The Dissertation Committee for Chang Hwan Peter Kim
certifies that this is the approved version of the following dissertation:

Systematic Techniques for Efficiently Checking Software Product Lines

Committee:

Don Batory, Supervisor

Sarfraz Khurshid, Co-Supervisor

William Cook

Darko Marinov

Vitaly Shmatikov

Systematic Techniques for Efficiently Checking Software Product Lines

by

Chang Hwan Peter Kim, BASc., MASc.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2013

To my family

Acknowledgments

First and foremost, I would like thank my academic parents, Professor Don Batory and Professor Sarfraz Khurshid. It was an honour for me to learn from researchers who are world experts in automated software engineering. They taught me what good research is, how to conduct research, and how to present ideas and results. They were also great advisors. During my studies, they were very patient with me, steered me in the right direction, and most importantly, encouraged me to keep going. In particular, there were countless times when I had either nothing to present or negative results for our weekly meetings and I absolutely dreaded entering their offices, but every time, I came out feeling encouraged and motivated. Also, I, and my actual parents, don't know how they were able to put up with me for over 5 years, but they did. I will forever be grateful for all they have done for me. Thank you, sirs.

I would also like to thank Professor William Cook, Professor Darko Marinov and Professor Vitaly Shmatikov for serving on my committee. I had the pleasure of discussing programming languages philosophy and research with and receiving advice from Professor Cook. I had the fortune of collaborating with Professor Marinov on a paper on dynamically pruning configurations to test during his sabbatical at UT-Austin and Groupon. I am very grateful because he not only proposed the idea behind the paper, but also was involved in implementation and evaluation. Professor Shmatikov not only served on my Research Preparation Exam (RPE) and

dissertation committees, but was also instrumental in helping me secure postdoctoral positions. Needless to say, I am indebted and extremely grateful.

I would like to thank UT-Austin and funding agencies in Canada and United States for supporting my PhD studies. In particular, my advisors and I were supported by a Natural Science and Engineering Research Council (NSERC) Postgraduate Scholarship, NSF (National Science Foundation) Science of Design Project CCF-0724979, NSF CCF-0845628, IIS-0438967, CNS-0958231. and AFOSR grant FA9550-09-1-0351. In addition, my collaborators were supported by CASED, CCF-1213091, CCF-1212683, CNS-0958199 and CCF-0746856.

Austin was my home for over 5 years. I will never forget watching countless soccer matches and drinking pints at various bars, frequenting Starbucks to the point where I should be paying rent, freezing in air-conditioned buildings that would put Canadian winter to shame, and lining up at Torchy's Tacos. Thank you, Austin, Texas and my friends, for the memories.

My family was very supportive of my pursuit of the PhD, offering to help any way they can. But more importantly, no matter where I was or what I was doing, they always just wanted to know if I was healthy and happy. Their health and happiness are what matter to me the most also. Thank you for your love and support.

Last but not least, I thank the Lord for letting me complete my studies. May He give us the strength to fight for what is right and live a life devoted to helping others. In the name of the Father, the Son and the Holy Spirit. Amen.

Chang Hwan Peter Kim

The University of Texas at Austin

December 2013

Systematic Techniques for Efficiently Checking Software Product Lines

Publication No. _____

Chang Hwan Peter Kim, Ph.D.
The University of Texas at Austin, 2013

Supervisor: Don Batory
Co-Supervisor: Sarfraz Khurshid

A *Software Product Line* (SPL) is a family of related programs, which of each is defined by a combination of *features*. By developing related programs together, an SPL simultaneously reduces programming effort and satisfies multiple sets of requirements. Testing an SPL efficiently is challenging because a property must be checked for all the programs in the SPL, the number of which can be exponential in the number of features.

In this dissertation, we present a suite of complementary static and dynamic techniques for efficient testing and runtime monitoring of SPLs, which can be divided

into two categories. The first prunes programs, termed *configurations*, that are irrelevant to the property being tested. More specifically, for a given test, a static analysis identifies features that can influence the test outcome, so that the test needs to be run only on programs that include these features. A dynamic analysis counterpart also eliminates configurations that do not have to be tested, but does so by checking a simpler property and can be faster and more scalable. In addition, for runtime monitoring, a static analysis identifies configurations that can violate a safety property and only these configurations need to be monitored.

When no configurations can be pruned, either by design of the test or due to ineffectiveness of program analyses, runtime similarity between configurations, arising due to design similarity between configurations of a product line, is exploited. In particular, *shared execution* runs all the configurations together, executing bytecode instructions common to the configurations just once. *Deferred execution* improves on shared execution by allowing multiple memory locations to be treated as a single memory location, which can increase the amount of sharing for object-oriented programs and for programs using arrays.

The techniques have been evaluated and the results demonstrate that the techniques can be effective and can advance the idea that despite the feature combinatorics of an SPL, its structure can be exploited by automated analyses to make testing more efficient.

Contents

Chapter 1

Introduction

Despite advances in software engineering, programmers must write more code than ever before in order to satisfy requirements that are growing not only in number, but also in complexity, and to satisfy customers who want the luxury of a tailor-made solution. Another problem is that the highly competitive nature of the software market means that companies must satisfy as many customers as possible in as little time as possible.

A solution to these two problems is the *Software Product Line (SPL)* paradigm, which develops a set of related programs together to reduce programming *and* satisfy multiple customers simultaneously. An SPL is a family of related programs in which each program consists of code fragments defined by a unique combination of *features*. An SPL allows structured development of common and different code fragments between the programs.

Because an SPL can represent many programs, a number exponential in the number of features, checking a property against an SPL is challenging. In particular, checking each program in the SPL, for example by running each program against a given test from start to finish, is expensive and may not even be feasible. But to ensure the correctness of every program, every program must be checked.

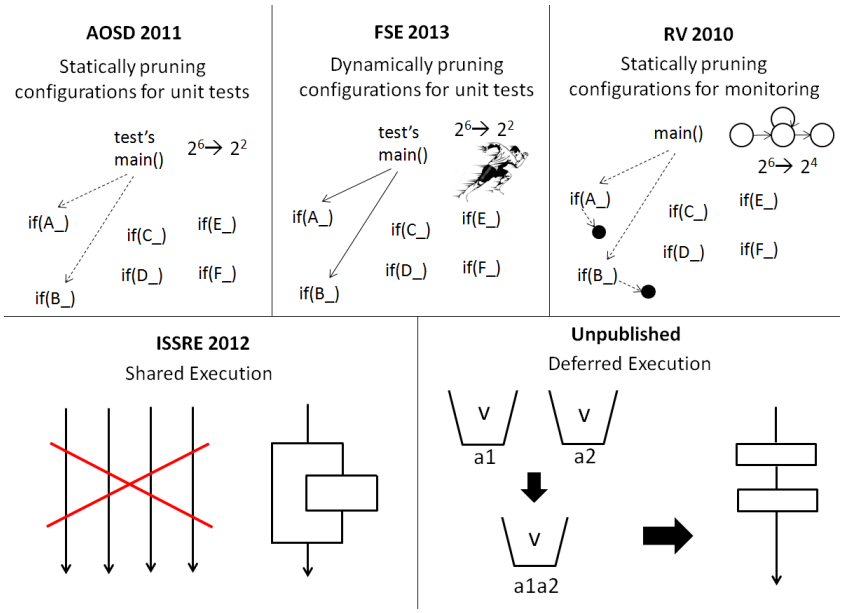


Figure 1.1: Thesis Overview

1.1 Dissertation Overview

Our thesis is that an SPL's feature-oriented structure can be exploited by automated techniques based on static and dynamic analyses to reduce the execution space without introducing a prohibitive overhead. We present a suite of complementary techniques that embodies this idea. The suite can be divided into two categories: three for pruning configurations and three for pruning bytecode instructions between configurations when configurations cannot be pruned. Figure 1.1 visually describes them, where the top three techniques prune configurations and the bottom two prune bytecode instructions. Here is a brief description of them from top left to bottom right, each of which is presented in a separate chapter and has appeared in the proceedings that introduce each paragraph:

- **AOSD 2011 [58], Chapter 3.** A unit test for a conventional program tests a small portion of the program. Similarly, a unit test for an SPL tests a

small portion the SPL, meaning that only a small number of features is likely to be reachable from the test. A static analysis was developed to identify such *reachable* features and further identify a subset of these features whose presence or absence can alter the test outcome. The test then needs to be run only on combinations of these *relevant* features (the diagram shows that only two features are relevant, so the test needs to be run at most 2^2 times).

- **FSE 2013 [62], Chapter 4.** Much of the effectiveness of the AOSD 2011 result was due to identifying reachable features, rather than identifying the relevant features, which are much more expensive to determine as their identification requires data-flow and control-flow analyses. FSE 2013 presents *SPLat*, a dynamic analysis alternative to AOSD 2011, which is in many cases faster and more scalable for determining reachable features. The key insight behind `SPLat` is that the reachable features can be determined *during* execution by modifying an existing test input generator for data structures called *Korat* [18].

- **RV 2010 [59], Chapter 5.** A safety property for an SPL must be checked against every program of the SPL. But if certain programs can never violate the safety property, then these programs do not need to be monitored for violation. This chapter presents a way to statically identify such programs or conversely, the programs that must be monitored. The technique determines instrumentations that must be executed for the property to be violated (shown by the black dots) and then identifies features that allow the instrumentations to be executed. Only programs with these features need to be monitored (in the diagram, there are two such features and four configurations with these two features present).

- **ISSRE 2012 [60], Chapter 5.** The previous three techniques prune config-

urations, but sometimes (e.g. when every configuration needs to be tested by design), every configuration must be tested. In this situation, we can still do better than just running the test from start to finish for each configuration. Because a product line's programs are similar by design, they are likely to be behaviorally similar as well. *Shared execution* executes all the configurations together, executing bytecode instructions common between configurations just once, splitting where executions differ and merging back to resume sharing execution.

- **Deferred execution, unpublished.** *Deferred execution* improves on shared execution by allowing multiple memory locations to be treated as a single memory location, which can increase the amount of sharing. When shared execution and conventional execution's running times grow proportionally to growth in the number of configurations due to use of different memory locations, deferred execution's running time only grows by a constant factor and can even remain constant.

These techniques have been evaluated and the results demonstrate that they can be effective and can advance the idea that despite the feature combinatorics of an SPL, its structure can be exploited by automated analyses to make testing more efficient.

1.2 Contributions

This dissertation makes the following contributions:

- **Techniques for pruning configurations.** These statically prune configurations to test (AOSD 2011 [58], Chapter 3), dynamically prune configurations to test (FSE 2013 [62], Chapter 4) or statically prune configurations to monitor (RV 2010 [59], Chapter 5).

- **Techniques for pruning bytecode instructions between configurations.** When configurations cannot be pruned for a test, all the configurations can be run together through shared execution (ISSRE 2012 [60], Chapter 6) or deferred execution (unpublished, Chapter 7).

- **Implementation.** Statically pruning configurations to test (AOSD 2011 [58], Chapter 3) is implemented as an Eclipse plugin that uses Soot [85], a popular static analysis framework for Java, and SAT4J [86], an off-the-shelf SAT solver. `SPLat` for dynamically pruning configurations to test (FSE 2013 [62], Chapter 4) is implemented for Java by extending `Korat` [18], a tool for generating structurally complex test inputs, and using SAT4J [86]. It is also implemented for Ruby on Rails, but by Darko Marinov, one of the co-authors of the FSE 2013 publication [62]. Statically pruning configurations to monitor (RV 2010 [59], Chapter 5) is implemented within the CLARA framework for partially evaluating runtime monitors [12], as an extension to Bodden et al.'s earlier whole-program analysis [32]. Shared execution (ISSRE 2012 [60], Chapter 6) and deferred execution are implemented on top of *Java PathFinder (JPF)*[83], a model checker for Java that can also function as an easy-to-extend, off-the-shelf VM.

- **Evaluation.** All techniques except customizable multiexecution have been evaluated. Java SPLs, some of which have been used by other research groups to evaluate their verification techniques, were used. Java tests and safety properties were manually constructed by us due to lack of availability. For `SPLat`, Darko Marinov evaluated the Ruby on Rails implementation against a large configurable system (with over 171KLOC in Ruby on Rails). The system uses over 170 configuration variables and contains over 19K tests (with over 231KLOC in Ruby on Rails). To the best of our knowledge, this is the largest and most complex industrial codebase used in research on testing

SPLs [4, 29, 51, 58, 60, 82], and `SPLat` scales to this size without much engineering effort.

Chapter 2

Background

This chapter presents SPL concepts common to the following chapters.

2.1 Features and Feature Model

A *feature* represents an end-user functionality that can be included or excluded from a program of a product line. For our purposes, it can simply be seen as a variable or an option whose value influences code selection. Although different types of features are allowed, such as string, number, and boolean, features in this dissertation are restricted to boolean options. A configuration is an n-digit boolean $f_1...f_n$ representing feature assignments for a product line with n features. 2^n configurations are possible, but a *feature model*, which defines the legal *feature combinations* or *configurations*, typically reduces that number. There are different ways to represent a feature model, but in this dissertation, a feature model is represented as a combination of context-sensitive grammar and boolean propositional logic constraints. For example, Figure 2.1 shows a sample feature model with four *optional* features (shown in square brackets), which may have `true` or `false`, and one *mandatory* feature (`Base`), which has the value `true`. The order of the features and `Product-`

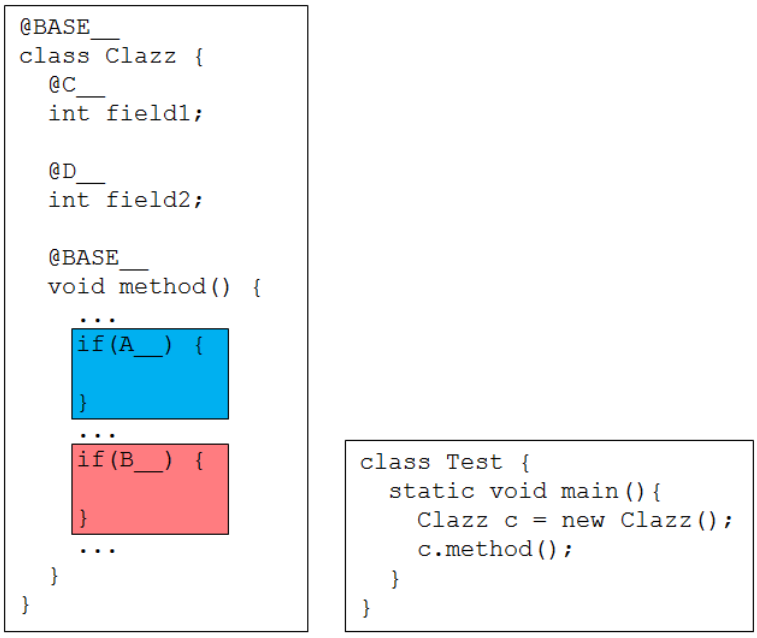
```
1 ProductLine :: A B C D Base;
2 A or B or C or D;
```

Figure 2.1: Sample Feature Model

`Line` symbol does not matter for our purposes. The constraint requires one of the optional features to be present, which eliminates the configuration where all features are absent and therefore makes the number of configurations in the SPL $2^4 - 1 = 15$.

2.2 Mapping Features to Code

To add or remove functionality, a feature must be able to control the presence or absence of a software artifact. In this dissertation, we consider a software artifact to be Java code fragment. One possible control mechanism is *conditional compilation*, i.e. the presence/absence of feature can determine syntactic presence/absence of a code fragment, as is done with `#ifdefs` in `C`. We use a more practical mechanism where control is achieved through ordinary conditional statements. More specifically, for each feature, there is a *feature variable*, which is a static boolean field whose value, for one configuration, is determined at the beginning of program execution and must remain fixed throughout the execution. A sample product line code is shown in Figure 2.2(a). Note that a feature variable (e.g. `A__` or `B__`, which correspond to features `A` and `B` respectively) can only appear in code through an if-statement as shown. Also, note that for the techniques in Chapter 3 and Chapter 5, a declaration of a class, method, or field is annotated with a feature, which means that the declaration only exists if the feature is present. Using language constructs such as conditional statements and annotations to express variability facilitates adoption of existing program analyses to the product line setting. However, not all product line variations may be easily represented using variability mechanisms of a conventional programming language. For example, pushing multiple alternative features



(a) Sample SPL Code (b) Sample SPL Test

Figure 2.2: Sample SPL Code and Test

into a single program may result in duplicate declarations. This can occur if two features introduce different implementations of the same method. A workaround is to factor code that is common in alternative features into a common feature that the alternative features refine. We deal with product lines that can be represented as conventional programs, albeit with some refactoring.

2.3 Product Line Test

A *test* of a product line is simply a `main()` method that executes some methods, references some code of the product line and produces a result that can be checked against the expected result. Figure 2.2(b) shows a sample test that calls the method in Figure 2.2(a). It is very important to note that a feature variable can have different values depending on which configuration is being executed for a test. Without

analyzing the feature model or the code, the test must be executed for every configuration of the feature model, i.e. for each configuration, feature variables' values must be set using the configuration's feature assignments and the test must be executed.

When necessary, the terms *test suite*, *test*, and *test case* are distinguished. A test suite is a collection of tests. A *test* does not have values assigned to feature variables, but it can have values assigned to other variables. A *test case* has values assigned to every variable except feature variables. Typically, *test* is considered to mean *test case*, unless noted otherwise.

Note that throughout the dissertation, the terms *configuration*, *feature combination*, *product* and *program* are used interchangeably to refer to both the set of feature assigments and the corresponding code. Also, the term *feature* will typically be used to refer to optional features since mandatory features, which do not affect the number of configurations, do not affect the time it takes test SPLs, which we are trying to reduce.

Chapter 3

Statically Pruning Configurations to Test

The contents of this chapter appeared in the 2011 Conference on Aspect-Oriented Software Development (AOSD 2011) [58]. [1]

3.1 Introduction

The most obvious challenge in testing or checking the properties of programs in an SPL is scale: an SPL with only 10 optional features has over a thousand (2^{10}) distinct programs. The need to assume the worst-case and test all programs is evident in the following scenario: suppose that every program of an SPL outputs a String that each feature might modify. To see if the output always conforms to a particular pattern, every possible feature combination must be tested.

Current practice often focuses on feature combinations that are believed to have a higher chance of falsifying certain properties [26][27][74]. In light of no

[1] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Product Line Testing. Aspect Oriented Software Development (AOSD), 2011. The paper was developed jointly with my co-authors, who are my supervisor and my co-supervisor. Implementation and evaluation were done by me.

other information, this is reasonable but critical combinations may be overlooked. Another approach is to apply traditional verification techniques directly – model checking [42][99] or bounded exhaustive testing [18][102] – on every product of the SPL. Again, feature combinatorics render brute force impractical. Yet another complicating factor is that features often have no formal specifications; even contracts are typically unavailable. Classen et al.[24] proposed a technique to efficiently check a temporal property against a product line that is represented as a state machine. However, to efficiently run a test against an object-oriented product line, which we are interested in, a technique tailored to object-oriented programs is required.

Our work improves the state-of-the-art by leveraging the semantics of *features*, i.e. increments in functionality. It is well-known that there are features whose absence or presence has no bearing on the outcome of a test. Such features are *irrelevant* — they augment, but do not invalidate, existing behavior. To illustrate potential benefits, suppose we determine that 8 of the 10 features in the above example do not modify the output String and thus are irrelevant. We can confidently run the String output test on only $2^2 = 4$ programs to analyze the entire product line, instead of a thousand.

In this chapter, we explore the concept of irrelevant features to reduce SPL testing. We find features that do not influence the result of a given test (these features are irrelevant). We accomplish this by representing an SPL in a form where conventional program analyses can be applied, determining the features that are irrelevant for a given test, and pruning the space of such features to reduce the number of SPL programs to examine for that test without reducing its ability to find bugs. This chapter makes the following contributions:

- **Technique.** We precisely define (ir)relevance in terms of changes that a feature can make to a program. We modify off-the-shelf static analyses for object-oriented programs to check for relevance.

- **Implementation.** We implement our technique as an Eclipse plugin that uses *Soot*[85], a popular static analysis framework for Java, and *SAT4J*[86], an off-the-shelf SAT solver.

- **Evaluation.** We demonstrate the effectiveness of our technique on concrete product lines and tests.

3.2 Motivating Example

Product Line. Suppose that we have the product line in Figure 3.1 that represents bank accounts where one can add money and be rewarded for being a valuable customer. The code of each feature in Figure 3.1 is painted a distinct color. For now, ignore underlined code. Our product line has four features:

- `Base` (clear color) represents the core functionality which allows money to be added, interest and overdraft penalty to be computed, and provides a class (`PremiumAccount`) that represents premium accounts with money already loaded in.

- `Loyalty` (blue) rewards a customer for adding money to the account. The feature adds a `points` field, which is incremented by a percentage of the money in the account when `Account.add(int)` is called. The feature also adds `PremiumAccount.overdraftPenalty()`, which overrides the method provided by `Base`.

- `Ceiling` (yellow) places a ceiling on the return value of `interest(double)` and `PremiumAccount.overdraftPenalty()`.

- `Fee` (dark grey) charges for adding money. The charge going into the bank's account is not shown.

Feature Model. The feature model for our example requires `Base` to be present in every program and requires one of the other three features, yielding a total of 7 distinct programs:

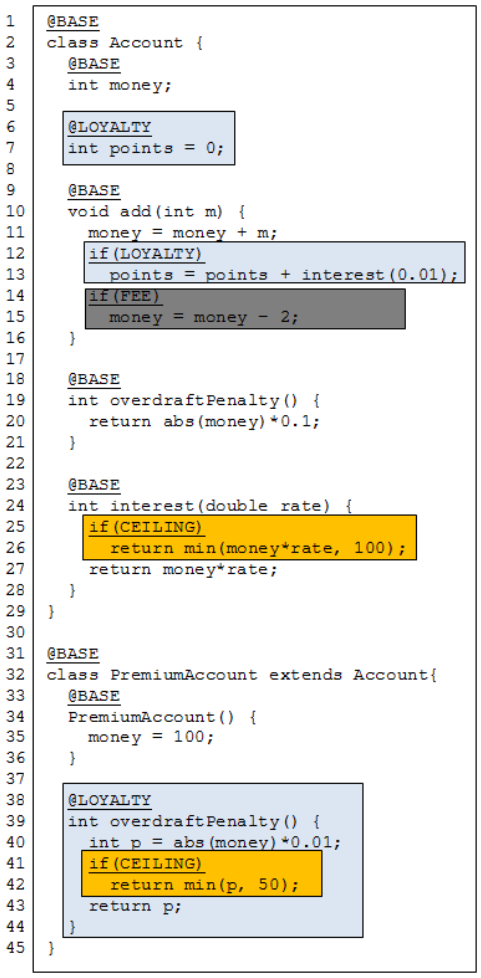
```
ProductLine :: [Ceiling] [Fee] [Loyalty] Base;
Ceiling or Fee or Loyalty;
```

Product Line Tests. Figure 3.2 shows three tests for our product line. `Test1` checks that there are no points when a premium account is created. `Test2` checks the penalty for $200 overdraft against a premium account. `Test3` adds $100 to an account and checks that there is at least that much in the account afterwards.

Although a test can be written fairly arbitrarily, such as bundling multiple tests into one and testing many functionalities at the same time, we assume a setting where a test exercises a small portion of the product line, the way a unit test does. To execute a test, all of its inputs (except the boolean feature variables like `LOYALTY` and `FEE` which are discussed later) must be set by the user.

Feature Combinatorics. Eliminating unnecessary feature combinations is the central problem in product line testing and we tackle this problem by determining what features are relevant to a test. We can intuitively understand what "relevance" means before we define it precisely. For example, consider `Test1`: only `Base` and `Loyalty` are relevant because only the code of these features is reachable from `Test1.main()`. For `Test2` and `Test3`, the relevant features are less obvious but can still be statically determined. In all cases, we can use knowledge of relevant features to reduce the set of SPL programs to test.

Solution Overview. Figure 3.3 shows an overview of our solution for reducing combinatorics in product line testing. We start with a product line that is encoded as an ordinary Java program, a feature model for the product line, and a product line test. We specialize the feature model with respect to the test to identify unbound features, a subset of which are relevant (Section 3.3). We then feed the



```
1  @BASE
2  class Account {
3    @BASE
4    int money;
5
6    @LOYALTY
7    int points = 0;
8
9    @BASE
10   void add(int m) {
11     money = money + m;
12     if(LOYALTY)
13       points = points + interest(0.01);
14     if(FEE)
15       money = money - 2;
16   }
17
18   @BASE
19   int overdraftPenalty() {
20     return abs(money)*0.1;
21   }
22
23   @BASE
24   int interest(double rate) {
25     if(CEILING)
26       return min(money*rate, 100);
27     return money*rate;
28   }
29 }
30
31 @BASE
32 class PremiumAccount extends Account{
33   @BASE
34   PremiumAccount() {
35     money = 100;
36   }
37
38   @LOYALTY
39   int overdraftPenalty() {
40     int p = abs(money)*0.01;
41     if(CEILING)
42       return min(p, 50);
43     return p;
44   }
45 }
```

Figure 3.1: Example Product Line

```
1  class Test1 {                          /***   Test1   ***/
2    static void main(String args) {
3      PremiumAccount a = new PremiumAccount();
4      assert a.points == 0;
5    }
6  }
7
8  class Test2 {                          /***   Test2   ***/
9    static void main(String args) {
10     PremiumAccount a = new PremiumAccount();
11     a.money = -200;
12     assert a.overdraftPenalty() == 2;
13   }
14 }
15
16 class Test3 {                          /***   Test3   ***/
17   static void main(String args) {
18     Account a = new Account();
19     a.add(100);
20     assert a.money >= 100;
21   }
22 }
```

Figure 3.2: Product Line Tests

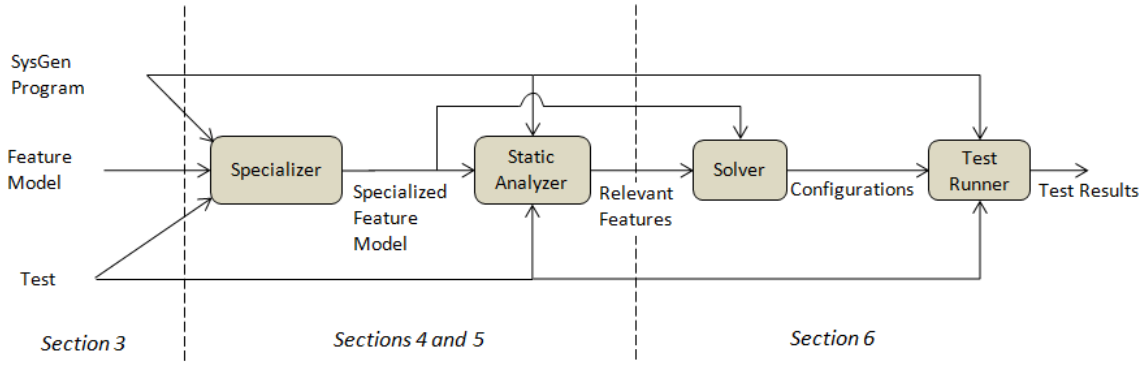


Figure 3.3: Overview of Our Technique

specialized feature model, the Java product line code and the test to a static analysis that identifies the relevant features (Section 3.4). Given the relevant features and specialized feature model, a solver determines the configurations against which the test must be run (Section 3.5). We begin by explaining relevant features.

3.3 Relevant Features

A feature is *relevant* if we need to consider both `true` and `false` values when running a test. As we explain in Section 3.3.2, a feature is considered to be relevant depending on whether its code can influence the test outcome. A product line's code, feature model and test are examined to reduce the set of features whose code needs to be statically analyzed. We describe how to do this next.

3.3.1 Pruning Features

A *bound* feature has its truth value fixed for a given test. Bound features are determined by adding constraints to the feature model to ensure that the test will compile [95]. For example, a tester may decide that certain features must always be present or absent when running a test by adding *test constraints* to the feature model (e.g. if a tester wants to run `Test2` with `Loyalty` present, the tester can add `Loyalty=true` to the feature model, which binds the feature to `true`). Constraints specialize the feature model, reducing feature combinations. The complete set of bound features are determined by mapping the specialized feature model to a propositional formula [7] and using a SAT solver to propagate constraints [46]. *Unbound* features, which can take either a `true` or `false` value, are simply the complement of bound features.

Of the unbound features, only the features whose code is *reachable* from the test's entry point (`main` method) need to be checked for relevance.[2] The static

[2]Unreachable features' code may also be relevant if the test uses reflection. See Section 3.7.1.

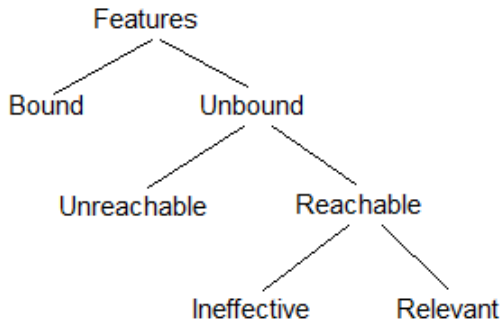


Figure 3.4: Classification of Features

analysis presented in Section 3.4 determines which features are reachable and only checks these features for relevance.

Figure 3.4 shows our classification of features. We use the term *ineffective* to describe reachable features that are not relevant (this term will be explained in detail in Section 3.3.2). We reserve the term *irrelevant* to describe any feature that is not relevant (i.e. ineffective, unreachable, and bound), for which we need only consider one truth value when running the test. Note that *whether the test passes or fails is independent of whether an irrelevant feature is present or not.* We discuss how to isolate relevant features in Section 3.3.2 but for now, it is apparent that:

- In `Test1`, `Base` and `Loyalty` are bound to `true` as the test references `PremiumAccount` (which belongs to `Base`) and `Account.points` (`Loyalty`). Note that `Base` is required anyway due to the feature model. Features `Fee` and `Ceiling` are unbound. These two features are also unreachable as their code is not executed by the test.

- In `Test2`, only `Base` is bound. Although the test references `PremiumAccount.overdraftPenalty()` of `Loyalty`, the method definition need

Also, note that bound features may actually be reachable as well but we just do not label them as such.

not exist as `Base` provides `Account.overdraftPenalty()`. Therefore, `Loyalty` is unbound. However, if the tester wanted to test only the former method definition, the constraint `Loyalty=true` would be added to the feature model. The reachable features are `Loyalty` and `Ceiling`.

- For `Test3`, only `Base` (`true`) is bound. All the three unbound features are reachable. For example, `Ceiling` is reachable as `interest(double)` is called by `Loyalty`.

Binding features reduces feature combinatorics (i.e,. the number of programs to test) from 2^n, where `n` is the number of unbound features in the entire product line, to 2^u, where `u` is the number of unbound features in the test. Determining reachable features `r` further reduces the number to 2^r. Relevant features `R`, a subset of reachable features, shrinks the number of programs to test to 2^R, where $2^R \leq 2^r \leq 2^u \leq 2^n$. We now discuss the conditions for relevance.

