASM Bytecode Framework [13]. The *open-access method selection* approach finds all non-public methods and classes, and changes it to public in a bytecode level.

D. Array input generation

Our array generation technique generates arrays as input data from non-array feedbacks. Like an object type, array's high dimension of the input domain makes it difficult to achieve high test coverage. Nevertheless, a simple array generation technique increases code coverage in spite of the complexity of array input.

Our array generation approach is as follows: if a method under test takes an array input, the algorithm finds all available values that match an array's element type from the previously generated inputs. When using a type-matching strategy, we also use our type-based input selection approach. After listing all the available values, our algorithm chooses a size of the input array by a randomly generated number (the size is up to the number of possible values). Based on the chosen size, the algorithm randomly selects the values among the available values and assigns them to a random position of the array.

IV. Evaluation and Results

This section reports our empirical evaluation on the performance and usefulness of the PE-RT approach, which combines all our suggested approaches. The goal of this experiment is to compare test coverage of a hybrid PE-RT approach to the current state-of-the-art approach, F-RT, in terms of time cost[4] and the number of test cases. Also, we

[3]We define compatible types that share a common ancestor.

Table I
SUBJECT OPEN SOURCE PROJECTS

Projects Name	Testing Classes	Methods	KLOC	Description
ISSTA Containers	5	71	2	Container classes presented by Visser *et al*. [14]
Java Collections 1.6	45	634	22	Java's collection library
ASM 3.1	111	1353	40	A Java bytecode manipulation and analysis framework
Apache Ant 1.7.1	769	3001	209	A Java-based build tool

show the comparison results among all our individual approaches, such as distance/type-based input selection, open-access method selection, and array generation approaches including two different object distance calculation methods.

A. Experimental setup

The subjects are selected from well-known open source projects, ISSTA Containers, Java Collections, Apache Ant, and ASM. Table I presents information about the subjects.

ISSTA Containers has container classes (`TreeMap`, `BinTree`, `FibHeap`, and `BinomialHeap`) used in Visser *et al*'s paper [14]. The Java collection library is a well-known JDK package (`java.util`) for Java developers. It contains the collection framework, container classes, and miscellaneous utility classes including list, set, and map classes. We chose the remaining two projects - ASM and Ant - among open source projects.

We evaluated each approach by measuring branch coverage criteria. We used a machine with Windows Vista, Intel Pentium 2.2Ghz Dual Core, 3GB RAM. All test coverage included in this section is branch coverage.

Table II
F-RT VS. PE-RT FOR COVERAGE

Projects	F-RT	PE-RT	Improvement
ISSTA Containers	86.6%	91.6%	5.0%
Java Collections	43.2%	64.2%	21.0%
ASM	22.4%	35.7%	13.3%
Apache Ant	41.9%	52.9%	11.0%

B. Overall Observation

Table II shows test coverages of the original F-RT and our PE-RT approaches. ISSTA containers is a relatively small project that has only 71 methods, and state-of-the-arts RANDOOP achieved over 86.6% of test coverage. Based on their result, it is believed that the borderline of test coverage that automatic tools can achieve has been nearly reached [9]. Contrary to expectations, our PE-RT approach achieved 91.6% of test coverage, which is 5% more than F-RT. PE-RT improves test coverage by 11-14% more test coverage than F-RT for ASM and Ant.

The Java collections show the best result that is raised by 21.0% of test coverage by PE-RT compared to F-RT's test coverage, in spite of the large number of methods and code size.

[4]In this paper, time cost means generation time of test cases.

Figure 1-4 shows test coverage for Java Containers and ASM with respect to the number of test cases and time. Most of our individual approaches increase test coverage in terms of both the number of test cases and generation time. Moreover, our combined approach, PE-RT, shows significant improvement.

Table III represents the number of test cases and generation time between F-RT and PE-RT to reach certain test coverage (goal coverage[5]). To achieve 42% of test coverage for Java Containers, F-RT generated 10700 test cases in 485 seconds, meanwhile PE-RT generated 1170 test cases in 6 seconds. PE-RT achieves the goal coverage more quickly with fewer test cases. Throughout all of the projects, almost half of the test time and test case size are reduced to achieve certain test coverage.

C. Distance-based input selection

The distance-based approach increases test coverage through all projects with respect to the number of test cases; however, it is very slow to generate test cases, as Ciupa *et al.* mentioned in their paper [1]. In Figure 2, the distance-based approach shows better test coverage at the beginning, but the original F-RT surpasses it quickly. This is because ARTOO's distance based approach is too meticulous to select an input, and object calculation is computationally expensive.

