Reducing Combinatorics in Testing Product Lines

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

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

Sarfraz Khurshid
University of Texas-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 all programs in the SPL, whose cardinality is exponential in the number of features. In reality, however, features are often *behavior-irrelevant* for a given test (i.e., they augment, but do not change, existing behavior), making many combinations redundant as far as testing is concerned. In this paper we show how to reduce the amount of effort in testing an SPL. We transform an SPL into a form where conventional static program analysis techniques can be directly used to find behavior-irrelevant features for a test. We use this information to reduce the combinatorial number of 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. In recognition of these benefits, research has focused on requirements [4][6][8] and development/synthesis [5][17][21]. In contrast, there is a comparative scarcity of work in testing SPLs [29][37], the phase to which a majority of software development is dedicated.

There are many challenges in testing or checking the properties of programs in an SPL. The most obvious is the sheer number: an SPL with 20 optional features has over a million (2^{20}) 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 test if the output String always conforms to a particular pattern, we need to test every possible feature combination.

Current practice often focusses on feature combinations that are believed to have a higher chance of falsifying certain properties [11][29]. In light of no other information, this approach is reasonable but incomplete. Another solution is to apply traditional verification techniques directly – model checking [16][38] or bounded exhaustive testing [9][41] – on every product of the SPL. While complete, feature combinatorics render brute force impractical. Yet another complicating factor is that features often have no formal specifications. Even contracts are usually unavailable.

Given this dismal situation, it is still possible to improve the state-of-the-art by leveraging the semantics of *features* — increments in functionality. In our experience, we noticed that features add, but do not remove, code [3]. We hypothesize that this extends to run-time, namely, that features are *behavior-irrelevant* for a given test, i.e., they augment, but do not invalidate, existing behavior. To illustrate potential benefits, suppose we determine that 18 of the 20 features in the above example do not modify the output String and thus are behavior-irrelevant. We can confidently run the String output test on only $2^2 = 4$ programs to analyze the entire product line, as opposed to over a million programs.

In this paper, we explore the concept of behavior-irrelevant features to reduce SPL testing. For a given test, we find features that do not alter the result of the test (these features are behavior-irrelevant). We accomplish this by transforming an SPL into a form where conventional program analysis techniques can be directly applied, determine the features that are behavior-irrelevant for the given test, and prune the space of such features to significantly reduce the number of SPL programs to examine for that test.

We make the following contributions:

- A technique that enables off-the-shelf tools to analyze all programs in an SPL.
- A technique to reduce the configurations to be examined for a given test.

2. MOTIVATING EXAMPLE

A micro-blogging site, such as Twitter and Facebook, allows users to post text status updates (e.g., how they are feeling and what they are doing now) for others to see [40]. Suppose that we have an SPL of micro-blogging sites, with the feature model given by the following grammar [4]:

```
MicroBlog ::  [Backup] [Censor] [Style] Base;
```

The grammar requires `Base` to be present in every program of the SPL, while all other features (`Backup`, `Censor`, `Style`) are optional, yielding a total of 8 distinct programs.

The feature modules of this SPL are shown in Figures 1-4 and two tests are given in Figure 5. These modules are written in the Jak language (although slightly different syntax would express these modules in Jx [31], Classbox/J [7], and FeatureC++ [2] languages). The distinguishing characteristic of these languages is that a feature module can add new classes to a program, add new members to existing classes, and can wrap existing methods. Wrapping in Jak is identical to method overrides in subclassing and is indicated by `Super` keyword (e.g., line 4 of Figure 2).[1]

The `Base` feature (Figure 1) defines the `Status` class, which represents the status (simply a wrapper for some text) of a single user, and the `Checker` class, which provides an interface for other features to implement in order to examine and update a given `Status` object. `Base` allows other features to refine or extend method `display()` and call `applyChecker(Checker)` with their own `Checker`, whose call-back method (line 22) examines the current status and may update the `valid` variable (line 17). The annotation `@Modifiable` on the field declaration of `valid` (line 4) will be explained later, in Section 5. Note that `Screen` class holds an internal variable `content` that represents the displayed content, which is modified by `print(..)` and read by `isEmpty()`.

```
 1 class Status {
 2   String text;
 3
 4   @Modifiable
 5   boolean valid = true;
 6
 7   void setText(String t){
 8     text = t;
 9     valid = true;
10   }
11
12   void display() {
13     Screen.print(text + ": " + valid);
14   }
15
16   void applyChecker(Checker c) {
17     valid = valid && c.check(this);
18   }
19 }
20
21 class Checker {
22   boolean check(Status s) { return true; }
23 }
24
25 class Screen {
26   Object content;
27
28   static void print(String t) { // writes t into content}
29   static boolean isEmpty() { // reads from content }
30 }
```

Figure 1: Base Feature

The `Censor` feature refines `display()` (lines 2-5 of Figure 2) by replacing an inappropriate word with a sequence of asterisks (line 10). The check always passes. Note that `refines` differs from `extends` in that the former transforms the original class, while the latter defines a new subclass that extends the original class.

