2010

Capture-based Automated Test Input Generation

Hojun Jaygarl
Iowa State University

Capture-based automated test input generation

By

Hojun Jaygarl

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee
Carl K. Chang, Major Professor
Jonny Wong
Morris Chang
Ying Cai
Simanta Mitra

Iowa State University
Ames, Iowa
2010

DEDICATION

To my brother, mother,

and

father.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

ACKNOWLEDGMENTS

I would like to express my thanks to people who helped me with various aspects of conducting research. My topmost gratitude goes to my advisor, Dr. Carl K. Chang, who has been guiding me throughout my long journey in the software engineering field. He has been a wonderful guider, great mentor, and excellent source of knowledge. He introduced the academic world to me and I received comprehensive training, which will benefit my life. His insights and words of encouragement have often motivated me and refreshed my mind for completing my graduate education. Professor Chang is a great role model, since he has abundant enthusiasm and energy to his life and to his work. I am also thankful for his care to his students. He always listens to our problems and helps resolve our confusions.

I will always remember my good times spent in the Software Engineering Lab. I also extend my thanks to my friends, Jinchun Xia, Hua Ming, and Katsunory Oyama. In addition, friendship from Ing-Xiang Chen is one of the best things I have ever received in my life.

I would also like to thank my committee members for their efforts and contributions to this work: Drs. Jonny Wong, Morris Chang, Ying Cai and Dr. Simanta Mitra. I have been privileged to have great mentors outside Iowa State University as well. I would like to thank Dr. Sunghun Kim, Dr. Heung Seok Chae and Dr. Tao Xie and his students for their valuable comments on improving my research.

My last, but deepest, thanks goes to my family—my parents, and my older brother—for their endless love and support. Without their support, I would not be able to finish this work. I hope my father is happy with me.

ABSTRACT

Testing object-oriented software is critical because object-oriented languages have been commonly used in developing modern software systems. Many efficient test input generation techniques for object-oriented software have been proposed; however, state-of-the-art algorithms yield very low code coverage (e.g., less than 50%) on large-scale software. Therefore, one important and yet challenging problem is to generate desirable input objects for receivers and arguments that can achieve high code coverage (such as branch coverage) or help reveal bugs. Desirable objects help tests exercise the new parts of the code. However, generating desirable objects has been a significant challenge for automated test input generation tools, partly because the search space for such desirable objects is huge.

To address this significant challenge, we propose a novel approach called Capture-based Automated Test Input Generation for Objected-Oriented Unit Testing (CAPTIG). The contributions of this proposed research are the following.

First, CAPTIG enhances method-sequence generation techniques. Our approach introduces a set of new algorithms for guided input and method selection that increase code coverage. In addition, CAPTIG efficently reduces the amount of generated input.

Second, CAPTIG captures objects dynamically from program execution during either system testing or real use. These captured inputs can support existing automated test input generation tools, such as a random testing tool called Randoop, to achieve higher code coverage.

Third, CAPTIG statically analyzes the observed branches that had not been covered and attempts to exercise them by mutating existing inputs, based on the weakest precondition analysis. This technique also contributes to achieve higher code coverage.

Fourth, CAPTIG can be used to reproduce software crashes, based on crash stack trace. This feature can considerably reduce cost for analyzing and removing causes of the crashes.

In addition, each CAPTIG technique can be independently applied to leverage existing testing techniques. We anticipate our approach can achieve higher code coverage with a reduced duration of time with smaller amount of test input. To evaluate this new approach, we performed experiments with well-known large-scale open-source software and discovered our approach can help achieve higher code coverage with fewer amounts of time and test inputs.

CHAPTER 1. INTRODUCTION

Conducting effective testing of object-oriented software is critical for providing high confidence on its correctness in view of the widespread use of object-oriented languages in the development of modern software systems.

Among different types of testing, unit testing has been widely adopted in practice as an important means to improving software quality. Unit testing is a testing method for checking functional behavior of a unit in a program. Each unit is a set of method invocation sequences (in short, method-sequences) with test inputs. Test inputs, such as a method's arguments and receiver object, affect behavior of a method invocation, test coverage, and efficiency of testing.

Manually writing test inputs of unit testing are labor intensive. It is difficult to explore all possible behaviors of a unit under testing. In particular, developers under the pressure of meeting release deadline often do not have sufficient resources to write test inputs for revealing bugs. To make matters worse, writing tests for object-oriented code requires complex data inputs.

To reduce this manual effort in object-oriented unit testing, many researchers have devoted efforts to introduce automated test input generation techniques [31][35][74][77][82][86] that automatically generate test inputs for a unit (e.g., a class or a method). These test inputs are in the form of method invocations, where (1) primitive values (such as integer values) for primitive-type method arguments are automatically generated, and (2) objects for receivers and non-primitive-type method arguments are automatically generated via composing method-sequences. However, most input generation algorithms are evaluated only with small or

toy software. To be more practical, these algorithms should scale up and yield higher code coverage for large-scale software (e.g., a system with over 1,000 methods).

There are many automated test input generation techniques. Yet, one important and challenging problem is to generate desirable objects for receivers or arguments of methods to achieve high code coverage (such as branch coverage). A desirable object means an object input with high potential to satisfy test criteria, such as "structural code coverage" to observe a wide range of behaviors of a unit. Desirable objects help exercise the not-yet-covered code. A code coverage criterion is generally used to show how well the unit has been exercised. In this thesis, we use branch coverage, which measures how many branches having been exercised. Achieving high code coverage helps test inputs to access parts of code rarely or never accessed under normal conditions, and assures important conditions have been tested.

If an automated test input generation can archive high code coverage, test inputs can be used to attack practical debugging problems. In this thesis, the generated test inputs are applied to a crash reproduction problem. Reproducing software crash is important, particularly for debugging the crash. However, crash reproduction is often labor intensive and challenging. This study addresses crash reproduction problem by the automated test input generation technique developed here.

The main purposes of this thesis are (1) to develop new techniques that generate desirable test inputs, including objects and method-sequences, to achieve higher code coverage with a smaller set of inputs, and (2) to develop a new technique that reproduces a given crash only, based on crash stack trace by using the proposed test input generation approach. The following subsections address challenges of this research area and briefly describe solutions.

1.1. Challenges in Test Input Generation

It is difficult and expensive to write test inputs manually, that cover all branches or statements in a software system, since a developer or tester cannot create test inputs to account for all possible situations within limited time. Making it worse, code becomes more complex after several releases of enhancements.

In addition, writing test inputs for object-oriented software becomes more difficult and complicated than that for software systems developed in procedural languages, such as C language. For example, a tester only needs to prepare input arguments for tested functions plus global variables in procedural languages. On the contrary, a tester needs to prepare both receiver object and arguments for object-oriented languages. Objects significantly increase the complexity of testing. In particular, input objects have a huge search space, since an object could have many member fields of primitive types and other object types (e.g., container classes such as `Stack` and `List`.) Because of information hiding, object fields often cannot be directly constructed, but are indirectly constructed through a sequence of specific method invocations. Because of these difficulties, automated test input generation approaches are still unsatisfactory and often impractical for object-oriented software.

Automated testing has the potential to address these test input generation problems or at least to assist them. In general, in order to generate desirable object inputs to visit all feasible execution paths, existing automated test input generation tools are employed two techniques: (1) direct input construction and (2) method-sequence generation.

The techniques of direct input construction, such as Korat [31], directly assign values to object fields under construction. However, these techniques require the value domain (i.e.,

the set of selected values) for each primitive-type field be manually specified. The techniques of method-sequence generation [41][74][66][66][85] propagate method-sequences to produce an object under construction. However, because of the huge search space of possible method-sequences, method-sequence generation techniques are often ineffective to find method-sequences that can produce desirable objects to cover target branches [57][59].

Failing to generate desirable objects is the main cause of low code coverage achieved by the aforementioned techniques. For example, Figure 1 shows an uncovered area of an open-source system, called Apache XML Security [3], after running a state-of-the-art random testing tool, Randoop [74], for 80 minutes. The target method `Algorithm()` takes a `Document` argument. The first statement `super(doc)` in the method checks the validity of the `Document` argument. Because Randoop failed to generate a desirable `Document` object, the `super(doc)` causes an exception, Thus, all lines after line 87 cannot be covered by Randoop.

```
80  /*
81   *
82   * @param doc
83   * @param algorithmURI is the URI of the algorithm as String
84   */
85  public Algorithm(Document doc, String algorithmURI) {
86
87    super(doc);
88
89    this.setAlgorithmURI(algorithmURI);
90    ...
```

Figure 1. A motivation example. `doc` is insufficient and causes exception in line 87. Line 89 cannot be reached unless a desirable `doc` object is put. The code fragment is from `org.apache.xml.security.algorithms.Algorithm`

Now, the main challenges are discussed (that we empirically observed) for a state-of-the-art random testing tool, called Randoop, to achieve high structural code coverage, such as branch coverage. The analysis of these challenges motivates the CAPTIG approach. Randoop was chosen in this preliminary empirical study for two main factors: (1) Randoop can be applied to any real-world code bases in a totally automated fashion without any manual effort, and (2) Randoop has been shown to outperform other systematic and pure random test input generation tools in terms of achieved branch coverage [74].

Randoop ran on three large-scale open-source systems, namely, Apache Commons Collections [1], Apache XML Security [3], and JSAP [17], until their code coverage was saturated, which means either coverage levels off without much further increase, or a tool cannot continue to run because of memory shortage. As a result, Randoop achieves 45.2% of branch coverage for Apache Commons Collection, 29.6% for Apache XML Security, and 54.1% for JSAP. Then, 10 source code files from each subject system were randomly selected, and the causes of uncovered branches were manually investigated. Table 1 summarizes major causes for uncovered branches by Randoop.

Table 1. Categories of main causes for not-covered branches from 10 source files from three open-source systems

cause No.	cause of uncovered branches	# of branches (%)
1	insufficient object	135 (46.3%)
2	string comparison	61 (20.9%)
3	container object access	39 (13.4%)
4	array comparison	25 (8.6%)
5	exception branches	18 (6.1%)
6	environmental setting	9 (3.1%)

The top main cause (No. 1) of an uncovered branch is that Randoop was unable to generate a desirable object required to exercise certain branches. This factor accounts for nearly 50% of the uncovered branches in this study. As shown in Figure 1, the generated object '`doc`' does not satisfy the desirable condition for covering the branches in the methods under test.

String comparison (cause No. 2) contributes as the second main cause. Most string comparisons were performed by simple constraints, such as equality (e.g., `equals()` and `equalsIgnoreCase()`), size (e.g., `length()`), and substring (e.g., `contains()`, `substring()`, and `charAt()`). However, Randoop failed to find a desirable string randomly to satisfy such constraints, since the input space of the string is huge.

The third main cause is container object access. Creating a certain size of a container with necessary elements is not easy. For example, creating `ArrayList` that contains 20 `BankAccount` instances is not easy.

Similar to the third main cause, the fourth main cause, array comparisons, includes accessing array elements and checking their size. This randomly creates an array with desirable elements or size is difficult.

Exception branches are related to a `try..catch` statement, or a branch that checks exception type. These branches are relatively difficult to cover by automated tools due to the pecularity of exception handling code that handles run-time errors. To determine a desirable condition, it should be investigated how exceptions are defined and where these exceptions have been thrown (e.g., an exception propagation path).

Finally, the last two main causes (cause No. 5 and 6) are branches related to an environmental setting (e.g., environment variables and file system structure) and non-deterministic execution (e.g., multi-thread and user interaction), which are difficult to be addressed by automated testing approaches. Indeed, the former cause can be alleviated with mock object-based approaches [66][84] and the latter cause with concurrent testing approaches [77].

The causes No. 1, 3, and 4 are related to a lack of desirable objects, and few existing approaches can effectively address these main causes, especially when testing real-world code bases. Thummalapenta *et al.* [82] pointed out that creating desirable objects is a main challenge in automated testing techniques, such as dynamic symbolic testing [63][77].

Motivated by the above observations, this research addresses the problem of generating method-sequence and desirable inputs for object-oriented unit testing with emphasis on achieving high coverage, usability, and scalability.

1.2. Challenges in Crash Reproduction

In software development and maintenance processes, reliable crash reproduction is an essential first step to debugging. Unfortunately, crash reproduction in a manual way is tedious and time-consuming for developers [27]. To assist crash reproduction, many approaches have been proposed [14][18][19][22][24][46][69][80], including capture-replay and post-failure-process approaches.

Capture-replay approaches record software executions, and reproduce the executions by playing the recorded executions. However, the capturing process usually incurs substantial performance overhead. Alternatively, post failure-process approaches, such as Windows Er-

ror Reporting System [46] and Breakpad [14] collect memory dumps or stack traces after crashes occur. Since these approaches do not incur any additional performance overhead, they are widely used in practice. The information collected from post-failure-process approaches is used to prioritize debugging efforts and provide debugging hints for developers [46].

When post-failure-process approaches are adopted, developers must reproduce reported crashes manually. A typical debugging scenario using crash information includes: (1) inspecting stack traces, (2) executing a debugger on methods listed in the stack traces, (3) observing input and output values of methods, and (4) tweaking the input values to understand and reproduce the crashes. This manual process requires time consuming and non-trivial human efforts.

Reproduced crashes are helpful to identify and finally repair the failure. Figure 2 shows the importance of crash reproduction. For the reported bug #437861, the developer could not reproduce the crash based on reporter's comment without a test case. For bug #702, the developer also had the same problem, but could reproduce the failure after obtaining the test case for the crash.

```
AspectJ Bug #437861
2003/09/27 Reporter said "don't have a simple test case yet , but
            hopeful this stack trace will be revealing." and
            added stack trace information.

<4 months later>

2004/01/14 Developer said "Priority will go up if a
            self-contained test case is submitted."
2004/03/18 Developer marked it as INVALID.


Apache Common Collection Bug #702
2005/06/08 Reporter reported bug with a method call and crash
            stack trace information.
2005/07/09 Developer said "Do you have a test case for this?"

<5 months later>

2005-12-11 Reporter added test cases.
2006-01-21 Developer said "As always, a good test case makes all
            the difference." and marked as FIXED.
```

Figure 2. A brief summary of comments in AspectJ bug #43786 and Apache Common Collection Bug #70. This example shows difficulty and importance of reproducing failure. Note, our approach reproduces these failures.

Despite of its importance, reproducing failure is often labor intensive and time consuming because software engineers frequently have difficulties in understanding comments in a bug report. The survey result by Bettenburg *et al.* [27] mentioned that incomplete information of bug reports is the most commonly encountered problem with bug report in developers' experience. Accordingly, it is highly important for software engineers to automatically reproduce a crash based on given information.

To illustrate the present state of the practice for crash reproduction, we study the bug reports from open-source systems and test input generation techniques. The first case studied how quickly crashes are fixed. It was assumed that crash fixing time is greatly affected by

crash reproduction time, since developers consider crash reproduction as the most useful activity to obtain information for fixing a bug [27]. Crashes were reproduced by this study's approach (see Section 7.3). The original failure fixing time for these failures was measured. Originally, 69.4% of the failures took over a week, and 61.1% took over a month. In addition, Anvik *et al.* [21] also discussed that approximately 50% of fixed bugs took over a week and 30% of fixed bugs took over a month to resolve. Therefore, it is believed that bug fixing time is not trivial and can be reduced by adopting an automated failure reproduction technique.

Table 2 presents the number of fixed bug reports (i.e., a bug report closed with a "fixed" status) of five open-source projects over six years. Approximately 68% of the failures took over a week and 46% of the failures took over a month for resolution. For this empirical study, bug reports from five open-source projects were used, including AspectJ Development Tools [6], Eclipse C/C++ Development Tools [10], Eclipse Equinox [11], Eclipse Graphical Editing Framework [12] and Eclipse Java Development tools [13].

Table 2. Amount of time to fix reported bugs

systems	total	over		period (yy/mm)	
		a week	a month	from	to
AspectJ Development Tools	612	68.3%	50.2%	02/07	08/03
Eclipse C/C++ Development Tools	5333	70.4%	53.8%	02/01	08/03
Equinox	2103	66.4%	53.0%	01/10	08/03
Eclipse Graphical Editing Framework	714	59.9%	39.9%	02/07	08/03
Eclipse Java Development tools	18167	68.4%	64.8%	01/10	08/03
total	26929	68.4%	46.3%	-	-

There is a difficulty of reproducing failure by using the random test input generation technique with an empirical study. A set of crash stack traces from bug reports was extracted from AspectJ Development Tools, Apache Commons Collections [1], and Apache Commons Math [2]. Based on the crash stack trace of the bug reports, a list of relevant classes and methods to being tested were extracted. Next, the extended version of Randoop on AspectJ Development Tools, Apache Commons Collections, and Apache Commons Math was run with the list of classes and methods for 10 minutes. Then, the number of reproducible failures was counted. These results show the random test input generation approach reproduced only 13 bugs among 101 crash stack traces, which is 12.9% of reproducibility. In brief, the random approach is not sufficient to reproduce a failure. Therefore, an automated failure reproduction technique, which is better than random approach, is desirable.

1.3. Research Objectives

The goal of CAPTIG is to automate the creation of test inputs that achieve high structural code coverage and apply generated test inputs to crash reproduction. The specific aims of the proposed research are the following.

Capture inputs dynamically from program execution: CAPTIG captures objects from normal program execution (e.g., results from system testing or real usage by end users). It is not easy to automatically generate the desirable inputs required to cover many branches. However, by running and using the software during this execution, many actual inputs will be created and can be collected via CAPTIG. The captured objects can be fed to the-state-of-art testing techniques as initial test inputs. Since these inputs reflect real usage, capturing the-

se inputs and exploiting them in automated testing provide great potential for being desirable in achieving new branch coverage.

Generate method-sequences by new techniques: CAPTIG generates test inputs by constructing a method-sequence generation technique. An enhanced method-sequence generation algorithm that improves the existing techniques is proposed. Moreover, CAPTIG evolves the captured objects. Sometimes the captured inputs may not be exactly desirable inputs to cover a target branch (e.g., a not-yet-covered branch), but provide a good basis to direct the execution to reach the target branch. As an initial input to a method-sequence generation of CAPTIG, captured inputs can evolve gradually toward reaching the target branch.

Statically analyze observed not-yet-covered branches and mutate captured inputs: CAPTIG diversifies test inputs by mutating existing captured objects to exercise not-yet-covered branches. Although existing techniques are assisted by captured inputs to cover more branches, there are still not-yet-covered branches. For some not-yet-covered branches, mutate the captured inputs on purpose to cover uncovered branches and employ a static analysis technique to mutate these captured inputs. First, CAPTIG identifies constraints of not-yet-covered branches. That is, CAPTIG collects preconditions, based on not-yet-covered branches information. Then, CAPTIG generates constraints and places them into a constraint solver to obtain counter-examples. CAPTIG parses the counter-examples to obtain a value that an input object must have to be desirable. To verify whether the diversified input satisfies the precondition of the target branch, CAPTIG invokes a method under test with the diversified input.

Reduce the number of generated method-sequences: Current automated test input generation tools are not concerned with the number of test inputs. CAPTIG prunes redundant method-sequences without running test inputs by checking the inclusion-relation of the generated test inputs. This can reduce the number of test inputs without abating code coverage. Reducing test inputs helps to not only analyze, manage, and understand generated test inputs, but also curtails the testing time for a regression test. Many test-suite reduction techniques have been proposed [32][50][70]. However, these techniques require execution of test cases to identify redundant test inputs, which can be very inefficient and impractical for huge test inputs.

Reproduce Software Crash: CAPTIG reproduces crashes, based on crash stack trace information. CAPTIG generates test cases that direct crashed methods to reproduce the original crashes. Reproduced crashes are helpful to identify and repair the failure.

The proposed techniques yield methods to capture and use objects from program execution to generate desirable objects and reproduce a crash. To the best of our knowledge, this approach is the first work to obtain a method-sequence generation basis from captured information of execution and to automatically reproduce a crash. The proposed approach also yields an efficient method to generate method-sequence by achieving higher code coverage. The techniques developed in this research can be useful in practical unit testing and crash reproduction by achieving high coverage with meaningful test inputs.

In addition, each of these CAPTIG techniques can be independently used and adopted to leverage existing testing techniques. To facilitate the evaluation of our novel approach, well-known, large-scale open-source software will be used.

1.4. Organization

The remainder of this thesis is organized as follows.

Preliminaries: Chapter 2 discusses the basic concept of this research, including automated software testing, test input generation, test case reduction and crash reproduction. While Chapter 2 describes concepts that directly related to this thesis, Chapter 9 gives wider review of literature, which describes the state-of-the-art works in this area.

Approaches: Chapters 3 through 6 describe this study's approach. Chapter 3 describes the method-sequence generation with explanation of existing method-sequence generation techniques and improvements. This chapter also describes the test input reduction technique used in this study. Chapter 4 describes the object-capture-based input generation technique, which captures objects dynamically from program executions and generates test inputs, based on captured objects. While both Chapter 3 and 5 describes automated test input generation, Chapter 5 describes the crash reproduction technique that shows usability of generated method-sequences and objects. Chapter 6 describes implementation of the techniques.

Evaluations and discussions: Chapters 7 and 8 assess the effectiveness of this study's techniques in comparison with exisiting automated test input generation techniques, using experiments, case studies (Chapter 7), and a discussion of this study's approaches, strengths, and limitations (Chapter 8). Chapter 10 offers concluding remarks.

CHAPTER 2. BACKGROUND OF RESEARCH

Software testing [20] is classified into unit testing, module testing, integration testing, system testing, acceptance testing, and field testing in terms of hierarchical levels of testing. A unit is the smallest component of testable software application and a unit testing focuses on testing each individual unit. Unit testing examines individual units before integrating them into a larger system. In contrast, system testing checks the end-to-end behavior of the entire application. This study's CAPTIG approach is in the context of a particular testing activity—writing unit tests.

Software testing addresses a variety of fields to satisfy many aspects of requirements, such as functional correctness, compatibility, performance, and usability. To satisfy these test requirements, inputs of unit testing should be desirable to reveal unsatisfied properties. Code coverage is used as a metric to show the quality of unit test cases. Test cases, which have good quality to satisfy test requirements, will achieve high code coverage. The goal of this study's approach is to achieve higher code coverage with a small set of test inputs.

The CAPTIG approach applies the generated test inputs to one of the practical testing problems—crash reproduction. Based on limited information on crashes, this study's crash reproduction approach reproduces the crash by using automated test input generation techniques developed through this study.

In this chapter, research background is discussed, including unit testing of object-oriented software, desirable object input construction, test case reduction, and crash reproduction.

2.1. Unit Testing of Object-oriented Software

The main difference between unit tests of procedural languages (such as C) and object-oriented languages (such as C++ and Java) is that in procedural languages, a tester only needs to prepare input arguments for tested functions plus global variables. However, in object-oriented languages, a tester needs to prepare both receiver and arguments. For example, suppose a class `BankAccount` has a private integer-typed member field `accountBalance` and a method `deposit(int amount)`. If testers want to test `deposit(int amount)`, they should prepare the arguments (such as -1, 0, 1, 100, and 1000000) as well as receiver objects (such as a `BankAccount` instance with `accountBalance`=-100, -1, 0 and 1000). Such a receiver object can be created by constructing and invoking a method-sequence. Figure 3 shows a sample legal method-sequence for the `BankAccount` class. Note, without the statement in line 05, the entire sequence becomes illegal because the method-sequence cannot be compiled successfully without a `BankAccount` instance.

```
01 Int a=1000;  //Variable declaration
02 Int b=-1;    //Variable declaration
03
04 //Construct with a initial balance
05 BankAccount account = new BankAccount(a);
06
07 //Test with -1
08 account.deposit(b);
09
10 //Test with 1000
11 account.deposit(a);
```

Figure 3. A method-sequence example

2.2. Desirable Object Construction

To address desired object construction challenges, existing automated test input generation tools adopt two main techniques: (1) direct input construction and (2) method-sequence generation.

2.2.1. Direct Input Construction

The techniques of direct input construction, such as Korat [31] and TestEra [59], directly assign values to object fields of the object under construction. If an object field of the object is also a non-primitive type (i.e., requiring an object), the techniques further construct an object for this object field and assign values to object fields of this second-level object. This procedure is conducted repeatedly until the object-field values of the object have been assigned to a small pre-defined bound of levels.

To avoid generating invalid objects (e.g., tree objects with cycles among tree nodes), these techniques require the specification of class invariants for checking and filtering invalid objects. However, in practice, class invariants are rarely documented. In addition, these techniques require the value domain (i.e., the set of selected values) for each primitive-type object field be manually specified.

