REDUCING THE COSTS OF BOUNDED-EXHAUSTIVE TESTING

BY

VILAS SHEKHAR BANGALORE JAGANNATH

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Advisers:

Professor Gul Agha
Assistant Professor Darko Marinov

Abstract

Bounded-exhaustive testing is an automated testing methodology that checks the code under test for *all inputs* within given bounds: first the user describes a set of test inputs and provides test oracles that can check test outputs; then a tool generates all the test inputs, executes them on the code under test, and checks the test outputs; finally the user inspects the failing tests to submit bug reports. The costs of bounded-exhaustive testing include machine time for test generation and execution (which translates into human time waiting for these results) and human time for inspection of results. We propose and evaluate three orthogonal techniques that reduce these costs. Sparse Test Generation skips the generation of some test inputs to reduce the time taken to reach the first failing test. Structural Test Merging generates a smaller number of larger test inputs (rather than a larger number of smaller test inputs) to reduce test generation and execution time. Oracle-based Test Clustering groups failing tests to reduce the inspection time. Results obtained from the bounded-exhaustive testing of the Eclipse refactoring engine show that these three techniques can substantially reduce the costs without significantly sacrificing fault-detection capability.

To my family.

Acknowledgments

I would like to thank:

- My parents, my sister and Diana for their love and support.
- Prof. Darko Marinov for his immense dedication, drive, guidance and support.
- Prof. Gul Agha for introducing me to graduate research and for his continued guidance and support.
- Yun Young Lee and Brett Daniel for their collaboration towards the work presented in this thesis.
- Danny Dig for inspecting the faults we found in Eclipse, and the students from the Fall 2008 Advanced Topics in Software Engineering class at our department for their feedback on this work.
- Members of both my research groups and my other friends at Illinois including Milos Gligoric, Steven Lauterburg, Rajesh Karmani, Vijay Anand Reddy, Kevin Lin, Lucas Cook and Alejandro Gutierrez for making my graduate student life fun and interesting.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

APFD	Average Percentage Fault Detection
OTC	Oracle-based Test Clustering
STG	Sparse Test Generation
STM	Structural Test Merging
TTFF	Time To First Failure

Chapter 1

Introduction

Testing is an important but expensive part of software development, estimated to take more than half of the total development cost [6]. One approach to reducing the cost is to automate testing. The increasing popularity of modern software development methodologies, such as Extreme Programming [5], has increased the use of automated testing, especially automated test *execution*, which involves automatically checking manually written tests. Automated test *generation* is also becoming more popular; it involves programmatic generation of test inputs for the code under test and programmatic validation of test outputs (namely *test oracles*).

Bounded-exhaustive testing is an automated test generation approach that checks the code under test for *all inputs* within given bounds [7, 8, 20, 26, 36]. The rationale is based on the *small scope hypothesis*, which argues that many faults can be revealed within small bounds [17, 24], and exhaustively testing within the bounds ensures that no "corner case" is missed. Bounded-exhaustive testing has been used in both academia and industry to test several real-world applications, with some recent examples including testing of refactoring engines [8, 15], compilers [15] and web-traversal code [26].

Bounded-exhaustive testing consists of three activities. First, the user describes a set of test inputs (this description is often called a *generator*) and also provides test oracles that can check test outputs. Second, the tool generates all the inputs described by the user, executes them on the code under test, and checks the outputs using the oracles. Third, the user inspects failing tests to submit bug reports or debug the code; typically, bounded-exhaustive testing produces a large number of *failures* for each *fault* found. Previous experience shows

that bounded-exhaustive testing can reveal important faults [8, 20, 35, 36] but can also have high costs. Two key costs in this context are *machine time* for test generation and execution (which also translates into human time for waiting for these results [10, 33]) and *human time* for inspection of failures.

We propose, and evaluate on a case study, three novel techniques [18] that reduce these costs of bounded-exhaustive testing. The three techniques can be used individually or synergistically to reduce the key costs of bounded-exhaustive testing. Specifically, we make the following contributions:

Sparse Test Generation (STG) We present a new technique that reduces the time to first failure (abbreviated *TTFF*), i.e., the time that the user has to wait after starting a tool for bounded-exhaustive testing until the tool finds *a* failing test. Note that in this context there could be a large number of failing tests (say, hundreds or even thousands) or no failing tests (if the code under test has no faults for any generated tests). TTFF measures only the time to the *first* failure (not all failures). It is an important practical metric that captures the user idle time. Previous research has shown, in a related context of regression testing, that reducing the time to failure can significantly help in development [10, 33]. STG works by making two passes through test generation. The first, *sparse* pass, skips some tests in an attempt to reduce TTFF. While this pass is related to test suite minimization/reduction/prioritization [12, 21, 31, 34, 37, 39], the main challenge is to skip tests while they are being generated and not to select some tests only after all have been generated. The second, *exhaustive* pass, generates all the tests to ensure exhaustive checking within the given bounds (because sampling some tests and failures could lead to missing some faults and because having more failures per fault can help in debugging [9, 19]). Effectively, STG trades off (substantially) decreasing TTFF for (slightly) increasing the total time required for test generation and execution.

Structural Test Merging (STM) We present a new technique that reduces the total time for test generation and execution. In bounded-exhaustive testing, users typically describe a test set with a *large number of small tests*, while we advocate also considering test sets with a *smaller number of larger tests*. Our technique is inspired by the work on test granularity [28, 29] which studied the cost-benefit trade-offs in using a larger number of smaller tests versus a smaller number of larger tests. That work mostly considered manually written tests for regression testing, while we focus on automatically generated tests. Furthermore, that work considered cases where larger tests can be automatically built from smaller tests by simply *appending* (e.g., if each test is a sequence of commands, a longer test sequence can be obtained by simply appending a number of shorter test sequences), while we consider cases where it is harder to build larger tests from smaller tests (e.g., simply appending two test input programs together while testing a compiler or a refactoring engine would likely result in a compilation error as these programs could have program entities with the same name; moreover, renaming would reduce the opportunity for speeding up test execution). Instead of simply appending tests, our technique *merges* them based on their structure, hence the name STM.

Oracle-based Test Clustering (OTC) We present a new technique that reduces the human time for inspection of failing tests. Bounded-exhaustive testing can produce a large number of failing tests, and a tester/developer has to map these failures to distinct faults to submit bug reports or debug the code under test. Our technique builds on the ideas from test clustering [9, 19, 22, 23, 27, 32] where the goal is to group failing tests such that all tests in the same group are likely due to the same underlying fault. Previous work mostly considered manually written tests or actual program runs, and clustering was based on *execution profiles* obtained from monitoring test execution. In contrast, we consider automatically generated test inputs, and our technique ex-

ploits information from test oracles. Typically, an oracle only states *whether* a test passed or failed, i.e., the output from an oracle is a boolean. However, in some domains oracles also state *how* the result is incorrect, i.e., the output from an oracle is an error *message*. This is the case with some of the oracles that we use, like the Compilation Failure Oracle (described in Chapter 3) which provides detailed compiler errors when there is a failure. OTC groups failing tests based on oracle messages, and our results suggest that it is beneficial to build such oracles whenever possible. The key to our technique is *abstracting* messages and not comparing them directly since that could lead to excessive clustering, i.e., multiple groups representing a single fault. For example, instead of comparing compiler errors verbatim, we were able to obtain better clustering of failures by abstracting away unnecessary details such as the line and column numbers.

