to appear, AOSD 2011

Reducing Combinatorics in Testing Product Lines

Chang Hwan Peter Kim
University of Texas at Austin
Austin, TX 78712 USA
chpkim@cs.utexas.edu

Don Batory
University of Texas at Austin
Austin, TX 78712 USA
batory@cs.utexas.edu

Sarfraz Khurshid
University of Texas at Austin
Austin, TX 78712 USA
khurshid@ece.utexas.edu

ABSTRACT

A *Software Product Line (SPL)* is a family of programs where each program is defined by a unique combination of *features*. Testing or checking properties of an SPL is hard as it may require the examination of a combinatorial number of programs. In reality, however, features are often *irrelevant* for a given test — they augment, but do not change, existing behavior, making many feature combinations unnecessary as far as testing is concerned. In this paper we show how to reduce the amount of effort in testing an SPL. We represent an SPL in a form where conventional static program analysis techniques can be applied to find irrelevant features for a test. We use this information to reduce the combinatorial number of SPL programs to examine.

1. INTRODUCTION

A *Software Product Line (SPL)* is a family of programs where each program is defined by a unique combination of features. By developing a set of programs with commonalities and variabilities in a systematic way, SPLs can significantly reduce both the time and cost of software development. But at the same time, SPLs require software engineering techniques distinct from those for conventional programs. In particular, testing, the phase to which the majority of software development is dedicated, becomes especially challenging [26].

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 focusses on feature combinations that are believed to have a higher chance of falsifying certain properties [10][11][25]. In light of no other information, this is reasonable but critical combinations may be overlooked. Another approach is to apply traditional verification techniques directly – model checking [16][35] or bounded exhaustive testing [7][37] – 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.

Given this dismal situation, it is still possible to improve 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 paper, 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. In a poster paper [19], we introduced the notion of analyzing features to reduce the effort of testing a product line. This paper expands the poster paper substantially by making the following new 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*[30], a popular static analysis framework for Java, and *SAT4J*[31], an off-the-shelf SAT solver.
- **Evaluation.** We demonstrate the effectiveness of our technique on concrete product lines and tests.

2. MOTIVATING EXAMPLE

Product Line. Suppose that we have the product line in Figure 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 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.

Unlike a conventional module such as a class, a feature module encapsulates semantically-related code that is scattered and tangled throughout a program. Also, a feature module is a part of a program if and only if the corresponding feature is selected, namely, blue code is present iff `Loyalty=true` (code annotated with `LOYALTY` like `points` declaration will disappear if the feature is `false`). In general a product line can be viewed as a template that can instantiate up to 2^n distinct programs, where `n` is the number of optional features.

Feature Model. A *feature model* defines the legal *feature combinations* or *configurations* (we use these terms interchangeably). For our example, the feature model is shown below as a context-sensitive grammar. It requires `Base` to be present in every program (only bracketed features are optional) and requires one of the other three features, yielding a total of 7 distinct programs:

```
ProductLine :: [Ceiling] [Fee] [Loyalty] Base;        // grammar
Ceiling or Fee or Loyalty;                          // constraints
```

Product Line Tests. A *product line test* is a program with a `main` method that executes some methods and references some code of the product line. Figure 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

```
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   PremiumAccount(){
34     money = 100;
35   }
36
37   @LOYALTY
38   int overdraftPenalty() {
39     int p = abs(money)*0.01;
40     if(CEILING)
41       return min(p, 50);
42     return p;
43   }
44 }
```

Figure 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 2: Product Line Tests

because only these features' code 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 shows an overview of our technique for reducing combinatorics in product line testing. We start with a product line that is encoded as a SysGen program (Section 3), 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 4). We then feed the specialized feature model, the SysGen program and the test to a static analysis that identifies the relevant features (Section 5). Given the relevant features and specialized feature model, a solver determines the configurations against which the test must be run (Section 6). We begin by explaining SysGen programs.

3. SYSGEN PROGRAMS

A *system generation (SysGen)* program encodes a product line as a single program using `#ifdef <feature>` declarations. At compilation time, variables denoting features are assigned boolean values and a particular program of the product line is produced. Our SysGen programs, including the running example, are slightly different: we use `if(feature)`-conditionals and annotations rather than `#ifdef <feature>` declarations, and feature variables only appear in these conditionals and annotations (these are the underlined statements and annotations in Figure 1). The benefit in doing so is that it enables off-the-shelf program analyses for conventional (Java) programs to be applied to product lines.

To instantiate a program from a SysGen representation, we simply set each *feature variable* (e.g. `LOYALTY`, `FEE`, and `CEILING`) to a boolean value that indicates the presence or absence of that feature. Nothing further needs to be done as the statements that are guarded by feature variables will be executable or not executable depending on these values. Classes, methods, and fields that are annotated with feature variables whose value is `false` are physically removed. Note that colors are not part of the SysGen representation, but we retain them in this paper for visual aid.[1]

4. RELEVANT FEATURES

A *relevant* feature is a feature for which we need to consider both `true` and `false` values when running a test. As we explain in Section 4.2, a feature is considered to be relevant depending on whether its code can influence the test outcome. We use the SysGen program, feature model and test to reduce the set of features whose code needs to be analyzed. We describe how to do this next.