All of these factors cause these techniques to be ineffective for testing real-world classes. For example, Korat [31] was evaluated on primarily data structures and its application requires much manual effort for preparing class invariants and value domains. In addition, the coverage of various branches requires a high bound of levels beyond the small bound that can be handled by the direct input construction approach.

Korat requires users to provide a `repOk()` Boolean method, which checks the validity of a given object against the required class invariant for the class of the object. TestEra [59] requires users to provide class invariants specified in the Alloy specification language [56]. Both Korat and TestEra use class invariants to efficiently prune both invalid objects (those violating class invariants) and redundant objects (those with equivalent states), when generating a bounded-exhaustive set of objects (whose size is within a relatively small bound). Their techniques also require users to provide a finite domain of values for primitive-type fields in the generated objects.

2.2.2. Method-sequence Generation

To generate inputs and method-sequences shown in Figure 3, random testing [49] has been applied. Random testing has many advantages, including practical applicability, free of bias, and ease of implementation. One of the important goals for random testing is to achieve high code coverage with a minimal set of test cases.

Various techniques on method-sequence generation have been proposed for generating objects used to test input generation. Random-testing techniques (such as JCrasher [41] and Randoop [74]) generate random method-sequences, sometimes with pruning based on feedback from previously generated sequences [74]. Evolutionary testing techniques (e.g., eToc [85] and Evacon [66]) use genetic algorithms to evolve initial method-sequences to ones more likely to cover target branches. Bounded-exhaustive testing techniques (such as JPF [86], Rostra [88], and Symstra [89]) for method-sequence generation produce exhaustive method-sequences up to a certain length, sometimes with pruning of equivalent concrete objects [86][88] or subsumed symbolic objects [89]. However, the coverage of various branches

leads to very long method-sequences, whose lengths are beyond the small bound that can be handled by these techniques.

In contrast, the CAPTIG approach is able to capture objects produced with long method-sequences (from real applications) and further evolve these objects by using more method-sequences or by directly mutating these objects. Recent sequence-mining techniques, such as MSeqGen [82], statically collect method-sequences (that can produce objects of a specific type) from various applications, and then apply dynamic symbolic execution [82] or random testing [74] on these collected sequences. In contrast, this study dynamically collects method-sequence from execution.

CAPTIG can be viewed as a novel integration of both direct input construction and method-sequence generation techniques for generating desirable inputs. The input capturing and diversifying of CAPTIG can be viewed as a type of direct input construction, but it does not suffer from the limitations of earlier techniques, such as Korat [31]. For example, CAPTIG does not require class invariants, and constructs valid method-sequences and objects by directly capturing them from normal program executions. CAPTIG can construct inputs of very large size beyond a size within a small bound. In addition, CAPTIG mutates captured inputs to attempt moving towards target branches while Korat does not exploit guidance from the target branch.

CAPTIG can also be viewed as a technique of method-sequence generation, since the test input generation techniques explore and generate new method-sequences derivied from the captured and mutated test inputs. The diversified inputs often reflect desirable inputs or inputs close to desirable inputs. Therefore, some other method invocations in method-

sequences generated by the test input generation techniques can evolve the captured and mutated inputs into the desirable ones.

2.3. Test-Suite Reduction

Test-suite reduction techniques reduce the size of a test-suite, while retaining test requirements of the test-suite. In other words, these techniques select a smaller representative subset of test inputs that can still achieve the same code coverage as the entire test input set. Test-suite reduction can diminish the cost of executing, managing, and analyzing test inputs. Given requirements on this set of test inputs, the test-suite reduction problem can be stated as follows [50].

Given: Test-suite TS and a set of test input requirements $r_1, r_2, \ldots, r_n$ that must be satisfied to provide the desired code coverage of the program, and subsets of TS, $T_1, T_2, \ldots, T_n$, one associated with each of the r_is, such that any one of the test inputs t_j belonging to T_i can be used to test r_i.

Problem: Find a minimal set of test inputs from TS that satisfies all of r_is.

Finding the minimal set is an NP-complete problem [46]. Thus, test-suite reduction techniques use approximation to find the minimal representative set. The heuristics used by Harrold *et al.* execute generated test inputs one-by-one [50]. If the test requirement (such as code coverage) is satisfied, unused test inputs are redundant and executed test inputs are representative. There are many algorithms for test-suite reduction [32][50][70]. These approaches need to run test inputs and check test requirements.

Test-suite reduction techniques have been reconsidered for object-oriented systems. Previously, test-suite reduction techniques do not need to consider the same input values. Certainly, the same input value leads to the same behavior of a unit under test. In an object-oriented system, the behavior of a method depends upon arguments and receiver objects. Therefore, it is important to consider both arguments and receiver objects.

JPF [86] and Rostra [88] detect redundant unit tests by using bounded exhaustive generation with state matching of argument and receiver objects. They share the use of state matching on objects and prune sequences that create a redundant object. They use Henkel and Diwan's term-based representation [51] for an object. However, Carlos *et al.* [74] found this representation cannot express reuse of an object (i.e., aliasing) and mutation (i.e., change of values) of an object via a method that mutates its parameters. Therefore, current test-suite reduction approaches need to be improved to address these limitations.

2.4. Crash Reproduction

To assist crash reproduction, many approaches have been proposed [14][18][19][22][24][46][69][76][80], including capture-replay and post-failure-process approaches. Capture-replay approaches monitor software execution by using software instrumentation or special hardware for storing monitored information to reproduce software execution [22] [69] [80]. For example, ReCrash instruments software capture all inputs of methods [22]. When a crash occurs, ReCrash reproduces the crash using the captured method input arguments. Capture-replay approaches reliably reproduce software execution, but incur non-trivial performance overhead. For example, ReCrash incurs 13-64% performance overhead. Due to this overhead, capture-replay techniques are not commonly used in practice.

Instead, post-failure-process approaches are widely deployed in practice [14][18][19] [24][46][79][76]. These approaches do not incur any overhead, since they do not capture information during software execution. These approaches store crash information, such as memory dumps or stack traces, only at the time of crashes. For example, the Windows Error Reporting (WER) system automatically dumps crash information, including stack traces, and sends it to a WER server [47]. Bug reporting systems, such as Bugzilla [79], also encourage users to include crash stack traces when they report bugs. This crash information is very useful for developers to debug crashes and prioritize their debugging efforts [46] [62]. However, developers should manually reproduce failures based on crash information.

CHAPTER 3. ENHANCED METHOD-SEQUENCE GENERATION

To generate test inputs automatically, random testing has been proposed [49], which selects test inputs randomly from the input domain of a program. The concept and implementation of random testing are relatively simple compared to other existing techniques, while random testing is offering several benefits as an effective black box testing technique, such as reliability. Random testing has been widely accepted because this technique does not require human intervention and avoids human bias for the generation of test cases. Human bias is problematic, because it only allows for input values and method-sequences that can be contrived by testers.

This chapter begins with explanations of existing method-sequence generation techniques and related problems. Then, it describes this study's approaches in two categories, input selection and method selection, to improve existing method-sequence generation techniques. Finally, it proposes a test input reduction technique to reduce the size of generated test inputs and method sequences.

3.1. Random Testing for Object-oriented Software

Random testing techniques have been also applied to testing of object-oriented software [41][49][74][77], which randomly create method-sequences to generate input objects. These techniques are spotlighted, due to their simplicity. Random testing for an object-oriented system is usually more scalable than other automated testing techniques [74]. For example, state-of-the-art automated test input generation techniques, including exhaustive testing [86][88][89], can only test a limited number of classes at a time, primarily because memory

usage becomes exponentially large as the number of classes increases. Random testing is easier to scale up, because this technique keeps input data, which is not exponentially large.

Although random testing techniques are scalable, they cannot achieve high code coverage on large-scale software systems (e.g., a system has over 1,000 methods). These techniques often generate many redundant test cases that do not contribute to increasing code coverage with low code coverage (e.g., lesser than 50% of code coverage - see our empirical study results). To be widely useful, random testing techniques should scale and yield higher code coverage even for large-scale systems.

Constructing unit tests for object-oriented 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 choice of method/input selection strategies can have a considerable impact on the effectiveness of method-sequence generation techniques. Therefore, finding a better method/input selection strategy is crucial for testing object-oriented programs.

3.2. Adaptive Random Testing for Object-oriented Software

It has been argued that random testing lacks a systematic approach to learn from previously executed input data. Alternatively, adaptive random testing (ART) [33] 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 area of automated test input generation. One study shows that ART is at least 50% more efficient than random testing.

There are many ART approaches [33][33][35]. Among them, the simplest adaptive random selection technique is Distance-based Adaptive Random Testing (D-ART), also called fixed-sized-candidate-set ART [32]. D-ART uses two input data sets; a candidate set has several input data and an executed set saves 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 chooses the farthest candidate as an input value.

Ciupa *et al.* suggest adaptive random testing for object-oriented software (ARTOO) [35] that applies the D-ART approach to the object-oriented system by choosing input objects from an object pool, which contains available object inputs. Because their object-oriented version of D-ART needs to calculate the distance between objects, they introduced a new notion, *object distance*. As a result, ARTOO found the first fault in a much smaller number of test inputs, using only 20% of the number of test inputs required by an undirected random testing approach.

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 a summation of these three components with weights and normalization.

A major problem with 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

[35], it took a much longer time (160% from unguided random testing) because of the calculation of the object's distance, although ARTOO generates a smaller number of test inputs.

3.3. Feedback-directed Input Generation

This section describes the feedback-directed input generation [73] technique in detail. The feedback-directed input generation technique incrementally generates method-sequences by selecting a target method being tested and selecting inputs to the method. Feedback-directed input generation starts with a set of primitive type declarations with predefined values as initial inputs for an object generation seed. It generates method-sequences based on relationships between the argument types and return types for each method. This technique executes the generated method-sequence to verify the newly created sequence (the one just extended) and obtain feedback (return values or generated objects) from the execution of the previously generated sequences. Feedback-directed input generation uses this feedback as an input for a new method-sequence. By doing this, executing generated method-sequences incrementally create new inputs. These modified inputs can be used as generation seeds to generate more method-sequences, which create more objects.

The following steps describe the process of feedback-directed input generation technique:

1) The feedback-directed input generation technique selects one of the methods under test.
2) To determine input arguments and a receiver for this selected method, the technique searches sequences from a sequence pool, which contains previously constructed sequences.

3) If the technique finds sequences that construct the same type of objects as the type of one of the arguments and the receiver of the selected method, the technique merges these sequences.
4) Then, the technique appends the selected method to the end of the merged sequence to make a new sequence.

Randoop is a well-known and state-of-the-art random testing tool for object-oriented systems that implement feedback-directed input. Randoop creates JUnit [26] test cases, based on the sequences. The experimental results show that Randoop outperforms systematic and unguided random test input generation tools in terms of code coverage and error detection [74].

For example, assume we specify the class `BankAccount` mentioned in Figure 3. Randoop might randomly choose one of `BankAccount`'s constructors as a target method and generate a sequence that creates a `BankAccount` instance. After creating the instance, Randoop might randomly choose one of `BankAccount`'s methods as a target method being tested. Suppose there are only two methods in the class `BankAccount`: `void deposit(int)` and `boolean verifyAccountOwner(BankClient)`. If Randoop chooses the `deposit(int)` as a method under test, it selects a predefined integer declaration as the input value of the method, and successfully composes a new method-sequence. On the contrary, if it chooses the `verifyAccountOwner(BankClient)` as a method under test and there is no method that returns a `BankClient` instance in the list of methods under tests, `verifyAccountOwner` cannot be tested. This input finding strategy is called bottom-up strategy.

However, Randoop achieves only 50% of code coverage as discussed in Chapter 1. Based on this study's analysis, there are four important reasons for the lower code coverage: First,

Randoop uses an undirected random selection strategy to select input data, although these inputs are based on feedback. Therefore, an adaptive distribution of input selection (a directed random selection) may be desirable. This problem is addressed through this study's enhanced input selection techniques.

Second, method selection for the next target method is random. In detail, it retrieves a complete method list of classes under test, then randomly chooses one method as a target method. Although this strategy retains fairness for a method selection, it makes the growth of code coverage slower when a strategy selects a method that does not contribute to increasing code coverage. This problem is addressed through this study's enhanced method selection techniques.

Third, Randoop only focuses on use of feedback from execution and a bottom-up (accumulative) strategy to determine proper inputs of the method under test. If required arguments or receiver objects for a method under test are not available (i.e., all of these instances cannot be produced at that moment), the random method selection algorithm gives up and tries to select another method for testing. This strategy makes the test of multi-parameter methods even more difficult. For example, if one method needs five different types of objects as input arguments, it is more likely to give up testing this method than a method with one argument. This problem is addressed by this study's on-demand input creation technique.

Fourth, Randoop generates too many redundant test cases, although it tries to eliminate redundant test cases, based on state matching during test input generation. Indeed, Randoop generates a huge number of redundant test cases. For example, it often generates more than 1,000 test cases in five seconds, but most of them are not helpful to increase code coverage.

Executing test cases takes a long time; thus, having a minimal set of test cases is important for efficient testing. Some test reduction techniques have been proposed [32][50][52][70] to obtain a smaller set of test cases by eliminating redundant test cases. However, to eliminate redundant test cases requires considerable extra time to execute all test cases to measure coverage and determine the redundancy. This problem is addressed by this study's test input reduction techniques.

3.4. Enhanced Input Selection

This study proposes three input selection approaches that collectively help achieve higher code coverage with a small set of test cases. In addition, each approach can be independently pluggable in the existing random testing algorithms. This study's approaches use different selection points in the construction of test inputs.

- *Simplified distance-based Selection* collects the farthest test input from the used values with lower computation cost than ARTOO.
- *On-demand Input Creation* immediately generates needed inputs (such as arguments and receiver objects) for the method under test. This technique also generates arrays by gathering values that have a requested element type.
- *Type-based Selection* gives equal chances between different data types to be chosen.

As said ealier, these approaches are pluggable in existing random testing algorithms.

3.4.1. Simplified Distance-based Input Selection

ARTOO's object calculation requires computational overhead as described in Section 3.2. This study suggests a simplified object distance, which is more efficient and requires lower computation cost. Input data types for this study are classified into three categories—primitive types (including boxed types and a string type), array types, and object types. This categorization removes vagueness of the ARTOO's object distance algorithm.

Primitive Type

This study determines the distance between two primitive types as follows:

- 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 constant value which is greater than 0.
- String type: the Levenshtein distance [64]. It measures the minimum number of edits needed to transform one string into the other, with the allowable edit operations being insertion, deletion, or substitution of a single character.

Array-type distance

For an array-type distance, we consider element type. If an array element type is a primitive, distance between two arrays is calculated as follows:

- *identical*: array size is identical, type of each element is compatible[1], and all the element values are the same.
- *same_size*: array size is identical, type of each element is compatible, but some of element values are not identical. Calculate $\sum|a(i) - b(i)|$, where $0 < i \leq$ the $size\ of$ the $array$, and $a(i)$ and $b(i)$are i_{th} elements of two arrays, respectively.
- *different_size* : array size is different, type of each element is compatible, calculate $\sum|x - y|$, where x and y are a size of each array.
- *different*: they have different types of elements.

Normalize and provide a weight for each category to make the following equation:

$$0 = identical < same\_size < different\_size < different = 1.$$

Object type distance

Categorize 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:

[1] This study defines compatible types that has the same super class.

$$0 = identical < similar < compatible < different = 1.$$

We test the equality of two objects by the `equal()` method and `==` operator to decide whether they are identical. If their types are identical, this study's 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, the algorithm explores their super classes to decide whether their distance is compatible or different.

3.4.2. On-demand Input Creation

This study proposes the on-demand input creation technique, which actively creates necessary test inputs on-demand in the absence of necessary objects (including an instance that has no chance to be created). If a method under test requires an input argument, Randoop checks the availability of this type of object. If there are no available instances, the tool will not test the selected method. On the contrary, this study's on-demand input creation approach generates all required arguments and receiver objects.

For example, when `boolean verifyAccountOwner(BankClient)` method has been chosen, and currently no sequences return a `BankClient` instance, the on-demand input creation technique searches the constructor of `BankClient` class to compose a legal sequence. On the contrary, Randoop gives up testing this method because no sequences return `BankClient` instance.

Algorithm 1 represents the `getInputSequences` method, a core component of the on-demand input creation approach. The `getInputSequences` method creates inputs on-demand if there is no corresponding feedback. First, on-demand input creation checks argument types

of the method under test and places the argument types into an array variable called *P* (Line 2). *S* is an array that has the same size as *P*. *S* records sequences that have corresponding feedback with respect to input arguments in *P*. *Pi* and *Si* indicate *ith* element of *P* and *S*, respectively. From lines 6 to 32, the algorithm traverses each argument to find a sequence that returns the same type of feedback. If the algorithm cannot find the same type of feedback, the algorithm reaches Line 17. If the argument type is an array, the algorithm generates a sequence that returns an array. *arrayGeneration* creates and returns an array by searching the feedback, based on an element type of array.

```
procedure getInputSequences(methodUnderTest)
input
  methodUnderTest a signature of method under test
output
  S               sequences for each argument
begin
01  //check input argument types of the method under test
02  P = getInputParamTypes(methodUnderTest)
03  S = {}
04
05  for (i = 0 to P - 1))
06      //search previously constructed sequences
07      sequencesForType = getSequencesFromInputType(Pi)
08
09      if sequencesForType > 0 then
10         //Select only one sequence for the type
11         Si = randomSelect(sequencesForType)
12         continue
13      end if
14
15      //if there is no feedback for the input arguments,
16      //create input arguments.
17      if Pi is array then
18         Si = arrayGeneration(Pi)
19         continue
20      end if
21
22      if Pi is abstract or interface then
23         constructors = findCompatibleConstructors(Pi)
24      else if Pi is object then
25         constructors = findObjectConstructors(Pi)
26      end if
27
28      constructor = randomSelect(constructors)
29      //Recursively call getInputSequences()
30      inputSeqforObject = getInputSequences(constructor)
31      Si = objCreationSeq(constructor, inputSeqforObject)
32  end for
33  return S
end
```

Algorithm 1. On-demand input creation

If the argument type is class, `findCompatibleConstructors` finds all constructors of the class (lines 22-26). If the class type is abstract or interface, `findObjectConstructors` finds all possible constructors of compatible classes to make an input instance. Note, if the constructor needs other objects as arguments again, this study's approach recursively searches

input objects within a preset depth. In Line 30, the algorithm calls `getInputSequences` recursively, since the selected constructor method might need other necessary input objects to be invoked. The statement in Line 31 creates an argument by calling the constructor. Finally, Line 33, the algorithm returns all the found and generated sequences that create arguments.

Array Input Generation

Array is one of the input types that make the size of its input domain exponentially huge. For example, suppose there is an array of type t. If the domain size of type t is k and the length of the array is θ in length, the domain size of the array is k^{θ}. Like an object type, an array's high dimension of the input domain makes it difficult to achieve high code coverage. Nevertheless, a simple array generation technique increases code coverage in spite of the complexity of array input. Since Randoop does not provide a direct approach to generate arrays, Randoop lacks the strength to cover more branches that takes an array type of input. As a seed statement, Randoop declares primitive data type variables with random values. Through execution feedback, Randoop might be able to generate array type variables, but the possibility for this is low.

This study's array generation approach is described 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, this study also uses the type-based input selection approach. After listing all the available values, the algorithm chooses a size of 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. By

doing this, our array generation technique generates arrays as input data from non-array feedback.

3.4.3. Type-based Input Selection

Randoop generates a sequence of statements. Each element in the sequence represents a particular statement, including method calls, assignments, and declarations. Every variable used as an input to a statement is the result of a previous statement in the sequence. Randoop keeps a set of all generated sequences. When Randoop selects a new method, called `foo`, to create a method-sequence, it iterates all statements in the existing available sequences from the set to search values with a type matched to the input type. When possible values are found, one is selected to use as an input for the method `foo`.

There are two possible type-matching techniques to choose input data for generating test inputs—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 two objects have the common ancestor class or interface, their types are compatible. For instance, consider an interface, *I*, and two classes, *C* and *D*, which implement *I*. In this instance, *C* and *D* are compatible types.

An input selection technique can be wrongly directed if only one matching strategy between two type-matching techniques is used. By retaining only the identical type-matching technique, compatible type values never receive a chance for selection. On the other hand, if the compatible type-matching technique is only used, identical types have a lower probability of being selected than compatible types because the number of compatible type values is

greater, in general, than that of identical type values. Input data must be selected with equal probability among compatible and identical type values.

To solve this problem, fair probability is set between two type-matching techniques instead of using only one matching technique. For example, input data can be selected with a probability 0.5 for compatible type values and 0.5 for identical type values.

3.5. Enhanced Method Selection

This section describes this study's method selection techniques for method sequence generation. The *coverage-based method selection* technique prioritizes target methods to maximize the coverage in a given test time. *Open-access selection* allows the testing of all protected and private classes and methods.

3.5.1. Coverage-based Method Selection

Existing random testing tools randomly choose the method under test. Alternatively, testers manually specify test order of methods under test. It is believed an appropriate selection of methods during test input generation improves code coverage. There are often over 1,000 testable methods in a large-scale software system [56]. Therefore, it is difficult to manually determine the optimal test order of methods under test.

The coverage-based method selection technique prioritizes methods under test, based upon their coverage to maximize the coverage in a given test time. A lower-coverage method has higher priority. Since this technique gives a higher priority to methods under test that have not been exercised during the method-sequence generation and avoids testing the higher coverage method multiple times, this technique increases coverage. This technique reduces

the generation time required towards preset coverage goal by giving the higher probability to be selected to low-coverage methods (i.e., that have lower branch coverage). The coverage-based method selection avoids the inefficiency of choosing high-coverage methods (have a higher branch coverage) multiple times during method-sequence generation.

This study prioritizes methods based upon probability weights. The following function *f*(*m*) returns a weight of a method, *m,* to be selected:

Equation 1. Weight of method selection

$$f(m) = \gamma \left(\frac{1}{\text{coverage of } m}\right) + \eta \left(\frac{1}{\text{coverage of the class of } m}\right) + \lambda \left(\frac{\max(\text{test counts of } a \mid a \in all\_methods)}{\text{test counts } of\ m}\right)$$

where *m* is a method under test, "coverage of *m*" is code coverage rate of *m*, "coverage of the class of *m*" is a code coverage rate of *m*'s receiver object, "test counts of m" indicates the number of times *m* has been successfully tested, and " $\max(\text{test counts of } a \mid a \in all\_methods)$" denotes the maximum of test counts among all methods under test. If code coverage of a method under test or a class of the method is higher, the method has a lower chance of being chosen for the test. γ, η, and λ are positive values are chosen based on our empirical study. This study uses branch coverage as code coverage, $\gamma = 2$, $\eta = 3$, and $\lambda = 1$ in its experiment.

According to the function *f*(*m*), if code coverage of a method under test and a receiver class of the method is higher and the count of a method that has been selected more than other methods, the method has a lower probability of being tested. In other words, the methods that do not have potential to increase code coverage will have lower probability of being tested.

3.5.2. Open-access Method Selection

The *open-access method selection* technique can test non-public methods. In Java, private and protected methods are difficult to test, since these non-public methods cannot be accessed from outside classes or packages. This study found that 17% for Ant, 28% for Java containers, and 34% for ASM methods are non-public methods. The *open-access method selection* increases code coverage by changing non-public methods to public ones.

To include non-public methods for testing, this study modified method accessibility by using the ASM Bytecode Framework [5]. The open-access method selection approach searches all non-public methods and classes, and changes them to public ones at the bytecode level.

3.6. Test Input Reduction

An important goal of automated test input generation techniques, including method-sequence generation, is to achieve high code coverage with a small set of inputs. Although researchers have proposed many techniques to achieve this goal, the method-sequence generation technique generates a huge number of redundant test inputs. For example, it often generates more than 1,000 test inputs in five seconds, but most of them are not helpful to in-

crease code coverage. Executing test inputs takes a long time; thus, having a smaller set of test inputs is important for efficient testing.

This study applied two reduction techniques: (1) coverage-based reduction and (2) sequence-based reduction. *Coverage-based reduction* was proposed by Harrold *et al.* [50]. This technique checked coverage for each test input to determine redundancy. Additionally, sequence-based reduction was proposed as a new approach in this paper. *Sequence-based reduction* detected redundancy by checking the inclusion-relationship of method-sequences. By appling both techniques, our experiments showed that approximately 99% of the test inputs were redundant and can be removed without diminishing code coverage.

3.6.1. Coverage-based Reduction

A newly created sequence may not improve code coverage. In this situation, the new sequence is redundant in terms of code coverage, although the new sequence is not identical with and behaviorally different from the previous sequences. Coverage-based reduction removes this kind of redundant sequence. The redundant test input, based upon coverage, is defined as follows.