Case Study We evaluated our three new techniques in the context of bounded-exhaustive testing of the Eclipse [2] refactoring engine using the ASTGen framework for bounded-exhaustive testing of refactoring engines [1, 8]. We chose ASTGen for three reasons: (1) It enabled finding actual faults in real large software (it had previously been used to find a few dozen new faults in the refactoring engines of Eclipse and NetBeans [3], two popular IDEs for Java [8]); (2) We were familiar with the framework; and (3) We personally experienced the costs of using ASTGen. We implemented the three techniques within ASTGen and evaluated the cost savings achieved by the techniques while testing 6 refactorings with 9 generators (more details in Chapter 5). The results showed that (1) STG can reduce TTFF almost 10x (an order of magnitude) when there is a failure, while increasing the total test generation and execution time only 10% when there is no failure; (2) STM can reduce the total time 2x-6x (in one instance from over 6 hours to 70 minutes) and even more (but with some reduction of the fault-detection capability); (3) OTC can reduce the number of tests to be inspected

by clustering hundreds of failing tests into a few groups (up to 11) such that almost all tests within the group are due to the same fault. In summary, the results show that the techniques can substantially reduce both the machine and the human costs of bounded-exhaustive testing without reducing the fault-detection capability.

The rest of this thesis is organized as follows: Chapter 2 introduces the driving example used throughout this thesis and presents the techniques in the context of the example. Chapter 3 presents the background necessary to introduce the details of the three techniques. Chapter 4 presents the implementation details of the three techniques introduced in this thesis. Chapter 5 details the experiments that we performed to evaluate the techniques. Chapter 6 discusses other work that is related to the techniques presented in this thesis. Finally, Chapter 7 concludes the thesis and discusses avenues for future work.

Note that the work presented in this thesis has already been published in the form of a conference paper at the 12th International Conference on Fundamental Approaches to Software Engineering (FASE 2009) [18]. The author of this thesis presented this work at the FASE 2009 conference. We would like to thank the audience of the talk, including Dimitra Giannakopoulou who chaired the session, for their comments and questions which have been incorporated in this thesis to improve the presentation and provide additional details where required. We would also like to thank the anonymous reviewers who reviewed our paper for FASE 2009 for their useful comments.

Chapter 2

Example

To illustrate how our techniques reduce the costs of bounded-exhaustive testing, we discuss testing the PullUpMethod refactoring in the Eclipse refactoring engine using the ASTGen framework. We first describe what PullUpMethod is. We then describe how to use ASTGen for bounded-exhaustive testing of this refactoring. Finally we discuss how our new techniques improve ASTGen.

A refactoring is a program transformation that changes the program code but not its external behavior [14]. Programmers undertake refactorings to improve design of their programs. For example, PullUpMethod is a refactoring that moves a method from a class into one of its superclasses (usually because the same method can be used by other subclasses of that superclass). Figure 2.1 shows a simple application of the PullUpMethod refactoring. Note that moving the method also requires properly updating the references within the method body, i.e., replacing `super.f` with `this.f`.

Performing refactorings manually is tedious and error-prone. Hence, most modern IDEs such as Eclipse, contain *refactoring engines*, which are development tools that automate applications of refactorings. To apply PullUpMethod, the developer informs the engine

```
// Before refactoring
class A {
  int f;
}
class B extends A {
  void m() {
    super.f = 0;
  }
}
```

```
// After refactoring
class A {
  int f;
  void m() {
    this.f = 0;
  }
}
class B extends A {
}
```

Figure 2.1: Successful application of the PullUpMethod refactoring

```
// Before refactoring
class A {
}
class B extends A {
  int f;
  void m() {
    this.f = 0;
  }
}
```

```
// Refactoring engine
// warning:

// Cannot pull up:
// method 'm'
// without pulling up:
// field 'f'
```

Figure 2.2: Application of the PullUpMethod refactoring resulting in a warning

about the method to move and the superclass to move it to in the *input* program. The engine first checks whether the move is permitted (e.g., PullUpMethod should not move a method to a superclass if the superclass already has a method with the same signature). If it is, the engine appropriately transforms the program. The *output* is either a transformed program, as illustrated in Figure 2.1 or a set of warning messages that indicate why the move is not permitted, as illustrated in Figure 2.2.

Testing the implementation of PullUpMethod requires generating a number of input programs, invoking the refactoring engine on them, and checking whether it gives the appropriate output (either a correctly transformed program or an expected set of warning messages). Testers can have good intuition about which input programs could reveal a fault in the engine. For instance, PullUpMethod may have faults if the subclass and superclass have some additional relationship, e.g., being an inner or a local class or being related through a third class. Also, there may be faults for some expressions and statements that include field and method references from the body of the method being pulled up or to the method being pulled up. However, it is time-consuming and error-prone to manually generate a large number of such input programs.

Our research group previously developed the ASTGen framework for bounded-exhaustive testing of refactoring engines [8]. ASTGen allows the tester to write *generators* that can automatically produce a (large) number of (small) input programs for testing refactorings. ASTGen generates *all* these inputs, executes the refactoring engine on them, runs several oracles to validate the outputs, and reports failures. Using ASTGen, several dozen new bugs were found in the refactoring engines of Eclipse and NetBeans, two popular IDEs for Java.

However, ASTGen required a lot of machine time to generate and execute the test inputs and a lot of human time to inspect the results (to filter true failures from false positives and to map true failures to distinct faults).

For instance, to test PullUpMethod, we can use a generator that produces programs with three classes in various relationships. For this specific case, ASTGen generates 1,152 input programs, of which 160 result in failing oracles. A detailed inspection of these failures shows that they reveal 2 distinct faults. While finding these faults is clearly positive, there are costs. Test generation and execution (including oracles) take about 27 minutes (on a typical desktop), and the time to find the first failure is about 9 minutes (in test run 389 of 1,152). Also, identifying the 2 distinct faults among 160 failing tests is labor-intensive and tedious.

This thesis proposes three techniques that reduce all these costs:

STG addresses the time to first failure (TTFF) by first sampling some inputs rather than exhaustively generating all inputs from the beginning. For our specific example, TTFF is reduced almost an order to magnitude, from about 9 minutes to 1 minute.

STM addresses the total time for test generation and execution. Instead of testing PullUpMethod for 1,152 (small) programs that exercise various features in isolation, STM builds larger programs that combine some of the features, e.g., combine several expressions or statements that include field and method references to/from the method being pulled up. The tester can choose how many features to combine. In this example, the least aggressive combination reduces the total time from 27 minutes to about 4 minutes, and the most aggressive combination reduces the total time further to under 1 minute.