The `Style` feature refines `display()` (lines 2-7 of Figure 3) and its `check(..)` method returns `true` if each word in the status text begins with an alphanumeric character (line 12, assume such a method exists). Note that the original code

```
 1 refines class Status {
 2   void display() {
 3     applyChecker(new CensorChecker());
 4     Super.display();
 5   }
 6 }
 7
 8 class CensorChecker extends Checker {
 9   boolean check(Status s) {
10     s.text = s.text.replace("darn", "****");
11     return true;
12   }
13 }
```

Figure 2: Censor Feature

```
 1 refines class Status {
 2   void display() {
 3     applyChecker(new StyleChecker());
 4     if(valid) {
 5       Super.display();
 6     }
 7   }
 8 }
 9
10 class StyleChecker extends Checker {
11   boolean check(Status s) {
12     return !s.text.startsAlphanumerically();
13   }
14 }
```

Figure 3: Style Feature

is executed only if the style check and previous checks pass (lines 4-5).

Lastly, the `Backup` feature adds its own field `prev` (line 2 of Figure 4), extending `setText(String)` to remember the previous value (line 5), and adds its own method `restore()` that restores the previous value (line 9).

Figure 5 shows two tests. We adopt a broad definition: an *SPL test*, or *test* for short, is a program with a `main` method that executes the methods and references classes and class members introduced by one or more SPL features. (As we will be using static analysis, the inputs to this test are irrelevant). Although the tests in Figure 5 exercise the functionalities of `Base` and `Style` features, we can in fact write a test fairly arbitrarily, such as bundling these two tests into one.

An SPL test evaluates one or more properties of a product line. A feature can alter properties of a product line. If the feature does not alter the properties that the SPL test evaluates, that feature can safely be ignored when running the test. *Determining whether a feature is relevant to a given test is the central problem.*

Given an SPL test, all features whose code that the test

```
 1 refines class Status {
 2   String prev;
 3
 4   void setText(String t) {
 5     prev = text;
 6     Super.setText(t);
 7   }
 8
 9   void restore() { text = prev; }
10 }
```

Figure 4: Backup Feature

[1]AOP offers a much broader set of extensions [19], but is concommitantly harder to analyze.

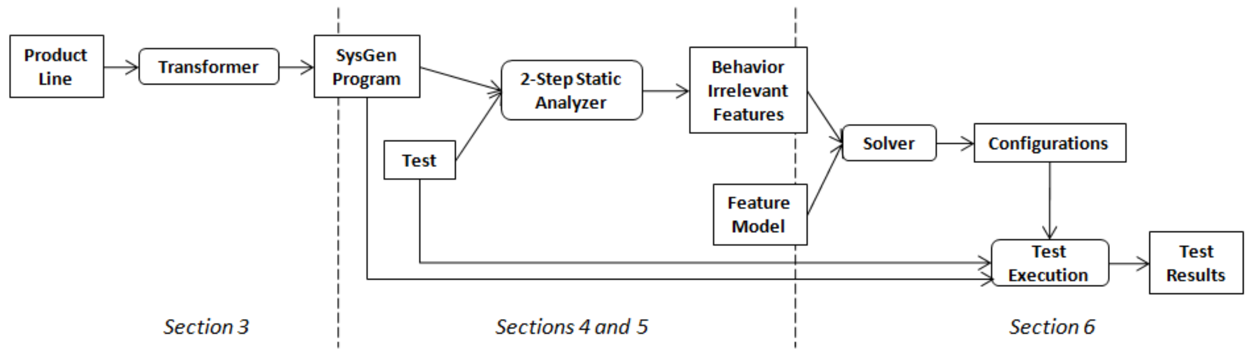

Figure 6: Overview of Our Technique

```
1  @FeaturesToTest({"Base"})
2  class BaseTest {
3    static void main(String args){
4      Status s = new Status();
5      s.setText("darn");
6      s.display();
7    }
8  }
9
10 @FeaturesToTest({"Style"})
11 class StyleTest {
12   static void main(String args)  {
13     StyleChecker sc = new StyleChecker();
14     Screen.print(sc.check("potato*") == true);
15     Screen.print(sc.check("!hello!") == false);
16   }
17 }
```

Figure 5: Test Classes

transitively references (transitively) needs to be present as the test program will otherwise not compile. In addition, an SPL test can be intended for a particular set of features (e.g., `@FeaturesToTest` annotation in line 10 indicates that `StyleTest` is intended for `Style` feature). Both the referenced features and the intended features are called *required* features. Besides required features, there are optional features whose inclusions into a program can yield different test outcomes. For example, for `BaseTest`, without understanding in detail how each feature works and how each feature interacts with others, we would have to run it on all eight combinations of `Censor`, `Style`, and `Backup` (`Base` is mandatory). In the worst case, an SPL with t tests and n optional features, we would have to execute $t \cdot 2^n$ test programs! We can do better than this.

Figure 6 displays a roadmap for the next sections. Rectangles are inputs/outputs and ovals are functions. Given a product line, **Transformer** maps it into a **SysGen Program** (Section 3) which enables us to use off-the-shelf program analysis techniques, rather than having to develop techniques specific to product lines. To reduce the feature combinations or SPL programs on which to run a test, we first run the **2-Step Static Analyzer** on the SysGen program with respect to that test, which yields a set of **Behavior-Irrelevant Features** that are inconsequential to the test (Section 4 and Section 5). Given a feature model of the SPL (which defines all legal combinations of features) and our knowledge of behavior-irrelevant features, we can identify the set of **Configurations** (SPL programs) that must be examined (Section 6).

