Runtime Verification 2010, Malta

Enforcing Safety Properties in Product Lines

Chang Hwan Peter Kim[1], Eric Bodden[2], Don Batory[1] and Sarfraz Khurshid[1]

[1] Department of Computer Science and
Department of Electrical and Computer Engineering
The University of Texas at Austin, USA
{chpkim@cs, batory@cs, khurshid@ece}.utexas.edu
[2] Software Technology Group
Technische Universität Darmstadt, Germany
bodden@st.informatik.tu-darmstadt.de

Abstract. A *product line* is a family of programs where each program is defined by a unique combination of features. Product lines, like conventional programs, can be checked for safety properties through execution monitoring. However, applying execution monitoring techniques for conventional programs against product lines is expensive because one would have to monitor all of the product line's programs, the number of which can be exponential in the number of features. Also, doing so would be wasteful because many programs can provably never violate a stated property. We introduce a monitoring technique dedicated to product lines that given a safety property, determines the statements that need to be instrumented for the corresponding execution monitor to trigger and the feature combinations required for these statements to be reached. Thus, we identify feature combinations that cannot possibly violate the stated safety property, reducing the number of programs to monitor. Experiments show that our technique is effective, particuarly for safety properties that crosscut many optional features.

1 Introduction

A software *product line* is a family of programs where each program is defined by a unique combination of *features*. By developing programs with commonalities and variabilities in a systematic way, product lines help reduce both the time and cost of software development [14]. Concomitantly, product lines pose significant new challenges as they involve reasoning about a family of programs, whose cardinality may be exponential in the number of features.

In this paper we consider the problem of runtime monitoring product lines for violations of safety properties. A safety property defines a set of unacceptable program executions that can only be determined through runtime monitoring [13]. We avoid monitoring every program of a product line by statically identifying feature combinations (i.e., programs) that provably can never violate the stated property. These programs do not require testing (for this property), which can significantly speed up the testing process overall.

We accomplish this goal by starting with analyses that shows how to evaluate runtime monitors at compile time [6, 7, 5, 4] for *single* programs. Our work extends these analyses by lifting them to understand features, making them aware of possible feature combinations. A programmer needs to apply our analysis to a product line once. The output is a bi-partitioning of feature configuations: (1) configurations that need to be monitored because violations may occur and (2) configurations for which no violation can happen.

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

To summarize, the contributions of this paper are:

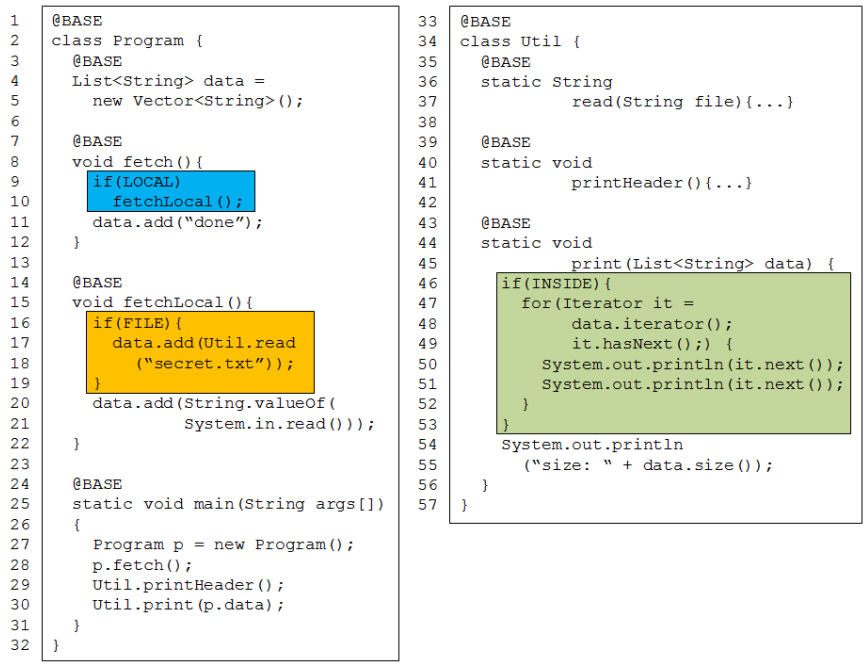
- A novel static analysis to determine, for a given product line and runtime monitor specification, the feature combinations (programs) that need to be monitored.
- An implementation of this analysis within the Clara framework for hybrid typestate analysis [3], as an extension to Bodden et al.'s earlier whole-program analysis [6].
- Experiments that show that our analysis can noticably reduce the number of configurations that require runtime monitoring and thus can therefore result in noticable savings in testing time.