OTC addresses the cost of failure inspection. It clusters the failing tests into groups that are likely to be due to the same fault, and thus the tester can inspect only one or a few tests from these "equivalence classes". Our clustering is based on oracle messages and can consider more or fewer details of the messages. The basic clustering splits 160

failing tests into 127 clusters, but our best clustering splits them into just 3 clusters that reliably find the 2 faults. In contrast, random sampling could miss faults, e.g., one of our experiments shows that it finds on average 1.77 out of 2 faults in this case.

Chapter 5 presents the details of our experimental evaluation. In brief, the three techniques significantly reduce the costs of bounded-exhaustive testing while preserving its fault-finding capabilities.

Chapter 3

Background

This chapter provides the background necessary to present the three techniques introduced in this thesis. We present the ASTGen framework for testing refactoring engines. The three techniques presented in this thesis build upon ASTGen to significantly reduce its costs.

3.1 ASTGen

We now describe in more detail two parts of the ASTGen framework —generators and oracles— that are relevant to present the three techniques introduced in this thesis. The current practice for testing refactoring engines, as seen in the Eclipse and NetBeans test suites, is that developers manually write all the unit tests for their refactorings, including input programs. In contrast, ASTGen allows the testers to write *generators*—pieces of code that implement a specific interface—which ASTGen runs to automatically generate input programs. ASTGen then applies refactorings on these inputs and runs the *oracles* on the outputs to check whether the refactoring was applied correctly.

3.1.1 Generators

Each generator is a piece of Java code that produces elements of Java abstract syntax trees (ASTs), which can be pretty-printed as Java source. Conceptually, generators are close to grammar-based generation [11, 25] where inputs are generated based on the grammar for the input. However, while using grammars, it is difficult to express semantic constraints that span across various parts of the program (for example, constraints that ensure that the

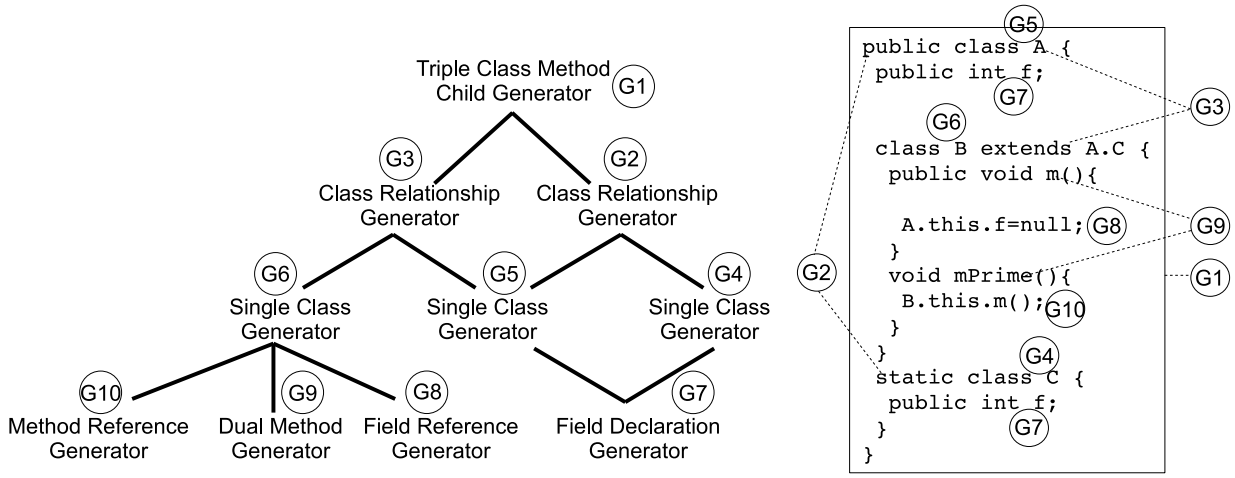

Figure 3.1: Triple Class Method Child Generator structure and a generated test input

generated program is suitable for the application of a specific refactoring). Therefore ASTGen uses Java code rather than a grammar formalism as explained elsewhere [8]. ASTGen provides (1) a large library of basic generators, (2) several mechanisms to compose and link simpler generators into more complex generators, and (3) customization of generators using Java code. Some of the generators that ASTGen provides include:

Field Declaration Generator produces many different field declarations that vary in terms of type (`int`, `byte`, `boolean`, array or non array, etc.), visibility (`private`, `public`, etc.), and name of the declared field.

Field Reference Expression Generator is linked to the Field Declaration Generator and produces different expressions that reference the declared field in various ways, including field accesses and operations (`this.f`, `new A().f`, `super.f`, `f++`, `!f`, etc.).

Single Class Field Reference Generator is composed on top of the Method Declaration Generator and produces classes with one field (obtained from the Field Declaration Generator) and one method that references the field in various ways.

Dual Class Relationship Generator is composed upon generators that produce classes (e.g., Single Class Field Reference Generator) and produces two classes with various

relationships between them (inheritance, inner class, local class, etc.).

While the main purpose of generators is to actually produce the required test inputs, they also encode the space of all inputs to be produced. Consider testing PullUpMethod in the following scenario:

Inputs: Programs with three classes `A`, `B`, and `C`.

- `B` extends `C`; `B` has a method `m` and a method `mPrime` that invokes `m`.
- `C` and `A` each have a field `f` that may be referenced by `m`.
- `B` or `C` is related to `A` in some way.

Test: Pull up method `m` from class `B` to class `C`.

The user can generate all these inputs by writing a generator that composes and links several library generators. Figure 3.1 shows the overall structure of a generator, called Triple Class Method Child Generator, that encodes this input space. The figure also shows a sample test input produced by this generator and how the input sub-parts match the sub-generators responsible for producing them. By iterating through all the variations of the sub-generators, the Triple Class Method Child Generator produces 1,152 test inputs. Note that the ASTGen generators are imperative style generators. They specify *how* the test inputs are to be generated. Hence, they can be directly executed to obtain the inputs.

3.1.2 Oracles

While generators are the core of ASTGen and help testers to produce a large number of input programs for testing refactorings, it would be impractical for the testers to manually check the result of each refactoring application. Oracles automate checking of the results so that the testers only have to inspect a smaller number of tests that fail the oracles (and likely detect faults). ASTGen provides two generic oracles and allows the users to write refactoring-specific oracles:

Compilation Failure Oracle flags tests where the refactored program has a compilation error: if the input program compiles, then the output program should also compile.

Erroneous Warning Oracle flags tests where the refactoring engine raised a warning about a refactoring application, but ignoring that warning results in a refactored program with no compilation errors (or custom failures). Previous work used an oracle that flagged any test that raised a warning [1,8], but we discovered that such an oracle has a large number of false positives.

Custom Oracles are specific to the refactoring being applied. For example, a custom oracle for the RenameMethod refactoring could check that renaming a method, say `m` to `p`, and then renaming back, `p` to `m`, results in the same program.

It is important to note that the output of traditional oracles are only booleans (pass or fail), but the ASTGen oracles can provide additional information about the failure, e.g., messages from the compiler or warnings from the refactoring engine.