3.3.2 Conditions for Relevance

A reachable feature is ineffective to a test if the feature does not alter the (1) control-flow or (2) data-flow of any feature whose code may be executed by the test. By *control-flow*, we mean the *control-flow graph (CFG)* which is a directed graph whose nodes are basic blocks that consist of straightline code. A feature *preserves a CFG* if it only adds more code to existing basic blocks without introducing edges between the existing basic blocks, thereby preserving the shape of the graph itself. By *data-flow*, we mean the graph of *def-use pairs* [1]. A feature preserves def-use pairs if it writes only to variables that it introduces. Trivially, the set of relevant features is the complement of the ineffective features in the set of reachable features. In Section 3.4, we precisely define the checks of relevancy, but for now, consider these examples:

- For `Test1`, as there is no reachable feature as explained before, there is no relevant feature and thus, only one configuration, such as {`Base=true,Loyalty=true,Fee=false,Ceiling=false`}, needs to be run.

- For `Test2`, both of the reachable features, `Loyalty` and `Ceiling`, are relevant as the former changes the inter-procedural CFG by replacing a called method with its own method and the latter adds an edge to a CFG to exit early.

- For `Test3`, `Ceiling`, `Fee`, and `Loyalty` are reachable. `Fee` is relevant as it alters a variable (`money`) of another feature (`Base`). `Ceiling` is relevant as it changes control-flow of `interest()` method called from line 13. `Loyalty` is relevant as it allows code of another relevant feature (`Ceiling`) to be reached. With three relevant features, `Test3` must be run on all configurations.

- Although there is no ineffective feature in the running example, if `Loyalty` did not call `interest(double)` and instead incremented `points` by a numeric value, it would be an ineffective feature for `Test3`.

We now present a static analysis that conservatively determines reachable and relevant features.

3.4 Static Analysis

Using an off-the-shelf inter-procedural context-insensitive and flow-insensitive points-to analysis called *Spark* [67], our Soot-based static analysis examines code that is reachable from the start of a given test and checks if a reachable feature's code alters the behavior of another feature. Our static analysis identifies two classes of effects that a relevant feature can have: *direct* and *indirect*. The check for direct effects examines two types of changes that a feature can make: introductions (Section 3.4.1)

and modifications (Section 3.4.2). The check for indirect effects (Section 3.4.3) determines if a feature's code can allow a direct effect to be reached. If a feature is determined to have an effect by any of these checks, the feature is relevant.

Note that our static analysis is run once against the product line code, not on each configuration or feature combination of the product line. Namely, one *Abstract Syntax Tree (AST)* and one inter-procedural control-flow graph are created for the entire product line and analyzed. Also, determining direct and indirect effect of a feature does not involve considering combinations of features. Thus the algorithmic complexity of our static analysis is not in any way exponential in the number of features.

3.4.1 Introductions

An *introduction* adds a class, field, method or another type of class member. For example, `Base` introduces `Account`, `PremiumAccount` and `Account.money`. `Loyalty` introduces `Account.points` and `PremiumAccount.overdraftPenalty()`.

In general, the only way an introduction of feature `F` can influence the outcome of a test execution is for it to (a) override the introduction of another feature `G` and (b) is reachable from the test. By design, a feature can only override methods, not variable declarations, of another feature. An overriding method introduction that is reachable from the test affects control-flow of other features because it effectively replaces the CFG of the overriden method with its own. A feature with an overriding introduction is relevant.

For example, in `Test2`, `Loyalty` is relevant because it introduces a reachable method `PremiumAccount.overdraftPenalty()` that overrides `Base`'s introduction of `Account.overdraftPenalty()`.

3.4.2 Modifications

A *modification* adds a contiguous block of statements to an existing method. Modifications of SPLs are always enclosed by if-conditions of feature variables, such as lines 12-13 and 41-42 of Figure 3.1. Our static analysis for modifications was inspired by a similar analysis for aspects [25] that checks for data-flow and control-flow effects. Section 3.4.2 presents the control-flow check and Section 3.4.2 presents the data-flow check.

Control-Flow Check

The only way a modification does not preserve a CFG as described in Section 3.3.2 is if it adds a branching statement (i.e. `continue`, `break`, and `return` for Java) to the control-structure (i.e. loop, switch, and function) of another feature. A feature with such a modification has a control-effect and is relevant.

For example, in `PremiumAccount`'s `overdraftPenalty()`, `Ceiling`'s modification (line 42) optionally changes the control-flow of the method by returning a value different from what `Loyalty` returns. Therefore, `Ceiling` is relevant to `Test2`, which invokes `PremiumAccount`'s `overdraftPenalty()`. Also, `Ceiling` is relevant to `Test3` because line 26, reachable through line 13, changes the control-flow of `interest()`.

Data-Flow Check

The modifications made by feature F preserve def-use pairs if F's statements write (i) to fields that F introduced or (ii) to fields introduced by another feature, G, but whose base object (e.g., base object for the expression `x.money` is `x`) was allocated by F. The reason for Condition (i) is the following: a field introduced by F cannot have existed before F was added. As a result, writing to the field cannot possibly override existing values. As for Condition (ii), F should be able to modify objects

that it itself created. Here are three examples:

- Example satisfying (i): given `Test3` and the product line code, we see that `Loyalty`'s modification (line 13) satisfies (i) because it only updates a field, `points`, that `Loyalty` itself introduced.

- Example satisfying (ii): suppose that `Loyalty` has a modification that does the following:

  ```
  if(LOYALTY) {
    Account account = new Account();
    account.money = 100;
  }
  ```

  Even though `Loyalty` writes to a field (`money`) that is introduced by another feature (`Base`), this is allowed because the modification only affects the object `account` which cannot exist without `Loyalty`.

- Example not satisfying (i) and (ii): `Fee`'s modification (line 15) assigns to another feature's field `money` of the object `a` that was created by `Test3`, not `Fee`.

Our data-flow check evaluates both (i) and (ii). For each reachable `if(F)` statement, the check finds field writes occuring in the statement's control-flow (other `if(G)` statements in that control-flow, where `G` is not equal to `F`, are skipped as they will be visited later). Then for each field write found, `F` is checked against the feature that declared the field. If the two features are the same, the `if(F)` statement satisfies condition (i). If the two are different, then for each possible allocation site of the base object of the field being written, the feature of the allocation site must be `F` for the `if(F)` statement to satisfy condition (ii). If neither condition is satisfied, `if(F)` statement produces a data-flow effect and `F` is relevant.

We modified a Soot-based side-effect analysis [66] to implement the data-flow check. We chose this particular analysis because it was easy to modify for our needs. The analysis is as precise as Spark, which as mentioned is both context-insensitive and flow-insensitive. We argue in Section 3.7.2 that a highly precise static analysis is not necessary for our problem.

3.4.3 Indirect Effect

There are times when a feature satisfies both the control-flow and data-flow checks of irrelevancy, but the feature is still relevant because it enables the code of relevant features to be reached.

Indirect Data-Flow Effect. Consider Figure 3.5. An unbound feature `A` that writes only to its own variables can affect the outcome of a test for `m()` if its variables are read by a relevant feature `C` (relevant because it writes to `A`'s variable). In fact, a program with `C` will not even compile correctly without `A`. This is not a problem because a previously developed technique [95] ensures that `A=true` when `C=true` by constructing the implementation constraint $\texttt{C} \implies \texttt{A}$.

```
1  @BASE
2  class Program {
3    @A
4    int a = 0;
5
6    @BASE
7    void m() {
8       if (B) {
9          if (C) { a = a + 2; }
10      }
11   }
12 }
```

Figure 3.5: An Example Illustrating Indirect Effect

Indirect Control-Flow Effect. `C` is relevant in Figure 3.5 because it writes to `A`'s variable. `B`'s code does not change control-flow or data-flow of another feature, but it does *enable* a relevant feature, `C`, to be reached. Generating the reachability

constraint `C` $\Longrightarrow$ `B` allows `B` to be treated as an irrelevant feature without fearing that `B` will be turned off when `C` is on. However, in general, generating such reachability constraints efficiently can be difficult as there are many ways to reach a statement. So instead, we make a conservative approximation and consider each reach-enabling feature like `B` to be relevant, taking both of their truth values, guaranteeing that `C`'s code will be reachable. For this reason, in `Test3`, `Loyalty`, whose code does not alter control-flow or data-flow but does enable through line 13 `Ceiling`'s modification of `interest(double)` to be reached, is considered relevant along with `Ceiling` and `Fee`. Indirect control-flow effect is determined by collecting features of the transitive callers of a relevant feature's direct effect.

3.5 Configurations to Test

Given relevant features and the feature model specialized for the test, we now identify the configurations on which to run the test. Our algorithm, shown in Figure 3.6, relies on the SAT4J [86] SAT solver, which can enumerate solutions to a propositional formula. Our algorithm iterates through each possible combination of the relevant features and treats irrelevant features as don't-cares. More specifically, we find a solution to the specialized feature model and add it to the configurations to test (lines 6-7). We then ensure that the configuration's assignments to the relevant features do not appear again by creating a *blocking clause* [86] consisting of the assignments and conjoining the negation of the clause to the feature model (lines 9-16). We then check if there is another configuration and repeat the process until there are no more configurations.[3]

Once the configurations to test have been identified, a *test runner*, shown in Figure 3.3, goes through each configuration, creating a concrete program corre-

[3] A simple variation of our algorithm terminates after collecting k configurations, in case there is a huge number of configurations to test.

```
1  Set<Configuration> solve
2      (FeatureModel specializedFM, Set<Feature> relevantFeatures) {
3    Set<Configuration> configs = new HashSet<Configuration>();
4
5    while(specializedFM.isSatisfiable()) {
6      Configuration c = specializedFM.getOneSolution();
7      configs.add(c);
8
9      PropositionalFormula blockingClause =
10       new PropositionalFormula();
11     for(VariableAssignment varAssignment: c.getVarAssignments())
12     {
13       if(relevantFeatures.contains(varAssignment.getVariable()))
14         blockingClause = blockingClause.and(varAssignment);
15     }
16     specializedFM = specializedFM.and(not(blockingClause));
17   }
18
19   return configs;
20 }
```

Figure 3.6: Algorithm to Find Test Configurations

Table 3.1: Configurations to Test

Test1	Test2	Test3	Base	Loyalty	Fee	Ceiling
No	No	Yes	1	0	0	1
No	Yes	Yes	1	0	1	0
No	Yes	Yes	1	0	1	1
Yes	Yes	Yes	1	1	0	0
No	Yes	Yes	1	1	0	1
No	No	Yes	1	1	1	0
No	No	Yes	1	1	1	1

sponding to the configuration and running the test against that program.

Examples. Table 3.1 shows the results of analyzing our running example. Without analysis, each row, a configuration in the original feature model, would have to be executed for each test. However, with our analysis, given a test, only the rows with `Yes` entries in the column corresponding to the test need to be examined. For `Test1`, as stated in Section 3.4, there are no relevant features and thus the enumeration algorithm returns just one configuration, {`Base=true,Loyalty=-true,Fee=false,Ceiling=false`}, to test. For `Test2`, four combinations of the relevant features `Loyalty` and `Ceiling` must be tested. For `Test3`, all seven configurations must be tested.

3.6 Case Studies

We implemented our technique as an Eclipse plugin and evaluated it on three product lines: *Graph Product Line (GPL)*, which is a set of programs that implement different graph algorithms [72]; *notepad*, a Java Swing application with functionalities similar to Windows Notepad; and *jak2java*, which is a feature-configurable tool that is part of the AHEAD Tool Suite [8].

Multiple tests were considered for each product line. Each test, essentially a unit test, creates and calls the product line's objects and methods corresponding to the functionality being tested. We ran our tool on a Windows XP machine with Intel Core2 Duo CPU with 2.4 GHz and 1024 MB as the maximum heap space. Note that although the product lines were created in-house, they were created long before this chapter's technique was conceived (GPL and jak2java were created over 5 years ago and notepad was created 2 years ago). In fact, these product lines were originally written in Jak [9] and for the purpose of this chapter's technique, we developed a Jak-to-Java translator to convert them into the Java representation. Our plugin, the examined product lines and tests, as well as the detailed evaluation results are available for download [54].

3.6.1 Graph Product Line (GPL)

Table 3.2 shows the results for GPL, which has 1713 LOC with 18 features and 156 configurations. Variations arise from algorithms and structures of the graph (e.g. directed/undirected and weighted/unweighted) that are used. We report two representative tests below.

CycleTest. 10 features are unbound (actual features are listed in [54]). Applying the static analysis, we find that 7 out of the 10 are reachable. Out of these 7, only 1 feature, `Undirected`, is relevant. `Undirected` is relevant because it fails the data-flow check by adding an extra edge for every existing edge against

the graph which was created by the `Base` feature. The other reachable features perform I/O operations on their own data and are not considered to be relevant (see Section 3.7.1 for a discussion on I/O). With no analysis, the test would have to be run on 156 configurations, the number of programs in the product line. By specializing the feature model for this test and determining bound and unbound features, we reduce that number to 40. By applying the static analysis, we reduce the number to 2. The time taken to specialize the feature model is negligible. The static analysis takes less than a minute and a half.

Our technique achieves a useful reduction in the configurations to test. Such a reduction pays dividends in two ways. First, there is a good chance that it takes less time to perform the static analysis (1.20 minutes) and run the test on the reduced set (2) of configurations than to run the test on the original set (156) of configurations. But more importantly, redundant test results are eliminated and need not be analyzed by the tester. As far as the tester is concerned, there is no extra information in the other 154 test results and any information related to success or failure of the test can be obtained from these 2 configurations.

StronglyConnectedTest. This test requires a number of features to be bound for compilation, leaving only 4 features unbound. Out of those 4, 3 are reachable, but none are relevant. Just determining the unbound features already reduces the number of configurations, and applying the static analysis returns the best possible outcome, i.e. running the test on just 1 configuration. Like the previous test, the static analysis takes just over a minute.

3.6.2 Notepad

Table 3.3 shows the results for `Notepad`, which has 2074 LOC with 25 features and 7056 configurations. Variations arise from the different permutations of functionalities, such as saving/opening files and printing, and user interface support for

Table 3.2: GPL Results

Lines of code	1713
Features	18
Configurations	156
CycleTest	
Unbound features	10
Reachable features	7
Relevant features	1: Undirected (data-flow)
Configurations with unbound features	40
Configurations to test	2
Duration of static analysis	72 sec. (1.20 min.)
StronglyConnectedTest	
Unbound features	4
Reachable features	3
Relevant features	0
Configurations with unbound features	16
Configurations to test	1
Duration of static analysis	72 sec. (1.20 min.)

them (each functionality can have an associated toolbar button, menubar button, or both). We wrote tests for the example functionalities mentioned.

PersistenceTest. Binding still leaves 22 features unbound, but static analysis cuts down that number to 3 reachable features and only one relevant feature. The `UndoRedo` feature is relevant because it fails the data-flow check by attaching an event listener to the text area, which is allocated by another feature. Binding reduces 7057 configurations to 5256 and this is reduced to 2 configurations after running the analysis. Although Notepad is not large, it uses Java Swing, whose very large call-graph must be included in order for application call-back methods to be analyzed. This substantially raised the analysis time to 45 minutes. A common solution to this problem is to skip over certain method calls, especially those that are deep, in the framework, but this must be done with great care as doing so could prevent call-back methods from being reached. Reducing analysis time is a subject for further work.

PrintTest. The numbers are similar to the previous test, but this time,

Table 3.3: Notepad Results

Lines of code	2074
Features	25
Configurations	7057
PersistenceTest	
Unbound features	22
Reachable features	3
Relevant features	1: UndoRedo (data-flow)
Configurations with unbound features	5256
Configurations to test	2
Duration of static analysis	2856 sec. (47.60 min.)
PrintTest	
Unbound features	22
Reachable features	4
Relevant features	2: UndoRedo (data-flow) Persistence (introduction)
Configurations with unbound features	5256
Configurations to test	4
Duration of static analysis	2671 sec. (44.51 min.)

`Persistence` is also found to be relevant because one of its methods overrides a method of an off-the-shelf file filter class in the Swing framework. Still, we only have to test 4 configurations rather than 5256. The duration is long for the same reason as mentioned previously.

3.6.3 jak2java

Table 3.4 shows the results for `jak2java`, which has 26,332 LOC with 17 features and 5 configurations. Despite the large code base and the number of features, there are only five configurations total because of the many constraints in the feature model. We wrote tests to execute the methods that we know are modified by other features. We aimed to find out whether these modifications would render these other features relevant to the method being executed. Here are some representative results.

ReduceToJavaTest. Features `sm5` and `j2jClassx` are relevant because they introduce methods that override methods of another feature. Feature `j2jSmx`

Table 3.4: jak2java Results

Lines of code	26332
Features	17
Configurations	5
ReduceToJavaTest	
Unbound features	4
Reachable features	3
Relevant features	3: sm5 (introduction), j2jSmx (control-flow), j2jClassx (introduction)
Configurations with unbound features	5
Configurations to test	5
Duration of static analysis	254 sec. (4.24 min.)
ArgInquireTest	
Unbound features	4
Reachable features	-
Relevant features	-
Configurations with unbound features	5
Configurations to test	-
Duration of static analysis	-

is relevant because it fails the control-flow check by returning early from the method of another feature. Unfortunately, all the configurations in the product line must be tested. The reason for this is that calling `reduce2java`, the method being tested, is very much like calling the `main` method of a product line, which reaches a large portion of the product line's code base. Because there is a large amount of code to analyze, the static analysis takes 4.24 minutes. All the configurations have to be tested because a large fraction of the product line's interactions are reachable.

ArgInquireTest. This test calls a method that is conditionally overriden, which our static analysis cannot handle (which will be discussed shortly in the last bullet of Section 3.7.1). Therefore, only the feature model could be analyzed, which determines that the 5 configurations with unbound features must be tested.

3.7 Discussion

We now discuss assumptions and limitations, the effectiveness of our work, testing missing functionality, threats to validity, and a perspective.

3.7.1 Assumptions and Limitations

Off-the-shelf program analyses have well-known limitations. Indeed, although the last assumption of the list below is unique to our work, we believe the rest are not.

- **Reflection.** Any change to the code base, including the addition of a class member, can change the outcome of reflection. We assume that reflection is not used. Another possibility is to check if reflection is used in the control-flow of the test and consider *any* unbound feature to be relevant. Related work, such as [41], also do not consider reflection.

- **Native Calls.** It is hard to determine if a native call, such as an I/O operation, has a side-effect using Soot. Rather than making the overly conservative assumption that every native call has a side-effect, we assume that native calls have no side-effect. Consequently, features can perform reads/writes to files or standard input/output without being considered relevant.

- **Timing.** If a test uses the duration of its execution as an outcome, any feature that adds instructions to the test will be considered relevant. Rather than checking if a test indeed uses such a timer, we assume that it does not.

- **Exceptions.** A reachable feature that can throw an exception can be considered to have a control-effect and thus be identified as a relevant feature. The problem with doing this is that a majority of reachable features would be considered relevant because in Java, *unchecked exceptions* (`RuntimeException`, `Error`, and their subclasses) can be thrown in many expressions including

pointer, arithmetic, and array operations [76]. A possible solution is to rely on a lightweight specification such as a contract (as a related work [25] does) to determine whether a reachable feature can throw an exception and consider it to be relevant if it can. Another possiblity is to consider only reachable features that can throw *checked exceptions* to be relevant and assume that unchecked exceptions can be found when running unit tests of these features. For now, we leave exceptions as future work.

- **Local Variables.** A method can declare variables local to it. We assume a feature's modification does not reference or modify local variables introduced by other features. Features are written in a dedicated language like *Jak* [9] that restricts a feature's modifications in this way. We assume this restriction holds and we use an off-the-shelf side-effect analysis, which, by definition of "side-effect" of a method, need not consider writes to local variables. Our benchmarks satisfy this assumption as they were translated from `Jak` to `Java` representation using a translator. It would not require much effort to remove this limitation.

- **Conditional Method Overriding and Field Hiding.** We assume that a method is not conditionally overriden due to a feature, i.e. the overriding method cannot be annotated with a feature (`A`) that is different from the feature (`B`) of the overriden method. The reason this is not allowed is because allowing it would require the call-graph to include calls to both the overriding method (for the case where `A` is true) and the overriden method (for the case where `A` is false), which standard static analysis frameworks like Soot cannot construct because they are unaware of features (their call-graph construction would have to be modified to include both calls, which would reqire non-trivial programming effort). Similarly, we assume that a field is not conditionally hidden due to a feature, i.e. the hiding field cannot be annotated with a

feature (`A`) that is different from the feature (`B`) of the hidden field. Allowing this would require a field reference to be traced back to both the hiding field (for the case where `A` is true) and the hidden field (for the case where `A` is false), which standard static analysis frameworks against cannot establish because they are unaware of features (modifying this would again require non-trivial programming effort).

3.7.2 Effectiveness

Our technique works because there are tests that exercise a small portion of the product line involving a few features. Even just binding features can cut down many configurations. Further reductions are possible as not many features are reachable from a test and even fewer are relevant. Determining reachable and relevant features is difficult to do manually and requires a dedicated program analysis like ours. Although a highly precise program analysis can significantly reduce false positives, our case studies illustrate that a context-insensitive analysis suffices because only a small set of classes and methods, relevant to the functionality being tested, are instantiated and invoked.

3.7.3 Testing Missing Functionality

Suppose feature `Interest` should modify the data-flow of a method `deposit(int)`. The feature's author forgets to make the modification, which causes our analysis to report that `Interest` is irrelevant when testing `deposit(int)`.

`Interest` is irrelevant because it is missing functionality, rather than having an orthogonal functionality as previous example features did. Without a specification, e.g. that `Interest` is supposed to be relevant to `deposit(int)`, the burden of detecting missing functionality rests on the alertness of testers; no program analysis could detect this error. This is a general problem of testing and is

not limited to our work. In fact, our work helps in that it reports information on feature (ir)relevance, which may provide a clue to such errors.

3.7.4 Threats to Validity

Our technique can take longer than running the test on all the configurations, as is the case with `ReduceToJavaTest` for `jak2java` since there is no reduction in configurations. But we believe this case is an outlier and the static analysis is worth running to achieve even a small reduction for several reasons. First, testing a product requires it to be generated, which takes non-negligible time. Second, it takes time to run the tests themselves. Third, a configuration's test result may be redundant with another configuration's test result due to an irrelevant feature between the two, yet the tester will have to waste time analyzing both configurations' results. Further experience with our analysis will bear out these points.

3.7.5 Perspective

Initially, our belief was that existing analyses for conventional programs could be directly applied to a Java representation of a product line. However, we discovered that analyzing a Java product line was much more challenging than we had anticipated. Consider the following example. While some parts of our analysis, including reachability and data-flow check, are performed using a backend abstraction like 3-address code, other parts of our analysis, notably the control-flow check, must be performed on a frontend abstraction like Abstract Syntax Tree because branch statements like `break` and `continue` are often optimized away on the backend [13]. This presents the technical challenge of developing a bridge between frontend and backend analyses. For example, we only want to perform control-flow checks on the reachable methods, but these methods are determined by a backend analysis. Currently, we provide a string representation that the frontend and the backend

abstractions both map to. Developing a more robust intermediate abstraction may be necessary in the future.

3.8 Related Work

3.8.1 Product Line Testing and Verification

There is a considerable amount of research in product line testing and verification (see [73] for a survey). We discuss research most closely related to ours.

Monitoring. After developing this chapter's technique, we and a colleague published a paper on a related idea of statically eliminating monitors from configurations that provably cannot trigger a monitor used to enforce safety properties against a product line [59]. The two techniques are different both in setting and technique. In setting, in this chapter's technique, only one of the configurations that produce the identical test outcome needs to be tested. In [59], even if a hundred configurations are identical in the way they trigger a monitor (e.g. through the same feature), all hundred configurations need to be monitored because all hundred can be used by the end-user. In technique, [59] needs to determine configurations that do not satisfy an API-level property, i.e. a sequence of method calls, while this chapter's technique needs to determine configurations that do not satisfy a much lower level and orthogonal property, i.e. control-flow or data-flow effect. Thus the two works are complementary.

Model-Checking. Classen et al.[24] recently proposed a technique to check a temporal property against a product line that is in the form of *Feature Transition Systems (FTS)*, which is a preprocessor-like representation, but for transition systems. Their technique composes the product line's FTS with the automaton of the temporal property's negation and reports violating configurations. Although we both tackle the general problem of checking a property against a product line,

they work on a representation (transition systems) and setting (verifying temporal properties) different from ours (object-oriented programs and testing), making the two techniques complementary.

Sampling. In sampling, an SPL tester selects a subset of configurations to test using domain knowledge, such as the tendency for certain configurations to be more problematic than others. For example, an SPL tester may choose a subset of features for which all combinations must be examined, while for other features, only t-way (most commonly 2-way) interactions are tested [26][27][74]. Sampling approaches can miss problematic configurations, whereas we use a program analysis to safely prune feature combinations.

Test Construction. Instead of generating tests from a complete specification of a program, tests are generated incrementally from feature specifications [98]. There is also research on constructing a product line of tests so that they may be reused [10][79]. We address the different problem of minimizing test execution for a single given test.

3.8.2 Program Slicing

Determining relevant features is closely related to *backwards program slicing* [100], which uses a dataflow analysis to determine the minimal subset of a program that can affect the values of specified variables at a specified program point. Our definition of relevance is more conservative than a "slice" [100] but at the same time, requires less precision: our goal is to reduce feature combinations to test by determining features, not statements, for which we need only assign one truth value.

3.8.3 Feature Interactions

There is a large body of work on detecting feature interactions using static analysis [80][90][31][25][68], of which *harmless advice* [31] and *Modular Aspects with*

Ownership (MAO) [25] are the most relevant. Harmless advice introduces a type system in which aspects can terminate control-flow but cannot produce data-flow to the base program. MAO relies on contracts to determine if an aspect changes the control-flow or data-flow of another module. Our analysis was inspired by MAO, but is technically closer to harmless advice, as both perform an inter-procedural analysis and do not rely on contracts. But unlike harmless advice, our approach does not require every feature to be harmless or irrelevant.

More importantly, MAO and harmless advice assume a setting where all modules (i.e. aspects/features) are required for the program to work, which is sharply different from SPLs. Indeed, related work in feature interactions perform analysis more for modular reasoning of a single program, rather than for reducing combinatorics in product line testing.

3.8.4 Compositional Analysis and Verification

Currently, we perform a static analysis for each test. With multiple tests, it is possible that the same classes and methods of the product line will be analyzed multiple times. It may be possible to analyze the product line once and combine the result against that of analyzing each test using compositional static analysis [28] and verification [36][68].

3.8.5 Reducing Testing Effort

Reducing testing effort for a single program, typically using output from some analysis, has a long history. For example, [84] identifies a subset of existing tests to run given a program change. Our technique, which identifies a subset of existing features that are relevant for a given test in a product line, is a technique for reducing testing effort that is specific to a product line.

3.9 Summary

Software Product Lines (SPLs) represent a fundamental approach to the economical creation of a family of related programs. Testing SPLs is more difficult than testing conventional programs because of the combinatorial number of programs to test in an SPL.

Features are a fundamental, but unconventional, form of modularity. Combinations of features yield different programs in an SPL and each program is identified by a unique combination of features. Features impose a considerable amount of structure on programs (that is why features are composable in combinatorial numbers of ways), and exploiting this structure has been the focus of this chapter's technique.

Our key insight is that every SPL test is designed to evaluate one or more properties of a program. A feature might alter any number of properties. In SPL testing, a particular feature may be relevant to a property (test) or it may not. Determining whether a feature is relevant for a given test is the critical problem.

We presented a framework for testing an SPL. Given a test, we determine the features that need to be bound for it to compile. This already reduces configurations to test. Of the unbound features, we determine the features reachable from the entry point of the test, further reducing configurations. And of the reachable features, we determine the features that affect the properties being evaluated, reducing configurations even more.

Several case studies were presented that showed meaningful reductions in the number of configurations to test, and more importantly, lends credence to the folktale that many features of a product line add new behavior without affecting existing behavior. We demonstrated the idea of leveraging such features to exhaustively but efficiently test product lines. Our work is a step forward in practical reductions in SPL testing.

Chapter 4

SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems

The contents of this chapter appeared in the 2013 Conference on Foundations of Software Engineering (FSE 2013) [62]. [1]

[1] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo d'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In Foundations of Software Engineering (FSE), 2013. Darko conceived the idea of the paper, co-developed with me the Java implementation, developed the Ruby on Rails implementation, conducted the Groupon (Ruby on Rails) evaluation, and contributed to writing the paper. I co-developed the Java implementation (with Darko), conducted the Java evaluation, and took the lead in writing the paper. Sabrina, Paulo and Marcelo assisted with the Java evaluation and contributed to writing the paper. Sarfraz and Don contributed to writing the paper.

4.1 Introduction

4.2 Introduction

The previous chapter's technique can be essentially divided into two analyses: *reachability* analysis that statically determines a subset of the features that are reachable from the test's entry point and *relevance* analysis that statically determines a subset of those *reachable* features that change data-flow or control-flow of other features. The test then needs to be run only on combinations of such *relevant* features. Evaluation showed that reachability analysis typically has a greater impact than relevance analysis, i.e. the difference between the total number of features and the number of reachable features is likely to be larger than the difference between the number of reachable features and the number of relevant features.

This chapter presents *SPLat*, a new lightweight technique for determining configurations to run against a test *during* test execution, rather than using an up-front static analysis like the previous chapter's technique. Although SPLat performs only a reachability analysis, the effectiveness of SPLat is comparable to the previous chapter's technique. And most importantly, SPLatis generally faster and more scalable than the reachability analysis of the latter *and* existing dynamic analysis techniques because SPLatuses *stateless exploration* [37], which explores different configurations by simply restarting execution rather than having to save and restore state.

As in the previous chapter, we assume a test exercises a subset of the code base, which means that some of the features are likely to never even be encountered during the test execution. Combinations of such unreachable features yield many test runs that have the same *trace* or sequence of bytecode instructions executed by the test. SPLat determines for each test the set of unique traces and hence the smallest set of configurations to run. Specifically, let $p_1 \ldots p_k$ (for $k \geq 1$) be the

configuration variables for a program. Each p_i takes a value from a finite domain D_i—in the case of SPLs, each variable takes a boolean value that represents whether the feature is selected or not. Let c and c' be two different configurations to run on a test t, and let τ and τ' be their traces. In both runs, t fixes the input values for non-configuration variables. Configuration c' is unnecessary to execute if c has been executed and $\tau' = \tau$. The set of all configurations with unique traces forms a *revealing subdomain* for t [101].

`SPLat` achieves an optimal reduction in the number of configurations, *i.e.* for any test, `SPLat` runs only configurations that have a unique trace. Experimental results show that `SPLat` yields a reduction in testing time that is proportional to the reduction in the number of configurations. Our insight into pruning configurations was inspired by the Korat algorithm for test-input generation [18], which introduced the idea of *execution-driven pruning* for solving data-structure invariants written as imperative code.

`SPLat` supports constraints among configuration variables, which defines the valid configurations. For a SPL, these constraints are expressed through a *feature model* [49] that (1) provides a hierarchical arrangement of features and (2) defines allowed configurations. `SPLat` uses SAT to prune invalid configurations and in tandem uses execution-driven pruning to further remove the valid configurations that are unnecessary for execution of each test.

`SPLat` is effective because it monitors the accesses of configuration variables during test execution. Monitoring is lightweight—both in terms of its execution overhead and in terms of its implementation effort. We developed two implementations, one for Java and one for Ruby on Rails. The Java implementation of `SPLat` was developed by this dissertation's author and Darko Marinov, one of the authors of the publication [62] corresponding to this chapter, and it leveraged the publicly available Korat code [64]. The Ruby on Rails implementation of `SPLat` was devel-

oped by Darko Marinov from scratch and took only two days to implement while being robust enough to run against a large, industrial codebase at Groupon, Inc. The results from Groupon, Inc. were obtained by Darko Marinov as well.