Definition 1 *A test input, t, is redundant for a test-suite, S, if and only if there exists a test input t' from S such that cov(t)⊆cov(t'), where cov(t) returns coverage by t, and cov(t') by t'.*

Figure 4 shows an example. There are two test cases `test12()` and `test34()`; both test `TreeMap.remove()` with the different input values. `test12()` uses a `Short` type variable as an input, and `test34()` uses an `Integer` type variable. They have values 1 and 10, respectively. `TreeMap` instance `var0` does not have any element inside. Thus, the `containsKey`

method cannot find an element by any key. Therefore, the code coverage of the target method is not different, although these two test cases use different types and values. Consequently, one of these test cases is redundant in terms of code coverage.

```
public void test12() throws Throwable {
  TreeMap var0 = new TreeMap();
  Short var1 = new Short((short)1);
  Object var2 = var0.remove((Object) var1);
  // Regression assertion
  assertTrue(var2 == null);
}

public void test34() throws Throwable {
  TreeMap var0 = new TreeMap();
  Integer var1 = new Integer(10);
  Object var2 = var0.remove((Object) var1);
  // Regression assertion
  assertTrue(var2 == null);
}
```

Figure 4. Two method-sequences with the same coverage

This study applies the coverage-based reduction technique to Randoop. To apply the coverage-based reduction technique, instrumented symbols in the bytecode to measure code coverage are used; thus, the reduction technique can check code coverage immediately after generating a new method-sequence. If a newly generated method-sequence covers only the previously covered areas, the sequence is considered redundant.

For more details, the coverage-based reduction technique adds a variable for each branch to record whether this branch has been reached. Therefore, if a new method-sequence covers one or more branches that have never been reached, then this sequence is not redundant.

Moreover, because the on-demand input creation technique saves not only different method-sequences, but also all involved objects (including return objects and involved receiver objects), the coverage-based reduction technique can easily reuse the existing receiver and arguments to directly execute the new sequence to measure code coverage.

For example, let's recall the method-sequence in Figure 3 as *seqA*:

```
01 Int a=1000;  //Variable declaration
02 Int b=-1;    //Variable declaration
03
04 //Construct with a initial balance
05 BankAccount account = new BankAccount(a);
06
07 //Test with -1
08 account.deposit(b);
09
10 //Test with 1000
11 account.deposit(a);
```

If the on-demand input creation technique generates and saves *seqA*, then there is a `BankAccount` instance with `deposit = 999` in an object pool. Now, assume a newly created sequence *seqB* by adding the following bold lines (lines 12 and 13) to *seqA*:

```
01 Int a=1000;  //Variable declaration
02 Int b=-1;    //Variable declaration
03
04 //Construct with a initial balance
05 BankAccount account = new BankAccount(a);
06
07 'Test with -1
08 account.deposit(b);
09
10 'Test with 1000
11 account.deposit(a);
12 int c=Integer.NAN;
13 account.deposit(c);
```

For testing code coverage of *seqB*, it is not necessary to rerun statements in lines 1-11 (i.e., *seqA*) of Figure 3, because the *seqA*'s related receiver object from the object pool were just queried. And, statements in lines 12 and 13 are executed. Finally, it is recognized that *seqB* touches some branches that have not yet been covered.

3.6.2. Sequence-based Reduction

A test input of object-oriented systems consists of method invocations that create and modify both receiver object and arguments. However, existing test input reduction techniques do not consider method-sequence. Previous approaches consider only coverage of test inputs or only receiver objects to remove redundant test inputs.

The sequence-based reduction technique prunes redundant method-sequences by checking the inclusion-relation of the generated test cases. Since this technique does not check code coverage, it does not need to execute any test cases to determine redundancy.

We define redundant test inputs, based upon sequence relationships as follows.

Definition 2 *A test input, t, is redundant for a test-suite, S, if and only if there exists a test input t' from S such that is seq(t)* $\subseteq$ *seq(t'), where seq(t) and seq(t') each returns a sequence set of method invocations of t and t', respectively.*

The sequence-based reduction technique checks whether a sequence is a subset of another sequence. If this is the case, eliminate the sequence. For example, *SeqB* is based on *SeqA*. Because *SeqA* is composed of statements in lines 1-11, and *SeqB* is composed of statement in lines 1-13, it is easy to see that sequence A is a subset of *SeqB* (i.e., *SeqA* $\subseteq$ *SeqB*). Therefore,

each branch reached by *SeqA* is definitely reached by *SeqB*. In this case, the sequence-based reduction technique eliminates *SeqA*.

Randoop incrementally generates test inputs based on previously constructed test inputs. It concatenates sequences that create input arguments of a method under test. After concatenation, the algorithm appends this method invocation under test at the end of the concatenated sequence. In this process, it is possible to generate many redundant method-sequences. To explain more clearly, assume a method `rec.method(arg)`, where `rec` is a receiver object and `arg` is an argument of method. There should be two sequences called *SeqRec* and *SeqArg* that create `rec` and `arg`, respectively. To create a valid method-sequence that tests `rec.method(arg)`, *SeqRec* and *SeqArg* need to be merged first. Name the merged sequence *SeqRecArg*. Finally, construct a new sequence called *SeqValid* by appending the statement to call `rec.method(arg)` at the end of *SeqRecArg*. It is obvious that each *SeqRec* and *SeqArg* is a subset of *SeqValid*.

Figure 5, 6, and 7 show test inputs generated by CAPTIG. The test input `test23()` in Figure 5 tests the `TreeMap.get()` method. Before calling the method, CAPTIG creates a `TreeMap` instance and a `Byte` type input value. Similarly, `test53()` in Figure 6 tests the `TreeMap.put()` method.

```
public void test23() throws Throwable {
  TreeMap var0 = new TreeMap();
  Byte var1 = new Byte((byte)10);
  Object var2 = var0.get((Object)var1);

  //Regression assertion
  assertTrue(var2 == null);
}
```

Figure 5. Test case 23

```
public void test53() throws Throwable {
  TreeMap var0 = new TreeMap();
  Object var1 = new Object();
  Object var2 = new Object();
  Object var3 = var0.put(var1, var2);

  //Regression assertion
  assertTrue(var3 == null);
}
```

Figure 6. Test case 53

Figure 7 shows a test case created by merging Figure 5 and Figure 6. The purpose of the test case `test72()` is to test the `TreeMap.containsKey()` method. As shown in Figure 7, the method-sequence includes the method-sequences from Figure 5 and Figure 6, and the `containsKey` method is appended at the end (except assertion and exception statements). In this case, our sequence-based reduction technique prunes `test23()` and `test53()` because they are redundant and unnecessary.

```
public void test72() throws Throwable {
  TreeMap var0 = new TreeMap();
  Byte var1 = new Byte((byte)10);
  Object var2 = var0.get((Object)var1);
  TreeMap var3 = new TreeMap();
  Object var4 = new Object();
  Object var5 = new Object();
  Object var6 = var3.put(var1, var2);

  try {
    boolean var7 = var3.containsKey((Object)var4);
    fail("Expected exception of type ClassCastException");
  } catch(ClassCastException e) {
    //Expected exception
  }

  //Regression assertion
  assertTrue(var2 == null);
  //Regression assertion
  assertTrue(var6 == null);
}
```

Figure 7. Test case 72

The benefits of the sequence-based reduction are the following. (1) Sequence-based reduction does not decrease code coverage, because code coverage of a subset sequence is also a subset of the code coverage of a superset sequence. (2) Sequence-based reduction does not require execution of test inputs.

CHAPTER 4. OBJECT-CAPTURE-BASED INPUT GENERATION

In this section, an approach that captures objects dynamically from program executions (e.g., ones from system testing or real use) is proposed. The captured objects assist an existing automated test input generation tool, such as a random testing tool, to achieve higher code coverage. Afterwards, this approach mutates collected instances, based on observed but not-covered branches.

4.1. Overview

CAPTIG accepts a set of classes under test (i.e., the classes whose unit tests are automatically generated). CAPTIG produces a set of unit tests for the given classes under test, with the objective of achieving high structural code coverage, such as branch coverage.

To generate unit tests for the classes under test, the key idea of CAPTIG is to capture objects that can be used directly or indirectly to generate unit tests for these classes. Some example types of objects include (1) classes under test (e.g., receiver classes), (2) arguments of a method under test, and (3) objects needed to directly or indirectly construct the first two types of objects.

To capture these types of objects, CAPTIG requires normal program executions, such as those resulting from system tests and user interactions with systems. A system with normal program executions (used by CAPTIG for capturing objects) is called a system under monitoring. Note, a system under monitoring is not necessarily the system where the classes under test are integrated. A system under monitoring can be another system that uses (either consumes or produces) objects falling into the first and third types of objects described above.

That is, CAPTIG can be applied even before classes under test are integrated. Existing manually written unit tests or automatically generated unit tests for these classes under test can also be used to produce normal program executions. In this case, a system under monitoring is just the classes under test.

Figure 8 shows the overview of CAPTIG's three phases—object capturing (CAP), object generation (MET), and object mutation (PRE). In the CAP phase, CAPTIG conducts bytecode instrumentation of the system under monitoring and then executes the instrumented system, instead of the original one with given program executions. During these executions, the instrumented system captures encountered objects and serializes them into a file. In the MET phase, CAPTIG generates the captured objects by feeding these objects to an existing automated test input generation tool that produces method-sequences. These generated method-sequences derive objects from the captured objects. In the PRE phase, to cover those branches not yet covered by the MET phase, CAPTIG mutates captured objects in an attempt to cover not-yet-covered branches.

4.2. Object Capture (CAP)

The CAP phase consists of two parts—instrumentation and object serialization. In the first part, instrumentation, CAPTIG inserts object-capturing code at each method's entry point of the system under monitoring. The inserted code invokes a `capture` procedure shown in Algorithm 2. Pseudo-code for object capturing and passes arguments to CAPTIG's serialization module. The `capture` procedure captures objects along with their types. Figure 9 shows an instrumented code example.

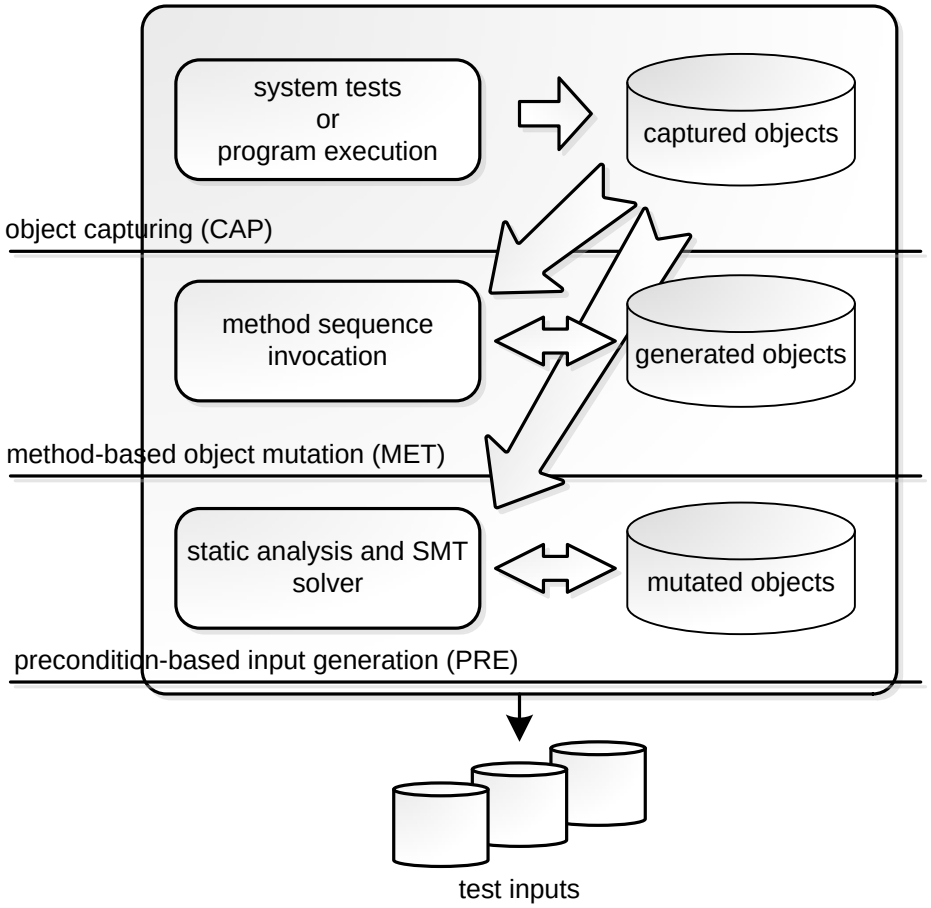

Figure 8. Overview of CAPTIG approach for object capturing

The next part is *object serialization*, described in Algorithm 2. First, run the instrumented system to serialize objects through normal program executions. If the instrumented capture procedure is invoked, the `capture` procedure receives a type and a concrete state of the object to be serialized. Then, the `capture` procedure keeps types and states of all serialized objects to maintain an object repository without redundant objects with the same state [88]. This study's approach serializes objects in memory first and serializes the objects into files whenever the number of objects reaches a preset limit or the program under monitoring finishes its execution.

```
procedure capture(objs[])
input
  objs[]
global
  Map<type, state> mapStates //keeps objects to avoid having redundants
  Set<object> setObjs  // keeps objects to be serialized
begin
01    for each obj ∈ objs[]
02       //get a type and linearized state of the object obj
03       type ← getType(obj);
04       state ← representState(obj);
05
06       //check whether the obj has been serialized
07       if (!isSerialized(type, state)) then
08         mapStates.add(type, state);
09         setObjs.add(obj);
10       end
11    end
12
13    //check whether enough number of obj has been serialized
14    if (hasEnoughInstances(setObjs)) then
15       // serialize objects and clear the setObjs set
16       serializeToFileAndClearSet_Thread(setObjs);
17    end
end
```

Algorithm 2. Pseudo-code for object capturing

```
public Algorithm(Document doc, String algorithmURI) {

  // instrumented
  ObjCapture.capture(new Object[]{this, doc, algorithmURI});

  super(doc);
  this.setAlgorithmURI (algorithmURI);
}
```

Figure 9. Code instrumented from the code shown in Figure 1

One challenge here is there are many types of objects in program executions and the same type of objects can also have many instances. It was observed that most of these objects have

isomorphic states, and these instances do not contribute to increasing code coverage. Therefore, it is desirable to capture only objects with non-isomorphic states for each class type. To check state isomorphism of objects, OCAT uses a concrete state representation. Xie *et al.* [88] defined a state representation of a program heap. This study adopted a part of its definition to define the state of an object to be a subset of a program heap state.

Let P be the set consisting of all primitive values, including `null`. Let O be a set of objects whose fields are from a set F.

DEFINITON 1. A state is an edge-labeled graph$\langle O, E\rangle$, where

$$E = \{\, \langle o, f, \bar{o}\rangle \mid o \in O, f \in F, \bar{o} \in O \cup P \,\} \ .$$

State isomorphism is defined as graph isomorphism, based on node bijection [31].

DEFINITON 2. Two states, $\langle O_1, E_1\rangle$ and $\langle O_2, E_2\rangle$, are isomorphic iff there is a bijection $\rho = O_1 \rightarrow O_2$ such that:

$$E = \{\, \langle \rho(o), \rho(f), \rho(\bar{o})\rangle \mid \langle o, f, \bar{o}\rangle \in E_1, \bar{o} \in O_1\} \ \cup \ \{\, \langle \rho(o), \rho(f), \bar{o}\rangle \mid \langle o, f, \bar{o}\rangle \in E_1, \bar{o} \in P\} .$$

Note, these two isomorphic states have the same fields for all objects and the same values for all primitive fields.

These preceding definitions of state representation and isomorphic states help prune redundant objects. If two objects have isomorphic states, then these two instances are considered as redundant. Other than using object-state, isomorphism to detect redundant objects, two alternative ways could be used. One way is to use an abstract state representation [86]. However, an abstract state representation is too coarse-grained, since it ignores the field val-

ues in an object and checks only the structural shape of the object. Another way is to use `equals()`; however, the outcomes of executing `equals()` depend on its implementation, and `equals()` can be implemented in various ways by different developers. For example, `equals()` may be implemented, based upon an object's reference value. Although the reference values of two instances are different, these two instances could have the same states. In this case, using `equals()` identifies these two instances have different states.

According to the empirical study in Chapter 7, using an abstract state representation incurs storage of few objects and using `equals()` incurs storage of too many isomorphic objects. Consequently, both aforementioned alternative ways are inadequate to identify different objects for CAPTIG. Therefore, concrete state representation is used.

4.3. Method-based Object Mutation (MET)

This section describes the MET phase, which generates objects by invoking method-sequences with captured objects. After objects are captured and serialized from a system under monitoring, they are de-serialized and used as test inputs. Particularly, they leverage a method-sequence generation technique by using the captured objects in two ways. First, the captured objects can be directly used. Second, captured objects contribute to the creation of other necessary objects for testing. Let C be a set of captured objects by CAPTIG. Consider two target methods m_i of class i and m_j of class j. Next, consider two sets of desired objects, D_{m_i} and D_{m_j}, that cover code in methods m_i and m_j, respectively. Let R_{m_j} be a set of returned objects of invoking m_j on D_{m_j}. If $D_{m_i} \subseteq C$, the method m_i can be directly covered by

using captured objects. If $D_{m_i} \nsubseteq C$, but $D_{m_i} \subseteq R_{m_j}$ and $D_{m_j} \subseteq C$, then code in m_i can be indirectly covered by feeding the returned objects of invoking m_j on captured objects.

Any method-sequence generation technique can generate more objects from captured objects. This thesis uses the feedback-directed input generation (Randoop) technique [74] that randomly generates method-sequences and verifies their validity by execution. Randoop extends method-sequences by repeating the processes, such as method selection, sequence selection, merging, extension, and execution. By repeating the sequence-construction process, Randoop incrementally generates new objects. Randoop generates method-sequences starting from a set of primitive-type declarations with predefined values. In this study's case, Randoop starts with a set of method calls that de-serialize captured objects and declares other primitive-type input values. The de-serialized captured objects provide a good basis to lead the method-sequence generation technique to generate desirable objects.

Figure 10 presents an example of a generated sequence with objects captured. A captured instance of `FastTreeMap` is modified by invoking `add()` and `putAll()` methods with other captured objects. First, the example loads a `FastTreeMap` instance from captured objects as a receiver `var0`. Then, `var1`, `var2`, and `var3` are loaded as input arguments for the last two subsequent method calls in Figure 10. In particular, the method call `var0.add()` adds the loaded instance to `FastTreeMap` and `var0.putAll()` adds all elements of a captured instance of `BeanMap`.

```
FastTreeMap var0 = (FastTreeMap) Serializer.loadObject
                        ("capobj/FastTreeMap/hash_32");
String var1 = (String) Serializer.loadObject
                        ("capobj/String/hash_98 ");
BeanMap var2 = (BeanMap) Serializer.loadObject
                        ("capobj/BeanMap/ hash_32");
Integer var3 = (Integer) Serializer.loadObject
                        ("capobj/Integer/hash_808");
var0.add (var3, var1);
var0.putAll ((java.util.Map) var2);
```

Figure 10. A generated method-sequence with captured objects

Using captured objects as initial inputs reduces the huge search space of desirable objects in the method-sequence generation process, because the captured objects are likely close to desirable objects. Therefore, captured objects make the method-sequence generation approach effective to produce desirable objects, and construct method-sequences with the captured objects to achieve high code coverage. This study's empirical results (Chapter 7) show that using captured objects with Randoop significantly increases the code coverage achieved by Randoop alone.

4.4. Precondition-based Object Mutation (PRE)

Generating objects by invoking method-sequences with captured objects may not cover all branches. The PRE phase statically analyzes conditions of uncovered branches after executing MET. Then, the phase generates method-sequences for uncovered branches by purposely mutating captured objects.

Figure 11 shows an overview of the PRE phase. (1) CAPTIG identifies not-yet-covered branches by analyzing source code and branch coverage information. (2) CAPTIG conducts a

static analysis (e.g., the weakest precondition analysis) to collect preconditions starting from the not-yet-covered branches in a backward traversal manner of code analysis. (3) CAPTIG uses a Satisfiability Modulo Theories (SMT) solver [43] to get a satisfiable model of the computed precondition. (4) CAPTIG uses the solved solution from a SMT solver as a concrete input value of the method that has the targeted not-covered-branch. (5) If the solution is related to a member field of an object input, CAPTIG loads and mutates a captured object. CAPTIG uses the Java reflection API [81] to modify a value of the corresponding member field of a captured object, based upon the solution of the SMT solver.

After mutating objects, CAPTIG saves mutated objects into the object pool of the CAPTIG system. Then, CAPTIG generates method-sequences, as it did in the MET phase, now to test the target method that has not-covered branches with the newly mutated object inputs. After all, CAPTIG concretely executes the generated sequence to verify whether a mutated object satisfies the condition of a target branch.

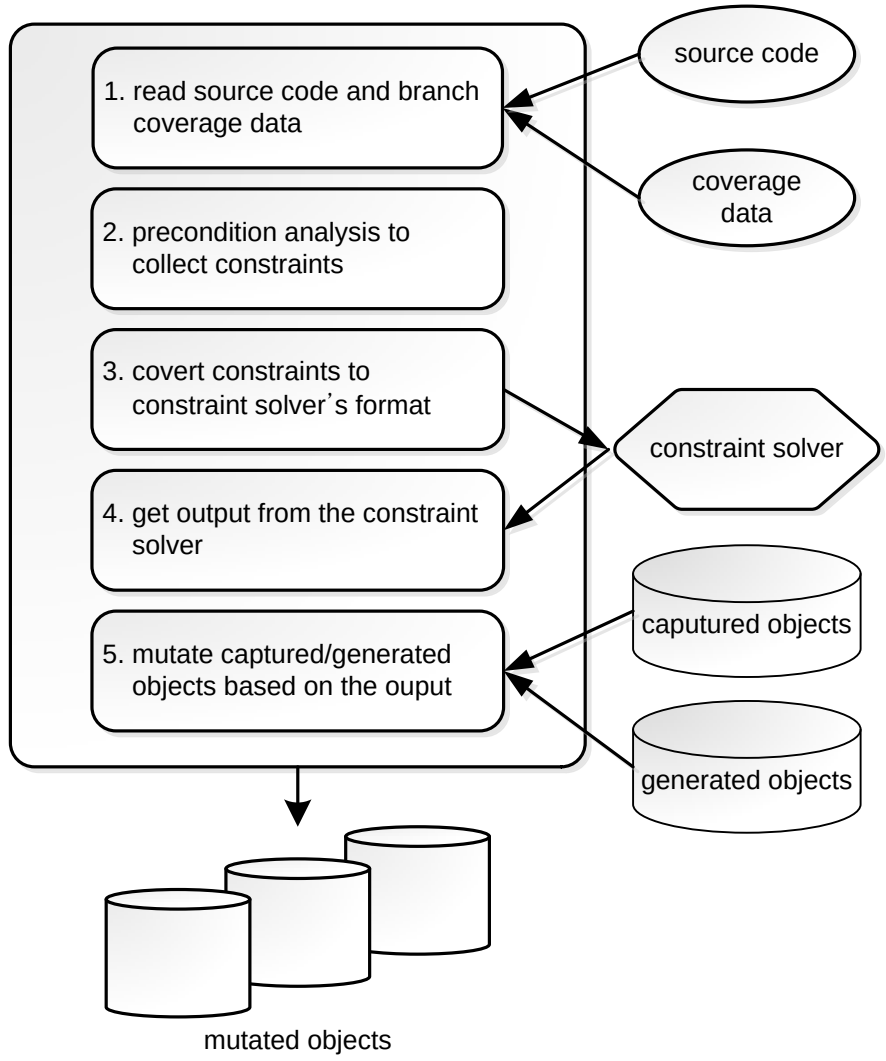

Figure 11. Overview of object mutation

The following steps illustrate the example of mutating an `AbstractRefereceMap` object in Apache Commons Collection (see Figure 12):

1. The coverage report indicates the true branch of "`parent.keyType > HARD`" has not been covered. CAPTIG parses the report and source code to determine the predicate of the not-covered branch.