3. THE SYSGEN PROGRAM OF AN SPL

Checking the behavior of an SPL is not just computationally difficult, but is also technically difficult because we are dealing with unconventional modules, i.e. partial programs or *feature modules*. Although it may be possible to develop analysis techniques specific to feature modules, we instead transform an SPL into an ordinary program automatically so that we can use standard program analysis tools. The idea of representing a product line as an ordinary program is not new: programmers have been creating product lines using *system generation (SysGen)* techniques with `#ifdef FEATURE` preprocessor directives for decades [18]. We do the same, but in a different way and for a different purpose.

Given a product line with a Jak-defined set of features, we (our tool) wrap each method refinement with `if(FEATURE)`, to turn the feature's code on or off dynamically, rather than statically with `#ifdef`. We also annotate each declaration with the name of the feature that introduced it. We then merge the code of all features into a single program, replacing (or inlining) each `Super` call with the body of the actual method being refined. If multiple features refine the same method (e.g. `Censor` in Figure 2 and `Style` in Figure 3 both refine `Status.display()`), we rely on standard feature model semantics in Jak-built programs that the order in which features are composed is defined by the feature model [3]. We refer to the single program produced as the *SysGen program* of the SPL.

Figure 7 is the SysGen program of our *MicroBlog* product line. Lines 15-16 show how the `Backup` method refinement is represented, lines 25-26 show `Censor` method refinement, and lines 27-33 show `Style` method refinement. Lines 10, 20, and 48 show declarations that are annotated with their features.

With an SPL in SysGen form, testing all the feature combinations of the SPL reduces to running the test with different combinations of values of feature configuration variables. For example, `BaseTest` is run eight times, on every combination of values of `Config.BASE`, `Config.CENSOR`, `Config.STYLE`, and `Config.BACKUP`, with `Config.BASE=true`. The screen is empty, for instance, when all features are present. Of course, this enumeration can be accomplished by a simple script that is inserted into the test's `main` method. But the problem is, as we noted earlier, that many tests will be redundant. A static analysis, which we develop in the

```
@Feature("Base")
class Status {
  @Feature("Base")
  String text;

  @Modifiable
  @Feature("Base")
  boolean valid = true;

  @Feature("Backup")
  String prev;

  @Feature("Base")
  void setText(String t){
    if(Config.BACKUP)
      prev = text;
    text = t;
  }

  @Feature("Backup")
  void restore() { text = prev; }

  @Feature("Base")
  void display() {
    if(Config.CENSOR)
      applyChecker(new CensorChecker());
    if(Config.STYLE) {
      applyChecker(new StyleChecker());
      if(valid)
        Screen.print(text + ":" + valid);
    }
    else
      Screen.print(text + ":" + valid);
  }

  @Feature("Base")
  void applyChecker(Checker c) {
    valid = valid && c.check(this);
  }
}

@Feature("Base")
class Checker {
  @Feature("Base")
  boolean check(Status status) { return true; }
}

@Feature("Censor")
class CensorChecker extends Checker {
  @Feature("Censor")
  boolean check(Status s) {
    s.text = s.text.replace("darn", "****");
    return true;
  }
}

@Feature("Style")
class StyleChecker extends Checker {
  @Feature("Style")
  public boolean check(Status s) {
    return !s.text.startsAlphanumerically();
  }
}

@Feature("Base")
class Screen {...}
```

Figure 7: SysGen Program for the *MicroBlog* SPL

next sections, exploits the SysGen form and identifies such redundancies.

A SysGen program may have limits; not in the case studies that we examine later, but in general. 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. The method and code that are common to the alternative features could be factored into a separate feature, which the alternative features refine. (In fact, we do exactly this in one of the case studies in Section 7). For now, we assume that product lines can be represented with this approach, albeit with some refactoring effort.

4. CONDITIONS FOR BEHAVIOR-IRRELEVANCE

Given a test, we need to run it only on combinations of features that can change the outcome of the test. The only way a feature can change the test's outcome is to alter the test's data-flow or control-flow.[2] We use *Def-Use (DU) pairs*, a variation of *DU-chains* [1] traditionally used for compiler optimizations, to define behavior-irrelevance.

A DU-pair is simply a variable assignment and a use of the variable that is reachable from that assignment without an intervening assignment to that variable. For example, Figure 8(a) shows the *control-flow graph (CFG)* and DU-pairs of `BaseTest` (Figure 5) with just the `Base` feature present.[3]

DU-pairs 2 and 3 show that the two variables being printed are defined in `Status.setText(String)`. DU-pair 1 shows that the text originates from an argument to the set method invocation. DU-pairs connect to form a dataflow graph called a *DU-graph*.

There are two ways of changing a DU-graph: 1) by overriding or *killing* the definition of a DU-pair in the graph by inserting an intervening assignment and 2) by changing the condition under which a DU-pair in the graph exists, which we call a *presence condition*. A behavior-irrelevant feature neither kills existing DU-pairs nor alters existing presence conditions. Consider the following examples.

- `Censor` is not behavior-irrelevant because it kills definitions of DU-pairs in `Base`. Figure 8(b) shows `Censor` composed with `Base` w.r.t. `BaseTest` (changes to `Base` are shown in red). `Censor` refines the `Status.display()` method and runs its own checker, which writes to `text` and `valid`, killing the definitions in the DU-pairs 2 and 3 and replacing the pairs with DU-pairs 4 and 5 respectively.