[1]Not all product line variations may be easily represented using variability mechanisms of a conventional programming language. For example, pushing multiple alternative features 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 SysGen programs, albeit with some refactoring.

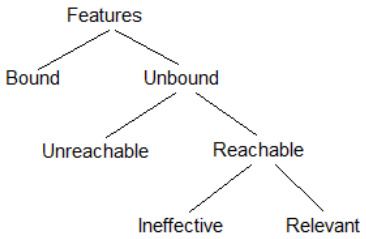

Figure 4: Classification of Features

4.1 Pruning Features

A *bound* feature has its truth value fixed for a given test. Bound features are determined by adding implementation constraints according to [33] to the feature model to ensure that the test will compile. Also, 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). Both implementation and test 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 [3] and using a SAT solver to propagate constraints [17]. *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 analysis presented in Section 5 determines which features are reachable and only checks these features for relevance.

Figure 4 shows our classification of features. We use the term *ineffective* to describe reachable features that are not relevant. 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 4.2 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 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

[2]Unreachable features' code may also be relevant if the test uses reflection. See Section 8.1. Also, note that bound features may actually be reachable as well but we just do not label them as such.

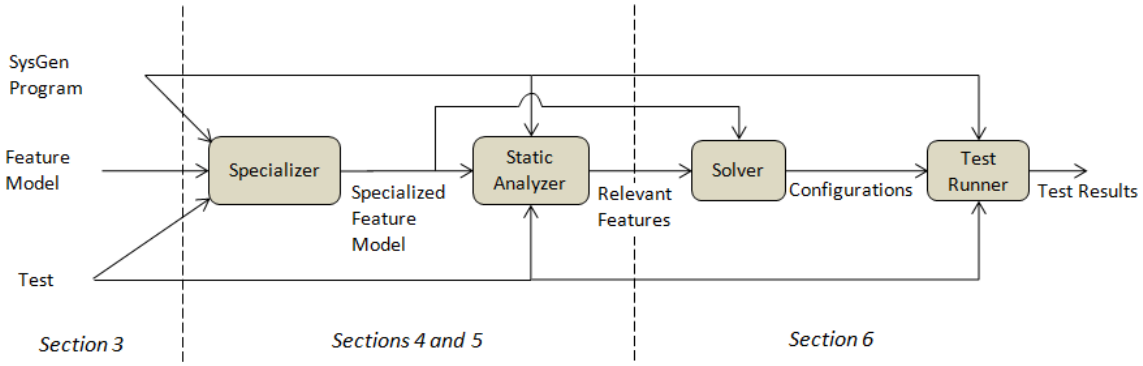

Figure 3: Overview of Our Technique

`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.

4.2 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 5, 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 the test.
- For `Test2`, both of the reachable features, `Loyalty` and `Ceiling`, are relevant as the former changes the interprocedural 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.

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

5. STATIC ANALYSIS

Using an off-the-shelf inter-procedural context-insensitive and flow-insensitive points-to analysis called *Spark* [21], 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 5.1) and modifications (Section 5.2). The check for indirect effects (Section 5.3) 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.

5.1 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()`.

5.2 Modifications

A *modification* adds a contiguous block of statements to an existing method. Modifications of SysGen programs are

always enclosed by if-conditions of feature variables, such as lines 12-13 and 40-41 of Figure 1. Our static analysis for modifications was inspired by a similar analysis for aspects [9] that checks for data-flow and control-flow effects. Section 5.2.1 presents the control-flow check and Section 5.2.2 presents the data-flow check.

5.2.1 Control-Flow Check

The only way a modification does not preserve a CFG as described in Section 4.2 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.overdraftPenalty()`, `Ceiling`'s modification (line 41) 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.overdraftPenalty()`. Also, `Ceiling` is relevant to `Test3` because line 26, reachable through line 13, changes the control-flow of `interest(double)`.

5.2.2 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 SysGen program, 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 [20] 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 8.2 that a highly precise static analysis is not necessary for our problem.

5.3 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 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 [33] 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 5: An Example Illustrating Indirect Effect

Indirect Control-Flow Effect. `C` is relevant in Figure 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 $\texttt{C} \implies \texttt{B}$ 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`.