2. CAPTIG checks variables in the predicate to determine whether they are input arguments including arguments, a receiver, or object fields of a receiver and arguments. Here, `HARD` is a constant field, whose value is 0 and `parent.keyType` is a protected member field of the receiver.
3. CAPTIG converts the predicate to the input format of Yices, a SMT solver: `(define var::int)(assert+ (> var 0))(check)`
4. Yices outputs (= var 1) from the input and OCAT parses the output.
5. CAPTIG randomly selects an instance of `AbstractReferenceMap` and modifies its `parent.keyType` to 1.

```
public static final int HARD = 0;
protected int keyType;

...

public Object getKey() {
return (parent.keyType > HARD)
          ? ((Reference) key).get()
          : key;
}
```

Figure 12. An object mutation example

When modifying member-field values of an object, CAPTIG does not change a private field value as a default setting, since modifying a private field value might break class invariants and make the object invalid. Indeed, the empirical results in Chapter 7 show only a few cases that a variable in a target predicate is related to a private field. However, it is good to have an optional functionality to handle this kind of occasional situations. To avoid invalid objects caused by modifying private field values, CAPTIG provides an option of allowing

developers to provide a predicate method (also called `repOk()` [31]) that checks class invariants. Programming best practices suggest that a programmer provides such a method when writing a class. After mutating an object, CAPTIG checks whether `repOk()` returns true to ensure state validity of the object [65]. For example, CAPTIG executes `repOk()` to check state validity and `getKey()` to verify it after Step 5. If it is still not a desirable receiver, it throws an exception when executing `getKey()`, or when `repOk()` returns `false`. CAPTIG repeats Step 5 for up to a preset number of times. If the mutated object is valid, CAPTIG serializes the object as a new instance and uses it as a test input for the method under test. Otherwise, CAPTIG tries to mutate other objects.

This study's mutation technique is related to dynamic symbolic execution techniques [63][77] in that both use constraint solvers to change objects and cover more branches using the changed objects. However, as shown in Figure 1, without desirable objects at the first phase, sometimes it is difficult to formulate all conditions and determine all constraints. There exist unrevealed conditions; thus, a constraint solver becomes ineffective in such cases. For example, without a desirable `doc` object, a program execution throws an exception at Line 87 in Figure 1. Moreover, although conditions are perfectly formulated and constraints are well determined, it is difficult to construct a desirable object that satisfies these constraints. Captured desirable objects help cover unrevealed conditions without perfect formulation of conditions. This study's CAPTIG approach uses desirable objects first by object capturing and object generation, and then applies object mutation to captured objects.

CHAPTER 5. CRASH REPRDUCTION

Generated test inputs cannot directly be used to find and fix a bug, although they achieve high code coverage. To show usability of generated method-sequences and objects, we attempt to solve a crash reproduction problem. In software development, reproducing given software crash in a predictable way is important. The reproduced crash is helpful to identify and finally repair the crash.

However, the process of reproducing crash is often labor intensive and time consuming, since software engineers frequently suffer to understand comments on a bug report and reproduce the crash to reveal its cause. In a practical testing process, a user or tester reports program crashes to bug report management systems for further tracking and reviewing. Then, a developer tries to understand and reproduce the reported crashes. Accordingly, technology that can assist software engineers to reproduce a crash effectively, based upon given information, is highly regarded.

To address this crash reproduction problem, we propose techniques and present an implementation, which automatically reproduces crashes using information collected from post-failure-process approaches. Our approach has many clear benefits: It is a fully automated approach. This can be applied to existing and widely deployed post-failure-process approaches such as WER [46], Apple Crash Reporter [8], Google Break Pad [14], and Bugzilla [79], without requiring additional instrumentation or special deployment. Therefore, this study's approach does not incur any performance overhead.

Our approach emulates developers' typical manual reproduction scenario. This approach first collects crashed call stack trace information from bug reports or post-failure-process sys-

tems. Crash call stack trace information (in short as crash stack trace) is active stack frames in the calling stack upon execution of a system at the moment of the system crash. Based upon the crash stack trace, our approach automatically generates test cases that reproduce a crash. For example, our approach invokes each method in the stack traces using generated inputs as a form of test case. Execution of these test cases can reproduce crashes. Our work focuses on reducing the amount of time for software engineers to reproduce a crash.

To illustrate the approach effectively, we use a simple source code example shown in Figure 13. The `BankAcc` class has one constructor and two public methods. The `deposit()` method deposits money into an account and the `withdraw()` method withdraws money from an account. The `AccCheck` class has the `checkAccounts()` method, which invokes three methods, (a) `deposit()`, (b) `summarize()`, and (c) `withdraw()` in lines 06, 11, and 13, respectively. When one of these methods are invoked, `checkAccounts()` crashes and produces stack traces as shown in Figure 13.

```
class BankAccount
{
01  public int pid;
02  public String name;
03  public double balance;
04
05  public BankAccount (int pid, String name, double balance)
06  { ... };
07  public void deposit (double amount) { ... };
08  public void withdraw (double amount) {
09   ...
10   throw new BankAccountException ();
11   ...
12  };
}

class AccCheck
{
01   static boolean checkAccounts(int branchID,
02                      ArrayList<BankAccount> arrAcc) {
03    int cnt = 0;
04    for (BankAccount acc : arrAcc)
05      if (acc.balance > 1000 && acc.balance <= 2000) {
06        acc.deposit(100);                    // crash method (a)
07        cnt++;
08      }
09
10    if (cnt > 500)
11      summarize(arrAcc);                     // crash method (b)
12    if (branchID == 1000 && arrAcc.get(0).pid == 10) {
13      arrAcc.get(0).withdraw(500);           // crash method (c)
14      return true;
15    }
16    return false;
17  }
18
19  static void summarize (ArrayList<BankAccount> arrAcc)
20  {  ...  }
}
```

Figure 13. Bank account and account check classes and methods

Using the crash stack traces shown in Figure 14, a developer can employ CAPTIG to generate test cases as shown in Figure 17, 19, and 21, which reproduce the original crashes. By executing these test cases, the developer can easily locate and fix the bugs.

```
BankAccountException:
at bank.BankAccount.deposit(BankAccount.java:7)
at bank.AccCheck.checkAccounts(AccCheck.java:6)
```

(a) crash stack trace of `deposit()`

```
BankAccountException:
at bank.AccCheck.summarize(AccCheck.java:216)
at bank.AccCheck.checkAccounts (AccCheck.java:11)
```

(b) crash stack trace of `summarize()`

```
BankAccountException:
at bank.BankAccount.withdraw(BankAccount.java:10)
at bank.AccCheck.checkAccounts (AccCheck.java:13)
```

(c) crash stack trace of `withdraw()`

Figure 14. Examples of crash stack traces

5.1. Architecture of Crash Reproduction

Our crash reproduction approach consists of three main modules: (1) stack trace processing, (2) test case template generation, and (3) input generation as shown in Figure 15. (1) As inputs, our approach only takes a crash stack trace and the location of classes (i.e., class paths) that includes crashed and related methods. Then, the stack trace processing module analyzes a given crash stack trace to extract initial information of the crash, such as the exception type, crash location, and crashed methods. (2) Based on the extracted information, the test case template generation module constructs a test case template, which contains receiver object, argument, their constructors, and crashed methods. (3) The input generation module generates inputs based on the test case template to complete a test case. To generate inputs, three input generation techniques are proposed—feedback-directed, object-capture-based, and precondition-based input generation.

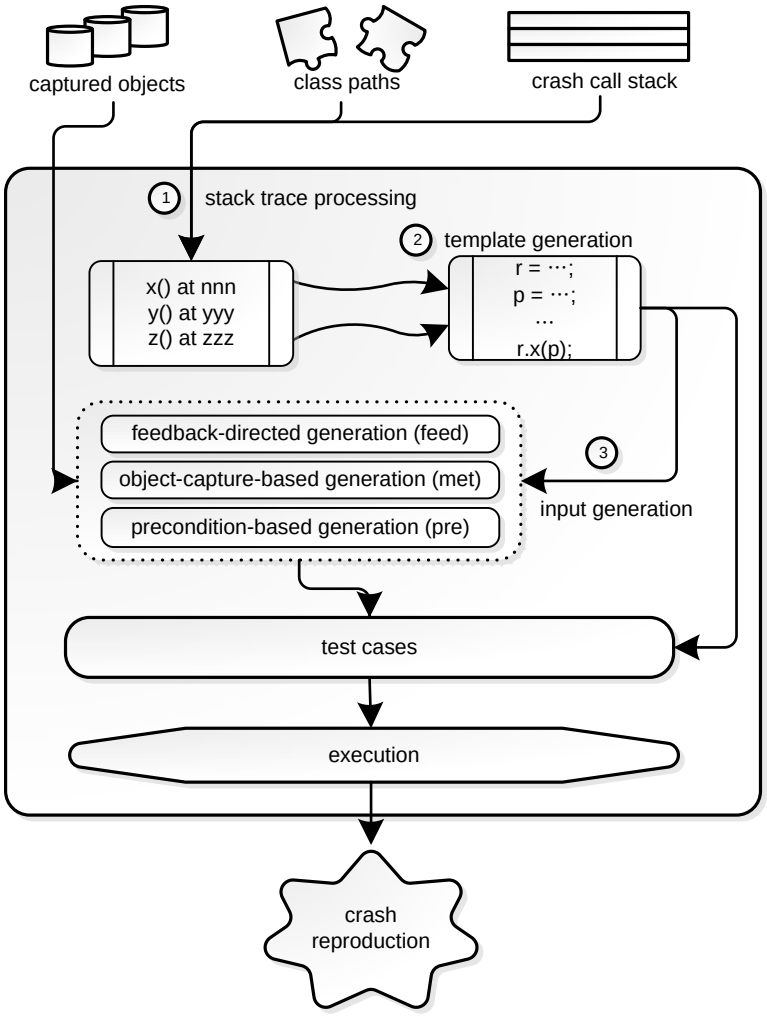

Figure 15. The architecture of crash reproduction.

5.2. Stack Trace Processing

CAPTIG extracts the following information from a given stack trace:

- an exception or assertion type,
- locations of the crash, and
- all class and method information in the stack trace.

First, CAPTIG extracts information in classes and methods from the crash stack trace. For example, from crash stack trace (a) in Figure 13, CAPTIG extracts class information (e.g., class name, type, namespace, hierarchy and etc.) of `bank.BankAccount` and `bank.AccCheck`, and method information of `deposit()` and `checkAccounts()`. Second, CAPTIG extracts object class types used in the receiver and arguments of crashed methods. For example, CAPTIG extracts crashed methods, `deposit()` and `checkAccounts()`, and argument classes of the crashed methods, `ArrayList`, `BankAccount`, `Double` and `Integer`[2] from Figure 13.

Third, CAPTIG finds all compatible class types of the extracted classes from the previous steps by analyzing their inheritance and implementation relationships of all classes in their class paths. If the extracted class is a concrete class, CAPTIG finds classes that share a superclass. If the extracted class is an interface or abstract class, CAPTIG finds all concrete classes that implement the interface or extend the abstract class. These extracted object class types are used in the input generation module.

5.3. Test Case Template Generation

The test case template generation module constructs test case templates, based on the information extracted by the stack trace processing module. A test case template contains intermediate information, such as a crashed method, its exception type, and argument types. This information is used to generate test inputs and test cases.

Figure 16 shows a test case template generated from crash stack trace (a) in Figure 13: lines 03-04 show the exception type and crashed method. Lines 01-02 indicate class types that

[2] Boxed data types are used for primitives.

should be prepared through input generation. Note, this example does not have a receiver class type for `AccCheck`, since `checkAccounts()` is a static method.

```
01: types: Integer ArrayList BankAccount
02: types(BankAccount): Integer String Double
03: exception: bank.BankAccountException
04: target: AccCheck.checkAccounts(Integer,
                                   ArrayList<BankAccount>)
```

Figure 16. A test case template of the crash (a) in Figure 13

5.4. Crash Input Generation (Feedback-directed)

The crash input generation modules generate test inputs for a receiver and arguments to reproduce crashes. To generate test inputs, an approach that incrementally conducts two input generation techniques is proposed—feedback-directed input generation and object-capture—based input generation. Each crash input generation technique generates test inputs to construct test cases that might reproduce crashes.

First, the feedback-directed input generation technique generates test inputs by constructing method-sequences [73]. Second, the object-capture-based technique generates test inputs by capturing objects from normal program execution. Since captured objects reflect a real usage of the system, using captured objects improves crash reproducibility [58]. The object-capture-based input generation technique is based on our capture-based input generation approach (including method-based and precondition-based object mutation) discussed in Section 4.

This study used an enhanced feedback-directed input generation algorithm [56][59] discussed in Chapter 3 to generate more diverse inputs. The technique was further modified to focus on crash information. First, higher priorities were set to the crashed method and related methods, while constructing method-sequences. Second, a criterion to stop generating inputs was set, when all instances of the necessary input types are created. This modification allows the algorithm to generate inputs that have a higher possibility of reproducing crashes.

Figure 17 presents an example of a test case by the feedback-directed input generation technique. The test case uses the method-sequence that creates an `ArrayList<BankAccount>` object and invokes `checkAccounts()` surrounded by the `try..catch` statement. This test case reproduces the crash (a) in Figure 14, located in line 06 of `AccCheck` in Figure 13.

```
public static void test1() {
01  int v0 = 1;
02  String v1 = "hi";
03  double v2 = 2000;
04  BankAccount v3 = new BankAccount(v0, v1, v2);
05  ArrayList<BankAccount> v4 = new ArrayList<BankAccount>();
06  v4.add(v3);
07
08  try {
09     boolean var = AccCheck.checkAccounts(v0, v4);
10     fail("Expected exception.");
11  } catch (bank.BankAccountException e) {
12     // Expected exception.
13  }
}
```

Figure 17. A test case that reproduces the crash (a) in Figure 14 based on the feedback-directed input generation technique.

5.5. Crash Input Generation (Object-capture-based)

The feedback-directed input generation technique is fast, but may not create all necessary objects because the search space for object generation is huge [58]. For example, in Figure 13, it is hard to generate inputs that reach the crashed invocation (b) in Line 11 by using the feedback-directed input generation technique, which cannot generate an array list that contains more than 500 objects of `BankAcc`.

To address this limitation, the object-capture-based input generation technique uses objects captured from normal program executions (e.g., system tests or planned normal executions). This technique provides additional objects and their variations to be used as inputs for a test case, which may increase the crash reproducibility of our approach.

5.5.1. Object Capturing

To capture objects, we use our object capturing approach (so-called CAP) described in Section 4.2. We first instruments the target system. Then, we execute the instrumented system, which captures objects into files during its execution. Figure 18 shows an example of a captured object from normal execution that creates `ArrayList<BankAccount>`. Note, the CAP process can be completed once in-house, it does not incur any additional overhead for deployed software.

```
<list>
  <BankAccount>
    <pid>1 </pid>
    <name>Kim</name>
    <balance>1201 </balance>
  </BankAccount>
  <BankAccount>
    <pid>2 </pid>
    <name>Steve</name>
    <balance>2238 </balance>
  </BankAccount>
...
...
  <BankAccount>
    <pid>1020 </pid>
    <name>Kevin</name>
    <balance>742 </balance>
  </BankAccount>
</list>
```

Figure 18. A captured object example - `ArrayList of BankAccount`

5.5.2. Method-based Object Mutation

To generate more crash reproducible objects, use the MET approach described in Section 4.3. The MET approach combines captured objects and feedback-directed input generation to generate more objects and their variations. The captured objects were used as seeds for the feedback-directed input generation. The feedback-directed input generation technique generates method-sequences related to the captured object and creates additional objects. These additional objects might increase crash reproducibility.

For example, suppose a crashed method takes a `Foo` object as an argument. The feedback-directed input generation identified a method-sequence, '`Foo getFoo(Bar bar);`', to create the `Foo` object. Suppose it was not possible to directly create `Bar` to invoke the method-sequence. Thus, the technique could not create the `Foo` object. Now, assume the object-capture-based technique cannot capture `Foo`, but is able to capture a `Bar` object. This object

can be a seed for the feedback-directed technique to invoke '`Foo getFoo(Bar bar)`' that creates a `Foo` object. Finally, the generated `Foo` object can be used to reproduce the original crash.

Figure 19 presents an example of a generated test case with captured objects. Two captured instances of `ArrayList` are de-serialized from lines 03 to 06. These two instances are merged by invoking `putAll()` methods in Line 07. Therefore, the captured instances can make crashed invocation (b), `checkAccounts()`, of Figure 13 and reproduce the crash (b) in Figure 14.

```
public static void test32() {
01  Integer v0 = (Integer) Serializer.
02            loadObject ("capobj/Integer/hash_808");
03  ArrayList v1 = (ArrayList) Serializer.
04            loadObject ("capobj/ArrayList/hash_35");
05  ArrayList v2 = (ArrayList) Serializer .
06            loadObject ("capobj/ArrayList/hash_48");
07  v1.putAll(v2);
08
09  try {
10     boolean v3 = AccCheck.checkAccounts (v0, v1);
11     fail ("Expected exception.");
12   } catch (bank.BankAccountException e) {
13    //  Expected exception
14   }
}
```

Figure 19. A test case that reproduces the crash (b) in Figure 14

5.5.3. Precondition-based Object Mutation

The precondition-based input generation technique (so-called PRE) statically analyzes the crashed methods and computes the weakest precondition [43] that could raise the original crash. Unlike the PRE phase in the object-capture-based input generation, this approach for

crash reproductions uses the crash stack traces instead of not-covered-branch information. This approach conducts a backward symbolic analysis from the crash location to the entry point of the crashed methods [36]. Then, it uses a Satisfiability Modulo Theories (SMT) solver [43] to check the satisfiability of the computed precondition. If it is satisfiable, the SMT solver returns a satisfiable model, which is a set of concrete values assigned to each relevant field or input argument to satisfy the computed precondition. Figure 20 shows an example of a satisfiable model. Finally, the PRE approach mutates objects, based on the satisfiable model to generate crash reproducible inputs. Currently, the PRE approach only supports two kinds of crashes—`NullPointerException` and user defined exceptions—raised by explicit throw statements (e.g., crash invocation (c) in Figure 13).

```
weakest precondition:
branchID == 1000                          (1)
arrAcc != null                            (2)
arrAcc.get(0) != null                     (3)
arrAcc.get(0).balance <= 1000 || > 2000   (4)
arrAcc.get(0).pid == 10                   (5)

A satisfiable model:
branchID = 1000                                  (1)
arrAcc = instanceof ArrayList<BankAccount>       (2)
arrAcc.get(0) = instanceof BankAccount           (3)
arrAcc.get(0).balance = 1000                     (4)
arrAcc.get(0).pid = 10                           (5)
```

Figure 20. The weakest precondition and a satisfiable model

To illustrate the PRE approach for crash reproduction, use the example in Figure 13. To reproduce the `BankAccountException` thrown by `withdraw()`, it is necessary to determine the feasible execution path from the entry point of `checkAccounts()` to the throw statement in `withdraw()`. Assume the crash stack trace (c) as shown in Figure 14. This stack trace in-

dicates the method ascend sequence for the backward computation. The backward computation begins from the exception thrown statement in `withdraw()`. Every path is enumerated in a depth first manner until the entry point of `checkAccounts()` is reached. Consider the backward path (`withdraw()`: lines 10, 09, 08, `checkAccounts()`: lines 13, 12, 10, 09, 05, 04, 03, 02, 01) traversed by `ComputeWP()`. Predicates `branchID == 1000`, `arrAcc != null`, `arrAcc.get(0) != null` and `arrAcc.get(0).pid == 10` are added to the precondition set when the algorithm computes the weakest precondition of Line 12 in `checkAccounts()`. These predicates are the essential conditions for successfully executing the field access statement and taking the desired branch at Line 12. Other predicates, such as `arrAcc.get(0).balance <=1000` or `>2000`, are also added, while the algorithm traverses the path in a backward manner until the entry point of `checkAccounts()`. Then, we check the satisfiability of this precondition using an SMT solver. Since this precondition is satisfiable, the SMT solver returns a satisfiable model as shown in Figure 20. Based on this model, the objects are mutated.

Figure 21 presents the generated test case guided by the satisfiable model in Figure 20.

```
public static void test50() {
01  // an object from feedback-direct technique
02  int v0 = 1;
03  String v1 = "hi";
04  double v2 = 2000;
05  BankAccount v3 = new BankAccount(v0, v1, v2);
06  ArrayList<BankAccount> v4 = new ArrayList<BankAccount>();
07  v4.add(v3);
08
09  // object mutation code generated by setValues()
10  v3.pid = 10;
11  v3.balance = 1000;
12  v5 = 1000
13
14  try {
15     boolean v6 = AccCheck.checkAccounts(v5, v4);
16     fail ("Expected exception");
17   } catch (bank.BankAccountException e) {
18    // Expected exception
19   }
}
```

Figure 21. A test case that reproduces the crash (c) in Figure 14

5.5.3.1. Weakest precondition computation

This section describes how to obtain the weakest precondition from crashed locations. The weakest precondition computation is an inter-procedural, path-sensitive, and context-sensitive backward analysis algorithm. Since a backward manner of analysis is used, the approach is similar to SnuggleBug [37] and XYLEM [68]. Similar precondition mapping rules (transformers) are used in SnuggleBug.

Unlike SnuggleBug, this approach narrows the overall backward analysis space by considering the method invocation sequences based on the crash stack traces. This can improve scalability, since this approach only considers relevant methods listed in the crash stack trace. Also, this approach is more precise than XYLEM, because an SMT solver [43] is employed to verify the satisfiability of the computed preconditions.

Algorithm 3. Weakest precondition computation algorithm for a crash condition presents our inter-procedural backward computation algorithm. Given a crash stack trace, `stackTrace`, the algorithm first creates a call stack `cs` from `stackTrace` (Line 01). The call stack `cs` indicates the method ascend sequence used for the backward computation process. The `startLine` of the backward computation is set to the line immediately before the crash location (Line 03) to compute the weakest precondition to reach the crash location with the desired crash condition `failCond`. A crash condition is the necessary precondition to reproduce the original crash at the crashed line. To begin the backward computation process, this approach invokes `ComputeMethod()` with the crashed method signature, the starting line number, the crash call stack created from `stackTrace`, the current invocation depth (initially, 0), and the crash condition (Line 04).

The starting instruction of the backward computation is retrieved from the `getStartingInstruction()` procedure (Line 05). This is the Java instruction where the backward computation starts. Then, a stack structure `stDfs` is created or retrieved from `htPost2Dfs` (Line 07-09). A `stDfs` holds the set of instructions whose preconditions will be computed. Initially, the starting instruction and the input `postCond` are pushed into `stDfs` (Line 09).

`ComputeMethod()` computes the weakest precondition in a backward and depth first manner [68]. Inside a loop, the (`inst`,`postCond`) pair that lies on the top of `stDfs` is popped out (Line 14). The `computePrecondition()` procedure computes the weakest precondition of the instruction `inst`, given the postcondition `postCond` (Line 15).

```
procedure ComputeWP
  input
    stackTrace  the stack trace
    failCond    the crash condition
  output
    preCond     a weakest precondition for crash condition
    satModel    a satisfiable model from SMT solver
  global
    DEPTH_MAX   maximum invocation depth
    htPost2Dfs  a hash table mapping postconditions to working stacks
    summary     a table to hold summary information for each method
  begin
  01 cs = create the initial call stack from crash stack trace stackTrace
  02 method = stackTrace.exceptionMethod
  03 startLine = stackTrace.exceptionLine - 1
  04 return ComputeMethod(method, startLine, cs, 0, failCond)
  end

  procedure ComputeMethod(m, startLine, cs, depth, postCond)
  begin
  05 instStart ← getStartingInstruction(m, startLine)
  06 push m onto cs // update call stack
  07 stDfs ← htPost2Dfs.get(postCond) // try to resume
  08 if stDfs is null or empty then
  09   stDfs ← initialize stack and push (instStart, postCond)
  10   htPost2Dfs.add(postCond, stDfs)
  11 end if
  12
  13 while stDfs ≠ 0 do // depth-first traversal
  14   (inst, postCond) ← stDfs.pop()
  15   preCond ← computePrecondition(inst, postCond)
  16
  17     if inst is an invocation && depth < DEPTH_MAX then
  18       m' ← getTargetMethod(inst)
  19       preCond' ← map preCond to m'
  20       preCond'' ← summary.get(<m', preCond', depth+1>)
  21
  22       if no summary information matched then
  23         preCond'' ← ComputeMethod(m', -1, cs, depth+1, preCond')
  24         summary.put(<m', preCond', depth+1>, preCond'')
  25       end if
  26       preCond ← map preCond'' back to current method
  27     end if
  28
  29     if inst is not the entry node then
  30       preds ← get predecessors of inst
  31       pushPredsToStack(preds, stDfs) // push in each predecessor
  32     else if depth == 0 && cs is the outermost invocation in stackTrace then
  33       (bSatisfiable, satModel) ← perform SMT check on preCond
  34       if bSatisfiable is true then
  35         return (preCond, satModel)
  36       end if
  37     else // inside a specific invocation
  38       return (preCond, NULL) // continue depth-first traversal in caller
  39     end if
  40   end while
  41 return NULL // failed to find a satisfiable weakest precondition
  end
```

Algorithm 3. Weakest precondition computation algorithm for a crash condition

To support inter-procedural analysis, this approach handles invocation instructions in the following manner. When the current invocation depth depth is less than the maximum allowed depth `DEPTH_MAX`, the precondition of the invocation instruction is computed by recursively calling `ComputeMethod()` for the target method (Line 23). The postcondition is mapped into the scope of the target method before calling `ComputeMethod()` (Line 19). The computed precondition is mapped back into the scope of the current method after calling `ComputeMethod()` (Line 26).