Chapter 4

Implementation

This chapter provides further details about the implementation of the three techniques presented in this thesis. As mentioned previously, the three techniques were implemented within the ASTGen framework which was introduced in Chapter 3.

4.1 Sparse Test Generation (STG)

Generators encode and produce all the test inputs within defined bounds, and bounded-exhaustive testing checks the code under test for all these inputs. This usually consumes a large amount of machine time since the number of inputs generated is fairly large. For example, ASTGen generators can generate thousands of test inputs, and it can take hours of machine time to execute the refactorings on all those inputs. Additionally, this time translates into human time required by the developer to wait for the generation and execution of the tests to complete. Note that as soon as a tool reports a failure, the developer can start inspecting it to file a bug report or to debug the fault that caused the failure. In theory, the time the tool takes for generation and testing after the first failure is not important since the developer does not have to idle. For this reason, we consider the Time to First Failure (TTFF) as the key metric in interactive bounded-exhaustive testing. If no generated test input results in a failure, the developer in theory has to wait for the entire generation and testing process to complete.

STG is our technique that aims to reduce the TTFF in cases where failures exist. Figure 4.1 provides the simplified pseudo-code for STG. The `stgMain` method is the entry point.

```
void stgMain(Generator gen, int maxJumpLength) {
  // Sparse generation phase
  while(gen.canGenerateMoreInputs()) {
    int jumpLength = randomBetween(0, maxJumpLength);
    gen.skip(jumpLength);
    if(gen.canGenerateMoreInputs()) {
      generateAndTestOneInput(gen);
    }
  }
  gen.reset();
  // Exhaustive generation phase
  while(gen.canGenerateMoreInputs()) {
    generateAndTestOneInput(gen);
  }
}

void generateAndTestOneInput(Generator gen) {
  Input in = gen.generateNextInput();
  Output out = performRefactoring(in);
  Result res = checkOutputWithOracles(out);
  logAndDisplayResult(res);
}
```

Figure 4.1: Simplified pseudo-code for STG

It accepts two parameters: the generator to be used and the `maxJumpLength` that is described later. The `stgMain` method contains two loops, both of which invoke the helper method `generateAndTestOneInput` that performs the generation and testing of a single input. The two loops contain the pseudo-code for the two phases of STG which are described below:

Sparse Generation is motivated by our observation that failing test inputs are often located close together in the sequence of inputs produced by a generator, and thus, to find a failure, it is often not necessary to exhaustively generate all the inputs, it is sufficient to generate only one input from a closely located group. Therefore, this phase makes "jumps" through the generation sequence. The jump length is not constant (since the failing tests may be in a stride that a constant jump would miss) but *each jump is (uniformly) random within some length limit.* Figure 4.1 shows the simplified pseudo-code for this phase in the first loop of the `stgMain` method.

The key challenge is to determine an appropriate maximum limit for the jump length

(the `maxJumpLength` parameter of the `stgMain` method in Figure 4.1). A lower maximum jump length reduces the number of inputs that are skipped and conversely results in the generation of a greater number of inputs during the Sparse Generation phase. This increases the overhead of STG compared to the basic, *dense* bounded-exhaustive testing which does not include the Sparse Generation phase. However, a higher maximum jump length increases the number of inputs that are skipped and reduces the number of inputs that are generated during the Sparse Generation phase. This decreases the chances of a failure being found during this phase and thus potentially increases the TTFF. In our experiments, we used a maximum jump length of 20 as it provides a good trade-off: the expected jump is of length (1+20)/2, which increases the total time by less than 10% when there is no failure. If Sparse Generation finds a failing test, it usually does so quickly as it significantly reduces the number of tests generated and executed; the results from Section 5 show that STG reduces the TTFF by an order of magnitude in most cases compared to the dense generation. However, STG is a heuristic and, in general, could keep missing failures until the very end while dense generation would have found those failures at the very beginning. So, in theory STG could result in a significantly larger TTFF than the dense bounded-exhaustive testing. For example, consider a fault that can be found only by the second input generated by the dense bounded-exhaustive testing. If the sparse generation phase skips over that second input, the fault wont be found until the second exhaustive phase.

Exhaustive Generation follows Sparse Generation and does basic bounded-exhaustive testing (1) to ensure that a failing test input will be found if one exists and (2) to find all the failing tests that Sparse Generation missed (which can help in clustering failures or debugging [9, 19, 23]). Figure 4.1 shows the simplified pseudo-code for this phase in the second loop of the `stgMain` method.

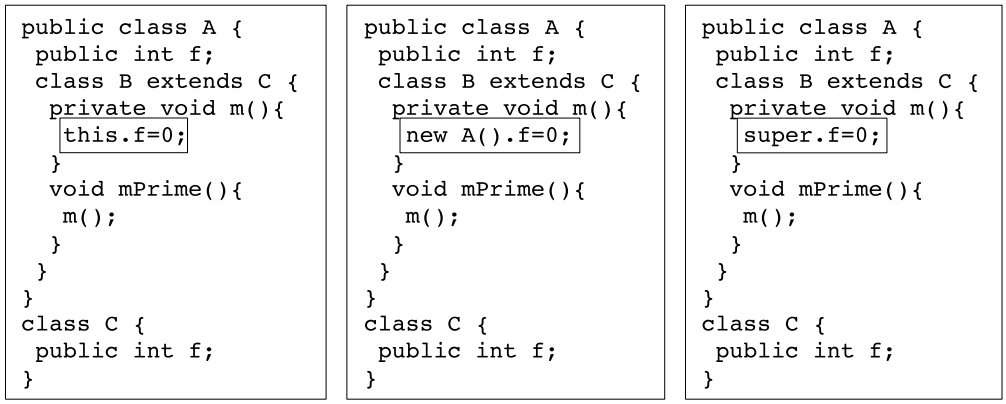

Figure 4.2: Unmerged test inputs

4.2 Structural Test Merging (STM)

While TTFF is an important metric in bounded-exhaustive testing, another important metric is the total time for test generation and execution. This time can be very long when generators produce a large number of inputs, which is the case for typical top-level ASTGen generators. For example, consider the number of inputs for the Triple Class Method Child Generator shown in Figure 3.1. Each of its sub-generators has a small number of variations by itself —G2 has 2 (`inner`, `outer`); G3 has 3 (`inner`, `method inner`, `outer`); G4, G5, and G6 have 1; G7 has 2 (`public`, `private`); G8 has 6 (`this.f`, `new A().f`, `super.f`, etc.); G9 has 4 (`public`, `private`, same/different signature); and G10 has 4 (`m()`, `new B().m()`, `this.m()`, `B.this.m()`)— but the top-level generator produces all combinations of those variations which is $2 \times 3 \times 1 \times 1 \times 1 \times 2 \times 6 \times 4 \times 4 = 1152$ combinations.