This chapter makes the following contributions:

- **Lightweight analysis of configurable programs:** We introduce the idea of lightweight monitoring for highly configurable systems to speed up test execution. `SPLat` instantiates this idea and can be easily implemented in different run-time environments.
- **Implementation:** We describe two implementations of `SPLat`, one for Java and one for Ruby on Rails.
- **Evaluation:** We evaluate `SPLat` on 10 Java SPLs. Experimental results show that `SPLat` effectively identifies relevant configurations with a low overhead. We also apply `SPLat` on a large configurable system (with over 171KLOC in Ruby on Rails). The system uses over 170 configuration variables and contains over 19K tests (with over 231KLOC in Ruby on Rails). To the best of our knowledge, this is the largest and most complex industrial codebase used in research on testing SPLs [4, 29, 51, 58, 60, 82], and `SPLat` scales to this size without much engineering effort.

4.3 Motivating Example

To illustrate the testing process, we use a simple Notepad product line. Figure 4.1 shows the feature model of Notepad. This model has one mandatory feature, `BASE`, and three optional features, `MENUBAR`, `TOOLBAR`, and `WORDCOUNT`. The constraint requires every Notepad configuration to have a `MENUBAR` or `TOOLBAR`. For example, assigning `false` to both `TOOLBAR` and `MENUBAR` would violate the disjunction

constraint and therefore be invalid. In contrast, assigning `false` to one of these two features and `true` to the other feature is valid.

```
ProductLine :: [MENUBAR] [TOOLBAR] [WORDCOUNT] BASE;
MENUBAR or TOOLBAR;
```

Figure 4.1: Notepad Feature Model

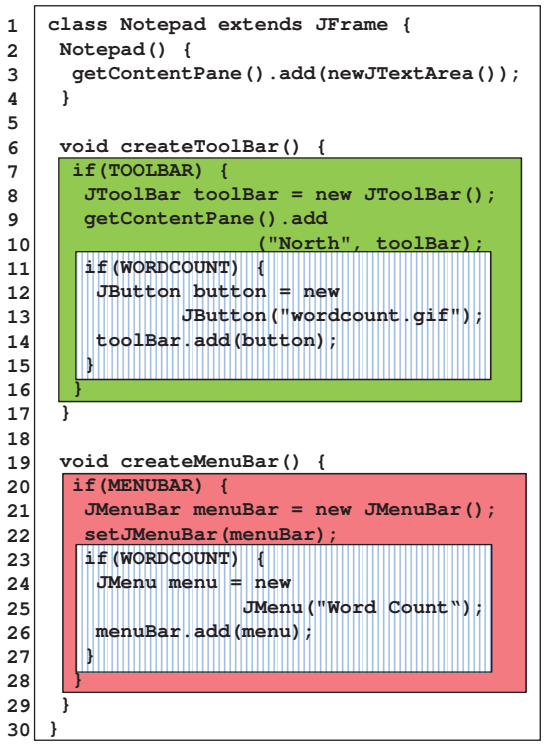
Figure 4.2(a) shows the code for Notepad. `BASE` (clear color) represents the core functionality, which, in this case, corresponds to constructing a `Notepad` with a `JTextArea` that the user types into. `TOOLBAR` (green color) adds a `JToolBar` to the frame. `MENUBAR` (red color) sets a `JMenuBar` against the frame. `WORDCOUNT` (blue color) adds its toolbar icon if the toolbar is present or its menubar item if the menubar is present.

Figure 4.2(b) shows an example test that instantiates the `Notepad` class and creates a toolbar for it. Note that test does *not* call the `createMenuBar()` method. To be able to execute a test, each variable in the test, except the feature variables, must be given a value.

We use the automated GUI testing framework FEST [33] to run the test. The helper method `newFixture()` is not shown for simplicity. The test execution launches the frame, simulates a user entering some text into the `JTextArea` of the frame, checks that the text area contains exactly what was entered, and closes the frame.

Without analyzing the feature model or the code, this test would need to be run on all 8 combinations of the 3 optional features, to check all potential test outcomes. However, some configurations need not be run. Analyzing the feature model, we note that two configurations are *invalid*: $MTW = 000$ and $MTW = 001$, where M, T, and W stand for `MENUBAR`, `TOOLBAR`, and `WORDCOUNT` respectively. Hence, no more than 6 configurations need to be run.

`SPLat` further reduces that number by dynamically analyzing the code that



```
1  class Notepad extends JFrame {
2   Notepad() {
3    getContentPane().add(newJTextArea());
4   }
5
6   void createToolBar() {
7    if(TOOLBAR) {
8     JToolBar toolBar = new JToolBar();
9     getContentPane().add
10                    ("North", toolBar);
11    if(WORDCOUNT) {
12     JButton button = new
13              JButton("wordcount.gif");
14     toolBar.add(button);
15    }
16   }
17  }
18
19  void createMenuBar() {
20   if(MENUBAR) {
21    JMenuBar menuBar = new JMenuBar();
22    setJMenuBar(menuBar);
23    if(WORDCOUNT) {
24     JMenu menu = new
25                  JMenu("Word Count");
26     menuBar.add(menu);
27    }
28   }
29  }
30 }
```

(a) Code

```
1  public void test() {
2   Notepad n = new Notepad();
3   n.createToolBar();
4
5   // Automated GUI testing
6   FrameFixture f = newFixture(n);
7   f.show();
8   String text = "Hello";
9   f.textBox().enterText(text);
10  f.textBox().requireText(text);
11  f.cleanUp();
12 }
```

(b) Test

Figure 4.2: Notepad SPL and Example Test

the test executes. For example, executing the test against the configuration $c := MTW = 100$ executes the same trace as configuration $c' := MTW = 101$. The

reason is that the test only calls `createToolBar()`, which is empty in both configurations c and c' since `TOOLBAR` is false in both configurations. Although the code in `createMenuBar()` is different in c and c', the test never executes it. Therefore, having executed c, execution of c' is unnecessary. We will show in Section 4.4.3 that `SPLat` runs this test for only three configurations (*eg* $MTW = 010$, $MTW = 011$, $MTW = 100$).

4.4 Technique

Given a test for a configurable system, `SPLat` determines all relevant configurations on which the test should be run. Each configuration run executes a unique trace of the test. `SPLat` executes the test on one configuration, observes the values of configuration variables, and uses these values to determine which configurations can be safely pruned. `SPLat` repeats this process until it explores all relevant configurations or until it reaches a specified bound on the number of configurations to examine. We first describe the feature model interface and then the core algorithm.

4.4.1 Feature Model Interface

Figure 4.3 shows the code snippet that defines the `FeatureModel` interface. The type `FeatureVar` denotes a feature variable. A `VarAssign` object encodes an assignment of boolean values to feature variables. An assignment can be *complete*, assigning values to all the features, or *partial*, assigning values to a subset of the features. A complete assignment is *valid* if it satisfies the constraints of the feature model. A partial assignment is *satisfiable* if it can be extended to a valid complete assignment.

The `FeatureModel` interface provides queries for determining the validity of feature assignments, obtaining valid configurations, and checking if specified features are mandatory. Given an assignment α, the method `getValid()` returns

```
class FeatureVar {...}
class VarAssign { ...
 Map<FeatureVar, boolean> map; ...}
interface FeatureModel {
 Set<Assign> getValid(Assign a);
 boolean isSatisfiable(Assign a);
 boolean isMandatory(FeatureVar v);
 boolean getMandatoryValue(FeatureVar v);
}
```

Figure 4.3: Feature Model Interface

the set of all complete assignments that (1) agree with α on the values of feature variables in α and (2) assign the values of the remaining feature variables to make the complete assignment valid. If the set is not empty for α, we say that α is *satisfiable*; the method `isSatisfiable()` checks this. The method `isMandatory()` checks if a feature is mandatory according to the feature model and the method `getMandatoryValue()` returns the mandatory value for the specified feature. We build on a SAT solver (SAT4J [86]) to implement these feature model operations.

4.4.2 Main Algorithm

Figure 4.4 lists the `SPLat` algorithm. It takes as input a test `t` for a configurable system and a feature model `fm`. To enable exploration, the algorithm maintains a `state` that stores the values of feature variables (line 2) and a `stack` of feature variables that are read during the latest test execution (line 1). `SPLat` performs a mostly stateless exploration of paths: it does not store, restore, or compare program states as done in stateful model checking [4, 29, 51, 60, 82]; instead, `SPLat` stores only the feature decisions made along one path and re-executes the code to explore different program paths, which corresponds to valid and dynamically reachable configurations. To that end, `SPLat` needs to be able to set the values of

feature variables, to observe the accesses to feature variables during a test run, and to re-execute the test from the beginning.

The algorithm first initializes the values of feature variables (lines 6–11) using the feature model interface. Mandatory features are set to the only value they can have, and optional features are initially set to `false`. Note that initial assignment may be invalid for the given feature model. For example, initially setting feature variables to `false` would violate the constraint in our Notepad example. We describe later how `SPLat` enforces satisfiability *during* execution (in line 43). It adjusts the assignment of values to feature variables *before* test execution gets to exercise code based on an invalid configuration. Such scenario could potentially lead to a "false alarm" test failure as opposed to revealing an actual bug in the code under test. Note that the calls to `state.put()` both in the initialization block and elsewhere not only map a feature variable to a boolean value in the state maintained by `SPLat` but also set the value of the feature variable referred to by the code under test.

`SPLat` then instruments (line 12) the code under test to observe feature variable reads. Conceptually, for each read of an optional feature variable (*eg* reading variable `TOOLBAR` in the code `if(TOOLBAR)` from Figure 4.2), `SPLat` replaces the read with a call to the `notifyFeatureRead()` method shown in Figure 4.4. The reads are statically instrumented so that they can be intercepted just before they happen during test execution. Mandatory feature variable reads need not be instrumented because the accessed values remain constant for all configurations.

`SPLat` next runs the test (line 16). The test execution calls the method `notifyFeatureRead()` whenever it is about to read a feature variable. When that happens, `SPLat` pushes the feature variable being read on the `stack` if it is not already there, effectively recording the order of the first reads of variables. This `stack` enables backtracking over the values of read feature variables. An important

```
1  Stack<FeatureVar> stack;
2  Map<FeatureVar, Boolean> state;
3  FeatureModel fm; // input, shared with instrumented code
4
5  void SPLat(Test t) {
6   // Initialize features
7   state = new Map();
8   for (FeatureVar f: fm.getFeatureVariables())
9    state.put(f, fm.isMandatory(f) ?
10                fm.getMandatoryValue(f):
11                false);
12   instrumentOptionalFeatureAccesses();
13   do {
14    // Repeatedly run the test
15    stack = new Stack();
16    t.runInstrumentedTest();
17    VarAssign pa =
18     getPartialAssignment(state, stack);
19    print("configs covered: ");
20    print(fm.getValid(pa));
21
22    while (!stack.isEmpty()) {
23     FeatureVar f = stack.top();
24     if (state.get(f)) {
25      state.put(f, false); // Restore
26      stack.pop();
27     } else {
28      state.put(f, true);
29      pa = getPartialAssignment(state, stack);
30      if (fm.isSatisfiable(pa))
31       break;
32     }
33    }
34   } while (!stack.isEmpty());
35 }
36
37 // called-back from test execution
38 void notifyFeatureRead(FeatureVar f) {
39  if (!stack.contains(f)) {
40   stack.push(f);
41   VarAssign pa =
42    getPartialAssignment(state, stack);
43   if (!fm.isSatisfiable(pa))
44    state.put(f, true);
45  }
46 }
```

Figure 4.4: SPLat Algorithm

step occurs during the call to `notifyFeatureRead()` (line 43). The initial value assigned to the reached feature variable may make the configuration unsatisfiable. More precisely, at the beginning of the exploration, `SPLat` sets an optional feature value to `false`. When the code is about to read the optional feature, `SPLat` checks whether the `false` value is consistent with the feature model, *i.e.* whether the *partial* assignment of values to feature variables on the `stack` is satisfiable for the given feature model. If it is, `SPLat` leaves the feature as is. If not, `SPLat` changes the feature to `true`.

Note that updating a feature variable to `true` *guarantees* that the new partial assignment is satisfiable. The update occurs *before* execution could have observed the old value which would make the assignment unsatisfiable. The reason why this change of value keeps the assignment satisfiable follows from the overall correctness of the `SPLat` algorithm: it explores only satisfiable partial assignments (line 30), and it checks if the assignment is satisfiable in *every* variable read (line 43); thus, if a partial assignment was satisfiable considering all features on the `stack`, then it must be possible to extend that assignment with at least one value for the new feature that was not on the `stack` but is being added. If the variable associated with the new feature stores `false` at the moment execution accesses that variable, and if the partial assignment including that feature variable is *not* satisfiable, then we can change the value to `true` (line 44). Recall that optional feature variables are initialized to `false`.

After finishing one test execution for one specific configuration, `SPLat` effectively covers a set of configurations. This set can be determined by enumerating every complete assignment that (1) has the same values as the partial assignment specified by variables `state` and `stack` (lines 17–18) and (2) is valid according to the feature model (line 20).

`SPLat` then determines the next configuration to execute by backtracking

on the `stack` (lines 22–33). If the last read feature has value `true`, then `SPLat` has explored both values of that feature, and it is popped off the stack (lines 24–27). If the last read feature has value `false`, then `SPLat` has explored only the `false` value, and the feature should be set to `true` (lines 27–32). Another important step occurs now (line 30). While the backtracking over the `stack` found a partial assignment to explore, it can be the case that this assignment is not satisfiable for the feature model. In that case, `SPLat` keeps searching for the next satisfiable assignment to run. If no such assignment is found, the `stack` becomes empty, and `SPLat` terminates.

4.4.3 Example Run

We demonstrate `SPLat` on the example from Figure 4.2. According to the feature model (Figure 4.1), `NOTEPAD` and `BASE` are the only mandatory features and are set to `true`. The other three feature variables are optional and therefore `SPLat` instruments their reads (Figure 4.2(a), lines 7, 11, 20, and 23). Conceptually, the exploration starts from the configuration $MTW = 000$.

When the test begins execution, `notifyFeatureRead()` is first called when `TOOLBAR` is read. `TOOLBAR` is pushed on the stack, and because its assignment to `false` is satisfiable for the feature model, its value remains unchanged (*i.e.* stays `false` as initialized). Had the feature model required `TOOLBAR` to be `true`, the feature's value would have been set to `true` at this point.

With `TOOLBAR` set to `false`, no other feature variables are read before the test execution finishes. (In particular, `WORDCOUNT` on line 11 is not read because that line is not executed when `TOOLBAR` is `false`.) Therefore, this one execution covers configurations $MTW = -0-$ where $-$ denotes a "don't care" value. However, configurations $MTW = 00-$ are invalid for the given feature model, so this one execution covers two valid configurations where `TOOLBAR` is false and `MENUBAR` is

true (MTW=10-). Note that even though the value of WORDCOUNT does not matter here, it is given a value nonetheless for an execution because in an execution, each variable must have a concrete value. So let us say MTW=100 here.

SPLat next re-executes the test with TOOLBAR set to true, as it is satisfiable for the feature model. WORDCOUNT is encountered this time, but it can remain false, and the execution completes, covering MTW=-10 (again, for an execution, all variables need to be set, so let us say that MTW=010 is what actually executes). SPLat then sets WORDCOUNT to true, and the execution completes, covering MTW=-11 (let us say MTW=011 was used). SPLat finally pops off WORDCOUNT from the stack because both its values have been explored, and pops off TOOLBAR for the same reason, so the exploration finishes because the stack is empty. In summary, the test's first execution covers MTW=10- (MTW=100 is executed), second execution covers MTW=-10 (MTW=010 is executed) and third execution covers MTW=-11 (MTW=011 is executed). Therefore, the technique covers all 6 valid configurations by executing just three configurations.

4.4.4 Reset Function

While a stateless exploration technique such as SPLat does not need to store and restore program state in the middle of execution like a stateful exploration technique does, the stateless exploration does need to be able to restart a new execution from the initial program state unaffected by the previous execution. Restarting an execution with a new runtime (*eg* spawning a new *Java Virtual Machine (JVM)* in Java) is the simplest solution, but it can be both inefficient and unsound. It is inefficient because even without restarting the runtime, the different executions may be able to share a runtime and still have identical initial program states, *eg* if the test does not update any static variables in the JVM state. It can be unsound because a new runtime may not reset the program state changes made by the pre-

vious executions (*eg* previous executions having sent messages to other computers or having performed I/O operations such as database updates). We address these issues by sharing the runtime between executions and requiring the user to provide a *reset function* that can be called at the beginning of the test.

Our sharing of the runtime between executions means that settings that would normally be reset automatically by creating a new runtime must now be manually reset. For example, Java static initializers must now be called from the reset function because classes are loaded only once. However, we believe that the benefit of saving time by reusing the runtime outweighs the cost of this effort, which could be alleviated by a program analysis tool. Moreover, for the Groupon code used in our evaluation, the testing infrastructure was already using the reset function (developed independently and years before this research); between any test execution, the state (of both memory and database) is reset (by rolling back the database transaction from the previous test and overwriting the state changes in the `tearDown` and/or `setUp` blocks after/before each test).

4.4.5 Potential Optimization

The algorithm in Figure 4.4 is not optimized in how it interfaces with the feature model. The feature model is treated as a blackbox, read-only artifact that is oblivious to the exploration state consisting of the `state` and `stack`. Consequently, the `isSatisfiable()` and `getValid()` methods are executed as if the exploration state was completely new every time, even if it just incrementally differs from the previous exploration state. For example, when running the test from Figure 4.2, `SPLat` asks the feature model if $MTW = -1-$ is satisfiable (line 30 of the `SPLat` algorithm) after the assignment $MTW = -0-$. The feature model replies `true` as it can find a configuration with the feature `TOOLBAR` set to true. Then when `WORDCOUNT` is encountered while `TOOLBAR=true`, `SPLat` asks the feature model

if the assignment $MTW = -10$ (`TOOLBAR=true` *and* `WORDCOUNT=false`) is satisfiable (line 43 of the `SPLat` algorithm). Note that the feature model is not aware of the similarity between the consecutive calls for $MTW = -1-$ and $MTW = -10$. But if it were, it would only have to check the satisfiability of `WORDCOUNT=false`.

The change to the algorithm to enable this synchronization between the exploration state and the feature model is simple: every time a feature variable is pushed on the `stack`, constrain the feature model with the feature's value, and every time a feature variable is popped off the `stack`, remove the corresponding feature assignment from the feature model. A feature model that can be updated implies that it should support incremental solving, *i.e.* a feature model should not have to always be solved in its entirety. Our current `SPLat` tool for Java does not exploit incremental solving, meaning that the tool has not reached the limits of the underlying technique and can be made even faster.

4.4.6 Implementation

Darko Marinov and I implemented `SPLat` for Java and Darko Marinov implemented it for Ruby on Rails. We selected these two languages motivated by the subject programs used in our experiments (Section 4.5).

For Java, we implemented `SPLat` on top of the publicly available `Korat` solver for imperative predicates [64]. Korat already provides code instrumentation (based on the BCEL library for Java bytecode manipulation) to monitor field accesses, and provides basic backtracking over the accessed fields. The feature variables in our Java subjects were already represented as fields. The main extension for `SPLat` was to integrate Korat with a SAT solver for checking satisfiability of partial assignments with respect to feature models. As mentioned earlier, we used SAT4J [86].

For Ruby on Rails, we have an even simpler implementation that only mon-

itors accesses to feature variables. We did not integrate a SAT solver, because the subject code did not have a formal feature model and thus we treated all combinations of feature variables as valid.

4.5 Evaluation

Our evaluation addresses the following research questions:

RQ1 How does `SPLat`'s efficiency compare with alternative techniques for analyzing SPL tests?

RQ2 What is the overhead of `SPLat`?

RQ3 Does `SPLat` scale to real code?

In Section 4.5.1, we compare `SPLat` with related techniques using 10 SPLs. In Section 4.5.2, we report on the evaluation of `SPLat` using an industrial configurable system implemented in Ruby on Rails.

4.5.1 Software Product Lines

We evaluate our approach with 10 SPLs listed in Table 4.1.[2] Note that most of these subjects are configurable programs that have been converted into SPLs. A brief description for each is below:

- **101Companies** [45] is a human-resource management system. Features include various forms to calculate salary and to give access to the users.
- **Email** [40] is an email application. Features include message encryption, automatic forwarding, and use of message signatures.

[2] All subjects except 101Companies have been used in previous studies on testing/analyzing SPLs, including GPL by [4, 20], Elevator, Email, MinePump by [4], JTopas by [21], Notepad by [59, 58], XStream by [30, 88] Prevayler by [95, 3], and Sudoku by [3].

- **Elevator** [78] is an application to control an elevator. Features include prevention of the elevator from moving when it is empty and a priority service to the executive floor.
- **GPL** [72] is a product line of graph algorithms that can be applied to a graph.
- **JTopas** [47] is a text tokenizer. Features include support for particular languages such as Java and the ability to encode additional information in a token.
- **MinePump** [65] simulates an application to control water pumps used in a mining operation. Features include sensors for detecting varying levels of water.
- **Notepad** [59] is a GUI application based on Java Swing that provides different combinations of end-user features, such as windows for saving/opening/printing files, menu and tool bars, etc. It was developed for a graduate-level course on software product lines.
- **Prevayler** [70] is a library for object persistence. Features include the ability to take snapshots of data, to compress data, and to replicate stored data.
- **Sudoku** [81] is a traditional puzzle game. Features include a logical solver and a configuration generator.
- **XStream** [71] is a library for (de)serializing objects to XML (and from it). Features include the ability to omit selected fields and to produce concise XML.

Table 4.1 shows the number of optional features (we do not count the mandatory features because they have constant values), the number of valid configurations, and the code size for each subject SPL. More details of the subjects and results are available at our website [61].

Table 4.1: Subject SPLs

SPL	*Features*	*Confs*	*LOC*
101Companies	11	192	2,059
Elevator	5	20	1,046
Email	8	40	1,233
GPL	14	73	1,713
JTopas	5	32	2,031
MinePump	6	64	580
Notepad	23	144	2.074
Prevayler	5	32	2,844
Sudoku	6	20	853
XStream	7	128	14,480

Tests

We prepared three different tests for each subject SPL. The first test, referred as `LOW`, represents an optimistic scenario where the test needs to be run only on a small number of configurations. The second test, referred as `MED` (for MEDIUM), represents the average scenario, where the test needs to be run on some configurations. The third test, referred as `HIGH`, represents a pessimistic scenario, where the test needs to be run on most configurations.

To prepare the LOW, MED, and HIGH tests, we modified existing tests, when available, or wrote new tests because we could not easily find tests that could be used without modification. Because some subjects were too simple, tests would finish too quickly for meaningful time measurement if test code only had one sequence of method calls. Therefore, we used loops to increase running times when necessary. Each test fixes all inputs except the feature variables. The tool, test suites and subjects are available on the project website [61].

Comparable Techniques

We compared `SPLat` with different approaches for test execution. We considered two naïve approaches that run tests against *all* valid configurations: NewJVM and

ReuseJVM. The ***NewJVM*** approach spawns a new JVM for each distinct test run. Each test run executes only *one* valid configuration of the SPL. It is important to note that the cost of this approach includes the cost of spawning a new JVM. The ***ReuseJVM*** approach uses the same JVM across several test runs, thus avoiding the overhead of repeatedly spawning JVMs for each different test and configuration. This approach requires the tester to explicitly provide a reset function (Section 4.4.4). Because the tester likely has to write a reset function anyway, we conjecture that this approach is a viable alternative to save runtime cost. For example, the tester may already need to restore parts of the state stored outside the JVM such as files or database.

We also compared `SPLat` with a simplified version of a previously proposed static analysis [58] for pruning configurations. Whereas [58] performs reachability analysis, control-flow and data-flow analyses, the simplified version, which we call ***SRA*** (Static Reachability Analysis), only performs the reachability analysis to determine which configurations are reachable from a given test and thus can be seen as the static analysis counterpart to `SPLat`. SRA builds a call graph using inter-procedural, context-insensitive, flow-insensitive, and path-insensitive points-to analysis and collects the features syntactically present in the methods of the call graph. Only the valid combinations of these *reachable* features from a test need to be run for that test.

Finally, we compared `SPLat` with an artificial technique that has zero cost to compute the set of configurations on which each test need to run. More precisely, we use a technique that gives the same results as `SPLat` but only counts the cost of executing tests for these configurations, not the cost of computing these configurations. We call this technique ***Ideal***. The overhead of `SPLat` is the difference between the overall cost of `SPLat` explorations and the cost of executing tests for Ideal.

Table 4.2: Experimental Results for Various Techniques

	All Valid		SPLat				Static Reachability (SRA)		
Test	*NewJVM*	*ReuseJVM*	*Confs*	*SPLatTime*	*IdealTime*	*Overhead*	*Confs*	*Overhead*	*Time*
101Companies (192 configs)									
LOW	35.46	2.13 (6%)	32 (16%)	1.64 (77%)	0.72	0.92 (127%)	96	84.04	1.28
MED	49.37	3.90 (7%)	160 (83%)	6.84 (175%)	3.58	3.26 (91%)	192	82.54	3.99
HIGH	283.69	45.26 (15%)	176 (91%)	47.6 (105%)	41.59	6.01 (14%)	192	81.93	45.16
Elevator (20 configs)									
LOW	10.74	5.17 (48%)	2 (10%)	1.33 (25%)	0.71	0.62 (87%)	2	23.29	0.76
MED	50.97	46.65 (91%)	10 (50%)	23.62 (50%)	23.14	0.48 (2%)	20	23.74	46.17
HIGH	62.57	59.48 (95%)	20 (100%)	60.71 (102%)	59.28	1.43 (2%)	20	24.38	60.43
Email (40 configs)									
LOW	40.63	10.74 (26%)	1 (2%)	1.00 (9%)	0.87	0.13 (14%)	1	23.62	0.87
MED	57.56	48.87 (84%)	30 (75%)	36.99 (75%)	37.14	-0.15 (0%)	40	22.81	49.02
HIGH	58.02	48.93 (84%)	40 (100%)	48.96 (100%)	49.26	-0.31 (0%)	40	23.84	49.16
GPL (73 configs)									
LOW	19.21	2.23 (11%)	6 (8%)	0.79 (35%)	0.29	0.49 (168%)	6	104.97	0.30
MED	190.53	171.62 (90%)	55 (75%)	130.87 (76%)	128.52	2.35 (1%)	55	99.41	128.69
HIGH	314.20	285.89 (90%)	70 (95%)	278.77 (97%)	277.48	1.29 (0%)	73	103.52	286.28
JTopas (32 configs)									
LOW	26.59	16.83 (63%)	8 (25%)	6.29 (37%)	4.49	1.80 (40%)	32	86.87	16.44
MED	29.04	18.55 (63%)	16 (50%)	13.16 (70%)	9.71	3.46 (35%)	32	86.87	18.70
HIGH	28.92	18.93 (65%)	32 (100%)	25.31 (133%)	18.43	6.88 (37%)	32	86.87	18.48
MinePump (64 configs)									
LOW	23.71	7.53 (31%)	9 (14%)	3.65 (48%)	1.90	1.75 (91%)	64	22.69	7.49
MED	59.72	14.78 (24%)	24 (37%)	10.43 (70%)	6.26	4.17 (66%)	64	22.38	15.35
HIGH	13.72	5.75 (41%)	48 (75%)	37.80 (657%)	4.81	32.99 (685%)	64	22.18	5.77
Notepad (144 configs)									
LOW	398.22	135.60 (34%)	2 (1%)	3.06 (2%)	2.45	0.61 (24%)	144	80.40	135.47
MED	418.23	156.27 (37%)	96 (66%)	104.95 (67%)	104.91	0.04 (0%)	144	80.62	156.35
HIGH	419.99	153.39 (36%)	144 (100%)	153.11 (99%)	152.16	0.94 (0%)	144	81.29	151.94
Prevayler (32 configs)									
LOW	65.34	40.23 (61%)	12 (37%)	22.49 (55%)	22.8	-0.31 (-1%)	32	205.54	45.39
MED	121.38	96.50 (79%)	24 (75%)	102.49 (106%)	105.86	-3.37 (-3%)	32	214.67	111.37
HIGH	149.08	120.7 (80%)	32 (100%)	127.17 (105%)	131.37	-4.20 (-3%)	32	290.66	135.61
Sudoku (20 configs)									
LOW	51.11	48.10 (94%)	4 (20%)	42.72 (88%)	24.12	18.6 (77%)	10	31.87	24.28
MED	118.14	105.67 (89%)	10 (50%)	58.31 (55%)	54.16	4.15 (7%)	10	31.75	53.67
HIGH	489.60	334.82 (68%)	20 (100%)	316.47 (94%)	332.36	-15.89 (-4%)	20	31.74	338.48
Xstream (128 configs)									
LOW	111.26	30.04 (27%)	2 (1%)	1.57 (5%)	1.08	0.49 (45%)	2	106.50	1.06
MED	105.10	9.04 (8%)	64 (50%)	5.77 (63%)	5.26	0.51 (9%)	64	109.22	5.14
HIGH	101.66	8.68 (8%)	128 (100%)	9.16 (105%)	8.59	0.57 (6%)	128	105.68	8.74

Results

Table 4.2 shows our results. We performed the experiments on a machine with X86_64 architecture, Ubuntu operating system, 240696 MIPS, 8 cores, with each core having an Intel Xeon CPU E3-1270 V2 at 3.50GHz processor, and 16 GB memory. All times are listed in seconds. Our feature model implementation solves the feature model upfront to obtain all valid configurations; because this solving needs to be done for every feature model (regardless of using `SPLat` or otherwise), and because it takes a fraction of test execution time, we do not include it.

Here is a description of each column in Table 4.2:

- **Test** refers to one of the three categories of tests described earlier.
- **All Valid** identifies the techniques that run the test against all valid configurations, namely **NewJVM** and **ReuseJVM**. **ReuseJVM** shows time absolutely and as a percentage of **NewJVM** duration.
- Columns under **`SPLat`** details information for `SPLat`:
  - **Confs** shows the number of configurations that `SPLat` runs for a particular test.
  - **SPLatTime** shows the time it takes to run a test using `SPLat`. `SPLat` reuses the same JVM for different executions, like ReuseJVM. The time is shown absolutely and as a percentage of **ReuseJVM** (not **NewJVM**).
  - **IdealTime** shows the time in seconds for running `SPLat` without considering the cost to determine which configurations to run for the test; therefore, this number excludes instrumentation, monitoring, and constraint solving.
  - **Overhead** shows the overhead of `SPLat`, calculated by subtracting **IdealTime** from **SPLatTime**, and dividing it by **IdealTime**.
- Columns under **Static Reachability (SRA)** show results for our static analysis:

– **Confs** shows the number of configurations reachable with such analysis,

– **Overhead** shows the time taken to perform the static reachability analysis, and
– **Time** shows the time taken to run the configurations determined by this analysis.

Efficiency. The **ReuseJVM** column shows that reusing JVM saves a considerable amount of time compared to spawning a new JVM for each test run. For example, for half of the tests, reusing JVM saves over 50% of the time, because running these tests does not take much longer than starting up the JVM. For tests that take considerably longer than starting up the JVM, such saving is not possible.