During this computation, a table is maintained, `summary`, which holds all of the input-output summaries for each method. A summary in the table maps a method and its corresponding postcondition to a computed precondition. In this way, the summary information can be reuseed whenever a match is found in the table. By reusing the summary information, re-computing the precondition of the same method is avoided with the same postcondition for multiple times (lines 20-24).

To continue the depth first traversal, all predecessors of the current instruction are pushed into `stDfs` (lines 30-31). The backward computation continues until it reaches the entry of method `m`. If the current method is not in any invocation context (i.e., `depth` is 0) and method `m` is the outermost method in the crash stack trace, the algorithm employs an SMT solver to check the satisfiability of the computed precondition `preCond` (lines 32-33). Otherwise, the precondition is returned to the caller method, and the depth first computation goes on inside the caller's `ComputeMethod()` procedure (Line 38).

`computePrecondition()` computes the weakest precondition for the successful execution of each instruction with a given postcondition. Table 3 lists some of the mapping rules that

map postconditions to preconditions according to the instruction type. The φ operator in the table represents a replacement operation of variable instances. For example, φ(v1/v2) means that all occurrences of v2 in the postcondition are replaced with v1. The mapping rules are reused from SnuggleBug [37].

Table 3. Some mapping rules for `computePrecondition`

instruction type	mapping rules
v1 = v2	φ(v2/v1)
v1 = v2 op v3	φ(v2 op v3/v1)
v1 = v2.f1	φ(v2.f1/v1) v2 ≠ null
v1 = v2[i]	φ (v2[i]/v1) v2 ≠ null, i ≥ 0, i < v2.length
v1[i] = v2	φ(v2/v1[i]) v1 ≠ null, i ≥ 0, i < v2.length
v1 = new T	φ(instanceof(T)/v1)
v1 = v2.foo()	φ(ret of v2.foo()/v1) v2 ≠ null
return v1	φ(v1/ret of current method)

5.5.3.2. Object mutation based on precondition

This section presents the mutation approach on the top of the weakest precondition computation. Object mutation is based on the satisfiable model returned by `ComputeWP()` in Algorithm 3. The mutated objects are used to reproduce the original crash.

Algorithm 4 presents an overview of the mutation approach. Given the crash stack trace, `stackTrace`, and the set of objects to mutate, this procedure outputs a set of mutated objects that fully or partially satisfy the weakest precondition to reproduce the original crash.

```
procedure MutateObjects
input
  stackTrace   the stack trace
  objects      objects to mutate
output
  objects      objects mutated
begin
01 if stackTrace.exceptionType == explicit 'throw' then
02   failCond = {TRUE}
03 else if stackTrace.exceptionType==NullPointerException then
04   listRef ← getObjectRefernces(stackTrace.exceptionLine)
05   foreach ref in listRef do
06     failCond = failCond or {ref == NULL}
07   end foreach
08 end if
09
10 (preCond, satModel) = ComputeWP(stackTrace, failCond)
11 if satModel != NULL then
12   // set member fields according to model
13   setValues(objects, satModel)
14   return objects
15 else
16   return NULL
17 end if
end
```

Algorithm 4. Object-mutation algorithm based on precondition

Before computing the weakest precondition with the `ComputeWP()` procedure, first generate the crash condition. The crash condition is the precondition that should be satisfied when executing the crashed line to reproduce the original crash. Different exception types have different crash conditions. For a user defined exception thrown explicitly, the crash condition is simply a `TRUE` predicate (lines 01-02). This means the exception will be thrown whenever the throw instruction is executed. For a `NullPointerException`, its crash condition is con-

structed differently (lines 03-07). To reproduce a `NullPointerException`, a null de-reference is required for the crashed line.

After generating the crash condition, now compute the weakest precondition to reproduce the original crash by `ComputeWP()` (Line 10). If the weakest precondition exists, obtain a satisfiable model, `satModel` by `ComputeWP()`. Consequently, the input objects can be mutated based on the `satModel` (lines 11-13). In the `setValues()` procedure, mutate objects by assigning new values indicated in the satisfiable model to their accessible fields.

To avoid violating class invariants [31], the algorithm only mutates publicly accessible object member fields. With this restriction, it cannot guarantee the mutated objects satisfy all of the predicates in the computed weakest precondition. As a result, the executions with mutated objects could still fail to reproduce the original crash. However, even with limited mutation and satisfying only partial of the weakest precondition predicates, the mutated objects are still useful to reproduce additional crashes. This is later shown in our evaluation section.

5.5.3.3. Imprecision in object mutation

Due to the lack of information in crash stack traces, sometimes it is not possible to distinguish which object is truly responsible for a crash. For example, a null de-reference in line '`int total = o1.count() + o2.count();`' could have been caused by either `o1` or `o2`. However, the crash stack trace only indicates the crashed line number and does not indicate the responsible object reference. In such cases, the mutation approach might not be precise.

5.5.3.4. Loop Handling

Due to the undecidability problem in software verification [24], the inter-procedural, path-sensitive, and context-sensitive backward analysis could run forever. This problem is mainly caused by loops and recursive calls in crashed code. In these cases, the number of elements in `stDfs` continues to increase as the backward computation goes on. To address this issue, an option is introduced, specified by users, to limit the maximum number of times a loop can be computed. Hence, the algorithm, given this value, could enumerate every execution path in a finite number of steps and guarantee return. Recursive calls are handled similarly by setting a maximum invocation depth before the analysis starts. However, this kind of approximations might cause CAPTIG to fail to find the weakest precondition, even if one actually exists. In this experiment, both option values are set as two.

CHAPTER 6. IMPLEMENTATION

This chapter describes implementation of our approach. First, individual implementations of each approach are explained, and then the integerated implementation of the individual implmentations is described.

6.1. Individual Implementation

Previously, the approaches have been implemented using four independent tools. The following is a brief introduction for each tool.

PERT – This tool name stands for "Practical Extensions of a Randomized Testing Tool." This tool implements the four techniques used here, namely, simplified distance-based input selection, type-based input selection, open-access method selection, and array generation testing approaches. These techniques were incorporated to a state-of-the-art random testing tool, Randoop.

GenRed – This tool name stands for "A Tool for Generating and Reducing Object-Oriented Test Cases." This tool implements two techniques that achieve given code coverage more quickly (i.e., less time and number of tests). On-demand input creation actively creates necessary input objects, although there are no available objects, and coverage-based method selection prioritizes methods to increase the code coverage rate. GenRed also implements test case reduction techniques, such as coverage-based reduction and sequence-based reduction that reduce redundant testing. Finally, GenRed integrates two separated processes—the generation and reduction—of test cases.

OCAT – This tool implements capture and generation of objects for Java. OCAT stands for "Object Capture based Automated Testing." The tool conducts bytecode instrumentation of the system under monitoring and then executes the instrumented system instead of the original one with the given program executions. During these executions, the instrumented system captures encountered objects and serializes them into a file. Then, OCAT generates objects by using the captured objects; these generated method-sequences derive objects from the captured objects to achieve high code coverage.

STAR – This tool implements the technique used in this study, which reproduces crashes using information collected from post-failure-process approaches. STAR stands for "Stack Trace-based Automated Reproduction framework." As inputs, STAR only takes a crash stack trace and the location of classes (i.e., class paths) that includes crashed and related methods. Then, STAR conducts stack track, the test case template generation and crash input generation.

6.2. CAPTIG Implementation

These four tools are combined into one—CAPTIG. Next, we briefly describe the implementation of each phase in our approaches for CAPTIG.

Enhanced Method-sequence Generation

We have implemented method-sequence generation techniques on top of Randoop. The implemented techniques include simplified distance-based input selection, input on-demand creation, array input generation, type-based input selection, coverage-based method selection, open-access method selection, coverage-based reduction, and sequence-based reduction. Pre-

viously, these techniques were separately implemented into two different tools—PERT and GenRed. Both GenRed and PERT were implemented on top of Randoop.

Object Capturing

Instrumentation techniques are widely used to capture objects and infer their associated invariants [22][45]. Similarly, CAPTIG uses an instrumentation framework called ASM [5] to insert object-capturing code. For storing objects, CAPTIG uses the XStream framework [86], which serializes objects into XML files. XStream can serialize objects that do not implement the `java.io.Serializable` interface; whereas, the Java serialization technique cannot serialize objects that do not implement that interface. Figure 22 shows an example of serialized objects in a form of XML.

Method-based Object Mutation

The captured objects are de-serialized for Randoop, a tool that implements Randoop. The de-serialized objects are used as seeds for Randoop to generate more objects. To measure branch coverage of tests generated by Randoop with seeded captured objects, Cobertura [9] is used. Cobertura instruments the target Java Bytecode and generates a coverage report after executing tests.

Precondition-based Object Mutation

To compute the weakest preconditions for the crashes, CAPTIG first translates the related program code from Java bytecode into static single assignment (SSA) form [42] using WALA APIs [15]. It then applies the inter-procedural backward computation algorithm to compute the weakest preconditions. CAPTIG employs a satisfiability modulo theories solver, Yices [43], to check for the satisfiability of the computed preconditions. Finally, CAPTIG

mutates objects by using Java reflection APIs [16], based on models from Yices. The generated test cases are finally converted to JUnit [26] test cases that reproduce a crash.

Stack Trace Processing

We parse a crash stack trace to extract necessary information. To identify receiver and argument classes, and its compatible classes, we use a program analysis library, WALA [15].

```
<org.apache.commons.collections.FastTreeMap serialization="custom">
    <unserializable-parents/>
    <tree-map>
      <default/>
      <int>0 </int>
      <string>first</string>
      <string>First Item</string>
      <string>second</string>
      <string>Second Item</string>
    </tree-map>
  <org.apache.commons.collections.FastTreeMap>
    <default>
      <fast>true</fast>
      <map>
        <no-comparator/>
        <entry>
          <string>first</string>
          <string>First Item</string>
        </entry>
        <entry>
          <string>second</string>
          <string>Second Item</string>
        </entry>
      </map>
    </default>
  </org.apache.commons.collections.FastTreeMap>
</org.apache.commons.collections.FastTreeMap>
```

Figure 22. An example of a serialized object

CHAPTER 7. EXPERIMENTAL EVALUATION

In this section, some results are presented of the work completed thus far in pursuance of the research objectives. This section proceeds with the following research questions.

Research questions for enhanced method-sequence generation:

RQ1. How much can new method-sequence selection algorithms improve code coverage?

RQ2. How many test inputs can CAPTIG reduce?

RQ3. How much time can CAPTIG reduce to execute test cases?

Research questions for object-capture-based input generation:

RQ4. How much can CAPTIG improve code coverage through captured objects?

RQ5. How much can mutated objects further improve code coverage?

Research questions for crash reproduction:

RQ6. How many crashes can CAPTIG reproduce, based on crash stack traces?

RQ7. Are generated test cases that reproduce crashes helpful for debugging?

7.1. Enhanced Method-sequence Generation

This section reports the empirical evaluation on the performance and usefulness of the enhanced method-sequence generation approach used in this research. The goal of this experiment is to compare code coverage of this approach to the current state-of-the-art approach, Randoop, in terms of time cost (generation time of test inputs) and the number of test inputs. In addition, the comparison results are shown among all the individual approaches. Note, the branch coverage was used for code coverage measurement.

7.1.1. Evaluation Setup

The subjects were selected from well-known open-source systems, ISSTA Containers, Java Collections, ASM, Apache Commons Collections, and Apache Ant. Table 4 presents information about the subjects.

ISSTA Containers has container classes (`TreeMap`, `BinTree`, `FibHeap`, and `BinomialHeap`) used in Visser *et al.*'s paper [86]. 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. ASM [5] is a Java bytecode manipulation and analysis framework. ASM can modify existing classes or dynamically generate classes, directly in binary form. Apache Commons Collections [1] provides new interfaces, implementations, and utilities, such as buffer, queue, map, and bag. Container classes are commonly used to evaluate a generation tool, since it is difficult to make a desirable container class instance (e.g., a long list with various values) automatically. Apache Ant [4] is a Java-based build tool similar to the tool called MAKE in C/C++.

Each approach was evaluated by measuring branch coverage criteria. A machine was used with Windows Vista, Intel Pentium 2.2Ghz Dual Core, 3GB RAM.

Table 4. Subject open-source systems for enhanced method sequence generation

systems	classes	methods	KLOC	description
ISSTA Containers (ISSTA)	5	7	2	toy container classes
Java Collections 1.6 (JDK)	45	634	22	Java's collection library
ASM 3.1	111	1353	40	a Java bytecode manipulation framework
Apache Commons Collections 3.2 (ACC)	273	2468	63	an extended collection library
Apache Ant 1.7.1 (ANT)	769	3001	209	a Java-based build tool

This section addressed the following research questions:

RQ1. How much can new method-sequence selection algorithms improve code coverage?

RQ2. How many test inputs can CAPTIG reduce?

RQ3. How much time can CAPTIG reduce to execute test cases?

7.1.2. Overall Results

Table 5 shows code coverage of the original Randoop and these approaches. ISSTA is a relatively small system that has only 71 methods, and state-of-the-art Randoop achieved over 86.6% of the code coverage. Based upon Randoop's results, it is believed from the borderline of code coverage that automated tools can achieve has been nearly reached. Contrary to expectations, this approach achieved 91.6% of code coverage, which is 5% more than Randoop. This approach improves code coverage by 12-15% more code coverage than Randoop for ASM and ANT. JDK shows the best results—raised by 22.8% of code coverage—by this approach compared to Randoop's code coverage, in spite of the large number of methods and code size.

Table 5. Overall results of enhanced method-sequence generation compared to Randoop based on code coverage improvement

systems	Randoop	CAPTIG	improvement
ISSTA	86.6%	91.6%	5.0%
JDK	43.2%	66.0%	22.8%
ASM	22.4%	37.9%	15.5%
ANT	41.9%	54.5%	12.5%

Table 8 represents the number of test inputs and generation time between Randoop and the approaches used in this study to reach certain code coverage (coverage goal[3]). To achieve 42% of code coverage for JDK, Randoop generated 10,700 test cases in 485 seconds; meanwhile, this approach generated 1170 test cases in 6 seconds. CAPTIG achieves the coverage goal more quickly with fewer test cases. Throughout all of these systems, almost one-half of the test time and test case size are reduced to achieve certain code coverage.

Table 6. Overall results of enhanced method-sequence generation compared to Randoop based on certain coverage goal

systems	coverage goal	# test inputs (Randoop:CAPTIG)	ratio	time (sec) (Randoop:CAPTIG)	ratio
ISSTA	82%	14209:8347	1:0.59	340:191	1:0.56
JDK	42%	10700:1170	1:0.11	485:6	1:0.01
ASM	20%	20469:2867	1:0.15	471:37	1:0.08
ANT	40%	19466:8145	1:0.42	712:243	1:0.34

[3] The coverage goal was determined by subtracting 1-4% from the highest coverage of Randoop for each subject system with 2500-3000 seconds of generation time.

Figures 23-Figure 26 show code coverage for ISSTA, ANT, ACC, and JDK with respect to the number of test inputs and the amount of time. This study's combined approach shows significant improvement. Next, each individual approach will be discussed in the following sections.

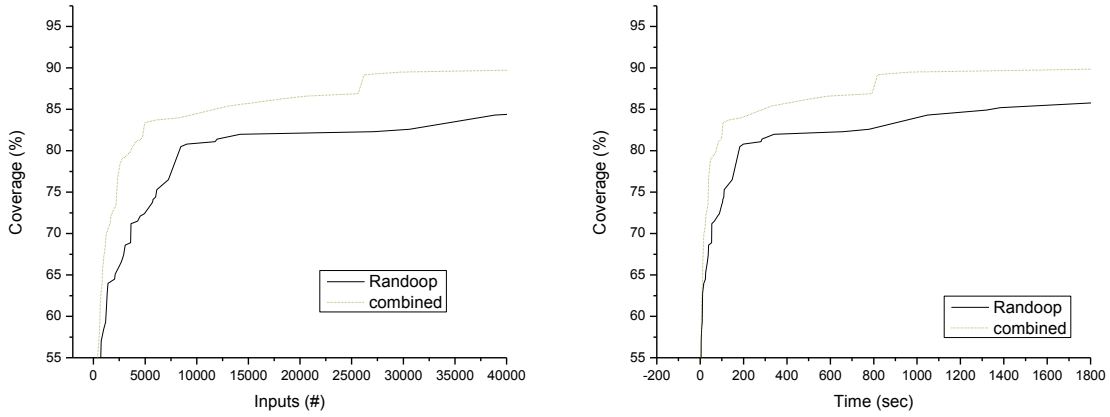

Figure 23. Overall results of enhance method-sequence generation for ISSTA

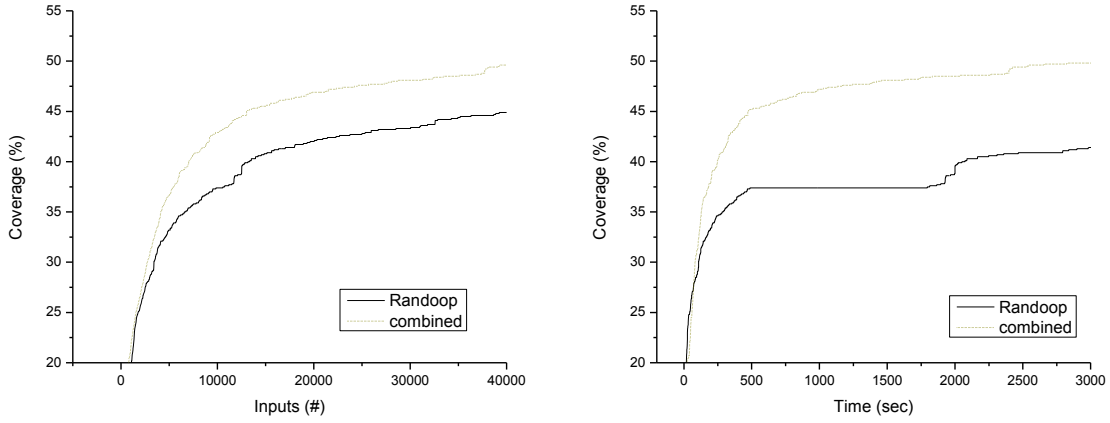

Figure 24. Overall results of enhance method-sequence generation for ANT

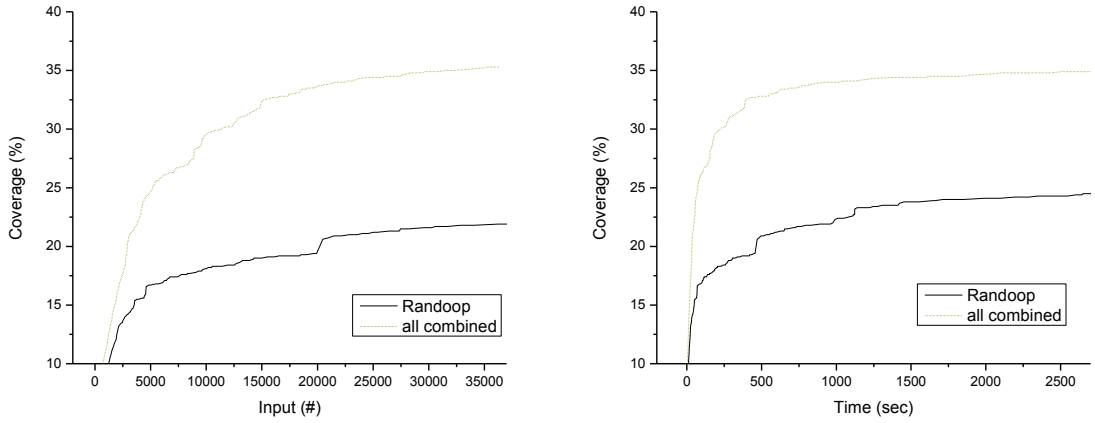

Figure 25. Overall results of enhance method-sequence generation for ASM

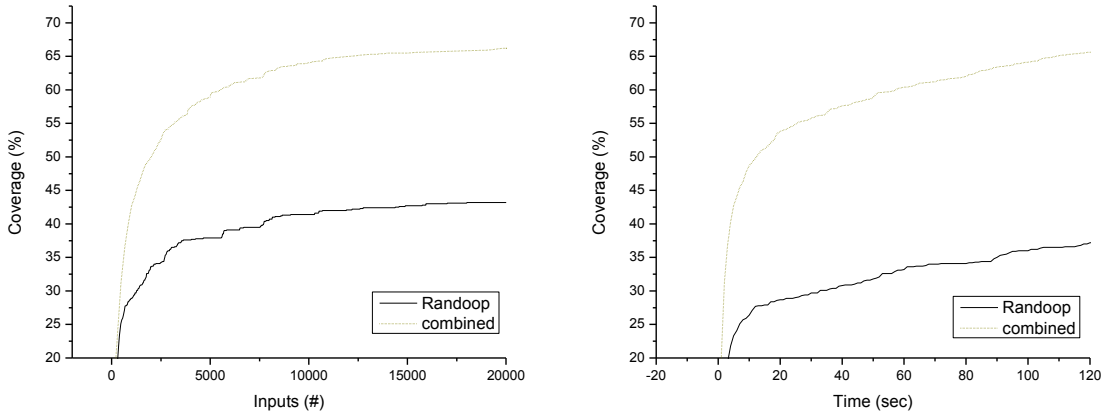

Figure 26. Overall results of enhance method-sequence generation for JDK

7.1.3. Simplified Distance-based Input Selection

The simplified distance-based approach increases code coverage through all systems with respect to the number of test inputs. However, it is very slow to generate test cases, as Ciupa *et al.* mentioned in their ARTOO paper [35]. In Figures 27-29, ARTOO's distance-based ap-

proach generally shows better code coverage at the beginning, but the Randoop sometimes surpasses it. This is because ARTOO's distance-based approach is too meticulous to select an input, and object calculation is computationally expensive.

This study's simplified distance-based input selection reduces the computation time, and increases code coverage in terms of both generation time and the number of test cases. Figures 27-29 show the simplified distance-based input selection overcomes Randoop and ARTOO's distance-based approach constantly through ANT, ISSTA and JDK.