STM reduces the number of test inputs while still aiming to preserve their exhaustiveness: instead of producing a large number of small input programs, STM produces a smaller number of larger input programs by *merging* appropriate program elements. For example, the Triple Class Method Child Generator produces the three inputs shown in Figure 4.2. The only difference between the three inputs are the highlighted statements, generated by the Field Reference sub-generator (G8). Figure 4.3 shows an input that contains all these three statements. This single, merged input encodes the same input space as the three unmerged

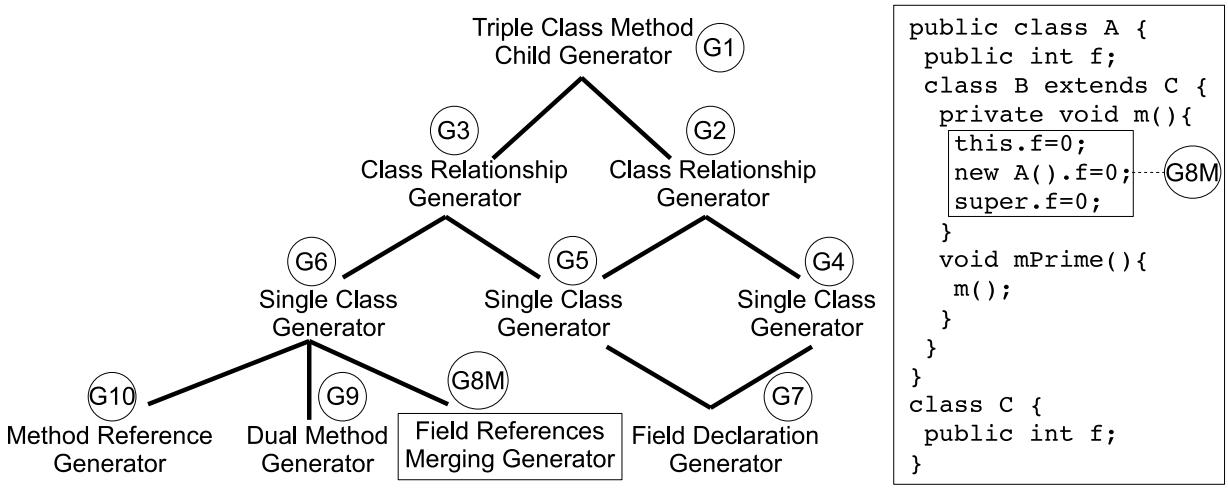

Figure 4.3: Merged generator structure and a generated merged test input

inputs. This structural merging transformation is the crux of our STM technique.

STM exploits the compositional structure of the sub-generators to produce merged test inputs. Figure 4.3 shows an alternative structure for the Triple Class Method Child Generator: a new Field Reference Merging Single Class Generator (G8M) merges together all the program elements produced by the original Field Reference Generator (G8). While figures 3.1 and 4.3 show a generator before and after a *single* application of the structural merging transformation, it is possible to apply the transformation *multiple* times within the hierarchical structure of a generator. Each application leads to a multiplicative reduction in the number of generated inputs. For example, the original Method Reference Generator (G10) can also be modified to a generator G10M that merges together all the different method invocation statements. Together, G8M and G10M produce inputs that merge *both* field references and method references. We refer to the number of transformation applications as *merging level*: for the Triple Class Method Child Generator, merging level M1 has only G8M, and merging level M2 has both G8M and G10M. The unmerged generator produces 1,152 inputs, and levels M1 and M2 reduce the number of inputs to 192 and 48, respectively. The total testing time reduces from 27:02 to 3:57 and 0:47 minutes, respectively; and the TTFF reduces from 9:09 to 1:25 and 0:17 minutes, respectively.

While STM achieves significant time savings, it is important to note its two potential drawbacks. One potential drawback is that larger inputs, through the interference of program elements, can mask some test failures [28, 29]. Consider, for example, merging together all the different field references (as in Figure 4.3). There may be a failure triggered by one of the field reference statements which gets masked by the presence of the other field reference statements. However, this interference can also go the other way: larger inputs may trigger new failures that smaller inputs do not trigger. The other drawback is the effect of larger inputs on debugging. STM produces fewer larger inputs rather than more smaller inputs, but smaller (failing) inputs typically make it easier to perform fault localization. We could take two approaches to address this. One approach is to reduce inputs by applying Delta Debugging [40] on the larger failing input to try to isolate the part of the input that triggers the failure. Another approach, enabled by the fact that larger inputs are produced by merging generators, is to regenerate the individual small inputs that represent the larger failing input by using the generator with no merging (i.e., merging level M0).

4.3 Oracle-based Test Clustering (OTC)

Prior experience with bounded-exhaustive testing in academia and industry shows that it can find faults in real code [8, 20, 35, 36] but also produces a large number of failures. Identifying a few faults out of many failures is a challenging task. OTC is a new technique that helps in this task by splitting failing tests into groups such that all tests in the same group are likely due to the same fault.

OTC exploits information from oracles to group failing tests. Recall that ASTGen oracles provide messages about the failures, e.g., if a refactored program does not compile, ASTGen reports the compilation error provided by the compiler. Initially, we used these messages to cluster the failing tests by grouping together those tests that have *exactly the same messages*. (A test can produce multiple messages, which our experiments compare as lists, not bags

or sets.) However, directly using *concrete* messages provided by the compiler resulted in a large number of small clusters, e.g., two compilation errors may differ only in line or column numbers, say, "3:8:field f not visible" and "2:6:field f not visible". So we experimented with *abstract* messages that ignore some details such as line and column numbers. Going further in that direction, one can even consider ignoring the messages and clustering based on *which* oracle failed, *not how* it failed. The trade-off is that creating too many clusters increases inspection effort, while creating too few clusters increases the chances of missing a fault. Our evaluation compares four clustering options: Concrete Message, Abstract Message, Oracle Name, and Random Selection (which is the base case with no clustering); the results show that Abstract Message provides the best trade-off.

Chapter 5

Evaluation

We evaluated our three new techniques with the ASTGen framework for bounded-exhaustive testing of refactorings engines. More specifically, we evaluated the three techniques by applying them to test 6 refactorings in the Eclipse refactoring engine using the 9 generators listed in Table 5.1. For each generator and several merging levels, we tabulate the number of inputs generated, various times and APFD metric (described below), the number of failing inputs, and the number of distinct faults. Previous work tested these refactorings with these generators and found a number of faults [8]. The goal of this study was to evaluate whether the new techniques reduce the testing costs, and not to find new faults. However, due to our use of OTC, we also found a new fault in the PushDownMethod refactoring, previously missed [8] due to random sampling of failures that were inspected.

5.1 Sparse Test Generation (STG)

Table 5.1 shows the time results for ASTGen with and without STG. The 'Dense' subcolumns show the total time and time to first failure (TTFF) for bounded-exhaustive testing without STG. If no failure exists, TTFF shows 'n/a'. The 'Sparse' column shows average values for TTFF if a failure exists (roughly the top half of the table) and the total time if no failure exists (the bottom half of the table). These times are averaged over 20 random seeds, with the jump limit of 20 as discussed in Section 4.1. The main questions about STG are how it affects TTFF and the total time.