`SPLat` further reduces the execution time over **ReuseJVM** by determining the reachable configurations. For example, for the `LOW` test for `Notepad`, reusing the JVM takes 34% of the time to run with a new JVM, and with `SPLat`, it takes just 2% of the already reduced time. In fact, the table shows that in most cases, as long as `SPLat` can reduce the number of configurations to test (*i.e.* **Confs** is lower than the total number of configurations), it runs faster than running each configuration (*i.e.* less than 100% of **ReuseJVM**).

Comparison with Static Reachability Analysis. The static reachability analysis yields less precise results compared to `SPLat`: the number of configurations in the column **Confs** is larger than the number of configurations in the corresponding column for `SPLat`. In fact, for `JTopas`, `Notepad` and `MinePump`, the SRA reports all features as being accessed from the call graph, and therefore reports that all valid configurations have to be tested. For example, for `JTopas`, this is due to its tests invoking the `main` method of the SPL, from which all feature variable accesses may be reached using different input values, which the analysis is insensitive to. For `Notepad`, this is due to the the use of the `FEST` automated GUI testing

framework, which relies heavily on reflection. Because the method being invoked through reflection cannot necessarily be determined statically, the analysis yields a conservative result. For `MinePump`, each test happens to exercise a sequence of methods that together reach all feature variable accesses.

Note that the SRA approach first statically determines the configurations to run (which takes the time in column **SRA Overhead**) and afterwards dynamically runs them one by one (which takes the time in column **SRA Time**). Comparing just the static analysis time (**SRA Overhead**) with the `SPLat` overhead (**SPLat Overhead**) shows that SRA has a considerably larger overhead, in some cases two orders of magnitude larger. Although the static analysis overhead can be offset by (re)using the reachable configurations it determines against tests that have the same code base but have different inputs, in general, it would require a very large number of such tests for the approach to have a smaller overhead than `SPLat`. Moreover, comparing just the time to execute the configurations computed by SRA (column **SRA Time** with the time to execute the configurations computed by `SPLat` (column **IdealTime**) shows that SRA again takes longer because SRA computes a higher number of configurations than `SPLat` due to the conservative nature of static analysis.

RQ1. Based on the comparison with **NewJVM**, **ReuseJVM**, and **SRA**, we conclude the following:

> `SPLat` is more efficient than the techniques that run all valid configurations for tests of SPLs or prune reachable configurations using static analysis. Moreover, compared with static analysis, `SPLat` not only gives results faster but also gives more precise results.

Overhead. Table 4.2 also shows the overhead that `SPLat` has over the Ideal technique (column **SPLat Overhead**). The overhead is generally small, except for the `LOW` tests and tests for several subjects (*eg* `JTopas` and `Mine`). The overhead is

high for the LOW tests because these tests finish quickly (under 7 seconds, often under 1 second), meaning that instrumentation, monitoring and feature model interaction take a larger fraction of time than they would for a longer executing test. The overhead is high for JTopas because the feature variables are accessed many times because they are accessed within the tokenizing loop. The overhead is high for MinePump because feature accesses and their instrumentation take relatively longer to execute for this particular test as the subject is very small.

SPLat, due to its cost in monitoring feature variables, should not execute a test faster than knowing the reachable configurations upfront and running the test only on those configurations. Thus, the occasional small negative overheads for Email, Prevayler, Sudoku are due to the experimental noise and/or the occasionally observed effect where an instrumented program runs faster than the non-instrumented program. It is important to note that efficiency and overhead are orthogonal. As long as the reduction in time due to the reduction in configurations is larger than the overhead, SPLat saves the overall time. To illustrate, the GPL's LOW test incurs over 168% overhead, but the reduction in configurations outweighs the overhead, and SPLat takes only 35% of running all valid configurations with the same JVM.

RQ2. Based on the discussion about overhead, we conclude the following:

SPLat can have a large relative overhead for short-running tests, but the overhead is small for long-running tests.

4.5.2 Configurable Systems

Groupon. Groupon is a company that "features a daily deal on the best stuff to do, see, eat, and buy in 48 countries" (http://www.groupon.com/about).

Groupon PWA is name of the codebase that powers the main groupon.com website. It has been developed for over 4.5 years with contributions from over 250 engineers. The server side is written in Ruby on Rails and has over 171K lines of Ruby code.

Groupon PWA code is highly configurable with over 170 (boolean) feature variables. In theory, there are over 2^{170} different configurations for the code. In practice, only a small number of these configurations are ever used in production, and there is one default configuration for the values of all feature variables.

Groupon PWA has an extensive regression testing infrastructure with several frameworks including Rspec, Cucumber, Selenium, and Jasmine. The test code itself has over 231K lines of Ruby code and additional code in other languages. (It is not uncommon for the test code to be larger than the code under test [96].)

Groupon PWA has over 19K Rspec (unit and integration) tests. A vast majority of these tests run the code only for the default configuration. A few tests run the code for a non-default configuration, typically changing the value for only one feature variable from the default value. Running all the Rspec tests on a cluster of 4 computers with 24 cores each takes under 10 minutes.

`SPLat` Application. Darko Marinov implemented `SPLat` for Ruby on Rails and applied it to Groupon PWA. He did not have to implement the reset function because it was already implemented by Groupon testers to make test execution feasible (due to the high cost of re-starting the system). Moreover, no explicit feature model was present, so feature model constraints did not need to be solved.

We set out to evaluate how many configurations each test could cover if we allow varying the values of all feature variables encountered during the test run. We expected that the number of configurations could get extremely high for some tests to be able to enumerate all the configurations. Therefore, we set the limit on the number of configurations to no more than 16, so that the experiments finished in a reasonable time. This limit was reached by 2,695 tests. For the remaining

Table 4.3: Reachable Configurations

Configs	*Tests*	*Configs*	*Tests*
1	11,711	2	1,757
3	332	4	882
5	413	6	113
7	19	8	902
9	207	10	120
11	29	12	126
13	6	14	32
15	10	16	349
17	2,695	-	-

17,008 tests, Table 4.3 shows the breakdown of how many tests reached a given number of configurations. We can see that the most common cases are the number of configurations being powers of two, effectively indicating that many features are encountered independently rather than nested (as in Figure 4.2 where the read of `WORDCOUNT` is nested within the block controlled by the read of `TOOLBAR`).

We also evaluated the number of features encountered. It ranges from 1 up to 43. We found 43 is a high number in the absolute sense (indicating that a test may potentially cover 2^{43} different configurations), 43 is also a relatively low number in the relative sense compared to the total of over 170 features. Table 4.4 shows the breakdown of how many tests reached a given number of feature variables. Note that the numbers of configurations and feature variables may seem inconsistent at a glance, *eg* the number of tests that have 1 configuration is larger than the number of tests than have 0 feature variables. The reason is that some tests force certain values for feature variables such that setting the configuration gets overwritten by the forced value.

In summary, these results show that the existing tests for Groupon PWA can already achieve a high coverage of configurations, but running all the configurations for all the tests can be prohibitively expensive. We leave it as a future work to explore a good strategy to sample from these configurations [26, 27, 74].

Table 4.4: Accessed Features

Vars	*Tests*	*Vars*	*Tests*	*Vars*	*Tests*
0	11,711	1	1,757	2	1,148
3	1,383	4	705	5	389
6	466	7	323	8	425
9	266	10	140	11	86
12	80	13	34	14	28
15	54	16	62	17	1
19	14	20	260	21	109
22	45	23	19	24	22
25	9	26	2	27	14
28	17	29	6	30	8
31	24	32	6	33	14
34	31	35	11	36	15
37	8	38	2	39	2
40	3	42	2	43	2

RQ3. Moreover, based on the fact that we could run `SPLat` on the codebase as large as Groupon PWA, we conclude the following:

> `SPLat` scales to real, large industrial code. The implementation effort for `SPLat` is relatively low and the number of configurations covered by many real tests is relatively low.

4.5.3 Threats to Validity

The main threat is to external validity: we cannot generalize our timing results to all SPLs and configurable systems because our case studies may not be representative of all programs, and our tests may be covering an unusual number of configurations. To reduce this threat, we used multiple Java SPLs and one real, large industrial codebase. For SPLs, we designed tests that cover a spectrum of cases from `LOW` to `MED(IUM)` to `HIGH` number of configurations. For the Groupon codebase, we find that most real tests indeed cover a small number of configurations. Our study has the usual internal and construct threats to validity.

We believe that SPLat is a helpful technique that can be used in practice to improve SPL testing. An important threat to this conclusion is that our results do not take into account the cost of writing a reset function. Although other techniques that use stateless exploration also require reset functions (*eg* VeriSoft [37]), the cost of developing such functions could affect practicality. NewJVM, ReuseJVM, and SPLat all require the state outside of the JVM to be explicitly reset, but only NewJVM automatically resets JVM-specific state by spawning a new JVM for each test run.

4.6 Related Work

4.6.1 Dynamic Analysis

Korat. SPLat was inspired by Korat [18], a test-input generation technique based on Java predicates. Korat instruments accesses to object fields used in the predicate, monitors the accesses to prune the input space of the predicate, and enumerates those inputs for which the predicate returns true. Directly applying Korat to the problem of reducing the combinatorics in testing configurable systems is not feasible because the feature model encodes a *precondition* for running the configurable system, which must be accounted for. In theory, one could automatically translate a (declarative) feature model into an imperative constraint and then execute it before the code under test, but it could lead Korat to explore the *entire* space of feature combinations (up to 2^N combinations for N features) before *every* test execution. In contrast, SPLat exploits feature models while retaining the effectiveness of execution-driven pruning by applying it with SAT in tandem. Additionally, SPLat can change the configuration being run during the test execution (line 44 in Figure 4.4), which Korat did not do for data structures.

Shared execution. Starting from the work of d'Amorim *et al.* [29], there

has been considerable ongoing research on saving testing time by sharing computations across similar test executions [4, 5, 24, 29, 43, 51, 60, 63, 82, 97]. The key observation is that repeated executions of a test have much computation in common. For example, Shared Execution [60] runs a test simultaneously against several SPL configurations. It uses a *set* of configurations to support test execution, and splits and merges this set according to the different decisions in control-flow made along execution. The execution-sharing techniques for testing SPLs differ from `SPLat` in that they use *stateful* exploration; they require a dedicated runtime for saving and restoring program state and only work on programs with such runtime support. Consequently, they have high runtime overhead not because of engineering issues but because of fundamental challenges in splitting and merging state sets at proper locations. In contrast, `SPLat` uses *stateless* exploration [37] and never merges control-flow of different executions. Although `SPLat` cannot share computations between executions, it requires minimal runtime support and can be implemented very easily and quickly against almost any runtime system that allows feature variables to be read and set during execution.

Sampling. Sampling exploits domain knowledge to select configurations to test. A tester may choose features for which all combinations must be examined, while for other features, only t-way (most commonly 2-way) interactions are tested [26, 27, 74]. Our dynamic program analysis *safely* prunes feature combinations, while sampling approaches can miss problematic configurations [4].

Spectrum of SPL testing techniques. Kästner *et al.* [51] define a spectrum of SPL testing techniques based on the amount of changes required to support testing. On the one end are black-box techniques that use a conventional runtime system to run the test for each configuration; `NewJVM` is such a technique. On the other end are white-box techniques that extensively change a runtime system to make it SPL-aware; shared execution is such a technique. `SPLat`, which only re-

quires runtime support for reading and writing to feature variables, is a lightweight white-box technique that still provides an optimal reduction in the number of configurations to consider.

4.6.2 Static Analysis

Chapter 3 presented a static analysis that performs reachability, data-flow and control-flow checks to determine which features are relevant to the outcome of a test. The analysis enables one to run a test only on (all valid) combinations of these relevant features that satisfy the feature model. `SPLat` is only concerned with reachability, so even if it encounters a feature whose code has no effect, it will still execute the test both with and without the feature. But a large portion of the reduction in configurations in running a test is simply due to the idea that many of the features are not even reachable. Indeed, as Section 4.5 shows, `SPLat` determines reachable configurations with much greater precision and is likely to be considerably faster than the static analysis because `SPLat` discovers the reachable configurations during execution. Static analysis may be faster if its cost can be offset against many tests (because it needs only be run once for one test code that allows different inputs), and if a test run takes a very long time to execute (*eg* requiring user interaction). But such situations do not seem to arise often, especially for tests that exercise a small subset of the codebase.

4.7 Summary

`SPLat` is a new technique for reducing the combinatorics in testing configurable systems. `SPLat` dynamically prunes the space of configurations that each test must be run against. `SPLat` achieves an optimal reduction in the number of configurations and does so in a lightweight way compared to previous approaches based on static analysis and heavyweight dynamic execution. Experimental results on 10

software product lines written in Java show that SPLat substantially reduces the total test execution time in most cases. Moreover, our application of SPLat on a large industrial code written in Ruby on Rails shows its scalability.

Chapter 5

Statically Reducing Configurations to Monitor in a Software Product Line

The contents of this chapter appeared in the 2010 Conference on Runtime Verification (RV 2010) [59]. [1]

5.1 Introduction

In this chapter, we consider the problem of runtime-monitoring SPLs for *safety property* [87] violation, i.e. dynamically observing program behavior to check conformance to expected properties. Our technique statically identifies feature combinations (i.e., programs) that provably can never violate the stated property. Thus, these programs do not need to be monitored. Achieving this reduction is beneficial

[1] Chang Hwan Peter Kim, Eric Bodden, Don S. Batory, and Sarfraz Khurshid. Reducing Configurations to Monitor in a Software Product Line. In Runtime Verification (RV), 2010. The paper extends my individual project work for a Computer Security course at UT-Austin. Eric assisted with implementation and paper writing. Don and Sarfraz helped with paper writing.

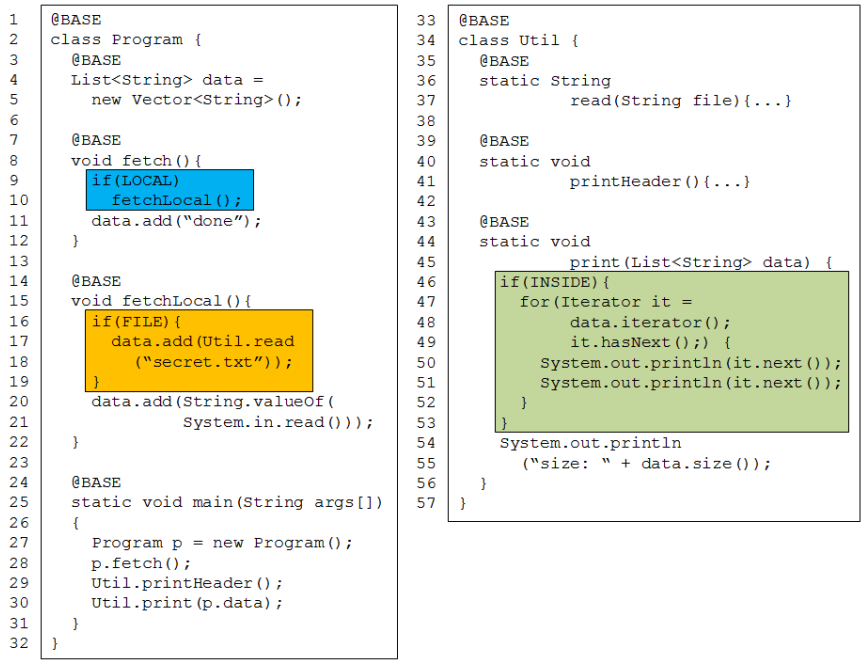
in two settings under which monitors are used. First, it can significantly speed up the testing process as these programs do not need to be run to see if the property can be violated. Second, if the monitor is used in production, it can speed up these programs because they are not monitored unnecessarily.

We accomplish this goal by starting with analyses that evaluate runtime monitors at compile time for *single* programs [14, 32, 15]. Our work extends these analyses by lifting them to understand features, making them aware of possible feature combinations. A programmer applies our analysis to an SPL once at each SPL release. The output is a bi-partitioning of feature combinations: (1) configurations that need to be monitored because violations may occur and (2) configurations for which no violation can happen.

To validate our work, we analyze two different Java-based SPLs. Experiments show we can statically rule out over half of the configurations for these case studies. Further, analyzing an entire SPL is not much more expensive than applying the earlier analyses to a single program.

To summarize, the contributions of this chapter are:

- **Technique.** A novel static analysis to determine, for a given SPL and runtime-monitor specification, the feature combinations (programs) that require monitoring,
- **Implementation.** An implementation of this analysis within the CLARA framework for hybrid typestate analysis [12], as an extension to Bodden et al.'s earlier whole-program analysis [32], and
- **Evaluation.** Experiments that show that our analysis noticeably reduces the number of configurations that require runtime-monitoring and thus saves testing time and program execution time for the programs studied.



```
1  @BASE
2  class Program {
3    @BASE
4    List<String> data =
5      new Vector<String>();
6
7    @BASE
8    void fetch(){
9      if(LOCAL)
10       fetchLocal();
11     data.add("done");
12   }
13
14   @BASE
15   void fetchLocal(){
16     if(FILE){
17       data.add(Util.read
18         ("secret.txt"));
19     }
20     data.add(String.valueOf(
21               System.in.read()));
22   }
23
24   @BASE
25   static void main(String args[])
26   {
27     Program p = new Program();
28     p.fetch();
29     Util.printHeader();
30     Util.print(p.data);
31   }
32 }
```

```
33 @BASE
34 class Util {
35   @BASE
36   static String
37            read(String file){...}
38
39   @BASE
40   static void
41            printHeader(){...}
42
43   @BASE
44   static void
45            print(List<String> data) {
46     if(INSIDE){
47       for(Iterator it =
48            data.iterator();
49            it.hasNext();) {
50         System.out.println(it.next());
51         System.out.println(it.next());
52       }
53     }
54     System.out.println
55       ("size: " + data.size());
56   }
57 }
```

Figure 5.1: Example Product Line

5.2 Motivating Example

Figure 5.1 shows a simple example SPL, whose programs fetch and print data.[2] Local data is fetched if the `Local` feature is selected (blue code), local data from a file is fetched if `File` is selected (yellow code) and internal contents of data are printed if `Inside` is selected (green code). Each member (class, field, or method) is annotated with a feature. In this example, every member is annotated with `Base` feature, meaning that it will be present in a program only if the `Base` feature is selected.

The feature model for our SPL is shown in Figure 5.2. `Base` is a required feature. `Inside`, `File`, and `Local` are optional features. The model further requires at least one of the optional features to be selected (second line). In total, the

[2]For presentation, we omit the class of field references in feature-conditionals and capitalize feature identifiers.

feature model allows seven distinct programs (eight variations from three optional features then remove the case without any optional feature).

```
Example :: [Inside] [File] [Local] Base;
Inside or File or Local;
```

Figure 5.2: Example Feature Model

5.2.1 Example Monitor Specifications: ReadPrint and HasNext

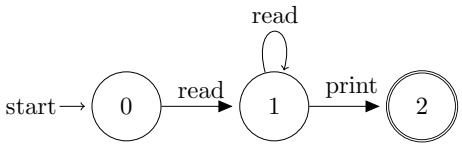
Researchers have developed a multitude of specification formalisms for defining runtime monitors. As our approach extends the CLARA framework, it can generally apply to any runtime-monitoring approach that uses AspectJ aspects for monitoring. This includes popular systems such as JavaMOP [22] and tracematches [2]. For the remainder of this chapter, we use the tracematch notation because it can express monitors concisely. Figure 5.3(a) shows a simple example. `ReadPrint` prevents a `print` event after a `read` event is witnessed. In line 3 of Figure 5.3(a), a `read` symbol captures all those events in the program execution, known as *joinpoints* in AspectJ terminology, that are immediately *before* calls to `Util.read*(..)`. Similarly, the symbol `print` captures joinpoints occurring immediately *before* calls to `Util.print*(..)`. Line 6 carries the simple regular expression "`read+ print`", specifying that code body in lines 6–8 should execute whenever a `print` event follows one or more `read` events on the program's execution. Figure 5.3(b) shows a finite-state machine for this tracematch, where symbols represent transitions.

Figure 5.4 shows another safety property, `HasNext` [32], which checks for iterators if `next()` is called twice without calling `hasNext()` in between. Note that this tracematch only matches if the two `next()` calls bind to the same `Iterator` object `i`, as shown in Figure 5.4(a), lines 2–4. When the tracematch encounters an event matched by a declared symbol that is not part of the regular expression, such as `hasNext`, the tracematch discards its partial match.

```
aspect ReadPrint {
  tracematch() {
    sym read before: call(* Util.read*(..));
    sym print before: call(* Util.print*(..));

    read+ print {
      throw new RuntimeException(``ReadPrint violation!'');
    }
  }
}
```

(a) ReadPrint Tracematch



(b) Finite-State Machine

Figure 5.3: ReadPrint Safety Property

Therefore, the tracematch would match a trace "`next(i1) next(i1)`" but not "`next(i1) hasNext(i1) next(i1)`", which is exactly what we seek to express.

A naive approach to runtime-monitoring would insert runtime monitors like `ReadPrint` and `HasNext` into every program of a product line. However, as we mentioned, it is often unnecessary to insert runtime monitors into some programs because these programs provably cannot trigger the runtime monitor.

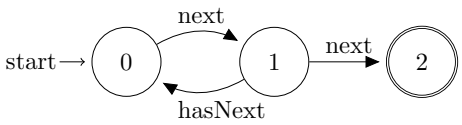
5.2.2 Analysis by Example

Our goal is to statically determine the feature configurations to monitor, or conversely the configurations that cannot trigger the monitor. For our running example, let us first deduce these configurations by hand. For `ReadPrint`, both `read` and `print` symbols have to match, meaning that `File` (which calls `read(..)` in line 17) and `Base` (which calls `print*(..)` in lines 29 and 30) have to be present for the monitor to trigger. Also, `Local` needs to be present because it enables `File`'s code to be reached. Therefore, the `ReadPrint` monitor has to be inserted if and

```
aspect HasNext {
  tracematch(Iterator i) {
    sym next    before: call(* Iterator.next()) && target(i);
    sym hasNext before: call(* Iterator.hasNext()) && target(i);

    next next {
      throw new RuntimeException(``HasNext violation!'');
    }
  }
}
```

(a) HasNext Tracematch



(b) Finite-state machine

Figure 5.4: HasNext Safety Property [32]

only if these three features are present, which only holds for two out of the seven original configurations.

We represent the condition under which a monitor has to be inserted by treating a monitor, e.g. `ReadPrint`, as a feature itself and constructing its *presence condition*: `ReadPrint iff (File and Local and Base)`. Similarly, the monitor for `HasNext` only has to be inserted iff `Iterator.next()` can be called, i.e., on the four configurations with `Inside` and `Base` present. The presence condition for `HasNext` is `HasNext iff (Inside and Base)`. The goal of our technique is to extend the original feature model so that tracematch monitors are now features and the tracematch presence conditions are part of the revised feature model (the extension is shown in italics):

```
// ReadPrint and HasNext are now features themselves
Example :: [ReadPrint] [HasNext] [Inside] [File] [Local] Base;
Inside or File or Local;

// Tracematch presence conditions
ReadPrint iff (File and Local and Base);
HasNext iff (Inside and Base);
```

Note that, although a tracematch is itself a feature which can be selected or not, it is different from other features in that its selection status is determined *not* by the user, but instead by the presence or absence of other features.

5.2.3 The Need for a Dedicated Static Analysis for Product Lines

As mentioned earlier, there exist static analyses that improve the runtime performance of a monitor by reducing its instrumentation of a single program [14, 32, 15]. We will refer to these analyses as *traditional program analyses (TPA)*. There are two ways to apply such analyses to product lines. One way is inefficient, the other way imprecise. Running TPA against each instantiated program will be very inefficient because it will have to inspect every program of the product line separately. The other way is to run TPA against the product line itself. This is possible because a product line in a SysGen program representation can be treated as an ordinary program (recall that a SysGen program uses ordinary program constructs like if-conditionals, rather than pre-processor constructs like `#ifdefs`, to represent variability). However, this second way will be imprecise. For example, suppose we apply TPA on the `ReadPrint` and `HasNext` tracematches for our example SysGen program: both tracematches may match in the case in which all features are enabled. Being oblivious to the notion of features, the analysis will therefore report that the tracematches have to be present for every program of the product line. This shows that a static analysis, to be both efficient and effective on an SPL, has to be aware of the SPL's features.

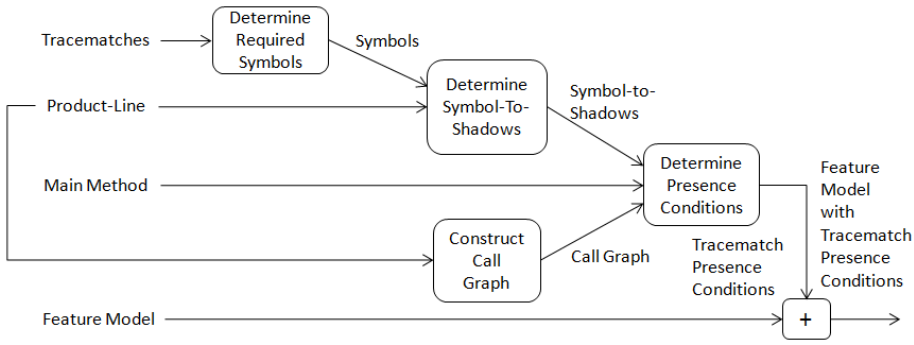


Figure 5.5: Overview of Our Technique

5.3 Product Line Aware Static Analysis

Figure 5.5 displays an overview of our approach. First, for a tracematch, our analysis determines the symbols required for the tracematch to trigger ("Determine Required Symbols"). For each of these symbols, we use the aspect weaver to identify the statements that are matched ("Determine Symbol-To-Shadows"). We elaborate on these two steps in Section 5.3.1. Then, for each of the matched statements, we determine the feature combinations that allow the statement to be reachable from the program's `main()` method. This results in a set of *presence conditions*. We combine all these conditions to form the presence condition of the tracematch. We repeat the process for each tracematch ("Determine Presence Conditions") and add the tracematches and their presence conditions to the original feature model ("+"). We explain these steps in Section 5.3.2.

5.3.1 Required Symbols and Shadows

A safety property must be monitored for a feature configuration c if the code in c may drive the finite-state monitor from its initial state to its final (error) state. In earlier work [32], Bodden et al. described three different algorithms that try to determine, with increasing levels of detail, whether a single program can drive

a monitor into an error state, and using which transition statements. The first, called *Quick Check*, rules out a tracematch if the program does not contain transition statements required to reach the final automaton state. The second, called *Consistent-Variables Analysis*, performs a similar check on every consistent variable-to-object binding. The third, called *Active-Shadows Analysis*, is flow-sensitive and rules out a tracematch if the program cannot execute its transition statements in a property-violating order.

In this chapter, we limit ourselves to extending the Quick Check to SPLs. The Quick Check has the advantage that, as the name suggests, it executes quickly. Nevertheless, our results show that even this relatively pragmatic analysis approach can noticeably reduce the number of configurations that require monitoring. It should be possible to extend our work to the other analyses that Bodden et al. proposed, but doing so would not fundamentally alter our technique.

Required Symbols

A symbol represents a set of transition statements with the same label. Given a tracematch, we determine the *required symbols*, i.e., the symbols required to reach the error state, by fixing one symbol s at a time and checking whether removing all automaton edges labeled with s prevents the final state from being reached. For any given program p, if there exists a required symbol s for which p contains no s-transition, then p does not have to be monitored. For the `ReadPrint` property, the symbols `read` and `print` are required because without one of these, the final state in Figure 5.3(b) cannot be reached. For the `HasNext` property, only the symbol `next` is required. This is because one can reach the final state without seeing a `hasNext`-transition. If a tracematch has no required symbol, e.g. $a \mid b$ (either symbol will trigger the monitor, meaning that neither is required), it has to

be inserted in all programs of the product line.[3]

Symbol-to-Shadows

For each required symbol, we determine its *joinpoint shadows* (*shadows* for short), i.e., all program statements that may cause events that the symbol matches. We implemented our analysis as an extension of the CLARA framework. CLARA executes all analyses right after the advice-matching and weaving process has completed. Executing the analysis after weaving has the advantage that the analysis can take the effects of all aspects into account. This allows us to even handle cases correctly in which a monitoring aspect itself would accidentally trigger a property violation. A re-weaving analysis has access to the weaver, which in turn gives detailed information about all joinpoint shadows.

In the `ReadPrint` tracematch, the `read` symbol's only shadow is the `read-("secret.txt")` call in line 17 of Figure 5.1 and the `print` symbol's shadows are the calls `printHeader()` in line 29 and `print(p.data)` call in line 30. For the `HasNext` tracematch, the `next` symbol's shadows are the `next()` calls in lines 50 and 51, and the `hasNext` symbol's only shadow is the `hasNext()` call in line 49.

5.3.2 Presence Conditions

A tracematch monitor must be inserted into a configuration when each of the tracematch's required symbols is present in the configuration. The *presence condition (PC)* of a tracematch is thus the conjunction of the presence condition of each of its required symbols. In turn, a symbol is present if any one of its shadows is present. Thus, the PC of a symbol is the disjunction of the PC of each of its shadows. The

[3] In practice, such a tracematch will be rare because the regular expression is generally used to express a sequence of events (meaning one of the symbols will be required), rather than a disjunction of events, which is typically expressed through a pointcut.

```
ReadPrint iff (pc(read) and pc(print))
ReadPrint iff ((pc(line17)) and (pc(line29) or pc(line30)))
ReadPrint iff (([File and Base]) and ([Base] or [Base]))
ReadPrint iff (File and Base)
```

Figure 5.6: Computing ReadPrint's Presence Condition

PC of a shadow is the conjunction of features that are needed for that shadow to appear in an SPL program. A first attempt to compute the PC of a tracematch is therefore:

```
tracematch iff (pc(reqdSymbol_1) and ... and pc(reqdSymbol_n))
pc(symbol_i) = pc(shadow_i1) or ... or pc(shadow_im)
pc(shadow_j) = feature_j1 and ... and feature_jk
```

For example, Figure 5.6 shows how we determine the PC of the `ReadPrint` tracematch. The required symbols of this tracematch are `read` and `print`. `read` has one shadow in line 17 of Figure 5.1 and `print` has two shadows in lines 29 and 30. For the shadow in line 17 to be syntactically present in a program, the `if(FILE)` conditional in line 16 must be `true` and the `fetchLocal()` method definition (annotated with `BASE` in line 14) must be present. That is, `pc(line17) = [File and Base]`. Similarly, `pc(line29)` and `pc(line30)` are each expanded into `[Base]` because each of the shadows just requires `BASE`, which introduces the `Program` class and its `main`-method definition.