6. 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 6, relies on the SAT4J [31] SAT solver, which can enumerate solutions to a propositional formula. Our algorithm iterates through each possible combination of the relevant features and treats

```
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 6: Algorithm to Find Test Configurations

Table 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

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* [31] 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, goes through each configuration, creating a concrete program corresponding to the configuration from the SysGen program and running the test against that program.

Examples. Table 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 5, 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.

7. 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 [23]; *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 [2].

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 paper 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 [4] and for the purpose of this paper, we developed a Jak-to-SysGen translator to convert them into the SysGen representation. Our plugin, the examined product lines and tests, as well as the detailed evaluation results are available for download [18].

7.1 Graph Product Line (GPL)

Table 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. 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 are not considered to be relevant (see Section 8.1 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]A simple variation of our algorithm terminates after collecting k configurations, in case there is a huge number of configurations to test.

Table 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.)

Table 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.)

7.2 Notepad

Table 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 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, `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.

7.3 jak2java

Table 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` 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.

Table 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	0
Relevant features	0
Configurations with unbound features	5
Configurations to test	1
Duration of static analysis	100 sec. (1.66 min.)

ArgInquireTest. This test fares better because the method being called, `argInquire`, does not reach a large portion of the product line, taking 1.6 minutes to determine that only 1 configuration needs to be tested.

8. DISCUSSION

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

8.1 Assumptions and Limitations

Off-the-shelf program analyses have well-known limitations. Indeed, the first three assumptions below are not unique to our work but the last assumption is.

- **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 [15], 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.

- **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* [4] 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 `SysGen` representation using a translator. It would not require much effort to remove this limitation.

8.2 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 cannot be done 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.

8.3 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.

8.4 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 synthesized, 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.

8.5 Perspective

Initially, our belief was that existing analyses for conventional programs could be directly applied to a SysGen program. However, we discovered that analyzing a SysGen program 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 [6]. 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.

9. RELATED WORK

9.1 Product Line Testing and Verification

There is a considerable amount of research in product line testing and verification (see [24] for a survey). We discuss research most closely related to ours.

Model-Checking. Classen et al.[8] 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 like SysGen, 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. Sampling exploits domain knowledge, rather than program analysis results, to select configurations to

test. An SPL tester may choose a set of features for which all combinations must be examined, while for other features, only t-way (most commonly 2-way) interactions are tested [10][11][25]. Sampling approaches can miss problemmatic 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 [34]. There is also research on constructing a product line of tests so that they may be reused [5][27]. We address the different problem of minimizing test execution for a single given test.

9.2 Program Slicing

Determining relevant features is closely related to *backwards program slicing* [36], 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" [36] 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.

9.3 Feature Interactions

There is a large body of work on detecting feature interactions using static analysis [28][32][13][9][22], of which *harmless advice* [13] and *Modular Aspects with Ownership (MAO)* [9] 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.

9.4 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 [12] and verification [14][22].

9.5 Reducing Testing Effort

Reducing testing effort for a single program, typically using output from some analysis, has a long history. [29] identifies a subset of existing tests to run given a program change. We address a complementary problem, namely to identify a subset of existing features that are relevant for a given test in a product line.

10. CONCLUSIONS

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 our paper.

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 folk-tale 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.

Acknowledgements. Kim is supported by an NSERC Postgraduate Scholarship. Kim and Batory are supported by NSF's Science of Design Project #CCF-0724979. Khurshid is supported by NSF #CCF-0845628.

11. REFERENCES

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

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

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

[4] D. Batory, B. Lofaso, and Y. Smaragdakis. Jts: Tools for implementing domain-specific languages. In *In Proceedings Fifth International Conference on Software Reuse*, pages 143–153. IEEE.

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

[6] E. Bodden. Private and Soot newsgroup correspondence, 2010.

[7] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In

ISSTA'02, July 2002.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. 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

[9] C. Clifton, G. T. Leavens, and J. Noble. MAO: Ownership and effects for more effective reasoning about aspects. In *ECOOP'07*.

[10] M. B. Cohen, M. B. Dwyer, and J. 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.

[11] M. B. Cohen, M. B. Dwyer, and J. 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.

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

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

[14] D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *ASE'02*.

[15] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *OOPSLA'01*.

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

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

[18] C. H. P. Kim. Reducing combinatorics in product line testing: Tool and results. Available from `http://userweb.cs.utexas.edu/~chpkim/spltesting`, 2010.

[19] C. H. P. Kim, D. Batory, and S. Khurshid. Eliminating Products to Test in a Software Product Line. In *ASE2010 (Tentatively Accepted Poster Session Paper)*. Available from `http://userweb.cs.utexas.edu/~chpkim/chpkim-ase10-short.pdf`.

[20] A. Le, O. Lhoták, and L. Hendren. Using inter-procedural side-effect information in jit optimizations. In *Compiler Construction*, volume 3443 of *LNCS*, 2005.

[21] O. Lhoták and L. 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.

[22] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. *SIGSOFT Softw. Eng. Notes*, 27(6):89–98, 2002.

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

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

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

[26] C. Nebut, Y. L. Traon, and J.-M. Jézéquel. System testing of product lines: From requirements to test cases. In *Software Product Lines*, pages 447–478. Springer-Verlag, 2006.

[27] K. Pohl and A. Metzger. Software product line testing. *Commun. ACM*, 49(12):78–81, 2006.

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

[29] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22, 1996.

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

[31] SAT4J. SAT4J. `http://www.sat4j.org/`.

[32] G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In *ECOOP'02*.

[33] S. Thaker, D. S. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In C. Consel and J. L. Lawall, editors, *GPCE*, pages 95–104. ACM, 2007.

[34] E. Uzuncaova, D. Garcia, S. Khurshid, and D. S. Batory. Testing software product lines using incremental test generation. In *ISSRE'08*.

[35] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of the 15th Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

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

[37] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE'04*.