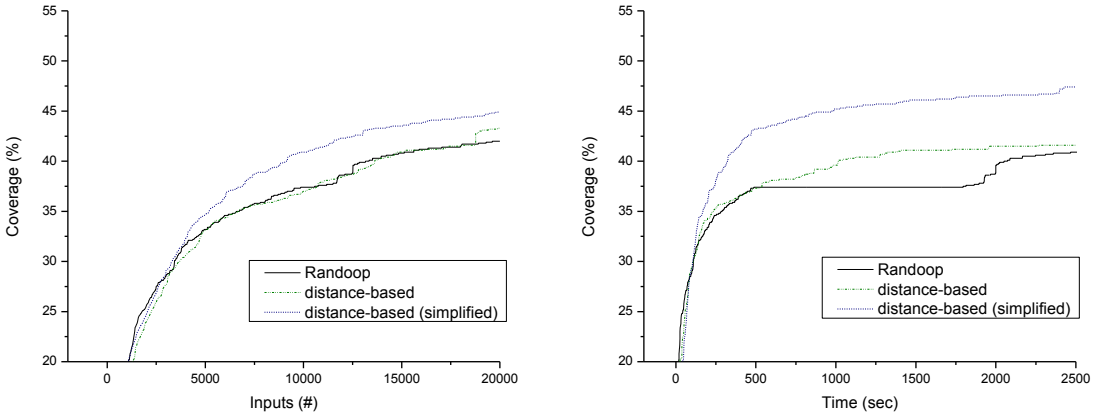

Figure 27. Results of distance-based input selection for ANT

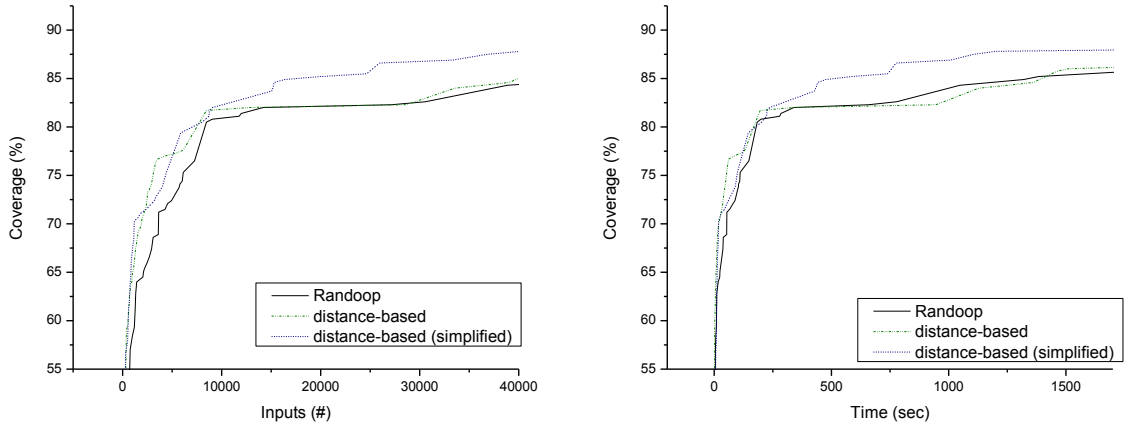

Figure 28. Results of distance-based input selection for ISSTA

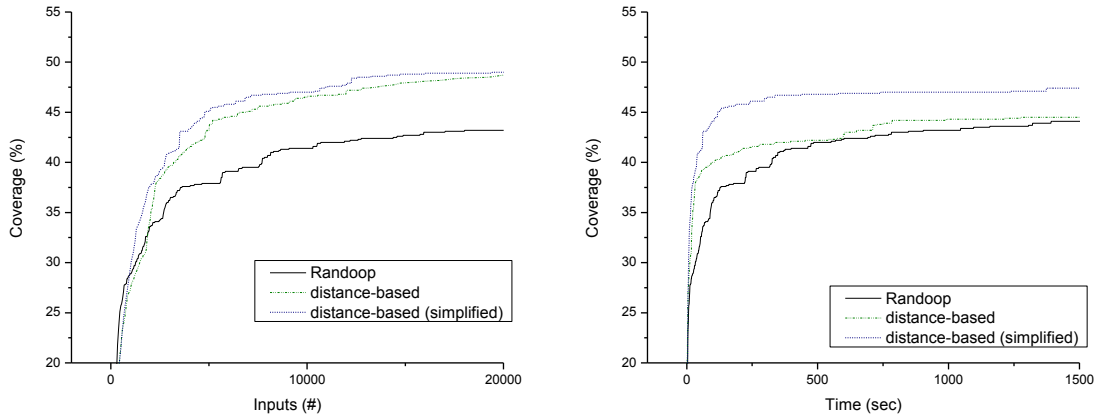

Figure 29. Results of distance-based input selection for JDK

Table 7 lists the results of direct comparison between ARTOO and this study. Table 7 shows the number of test cases and time to reach certain code coverage measured by both distance-based approaches. The results show that simplified object-distance calculation reduces computation time and keeps efficiency of their guided input selection strategy.

Table 7. Results of distance-based input selection compared to ARTOO.

system	coverage goal	# test inputs (ARTOO:simplifed)	ratio	generation time (sec) (ARTOO:simplified)	ratio
ISSTA	82%	12865:9088	1.4:1	338:237	1.43:1
JDK	42%	4340:3518	1.2:1	7997:1977	4.0:1
ASM	20%	13673:13802	0.99:1	4867:479	10.1:1

7.1.4. On-demand Input Creation

The array generation approach shows approximately 10% of code coverage increments for JDK and 5-7% for ASM. For ISSTA Containers, the array generation approach shows 1-2% lower coverage than Randoop's code coverage because methods of five ISSTA do not take an array input. Figures 30 and 31 illustrate the array generation approach has the improvement for ASM and JDK.

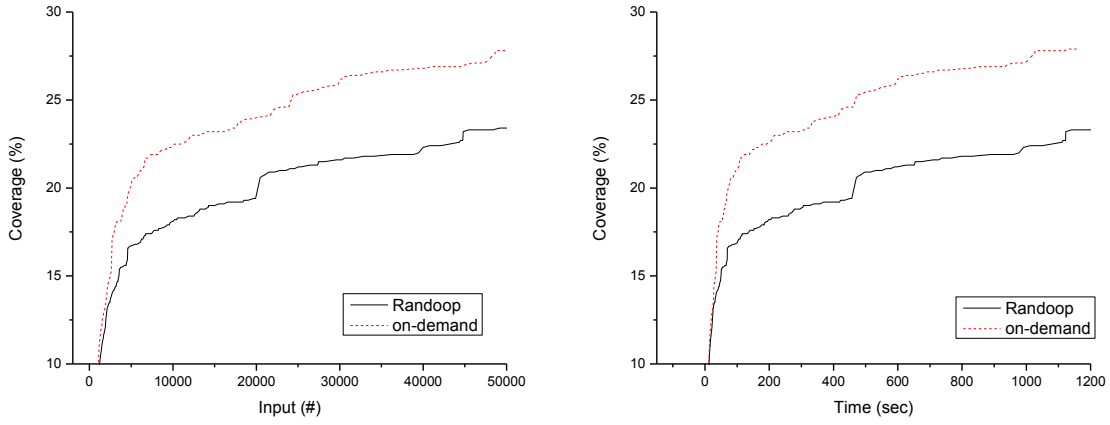

Figure 30. Results of on-demand input creation for ASM

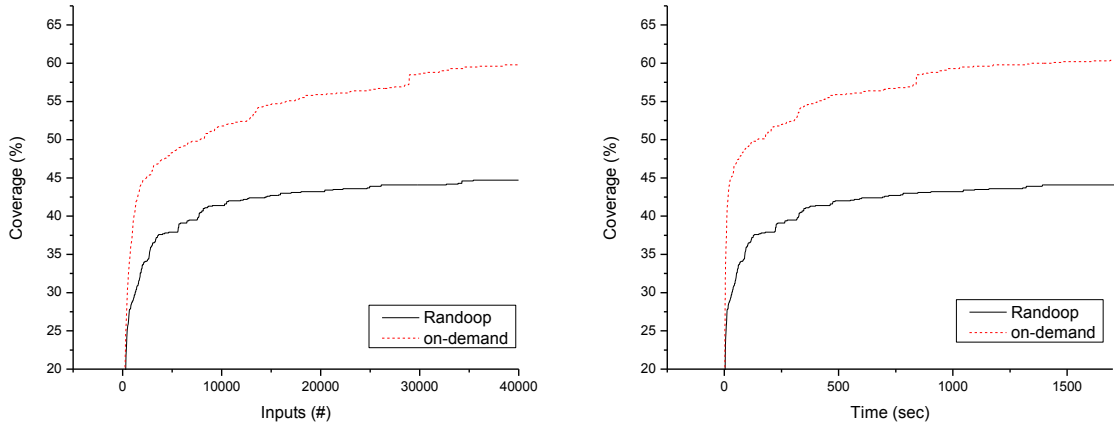

Figure 31. Results of on-demand input creation for JDK

7.1.5. Type-based Input Selection

The type-based input selection constantly increases 24% average code coverage for all of the systems. According to Figures 32 and 33, this study's type-based input selection approach is always better than Randoop. The results show 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, an improvement of testability is seen.

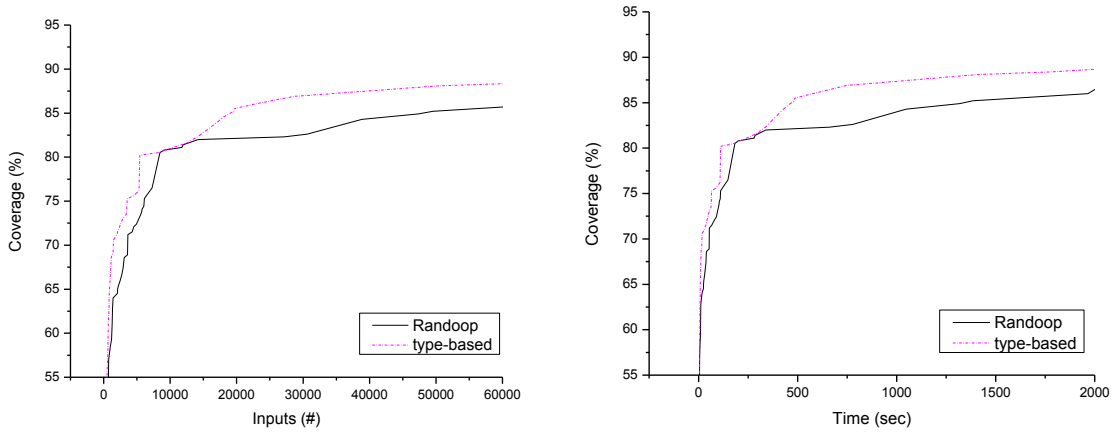

Figure 32. Results of type-based input selection for ISSTA

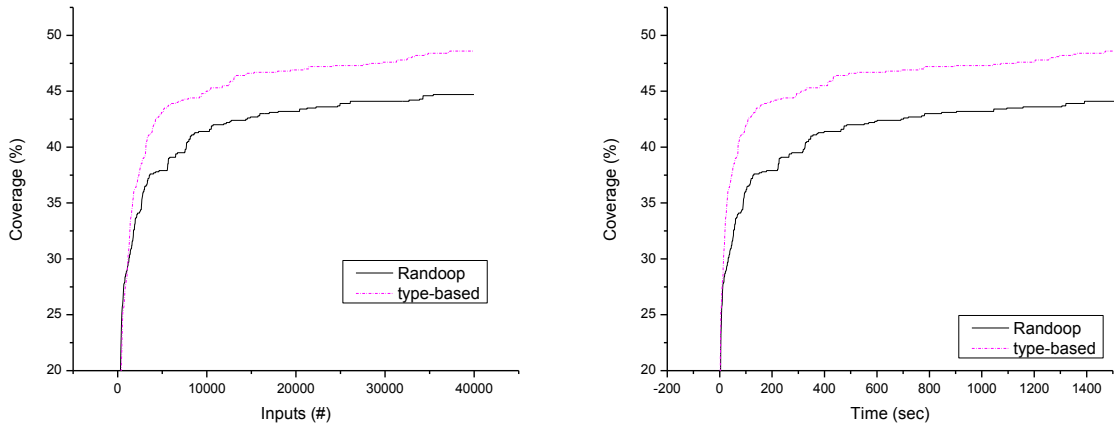

Figure 33. Results of type-based input selection for JDK

7.1.6. Coverage-based Method Selection

As seen in Table 8, to achieve 40% of coverage for JDK, Randoop generates test cases for 471 seconds; meanwhile this study's coverage-based method selection approach generates better results in only 13 seconds. Similarly, for ASM, Randoop and this study take 1092

and 90 seconds, respectively, to achieve 25% of goal branch coverage. For ACC, Randoop and this study take 1092 and 90 seconds, respectively, to achieve 40% of goal branch coverage. To achieve 45% of branch coverage for ANT, Randoop takes 305 seconds and this study takes 123 seconds. The results show this study achieved the coverage goal more quickly. Throughout all the subject systems, the test time is significantly reduced for achieving certain branch coverage.

Table 8. Results of distance-based input selection compared to Randoop in terms of coverage goal

system	coverage goal	time (sec) (Randoop:cov-based)	ratio
JDK	40%	471 : 13	36.2 : 1
ASM	25%	1092 : 90	12.1 : 1
ACC	40%	3539 : 235	15.0 : 1
ANT	45%	305 : 123	2.5 : 1

Figures 34-36 show branch coverage for systems with respect to generation time. Through these systems, this study's coverage-based method selection constantly outperforms Randoop in terms of code coverage and generation time. Therefore, the on-demand input creation and coverage-based method selection techniques show better performance than Randoop.

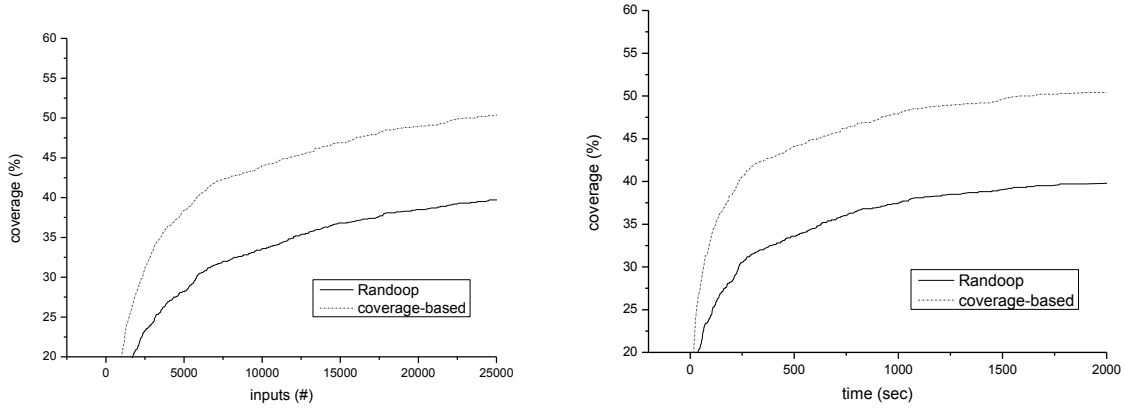

Figure 34. Results of coverage-based method selection for ACC

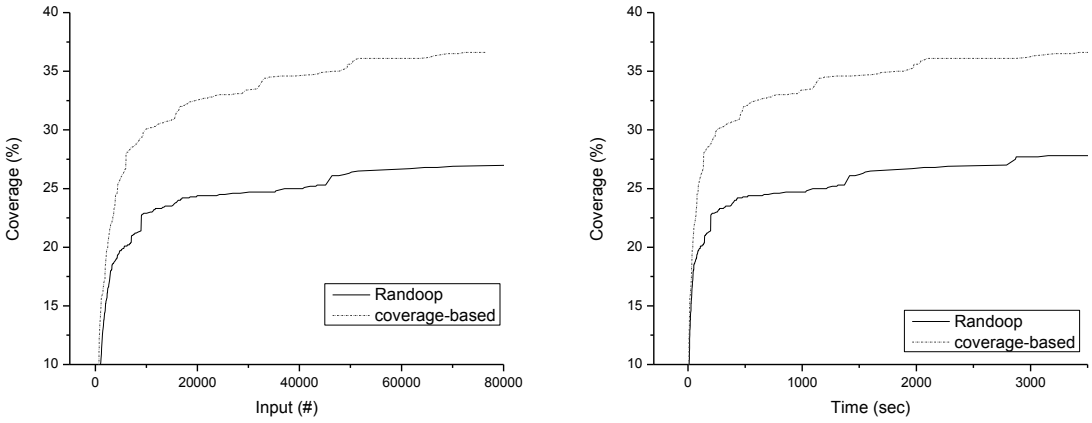

Figure 35. Results of coverage-based method selection for ASM

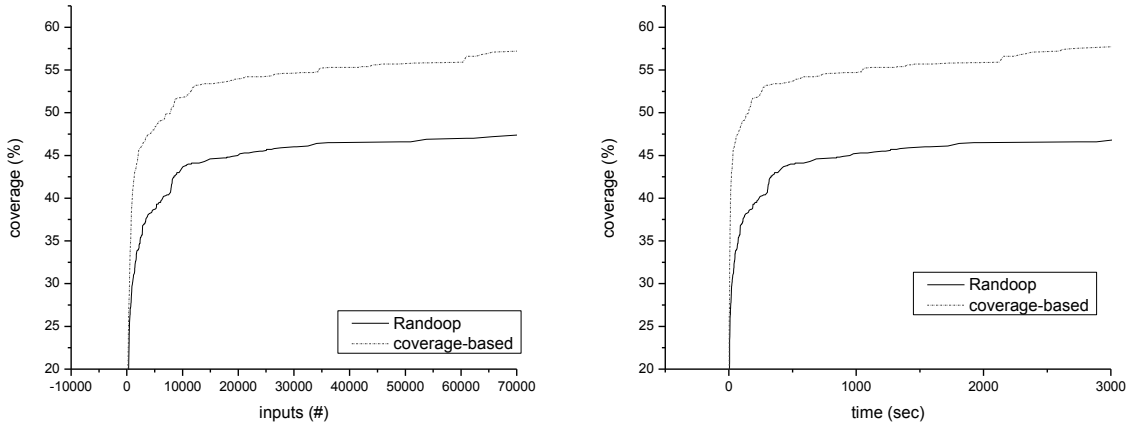

Figure 36. Results of coverage-based method selection for JDK

Table 9 compares code coverage of test inputs generated by this study and Randoop in 2500-3000 seconds. The results for Java Collections show 13.7% of code coverage increments by GENRED compared to Randoop's coverage. The test results for Apache Commons-Collections also show 12.2% of code coverage increments. Similarly, the test results for ASM and Ant show 9.2% and 5.3% of code coverage increments, respectively.

Table 9. Comparison results of coverage-based method selection in terms of coverage goal

system	Randoop	cov-based	improved
JDK	43.4%	57.1%	13.7%
ASM	26.9%	36.1%	9.2%
ACC	40.3%	52.5%	12.2%
ANT	48.6%	53.9%	5.3%

7.1.7. Open-access Method Selection

The open-access method selection gives 7-9% code coverage improvement on average, since it provides a way to test non-public methods without changing and analyzing source

codes. The open-access approach shows the best coverage besides the all combined approach for ASM system with respect to both time and the number of test inputs.

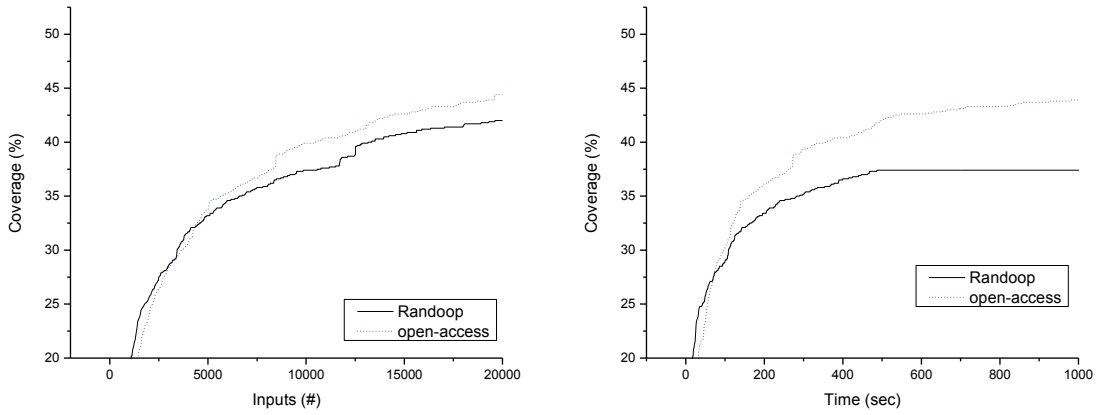

Figure 37. Open-access method selection results for ANT

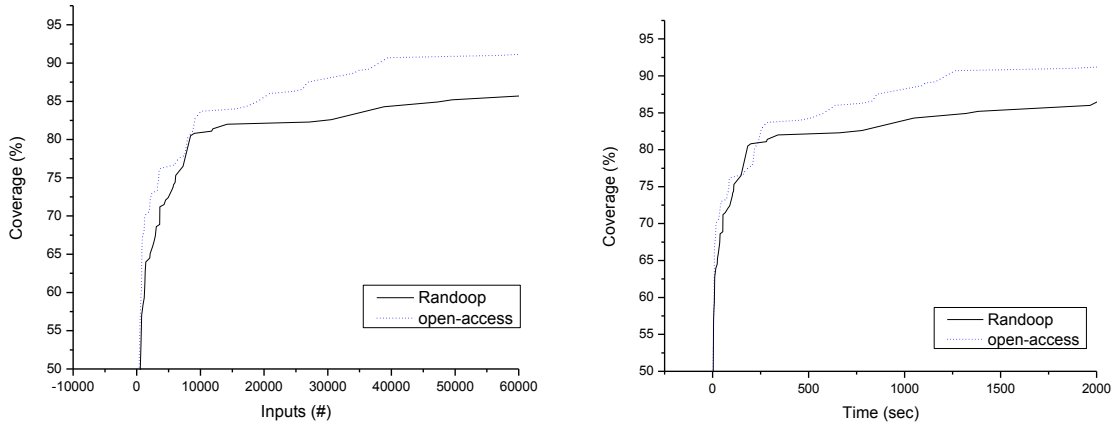

Figure 38. Open-access method selection results for ISSTA

7.1.8. Test Input Reduction

This study's method-sequence reduction approach was considered with and without test reduction techniques. Table 10 shows the overall results of the test-suite reduction. The per-

centages in Table 10 show an average of the results through four open-source systems. Sequence-based reduction removes 51.8% of the test cases and the testing time (including compile and execution time) without reducing coverage. Coverage-based reduction removes 98.4% of test cases and 89.4% of testing time. Both approaches eliminated 99.4% of the test cases and 94.5% of the testing time. According to the test case reduction rate, the sequence-based reduction technique is not effective compared to coverage-based reduction. However, the sequence-based reduction technique, because the technique does not need to check code coverage, is able to remove some test cases, which cannot be removed by the coverage-based reduction technique.

Table 10. Overall results of test-suite reduction

type	sequence-based	coverage-based	both
removed test cases	51.8%	98.4%	99.4%
reduced testing time	51.8%	89.4%	94.5%

Tables 11 and 12 show the results for each system. For Apache Common Collections, sequence-based reduction shows that 44.9% of the test cases remained as non-redundant test cases. Note, only 32.9% of the testing time was used. Coverage-based reduction reduces the size of test-suite and testing time significantly. For ASM, the coverage-based reduction technique needs only 469 test cases and 30 seconds to compile and run, and achieves 80% branch coverage. The combined approach (denoted as "both" in Tables 11 and 12) also showed good results for ASM. This result has only 124 test cases and 7 seconds testing time with 80% branch coverage.

Table 11. Comparison results between Randoop and CAPTIG in terms of the number of generated test cases

system	the number of test cases			
	base	seq-based	cov-based	both
JDK	46,964	24,094 (51.3%)	440 (0.9%)	181 (0.4%)
ASM	73,823	33,601 (45.5%)	469 (0.6%)	124 (0.2%)
ACC	27,106	12,158 (44.9%)	1,032 (3.8%)	413 (1.5%)
ANT	35,522	17,185 (48.4%)	822 (2.3%)	347 (1.0%)

Table 12. Comparison results between Randoop and CAPTIG in terms of testing time

system	testing time (seconds)			
	base	seq-based	cov-based	both
JDK	412	156 (37.9%)	7 (1.7%)	13 (3.2%)
ASM	107	73 (68.2%)	30 (28.0%)	7 (6.5%)
ACC	605	197 (32.6%)	43 (7.1%)	54 (8.9%)
ANT	176	99 (56.3%)	26 (14.8%)	14 (8.0%)

Figure 39 shows the testing time for reduced test cases for each system. By combining both reduction approaches, the testing time of the reduced test-suite takes less than 10% from the original test-suite.

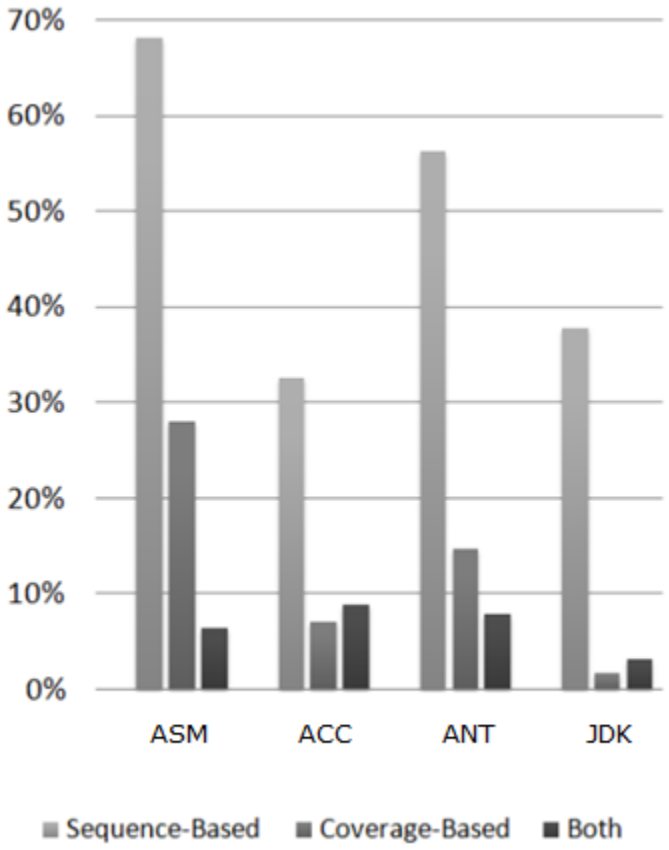

Figure 39. Testing time with reduced test cases

7.2. Captured and Mutated Objects

This study's approach was evaluated by comparing code coverage of the capture-based input generation approach and a state-of-the-art tool, Randoop. The evaluation was designed to address the following research questions:

RQ4. How much can CAPTIG improve code coverage through captured objects?