- `Style` is also not behavior-irrelevant . It changes presence conditions of DU-pairs in `Base`. Figure 8(c) shows `Style` composed with `Base` w.r.t. `BaseTest`. `Style` runs its check and calls the original instruction, which prints `text` and `valid`, only if `valid` is `true`. Like `Censor`, this feature kills DU-pair 3, but also conditionalizes DU-pair 2, causing it to exist only when `valid=true`.

[2]In this paper, we assume that a feature does not alter control-flow by throwing exceptions.
[3]Some DU-pairs, such as that involving the `Status` object creation and the dereference in the next statement that invokes `Status.setText(..)`, are not shown as they are not relevant to the example.

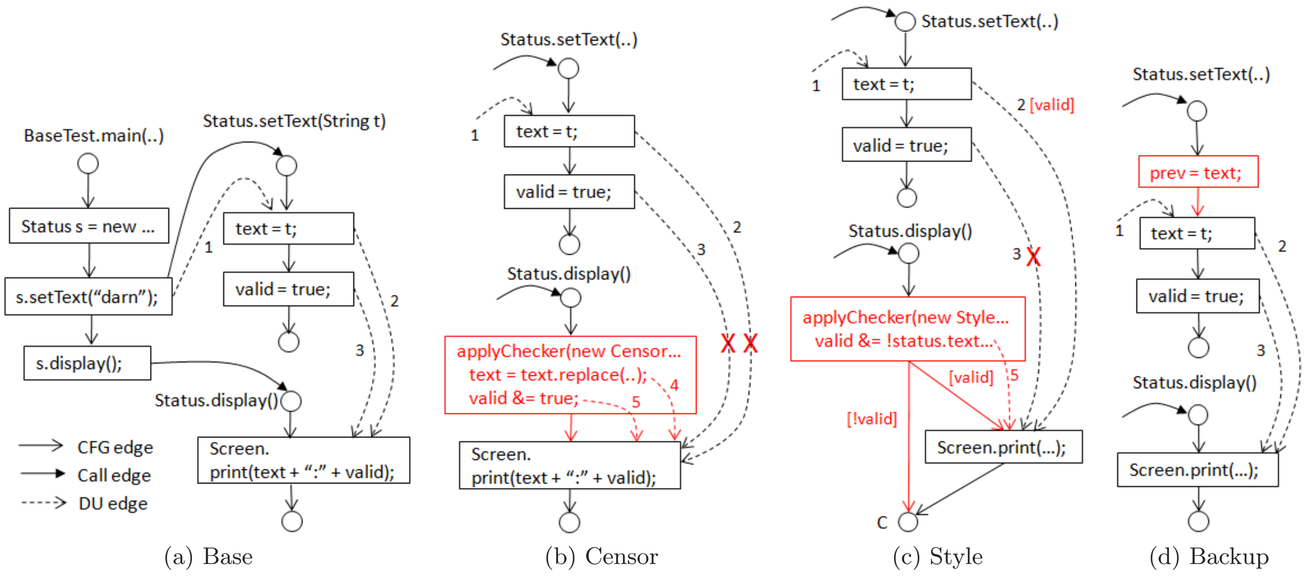

Figure 8: DU-Graph of `BaseTest` with Different Feature Combinations

- `Backup` is behavior-irrelevant as it neither kills definitions of DU-pairs nor changes presence conditions of DU-pairs. Figure 8(d) shows `Backup` composed with `Base` w.r.t. `BaseTest`. The new code writes `text` to `prev`, which cannot affect any existing DU-pair because `prev` did not exist before `Backup` was added.

A feature can both introduce new members (classes, fields, and method) to a program and refine existing methods. Adding a new member can alter program behavior if it a) overrides existing members [26, 36] (e.g. replacing an existing method with an empty method) or b) it is referenced through reflection (e.g. adding a new field to a class changes the outcome of iterating over members of the class) [15]. There is a simple, conservative static analysis that we have written previously to address problem a) [26]. b) can be addressed through a static analysis that checks for reflection. For the purpose of this paper, we assume that these two problems do not occur or that they have already been reported.

With these assumptions, features that only add new members are behavior-irrelevant. Such features are common in *layered-designs*, where features increment behavior by using or referencing code added by previously defined features. For example, the base feature may define data, the subsequent feature may define operations on the data, and the following feature may instantiate data and invoke their operations. We encounter layered-design features in Section 7.

5. TWO-STEP STATIC ANALYSIS

To check if a feature is behavior-irrelevant we must check if it a) alters presence conditions or b) kills the definition of a DU-pair in the DU-graph of a test. As detecting changes to a DU-graph for large OO-programs is expensive, we conservatively approximate that a feature is behavior-irrelevant by statically checking that it 1) only adds new code to existing basic blocks of the CFG, which guarantees a) doesn't occur, and 2) only writes to variables that it created and could not have existed before the feature was added, which guarantees b) doesn't occur. Section 5.1 explains 1) and Section 5.2 explains 2). This two-step analysis is based on the analysis in [10], but the two are considerably different, as will be explained in Section 8. We implemented the analysis using the Soot framework [35].

5.1 Control-Effect Analysis

Before we create a SysGen program, we look at each method refinement in each feature module and see if the following two conditions hold: 1) the original method, i.e. `Super.methodName(..)`, is called precisely once and in every path between the first and the last statement of the method refinement inclusively and 2) the original *parameters* are passed to the original method and the original return value is returned. Satisfying these two conditions guarantees that a feature preserves control-flow through the original method while adding new instructions to the original method. The feature's change is analogous to adding instructions to basic blocks of a CFG. Here are three examples:

- `Censor`'s method refinement (Figure 2, lines 2-5) satisfies Condition 1 as `Super.display()` is called precisely once and in every path of the method refinement. Condition 2 is not relevant as there are no parameters.