The solution in Figure 5.6 is imprecise in that it allows configurations where a shadow is syntactically present, but not necessarily reachable from the `main` method. For example, according to the algorithm, the `read(..)` shadow (line 17) is "present" in configurations {`Base=true`, `Local=false`, `File=true`, `Inside=DONT_CARE`} even though it is not reachable from `main` due to `Local` being turned off. Based on this observation, the algorithm that we implemented can take into account the shadow's callers in addition to its syntactic containers. The algorithm therefore conjoins a shadow's imprecise PC with the disjunction of precise PC of each of its callers, recursively. For the line 17 shadow, which is called

by line 10, which is in turn called by line 28, this precise algorithm would return:

```
pc(line17) = [enclosingFeatures and (pc(caller1) or ... or pc(caller_m))]
           = [enclosingFeatures and (pc(line10))]
           = [enclosingFeatures and
                 (enclosingFeaturesLine10 and (pc(line28)))]
           = [File and Base and (Local and Base and (Base))]
           = File and Local and Base
```

Substituting this in Figure 5.6, we get `ReadPrint iff (File and Local and Base)`, which is optimal for our example and, as mentioned in Section 5.2.2, is what we set out to construct. Similarly, `HasNext`'s presence condition is:

```
HasNext iff (pc(next))
HasNext iff (pc(line50) or pc(line51))
HasNext iff ([Inside and Base and (Base)] or [Inside and Base and (Base)])
HasNext iff (Inside and Base)
```

Note that, even though `HasNext` is more localized than `ReadPrint`, i.e., in one optional feature (`Inside`) as opposed to two optional features (`File` and `Local`), it is required in more configurations (4 out of 7) than `ReadPrint` (2 out of 7).[4] This is because the feature model allows fewer configurations with both `Local=true` and `File=true` than configurations with just `Inside=true`.

There may be shadows that can only be reached through a cyclic edge in a call-graph. Rather than including the features controlling the cyclic edge in the presence condition of such a shadow, for simplicity, we ignore the cyclic edge. This is not optimally precise but sound. For example, `Util.read(..)` call in Figure 5.7 is actually only present in an execution if the execution traverses the cyclic edge from `c()` to `a()`, which is possible only if `X=true`. Instead of adding this constraint on `X` to the presence condition of `Util.read(..)`, we simply insert the monitor for both values of `X`.

[4] `Base` is a required feature according to the feature model.

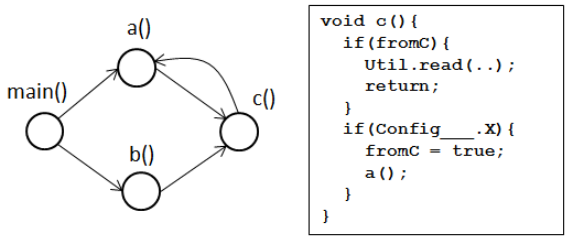


Figure 5.7: Example of Computing a Presence Condition with Cycles in the Call-Graph

5.3.3 Precision on a Pay-As-You-Go Basis

While considering the callers of a shadow makes its presence condition more precise, doing so is entirely optional for the following reason: without considering the callers, a shadow will simply be considered to exist both when a caller is present and when a caller is not present, which will insert a monitor even if a required symbol's shadow cannot be reached. For example, it would be sound, although not optimally precise, to return the imprecise presence condition of the shadow at line 17. But users of our approach can even go beyond that. Our analysis is pessimistic, i.e., starts from a sound but imprecise answer that ignores the call graph and then gradually refines the answer by inspecting the call graph. Therefore, our analysis can report a sound intermediate result at any time and after a certain number of call sites have been considered, we can simply stop going farther in the call-graph, trading precision for less computation time and resources. Being able to choose the degree of precision is useful especially because the call graph can be very large, which can make computing the presence condition expensive both time-wise and memory-wise. Our technique works with any kind of call graph. In our evaluation, we found that even simple context-insensitive call graphs constructed from *Spark* [67] are sufficient.

5.4 Evaluation

We implemented our analysis as an extension of the CLARA framework for hybrid typestate analysis [12] and evaluated it on the following SPLs: *Graph Product Line (GPL)*, a set of programs that implement different graph algorithms [72] and *Notepad*, a Java Swing application with functionality similar to Windows Notepad. We considered three safety properties for each SPL. For each property, we report the number of configurations on which the property has to be monitored and the time taken (duration) to derive the tracematch presence condition. We ran our tool on a Windows 7 machine with Intel Core2 Duo CPU with 2.2 GHz and 1024 MB as the maximum heap size.

Note that, although the product lines were created in-house, they were created long before this chapter's technique was conceived (GPL over 5 years ago and Notepad 2 years ago). Our tool, the examined product lines and monitors, as well as the detailed evaluation results are available for download [55].

Table 5.1: Graph Product Line (GPL) Results

Lines of code	1713
No. of features	17
No. of configurations	156
DisplayCheck	
No. of configurations	55 (35%)
Duration	69.4 sec. (1.2 min.)
SearchCheck	
No. of configurations	46 (29%)
Duration	110.2 sec. (1.8 min.)
KruskalCheck	
No. of configurations	13 (8%)
Duration	69.8 sec. (1.2 min.)

5.4.1 Case Studies

Graph Product Line (GPL)

Table 5.1 shows the results for GPL, which has 1713 LOC with 17 features and 156 configurations. The features vary algorithms and structures of the graph (e.g. directed/undirected and weighted/unweighted).

The `DisplayCheck` safety property checks if the method for displaying a vertex is called outside of the control flow of the method for displaying a graph: a behavioral API violation. Instead of monitoring all 156 configurations, our analysis reveals that only 55 configurations, or 35% of 156, need monitoring. The analysis took 1.2 minutes to complete. The tracematch presence condition that represents these configurations is available on our website [55].

`SearchCheck` checks if the search method is called without first calling the `initialize` method on a vertex, which would make the search erroneous. Our analysis shows that only 29% of the 156 configurations need monitoring. The analysis took 1.8 minutes to complete.

`KruskalCheck` checks if the method that runs the Kruskal's algorithm returns an object that was not created in the control-flow of the method, which would mean that the algorithm is not functioning correctly. In 1.2 minutes, our analysis showed that only 8% of the GPL product line needs monitoring.

Table 5.2: Notepad Results

Lines of code	2074
No. of features	25
No. of configurations	144
PersistenceCheck	
No. of configurations	72 (50%)
Duration	296.3 sec. (4.9 min.)
CopyPasteCheck	
No. of configurations	64 (44%)
Duration	259.9 sec. (4.3 min.)
UndoRedoCheck	
No. of configurations	32 (22%)
Duration	279.8 sec. (4.7 min.)

Notepad

Table 5.2 shows the results for Notepad, which has 2074 LOC with 25 features and 144 configurations. Variations arise from permuting end-user features, such as saving/opening files, printing, and user interface support (e.g. menu bar or tool bar). The analysis, for all safety properties, takes notably longer than that for GPL because Notepad uses the Java Swing framework, which heavily uses call-back methods that increase by large amounts the size of the call graph that our analysis needs to construct and to consider.

`PersistenceCheck` checks if `java.io.File*` objects are created outside of persistence-related functions, which should not happen. Our analysis completes in 4.9 minutes, reducing the configurations to monitor by 50%.

`CopyPasteCheck` checks if a paste can be performed without first performing a copy, an obvious error with the product line. The analysis completes in 4.3 minutes, reducing the configurations to monitor to 44% of the original number.

`UndoRedoCheck` checks if a redo can be performed without first performing an undo. The analysis takes 4.7 minutes and reduces the configurations to 22%.

5.4.2 Discussion

Limitations

Because this chapter's technique allows declarations to be annotated like the technique for statically pruning configurations to test (Chapter 3), it also does not handle conditional method overriding and field hiding (Section 3.7.1, last bullet). Conditional method overriding and field hiding were not encountered in the evaluation of this chapter's technique.

Cost-Benefit Analysis.

As the `Duration` row for each product-line/tracematch pair shows, our analysis introduces a small cost. Most of the duration is from the weaving that is required to determine the required shadows and from constructing the inter-procedural call-graph that we then traverse to determine the presence conditions. Usually, monitors are used in testing. Then, the one-time cost of our analysis is worth incurring if it is less than the time it takes to test-run each saved configuration with complete path coverage (complete path coverage is required to see if a monitor can be triggered). Consider `Notepad` and `PersistenceCheck` pair, for which our technique is least effective as it takes the longest time, 4.1 seconds, per saved configuration (144-72=72 configurations are saved in 296.3 seconds of analysis time). The only way our technique would not be worth employing is if one could test-run a configuration of `Notepad` with complete path coverage in less than 4.1 seconds. Executing such a test-run within this time frame is unrealistic, especially in a UI-driven application like `Notepad`.

In another scenario where a monitor is used in production, our analysis allows developers to shift runtime-overhead that would incur on deployed systems to a development-time overhead that incurs through our static analysis.

Ideal (Product Line, Tracematch) Pairs.

Our technique works best for pairs where the tracematch can only be triggered on few configurations of the product line. Ideally, a tracematch would crosscut many optional features or touch one feature that is present in very few configurations. This is evident in the running example, where the saving for `ReadPrint`, which requires two optional features, is greater than that for `HasNext`, which requires one optional feature. It is also evident in the case studies, where `KruskalCheck` and `UndoRedoCheck`, which are localized in a small number of features but requires other features due to the feature model, see better saving than their counterparts. Without any constraint, a tracematch requiring x optional features needs to be inserted on $1/(2^x)$ of the configurations (`PersistenceCheck` requires one optional feature, hence the 50% reduction). A general safety property, such as one involving library data structures and algorithms, is likely to be applicable to many configurations of a product line (if a required feature uses it, then it must be inserted in all configurations) and thus may not enable our technique to eliminate many configurations. On the other hand, a safety property crosscutting many optional features makes an ideal candidate.

5.5 Related Work

Statically Evaluating Monitors. There exist static analyses that improve the runtime performance of a monitor by reducing its instrumentation of a single program [14, 32, 15]. We will refer to these analyses as *traditional program analyses (TPA)*. There are two ways to apply such analyses to product lines. One way is inefficient, the other way imprecise. Running TPA against each instantiated program will be very inefficient because it will have to inspect every program of the product line separately. The other way is to run TPA against the product line itself. This

is possible because a product line in our technique is represented as an ordinary program. However, this second way will be imprecise. For example, suppose we apply TPA on the `ReadPrint` tracematch for our example: the tracematch may match in the case in which all features are enabled. Being oblivious to the notion of features, the analysis will therefore report that the tracematches have to be present for every program of the product line. This shows that a static analysis, to be both efficient and effective on an SPL, has to be aware of the SPL's features. Focused proerty monitoring modifies TPA to understand product lines.

Testing Product Lines. The idea of reducing configurations for product line monitoring originated from our work on product line testing [58], which finds "sandboxed" features, i.e. features that do not modify other features' control-flow or data-flow, and treats such features as don't-cares to determine configurations that are identical from the test's perspective. But the two works are different both in setting and technique. In setting, in [58], only one of the identical configurations needs to be tested. With this chapter's technique, even if a hundred configurations are identical in the way they trigger a monitor (e.g. through the same feature), all hundred configurations need to be monitored because all hundred can be used by the end-user. In testing mode, it would be possible to run just one of the hundred configurations if our technique could determine that the configurations are identical in the way they trigger the monitor. However, this would require a considerably more sophisticated analysis and is beyond the scope of this chapter's technique. In technique, the static analysis employed in [58] is not suitable for our work because a sandboxed feature can still violate safety properties and cause a monitor to trigger. Thus the two works are complementary.

Model-Checking Product Lines. Works in model-checking product lines [24, 39] are similar in intent to ours: programmers can apply model checking to a product line as a whole, instead of applying it to each program of the product line. In

the common case, these approaches yield a far smaller complexity and therefore have the potential for speeding up the model-checking process. However, these approaches do not model-check concrete product lines. Instead, they assume a given abstraction, such as a transition system, of a product line. Because our technique works on *SysGen* and Java, we need to consider issues specific to Java such as the identification of relevant events, the weaving of the runtime monitor and the static computation of points-to information. Also, model-checking answers a different question than our analysis: model-checking a product line can only report the configurations that may violate the given temporal property. Our analysis further reports a subset of instrumentation points (joinpoint shadows) that can, in combination, lead up to such a violation. As we showed in previous work [11], identifying such shadows requires more sophisticated algorithms than those that only focus on violation detection.

Safe Composition. [95, 50] collect implementation constraints in a product line that ensure that every feature combination is compilable or type-safe. Our work can be seen as a variant of safe composition, where a tracematch is treated as a feature itself that "references" its shadows in the product line and requires features that allow those shadows to be reached. However, our analysis checks a much stronger property, i.e. reachability to the shadows, than syntactic presence checked by the existing safe composition techniques. Also, collecting the referential dependencies is much more involved in our technique because it requires evaluating pointcuts that can have wildcards and control-flow constraints.

Relying on Domain Knowledge. Finally, rather than relying on static analysis, users can come up with a tracematch's presence condition themselves if they are confident about their understanding of the product line and the tracematch pair. However, this approach is highly error-prone as even a slight mistake in the presence condition can cause configurations that must be monitored to end up not being monitored. Also, our approach promotes separation of concerns by allowing

a safety property to be specified independently of the product line variability.

5.6 Summary

A product line enables the systematic development of a large number of related programs. It also introduces the challenge of analyzing families of related programs, whose cardinality can be exponential in the number of features. For safety properties that are enforced through an execution monitor, conventional wisdom tells us that every configuration must be monitored. In this chapter, we presented a static analysis that minimizes the configurations on which an execution monitor must be inserted. The analysis determines the required instrumentation points and determines the feature combinations that allow those points to be reachable. The execution monitor is inserted only on such feature combinations. Experiments show that our analysis is effective (often eliminating over one half of all possible configurations) and that it incurs a small overhead.

Chapter 6

Shared Execution for Efficiently Testing Product Lines

The contents of this chapter appeared in the 2012 International Symposium on Software Reliability Engineering (ISSRE 2012) [60]. [1]

6.1 Introduction

Chapters 3 - 5 presented ways to prune configurations in the context of checking properties using testing and runtime monitoring. Unfortunately, some properties may necessitate checking against every configuration, either by design (e.g. because each program has a unique behavior that must be checked) or due to inability of analysis techniques to prune configurations. This chapter considers the problem of reducing the cost of running tests when configurations cannot be pruned a priori. Specifically, we aim to optimize testing of product lines by reducing the cost of executing each test. Since programs in an SPL are likely to be syntactically similar

[1] Chang Hwan Peter Kim, Safraz Khurshid, and Don S. Batory. Shared Execution for Efficiently Testing Product Lines. In ISSRE (International Symposium on Software Reliability Engineering), 2012. The paper was developed jointly with my co-authors, who are my supervisor and my co-supervisor. Implementation and evaluation were done by me.

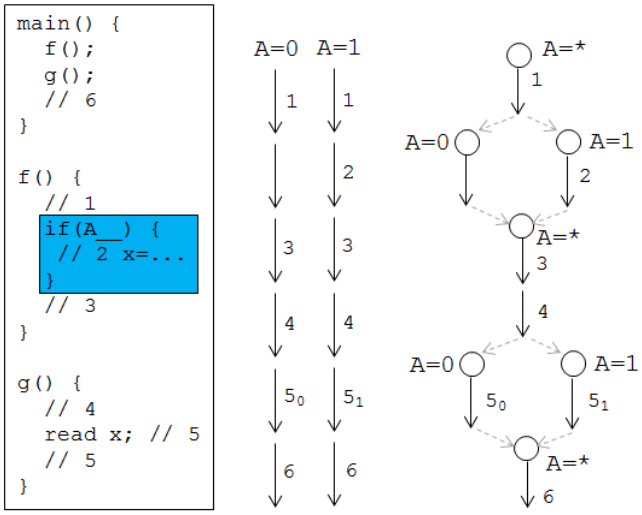
(i.e. share common code) by design, they likely have semantic (i.e. run-time) commonality as well. Namely, it is likely that many bytecode instructions executed across the test-and-program combinations will have identical behavior.

This chapter presents the idea of *shared execution*, which essentially product lines the execution itself by executing common instructions once, rather than multiple times, to eliminate redundancy and reduce execution time. Conceptually, shared execution runs an instruction that is common to multiple program executions just once by using a single call stack and memory that keeps track of each program's data. We make the following contributions:

- **Technique.** We define shared execution as a bytecode level algorithm that can be implemented on top of any virtual machine (VM).
- **Implementation.** We implement shared execution on top of *Java PathFinder (JPF)*[83], a model checker for Java that can also function as an easy-to-extend, off-the-shelf VM. We use only the VM portion of JPF.
- **Evaluation.** We show, using non-trivial subjects used in prior publications of other research groups, that shared execution, despite its overhead, can run a product line test case up to 50% faster than the conventional way of running the test case for each configuration from start to finish.

6.2 Shared Execution: Basic Technique

Figure 6.1(a) shows the skeleton of a typical SPL that has six fragments of code labeled as comments, whose numbers indicate the order in which the fragments are executed. Note that a program with a feature included can behave differently from a program without that feature. For example, `x` could have two different values in executions of fragment 5 and subsequently two different program behaviors, depending on whether feature `A` is selected or not. More generally, in a product line



(a) Sample SPL (b) Conventional execution (c) Shared execution

Figure 6.1: Shared Execution on Sample SPL

with N optional features, fragment 5 could yield (2^N) different values of x and trigger as many different program behaviors.

Conventionally, different program behaviors are produced by running each configuration from start to finish. Figure 6.1(b) shows this process for the example. Note the similarities between the execution traces produced: the only differences are that the A=1 configuration executes fragment 2 while A=0 does not and that fragment 5 behaves differently due to the configurations having different values of x. Note that the majority of computations are repeated across both traces. Although a computation can only be repeated once in this example, it can be repeated in general up to 2^N-1 times since up to 2^N traces can be produced.

Instead of running each configuration separately, our idea is to execute all the configurations together, executing a computation just once for the configuration(s) for which it is shared. Figure 6.1(c) shows the resulting trace, which is essentially a superimposition of all the traces produced in conventional execution in Figure 6.1(b).

Note how fragments 1, 3, 4 and 6 are executed once, rather than twice. Fragments in branches, such as the empty fragment and fragment 2, are not shared by all the configurations and each branch is executed one after another before shared execution resumes.

We now present the basic ideas behind our approach.

6.2.1 Bookkeeping

Since each variable can have as many different values as there are configurations, memory M must be able to map a variable v *and* a configuration c to a value o. Namely, $M : V \times C \rightarrow O$. *Variable* refers to any data storage that can be accessed by a programmer-defined symbol, i.e. fields (including array elements) and local variables. Conceptually, memory can be thought of as an array of length 2^N with one array element holding the memory of one product line configuration. Section 6.4 will present a more efficient representation.

We define *shared execution* as executing instructions for a set of configurations using a common *call stack*, which stores information including active method calls, program counter of the current method call, and instruction operands to represent a point in program execution or *execution point*. Note that since we are treating local variables as part of memory, a stack frame of the call stack only includes the stack operands and associated attributes (e.g. which operands are references). Before (or after) each instruction execution, there exists a *state* `S` with configuration set $S_{configs}$ and the call stack $S_{callstack}$ in use. At the beginning of shared execution, $S_{configs}$ has all configurations of the product line and $S_{callstack}$ just has the main function's stack frame whose PC is set to the first instruction.

6.2.2 Splitting

Since the idea of shared execution is to use one call stack for multiple configurations, instructions execute as they would for conventional programs except that loads and stores must now access memory from a configuration set. Storing value o to a variable v in state S simply means that $\forall c \in S_{configs}, M(v, c) = o$. Loading is a bit trickier. If $M(v, c)$ returns the same o for every c in set $S_{configs}$, shared execution continues with the read value o. But if $M(v, c)$ returns multiple values, shared execution cannot continue since we can only push one unique value on the call stack. So shared execution splits such that there will be as many call stacks as there are unique values. Namely, the current state S's children states, $S_{children}$, are created such that:

- The union of every child state's configuration set equals the parent configuration set, $S_{configs}$.
- Each configuration in a child state's configuration set holds the same value of v.
- Each child state's call stack is set to a clone of $S_{callstack}$.

We say that S is *split* into children states with respect to v. Each child state, which is ready to have a unique value of v pushed onto its call stack, is set as the current state and executed from where the splitting occurred. As execution proceeds, if a load of another variable y yields multiple values, the executing state will be split in a similar way.

In Figure 6.1(c), right before `A__` is read, S has $S_{configs}$ equal to $A = *$ (the wildcard, which represents all possible values, is used to represent multiple configurations concisely. So $A = *$ represents $\{A = 0, A = 1\}$). When the read occurs, S is split (dashed lines outwards) into $S_{A\_\_=0}$ and $S_{A\_\_=1}$ since `A__` has different values:

0 in A=0 and 1 in A=1.[2] The first child state (circle labeled A=0) runs and then execution backtracks to load the call stack of the second child state (circle labeled A=1) for it to run.

6.2.3 Merging

We could execute each child state until the end of the program, but doing so, we would miss out on opportunities to share execution after splitting. For optimal shared execution, we should wait until all children states come to a common execution point, i.e. where their call stacks are equivalent, and then resume shared execution with the same call stack.

There are two issues to consider for finding a common execution point. First, the children states could have considerably different paths of execution. Second, finding a common execution point close to the splitting point allows sharing to resume earlier, but it may require more checks, each of which comes with the cost of comparing potentially up to 2^N call stacks, where N is the number of features.

Conservative Merging

A reasonable compromise that addresses both issues, which we call *conservative merging*, is to wait until each child state's execution reaches the end of the method (just before a return statement) of where splitting occurred since each child is guaranteed to reach this execution point.[3]

Although a return value is in practice written and read off the call stack, for uniformity in explanation, we treat a return value as a variable (written and read off of memory) throughout the chapter. This means that even with different return

[2]Thus, a feature variable is treated like an ordinary variable in shared execution.

[3] A child state's execution may not reach the end of the method due to abnormal program execution (e.g. exception or system exit), in which case the child state can be simply executed until the end of the program and shared execution resumes with the remaining children. However, our implementation currently does not handle abnormal program execution.

values, the children states would have the identical call stack at the function return, allowing shared execution to resume. Then splitting would occur when the return value is read *after* the function returns.

For example, in Figure 6.1(a), when the state executing fragment 1 splits, A=0 and A=1 children states would execute until the end of fragment 3 and then merge, allowing fragment 4 to be shared. Then fragment 5 would cause the state to split again and merge at the end of fragment 5, allowing fragment 6 to be shared. But note that the proposed solution is not as optimal as Figure 6.1(c) in that the former does not allow fragment 3, or any instruction executed between the splitting point and the end of the method, to be shared.

Predictive Merging

Predictive merging improves on conservative merging by using what we call a *merge point*. When splitting occurs, we determine an optimistic merge point, i.e. an execution point *before* the end of the method that each child state is likely, but not guaranteed, to reach. We then execute each child state until it reaches this optimistic merge point or the pessimistic merge point, i.e. the end of the method where splitting occurred. If the children have all stopped at the same execution point, i.e. all at optimistic or all at pessimistic merge point, we resume shared execution with the parent state. Otherwise, we execute each child stopped at the optimistic merge point until the pessimistic, but guaranteed to be common, merge point and then resume shared execution.

In Section 6.4.2, we discuss in detail how optimistic merge points are determined. For now, it suffices to know that when splitting is due to a read of a boolean variable that is followed by an if-statement, such as `if(A__)`, the optimistic merge point is determined to be the end of the if-statement since programs are typically written such that both `true` and `false` branches will end up at this point. If the

`true` branch executes a control-flow breaking instruction such as a `return`, shared execution will resume at the pessimistic merge point.[4]

For example, in Figure 6.1(a), when the state executing fragment 1 splits, the optimistic merge point is set to the end of the `if(A__)` block. The children states would execute until the *beginning* of fragment 3 and then merge, allowing fragment 3 and 4 to be shared as Figure 6.1(c) shows. But if there is a `return` within the if-condition, merging would not be possible the first time around since `A=1` state would end up at the end of the method while `A=0` ends up at the beginning of fragment 3. Then state `A=0` would be executed from the beginning of fragment 3 until the end of the method before merging with state `A=1`.

6.2.4 Putting Ideas Together

The ideas of bookkeeping, splitting, and merging can be summarized in an algorithm that intercepts every bytecode instruction, as shown in Figure 6.2. As shared execution works with call stacks, the algorithm must be written in terms of bytecode instructions.

Initialization.

S, which represents the state before the currently executing instruction, is initialized so that $S_{configs}$ is set to all product line configurations and $S_{callstack}$ is identical to the VM call stack (line 2). Memory M requires all configurations of the product line for initialization (line 3) to explicitly map feature variables to values across configurations since feature variables can only be read from but not written to. For example, for feature variable `A__`, configuration `A=1` maps to the value 1 and `A=0` to the value 0.

[4] Any previously stopped executions at the optimistic point are run to the pessimistic merge point.

```
1 class SharedExecution extends VMListener {
2   State s = new State(allConfigs, VM.getCallstack());
3   SPLMemory m = new SPLMemory(allConfigs);
4
5   void beforeInstruction(Instruction insn) {
6     if(s.isAtMergePoint())
7       tryMerge();
8     else if(isLoad(insn)) {
9       s.children = split(s, getVariable());
10      if(s.children != null) {
11        s.mergePoints = getMergePoints();
12        loadState(s.children.get(0));
13      }
14      else
15        VM.load(getVariable(), m.values
16                (getVariable(), s.configs).first());
17    }
18    else if(isStore(insn))
19      m.set(getVariable(), s.configs, VM.getTopValue());
20  }
21
22  void tryMerge() {
23    if(s.isLastRemainingChild()) {
24      if(atSameExecPoint(s.parent.children) {
25        s.parent.callstack = s.parent.children.
26                             get(0).callstack;
27        loadState(s.parent);
28      }
29      else {
30        s.parent.setMergePoint(RETURN_MERGE_POINT);
31        loadState(s.parent.nextRemainingChild());
32      }
33    }
34    else
35      loadState(s.parent.nextRemainingChild());
36  }
37
38  void loadState(State t) {
39    s = t;
40    VM.changeCallstack(s.callStack);
41  }
42 }
```

Figure 6.2: Shared Execution Algorithm

Loads and Stores.

If a load returns multiple values for a variable, the state is split into children such that each child has configurations that map to the same variable value and a clone of the executing call stack. The parent state keeps track of both optimistic and pessimistic merge points where all of its children will stop their execution (line 11). Then the first child state is set as the current state and the executing (VM's) call stack is changed to the current state's call stack (lines 12, 39 and 40). On the other hand, if the load returns one value of the variable, the value is simply pushed on the stack as it would be in conventional execution (line 15). Note that the load of a feature variable, whose values across configurations have been explicitly stored during initialization, will be what first triggers splitting.

A store just sets the variable value for each configuration of the current state (lines 18-19).

Merge.

Following a split, we check if the current state is at a merge point, i.e. its call stack is at a return instruction (or an earlier instruction for an optimistic merge point) and the stack's depth is equal to that of where splitting occurred (line 6). [5] If the state is at a merge point, we backtrack execution to the next child state (line 35), allow it to execute until a merge point, and repeat the process until the last child comes to a merge point (line 23). Then, if the children's call stacks are not identical, children are executed up to the pessimistic merge point, i.e. the end of the method (lines 24, 30, 31). At this point, the parent state's call stack takes a child's call stack and is set as the current state (lines 26 and 27), completing the merge.[6]

[5]Note that we use the stack depth because comparing method signatures alone will not suffice due to recursion.

[6] Note that the merge attempt is the first step in `beforeInstruction()` because processing a load instruction before merging can end up splitting the current state without taking its siblings into account, which can reduce the configuration set corresponding to a unique value of a variable,

6.3 Example

We demonstrate shared execution on the example SPL in Figure 6.3(a) (line numbers in this section refer to this figure). Features `A___` and `B___` are simply boolean variables that are assigned concrete values for a particular program. Although a product line can be written in any way with these feature variables, there typically exists code for a feature and the code is placed in the true branch of the corresponding boolean variable, as shown. An SPL test case is simply an execution of the main method with all variables, except the feature variables, assigned concrete values. The test case must be run on all 4 combinations of the feature variables.

6.3.1 Splitting and Merging

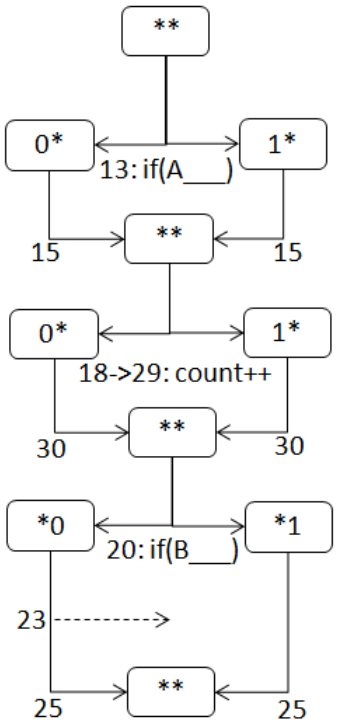
We demonstrate shared execution on the example using Figure 6.3(b), which shows how states split and merge throughout execution. Splitting first occurs in line 13 because `A_` has multiple values, i.e. 1 (`true`) for the configurations `AB`=1* and 0 (`false`) for the configurations `AB`=0*. Therefore, as the top of Figure 6.3(b) shows, the state with configurations `AB`=** is split into two children. The optimistic merge point is set to the end of the if-statement (line 15). The 0* state does nothing ends up on line 15. Then the 1* state appends 2 and ends up on line 15. The two states merge because their call stacks are equivalent on line 15. Figure 6.4 shows the memory snapshot at line 17. Note that line 16 computation is shared by all configurations.

Appending 3 causes a split because line 18 leads to line 29, where reading `count` for increment yields multiple values (1 for configurations 0* and 2 for configurations 1*). The split states merge in line 30. Then, when line 20 causes a split due to reading `B_`, optimistic merge point is set to line 23, the end of the if condition. Although the state *0 hits this merge point, the state *1 returns and

which in turn reduces sharing.

```
1  class Example {
2    static boolean A___;
3    static boolean B___;
4
5    static char[] array =
6              new char[5];
7    static int count = 0;
8    static int x = 0;
9
10   static void main(){
11     append('1');
12
13     if(A___)
14       append('2');
15
16     x++;
17
18     append('3');
19
20     if(B___) {
21       return;
22     }
23
24     x++;
25   }
26
27   static void append
28               (char c){
29     array[count++] = c;
30   }
31 }
```

(a) Product Line Test



(b) Split Merge Diagram

Figure 6.3: Example Product Line

Config (A_B_)	String	X
00	"1"	1
01	"1"	1
10	"12"	1
11	"12"	1

Memory snapshot at line 17

Config (A_B_)	String	X
00	"13"	2
01	"13"	1
10	"123"	2
11	"123"	1

Memory snapshot at line 25

Figure 6.4: Memory snapshots

misses it, causing merge at the conservative merge point (line 25). Figure 6.4 shows the memory snapshot at the end of the test case (line 25). Note that although each configuration produces a distinct row or test case output, each computation was shared by at least 2 configurations.

6.4 Shared Execution: Optimizations

For shared execution to be practical, we need an efficient memory representation and an optimistic merging strategy. Also, garbage collection needs to be modified to support shared execution. In this section, we discuss these optimization related issues.

6.4.1 Memory

As mentioned in Section 6.2.1, to access a variable value, memory must now be addressed by both the variable *and* a configuration. The easiest way to structure memory would be to allocate a value slot for each configuration and variable pair. However, this would be wasteful for the following reasons.

Ownership.

A variable can only exist in the configurations that created it. Suppose that a method `bar()` is called in a state with configurations where $A = 1$, as shown in Figure 6.5. The local variables allocated for that method call, such as `b`, cannot exist in configurations where the feature is absent and therefore do not even need to have storage for these configurations. Similarly, an object and its fields cannot exist in a configuration that is not a part of the state that created the object. Thus, it would be impossible to access object 2 and its fields in a configuration with $A = false$, while object 1 and its fields can be accessed in any configuration (since the object is owned by all configurations). We say that a variable's *owner* is the configuration

```
1  class Program {
2    Obj x = new Obj(); // 1
3
4    void foo(){
5      if(A) {
6        bar();
7        x = new Obj(); // 2
8      }
9      m(x);
10     ...
11   }
12
13   voix bar(){
14     int b = 5;
15     ...
16   }
17 }
```

Figure 6.5: Memory Example

set that created it either through a method call (for local variables) or through an object creation (for instance fields). For a variable, only as many value slots as there are owner configurations are allocated.