RQ5. How much can mutated objects further improve code coverage?

7.2.1. Evaluation Setup

This study's evaluation was conducted on a machine with Linux, Intel Xeon 3.00GHz, and 16GB memory. A widely used Java code coverage analysis tool was adopted, Cobertura [9], to measure branch coverage.

Three open-source projects, Apache Commons Collections (ACC), Apache XML Security (AXS), and JSAP, were used as test subjects. Table 13 shows the subjects' information. ACC [1] is an extended collection library that provides new utilities, such as buffer, queue, and map. AXS [3] is a library implementing the XML Digital Signature Specification and XML Encryption Specification. JSAP [17] is a command-line argument parser.

Table 13. Subject open-source systems for capture-based input generation

system	time (sec)	# objects	size (MB)
Apache Commons Collections 3.2 (ACC)	273	2522	63
Apache XML Security 1.0 (AXS)	179	1185	40
JSAP 2.1 (JSAP)	77	462	11

We apply our approach to instrument each subject system to enable object capturing. Then, the system tests were executed to include each subject system to capture objects. Table 14 shows the statistics of the object-capturing phase, including time spent in capturing objects, the number of captured objects, and the size of the serialized file. For ACC, this study's approach captured and serialized 14,999 objects in 311 seconds, and the serialized file is 7.6MB. For AXS, this approach took 366 seconds to capture 11,390 serialized objects in 410MB. For JSAP, this approach captured and serialized 292 objects in 1 second. The serialized file was

15KB. The branch coverage of the captured executions for each subject was 52.7% (ACC), 36.0% (AXS), and 32.9% (JSAP).

Table 14. Statistics of captured objects: showing time to capture, number, serialized file size of captured objects, and branch coverage of captured executions.

system	time (sec)	# objects	size	coverage
ACC	311	14999	7.6MB	52.7%
AXS	366	11390	410MB	36.0%
JSAP	1	292	15KB	32.9%

The captured objects and the serialized file size depend on the subject system and the executed tests. For example, the AXS's serialized file is relatively big, since AXS loads and maintains XML file contents, and the contents are stored in objects. However, the serialized files are manageable, even if the entire suite of tests is executed for a relatively large software system (i.e., a system that has over 1,000 methods), such as ACC and AXS. In addition, it is possible to limit the number of captured objects and size of the serialized files.

7.2.2. Captured Objects

RQ4. How much can CAPTIG improve code coverage through captured objects?

This section shows coverage improvement by leveraging captured objects through CAPTIG to assist Randoop (denoted as CAP + MET). First, as described in Section 4.1, the objects were captured by executing tests provided with each subject system. The number of captured objects is shown in Table 14. Then, the captured objects are fed as test inputs for Randoop and the branch coverage is measured. Similarly, Randoop is run alone on subject systems and the branch coverage is measure. Each tool is run until the tool's coverage is sat-

urated, which means either one of the tools' coverage levels off without much further increase, or one of the tools runs out of memory and cannot continue to run. Since the number and quality of captured objects depend upon the tests initially provided with the subject system, the coverage of such tests is measured (denoted as a captured execution).

Figure 40 shows the branch coverage of three subjects with two approaches, Randoop and CAPTIG (CAP + MET), based on the number of generated tests. As a baseline, the coverage of the captured executions is shown. The x-axis indicates the number of tests generated by each approach. As the number of tests grows, generally the branch coverage also increases.

As Figure 40 indicates, there is a substantial coverage difference between CAPTIG and Randoop. For ACC, after 46,000 tests are generated, CAPTIG achieves 64.2% branch coverage, 19.0% improvement from Randoop's achieved coverage, 45.2%. Similarly, for AXS, CAPTIG improves 28.5% branch coverage in comparison to Randoop. For JSAP, the coverage improvement is 17.3% over Randoop.

These encouraging results suggest that captured objects play an important role to improve code coverage by assisting an existing testing technique. It was observed that the captured objects are influenced by the original set of tests provided with the subject system. However, the CAPTIG's achieved coverage is much higher than the coverage achieve by simply executing these original tests shown in the last column of Table 14. And, captured objects can lead a random generation technique to cover more branches.

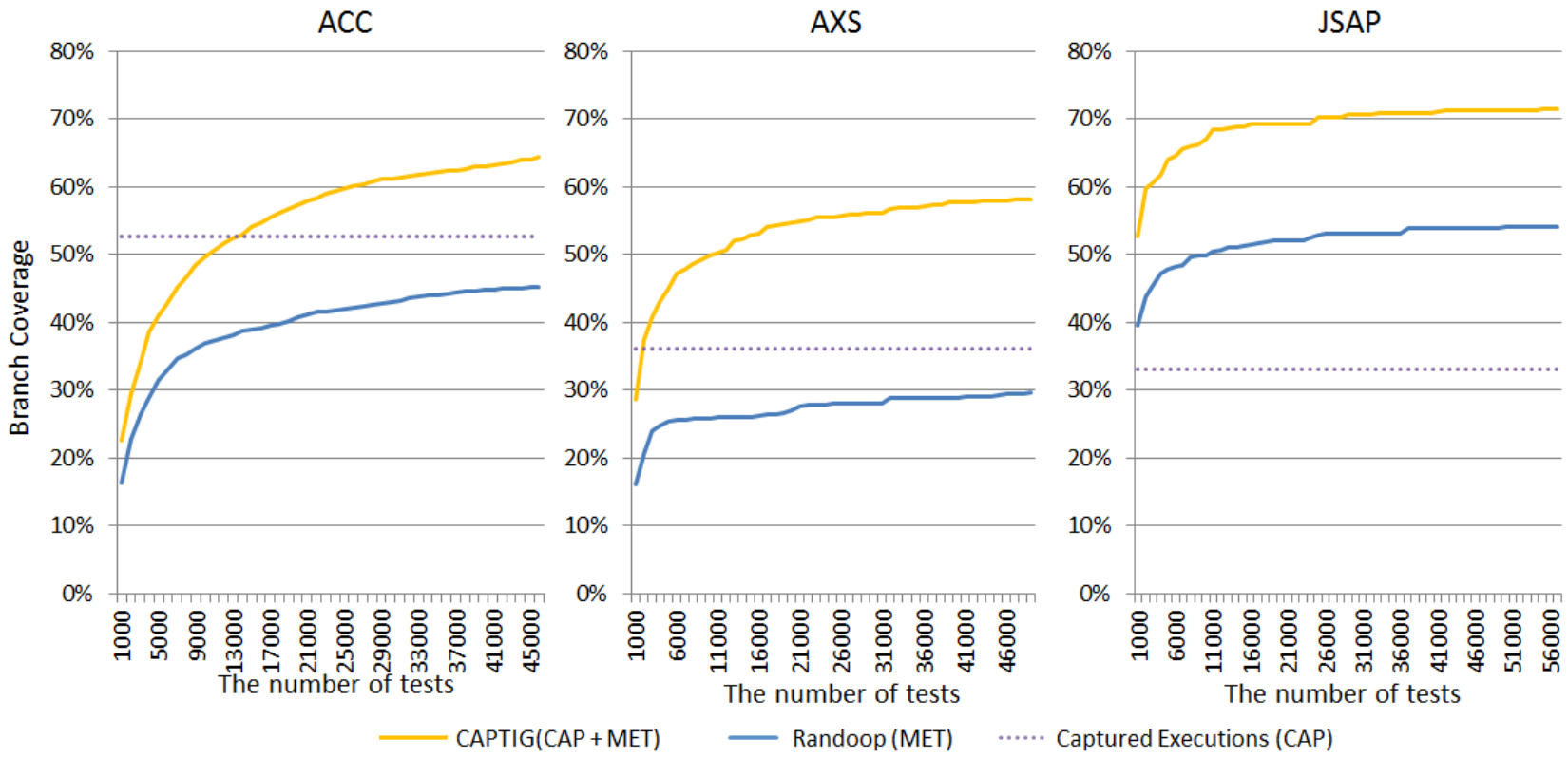

Figure 40. Branch coverage achieved by CAPTIG, Randoop, and Captured Executions

7.2.1. Mutated Objects

RQ5. How much can mutated objects further improve code coverage?

Next, the PRE phase is evaluated by using static analysis and an SMT solver. Although the CAP and MET phases increase the code coverage substantially, still there are not-covered branches. For example, for JSAP, 28.6% of the branches are not covered after the CAP and MET phases.

We further apply the PRE phase described in Section 4.4 and evaluate whether PRE improves the coverage further, although the current mechanism of mutating objects can handle only a limited set of situations. Note, that we do not modify private fields of objects in our evaluation. The coverage distribution bars in Figure 41 present branch coverage improvement contributed by Randoop, CAPTIG (CAP + MET), and CAPTIG (PRE). For ACC, 45.2% are covered by Randoop, and CAPTIG (CAP + MET) further improves this coverage percentage by 19.0%. Finally, CAPTIG (PRE) increases an additional 5.4%. Similarly, after CAPTIG (CAP + MET) improves the coverage by 28.5% for AXS and 17.3% for JSAP, CAPTIG (PRE) further improves the coverage by 6.9% more for AXS and 6.2% for JSAP.

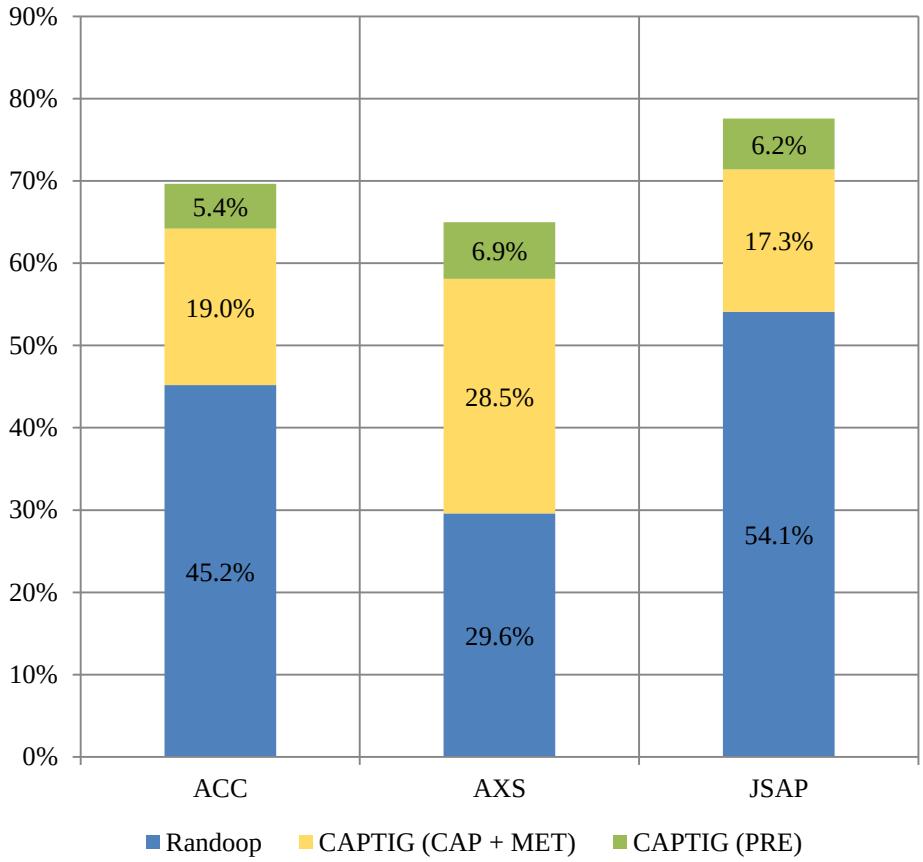

Figure 41. Branch coverage distributions of each approach with captured and mutated objects.

Such an improvement may seem marginal. However, since code coverage becomes saturated after more than 46,000 tests, further improvement would be difficult to obtain, in general. The PRE phase can still improve 6.2% of branch coverage on average, which is not trivial.

Overall, by combining random and systematic approaches, CAPTIG (CAP + MET + PRE) improves branch coverage on average by 27.8% (with a maximum increase of 35.4% for AXS) in comparison to the coverage by Randoop alone.

7.3. Crash Reproduction

This section presents this study's experimental setup and evaluation results of the crash reproduction approach. The evaluation was designed to address the following research questions:

RQ6. How many crashes can CAPTIG reproduce, based on crash stack traces?

RQ7. Are generated test cases that reproduce crashes helpful for debugging?

7.3.1. Evaluation Setup

Subjects

CAPTIG's crash reproduction feature is evaluated by using bug reports (from July 2003 to November 2009), which include stack traces from three open-source systems, AspectJ Development Tools (AJDT) [6], Apache Commons Collection (ACC) [1] and Apache Commons Math (ACM) [2].

AJDT is a tool that supports aspect-oriented software development, including an AspectJ compiler. Eight versions of AJDT are used (1.1.0, 1.1.1, 1.2.0, 1.2.1, 1.5.0, 1.5.2, 1.5.3, and 1.5.4). Particularly, the `org.aspectj` package was used for AJDT. ACC implements known data structures, such as buffer, queue, map, and bag. Two versions of ACC, versions 3.1 and 3.2, were used. The ACM is a library of self-contained mathematics and statistics components. Thus, ACM versions 1.2 and 2.0 were used for this study's evaluation.

Bug Report Collecting and Processing

Bug reports include metadata and textual contents to describe crashes. Some users include crash stack traces in their bug reports, and this study uses bug reports, which contain crash

stack traces for this experiment. From bug reports, a bug id, version number, and crash stack traces were extracted by using InfoZilla [27].

Some bug reports were excluded. First, we do not use bug reports with invalid crash traces. After extracting stack traces, method names, and their line numbers were obtained. Then, the methods were checked to determine if they exist and the line numbers were matched with the corresponding version of the system. We prune cases where users report an incorrect version or wrong stack trace information. In addition, all non-fixed bug reports were excluded. Even if they included valid stack traces, it could not be determined whether they are real bugs.

Table 15 shows the number of the bug reports[4] used in this study's experiments. Six hundred seventy-four bug reports were extracted, but only 184 bug reports included stack traces. After pruning the non-fixed bug reports and bug reports with invalid stack traces, 101 bug reports remained.

Table 15. Bug reports used in this experiment

system	# of bug reports	# of bug reports with stack traces	# of valid stack traces
AJDT	461	162	83
ACC	97	8	8
ACM	116	14	10
Total	674	184	101

[4] Bug reports for AJDT can be found in http://bugs.eclipse.org, and bug reports for both ACC and ACM can be found in https://issues.apache.org/.

Criteria of Reproducibility

To determine whether a crash is reproduced or not, the exception type and location of the original and reproduced crashes [22] were checked. If the reproduced exception type and the original type were identical, and both crashed in the same line, it was assumed the crash is successfully reproduced.

Object Capturing

This study's crash reproduction approach requires captured objects from normal program execution. To capture objects, subject systems were instrumented and executed system tests were provided by each subject. The instrumented systems capture objects and serialize them during execution of the system tests.

By executing instrumented systems with their system tests, 12,420 objects were captured, serialized, and zipped in 342MB on average for each version of AJDT, 61,408 objects were zipped in 104MB for ACC and 7,012 objects were zipped in 4MB for ACM. Often, executing instrumented systems incurs a huge overhead [22][69][76][80]. However, note, this capturing process is conducted once in-house. The deployed software does not incur any additional overhead.

7.3.2. Reproducibility

This section shows how many crashes are reproducible by this study's approach.

Table 16 shows the number of reproducible crashes from the feedback-directed, object-capture-based input generation techniques. The total number of collected stack traces is 101. The "feed" column shows the number of reproducible crashes using feedback-directed input generation technique. The column denoted as "met" represents the number of additional reproducible crash-

es by the MET technique. Similarly, “pre” presents the number of additional reproducible crashes by the PRE technique.

Table 16. Reproducible crashes of AJDT, ACC and ACM.

system	# of crashes	# of reproducible crashes			
		feed	met	pre	total
AJDT 1.1.x	21	0	0	+10	10
AJDT 1.2.x	16	2	+2	0	4
AJDT 1.5.x	46	9	+10	+5	24
ACC	8	1	+2	+2	5
ACM	10	1	0	+1	2
total	101	13	+14	+18	45
percent		12.9%	+13.9%	+17.8%	44.6%

For feedback-directed and object-capture-based input generation, three minutes were set as a time limit to generate each test. If these techniques could not generate reproducible test cases in 3 min, it was assumed they could not reproduce the corresponding crash.

The feedback-directed input generation technique reproduces 13 crashes for all subjects. The MET technique reproduces 14 more crashes from the feedback-directed technique. Eighteen more crashes are reproduced by the PRE technique. Overall, the result is very promising: our approach reproduces 45 (44.6%) crashes. We anticipated this study’s approach significantly reduces developers’ efforts on reproducing crashes, since 44.6% of the crashes can be reproduced automatically.

We also examine the bug fix time for these 44.6% reproducible bugs. Surprisingly, approximately 68% of these crashes took over a week and 46% took over a month to be resolved. The main reason of this long bug fix time is due to the challenges of reproduction [27]. Developers do not have enough resources to reproduce all bugs. Irreproducible bugs receive low fix priorities. It is anticipated that this study's crash reproduction approach helps developers reproduce crashes quickly and thus shorten the overall bug fix time.

7.3.3. Usability

In this section, the usefulness of the reproducible test cases for debugging is evaluated.

The source code (a) in Figure 42 shows a buggy `TreeList` method in ACC 3.1[5]. This method deletes one node from a tree. `TreeList` can crash with a null-pointer-exception when the iterator method `previous()` is called, because `remove(int)` incorrectly removes a node. More specifically, a tree misses a link of a left node after deletion. The link should be kept before deletion.

According to the bug report history, the bug was reproduced 5 months later. This bug could not be easily reproduced, and so was treated as non-severe. However, after a manually written reproducible test case was submitted, this bug was immediately resolved soon with the developer's comment, "As always, a good test case makes all the difference."

Comparably, this study's approach effectively generates a test case, shown in (b), which reproduces this bug. In lines 01-02 of (b), the test case loads an object of `TreeList`, which

[5] http://issues.apache.org/jira/browse/COLLECTIONS-70

has several nodes with integer values. A statement in Line 04 removes one node. A crash occurs when `previous()` is invoked in Line 09.

```
private AVLNode removeSelf() {
       ...
822  if(heightRightMinusLeft() > 0) {
       ...
834   } else{
835
836     AVLNode leftMax = left.max();
837     value = leftMax.value;
838     if(rightIsNext ) {
839       right = leftMax.right;
840     }
841     left = left.removeMax(); //buggy
842     if(relativePosition > 0) {
843       relativePosition--;
844     }
845   }
       ...
}
```

(a) a buggy code

```
public void test87() throws Throwable{
01  TreeList var0 =(TreeList ) Serializer
02                   .loadObjectFromFile("TreeList/hash_924548");
03  java.lang.Integer var1 =(java.lang.Integer) 10;
04  var0.remove(var1);
05  java.util.ListIterator var2 = var0.listIterator();
06  java.lang.Object var3 = var2.next();
07  java.lang.Object var4 = var2.next();
08  try{
09     java.lang.Object var5 = var2.previous();
10     fail("Expected exception of NullPointerException");
11   } catch(java.lang.NullPointerException e ) {
12    // Expected exception
13   }
}
```

(b) a test case reproducing the crash in (a)

```
+++    AVLNode leftPrevious = left.left;
841     left = left.removeMax();
+++    if(left == null ) {
+++      left = leftPrevious;
+++      leftIsPrevious = true;
+++     }
```

(c) fixed code of (a). Line +++ are added code.

Figure 42. A buggy code example from Apache Commons Collections and its reproducible test cases

This generated test case exactly reproduces the crash specified in the bug report and would have helped developers investigate the method in Line 04 in (b). The fixed code (c) shows the added statements to fix the code (a) by saving previous node information.

As another example, Figure 43 shows an erroneous source code snippet[6]. The crash happens in the `Erf.erf()` method. When the method is invoked with an input value greater than 26 (e.g., `Erf.erf(27)`), the crash happens with `MaxIterationsExceededException`. This study's approach identifies the crash condition and successfully generates a crash reproducible test case. By running the test case, developers can easily locate and fix the bug.

```
public static double erf(double x)
throws MathException {
51 double ret = Gamma.
               regularizedGammaP(0.5,  x * x,
                                 1.0e-15,  10000);
52 if (x < 0)  {
53   ret = -ret;
54 }
55 return ret;
}
```

Figure 43. A code snippet that illustrates #301 bug of Apache Commons-Math 2.0

7.4. Summary of Results

The experimental result shows this study's enhanced method-sequence generation approaches increases code coverage by 13.95% as compared to Randoop. In addition, this reduction approach removes 51.8% of the redundant test cases without the need to execute them.

[6] https://issues.apache.org/jira/browse/MATH-301

Captured object and mutated object help a state-of-the-art random testing tool, Randoop, to achieve high branch coverage: on average 70.7%, with 27.7% improvement while Randoop alone can only achieve 43.0%. This study's crash reproduction approach reproduces 44.6% of the original crashes automatically on 101 crash stack traces. These techniques are directly applicable to the existing widely deployed post-failure-process approaches and reproduce crashes without incurring any run-time overhead.

CHAPTER 8. DISCUSSIONS

In this chapter, issues regarding this study's two main approaches - the object-capture-based input generation and crash reproduction, are discussed. We also discuss threats to the validity of this study's approach.

8.1. Object-capture-based Input Generation

The achieved coverage of this study's object-capture-based input generation approach depends upon both quality and quantity of captured objects. However, capturing objects is a relatively easier process than writing unit test cases. For example, objects can be captured from:

- executions of system tests, and
- typical system executions by individual developers or users.

These executions can be completed independently and captured objects are reused for test input generation. Even captured object instances from a system can be reused for different system testing tasks as long as they share some objects. For example, objects in the JDK package (i.e., `java.util`) are commonly used for many Java applications. Note, a system under monitoring (a system with executions used by CAPTIG for capturing objects) is not necessarily the system where the classes under test are integrated. A system under monitoring can be another system that uses (either consumes or produces) objects falling into the same type of objects shared between a system under test and a system under monitoring. That is, CAPTIG can be applied even before classes under test are integrated. Note, existing manually written unit tests, or automatically generated unit tests for the classes under test can also be

used to produce normal program executions. In this case, a system under monitoring is just the classes under test.

8.1.1. Software Evolution

Captured objects may be obsolete and no longer valid during software evolution, since an object from running a new system version may have different fields and methods from a same-type object from running a previous version. Nonetheless, if both the (dynamically collected) method-sequences and the states of objects of concern, it is possible to locally reproduce the objects by executing the dynamically collected method-sequences and update the obsolete objects, based on the captured state information. Such an extended capability for CAPTIG remains as our future work.

8.1.2. Branches Still Not Covered

The evaluation result shows OCAT increases 25.5% of the branch coverage on average by using both generated and mutated objects. However, there are still more than 20% of branches not covered. As shown in Table 1, some not-covered branches (due to string manipulation, container object access, and exception branches) are difficult to cover. It is planned to further improve branch coverage via the following directions.

Cross-system object capturing. The sources for capturing objects do not need a limit to be only subject systems. Suppose one is testing ACC and it requires an object, FOO. It is possible to use other systems that use FOO and capture the objects from these systems. Capturing objects from one system and using them to test other systems may help further increase the applicability of this study's approach.

Iterative generation and mutation phases. Two phases, object generation and object mutation, can be iteratively applied to generate a larger number of objects over iteration. The quality of the generated objects may decrease when the number of iterations increases. However, the iterations could likely improve branch coverage gradually over time, since the two phases have complementary strengths to achieve structural code coverage, and the new objects generated from the object mutation phase could be exploited by the object generation phase.