- `Style`'s method refinement (Figure 3, lines 2-7) fails to satisfy Condition 1 as `Super.display()` is called conditionally.

- `Backup`'s method refinement (Figure 4, lines 4-7) satisfies Condition 1 as `Super.setText(t)` is called precisely once. Condition 2 is also satisfied because the original parameter, `t`, is passed to the `Super` call.

  Note: we insist on the original *parameter* to be passed unchanged to the `Super` call to simplify analysis. If `t` were assigned to a local variable and that local variable were passed to the `Super` call, our analysis would conservatively report that Condition 2 failed.

Let F be the feature that introduces method m. Let R be a feature that refines m. If the control-effect conditions 1) and 2) do not hold for R, then R is behavior-relevant. Only `Style`, which conditionalizes a `Base` method, is behavior-relevant in this sense. If R is in the control-flow of another method-refining feature S, S is considered to be behavior-relevant as well to allow R to be reached. This is elaborated in a technical report [20].

5.2 Heap-Effect Analysis

In addition to checking that a feature preserves control flow, we check that a feature preserves variable values that existed previously. We say a feature F *preserves existing values* if F's method refinements write A) to a field that F introduced or B) to a field introduced by another feature, G, but whose base object (e.g., base object for the expression `s.text` is `s`) was allocated by F. The reason for Condition A is the following: a field introduced by F cannot have existed before F was added. As a result, writing to the field cannot possibly overwrite existing values. As for Condition B, F should be able to modify objects that it itself created. It should not matter who declared the field if the field belongs to an object created by the feature. We run the heap-effect analysis on the SysGen program, rather than Jak modules, because there is an existing off-the-shelf analysis for it.

Here are three examples:

- Example satisfying A): given `BaseTest` (Figure 5) and the SysGen program (Figure 7), we see that `Backup` satisfies the heap-effect conditions because the feature's only method refinement, lines 15-16 in Figure 7, updates field `prev`, which the feature itself introduced.

- Example satisfying B): instead of `Backup` writing to `prev` field, suppose it did the following:

  ```
  Status s = new Status();
  s.setText("hello");
  ```

  Even though `Backup` updates field `text` through `setText`, with `text` having been introduced by another feature (`Base`), this is acceptable because the modification only affects the object `s` which did not exist prior to the addition of `Backup`.

- Example not satisfying A) and B): `Censor` modifies `s.text` and `s` was created by `BaseTest`, not `Censor`.

The heap-effect analysis checks both conditions. It finds writes to fields that each method refinement makes, i.e., it finds writes occuring in the control-flow of each `if(FEATURE)` statement. For each write, the feature of the field (denoted by `@Feature` Java 5 annotation) is compared against the feature of the method refinement. If the two features are the same, the heap-effect analysis passes (condition A). 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 the same as the feature of the method refinement for the heap-effect analysis to pass (condition B). A feature S can fail the heap-effect analysis due to another feature R, which in the control-flow of S's method refinement, writing to a field or object not owned by S. This is elaborated in a technical report [20].

We modified an off-the-shelf inter-procedural side-effect analysis [22] that uses a context-insensitive and flow-insensitive points-to analysis, called *Spark* [24]. We chose this particular analysis because it was easy to modify and we are dealing with large programs, for which context-sensitive points-to analyses are very expensive [33].

But a context-insensitive analysis is considerably less precise than a context-sensitive one as the former merges call sites instead of following call paths. For example, in Figure 7, the two calls to `applyChecker(..)` (line 26 and and line 28) are merged, and in line 38, the actual subclass of `c` cannot be determined, and the analysis falsely concludes that `c.check(this)` can dispatch to either `CensorChecker.check(..)` or `StyleChecker.check(..)`. This in turn causes the analysis to conservatively conclude that `Style`, with the method refinement in line 28, can modify `s.text` through line 52 (which is what `Censor`, not `Style`, actually does).

To circumvent the inability to distinguish calls sites, whenever we see that a feature calls a method of another feature in the control-flow of a method refinement, we inline the call. Thus, the method `applyChecker(..)` (of `Base` feature) is inlined in Figure 7, lines 26 (of `Censor` feature) and 28 (of `Style` feature). Combining inlining with context-insensitivity analysis is a middle-ground between a context-insensitive analysis like Spark and a context-sensitive analysis like Paddle [23]. This middle-ground is shown to be effective in Section 7, although we leave detailed comparisons between the three possibilities for future work.

In some cases, by design, features are expected to modify fields introduced by other features. We make these fields explicit with `@Modifiable` annotations. For example, `Base`, which declares `valid` as a modifiable field (Figure 7, line 6), expects refining features to contribute to the value of the field by calling `applyChecker(..)`. We allow writes to the field to be ignored.

Writes to library-owned data, such as input/output effects, are considered behavior-irrelevant. To consider such a write to be behavior-relevant, one must make the data, e.g. `Screen` in Figure 1, a part of a feature, e.g. `Base`.

6. FINDING CONFIGURATIONS TO TEST

We now address the key problem of systematically identifying feature configurations to run for a given test. Figure 9 lists our algorithm. We start by producing the SysGen program *sg* of an SPL. Then, given *sg* and program *test*, we determine the set of behavior-irrelevant features, `irrelevantFeatures`, using the two step analysis described in Section 5.