2 Motivating Example

Figure 1 shows an elementary product line, whose programs fetch data and print it. There are different ways of representing a product line. In this paper, we use a *SysGen* representation [10], where a product line is an ordinary Java program whose members are annotated and statements are conditionalized using feature identifiers (in a manner similar to `#ifdef`).[3] Local data is fetched if the `Local` feature is selected (blue code), local data from a file is fetched if `File` is selected (yellow code) and internal contents of data are printed if `Inside` is selected (green code). Each member (class, field, or method) is annotated with a feature. In this example, every member is annotated with `Base` feature, meaning that it will be present in a program only if the `Base` feature is selected. In SysGen, a program (also referred to as a *configuration* or *feature combination*) is instantiated by assigning a Boolean value for each feature and statically evaluating feature-conditionals and feature-annotations.

Product lines are usually associated with a *feature model* [2] that constrains the allowed combinations of features. The feature model for our product line is expressed below as a context-sensitive grammar. `Base` is a required feature: optional features (`Inside`, `File`, `Local`) are listed in brackets.

[3] Note that we capitalize feature identifiers for presentation purposes. Although feature identifiers in an if-conditional are implemented as field references, for presentation purposes we do not qualify the field reference with its class.



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

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

Fig. 1. Example Product Line

```
Example :: [Inside] [File] [Local] Base;
Inside or File or Local;
// Implementation constraints
(Inside implies Base) and (File implies Base) and (Local implies Base);
```

The model further requires at least one of the optional features to be selected (line 2). In the last line, the feature model enforces additional implementation constraints that must hold for all programs in the product line to compile. Example: the code of the `File` feature references methods in `Base`. A technique described elsewhere [15] can generate these implementation constraints automatically. In total, the feature model allows seven configurations (i.e. our product line has seven distinct programs).

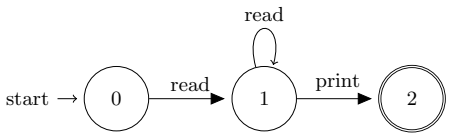
2.1 Problem

Researchers have developed a multitude of specification formalisms for defining runtime monitors. As our approach extends the Clara framework, it can generally apply to any runtime-monitoring approach that uses AspectJ aspects for monitoring. This includes popular systems such as JavaMOP and tracematches. For the remainder of this paper, we will use the tracematch [1] notation because it can express monitors concisely. Figure 2(a) shows a simple example. `ReadPrint` prevents a `print` event after a `read` event is witnessed. In line 3 of

Figure 2(a), a `read` symbol captures all those events in the program execution, known as *joinpoints* in AspectJ terminology, that are immediately *before* calls to `Util.read*(..)`. Similarly, the symbol `print` captures joinpoints occurring immediately *before* calls to `Util.print*(..)`. Line 6 carries the simple regular expression "`read print`", specifying that code body in lines 6–8 should execute whenever a `print` follows a `read` on the program's execution. Figure 2(b) shows a finite-state machine for this tracematch, where symbols represent transitions.

```
1  aspect ReadPrint {
2    tracematch() {
3      sym read before:  call(* Util.read*(..));
4      sym print before: call(* Util.print*(..));
5
6        read+ print {
7          throw new RuntimeException("possible leak!");
8        }
9    }
10 }
```

(a) ReadPrint Tracematch



(b) Finite-State Machine

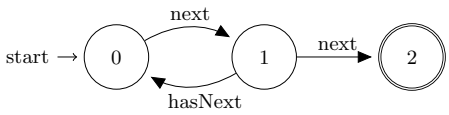
Fig. 2. ReadPrint Safety Property

Figure 3 shows a second and different safety property: `HasNext` [6]. The property checks if `Iterator.next()` is called twice without calling `Iterator.hasNext()` in between. Note that this tracematch only matches if the two `Iterator.next()` calls bind to the same `Iterator` object `i`, as shown in Figure 3(a), lines 2–4. When the tracematch encounters an event matched by a declared symbol that is not part of the regular expression, such as `hasNext`, the tracematch discards its partial match. Therefore, the tracematch would match a trace "`next(i1) next(i1)`" but not "`next(i1) hasNext(i1) next(i1)`", which is exactly what we seek to express.