8.1.3. Validity of Generated Objects

In the object generation phase, assume the generated method-sequences indirectly create valid objects, if invocations of the method-sequences do not throw exceptions. Indeed, the chance of creating invalid objects may be low in this case. However, this assumption may not be true, in general. In the object mutation phase, CAPTIG does not change private fields of objects as a default option. Even if CAPTIG does not change private fields and changes only public fields, it may still be likely to create invalid objects, which violates class invariants (either explicitly specified by developers or not specified at all). To avoid invalid objects, the developer could write the implementation of a special class-invariant-checking method, called `repOk()`, which checks the validity of mutated objects. However, this method can be difficult or time-consuming to implement. To explore this issue in future work, it is planned to empirically investigate how a high percentage of invalid objects could be generated by the object generation and object mutation phases, respectively, when `repOk()` method is not provided.

8.2. Crash Reproduction

This study's evaluation results show that CAPTIG reproduces 44.6% of crashes, which is promising. However, still 55.4% of crashes are not reproduced. This section discusses the challenges of reproduction, and presents threats to the validity of the evaluation.

8.2.1. Irreproducible Crashes

To understand the challenges of reproduction, manually inspect the main causes of irreproducible crashes by CAPTIG. The main causes[7] are listed in Table 17. We discuss each cause in detail.

Table 17. Main causes of irreproducible crashes in AJDT.

cause	# of crashes
mutation challenges	21 (25.3%)
insufficient objects	11 (13.3%)
XStream crash	6 (7.2%)
fail to compute preconditions	4 (4.8%)
other	3 (3.6%)

8.2.2. Mutation Challenges

After computing the weakest precondition to reproduce crashes, CAPTIG mutates objects accordingly. However, this process has several challenges.

Challenge 1: Keeping Class Invariants - To avoid violating class invariants, CAPTIG only mutates object member fields that are publicly accessible. This limits CAPTIG's muta-

[7] Note, an XStream crash is a technical engineering issue, while the others illustrate the challenges of this study's approach.

tion ability. Suppose one of the predicates in the computed weakest precondition is related to a private field. Even though CAPTIG has obtained a satisfiable model, it could not directly mutate the private field, since such mutation would violate class invariants. By mutating only public fields, the mutated objects may only partially satisfy the predicates in the computed weakest preconditions and may not reproduce the original crashes.

Challenge 2: Understandings in Abstract Semantics - In some cases, a predicate in the precondition could not be satisfied by simply mutating object member fields. In total, 25% of the irreproducible crashes fall into this category. For example, predicate `specialContainer.size() == 10` could not be satisfied because CAPTIG does not know how to increase the number of objects in this special container class. Satisfying such predicates requires understanding the abstract semantics of the target program. Understanding the abstract semantics could be completed by using static analysis techniques, such as abstract interpretation [75]. Human assistance is also helpful in satisfying this kind of predicates by providing program semantic information.

8.2.3. Lack of Objects

CAPTIG requires input objects to reproduce crashes generated with two different techniques—feedback-directed and object-capture-based. However, when irreproducible crashes are investigated, it was determined that many necessary object types are not instantiated. Without necessary object inputs, the generated test cases cannot be executed, and thus cannot reproduce the original crashes. As shown in Table 4, 11 out of the 83 crashes in AJDT are not reproduced by CAPTIG for this reason. In this study's experiment, the object capturing

ability of CAPTIG is limited to the provided system tests. Adding more object capturing or object generation approaches to CAPTIG might be helpful to address this challenge.

8.2.4. Failure to Compute Weakest Preconditions

Four crashes are not reproduced because CAPTIG could not compute the weakest preconditions, which satisfy the corresponding crash conditions. Due to the undecidability problem [24], a precondition computation algorithm could run without termination. To address this issue, some approximations were made for this study's weakest precondition computation algorithm `ComputeWP()`. These include unrolling loops only a limited number of times and setting a maximum depth for invocations. However, with these approximations, CAPTIG might not find a feasible weakest precondition to reproduce the target crash. For example, a crash that occurs inside a loop might not be reproducible, as it requires CAPTIG to unroll the loop many times to find a feasible weakest precondition.

Another reason for failing to compute a weakest precondition is the presence of non-linear arithmetic in the predicates. General SMT solvers have limited support for non-linear arithmetic over an infinite solution domain [43].

8.3. Threats to the Validity

The following threats to validity of this study's evaluation were determined.

The subject software might not be representative. In this experiment, open-source systems were used as subjects. Since systems with high quality bug reports and crash stack traces were intentionally selected, there might exist a subject selection bias. In addition, all of the

subjects are open-source. Hence, they might not be representative of closed-source systems. This threat could be reduced by more experiments on wider types of subjects in future work.

CAPTIG results rely on captured objects. In this study's evaluation, system tests provided by the open-source systems were used to capture objects. As a result, the quality and quantity of captured objects depend on the subject system and given capturing executions. Thus, CAPTIG's achieved coverage improvement and the reproducibility of CAPTIG may depend on the quality and quantity of the system tests initially provided with the subject systems. In future work, it is planned to empirically investigate the impact of either quality or quantity of existing system tests or other program executions on the effectiveness of CAPTIG.

CHAPTER 9. REVIEW OF LITERATURE

This chapter describes related works of random input generation, object input construction and crash reproduction. Related works in this chapter are wider than background works in Chapter 2, which discusses works directly related to this study.

9.1. Random Input Generation

There are several random testing tools for the object-oriented system testing [21][30][35][41][72]. JCrasher [41] uses return values as inputs, building parameter graphs to determine method calls whose return values are the same type of input values. Agitator [30] creates test inputs by using several well-known techniques, such as symbolic execution, feedback-driven, and model-based random input generation. Jartege [71] uses a model-based random generation to generate random inputs, based on JML specifications. RUTJ [21] is a GUI-based randomized unit-testing tool that requires the user to write code as seed inputs. ARTOO [35] applies the distance-based adaptive random testing [33] to choose inputs from an object pool. ARTOO incorporates AutoTest [67], which uses a formal specification to determine whether randomly generated method calls are error revealing.

Although some of these state-of-the-art random testing techniques scale up to test large-scale software systems, random testing is still commonly thought of as ineffective for large-scale software systems. For example, Randoop can only achieve less than 50% branch coverage with enormous test cases toward large-scale software systems. Most of the generated test cases are redundant.

9.2. Object Input Construction

Software systems, based on object-oriented programming techniques, are in widespread use on contemporary computing systems. However, testing an object-oriented system is challenging. For example, object-oriented programming increases the complexity of testing. Object type arguments have a large search space for objects, because an object can have many member fields of primitive types (e.g., integer) and other object types (e.g., a container class, such as stack and list). These difficulties in testing of objected-oriented systems make code coverage of automated test input generation tools unsatisfactory[8]. Many researchers have devoted their efforts on this issue and have introduced several tools.

There are two main types of techniques for generating desirable objects—direct object construction and method-sequence generation.

9.2.1. Direct Input Construction

Two representative techniques for direct object construction are Korat [31] and TestEra [59]. Korat requires users to provide a `repOk()` predicate method, which checks the validity of a given object against the required class invariant for the class of the object. TestEra requires users to provide class invariants specified in the Alloy specification language [55]. Both Korat and TestEra use class invariants to efficiently prune both invalid objects (those violating class invariants) and redundant objects (those with isomorphic states) when generating a bounded-exhaustive set of objects (whose size[9] is within a relatively small bound). These techniques also require users to provide a finite domain of values for primitive-type

[8] This study's results show they are lower than 50% of branch coverage.
[9] The size of an object is the total number of objects used to construct direct or indirect non-primitive fields of the object.

fields in the generated objects. The object-capturing phase of our CAPTIG approach can be seen as a type of direct object construction. However, CAPTIG constructs objects from normal program executions (from system tests or real use) and these captured objects are valid by construction, without requiring class invariants or a finite domain of values for primitive-type fields of objects.

9.2.2. Method-sequence Generation

Various techniques on method-sequence generation have been proposed for generating objects used in test input generation. Random-testing techniques (such as JCrasher [41] and Randoop [74]) generate random method-sequences, sometimes with pruning based on feedback from previously generated sequences. Evolutionary testing techniques (e.g., eToc [85] and Evacon [54]) use genetic algorithms to evolve initial method-sequences to ones more likely to cover target branches. This study's CAPTIG approach integrates Randoop, a random-testing technique, for evolving captured objects, and, in principle, can integrate an evolutionary technique to evolve captured objects. The object-capturing and object-mutation phases of CAPTIG provide benefits beyond the integrated random testing technique in achieving branch coverage, as shown in the empirical evaluation.

9.2.3. Exhaustive Testing

Bounded-exhaustive testing techniques (such as JPF [86], Rostra [88], and Symstra [89]) for method-sequence generation produce exhaustive method-sequences to a certain length, sometimes with pruning of equivalent concrete objects [86][88] or subsumed symbolic objects [89]. However, the coverage of various branches requires long method-sequences, whose lengths are beyond the small bound that can be handled by these techniques. In con-

trast, this CAPTIG approach is able to capture objects produced with long method-sequences (from real system executions) and further evolve these objects with more method-sequences or directly mutate these objects.

9.2.4. Sequence Mining

Recent sequence-mining techniques, such as MSeqGen [82], statically collect method-sequences (that can produce objects of a specific) from various applications, and then apply dynamic symbolic execution [82] or random testing [73] on these collected sequences. MSeqGen shares the same spirit with CAPTIG in exploiting code elsewhere beyond the code implementation under test to improve automated test input generation. This study's CAPTIG approach has unique benefits over MSeqGen in the following two main aspects. First, CAPTIG dynamically captures real program execution environments, such as user inputs, global states, and file I/O; whereas, MSeqGen statically collects partial method-sequences (from code bases), whose later execution often does not reproduce desirable program execution environments. Second, CAPTIG can capture objects produced or affected by multi-threading; whereas, MSeqGen does not support the collection of method-sequences involving multi-threading. On the other hand, MSeqGen has its advantages, complementing CAPTIG in addressing the issue of generating desirable objects. For example, since CAPTIG dynamically captures objects from program executions, the ability to capture objects relies on the quality of not only the programs under monitoring, but also the system tests or real use that produces the program executions; whereas, MSeqGen relies on the quality of only the programs under mining.

9.3. Crash Reproduction

9.3.1. Capture-Replay

Capture-replay techniques have been used for debugging [22][80] to reproduce crashes. Normally, these techniques record and replay program execution by setting checkpoints, and capturing global states and event logs. They set up monitoring environments, monitor execution of the target program, record the status and behavior of a checkpoint, and convert the captured program execution into unit tests. The common applications of these approaches are regression testing, based on recorded execution.

For example, the techniques need to change a platform (e.g., modifying java virtual machine) or a target program (e.g., bytecode instrumentation) to build an environment for monitoring. Next, the techniques set checkpoints to record the state of a program under monitoring at the moments. Then, the execution of the program under monitoring saves log events between checkpoints. Finally, the recorded execution can be replayed, based on the recorded data. To record a failure, a software engineer runs the subject program until the failure reoccurs. Then, the software engineer replays the recorded execution that has exactly the same unit behavior that causes the failure.

Among them, jRapture [80] and ReCrash [22] have recently been used to generate unit tests to reproduce a given program failure, based on recording failure execution. These techniques have recently been used to generate unit tests for testing purposes, such as regression testing. During system testing, these techniques capture the interaction (i.e., method calls) of a unit such as the class under test with other classes interacting with the unit. Based upon captured interactions, these techniques then generate unit tests for the class under test; in con-

trast to the system tests, these unit tests are less expensive to run when the class undergoes some changes. The execution of the generated unit tests would replay exactly the same unit behavior exercised in the capturing phase; the code coverage of the unit achieved in the capturing phase is the same as the code coverage of the unit achieved by the generated unit tests. In contrast, CAPTIG evolves and mutates the captured inputs, achieving higher structural code coverage than the capturing phase.

9.3.2. Post-Failure-Process

Bugzilla [79] and JIRA [7] are bug report management systems that allow users to report bugs along with crash stack traces. Microsoft Windows Error Reporting (WER) [47], Apple Crash Reporter [8], Google Breakpad [14], and Mozilla Talkback [18] automatically collect crash information after crashes occur. This collected information is useful for debugging or prioritizing debugging efforts [46]. However, these systems do not support automated crash reproduction. CAPTIG facilitates automated crash reproduction using crash data collected by these systems.

CHAPTER 10. CONCLUSIONS AND FUTURE WORKS

This chapter describes our three approach enhanced method-sequence generation, object-capture-based input generation and crash reproduction. Lastly, future works are described.

10.1. Enhanced Method-sequence Generation

Extended techniques were proposed, such as simplified distance-based input selection, on-demand input creation, type-based input selection, coverage-based method selection, open-access selection, and sequence-based reduction techniques, to increase coverage and eliminate redundant test cases, based on achieving preset coverage goal. This study implemented these techniques on top of Randoop and tested with large-scale software systems. This study's experiments show that all techniques increased code coverage. This study's techniques yielded the best code coverage when all are applied together; they increased code coverage up to 22.8% and eliminated 51.8% of redundant test inputs.

This proposed approach is applicable for other existing testing techniques. Anyone who wants to increase random testing coverage should consider integrating these suggested techniques into their tools.

10.2. Object-capture-based Input Generation

In automated unit testing of object-oriented software, one important and yet challenging problem is to generate desirable objects for receivers or arguments to achieve high code coverage (such as branch coverage) or find bugs. To address this significant problem, the CAPTIG approach was proposed, which captures objects from program executions, generates more objects using captured objects and method-sequences (exploiting a state-of-the-art ran-

dom-testing tool called Randoop [74]), and mutates the objects to cover those not-yet-covered branches. CAPTIG was evaluated with three open-source systems. This study's empirical results showed that CAPTIG helps Randoop achieve high branch coverage—70.7%, on average, improved from 43.0% achieved by Randoop alone.

10.3. Crash Reproduction

Automated crash reproduction techniques were proposed by using only stack traces, and presented CAPTIG, an implementation of this study's techniques. The experiments on the bug reports of AspectJ Development Tools, Apache Commons Collection, and Apache Commons Math showed that CAPTIG reproduces 44.6% of crashes and the reproduced crashes are useful for debugging.

CAPTIG is directly applicable to existing bug report management systems and post-failure process systems, such as Microsoft Windows Error Reporting, Apple Crash Reporter, Google Breakpad, and Mozilla Talkback, without having to deploy new software. Therefore, CAPTIG does not incur any run-time performance overhead. In addition, automated crash reproduction will significantly reduce developers' debugging efforts.

10.4. Future Works

Instead of capturing object states, we capture (1) method sequence without actual input arguments and (2) class usage patterns that reveal method invocation orders. In addition, the capture-based input generation technique need to be applied to other input generation technique rather than the method sequence generation technique. This work remains as our future work.

REFERENCES

[1] Apache Software Foundation. Apache Commons Collections http://commons.apache.org/collections/

[2] Apache Software Foundation. Apache Commons Math. http://commons.apache.org/math/

[3] Apache XML Security. http://santuario.apache.org/

[4] Apache Ant. http://ant.apache.org.

[5] ASM: Java bytecode manipulation and analysis framework. http://asm.objectweb.org/

[6] AspectJ compiler. http://eclipse.org/aspectj/

[7] Atlassian Inc. JIRA. http://www.atlassian.com/software/jira/

[8] CrashReporter. Technical Report TN2123, Apple Inc., 2004.

[9] Cobertura. http://cobertura.sourceforge.net/.

[10] Eclipse C/C++ Development Tools. http://www.eclipse.org/cdt/

[11] Eclipse Equinox. http://www.eclipse.org/equinox/

[12] Eclipse Graphical Editing Framework. http://www.eclipse.org/gef/

[13] Eclipse Java Development tools. http://www.eclipse.org/jdt/

[14] Google Inc. Breakpad. http://code.google.com/p/google-breakpad/

[15] IBM. T.J. Watson Libraries for Analysis (WALA). Online manual. http://wala.sf.net

[16] Java SDK. http://java.sun.com

[17] JSAP http://www.martiansoftware.com/jsap/

[18] Mozilla Foundation. Talkback. http://talkback.mozilla.org

[19] CrashReporter. Technical Report TN2123, Apple Inc., Cupertino, CA, 2004.

[20] P. Ammann and J. Offutt. Introduction to Software testing. Cambridge University Press, Cambridge, UK, 2008.

[21] J. H. Andrews, S. Haldar, Y. Lei, and Felix. Tool support for randomized unit testing. In International workshop on Random testing (RT), pages 36–45, 2006.

[22] J. Anvik, L. Hiew and G. C. Murphy. Coping with an open bug repository. In Proc. OOPSLA workshop on eclipse technology eXchange, pages 35-39, 2005

[23] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Pre-serving Objects. In Proc. European Conference on Object-Oriented Programming (ECOOP), pages 542–565, 2008.

[24] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In Proc. International Conference on Software Engineering (ICSE), pages 211–220, New York, NY, USA, 2008. ACM.

[25] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. Finding similar failures using callstack similarity. In Proc. SysML: Workshop on Tackling Computer Systems Problems with Machine Learning Techniques. USENIX, 2008.

[26] K. Beck and E. Gamma. Test infected: Programmers love writing tests. Java Report, 3(7):37–50, 1998.

[27] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 308–318, New York, NY, USA, 2008. ACM.

[28] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Extracting structural information from bug reports. In Proc. International working conference on Mining Software Repositories (MSR), pages 27–30, New York, NY, USA, 2008. ACM.

[29] N. Bjorner, N. Tillmann, and A. Voronkov. Path feasibility analysis for string-manipulating programs. In Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS), pages 307–321, 2009.

[30] M. Boshernitsan, R. Doong, and A. Savoia. In International Symposium on Software Testing and Analysis (ISSTA), pages 169–180, New York, NY, USA, 2006.

[31] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In Proc. International Symposium on Software Testing and Analysis (ISSTA), pages 123–133, 2002.

[32] T. Y. Chen and M. F. Lau. Dividing strategies for the optimization of a test-suite. In Information Processing Letters, pages 135–141, 1996.

[33] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In Proc. Asian Computer Science Conference (ASIAN), pages 320–329, 2004.

[34] T. Y. Chen, F. Kuo, R. Merkel, and S. Ng, Mirror adaptive random testing, In Proc. International Conference On Quality Software (QSIC), 2003. pp. 4–11, Nov. 2003.

[35] K. P. Chan, T. Y. Chen, and D. Towey, Restricted random testing, In Proc. International Conference on Software Quality (ECSQ). London, UK, 2002, pp. 321–330.

[36] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. ARTOO: adaptive random testing for object-oriented software. In International Conference on Software Engineering (ICSE), pages 71–80, 2008.

[37] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest pre-conditions. In Proc. Programming Language Design and Implementation (PLDI), pages 363–374, 2009.

[38] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In Proc. International Conference on Software Engineering (ICSE), pages 261–270, Minneapolis, Minnesota, May 2007.

[39] D. R. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML - Progress and issues in building and using ESC/Java2. In Proc. Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS), pages 108–128. Springer–Verlag, 2004.

[40] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. Symposium on Principles of Programming Languages (POPL), pages 238–252, New York, NY, USA, 1977. ACM.

[41] C. Csallner and Y. Smaragdakis. JCrasher: an automated robustness tester for Java. Software: Practice and Experience, 34:1025–1050, 2004.

[42] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. In ACM Trans. Program. Lang. Syst., 13(4):451–490, 1991.

[43] E. W. Dijkstra. A Discipline of Programming. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[44] B. Dutertre and L. de Moura. System description: Yices 1.0. In Proc. Satisfiability Modulo Theories Competitio (SMT-COMP), 2006.

[45] M. D. Ernst. Dynamically Discovering Likely Program Invariants. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, Aug 2000.

[46] M. R. Garey and D. S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Booksin the Mathematical Sciences). 1979.

[47] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. In Proc. Symposium on Operating Systems Principles (SOSP), pages 103–116, New York, NY, USA, 2009. ACM

[48] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In Proc. Programming Language Design and Implementation (PLDI), pages 188–198, 2009.

[49] R. Hamlet. Random testing. In Encyclopedia of Software Engineering, pages 970–978, 1994.

[50] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test-suite. In ACM Transactions on Software Engineering and Methodology (TOSEM), number 3, pages 270–285, 1993.

[51] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In European Conference Object-Oriented Programming (ECOOP), pages 431–456, 2003.

[52] J. Horgan and S. London. A data flow coverage testing tool for C. In Assessment of Quality Software Development Tools (AQSDT), pages 2–10, 1992.

[53] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In Proc. Programming Language Design and Implementation (PLDI), pages 188–198, 2009.

[54] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 12:26–60, 1990.

[55] K. Inkumsah and T. Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In Proc. Automated Software Engineering (ASE), pages 297–306, September 2008.

[56] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 62–73, 2001.

[57] H. Jaygarl, S. Kim, and C. K. Chang. Practical extensions of a randomized testing tool. In Proc. Computer Software and Applications Conference (COMPSAC), 2009.

[58] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: Object Capture-based Automated Testing. In Proc. International Symposium on Software Testing and Analysis (ISSTA), July 2010.

[59] H. Jaygarl, K.-S. Lu, and C. K. Chang. Genred: A tool for generating and reducing object-oriented test cases. In Proc. Computer Software and Applications Conference (COMPSAC),, 2010.

[60] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. Automated Software Engineering (ASE), 11(4):403–434, 2004.

[61] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: a solver for string constraints. In Proc. International Symposium on Software Testing and Analysis (ISSTA), pages 105–116, 2009.

[62] D. Kim, X. Wang, S. Kim, A. Zeller, S. Cheung, and S. Park. Which crashes should i fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. IEEE Transaction Software Engineering (TSE), 2010.

[63] J. C. King. Symbolic execution and program testing. Communications. ACM, 19(7):385–394, 1976.

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

[65] B. Liskov and J. Guttag. Program Development in Java: Abstraction, specification, and object-oriented design. Addison-Wesley, 2000.

[66] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In Extreme Programming Examined. Addison-Wesley, 2001.

[67] B. Meyer, I. Ciupa, A. Leitner, and L. Liu. Automatic testing of object-oriented software. In Theory and Practice of Computer Science (SOFSEM), pages 114–129, 2007.

[68] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for java. In Proc. International Conference on Software Engineering (ICSE), pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society.

[69] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. SIGARCH Comput. Archit. News, 33(2):284–295, 2005.

[70] A. J. Offutt, J. Pan, and J. M. Voas. Procedures for reducing the size of coverage-based test sets. In International Conference on Testing Computer Software (ICST), pages 111–123, 1995.

[71] C. Oriat. Jartege: a Tool for Random Generation of Unit Tests for Java Classes. In Quality of Software Architectures and Software Quality (QoSA/SOQUA), pages 242–256, 2004.

[72] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In European Conference Object-Oriented Programming (ECOOP), pages 504–527, 2005.

[73] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed input generation generation. In Proc. International Conference on Software Engineering (ICSE), pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

[74] C. Pacheco and M. D. Ernst. Randoop: feedback-directed input generation for Java. In Object Oriented Programming, Systems, Languages and Applications (OOPSLA), Oct. 2007.

[75] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proc. Symposium on Principles of Programming Languages (POPL), pages 238–252, New York, NY, USA, 1977. ACM.

[76] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automated test factoring for Java. In Proc. Automated Software Engineering (ASE), pages 114–123, November 2005.

[77] K. Sen. Effective random testing of concurrent programs. In Proc. Automated Software Engineering (ASE), pages 323–332, 2007.

[78] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In Proc. Computer Aided Verification (CAV), pages 419–423, 2006.

[79] N. Serrano and I. Ciordia. Bugzilla, itracker, and other bug trackers. IEEE Software, 22:11–13, 2005.

[80] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In Proc. International Symposium on Software Testing and Analysis (ISSTA), pages 158–167, 2000.

[81] Sun Microsystems. Java Reflection API, 2001. Online manual.

[82] S. Thummalapenta, T. Xie, N. Tillmann, P. de Halleux, and W. Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In Proc. European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), August 2009.

[83] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In Proc. Tests and Proofs (TAP), pages 134–153, 2008.

[84] N. Tillmann and W. Schulte. Mock-object generation with behavior. In Proc. Automated Software Engineering (ASE), pages 365–368, 2006.

[85] P. Tonella. Evolutionary testing of classes. In Proc. International Symposium on Software Testing and Analysis (ISSTA), pages 119–128, 2004.

[86] W. Visser, C. S. Pˇasˇareanu, and R. Pelánek. Test input generation for Java containers using state matching. In Proc. International Symposium on Software Testing and Analysis (ISSTA), pages 37–48, 2006.

[87] E. Y. C. Wong, A. T. S. Chan, and H.-V. Leong. Efficient management of XML contents over wireless environment by Xstream. In Proc. Symposium on Applied Computing (SAC), pages 1122–1127, 2004.

[88] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In Proc. Automated Software Engineering (ASE), pages 196–205, 2004.

[89] T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In Proc. Tools and Algorithms for Construction and Analysis of Systems (TACAS), pages 365–381, 2005.