STG reduces TTFF in all cases where the dense TTFF was significant (a minute or

Refactoring	Generator	ML	Inputs	Dense		Sparse	APFD [%]		Failures	Faults
				Time	TTFF		Dense	Sparse		
PushDown-Field	DualClass-FieldReference	M0	7416	133:32	7:33	0:47	74.98	98.16	1074	2
		M1	1236	22:43	0:01	0:02	99.23	88.62	179	1
		M2	12	0:21	0:00	0:01	95.83	74.17	4	1
		M3	3	0:13	0:00	0:00	83.33	63.33	1	1
Encapsulate-Field	DualClass-FieldReference	M0	23760	427:09	73:34	7:14	58.03	97.59	486	3
		M1	3960	71:50	12:03	1:11	69.82	97.77	354	3
		M2	72	1:19	0:13	0:03	74.31	80.56	31	2
		M3	18	0:26	0:06	0:03	58.33	73.15	8	2
	SingleClass-FieldReference	M0	8576	155:15	0:22	0:03	75.37	97.61	836	4
		M1	2144	39:04	0:21	0:03	66.86	97.59	242	4
		M2	1072	19:35	0:09	0:02	84.25	93.04	144	3
		M3	268	4:55	0:02	0:02	72.70	88.11	62	3
		M4	16	0:17	0:00	0:01	96.88	84.06	1	1
PushDown-Method	DualClass-MethodParent	M0	960	22:19	11:28	1:05	43.91	93.59	180	3
		M1	192	4:07	2:07	0:14	41.75	91.89	38	3
		M2	48	0:45	0:28	0:21	40.63	87.85	2	1
PullUp-Method	TripleClass-MethodChild	M0	1152	27:02	9:09	1:01	13.19	95.77	160	2
		M1	192	3:57	1:25	0:09	48.18	95.36	96	2
		M2	48	0:47	0:17	0:02	56.25	89.58	24	2
PullUp-Method	DualClass-MethodChild	M0	576	13:22	n/a	14:14	n/a	n/a	0	0
		M1	96	1:49	n/a	1:55	n/a	n/a	0	0
		M2	24	0:21	n/a	0:22	n/a	n/a	0	0
Rename-Field	DualClass-FieldReference	M0	23760	629:01	n/a	689:17	n/a	n/a	0	0
		M1	3960	107:26	n/a	117:48	n/a	n/a	0	0
		M2	72	1:56	n/a	2:04	n/a	n/a	0	0
		M3	18	0:34	n/a	0:34	n/a	n/a	0	0
	SingleClass-FieldReference	M0	8576	229:00	n/a	250:59	n/a	n/a	0	0
		M1	2144	57:28	n/a	62:56	n/a	n/a	0	0
		M2	1072	28:44	n/a	31:28	n/a	n/a	0	0
		M3	268	7:15	n/a	7:57	n/a	n/a	0	0
Rename-Method	SingleClass-MethodReference	M0	9540	173:32	n/a	190:11	n/a	n/a	0	0
		M1	4900	89:26	n/a	98:05	n/a	n/a	0	0
		M2	140	2:37	n/a	2:50	n/a	n/a	0	0
		M3	80	1:31	n/a	1:37	n/a	n/a	0	0

Table 5.1: Sparse Test Generation and Structural Test Merging Results
Legend: ML = Merging Level, TTFF = Time to First Failure, All times in minutes:seconds

more): the speedup ranges from 9.00x (for the PullUpMethod refactoring and the TripleClassMethodChild generator with merging level M0) to 10.58x (for the PushDownMethod refactoring and the DualClassMethodParent generator with merging level M0). The average speed up was an order of magnitude (10x). In a few cases with very small dense TTFF, STG had a slowdown of at most 1 second. Recall the two phases of STG described in Section 4.1; the reduction in TTFF implies that in these cases the first, sparse generation phase found a failure before the second, exhaustive phase.

STG increases the total time, as expected. With the jump limit of 20, the overhead of the additional sparse phase is expected to be slightly under 10% of the total time for dense generation. Our experiments confirm that this is indeed the case: the slowdown ranges from

Refactoring	Generator	ML	Random		Oracle		Abstract		Concrete	
			FD	NC	FD	NC	FD	NC	FD	NC
PushDownField	DualClassFieldReference	M0	1.99	1	2	2	2	5	2	68
		M1	1	1	1	1	1	3	1	59
		M2	1	1	1	1	1	2	1	4
		M3	1	1	1	1	1	1	1	1
EncapsulateField	DualClassFieldReference	M0	2.24	1	2.24	2	3	4	3	51
		M1	2.05	1	2.31	2	3	4	3	112
		M2	1.73	1	2	2	2	4	2	8
		M3	1.75	1	2	2	2	3	2	3
	SingleClassFieldReference	M0	2.84	1	3.11	2	4	4	4	71
		M1	2.48	1	2.46	2	4	4	4	73
		M2	2.26	1	2.26	1	3	4	3	58
		M3	2.30	1	2.30	1	3	5	3	24
		M4	1	1	1	1	1	1	1	1
PullUpMethod	TripleClassMethodChild	M0	1.77	1	1.77	1	2	3	2	127
		M1	1.56	1	1.56	1	2	2	2	84
		M2	1.62	1	1.62	1	2	2	2	24
PushDownMethod	DualClassMethodParent	M0	2.19	1	3	2	3	11	3	20
		M1	2.56	1	2.54	2	3	10	3	16
		M2	1	1	1	1	1	1	1	2

Table 5.2: Oracle-based Test Clustering Results
Legend: ML = Merging Level, FD = Faults Detected, NC = Number of Clusters

5.50% (for the PullUpMethod refactoring and the DualClassMethodChild generator with merging level M1), to 9.67% (for the RenameMethod refactoring and the SingleClassMethodReference generator with merging level M1). In summary, STG achieves a 10x speedup in TTFF for only a 10% slowdown in the total time.

We further evaluated STG using the Average Percentage Fault Detected (APFD) metric introduced by Rothermel et al. [31] to compare techniques for test prioritization and extended by Walcott et al. [37] for test selection. APFD measures the number of faults detected in terms of the number of tests executed, whereas TTFF is based on the first failure (not all faults) and actual time (not number of tests) as TTFF aims to capture the waiting time for testers in interactive bounded-exhaustive testing, similar to more recent extensions of APFD [10]. APFD ranges between 0 and 100%, with higher values being better. Table 5.1 shows APFD, with 'Sparse' averaged over 20 random seeds. The results show that STG improves APFD in all cases where the dense TTFF was significant.

5.2 Structural Test Merging (STM)

Table 5.1 shows the results for STM for several merging levels of each of the generators. The merging level number (e.g., 3 in M3) represents the number of structural merging transformations applied to the unmerged generator (labeled M0) to obtain the corresponding merged generator, as discussed in Section 4.2. The main questions about STM are how it affects times (total and TTFF) and the number of failures/faults detected.