Small number of unique values.

A variable owned by M configurations could have M unique values, but chances are that there will be far fewer number of values. The reason is as follows: a variable is owned by configurations with the creating feature present. For the variable to have as many values as there are owner configurations, the creating feature must interact with *all* the other features, which is possible but less likely than interacting with just some of the other features. Moreover, there will be variables used only by the creating feature, i.e. by the owner configurations, to achieve computations relevant only to it.

Dual Memory.

We exploit the notion of ownership and small number of unique values in the following way. Since a variable having one value across its owner configurations is no

different than a variable in conventional execution, we partition memory into two storages: 1) *conventional storage*, i.e. memory created by VM that maps a variable to a value ($M_1 : V \rightarrow O$) and 2) *multivalued storage*, i.e. memory created by our technique that maps a variable *and* a configuration to a value ($M_2 : V \times C \rightarrow O$) and is used to store variables with multiple values. This representation, called *dual memory*, exposes how loads and stores work in Figure 6.2:

- **Loads.** If a variable exists in multivalued storage, a load is performed for each configuration of the current state and splitting occurs if there is more than one unique value. If the variable doesn't exist in multivalued storage, then the variable is loaded from the conventional storage.

- **Stores.** A write could occur with the current state's configurations $S_{configs}$ spanning either 1) owner configurations or 2) a subset of the owner configurations. In scenario 1), the variable will have the same value across owner configurations after the write. Therefore, the write is performed just once for the conventional storage and the variable is removed from the multivalued storage since it no longer has multiple values. In scenario 2), $S_{configs}$ will take on the new value in multivalued storage. If the variable is managed by the conventional storage, its value is transferred to multivalued storage (to the complement of $S_{configs}$ with respect to the owner configurations) and the variable now becomes managed by multivalued storage.

For example, in line 16 of Figure 6.3, `x` is read from and written to conventional storage, but in line 24, `x` is transferred to multivalued storage. In line 11, `count` and `array[1]` are in conventional storage, but by line 15, they are in multivalued storage. `array`, `array[0]`, `array[3]` and `array[4]` remain in conventional storage throughout the test case.

6.4.2 Optimistic Merging

As mentioned in Section 6.2.3, when splitting occurs, we want to merge before the end of the method to potentially share the remaining instructions of the method. Although we could execute each child one instruction at a time and check if their call stacks are equivalent, this would perform checks too often. Thus, we use merge points, which are candidate execution points for merging, to minimize the number of call stack comparisons and maximize the likelihood of a comparison succeeding. Note that because children will merge within the same function of where splitting occurred (modulo abnormal program execution discussed earlier), only the top frames of the children's call stacks need to be checked for equivalence, meaning the cost of comparison boils down largely to the number of children, i.e. the number of unique values of the variable whose load caused the split.

If the variable load causing the split is followed by a conditional instruction, the merge point is set to the instruction that the last instruction of the `true` block of statements jumps to. This is because we assume that the conditional instruction is part of an `if-else` statement and both `true` and `false` branches are likely to end up there. If the variable load is not followed by a conditional instruction, an instruction following a store instruction or a method invocation is treated as a merge point. This is because a variable is likely to be loaded for computing a value to write to another variable or for invoking a method with the value as one of its arguments. Also, if the children all come to the same instruction following the same store instruction or the same method invocation, call stack comparison is likely to succeed since the different values have already been popped off the call stack. So these merge points are ideal places to compare call stacks. If the comparison fails, the children are executed from where they are until the end of the method is reached (i.e. the conservative merge point), where the call stacks will be equivalent.

For example, in Figure 6.5, loading `x` as an argument for method invocation

`m()` will cause a split since `x` will point to object 1 in configurations without `A` and to object 2 in configurations with `A`. The end of the method invocation, i.e. right before line 10, will be set as the optimistic merge point.

6.4.3 Garbage Collection

As shared execution requires more memory than running each configuration separately, garbage collection is especially important for optimal performance. Unfortunately, we cannot use the VM's garbage collection as-is since it is not aware of product lines. For example, it will collect objects that are still alive for some configurations. Another reason why we need modify the conventional garbage collector (GC) is to clean up our own multivalued storage.

In Figure 6.5, `foo()` is executed up to line 9 once with $A = true$ and once with $A = false$. At this point, as far as the conventional GC can tell, `x` can point to either object 1 or object 2 but not both, meaning one of the objects will be garbage collected, which is clearly erroneous since `m()` must be invoked with each object as its argument.

Shared execution GC is a simple modification of conventional GC such that objects are marked starting from the call stacks of leaf states of the hierarchy of states, rather than just from the executing call stack. Note that only the leaf states have call stacks and the non-leaf states exist for splitting/merging purposes. Also, the marking phase is changed such that if a variable has multiple values across its owner configurations, multiple references will be marked as being alive. Finally, when an object is garbage collected, shared execution GC removes variables corresponding to its fields from the multivalued memory. Note that nothing needs to be done with the object itself as an object is not a variable that can have different values across configurations.

6.5 Evaluation

Our technique can be implemented on top of any virtual machine, such as Jikes RVM [48] or Java PathFinder (JPF) [83]. Although JPF is typically used as a model checker rather than as a VM, we chose it as our platform because of its extensibility and our familiarity with it. As a VM, JPF is considerably slower than an ordinary VM, but since we are running both shared execution and conventional approach using JPF, using it should not affect our results. We evaluated shared execution on 3 subjects: Graph Product Line (GPL) (originally appeared in [72]), JTopas [47], and XStream [71], that have also been used for evaluating testing techniques by other groups (GPL by [20], JTopas by [21] and XStream by [30][88]). Shared execution implementation, subjects, and results can be downloaded from our website [56].

6.5.1 Graph Product Line (GPL)

GPL encodes programs that implement different graph algorithms. The product line, with 1713 LOC, 14 features and 146 configurations, was developed in our research group, but long before this chapter's technique was developed [72]. Note that there are only 146 configurations despite 14 features due to constraints. The features vary algorithms and structures of the graph (e.g. directed/undirected and weighted/unweighted). Due to lack of tests for this product line, we generated graphs and ran the main method on each graph against all the configurations. Although many algorithms are independent, some features do interact to produce different outcomes. For example, `DIRECTED` will change the outcome of `CYCLEDETECTION` and combinations of `BFS` (Breadth First Search) and `DIRECTED` may change the outcome of `NUMBERING` (how nodes are numbered).

Table 6.1 shows the results for GPL. As the headers show, three types of graphs were generated: 1) sequence of neural networks (each with one input node

Table 6.1: GPL Results

	Conventional	Shared Execution	Factor/Saving (%)
Network 1: 45 nodes, 80 edges			
No. of test case insns	76702636	4024943	19:1
Duration (sec.)	25	19	24
Network 2: 221 nodes, 400 edges			
No. of test case insns	2951629460	146355025	20:1
Duration (sec.)	503	277	45
Tree 1: 85 nodes, 84 edges			
No. of test case insns	234426320	11907505	20:1
Duration (sec.)	51	36	29
Tree 2: 341 nodes, 340 edges			
No. of test case insns	9318420952	448314944	21:1
Duration (sec.)	1619	756	53
Random 1: 101 nodes, 374 edges			
No. of test case insns	450951320	23119790	20:1
Duration (sec.)	101	66	35
Random 2: 101 nodes, 381 edges			
No. of test case insns	431427468	22113491	20:1
Duration (sec.)	95	61	36
Random 3: 101 nodes, 372 edges			
No. of test case insns	449992982	23077201	19:1
Duration (sec.)	98	65	34
Random 4: 101 nodes, 362 edges			
No. of test case insns	429500846	22067178	19:1
Duration (sec.)	94	61	35
Random 5: 101 nodes, 336 edges			
No. of test case insns	431307370	22318788	19:1
Duration (sec.)	93	62	33

and one output node), 2) tree with a fixed degree, and 3) a random graph with a fixed number of nodes and an average degree generated using an off-the-shelf random graph generator [38]. *No. of test case insns* only includes the bytecode instructions executed by the test case, and does not include instructions from the shared execution implementation. The table shows that shared execution executes about 1/20th bytecode instructions of the test case and saves between 24% and 53% of execution time over the conventional approach of running the test case against each configuration from start to finish. Note that the time saving is considerably larger for the larger network and tree. Time saving stays consistent across the random graphs, whose numbers of nodes and edges are nearly the same, suggesting that the results are probably representative of other graphs with similar numbers of nodes and edges.

6.5.2 JTopas

JTopas [47] is an open source Java program for parsing text that has 2031 lines of code. We converted this conventional program into an SPL simply by converting the following boolean configuration flags into boolean feature variables that our shared execution tool can recognize: `LINECOMMENTS`, `BLOCKCOMMENTS`, `COUNTLINES`, `IMAGEPARTS`, and `TOKENPOSONLY`. If `COUNTLINES` is `true`, each token will have line and column information. If `IMAGEPARTS` gives each token's string more structure, such as breaking it into lines. `TOKENPOSONLY` represents a token by its position in the original text, rather than string. `LINECOMMENTS` and `BLOCKCOMMENTS`, which return a single token representing a line comment or a block comment respectively if the feature is on and skips the corresponding characters if the feature is off, change the result of tokenizing an input embedded with comments significantly. These 5 features allow 32 configurations.

We simplified an existing test called `TestLargeSource`, which tokenizes a Java class with some methods, to run against the configurations. We created 9 test cases out of this test to not only test inputs of different sizes, but also inputs that are expected to result in different amount of instruction sharing. Since we expect tokenization across configurations to be more different the more comments there are, we used test cases with varying number of comments to see if shared execution results are consistent with this expectation. The Java code input for `Many` test cases is shipped with JTopas. `Some` and `Without` test cases simply remove comments from this Java code input for their own input. The test case number `N` (e.g. `Many N`) means the code input is tokenized `N` times. Table 6.2 shows that instruction sharing, and consequently time saving, indeed does increase the fewer comments there are. The table also shows that shared execution's overhead outweighs the benefit of executing 5.4 times less test code, as shared execution actually takes longer than conventional execution in these cases.

Table 6.2: JTopas Results

	Conventional	Shared Execution	Factor/Saving (%)
Many comments 1			
No. of test case insns	38195022	14988848	2.5:1
Duration (sec.)	16	25	-56
Many comments 2			
No. of test case insns	75706878	30919876	2.4: 1
Duration (sec.)	27	49	-81
Many comments 3			
No. of test case insns	113319726	46912432	2.4:1
Duration (sec.)	39	60	-53
Some comments 1			
No. of test case insns	34283622	3967675	8.6:1
Duration (sec.)	15	14	6.7
Some comments 2			
No. of test case insns	67826606	12661565	5.4:1
Duration (sec.)	26	28	-7.7
Some comments 3			
No. of test case insns	101424102	21416331	4.7:1
Duration (sec.)	33	39	-18
No comments 1			
No. of test case insns	33245790	2421775	14:1
Duration (sec.)	14	14	0
No comments 2			
No. of test case insns	65735326	4824448	14:1
Duration (sec.)	24	19	20
No comments 3			
No. of test case insns	98281742	7234081	14:1
Duration (sec.)	34	26	24

6.5.3 XStream

XStream [71] is an open source program for serializing objects to XML and back again that has 14,480 LOC. Like we did with JTopas, we converted this conventional program into an SPL by simply converting the following boolean configuration flags into feature variables. `TREESTRUCTURE` inlines references such that the produced XML is a hierarchy, not a graph. `CLASSALIAS` and `FIELDALIAS` allow class and field names to be aliased. `OMITFIELD` omits specified fields when producing XML. `IMPLICITARRAY` omits specified container objects to reduce XML clutter. `ATTRIBUTE` places specified fields in the tag of the owner object for readability. `BOOLEANCONVERTER` allows a boolean field to be represented with custom string representation for 'true' and 'false'. With these 7 features, XStream SPL encodes 128 configurations.

Like with JTopas, we developed 3 sets of 3 test cases, with different sets testing different levels of instruction sharing and cases within a set testing different input sizes. It was easier to write our own classes and objects to serialize than to reuse existing ones. The test is structured as follows: a contiguous block of `Variable` objects are sandwiched between contiguous blocks of `Common` objects in a list that is serialized. XML of each `Variable` object is different for each configuration because each feature influences it. On the other hand, XML of each `Common` object is identical for each configuration, meaning that serialization between configurations should be shared for these objects. As Table 6.3 shows, the first 3 test cases have 0 `Common` object and therefore not much instruction sharing (a bit higher than 5:1), but shared execution is still 11% - 15% faster. For the next 3 test cases, 60% of the objects are `Common`, which increases sharing to 7:1 and higher and the speedup to at least 24%. Then with 0 `Variable` objects, sharing increases to around 10:1 and speedup to as high as 35%.

Table 6.3: XStream Results

	Conventional	Shared Execution	Factor/Saving (%)
0 Common, 10 Variable			
No. of test case insns	99814860	16308440	6.1:1
Duration (sec.)	32	27	15
0 Common, 20 Variable			
No. of test case insns	174168126	31258531	5.6:1
Duration (sec.)	49	43	12
0 Common, 30 Variable			
No. of test case insns	248489238	46172217	5.4:1
Duration (sec.)	67	59	11
6 Common, 4 Variable			
No. of test case insns	95104366	11849715	8:1
Duration (sec.)	36	22	39
12 Common, 8 Variable			
No. of test case insns	163223088	22415718	7.3:1
Duration (sec.)	46	35	24
18 Common, 12 Variable			
No. of test case insns	231362556	33049336	7.0:1
Duration (sec.)	63	48	24
10 Common, 0 Variable			
No. of test case insns	89531110	8351763	11:1
Duration (sec.)	28	19	32
20 Common, 0 Variable			
No. of test case insns	153652858	15970598	9.6:1
Duration (sec.)	43	30	30
30 Common, 0 Variable			
No. of test case insns	217951258	23623858	9.2:1
Duration (sec.)	57	37	35

6.6 Discussion

6.6.1 Threats to Validity

The main threat is to external validity: the timing results cannot be generalized to all SPLs because our case studies may not be representative of all SPLs and tests. To reduce this threat, we used multiple subjects that have also been used by other groups for evaluating testing techniques. Also, we designed tests with varying levels of expected amount of sharing, which represent scenarios ranging from worst-case to best-case.

6.6.2 Correctness

Shared execution optimizes but is otherwise semantically equivalent to conventional execution. To test that our tool implements shared execution correctly, we check that every shared execution's output is identical to conventional execution's output. For the 3 case studies, we produced a console output for each configuration, each of which was identical to the corresponding output of conventional execution.

6.6.3 Native Code

Java VMs call native methods, which are blackbox to the VM. To handle shared execution, native code execution can be changed to understand it or it can be treated as an atomic operation. For our implementation, we chose the latter and we ensure that splitting and merging occurs before and after entering a native method, meaning that the native method never reads a variable with multiple values across configurations. The simplest, safest but also the most expensive way to achieve this would be to split on each configuration of the current state when a native method is invoked. Instead, we manually analyzed frequently executed native methods to determine under which circumstances we need to split. For example, before entering

`System.arraycopy` (native in JPF), our tool checks whether the source array arguments are multivalued and split if they are. Because there are not many native methods, manual analysis was not a significant issue.

6.6.4 Hybrid Approaches

Hybrid approaches would exploit shared execution but also allow conventional execution to minimize overhead. For example, instructions could be shared only up to the first variable load that causes a split (i.e. a feature variable load), at which point each configuration being tested would be run to completion. This would almost guarantee a time saving, although it may not be much if the splitting occurs early in the test. Another possibility is to switch to conventional execution after a tester specified limit, such as time, memory size or number of instructions executed. A more elaborate possibility is to split the configuration set to test at the very beginning of the test into N configuration sets and run shared execution on each configuration set. The split would be done in a way to maximize shared execution's effectiveness for each configuration set and could be performed manually using domain knowledge or automatically using static analysis.

6.6.5 Other Benefits of Sharing Execution

Although the main benefit of shared execution is that it saves execution time, there are other benefits. For example, it reduces the size of the execution trace for the entire product line significantly, which can make it easier to store and analyze. Also, it can be used to analyze behavioral properties related to product lines. For instance, suppose that a tester knows that a block of code must be shared by all configurations. Shared execution can be used to determine whether it is or not.

6.7 Related Work

6.7.1 Testing Conventional Programs

Clustered test execution [75][52] combines test cases with common initial segments into a hierarchical structure such that tests are executed together until they differ, at which point execution splits, much like shared execution. Unlike shared execution, these techniques run until completion rather than merging and thus is not able to share instructions after splitting. Also, these techniques require comparing test cases to find commonality, whereas the commonality in SPLs exists already, provided naturally by the way a product line is structured.

`Rozzle` [63] is a JavaScript multiexecution VM for exposing environment-specific malware that, like our work, explores multiple execution paths within a single execution path. However, our purpose is to optimize a given set of executions by exploiting similarity between them, while their purpose is to increase the ability to find bugs. The different purposes explain the numerous technical differences. Memory turns into an array of concrete values in our work, while memory is changed to store symbolic values in theirs. Operations against symbolic, not concrete, values is what saves execution in their work, while our work saves it by using a single call stack for multiple executions. Also, their work allows infeasible paths to be executed and symbolic values to be concretized to infeasible values as it may increase the ability to find bugs, whereas shared execution is strictly an optimization. Since each SPL configuration is expected to produce a unique concrete value in our setting, shared execution seems to be more appropriate than a symbolic approach like theirs because the latter would just delay the bytecode redundancy problem until concretization of symbolic values.

6.7.2 Testing Product Lines

Sampling relies on domain knowledge to select combinations of features to test [27][26][74]. It is practical but may miss problematic interactions, which our work does not. Model checking product lines [24] [23], which builds on standard model checking techniques, is different from shared execution in that they are not able to share instructions after splitting.

In [58], we statically determined features irrelevant to a test (e.g. unreachable or does impact outcome) to reduce combinatorics. In [59], we inserted monitors only for feature combinations that can trigger them by constructing path conditions over the features using static analysis. Shared execution, a dynamic analysis, complements these works by providing a practical reduction in a setting where the test case must be run on every feature combination because most of the features are relevant and interact, as far as can be determined statically.

Stricker et al. [92] compute dataflow dependencies in a product line and determine dependencies that must be tested for a given configuration. Due to similarities between configurations, chances are that dependencies between configurations will overlap, which their technique ensures will not be tested redundantly. Although both our work and theirs aim to eliminate redundancy in testing configurations, the works are different in that [92] eliminates redundant test cases for dataflow testing, while we eliminate redundant instructions when running a test case.

[69] proposes reusing execution traces to reduce product line testing. When running a test case for a given configuration, every use of a module (a programming construct with an interface) is recorded. If another configuration uses the same module in a way that is identical to the recorded trace, then the result is retrieved from the recorded trace without having to recompute it. Shared execution can achieve greater reuse and save more time than reusing execution traces by working at the finer grained, instruction level. Also, the two works are complementary since

a technique may incorporate both shared execution and trace reuse.

6.8 Summary

We presented shared execution, a technique for efficiently testing product lines that allows each variable to have as many values as there are configurations but carries out execution using a single call stack. Because there are likely to be features that do not interact with all the other features, a variable will have a number of values that is considerably smaller than the number of configurations being tested, meaning that many instructions will be shared across multiple configurations. This notion, coupled with the idea of ownership for low memory overhead, is what makes shared execution generally faster than running each configuration from start to finish. And while shared execution's performance has room for improvement, which may be filled, for example, by hybrid approaches, the benefit of shared execution may not be limited to time saving. Shared execution effectively product lines execution, which may allow us to systematically exploit commonalities and variabilities to tackle problems that remain unsolved.

Chapter 7

Deferred Execution for Efficiently Testing Product Lines

7.1 Introduction

The previous chapter presented the idea of shared execution, which executes a set of configurations together using a single call stack, executing common instructions once, splitting execution when different values need to be loaded on to the call stack, and merging executions when their call stacks are identical. Despite shared execution's overhead, the technique's evaluation showed that it can be effective against non-trivial configurable systems.

A basic assumption of shared execution is that the variable value placed on the stack frame will impact the test outcome and this is why shared execution splits as soon as the value read differs from configuration to configuration. However, even if a variable can have different values for different configurations, the test may never need all those different values to produce the final outcomes. In this chapter, we

present the idea of *deferred execution*, which improves shared execution by increasing the amount of shared computations by splitting execution not when different values are read, but when execution must branch subject to those values. This chapter makes the following contributions:

- **Technique.** Deferred execution is presented as a bytecode level algorithm that can be implemented on top of any standard virtual machine (VM). It also introduces the idea of *multiaddresses*, which allows different addresses to be treated as a single address to enable sharing of computations, which was not supported by shared execution.
- **Implementation.** Deferred execution is implemented using *Java PathFinder (JPF)* [83], a model checker for Java that can also function as an easy-to-extend, off-the-shelf VM. We use only the VM portion of JPF.
- **Evaluation.** We show that deferred execution, despite its overhead, can be useful in configurable systems that use design patterns and in certain cases, can even eliminate combinatorial explosion in running time that shared execution and conventional execution suffer from.

7.2 Multivalued Stack Operands

Deferred execution, like shared execution, intercepts bytecode instructions to optimize test execution. Deferred execution's changes to shared execution can be divided into two categories: allowing a stack operand to carry multiple values (Section 7.2) and enabling reads and writes against multiaddresses (Section 7.3). We discuss the former in this section and the latter in the next section.

A *memory address* is an identifier for a *storage* in memory, which stores a *multivalue*. A multivalue is a map from values to sets of configurations that has

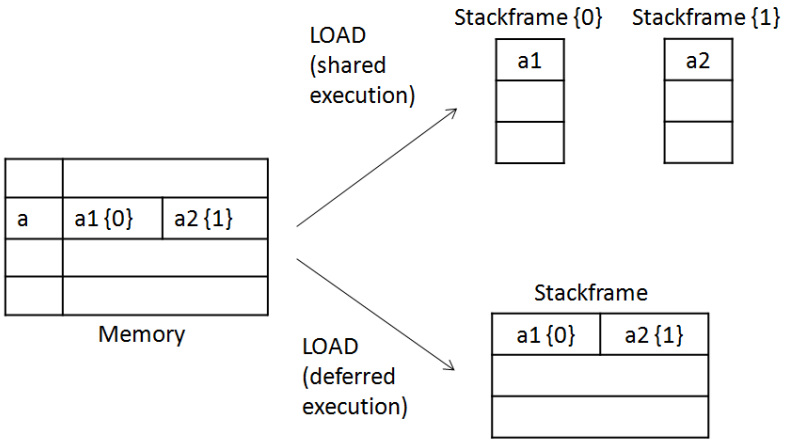


Figure 7.1: Multivalued Stack Operands Example

the form $\{v_1\{C_1\}, ..., v_N\{C_N\}\}$, where a value v_i corresponds to a set of configurations C_i, $i = 1...N$, and $C_i \subset allConfigs$, where $allConfigs$ represents all the configurations of the product line. A memory address can be qualified by a set of configurations to refer to the portion of the storage that spans the set of configurations. Figure 7.1 shows an example where in `Memory`, the memory address `a` stores the multivalue {`a1{0}, a2{1}`}. Seen another way, the memory address `a{0}` stores the multivalue `a1{0}` and the memory address `a{1}` stores the multivalue `a2{1}`. As shown by the top arrow in the figure, in shared execution, when a multivalue is loaded, execution splits by value, with each execution getting its own call stack that corresponds to a value and a set of configurations. In contrast, as shown by the bottom arrow in the figure, in deferred execution, a stack operand can be multivalued, which allows program execution to continue with a single call stack and allow more sharing. To allow a stack operand to be multivalued, the stackframe implementation and the following types of instructions had to be changed.

Reads/writes. In shared execution, a read pushed one value on the stackframe and a write popped one value off the stackframe. In deferred execution, a read/write may push or pop multiple values on or from the stackframe.

Method calls/returns. In shared execution, a method could not be entered with a multivalued argument, meaning a method had to be invoked against each combination of argument values. In deferred execution, a method can be entered with a multivalued argument, meaning code that creates a stackframe had to be changed to push multivalued arguments on the new stackframe. In shared execution, if a method returns multiple values, then multiple stackframes had to be created to accomodate each return value. In deferred execution, the return instruction was changed such that a multivalued return value is pushed on the caller stackframe.

Conditionals. In shared execution, a conditional instruction (`IF*`, `TABLESWITCH` and `LOOKUPSWITCH`) was executed against a single top operand value. In deferred execution, since the top operand can be multivalued, a conditional instruction may be executed as many times as there are top operand values, but the execution only splits into as many branches as there are unique values. This means that for an if-instruction, at most two children states, each with a stackframe whose program counter points to `false` or `true` branch, can be created.

Stack instructions. Stack instructions only change the stackframe, more specifically, popping top operand(s) as inputs, performing some computations on them, and pushing the output operand(s) on the stackframe. In shared execution, stack instructions did not have to be modified because an input operand could not have multiple values. In deferred execution, a stack instruction is executed for all combinations of input operands' values and the output operands, which may be multivalued as well, are placed on the stackframe.

7.3 Multiaddresses

Deferred execution also differs from shared execution in how memory instructions (loads/stores) work against multiple addresses. In Figure 7.2, arrows between `Shared Execution Memory` and `Stackframe` show that in shared execution,

when a load/store is to be performed against multiple addresses placed on the stack-frame, the values are loaded from or stored into the storage corresponding to each of the addresses. On the other hand, in deferred execution, arrows between `Deferred Execution Memory` and `Stackframe` show that a *multiaddress*, which is just a multivalue whose values are addresses, is formed from the multiple addresses, and the values are loaded from or stored into the corresponding storage *just once, rather than for each address*. In the example (Figure 7.2), the values `k{0,1}` are loaded from or stored into the multiaddress storage corresponding to the multiaddress `{a1{0}, a2{1}}`. Note that because `a1{0}` is a part of the multiaddress, `a1` is left with `a1{1}`, as shown in `Deferred Execution Memory` (since `a1{0}` and `a1{1}` together form `a1{allConfigs}`, which is written concisely as `a1`). Similarly, `a2` is left with `a2{0}`.

Deferred execution, shared execution and conventional execution all store exactly the same information, although deferred execution conceptually requires less storage space than shared execution, which conceptually requires less storage space than conventional execution. Figure 7.2 shows that shared execution's memory can be converted into conventional execution's memory by expanding an address-to-multivalue mapping into a set of address-to-value mappings. Similarly, deferred execution's memory can be converted into shared execution's memory by expanding a multiaddress-to-multivalue mapping into a set of address-to-multivalue mappings.

7.3.1 Writes

Figure 7.3, lines 4 - 13, shows how deferred execution handles writes. `a` is the address in which to store values. If `a` is not a multiaddress, then the values are placed into the storage corresponding to `a`. On the other hand, if `a` is a multiaddress, the corresponding multiaddress storage is created or retrieved and values are placed into the storage.

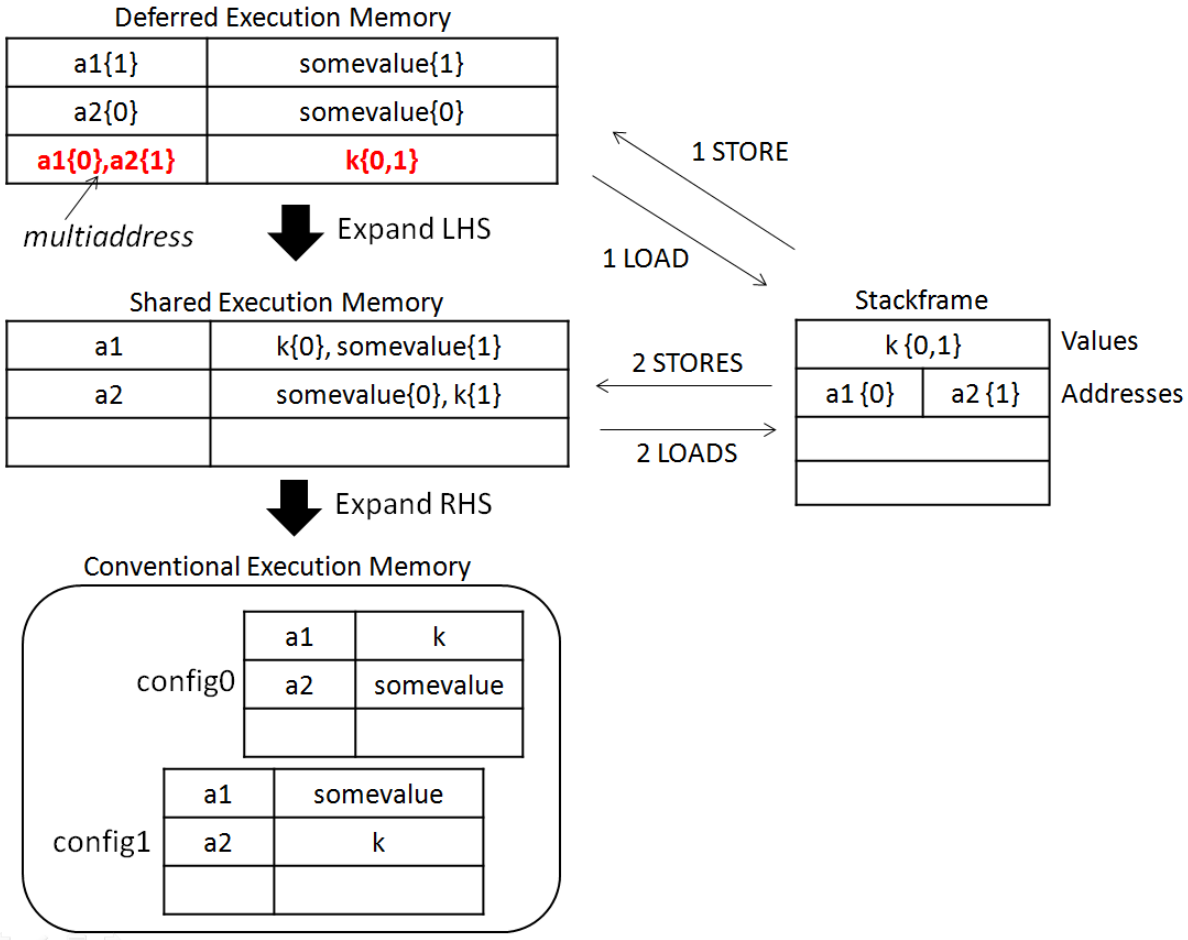


Figure 7.2: Loads/Stores in Shared Execution vs. Deferred Execution

```
1 class DeferredExecution extends VMListener {
2   void beforeInstruction(Instruction insn) {
3     ...
4     if(isStore(insn)) {
5       Values v = stackframe.pop();
6       Address a = stackframe.pop();
7       if(!a.isMultiaddress())
8         a.getStorage().put(v);
9       else {
10        a.getMultiaddrStorage()
11                        .put(v);
12      }
13    }
14    else if(isLoad(insn)) {
15      Address a = stackframe.pop();
16      Values values;
17      if(a.getStorage().exists())
18        values = a.getStorage().fetch();
19      else {
20        values =
21          a.findMatchingStorages().fetch();
22      }
23      stackframe.push(values);
24    }
25    ...
26  }
27  ...
28 }
```

Figure 7.3: Loads/Stores in Deferred Execution

For ease of implementation and quick access, an existing multiaddress storage should be accessed only by the multiaddress that created it. Unfortunately, a write to an address that is not identical to the multiaddress but overlaps with it must be able to access the multiaddress storage as well. A (multi)address $\{a1_1\{c_1\}, ..., a1_n\{c_n\}\}$, where c_i represents one configuration, *overlaps* with another (multi)address $\{a2_1\{c_1\}, ..., a2_m\{c_m\}\}$ if and only if there exists at least one *overlapping portion* pair $a1_i\{c_i\}$ and $a2_j\{c_j\}$ such that $a1_i = a2_j$ and $c_i = c_j$. In other words, two addresses overlap if they both share at least one memory address for one configuration. Figure 7.4 shows an example of an overlapping write, which adds the last row to the memory. Note that the write's multiaddress, {a3{0}, a2{1}}, overlaps with the existing multiaddress {a1{0}, a2{1}} since both multiaddresses share the address a2{1}. This is problematic because we now do not know whether to retrieve from the last row (m) or the second last row (k) when a2{1} is read (clearly, m should be read because it was the latest value written). To prevent overlapping storages, a storage for an address addr{config}, where config represents one configuration, can only be a part of one (multi)address storage (i.e. an address only appear once in the address column of the memory, either on its own row or in a multiaddress row). To satisfy this constraint, before the overlapping write takes effect, existing multiaddress storages that overlap with the write's address is destroyed, i.e. for each of the existing multiaddress storages, the storage's contents are moved into the individual addresses of the multiaddress. Therefore, in the example, before the write, k{0, 1} is moved into the individual addresses of {a1{0}, a2{1}}, meaning that k{0} is moved into a1{0} and k{1} is moved into a2{1}. Then the write is performed against the new multiaddress. The resulting memory is the bottommost memory in Figure 7.4.