Table IV
SIMPLIFIED OBJECT DISTANCE

Projects	Goal Coverage	# Test Cases ARTOO:Simple	Time (Sec) ARTOO:Simple
ISSTA	82%	12865:9088	338:237
Java Util	42%	4340:3518	7997:1977
ASM	20%	13673:13802	4867:479

Our simplified distance-based input selection reduces the computation time, and increases test coverage in terms of both generation time and the number of test cases. Table IV lists the result of direct comparison between Ciupa's approach and ours. In Table IV, we measure the number of test cases and time to reach certain test coverage by both distance-based approaches. Our simplified distance based approach shows consistent increments of approximately 2-5% through all projects.

[5]The goal coverage has been decided by substrating 1-4% from the highest coverage of F-RT for each project.

Table III
F-RT VS. PE-RT BASED ON CERTAIN COVERAGE

Projects	Goal Coverage	# Test Cases (F-RT:PE-RT)	Ratio	Time (Sec) (F-RT:PE-RT)	Ratio
ISSTA Containers	82%	14209:8347	1.7:1	340:191	1.8:1
Java Collections	42%	10700:1170	9.1:1	485:6	80:1
ASM	20%	20469:2867	7.1:1	471:37	12.7:1
Apache Ant	40%	19466:8145	2.4:1	712:243	2.9:1

D. *Type-based input selection*

The type-based input selection constantly increases 2-4% average test coverage for all of the projects. According to Figures 1-4, our type-based input selection approach is always better than F-RT. The result shows that there are many redundant compatible type input data, and few compatible type inputs are still necessary. By giving the equal selection chance to both the identical and compatible type inputs, we can get improvement of test coverage.

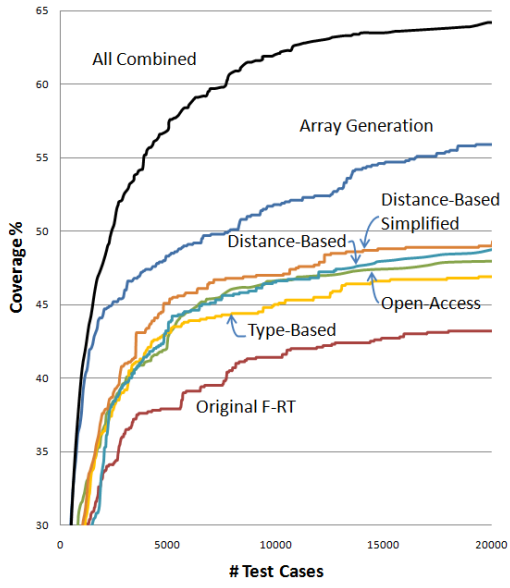

Figure 1. Java Containers for # Test Cases

E. *Array Generation*

Besides ISSTA Containers, the array generation approach shows approximately 10% of test coverage increments for Java Containers and 5-7% for ASM. Figures 1 and 2 illustrate that the array generation approach has the best improvement except the combined approach. The result shows that most of the projects need array inputs.

For ISSTA Containers, the array generation approach shows 1-2% lower coverage than the original F-RT test coverage because methods of five ISSTA containers do not take an array input.

F. *Open-access method selection*

The open-access method selection gives 7-9% average test coverage improvement, since it provides a way to test

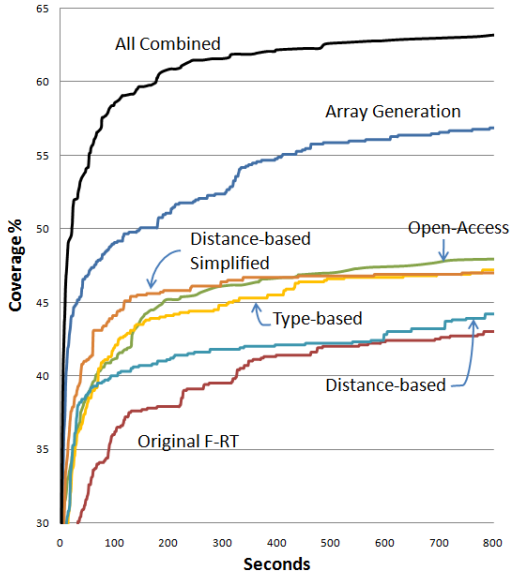

Figure 2. Java Containers for Time

non-public methods without changing and analyzing source codes. The open-access approach shows the best coverage besides the all combined approach for ASM project with respect to both time and test case size.

G. *Limitation*

Although our PE-RT approach shows better coverage, it is still low for the ASM (36%) and Ant (53%) projects. We found uncovered areas with the following obstacles.