Each merging level reduced both the total time and TTFF compared to its previous level and thus to M0. Level M1 achieved total time speedup ranging from 1.94x (for the RenameMethod refactoring and the SingleClassMethodReference generator), to 7.36x (for the PullUpMethod refactoring and the DualClassMethodChild generator). The average total time speedup for level M1 was 5x. Compared to level M0, level M2 achieved total time speedup ranging from 7.93x (for the EncapsulateField refactoring and the SingleClassFieldReference generator), to 381.52x (for the PushDownField refactoring and the DualClassFieldReference generator). The average total time speedup for level M2 was 130x. The merged generators also substantially reduced the TTFF. Level M1 achieved TTFF speedup ranging from 1.05x (for the EncapsulateField refactoring and the SingleClassFieldReference generator), to 453x (for the PushDownField refactoring and the DualClassFieldReference generator). On average, level M1 achieved 80x TTFF speedup. Level M2 achieved TTFF speedup ranging from 2.44x (for the EncapsulateField refactoring and the SingleClassFieldReference generator), to 453x (for the PushDownField refactoring and the DualClassFieldReference generator). On average, level M2 generators achieved 150x TTFF speedup.

Merging did not expose any new faults, but aggressive merging did mask some faults. In particular, level M1 masks only one fault (in PushDownField), but levels M2 and higher mask a much larger number of faults. However, even the highest level of merging finds at least one fault (when there is a fault at M0). Additionally, if one considers TTFF as the most important metric, masking faults at the higher merging levels is not detrimental but

actually beneficial: the user can start the exploration from a high level, quickly find failures, and start inspecting them, while the tool continues the exploration at a lower level. In summary, STM can substantially improve total time and TTFF while somewhat reducing the fault-detection capability of bounded-exhaustive testing.

5.3 Oracle-based Test Clustering (OTC)

Table 5.2 shows the results for the four clustering options discussed in Section 4.3. For each option, we present the number of clusters formed and distinct faults detected by inspecting a number of randomly selected tests from each cluster. The results are averaged over 1000 random seeds. For this experiment, we needed to choose a sampling strategy [9], which determines how many tests to select and from which clusters. The basic strategy, one-per-cluster, selects one test for each cluster; we used this strategy for Abstract Message and Concrete Message. For Random Selection and Oracle Name, which have fewer clusters, we used a strategy that selects more tests per cluster, specifically selects at least as many tests as Abstract Message selects (i.e., the number of clusters that Abstract Message has) and at most 1% of all failing tests. The main questions about OTC are how it affects the number of failures that need to be inspected and the number of faults detected.

To measure the number of faults detected by a set of selected tests, we had to map failing tests to the fault(s) they detect and also had to determine which faults are *distinct*. We performed two steps. First, we manually inspected all tests from each cluster based on Abstract Message with less than 30 tests and inspected at least 10 tests from each cluster with more than 30 tests. Since all inspected tests from each cluster detected the same fault(s), we extrapolated that all tests in a cluster can detect the same fault(s). We also patched 6 of these faults in Eclipse and confirmed their results from the first step. Second, we asked a researcher, Danny Dig (unaware of the details of this study but with multiple years of experience with Eclipse refactorings) to label the faults collected in the first step as

potential duplicates of each other or non-faults. This resulted in 12 distinct faults that we used in our experiments.

Abstract Message substantially reduces the number of tests to be inspected to find all the faults, e.g., PullUpMethod for M0 has 160 failing tests, but Abstract Message splits them into 3 clusters, and selecting any 3 tests, one from each cluster, always reveals all 2 faults. The results show that Abstract Message finds *all faults* that Concrete Message finds but requires inspection of much fewer tests, up to over an order of magnitude for lower merging levels. Also, Abstract Message finds *more faults* than Random Selection and Oracle Name while the same number or even fewer tests are inspected. In summary, Abstract Message was the most effective OTC option among the four we compared.

Chapter 6

Related Work

There is a large body of work on automated testing. Our focus is on bounded-exhaustive testing [7, 8, 20, 26, 35, 36] that tests the code for all inputs within given bounds. Bounded-exhaustive testing has been successfully used to reveal faults in several real applications [8, 20, 35, 36], but it has substantial costs in machine time for test generation and execution and human for inspection of failures. This thesis presents three new techniques that reduce the costs of such testing.

Sparse Test Generation (STG) is related to work on test selection/reduction/prioritization [12, 13, 16, 21, 30, 31, 34, 37–39] whose goal is to reduce the testing cost or to find faults faster by selecting a subset of tests from a test suite and/or ordering these tests. The previous techniques mostly consider regression testing where a test suite exists a priori, and the simplest techniques can randomly select or order these tests. In contrast, STG selects tests while they are being generated, and generation proceeds in a particular order (e.g., in AST-Gen depending on the state of generators), so arbitrary random sampling is not possible. Finally, STG does not compromise the fault-finding ability [16].

Structural Test Merging (STM) is related to work on test granularity [28, 29] which studied the cost-benefit trade-offs in testing with a larger number of smaller tests versus a smaller number of larger tests. The key difference is that previous work considered tests that can be easily *appended* while we consider tests that need to be *merged*. Note that appending tests only saves setup and teardown costs [29], while merging can also reduce test execution cost (e.g., merging 1,152 input programs into 192 input programs requires only 192 applications of the PullUpMethod refactoring). However, the results are similar in both

contexts: larger tests reduce the testing time, but too large tests may miss faults.

Oracle-based Test Clustering (OTC) is related to work on test clustering/filtering/indexing [9, 19, 22, 23, 27, 32]. Previous work performed clustering based on *execution profiles*, obtained from monitoring test execution. The main novelty of our technique is to exploit information-rich oracles, rather than execution profiles, to cluster failing tests. Our goal is to cluster failing tests to help in identifying the underlying faults. Dickinson et al. [9] present an empirical study that evaluates somewhat different techniques whose goal is to find failures among executions by using cluster analysis of execution profiles. Effectively, those techniques use cluster analysis as approximate oracles. Their results show that cluster filtering of executions can find failures more effectively than random sampling, and that clustering of executions can distinguish failing executions from passing ones.

Finally, note that the work presented in this thesis has already been published in the form of a conference paper at the 12th International Conference on Fundamental Approaches to Software Engineering [18].

Chapter 7

Conclusions and Future Work

Bounded-exhaustive testing checks the code under test for all inputs within given bounds. It can find faults but at potentially high costs, including machine time to generate and run tests, and human time to wait for the test results and to inspect failures to identify distinct faults. We presented three new techniques that reduce these costs: Sparse Test Generation skips some tests to reduce the time to first failure by an order of magnitude; Structural Test Merging generates larger tests to reduce test generation and execution time by order(s) of magnitude; and Oracle-based Test Clustering groups failing tests to reduce the inspection time by order(s) of magnitude.