Conceptually, overlap is determined by examining each memory address and comparing it with the write's address to see if there is any overlapping portion. How-

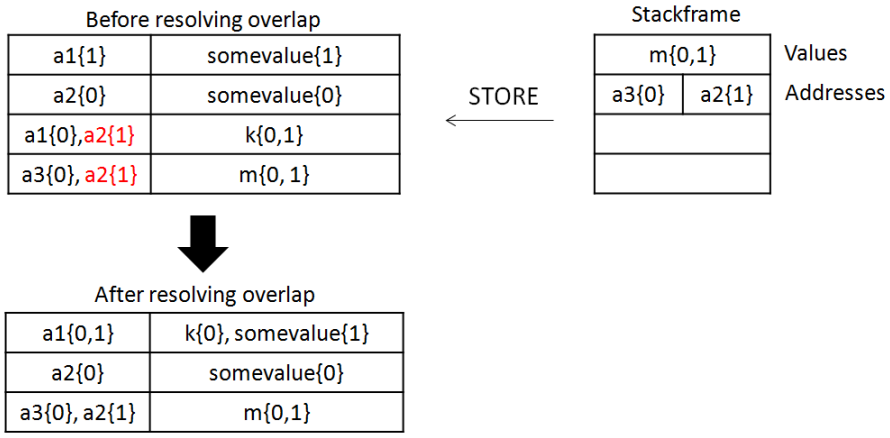


Figure 7.4: Example of an Overlap

ever, in implementation, overlap is determined much more efficiently using traceability links, as we will explain in Section 7.3.4.

7.3.2 Reads

Figure 7.3, lines 14 - 24, shows how deferred exceution handles reads. `a` is the address from which to read values. As with writes, ideally, there should be just one storage whose address is identical to `a` and the values should be fetched from the storage (lines 17 - 18). Unfortunately, there may not be such a storage, in which case we have to collect values from each storage whose address overlaps with `a` (lines 19 - 22) For example, suppose that a read is done against the address {`a1{0}, a2{1}`} in the bottom memory (`after resolving overlap`) in Figure 7.4. Since none of the existing addresses are identical to the read's address, values are read from `a1{0, 1}`, which matches the address for the portion `a1{0}`, and {`a3{0}, a2{1}`}, which matches the address for the portion `a2{1}`. The values read are {`k{0}, m{1}`}.

7.3.3 Method Invocations

In shared execution, an instance method could only be entered with one receiver. In deferred execution, an instance method invocation against multiple receivers splits execution by method dispatch, with multiple receivers entering the same dispatched method together. More specifically, in an executing state S that is executing the configurations $S_{configs}$ together, a method `x()` is invoked against the receivers $a = \{a_1\{C_1\}, ..., a_N\{C_N\}\}$, where C_i represents a set of configurations and $\bigcup_{i=1}^{N} C_i = S_{configs}$. Then the method invocation dispatches to one or more methods $\{Class_1.x(), ..., Class_K.x()\}$, where $K \leq N$, with one or more receivers $a_j = \{a_{j1}\{C_{j1}\}, ..., a_{jM}\{C_{jM}\}\}$ dispatching to (entering) a method $Class_j.x()$ together, such that $\bigcup_{j=1}^{K} a_j = a$. Also, if there is more than one dispatch (i.e. $K > 1$), the executing state S is split into children states $\{S_1, ..., S_K\}$, where a child state S_j, which has its own call stack, corresponds to a method dispatch $Class_j.x()$ and its receivers a_j. For a method dispatch, fields can be shared between the multiple receivers via multiaddress storages discussed earlier. Also, instructions pertaining to locals, not only of the dispatched method but also of methods transitively called from the dispatched method, will be shared between multiple receivers for the given method dispatch.

If there is a chain of instance method invocations with multiple receivers, combinatorial explosion of method dispatches can be eliminated through deferred execution. This situation can arise in object-oriented design patterns [35], such as *Visitor*, *Chain of Responsibility* and *Decorator*. For example, Figure 7.5 shows a product line of hierarchies of objects being visited. Each of the two parents can be linked to each of the two children, which allows four possible hierarchies. However, note that there is commonality between the hierarchies: each parent is linked to the same two children. In shared execution, `accept()` must be called against each of the two parents (line 15) and in turn, it must be called against each of the two chil-

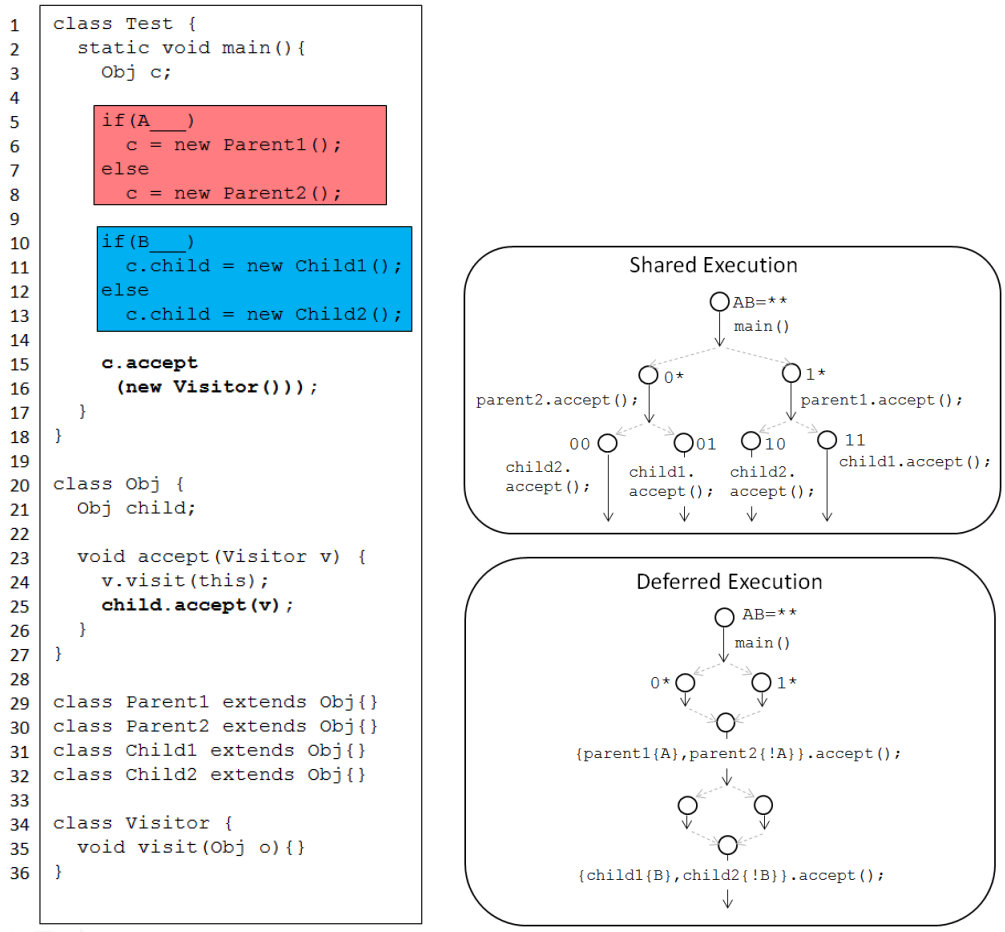
dren under each parent (line 25), meaning the method is called a number of times that is exponential in the number of features. However, in deferred execution, when the method is called against the two parents, because both invocations dispatch to the same method, the method is entered together by the two parents, meaning `this` within the method will be a multiaddress representing the two parents.[1] Similarly, the two children dispatch to the same `accept()` method and enter the method together. Thus, deferred execution eliminates the combinatorial explosion of `accept()` method invocations suffered by shared execution in this example.

7.3.4 Implementing Multiaddresses

Address Representation

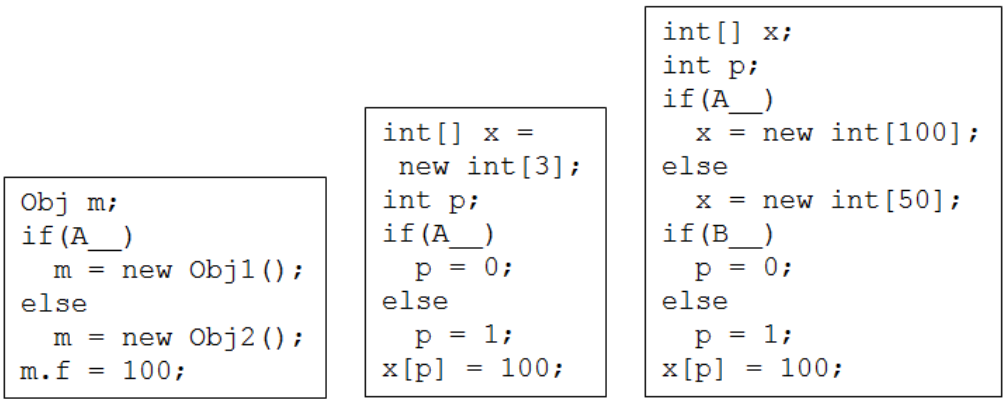
The idea of multiaddresses is not dependent on any particular address representation, but to understand how multiaddresses are actually formed, Namely, for a *Java Virtual Machine (JVM)*, which executes Java bytecode, an address for a store/load instruction consists of `base` (for local instructions, there is no base) and `offset`. A multiaddress can be formed due to an address's base, offset or both being multivalued on the stackframe, meaning that there are four possibilities (neither being multivalued, just base being multivalued, just offset being multivalued, and both being multivalued). Figure 7.6 shows an example for each of the three possibilities where base and/or offset is multivalued. Note that offset can only be multivalued with arrays because a field instruction's field index cannot be passed around as values. For example, in Figure 7.6(a), 100 is placed in the multiaddress storage for the multiaddress `{obj1{A}, obj2{!A}}.f`, meaning the field `f` is shared between the two objects. In Figure 7.6(c), 100 is placed in the multiaddress storage for the multiaddress `{base={addrOfArray100{A},`

[1] Note that each parent must be checked separately to check for null-pointer exception and to see if they dispatch to the same method and this is why execution splits in the diagram before entering the `accept()` method.



(a) Code (b) Execution

Figure 7.5: Visitor Example



(a) Multiple Bases (b) Multiple Offsets (c) Multiple Bases and Offsets

Figure 7.6: Different Ways of Forming a Multiaddress

`addrOfArray50{!A}}, offset={0{B},1{!B}}}`, which represents the four array locations due to combinations of the two arrays and two offsets. Figure 7.6(b) is similar to Figure 7.6(c), but the base is not multivalued.

Memory Structure

Shared execution's memory, described in Section 6.4.1, is extended to allow multiaddresses. There exists a separate memory for multiaddress storages, meaning that if there is no load/store that accesses a multiaddress, this separate memory is not needed and memory access is no different than as was in shared execution.

To efficiently determine overlap for reads and writes, when a multiaddress storage is created due to a write to $a = \{a_1\{c_1\}, ..., a_n\{c_n\}\}$, where c_i represents one configuration, each address/configuration pair $a_i\{c_i\}$ is linked to a. Thus, the next time a read/write is done against a (multi)address $b = \{b_1\{c_1\}, ..., b_m\{b_m\}\}$, rather than having to search the memory for overlapping storages, we can simply enumerate each address of the (multi)address and see if it is linked to a multiaddress storage. If there is at least one address that is linked to a multiaddress storage, b is overlapping. For example, in Figure 7.7, the write in line 12 creates a multiaddress storage for

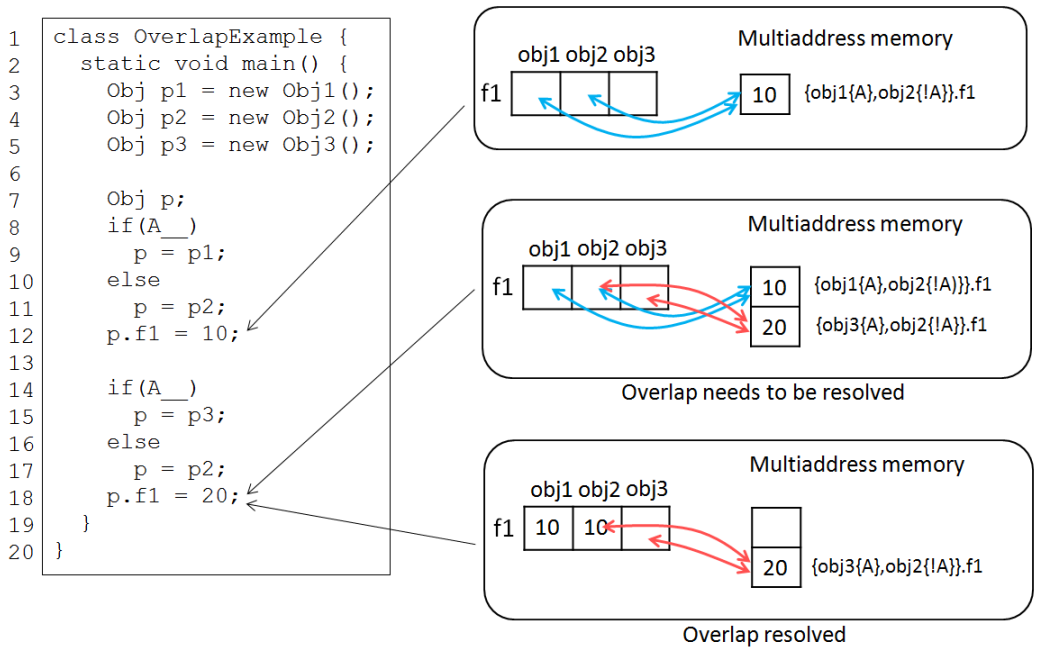


Figure 7.7: Traceability Links

$obj1$'s field1 (for A) and $obj2$'s field1 (for $!A$) and places the value 10 in the storage. Note that each object's field is bidirectionally linked to the multiaddress storage. Then line 18 writes 20 to $\{obj3\{A\}, obj2\{!A\}\}.f1$ and if we did not handle overlap, the memory in the right middle diagram would result, with $obj2$'s $f1$ erroneously having two values, 10 and 20. We handle the overlap by enumerating the write's multiaddress into individual addresses $obj3\{A\}$ and $obj2\{!A\}$ and checking to see if each is linked to a multiaddress storage. Since $obj2\{!A\}$ is linked to the existing multiaddress storage for $\{obj1\{A\}, obj2\{!A\}\}.f1$, the existing multiaddress storage is destroyed (its content 10 is moved into the individual storages). Then the write proceeds, creating a multiaddress storage for $\{obj3\{A\}, obj2\{!A\}\}.f1$ and placing the value 20 in there. Therefore, the write yields the memory shown in the right bottom diagram in Figure 7.7.

7.4 Evaluation

Deferred execution was implemented on top of Java PathFinder (JPF) [83], but can also be implemented on top of any standard virtual machine (VM), such as Jikes RVM [48]. JPF is typically used as a model checker rather than as a VM, but we use it as a VM because of its extensibility and our familiarity with it. JPF is considerably slower than a standard VM, but since our comparisons are relative to related implementations also in JPF, using JPF should not affect our results.

We evaluated our technique against scenarios where there are known opportunities for our technique to be effective. Table 7.1 shows the results, where all nmeric entries except *Configs* are in seconds. As discussed in Section 7.3, deferred execution can be particularly effective for chains of polymorphic method invocations, where each polymorphic method invocation can be against a different receiver for a different configuration. Therefore, we selected 3 SourceForge open-source programs that use the following Gang of Four (GoF) design patterns [35] that rely on a chain of polymorphic method invocations: *Decorator* (used by `FitNesse` [34]), *Chain of Responsibility* (used by [93]), and *Visitor* (used by [44]).

There are three tests for each subject, each of which corresponds to the expected level of saving achieved by deferred execution: `High`, `Medium` and `Low`. Each test exercises a fixed length of chain of method invocations. There are three test cases for a test, each of which introduces a number of variation points in the chain, thereby creating a "product-lined" version of a design pattern (the support for the design pattern was already in place in each subject). Namely, at each method invocation in the chain of method invocations, variability can be introduced such that the invocation's receiver is determined by an `if-else` of a feature as shown in Figure 7.5. Effectively, what was just one input to the test case, a chain of objects, now becomes multiple inputs or multiple chains of objects and therefore the test case must be run against each input. The three test cases encode a numbers

of configurations between 4 (2 method invocations or features) to 16 (4 method invocations or features), as shown in the column `Configs`. The test cases serve to show that deferred execution can scale as the number of configurations increases. Each row of the table corresponds to a test case. Each test case was executed using five different techniques: *Deferred*, *No-Multiaddress*, *Conventional*, *ISSRE12-Shared*, and *ISSRE12-Conventional*. *Deferred* is the technique presented in this chapter. *No-Multiaddress* is the technique presented in this chapter, but with the multiaddress functionality (described in Section 7.3) turned off to see how much difference the key contribution of this chapter's technique (multiaddresses) makes. *Conventional* runs each configuration from start to finish. *ISSRE12-Shared* is shared execution, which was published in [60] and is a technique that deferred execution improves on. *ISSRE12-Conventional* runs each configuration from start to finish, but is different from *Conventional* in that the former shares code base with *ISSRE12-Shared* and uses a slightly different native code implementation for strings. However, for the subjects studied, the entries in *ISSRE12-Conventional* are nearly identical to *Conventional*, so the former can be ignored.

Deferred execution implementation, subjects, and results can be downloaded from our website [57].

7.4.1 FitNesse

`FitNesse` is an acceptance testing framework that allows, among other functionalities, a table that holds test data to be manipulated through GoF decorators of operations over columns [34]. Decorators, which are objects that have the same interface as the object they are decorating, are connected through a chain (nesting) of method invocations, i.e. one decorator adds some functionality to an existing decorator, which adds to an existing decorator, and so on. The chain is product-lined such that each method invocation is conditionalized by a feature as was done in Fig-

Table 7.1: Evaluation

Subject	*Test*	*Configs*	*Deferred (sec.)*	*No-Multiaddress (sec.)*	*Conventional (sec.)*	*ISSRE12-Shared (sec.)*	*ISSRE12-Conventional (sec.)*
FitNesse	HIGH	4	8	16	5	11	5
FitNesse	HIGH	8	8+	29	9	28	8
FitNesse	HIGH	16	8+	54	16	85	15
FitNesse	MEDIUM	4	7	16	5	11	5
FitNesse	MEDIUM	8	8+	29	9	28	8
FitNesse	MEDIUM	16	8+	52	16	85	15
FitNesse	LOW	4	7	16	5	11	5
FitNesse	LOW	8	8+	28	9	28	8
FitNesse	LOW	16	8+	53	17	86	15
SuperCSV	HIGH	4	12	21	10	19	10
SuperCSV	HIGH	8	13+	40	19	36	19
SuperCSV	HIGH	16	13+	80	36	69	36
SuperCSV	MEDIUM	4	19	20	10	19	10
SuperCSV	MEDIUM	8	20	40	18	35	18
SuperCSV	MEDIUM	16	20+	78	35	68	36
SuperCSV	LOW	4	29-	28	14	27	14
SuperCSV	LOW	8	77-	57	27	57	27
SuperCSV	LOW	16	131-	112	52	113	52
HTMLCleaner	HIGH	4	8	16	5	11	5
HTMLCleaner	HIGH	8	8+	27	10	18	10
HTMLCleaner	HIGH	16	8+	42	19	29	19
HTMLCleaner	MEDIUM	4	13	18	6	12	6
HTMLCleaner	MEDIUM	8	16	29	11	20	11
HTMLCleaner	MEDIUM	16	18+	44	20	31	21
HTMLCleaner	LOW	4	33	67	18	47	18
HTMLCleaner	LOW	8	40	121	34	92	34
HTMLCleaner	LOW	16	50+	210	67	151	66

ure 7.5, meaning that the method invocation will be against one decorator (which increments a cell value) or another (which decrements a cell value) depending on whether the feature is present or not. Note that the two decorators are instances of two different classes with a common parent class and they dispatch the method invocation to the same method implementation (otherwise, deferred execution is guaranteed to not be effective). For `HIGH` test cases, each decorator is designed to work against a unique column, meaning that each decorator operates on independent data from one another and the running time should be linear in the number of decorators (which is twice the number of features). For `MEDIUM` test cases, different method invocations work against different columns, but the two decorators for one method invocation work against the same column, meaning that the second decorator for a method invocation has to do twice as much work, once against the column unaffected by the first decorator and once against the column affected by the first decorator. For `LOW` test cases, all the decorators work against the same column, meaning that the last decorator in the chain must perform the same work a number of times that is combinatorial in the number of features.

Let us examine the results for `High` test cases. Using the conventional approach (`ISSRE12-Conventional` and `Conventional` columns), as the number of configurations doubles (column `Configs`) , the running time approximately doubles as well. Shared execution actually performs even worse due to its overhead for these particular test cases (column `ISSRE12-Shared`). Deferred execution without multiaddress (column `No-Multiaddress`) performs better than shared execution, but still suffers from a nearly doubling growth and performs worse than the conventional approach. However, deferred execution with multiaddresses (column `Deferred`) outperforms the other techniques except for running 4 configurations using the conventional approach. Most importantly, deferred execution's running times do not double and in fact, it remains largely unaffected by the doubling of

configurations as expected. In fact, the running times remain nearly constant because it seems that the time it takes for decorators to do actual work (incrementing or decrementing integer cell values, which is not very expensive) is dominated by the time it takes to setup decorators through reflection. This means that as long as the decorators do not have to be setup a combinatorial number of times as with deferred execution, adding variability makes little difference. The results for `MEDIUM` and `LOW` test cases are very similar to those for `HIGH`, despite the expectation for deferred execution to not be as effective. We believe that the reason for this is again because the setup time is dominant, meaning that relatively, the time it takes to perform the decorator operations a number of times that is combinatorial in the number of features has little impact on the overall running time.

7.4.2 SuperCSV

`SuperCSV` is a software package for handling files in Comma Separated Values (CSV) format and uses GoF Chain of Responsiblity design pattern for allowing users to define a chain of operations called *Cell Processors* (such as constraint checks) against a CSV value represented as a cell in a table [93]. A cell processor (a *processing object* in GoF terminology) can handle the CSV value (the *command object* in GoF terminology) and/or pass it to the next cell processor in the chain. As with decorators, we product-lined the chain of cell processors such that each method invocation is conditionalized by a feature and called against one of two related cell processors that dispatch the invocation to the same method. Results show for all tests that for conventional execution, the running times approximately double as the number of configurations doubles, as expected. Shared execution performs considerably worse than conventional execution and the former also approximately doubles in execution time as the number of configurations doubles, for all tests. `No-Multiaddress` does not do any better than shared execution and in fact per-

forms slightly worse. However, for HIGH and MEDIUM, deferred execution largely outperforms the other techniques and most importantly, keeps the running time almost constant despite the increase in the number of configurations. The reason for this saving is that the two cell processors in one level are independent from the two cell processors from the next level, i.e. do not operate against the same data. However, for LOW test cases, where all the different levels of cell processors work on the same data, even though multiple objects can enter a method together, within the method, splitting will occur and at the end of the chain, a computation has to be performed for all combinations of the features. Indeed, the corresponding results are considerably worse than those for MEDIUM and HIGH and in fact, show that deferred execution performs the worst of all techniques because of the overhead associated with managing multiaddresses.

7.4.3 HTMLCleaner

HTMLCleaner is an open-source HTML parser written in Java that provides a GoF Visitor class to the user to perform operations against a hierarchy of HTML tags. The user creates an instance of the visitor, calls the root tag's `accept()` method with the visitor as an argument, which in turn calls back visitor's `visit()` method as the root tag as an argument and goes through the root tag's children to make them accept the visitor. We product-lined the chain of `accept()` method invocations such that at a method invocation, one of two possible tags can be chosen depending on the feature selection. Results show that conventional execution doubles as the number of configurations doubles, as expected, and shared execution and No-Multiaddress do a similarly poor job (the latter actually performing a poorer job), performing worse than conventional execution. However, deferred execution remains largely unaffected by the doubling of the number of configurations, exhibiting only constant increases for MEDIUM and LOW, and outperforms other techniques

for test cases with 16 configurations.

7.4.4 Overall Results

To summarize the performance of deferred execution, the table identifies the best and worst cases for deferred execution for the subject tests. For each row, we identify values in the `Deferred` column, which are the best (lowest times, which are marked '+') or the worst (highest times, which are marked '−') among all the techniques for that row. All the worst times are for `SuperCSV`'s `LOW` test cases and this is likely due to the subject SPL and the test itself rather than the corresponding design pattern, Chain of Responsibility, since all three design patterns evaluated are structurally similar (they all allow a chain of polymorphic method calls) from the perpsective of deferred execution. The fact that deferred execution can perform the worst shows that there must be a sufficient number of opportunies for exploiting multiaddresses that can offset the overhead. Despite these negative results, deferred execution outperforms the alternatives in 13 (marked +) out of 27 (48%) test cases. More importantly, as discussed, it is able to limit and even prevent the growth in the running time despite the growth in the number of configurations.

7.4.5 Threats to Validity

The subjects were selected specifically to demonstrate deferred execution's ability to reduce combinatorial explosion in the number of polymorphic method invocations for a chain of such method invocations. Therefore, the results cannot be generalized to settings where such a chain of method invocations do not occur. Also, we cannot generalize our results to all configurable systems that exhibit such a chain of method invocations because our subjects may not be representative of such configurable systems. To reduce this threat, we designed the experiment in the following ways. We used multiple Java programs from SourceForge, which are more realistic than

programs developed in-house. We used programs that use design patterns, where chains of method invocations occur naturally. Although we could not find tests that explicitly test a chain of method invocations and had to construct the tests ourselves, the tests were constructed to cover a range of configurations and a range of degrees of expected effectiveness.

7.5 Related Work

The key contribution of deferred execution is the idea of using multiaddresses to maximize sharing computations between executions. We are not aware of any work with this contribution, but there are related works.

Parallelization. Because each configuration corresponds to a separate execution, from parallel programing's perspective, the configurations are embarassingly parallel. Therefore, the easiest, but the most resource-intensive, alternative to deferred execution would be to execute each configuration using a dedicated processor. A more practical approach is to assume that the number of dedicated processors will be smaller than the number of configurations and to test as many configurations as there are number of processors in parallel at a time. A more sophisticated technique can make the most use of parallelization to speed up execution of multiple configurations. Deferred execution is fundamentally more economical than parallelization in that the former aims to use all the computing power that is available in a single processor. However, if the additional hardware resources are available, one could imagine combining deferred execution with parallelization. We are not aware of existing works that use parallelization to speedup execution of configurable systems. However, parallelization has been used to execute two versions of a software, one before a patch and another after, simultaenously and to execute the more reliable version when their executions differ [77].

Symbolic Execution for Multiexecution. *Rozzle* [63] is a JavaScript

multiexecution VM for exposing environment-specific malware that, like our work, explores multiple execution paths within a single execution. However, our purpose is to optimize a given set of executions by exploiting similarity between them, while their purpose is to find bugs. Our technique preserves the given set of executions and uses only concrete values, whereas their technique uses a form of symbolic execution (that does not use concrete values and does not execute all iterations of a loop) that allows infeasible and unsound executions (i.e. executes both paths of a branch even if only one of the paths may be satisfiable). Also, note that any form of symbolic execution is fundamentally different than our technique as the former explores the execution space with symbolic values for increased coverage, whereas our technique explores the execution space with a grouping of concrete values for increased speed.

Shared Execution. Deferred execution improves shared execution [60] by delaying splitting as late as possible, i.e. splitting on branches rather than on values, to maximize sharing between executions and introduces the idea of multiaddresses to enable this. Detailed technical differences were discussed throughout the chapter.

Variability-Aware Interpreters. [51] proposes the idea of variability-aware interpreters for executing multiple configurations in a single execution, like shared/deferred execution. However, their work does not merge paths, meaning that once execution splits due to different values from different configurations leading to different branches, the executions can never be merged to resume sharing. Also, their work is implemented on programs written in the `WHILE` toy language, while our work is implemented on programs written in Java.

Delta execution for online patch validation. In [97], for online path validation, a program and its patched version is run as a single program until the patch is encountered, which splits execution using a system fork with copy-on-write sharing of pages. Merging is attempted where processor states are likely to be identical, i.e. at a function return or earlier and execution splits again when a

page modified by the patched execution is accessed. The general ideas of running multiple executions using a single execution state (i.e. processor state in theirs call stack in ours), splitting due to accessing different memories, and merging at execution points with the same execution states are the same between our work and their work. However, their work is tailored to two nearly identical executions (what they call *multiple almost redundant executions*). On the other hand, our work is tailored to finding redundancy across many different executions induced by a family of programs with different functionalities. Because their executions are highly similar by definition, their focus is more on reducing overhead rather than finding fine-grained similarity across many, potentially very different, executions, which is our focus. Therefore, instead of monitoring memory access and splitting on variable values like we do, they monitor page access and split on page content, which is cheaper but can miss instruction-level similarity that our technique is able to detect.