- File I/O: For example, ASM needs a bytecode and Ant requires a Build.xml file as an input of the project. Moreover, XML file commonly has many keywords for their properties, and some codes are triggered by only those keywords.
- Concurrency: some codes are implemented as thread and need communication with other threads.
- User Interface: some codes are for user interfaces and need interaction.
- Environment: some codes are affected by environmental setting, such as OS, network connections, certain devices, time and a certain situation, not only by inputs. These codes might occur non-deterministic behaviors.

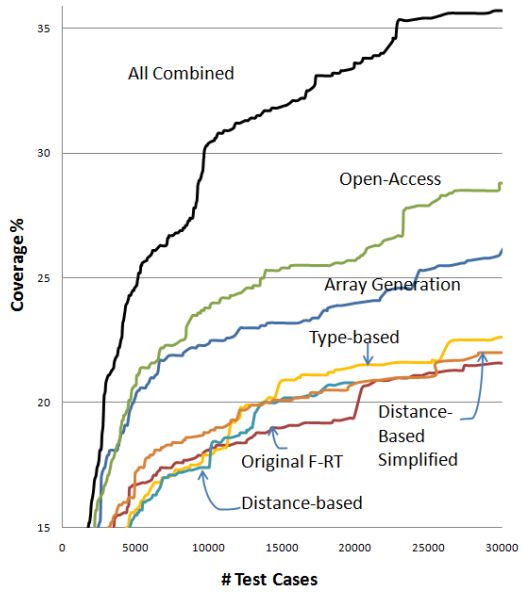

Figure 3. Coverage of ASM based on # Test Cases

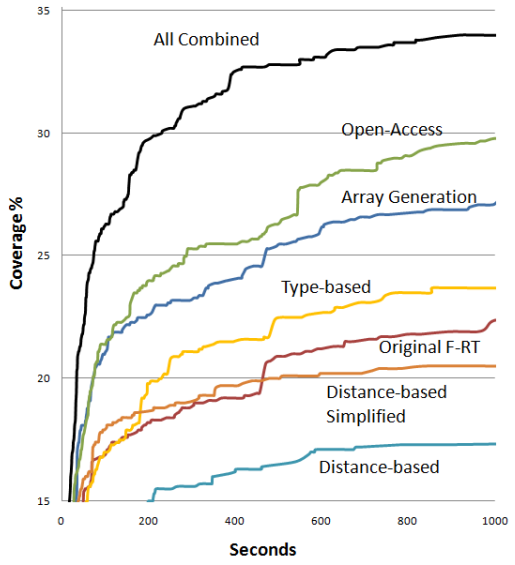

Figure 4. Coverage ASM based on Generation Time

V. Conclusions

In this paper, we proposed four extended techniques: distance-based, type-based input selection, open-access selection and array generation. We have implemented them on top of RANDOOP and tested with large-scale software systems. Our experiments show that all four techniques increase test coverage. Our approaches yield the best test coverage when we combine all four approaches together; they increase test coverage up to 21%.

Our proposed approaches are applicable for other existing testing techniques. Anyone who wants to increase their random testing coverage may benefit from integrating our suggested techniques into their tools.

Acknowledgment

We thank Carlos Pacheco for providing RANDOOP's source code and test coverage checking code, and supporting.

References

[1] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: adaptive random testing for object-oriented software," in *ICSE*, 2008, pp. 71–80.

[2] N. Tillmann and J. de Halleux, "Pex–white box test generation for .net," in *2nd International Conference on Tests and Proofs*, April 2008, pp. 134–153. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79124-9_10

[3] C. Pacheco, S. K. Lahiri, and T. Ball, "Finding errors in .net with feedback-directed random testing," in *ISSTA 2008: International Symposium on Software Testing and Analysis*, Seattle, Washington, July 20–24, 2008.

[4] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[5] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *ASIAN*, 2004, pp. 320–329.

[6] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *ECSQ '02: Proceedings of the 7th International Conference on Software Quality*. London, UK: Springer-Verlag, 2002, pp. 321–330.

[7] T. Chen, F. Kuo, R. Merkel, and S. Ng, "Mirror adaptive random testing," *Quality Software, 2003. Proceedings. Third International Conference on*, pp. 4–11, Nov. 2003.

[8] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA: IEEE Computer Society, 2007.

[9] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *OOPSLA 2007 Companion, Montreal, Canada*. ACM, Oct. 2007.

[10] "Junit," `http://junit.org`.

[11] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Object distance and its application to adaptive random testing of object-oriented programs," in *RT '06: Proceedings of the 1st international workshop on Random testing*. New York, NY, USA: ACM, 2006, pp. 55–63.

[12] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," Tech. Rep. 8, 1966.

[13] "Asm: Java bytecode manipulation and analysis framework," `http://asm.objectweb.org/`.

[14] W. Visser, C. S. Păsăreanu, and R. Pelánek, "Test input generation for java containers using state matching," in *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2006, pp. 37–48.