While the techniques presented in this thesis reduce the major costs of bounded-exhaustive testing, one cost that is harder to quantify has been ignored in this thesis. This cost is the human time required to implement the generators that exhaustively produce the test inputs. The ASTGen generators discussed in this thesis are implemented in an imperative/generating style where the user directly writes *how* to generate the inputs. In contrast, other bounded-exhaustive testing frameworks, such as Korat [7], enable the implementation of generators in a declarative/filtering style where the user writes *what* properties the generated inputs should satisfy. Experience has shown that certain properties of inputs are easier to express using an imperative/generating style while other properties of inputs are easier to express using a declarative/filtering style. Our research group has since worked on allowing users to combine both these styles to enable more compact and efficient implementation of generators through the UDITA [4, 15] language for test generation. The three techniques presented in this thesis have not been evaluated using UDITA; however, we expect the re-

sults to be similar to what we have observed using ASTGen. We leave this evaluation as future work.

The techniques presented in this thesis were evaluated by applying them on the bounded-exhaustive testing of the Eclipse refactoring engine. However, previous work [8] had applied bounded-exhaustive testing on both the Eclipse and NetBeans refactoring engines. Therefore, it would be interesting to expand the evaluation of the techniques by also applying them on the bounded-exhaustive testing of the NetBeans refactoring engine. It would also be interesting to apply bounded-exhaustive testing along with the techniques presented in this thesis in other domains like compilers and type checking systems.

References

[1] *ASTGen home page*, `http://mir.cs.illinois.edu/astgen/`.

[2] *Eclipse project*, `http://www.eclipse.org`.

[3] *NetBeans*, `http://netbeans.org`.

[4] *UDITA home page*, `http://mir.cs.illinois.edu/udita/`.

[5] Kent Beck, *Extreme programming explained: Embrace change*, 2000.

[6] Boris Beizer, *Software testing techniques*, 1990.

[7] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov, *Korat: automated testing based on Java predicates*, International Symposium on Software Testing and Analysis (ISSTA), 2002.

[8] Brett Daniel, Danny Dig, Kely Garcia, and Darko Marinov, *Automated testing of refactoring engines*, European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), 2007.

[9] William Dickinson, David Leon, and Andy Podgurski, *Finding failures by cluster analysis of execution profiles*, International Conference on Software Engineering (ICSE), 2001.

[10] Hyunsook Do and Gregg Rothermel, *An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-benefit models*, European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), 2006.

[11] Arthur Gibson Duncan and John S. Hutchison, *Using attributed grammars to test designs and implementations*, International Conference on Software Engineering (ICSE), 1981.

[12] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel, *Incorporating varying test costs and fault severities into test case prioritization*, International Conference on Software Engineering (ICSE), 2001.

[13] Emelie Engström, Per Runeson, and Mats Skoglund, *A systematic review on regression test selection techniques*, Information & Software Technology (2010).

[14] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the design of existing code*, 1999.

[15] Milos Gligoric, Tihomir Gvero, Vilas Jagannath, Sarfraz Khurshid, Viktor Kuncak, and Darko Marinov, *Test generation through programming in UDITA*, International Conference on Software Engineering (ICSE), 2010.

[16] Mats Per Erik Heimdahl and George Devaraj, *Test-suite reduction for model based tests: Effects on test quality and implications for testing*, IEEE International Conference on Automated Software Engineering (ASE), 2004.

[17] Daniel Jackson, *Software abstractions: Logic, language and analysis*, 2006.

[18] Vilas Jagannath, Yun Young Lee, Brett Daniel, and Darko Marinov, *Reducing the costs of bounded-exhaustive testing*, International Conference on Fundamental Approaches to Software Engineering (FASE), 2009.

[19] James A. Jones, Mary Jean Harrold, and James F. Bowring, *Debugging in parallel*, International Symposium on Software Testing and Analysis (ISSTA), 2007.

[20] Sarfraz Khurshid and Darko Marinov, *TestEra: Specification-based testing of Java programs using SAT*, Automated Software Engineering Journal (2004).

[21] Jung-Min Kim and Adam Porter, *A history-based test prioritization technique for regression testing in resource constrained environments*, International Conference on Software Engineering (ICSE), 2002.

[22] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff, *Sober: statistical model-based bug localization*, European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), 2005.

[23] Chao Liu, Xiangyu Zhang, Jiawei Han, Yu Zhang, and Bharat K. Bhargava, *Indexing noncrashing failures: A dynamic program slicing-based approach*, IEEE International Conference on Software Maintenance (ICSM), 2007.

[24] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard, *An evaluation of exhaustive testing for data structures*, Tech. report, MIT CSAIL, 2003.

[25] Peter M. Maurer, *Generating test data with enhanced context-free grammars*, IEEE Software (1990).

[26] Sasa Misailovic, Aleksandar Milicevic, Nemanja Petrovic, Sarfraz Khurshid, and Darko Marinov, *Parallel test generation and execution with Korat*, European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), 2007.

[27] Andy Podgurski, David Leon, Patrick Francis, Wes Masri, Melinda Minch, Jiayang Sun, and Bin Wang, *Automated support for classifying software failure reports*, International Conference on Software Engineering (ICSE), 2003.

[28] Gregg Rothermel, Sebastian Elbaum, Alexey Malishevsky, Praveen Kallakuri, and Brian Davia, *The impact of test suite granularity on the cost-effectiveness of regression testing*, International Conference on Software Engineering (ICSE), 2002.

[29] Gregg Rothermel, Sebastian Elbaum, Alexey G. Malishevsky, Praveen Kallakuri, and Xuemei Qiu, *On test suite composition and cost-effective regression testing*, ACM Transactions on Software Engineering Methodology (2004).

[30] Gregg Rothermel and Mary Jean Harrold, *A safe, efficient regression test selection technique*, ACM Transactions on Software Engineering Methodology (1997).

[31] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold, *Test case prioritization: An empirical study*, IEEE International Conference on Software Maintenance (ICSM), 1999.

[32] Per Runeson, Magnus Alexandersson, and Oskar Nyholm, *Detection of duplicate defect reports using natural language processing*, International Conference on Software Engineering (ICSE), 2007.

[33] David Saff and Michael D. Ernst, *Reducing wasted development time via continuous testing*, IEEE International Symposium on Software Reliability (ISSRE), 2003.

[34] Amitabh Srivastava and Jay Thiagarajan, *Effectively prioritizing tests in development environment*, International Symposium on Software Testing and Analysis (ISSTA), 2002.

[35] Keith Stobie, *Model based testing in practice at Microsoft*, Electronic Notes in Theoretical Computer Science **111** (2005), 5–12.

[36] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson, *Software assurance by bounded exhaustive testing*, International Symposium on Software Testing and Analysis (ISSTA), 2004.

[37] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos, *Time-aware test suite prioritization*, International Symposium on Software Testing and Analysis (ISSTA), 2006.

[38] Shin Yoo and Mark Harman, *Regression testing minimisation, selection and prioritisation: A survey*, Journal of Software Testing, Verification and Reliability (2010).

[39] Yanbing Yu, James A. Jones, and Mary Jean Harrold, *An empirical study of the effects of test-suite reduction on fault localization*, International Conference on Software Engineering (ICSE), 2008.

[40] Andreas Zeller and Ralf Hildebrandt, *Simplifying and isolating failure-inducing input*, IEEE Transactions on Software Engineering (2002).