Delta execution for explicit-state model checking. In [29], multiple states are explored in a single execution to eliminate redundancy in explicit-state model checking. When model-checking a single program, entire program states/heaps may be similar or even identical and their technique is focused on eliminating redundant operations against program states/heaps. In our work, because the programs represent different combinations of features and therefore can run considerably differently, we have to find as much commonality as possible, which is why our technique is focused on much finer-grained redundancy, i.e. bytecode instructions against the same input operands. This difference in setting explains why their merging process is more expensive than ours: they examine the different executions using a standard model checking optimization called *linearization* to see which have identical heaps and thus can be treated as a single execution. In our setting, linearization is not effective because the likelihood of the different executions,

which represent different feature combinations or functionalities, having identical heaps is unlikely. And it is probably because merging is expensive that they require merge points to be syntactically specified, whereas our work tries to merge where call stacks are likely to be the same. However, although their merging cost is high, their technique does not need to split/merge as often as our shared execution technique because they allow input stack operands to be multivalued, which means splitting is triggered only by a branch point, like in our deferred execution. But their technique is not able to share instructions against different memory locations (i.e. reads/writes must be repeated for different memory locations and the same method must be invoked against each receiver), which is the key contribution behind deferred execution. Also, their technique differs from both shared and deferred execution in that they require source code instrumentation, although they suggest the idea of an interpreter-based approach like ours.

Thin slicing. In [91], the definition of a program slice is refined to include only the statements that produce the value of a variable (i.e. local, field, array element) at a statement, ignoring statements that affect container or *base pointer* of the variable (i.e. objects and arrays) based on the notion that the latter statements do not help with program understanding. For example, suppose that slicing is performed against an element in a Java `List` at a statement (e.g. `list.get(i)`). The slice would not include statements affecting the list itself, such as those that add new elements to the list. This idea of *thin slicing* is similar to our work in that both works abstract base pointers away (they ignore them and we treat multiple base pointers as a single multiaddress) to reduce the analysis result (slice in theirs and execution traces in ours). However, the idea of deferring execution to speed up execution of multiple programs is different from the idea of reducing program slices to improve program understanding.

7.6 Summary

We presented deferred execution, a technique that executes multiple configurations of a configurable system in a single execution so that a bytecode instruction common to the different configurations is executed just once. The technique can allow more bytecode instructions to be shared than the predecessor technique of shared execution by deferring splitting as late as possible, namely, splitting on branches rather than on variable values. To enabling deferring, the technique uses 1) a specialized stackframe in which an operand can carry different values for different configurations and 2) multiaddresses, each of which represents a group of memory locations that can be treated as a single location. Deferred execution is useful for programs that use arrays and especially for object-oriented programs, where multiple receivers can enter a method together. Deferred execution was evaluated on three open-source subjects that use well-known design patterns and results showed that in certain cases where running times for shared and conventional executions increase proportionately to the increase in the number of configurations, deferred execution's running times only increase by a constant factor and can even remain constant.

Chapter 8

Discussion and Future Work

This chapter presents some practical considerations as well as future work ideas.

8.1 Threats to Validity

Our case studies suffer from external validity: we cannot generalize our results to all SPLs because the SPLs and tests/safety properties used in our evaluation may not be representative of all possible product lines and tests/safety properties. To reduce this threat, for SPLs, we used multiple Java SPLs, many of which have also been used by other research groups as noted in the previous chapters, and one real, large industrial codebase belonging to Groupon. For tests/safety properties, we designed tests, which simulate worst, average and best scenarios, and realistic safety properties and we used a very large suite of existing tests for Groupon's codebase.

8.2 Integrating the Techniques

At present, the user of our tool-set chooses which of the five techniques to use when given a test to run or a safety property to check. For checking safety properties, the choice is straightforward (i.e. use static pruning of configurations to monitor

from Chapter 5), but for running a test, the choice is not as obvious. Consider the case when the test checks a small portion of the product line, like a unit test does. If there exist many inputs and each test run takes a long time, choose the static approach (Chapter 3), and otherwise, choose the dynamic approach (Chapter 4). Alternatively, if the test checks a large portion of the product line, like a system integration test does, then the user should choose deferred execution (Chapter 7, which improves on Chapter 6).

To ease the requirement on the user to determine what type a test is, it may be possible to integrate the test running techniques into a tool chain to automatically determine which technique to use. However, the extra level of analysis required to determine which technique to use may be end up being more expensive than obtaining the information through user knowledge. One can also imagine an integration that would allow multiple techniques to be applied for a product line test, e.g. first apply a technique for pruning configurations and then apply a technique for pruning bytecode instructions between the remaining configurations. However, a more sophisticated integration than just applying the techniques in series may be required, since the increased saving in execution time may not always be offset by the overhead of applying two separate techniques (especially since shared/deferred execution already performs the reachability analysis that `SPLat` and its static counterpart in Chapter 3 perform).

8.3 Improving the Techniques

The presented techniques may be improved in at least two dimensions: speed and scalability. All the techniques 1) eliminate redundancy, be it in configurations or in bytecode instructions, and 2) incur an overhead in doing so. This means that opportunities for improving speed are in identifying more redundancy and in reducing the overhead. For example, just as deferred execution improves shared execution's

speed (by identifying redundancy in memory locations), another technique may in turn improve deferred execution's speed. Unfortunately, it seems hard to optimize both 1) and 2) together. Namely, the three techniques that prune configurations (Chapter 3, Chapter 4, and Chapter 5) clearly incur lower overhead than the two techniques that prune bytecode instructions (Chapter 6 and Chapter 7) because not every bytecode instruction has to be intercepted and memory is not expanded to hold memories of all possible configurations. However, the former three techniques cannot eliminate as much redundancy as the latter two can.

A scalable technique should work against large and complex programs without many limitations. Unfortunately, using a specialized VM like JPF, which requires native code to be explicitly modeled and performs much slower than a standard VM, does not make the implementation of a technique very scalable. One way to address the scalability of shared/deferred execution would be to implement them using a standard VM; we chose JPF as the platform due to its ease of programming and extensibility. Also, while it may not be possible to achieve both the maximal redundancy elimination and minimal overhead, characteristics of techniques that achieve maximal redudancy elimination, such as deferred execution, and minimal overhead, such as SPLat, may be exploited to develop a scalable and effective technique.

8.4 New Problems and Solutions

This thesis only includes techniques for tests and safety properties, but there exist other types of analyses for product lines as well. [16, 19] present techniques for efficiently performing intra- and inter-procedural dataflow analyses for a product line. [24, 23] present techniques for efficiently model-checking product lines. Additionally, there may be other types of properties that require techniques dedicated to them. One can even imagine a product line of product line analyses problems and

solutions. Finally, as the product line evolves, it may be possible to converge on a unifying or integrated solution.

8.5 Customizable Multiexecution

In this section, a future work idea that improves on shared execution and deferred execution is presented. One problem with shared/deferred execution, which will be referred to collectively as *multiexecution*, is that their *applicability* is limited because they enforce a strict rule for allowing computations to be shared. Namely, they require call stacks of the different executions to be identical, even though the only requirement for sharing a computation is that 1) the different executions are about to execute the computation and 2) inputs to the computation are identical (program counter is not an input). Another problem is as follows. While numerous techniques have been developed to enable multiexecution in various research areas [6, 63, 60, 29, 97, 77], the techniques do not seem to share any implementation code, meaning that there is a considerable amount of redundant programming for bookkeeping, backtracking, and equivalence checking across these techniques and multiexecution techniques that will be developed in the future. Yet another problem with multiexecution is the user's lack of control, which makes it difficult to offset the overhead of multiexecution. To illustrate, suppose that there is an expensive computation that the user knows can be executed identically across different configurations, but the multiexecution implementation misses the opportunity to execute that computation just once for the different configurations. With current techniques, it is not possible to configure the multiexecution implementation such that the opportunity is not missed.

We introduce the idea of *customizable multiexecution*, a framework for multiexecution techniques that allows the user of the framework to define where sharing is likely to take place across different executions using unconventional aspects called

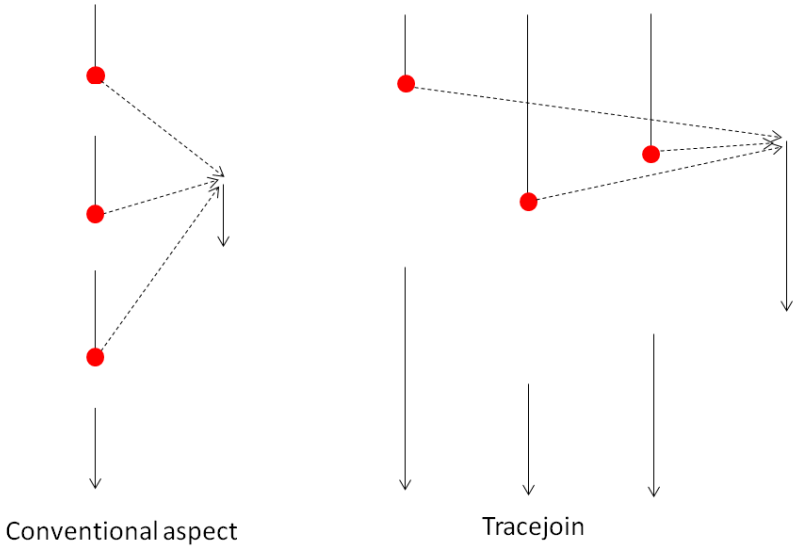


Figure 8.1: Tracejoin for Customizable Multiexecution

tracejoins and frees her from having to implement the bookkeeping code. Figure 8.1 illustrates the difference between a conventional aspect [53] and a tracejoin. A conventional aspect intercepts events scattered throughout a single execution and performs the same computation at each event. In contrast, a tracejoin intercepts events *across* executions and performs the same computation *just once* for all the events. While a conventional aspect aims to reduce development effort by localizing the code for the common computation, a tracejoin aims to reduce the execution time by executing the common computation just once.[1] Appendix discusses tracejoins in more detail.

[1]Technically, one could also apply a tracejoin for a single execution to execute the common computation just once, but this will be explored in the future.

Chapter 9

Retrospective

In this chapter, we reflect on the dissertation work and discuss what was done correctly (Section 9.1), difficulties encountered (Section 9.2) and what could have been done differently (Section 9.3).

9.1 Positives

One of the merits of this dissertation is that its techniques have the common focus of efficiently checking a property against a product line. Forming a coherent set of techniques may not have been possible had we pursued our plan to examine other problems of verification, such as fault localization and repair, after completing the works on pruning configurations. By staying the course and identifying the problem with techniques on pruning configurations, namely that the techniques are completely ineffective in cases where every configuration must be checked, we were able to develop shared execution. Similarly, deferred execution and customizable multi-execution were developed by examining problems with shared execution. Forming a dissertation this way, through a progression of techniques, seems to be a good practice.

9.2 Difficulties

Skills in static program analysis, dynamic program analysis and runtime monitoring needed to be acquired in order to develop the techniques in this dissertation and the acquisition took longer and was more difficult than had originally been anticipated. Also, the same can be stated for using these skills to implement the tools behind the techniques (using the `Soot` static analysis framework, `JPF`, `tracematches`, and `Korat`). In addition, studying the vast amount of related work in verification research took a considerable amount of effort.

Another difficulty was in finding product lines with tests and safety properties. because product line verification research is a relatively new research area. We ended up manually constructing tests and safety properties and converting existing open-source software into product lines. Consequently, as discussed in Section 8.1, the product lines and tests used may not be representative of all possible product lines and tests. The subjects that were used to evaluate our techniques have been added to *Software-artifact Infrastructure Repository* (SIR) [89], a repository for benchmarks that verification researchers can use to evaluate their techniques.

9.3 Hindsight

The problems and solutions in this dissertation were largely discovered and developed with the mindset that they are unique to product line verification. As a result, it took a while to discover and develop the problems and solutions. It turned out that similar problems and solutions exist for verification of conventional programs. For example, pruning configurations is similar to pruning paths, which is used by `Korat` [18] to efficiently generate test cases. Also, pruning bytecode instructions between configurations is similar to pruning computations between paths of a model checking search [29] and program versions [97]. So a better approach may have been

to examine these existing non-SPL verification problems and solutions and adapt them to work with product lines.[1] Had we taken this approach, we would have accomplished considerably more in a shorter period of time.

Having stated the previous remark, it may be possible that there are product line verification problems and solutions that are completely unique to product lines. For example, the dissertation's techniques largely treat the product line as a monolithic artifact with variation points (if-conditionals) spread out through it. It may be possible to develop more efficient and effective techniques by exploiting the notion that features are modules just like the modules of the programming language.

[1]The dissertation author has realized the hypocrisy of a product line researcher not reusing existing solutions from related research areas.

Chapter 10

Conclusion

An SPL represents a set of related programs, each of which is defined by a combination of features, which are user choices that enable or disable functionality. An SPL serves a dual purpose: to reduce development effort by allowing multiple programs with commonalities to be treated together and to reduce the time-to-market by targeting multiple customers simultaneously. Checking a property against a product line is more difficult than checking it against one program because the property must be checked for each program in the product line, where the number of distinct programs can be exponential in the number of features.

We presented a suite of complementary techniques based on static and dynamic analysis for efficiently testing product lines. The first set prunes configurations and the second prunes bytecode instructions between configurations even when configurations cannot be pruned.

In the first set, we exploit the fact that a test is likely to exercise a subset of the product line code. *Statically pruning configurations to test* (Chapter 3) is a technique that uses static analysis to identify features that are reachable from the test and further reduces the set of features to those that can influence the test output. The test then needs to be run only on the combinations of those features.

The second technique, *SPLat* (Chapter 4), presents a lightweight dynamic analysis alternative, discovering the reachable features during execution using stateless exploration, which is scalable. The third technique, *statically pruning configurations to monitor* (Chapter 5), statically reduces the configurations to monitor for safety property violation by inserting the monitor only on configurations that allow the monitor's instrumentations to be reached.

Even when configurations cannot be pruned, a tester does not have to resort to executing each configuration from start to finish by using *shared execution* (Chapter 6), which executes all the configurations together to exploit the redundant bytecode instructions between the configurations arising due to behavioral similarity inherent in a product line. *Deferred execution* (Chapter 7) improves on shared execution by allowing different memory locations to be treated identically, which can considerably increase the amount of shared computations.

We discussed practical considerations and future work ideas (Chapter 8), including integrating the techniques into a tool chain, making the techniques faster and more scalable, and identifying new types of properties that require program analyses techniques for efficient checking. As a more concrete future work item, we also presented *customizable multiexecution*, which provides a unifying framework for multiexecution and gives the user the ability to control where sharing takes place.

Even though reducing the execution space is not a new problem and advances to address the problem, including the presented techniques, are often not fundamentally groundbreaking, they are nonetheless important because they save what is arguably the world's most important commodity: time. This importance will only continue to grow as programs grow in complexity and size. Also, in a world that values productivity and always demands more of it, techniques for efficiently testing product lines will too be valued and more value will be demanded from them.

Appendices

Chapter A

Tracejoins

A.1 Motivating Example

Figure A.1(a) shows an example program that calls `foo()` identically for different inputs. Suppose that the test is multiexecuted for a positive input and a negative input. Neither shared nor deferred execution can allow `foo()` to be executed just once because the method is called in different branches. Also, existing multiexecution techniques do not give the user the freedom to choose when multiexecution takes place. For example, suppose that the user knows that multiexecution of `foo()` is likely to be only be beneficial for large or specific string inputs. Customizable multiexecution through tracejoins allows the user to specify multiexecution only for these situations.

A.2 Technique

Like in shared/deferred execution, memory is essentially an array that keeps memories of the different executions arising from different configurations (inputs). Also, like in shared/deferred execution, at each execution point (before or after a bytecode instruction), a state keeps track of the set of configurations that are executing

```
class Program {
  static void foo(String t) {
    assert !t.contains
      ("annoying string");
    assert !t.contains
      ("forbidden string");
  }

  static void main(int x){
    if(x > 0) {
      ...
      foo("hello world");
      ...
    }
    else {
      ...
      foo("hello world");
      ...
    }
  }
}
```

(a) Code

```
1  tracejoin Program {
2    pointcut joinpoints(String s):
3      execution
4        (void Program.foo(String)) &&
5        args(s);
6
7    boolean mergepoint(String[] ss):
8      new HashSet<String>(ss).size() == 1;
9
10   void around(String[] ss):
11     joinpoints(ss[-]) && mergepoint(ss){
12     proceed(ss);
13   }
14 }
```

(b) Tracejoin

Figure A.1: Example of Customizable Multiexecution

together and a hierarchy of states is formed from splitting/merging. Initially, configurations execute separately with a separate call stack. The user-specified tracejoin determines where each configuration should pause execution. Once all the configurations have been paused, the program states of the configurations are compared to see if the following instruction can be executed identically. At minimum, the configurations' call stacks must have the same program counter and the inputs to the instruction pointed to by the program counter must be identical for the call stacks. After the instruction is executed just once for all the configurations, the same checks are applied and sharing continues until a check fails, at which point execution is split by configurations and each configuration's execution may be paused by tracejoin, repeating the process.

Figure A.1(b) shows a tracejoin that allows `foo()` to be executed just once for two test cases with a positive input and a negative input. Line 11 states that executions should be paused according to `joinpoints()` *pointcut* and their program states compared according to `mergepoint()` function. A pointcut is an AspectJ construct that specifies which events to intercept. In this case, the pointcut pauses each configuration's execution immediatey after the execution enters `foo()` and

picks out the String argument as `s`. Once all the configurations have been paused, an array of String, `ss`, is available (note that `joinpoints(ss[-])` in line 11 specifies that each element of the String array has been picked out by the joinpoint). `mergepoint()` requires that all the array elements are identical (line 8). If `mergepoint()` check passes, an additional check is performed by the underlying customizable multiexecution implementation to see if the configurations' call stacks have the same program counter and the inputs to the next instruction are identical. Assuming that pushing the address of "annoying string" on the stack frame is the first instruction of `foo()`, the additional check passes and line 12 is executed, which pushes the address just once for all the configurations. Suppose that the next instruction is to load `t` on the stack frame. Technically, the value of `t` (pointer address) will be different from one configuration to another and therefore execution would have to split. However, because `mergepoint()` checked that the string objects are deeply equal, the multiple pointer addresses can be treated as a single address and all instructions of `foo()` can be shared. After `foo()` finishes however, execution will have to split since each configuration's execution must return to the respective call site.

A.3 Next Steps for Tracejoins

Some of the next steps in customizable multiexecution research are as follows. The technique should be fully developed, addressing efficiency (e.g. minimizing comparisons once shared mode has started), and allowing multiple tracejoins. The technique should then be evaluated. A possibility is to find traces of existing test cases that are known to have similarity at points in execution that would require the expressiveness of tracejoins' pointcuts to enable multiexecution. From such traces, one could also try to automatically extract tracejoins by using aspect mining techniques across the traces. Such automated extraction would be especially useful for regression testing,

since tests have to be executed multiple times and it seems possible that shareable portions of the tests do not change over runs. Also, although tracejoins can theoretically serve as a unifying framework for customizable multiexecution techniques, actually using them to (re)implement existing techniques should make a stronger case for tracejoins.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA*, pages 345–364, 2005.

[3] Sven Apel and Dirk Beyer. Feature cohesion in software product lines: an exploratory study. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 421–430, New York, NY, USA, 2011. ACM.

[4] Sven Apel, Alexander von Rhein, Philipp Wendler, Armin Groblinger, and Dirk Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *ICSE*, 2013.

[5] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '12, pages 165–178, New York, NY, USA, 2012. ACM.

[6] Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. *SIGPLAN Not.*, 47(1):165–178, January 2012.

[7] D. Batory. Feature models, grammars, and propositional formulas. Technical Report TR-05-14, University of Texas at Austin, Texas, March 2005.

[8] Don Batory. Ahead tool suite. `http://www.cs.utexas.edu/users/schwartz/ATS.html`.

[9] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. Jts: Tools for implementing domain-specific languages. In *In Proceedings Fifth International Conference on Software Reuse*, pages 143–153. IEEE.

[10] Antonia Bertolino and Stefania Gnesi. Pluto: A test methodology for product families. In Frank van der Linden, editor, *PFE*, volume 3014 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2003.

[11] Eric Bodden. Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. In *ICSE 2010*. ACM Press.

[12] Eric Bodden. Clara: a framework for implementing hybrid typestate analyses. Technical Report Clara-2. Available from http://www.bodden.de/pubs/tr-clara-2.pdf, 2009.

[13] Eric Bodden. Private and Soot newsgroup correspondence, 2010.

[14] Eric Bodden, Feng Chen, and Grigore Rosu. Dependent advice: a general approach to optimizing history-based aspects. In *AOSD 2009*. ACM.

[15] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *SIGSOFT 2008/FSE-16*. ACM.

[16] Eric Bodden, Mira Mezini, Claus Brabrand, Társis Tolêdo, Márcio Ribeiro, and Paulo Borba. SPLlift - transparent and efficient reuse of IFDS-based static program analyses for software product lines. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, June 2013. To appear.

[17] Jan Bosch and Jaejoon Lee, editors. *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Jeju Island, South Korea, September 13-17, 2010. Proceedings*, volume 6287 of *Lecture Notes in Computer Science*. Springer, 2010.

[18] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA'02*, July 2002.

[19] Claus Brabrand, Márcio Ribeiro, Társis Tolêdo, and Paulo Borba. Intraprocedural dataflow analysis for software product lines. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, AOSD '12, pages 13–24, New York, NY, USA, 2012. ACM.

[20] Isis Cabral, Myra B. Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In Bosch and Lee [17], pages 241–255.

[21] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodík. Angelic debugging. In Taylor et al. [94], pages 121–130.

[22] Feng Chen and Grigore Roşu. MOP: an efficient and generic runtime verification framework. In *OOPSLA 2007*, pages 569–588. ACM Press.

[23] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, and Axel Legay. Symbolic model checking of software product lines. In Taylor et al. [94], pages 321–330.

[24] Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-Franois Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines (to appear). In *32nd International Conference on Software Engineering, ICSE 2010, May 2-8, 2010, Cape Town, South Africa, Proceedings*. IEEE, 2010. Acceptance rate: 13.7

[25] Curtis Clifton, Gary T. Leavens, and James Noble. MAO: Ownership and effects for more effective reasoning about aspects. In *ECOOP'07*.

[26] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*. ACM, 2006.

[27] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Interaction testing of highly-configurable systems in the presence of constraints. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 129–139, New York, NY, USA, 2007. ACM.

[28] Patrick Cousot and Radhia Cousot. Modular static program analysis. In *Proceedings of Compiler Construction*, pages 159–178. Springer-Verlag, 2002.

[29] Marcelo d'Amorim, Steven Lauterburg, and Darko Marinov. Delta execution for efficient state-space exploration of object-oriented programs. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 50–60, New York, NY, USA, 2007. ACM.

[30] Brett Daniel, Tihomir Gvero, and Darko Marinov. On test repair using symbolic execution. In Paolo Tonella and Alessandro Orso, editors, *ISSTA*, pages 207–218. ACM, 2010.

[31] Daniel S. Dantas and David Walker. Harmless advice. *SIGPLAN Not.*, 41(1):383–396, 2006.

[32] Erik Ernst, editor. *ECOOP 2007 - Object-Oriented Programming, 21st European Conference, Berlin, Germany, July 30 - August 3, 2007, Proceedings*, volume 4609 of *Lecture Notes in Computer Science*. Springer, 2007.

[33] FEST. FEST: Fixtures for Easy Software Testing. `http://fest.easytesting.org/`.

[34] FitNesse. FitNesse: The fully integrated standalone wiki and acceptance testing framework. `http://fitnesse.org/`.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[36] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *ASE'02*.

[37] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM.

[38] GraphStream. GraphStream: A Dynamic Graph Library. `http://graphstream-project.org/`.

[39] Alexander Gruler, Martin Leucker, and Kathrin Scheidemann. Modeling and model checking software product lines. In *FMOODS 2008*, pages 113–131. Springer-Verlag.

[40] Robert J. Hall. Fundamental nonmodularity in electronic mail. *Autom. Softw. Eng.*, 12(1):41–79, 2005.

[41] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *OOPSLA'01*.

[42] Gerald Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[43] Petr Hosek and Cristian Cadar. Safe Software Updates via Multi-version Execution. In *International Conference on Software Engineering (ICSE 2013)*, 2013.

[44] HTMLCleaner. HTMLCleaner. `http://htmlcleaner.sourceforge.net/`.

[45] Human-resource management system. 101Companies. `http://101companies.org/index.php/101companies:Project`.

[46] Mikolás Janota. Do sat solvers make good configurators? In Steffen Thiel and Klaus Pohl, editors, *SPLC (2)*, pages 191–195. Lero Int. Science Centre, University of Limerick, Ireland, 2008.

[47] Java tokenizer and parser tools. JTopas. `http://jtopas.sourceforge.net/jtopas/index.html`.

[48] Jikes RVM. Jikes research virtual machine. `http://jikesrvm.org/`.

[49] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990.

[50] Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *Automated Software Engineering (ASE)*, 2008.

[51] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*, FOSD '12, pages 1–8, New York, NY, USA, 2012. ACM.

[52] Shadi Abdul Khalek and Sarfraz Khurshid. Efficiently running test suites using abstract undo operations. In *ISSRE*, 2011.

[53] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP'01*.

[54] Chang Hwan Peter Kim. Reducing combinatorics in product line testing: Tool and results. Available from `http://userweb.cs.utexas.edu/~chpkim/spltesting`, 2010.

[55] Chang Hwan Peter Kim. Reducing Configurations to Monitor in a Software Product Line: Tool and Results. Available from `http://userweb.cs.utexas.edu/~chpkim/splmonitoring`, 2010.

[56] Chang Hwan Peter Kim. Shared execution for efficiently testing product lines: Evaluation. `http://www.cs.utexas.edu/~chpkim/sharedexecution`, 2012.

[57] Chang Hwan Peter Kim. Deferred execution for efficiently testing product lines: Evaluation. `http://www.cs.utexas.edu/~chpkim/deferredexecution`, 2013.

[58] Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Product Line Testing. In *AOSD*, 2011.

Available from http://userweb.cs.utexas.edu/~chpkim/chpkim-productline-testing.pdf.

[59] Chang Hwan Peter Kim, Eric Bodden, Don S. Batory, and Sarfraz Khurshid. Reducing configurations to monitor in a software product line. In Howard Barringer, Yliès Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon J. Pace, Grigore Rosu, Oleg Sokolsky, and Nikolai Tillmann, editors, *RV*, volume 6418 of *Lecture Notes in Computer Science*, pages 285–299. Springer, 2010.

[60] Chang Hwan Peter Kim, Sarfraz Khurshid, and Don S. Batory. Shared execution for efficiently testing product lines. In *ISSRE*, pages 221–230. IEEE, 2012.

[61] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, and Don Batory. SPLat: Evaluation. http://www.cs.utexas.edu/~chpkim/splat, 2013.

[62] Chang Hwan Peter Kim, Darko Marinov, Sarfraz Khurshid, Don Batory, Sabrina Souto, Paulo Barros, and Marcelo dAmorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *FSE*, 2013. Available from http://www.cs.utexas.edu/~chpkim/chpkim-splat.pdf.

[63] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Rozzle: De-cloaking internet malware. In *Oakland*, 2012.

[64] Korat home page. http://mir.cs.illinois.edu/korat/.

[65] Jeff Kramer. Conic: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1), 1983.

[66] Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in jit optimizations. In *Compiler Construction*, volume 3443 of *LNCS*, 2005.

[67] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[68] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. *SIGSOFT Softw. Eng. Notes*, 27(6):89–98, 2002.

[69] J. Jenny Li, Birgit Geppert, Frank Rößler, and David M. Weiss. Reuse execution traces to reduce testing of product lines. In *SPLC (2)*, pages 65–72. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007.

[70] Library for object persistence. *Prevayler*. `http://spl2go.cs.ovgu.de`.

[71] Library to serialize objects to XML and back again. XStream. `http://xstream.codehaus.org/`.

[72] Roberto E. Lopez-herrejon and Don Batory. A standard problem for evaluating product-line methodologies. In *Proc. 2001 Conf. Generative and Component-Based Software Eng*, pages 10–24. Springer, 2001.

[73] Robyn Lutz. Survey of product-line verification and validation techniques. Technical report, Jet Propulsion Laboratory, NASA, May 2007.

[74] John McGregor. Testing a Software Product Line. Technical Report CMU/SEI-2001-TR-022, CMU/SEI, March 2001. Available from `http://www.sei.cmu.edu/pub/documents/01.reports/pdf/01tr022.pdf`.

[75] Sowmiya Chocka Narayanan. Clustered test execution using java pathfinder. In *Master's Thesis. Department of Electrical and Computer Engineering. University of Texas at Austin*, 2010.

[76] Oracle. Java lesson: Exceptions. Document available at `http://download.oracle.com/javase/tutorial/essential/exceptions/`.

[77] Cristian Cadar Petr Hosek. Safe software updates via multi-version execution. In *International Conference on Software Engineering (ICSE 2013)*, pages 612–621, 5 2013.

[78] Malte Plath and Mark Ryan. Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84, 2001.

[79] Klaus Pohl and Andreas Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.

[80] Christian Prehofer. Semantic reasoning about feature composition via multiple aspect-weavings. In *GPCE'06*.

[81] Puzzle game. *Sudoku*. `https://code.launchpad.net/~spl-devel/spl/default-branch`.

[82] Alexander Von Rhein, Sven Apel, and Franco Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *JPF Workshop*, 2011.

[83] RIACS/NASA Ames Research Center. Java PathFinder. `http://javapathfinder.sourceforge.net/`.

[84] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22, 1996.

[85] Sable Group. Soot: a Java optimization framework. http://www.sable.mcgill.ca/soot/.

[86] SAT4J. SAT4J. http://www.sat4j.org/.

[87] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[88] David Schuler, Valentin Dallmeier, and Andreas Zeller. Efficient mutation testing by checking invariant violations. In Gregg Rothermel and Laura K. Dillon, editors, *ISSTA*, pages 69–80. ACM, 2009.

[89] SIR. SIR: Software-artifact Infrastructure Repository. http://sir.unl.edu/portal/index.php.

[90] Gregor Snelting and Frank Tip. Semantics-based composition of class hierarchies. In *ECOOP'02*.

[91] Manu Sridharan, Stephen J. Fink, and Rastislav Bodík. Thin slicing. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 112–122. ACM, 2007.

[92] Vanessa Stricker, Andreas Metzger, and Klaus Pohl. Avoiding redundant testing in application engineering. In Bosch and Lee [17], pages 226–240.

[93] SuperCSV. SuperCSV. http://supercsv.sourceforge.net/.

[94] Richard N. Taylor, Harald Gall, and Nenad Medvidovic, editors. *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 2011.

[95] Sahil Thaker, Don S. Batory, David Kitchin, and William R. Cook. Safe composition of product lines. In Charles Consel and Julia L. Lawall, editors, *GPCE*, pages 95–104. ACM, 2007.

[96] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. Technical Report MSR-TR-2005-153, Microsoft Research, Redmond, Washington, November 2005.

[97] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validation with delta execution. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[98] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don S. Batory. Testing software product lines using incremental test generation. In *ISSRE'08*.

[99] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. of the 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[100] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[101] E.J. Weyuker and T.J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, 1980.

[102] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE'04*.

Vita

Chang Hwan Peter Kim earned a Bachelor of Applied Science in Computer Engineering from the University of Waterloo, Canada in June 2005. He earned a Master of Applied Science in Electrical and Computer Engineering from the same university under the supervision of Professor Krzysztof Czarnecki in June 2006. He earned a PhD in Computer Science from the University of Texas at Austin, USA under the supervision of Professor Don Batory and Professor Sarfraz Khurshid (defended in September 2013 and will have received degree in December 2013). In October 2013, he became a postdoctoral research assistant in the Department of Computer Science at the University of Oxford, UK under the supervision of Professor Marta Kwiatkowska.

Permanent Address: Toronto, Ontario, Canada

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.