Next, we convert the feature model, with both domain and implementation constraints (e.g. if feature A requires feature B for compilation, then A⇒B), into a propositional formula [4], which defines all legal feature configurations. The feature model is conjuncted with those features for which the test is intended (lines 8-10). Configurations that satisfy these constraints are `candidates` (line 12).

Ideally, all `candidates` will have their `irrelevantFeatures` set to `false`. Unfortunately, it is not that simple, since `constraints` may insist that `irrelevantFeatures` are required by other features. To reduce the set, lines 14 to 26 choose candidates with as many of the `irrelevantFeatures` features set to `false` as possible. If the resulting candidate hasn't been encountered before, it is added to `solutions`, which represents the set of configurations that must

```
void runProductLineTest
        (ProductLine productLine, Test test) {
  Program sg = transformIntoSysGen(productLine);

  Set<Feature> irrelevantFeatures =
    twoStepStaticAnalysis(sg, test.getMainMethod());

  PropositionalFormula constraints =
    test.getFeaturesToTest().and
      (productLine.getFeatureModel());

  Set<Configuration> candidates = solve(constraints);

  Set<Configuration> solutions = new SetImpl<Configuration>();

  for(Configuration c: candidates) {
    for(Feature irrFeature: irrelevantFeatures) {
      if(c.valueOf(irrFeature)) {
        c.setValueOf(irrFeature, false);
        if(!c.satisfies(constraints))
          c.setValueOf(irrFeature, true);
      }
    }
    if(solutions.add(c))
      run(test, sg, c);
  }
}
```

Figure 9: Algorithm for Running an SPL Test

Table 1: Configurations for BaseTest

In Solution	Base	Censor	Style	Backup
Yes	1	0	0	0
No	1	0	0	1
Yes	1	0	1	0
No	1	0	1	1
Yes	1	1	0	0
No	1	1	0	1
Yes	1	1	1	0
No	1	1	1	1

be tested.[4]

As an example, consider `BaseTest`. Each row displayed in Table 1 corresponds to a configuration in `candidates`. Each row whose `In Solution` is `Yes` is a configuration in `solutions`. Only `Backup` is irrelevant to `BaseTest`, meaning all combinations of `Censor` and `Style` must be tested. In this example, we reduced 8 tests to 4 tests.

Table 2 shows the configurations for `StyleTest`. Both `Base` and `Style` are required (as the test is for `Style` and the test references `Screen` class of `Base`). `Censor` and `Backup` are behavior-irrelevant as they do not refine methods (i.e. `StyleChecker.check(..)` and `String.startsAlphanumerically(..)`) that are called in the control-flow of `StyleTest`. Only one configuration, containing `Style` and `Base`, needs to be tested out of the four.

7. CASE STUDIES

We implemented our technique as an Eclipse plugin called `SPLTester`, which relies on Soot [35]. We evaluated our plugin on three product lines: *Graph Product Line (GPL)*, which is a set of programs that implement different graph algorithms [27] and *jampack* and *mmatrix*, which are feature-configurable tools that are part of the AHEAD Tool Suite [3]. For each product line, we took the `main` method as the sole test. All the features required to compile `main` formed the base program. Our analysis determined features whose combinations could change the control-flow or data-flow of the base program. As we did not model library code enabling input/output as part of the base program, features could have I/O effect and still be irrelevant to the base program. We ran our tool on a Windows XP machine with Intel Core2 Duo CPU with 2.4 GHz and 2 GB of RAM. We describe each benchmark in three dimensions: SysGen program, static analysis, and configuration reduction.

[4]We recognize an inherent inefficiency of enumerating `candidates` (which may be exponential in the number of optional features) and subsequently reducing them to the set `solutions`, which we expect to be much smaller. Computing `solutions` directly, a constraint satisfaction problem [6], is a subject for future work.

Table 2: Configurations for StyleTest

In Solution	Base	Censor	Style	Backup
Yes	1	0	1	0
No	1	0	1	1
No	1	1	1	0
No	1	1	1	1

Table 3: GPL Results

Static Analysis	
Lines of code	1713
# of method refining behavior-irrelevant features / # of method refining features	10/12
# of behavior-irrelevant features / # of features	15/17
@Modifiable annotations	Vertex.visited, Graph.last, Graph.current,Graph.accum, Graph.isDirected
Duration	1212578 ms (20.21 minutes)
Refining Feature	**Behavior-Irrelevant**
Shortest	Yes
MSTPrim	Yes
Undirected	Yes
Search	Yes
Weighted	Yes
MSTKruskal	Yes
BFS (Breadth-First Search)	No (heap-effect violation)
DFS (Depth-First Search)	No (heap-effect violation)
Cycle	Yes
StronglyConnected	Yes
Connected	Yes
Number	Yes
Configuration Reduction	
# of configurations without static analysis	56
# of configurations with static analysis	8

7.1 Graph Product Line (GPL)

In creating the SysGen program for GPL, we encountered two features, `Directed` and `Undirected`, each of which introduced the same method, but with different bodies. To prevent one feature from overriding code of the other feature, we refactored the feature modules such that they would refine an empty method defined in the `Base` feature. Other than this refactoring, the transformation to produce SysGen program from the Jak modules was automatic. Table 3 shows the results of running the two-step static analysis and configuration reduction.

Despite being a small product line in terms of code size, statically analyzing GPL led us to address important issues. First, we discovered that using an entirely context-insensitive points-to analysis yielded too many false positives because of the merging of call sites across features. To address this, we used the inlining strategy of Section 5.2.

We also had to inline some code manually, as Soot was not able to inline constructor calls.

Second, we initially found that many features modified variables introduced by another feature. For example, whether or not a vertex has been visited is stored in `Vertex.visited`, which different algorithms modify. Ideally, with a much more powerful static analysis, we would be able to check that the `visited` flag is modified by a graph algorithm (e.g. vertex numbering, cycle checking, etc.) during its execution, and the modification would not interfere with the execution of other graph algorithms. Lacking this however, we allowed five fields that are expected to be modified by other features to be marked with `@Modifiable` annotations.

The numbers indicate that our analysis was useful. As shown in the row "# of method refining behavior-irrelevant features / # of method refining features", out of 12 features that refine methods in the control-flow of the test, 10 are shown to be behavior-irrelevant. `BFS` and `DFS` are behavior-relevant because optional graph algorithms like `Number` calls methods in these features, which in turn call-back the algorithms' `preVisit` and `postVisit` methods to modify the algorithms' own data.

12 method refining features out of 17 features means that there are 17-12=5 features that only add declarations. Because these 5 features do not override methods and are not referenced through reflection (we checked these while performing the manual refactoring described at the beginning of this section), they are behavior-irrelevant layered-designs, as discussed in Section 4. Adding these 5 to the 10 method refining features that are behavior-irrelevant, we get 15 behavior-irrelevant features in total.

Although our analysis takes some time to run (20.21 minutes), it is because after each inlining operation, our non-optimized implementation has to re-run Soot, which has a considerable startup overhead regardless of the program analyzed.

In summary, without the static analysis, we would have to test 56 configurations, whereas with it, we would have to test only 8. (Note: Although there are 2 behavior-relevant features, the number of configurations need not be 2^2; there could be more or fewer, depending on the constraints imposed by the feature model.)

7.2 jampack

`jampack` (see Table 4) is a product line whose largest program (i.e., one containing all optional features) is over 39K lines of code. Despite its size and reasonable number of features (19), unlike GPL, there are only four features that refine methods. Most `jampack` features introduce classes into an existing hierarchy, which is common in layered designs.

Our analysis revealed that 18 features were behavior-irrelevant (since they just introduce methods and classes). Only one feature refines methods and it alteres the control-flow of another feature. With this information, we can safely ignore 18 out of 19 features, which, as we discuss shortly, reduces testing by a significant amount. Interestingly, although `jampack` is 20 times larger than GPL, analyzing `jampack` took much less time (3.64 min) to analyze than GPL because of there is only one feature refining methods in the control-flow of `main` method.

Without static analysis, `jampack` would have to be tested on 276 configurations. With static analysis, only 4 configurations have to be tested.

Table 4: jampack Results

Static Analysis	
Lines of code	39259
# of method refining behavior-irrelevant features / # of method refining features	0/1
# of behavior-irrelevant features / # of features	18/19
@Modifiable annotations	None
Duration	218140 ms (3.64 minutes)
Refining Feature	**Behavior-Irrelevant**
Java	No (cflow-effect violation)
Configuration Reduction	
# of configurations without static analysis	276
# of configurations with static analysis	4

Table 5: mmatrix Results

Static Analysis	
Lines of code	22492
# of method refining behavior-irrelevant features / # of method refining features	0/1
# of behavior-irrelevant features / # of features	10/11
@Modifiable annotations	None
Duration	968781 ms (16.15 minutes)
Refining Feature	**Behavior-Irrelevant**
Java	No (cflow-effect violation)
Configuration Reduction	
# of configurations without static analysis	65
# of configurations with static analysis	1

7.3 mmatrix

`mmatrix` (see Table 5) is a product line whose largest program (i.e., one containing all optional features) is 35K lines of code. Similar to `jampack`, `mmatrix` exhibits a layered design with only one feature that refines methods.

Our analysis revealed that 10 features were behavior-irrelevant (they just introduce methods and classes). The method refining feature alters the control-flow of another feature. With this information, we can safely ignore 10 out of 11 features. The analysis takes 16.2 minutes.

Without static analysis, `main` method of `mmatrix` would have to be run on 65 configurations to see how they change, if at all, the method's execution with the required features. With static analysis, only one has to be run.

8. RELATED WORK

8.1 Product Line Testing

It is well-known that the large number of configurations in a product line poses a serious challenge for testing [30]. There is prior research on sampling the configuration space on coverage criteria, not based on program analysis results, but rather, based on interactions that domain experts believe to exist. For example, an SPL tester may choose a set of features for which all combinations must be examined, while for other features, only pair-wise testing is done [11][29]. In contrast to these process-oriented approaches, our work employs program analysis to systematically walk through and prune feature combinations. However, it only does so in the context of a given test. The process-oriented approaches may be complementary to our approach in that they can be used to construct the given tests for which our technique determines the feature combinations to test.

There has been prior work on testing in the context of feature-oriented product lines. In [37], instead of generating tests from a complete specification of a program, tests are generated incrementally from specifications of features. We address a different problem.

8.2 Feature Interactions

There is a large body of work on detecting feature interactions using static analysis [32][36][13][10][25], of which *harmless advice* [13] and *Modular Aspects with Ownership (MAO)* [10] are most relevant to our work. They too try to determine if features, or code modules like aspects, are behavior-irrelevant. Our two-step static analysis was inpired by the analysis proposed in the MAO paper. However, our analysis differs in that ours performs a whole-program, inter-procedural analysis, while theirs relies on module specifications to perform intra-procedural analysis. Although ours also allows a form of module specifications, i.e. `@Modifiable` annotations, unlike MAO, the specifications are not central to the technique. Our analysis is similar to the analysis employed in harmless advice in that both are whole-program, inter-procedural analyses. However, unlike our analysis, theirs uses a type system that prevents harmful information flow but allows control-flow to be changed. Also, their approach requires every feature to be harmless (i.e., irrelevant), but this is inappropriate for SPLs in general. Our analysis, which provides a program analysis but leverages domain knowledge, can be seen as a middle-ground between a specification-based analysis like MAO and a specification-independent analysis like harmless advice.

More importantly, the two related works, as well as other works, assume a setting, such as an aspect-oriented program, where all modules are required for the program to work, which is sharply different from SPLs. Indeed, prior works performed analysis more for modular reasoning, rather than for reducing combinatorial testing.

Also, our technique's aim is to find combinations of optional features that can change a given test's outcome, *not* to reveal all possible interactions involving the optional features. For example, given a test, by turning off a feature irrelevant to that test, we may prevent execution of what our tool reports to be relevant features in the irrelevant feature's control-flow. Although this seems odd, it is expected because the "relevant" features are relevant *not* to the test, but to the irrelevant feature (if they were relevant to the test, the irrelevant feature would have also been reported to be relevant to the test). This "oddity" arises because our tool does not consider relevance as feature relations or feature interactions. In our approach, there must be another test for the irrelevant feature in order to reach the "relevant" features. This point is elaborated in a technical report under preparation [20].

8.3 Compositional Analysis and Verification

Although we are able to reuse an off-the-shelf static analysis by transforming a product line into an ordinary whole program, we lose the benefit of *compositionality* this way. Compositional analysis performs analysis per feature and merge the result with those of analyzing other features and is especially effective if the product line evolves frequently. A possible future direction is to develop effective compositional static analysis techniques for product lines, perhaps based on existing techniques like [12]. Feature specifications are not necessary for our technique, but if we support them for another purpose in the future, we may also benefit from considering compositional verification techniques [25, 14].

8.4 Reducing Testing Effort

There is also related work on reducing testing, typically using output from some analysis, although such work is not in the context of product lines. For example, a regression testing technique like [34] identifies a subset of existing tests to run given a program change or a feature. We address the opposite problem, i.e., we identify a subset of existing features to run given a test. The two techniques are complementary as both settings can occur. From a high level, both regression testing and our technique can be seen [28] as a form of slicing [39].

9. CONCLUSIONS

Software Product Lines (SPLs) represent a fundamental approach to the economical creation of a family of related programs. Testing SPLs is no less important, but has been largely an ad hoc and informal process. This was the starting point of our work.

Features are a basic, 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 exponential 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 structured way to analyze an SPL for a given test. We showed how conventional static analysis could eliminate feature configurations for a test by identifying features that are *behavior-irrelevant* — features that do not affect the properties that are being evaluated. We presented several case studies that showed sizable reductions in the number of tests to consider, and more importantly, lends credence to the folk-tail belief that many features of a product line are behavior-irrelevant.

We believe that testing SPLs is a topic of research that is long overdue. We have taken a step toward making SPL testing a more structured and formal process.

Acknowledgements. Kim and Batory are supported by NSF's Science of Design Project #CCF-0724979.

10. 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] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE'05*.

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

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

[5] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE'03*.

[6] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *CAiSE'05*.

[7] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/j: controlling the scope of change in java. In *OOPSLA '05*. ACM.

[8] Y. Bontemps, P. Heymans, P.-Y. Schobbens, and J.-C. Trigaux. Semantics of FODA feature diagrams. pages 48–58. Technical Report 6 – HUT-SoberIT-C6, Aug. 2004. Available from `http://www.soberit.hut.fi/SPLC-DWS/`.

[9] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA'02*, July 2002.

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

[11] 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.

[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] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[18] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE'08*.

[19] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP'01*.

[20] C. H. P. Kim, D. Batory, and S. Khurshid. Reducing combinatorics in testing product lines (technical report under preparation).

[21] C. Krueger. Variation management for software production lines. In *SPLC'02*, volume 2379 of *LNCS*.

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

[23] O. Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, Jan. 2006.

[24] 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.

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

[26] J. Liu and D. Batory. Automatic remodularization and optimized synthesis of product-families. In *GPCE*, 2004.

[27] 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.

[28] R. Mazumdar. Private correspondence, 2009.

[29] 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`.

[30] 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.

[31] N. Nystrom, S. Chong, and A. C. Myers. Scalable extensibility via nested inheritance. *SIGPLAN Not.*, 39(10):99–115, 2004.

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

[33] T. Reps. Undecidability of context-sensitive data-independence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, 2000.

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

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

[36] G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In B. Magnusson, editor, *ECOOP*, volume 2374 of *Lecture Notes in Computer Science*, pages 562–584. Springer, 2002.

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

[38] 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.

[39] 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.

[40] Wikipedia. Micro-blogging. `http://en.wikipedia.org/wiki/Micro-blogging`.

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