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

```
1  aspect HasNext {
2    tracematch(Iterator i) {
3      sym next    before: call(* Iterator.next()) && args(i);
4      sym hasNext before: call(* Iterator.hasNext()) && args(i);
5
6        next next {
7          throw new RuntimeException("check hasNext!");
8        }
9    }
10 }
```

(a) HasNext Tracematch



(b) Finite-state machine

Fig. 3. HasNext Safety Property [6]

2.2 Goal

Our goal is to statically determine the feature configurations to monitor or conversely the configurations that cannot trigger the monitor. For our running example, we can deduce these configurations by hand. For `ReadPrint`, both `read` and `print` symbols have to match, meaning that `File` (which calls `Util.read(..)` in line 17) and `Base` (which calls `Util.print*(..)` in lines 29 and 30) have to be present for the monitor to trigger. Also, `Local` needs to be present because it enables `File`'s code to be reached. Therefore, the `ReadPrint` monitor has to be inserted if and only if these three features are present, i.e. on two of the seven configurations. We represent the condition under which the monitor has to be inserted by treating a monitor as a feature itself (e.g. `ReadPrint`) and constructing the *presence condition* for it: `ReadPrint = File and Local` (we do not include `Base` because it's always present) Similarly, the `HasNext` property only has to be inserted iff `Iterator+.next()` can be called, i.e. on the four configurations with `Inside` present. The presence condition for `HasNext` is `HasNext = Inside`. The goal of our technique is to change the original feature model so that tracematches are now features themselves and the tracematch presence conditions are part of the feature model:

```
// ReadPrint and HasNext are now features themselves
Example :: [ReadPrint] [HasNext] [Inside] [File] [Local] Base;
Inside or File or Local;
// Implementation constraints
(Inside implies Base) and (File implies Base) and (Local implies Base);

// Tracematch presence conditions
ReadPrint = File and Local;
HasNext = Inside;
```

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

2.3 The Need for a Dedicated Static Analysis for Product-Lines

As mentioned earlier, there exists a static analysis that reduces a monitor's instrumentation against a single program [6, 7, 5], which we will refer to as a "traditional program analysis". There are two ways to use this analysis. One way is inefficient, the other way imprecise: running the traditional program analysis against each instantiated program will be very inefficient because it will have to inspect every product program separately.

Another way would be to run the traditional program analysis against the SysGen product line itself. By not evaluating the feature conditionals statically, the analysis can treat the product line as an ordinary program. However, this approach can be imprecise. For example, suppose we apply the traditional program analysis on the `ReadPrint` and `HasNext` tracematches: both tracematches may match in the case in which all features are enabled. Being oblivious to the notion of features, the analysis will therefore report that the tracematches always have to be present. This shows that a static analysis, to be both efficient and effective on a product line, has to be aware of the product line's features.

Figure 4 displays an overview of our technique. First, our analysis determines the symbols required for the tracematch to trigger ("Determine Required Symbols"). For each of these symbols, we use the aspect weaver to identify the statements that are matched by the tracematch's declared symbols ("Determine Symbol-To-Shadows"). We elaborate these steps in Section 3. Then, for each of these statements, we determine what feature combinations need to be present for the statement to be reachable from the program's `main()` method. This results in a set of *presence conditions*. We combine all these conditions to form the presence condition of the tracematch. We repeat the process for each tracematch ("Determine Presence Conditions") and add the tracematches and their presence conditions to the original feature model ("+"). We explain these steps in Section 4.

3 Required Symbols and Shadows

A safety property must be monitored for a feature configuration c if the code in c may drive the finite-state monitor from its initial state to its final (error) state. In earlier work [6], Bodden et al. described three different algorithms that try to determine, with increasing degrees of sophistication, whether a single program can drive a monitor into an error state, and using which transition statements. The first, called *Quick Check*, rules out a tracematch if the program does not even contain transition statements required to reach the final automaton state. The second, called *Consistent-Variables Analysis*, performs a similar check on every consistent variable-to-object binding. The third, called *flow-sensitive*, check rules

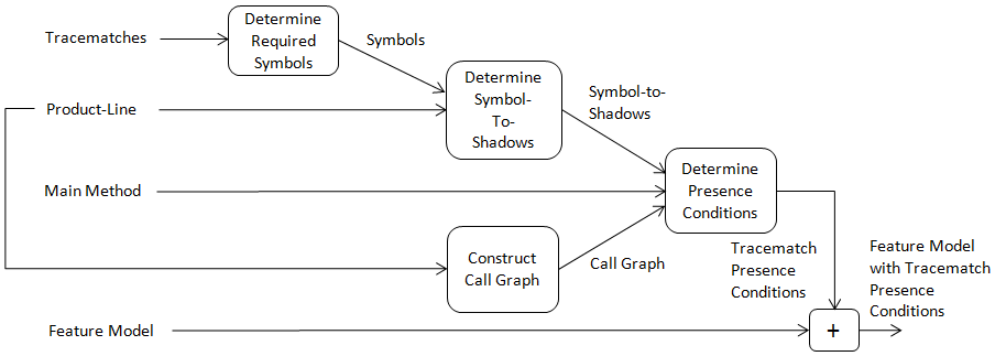


Fig. 4. Overview of Our Technique

out a tracematch if the program cannot execute its transition statements in a property-violating order.

In this paper, we limit ourselves to extending the Quick Check to product lines. The Quick Check has the advantage that, as the name suggests, it executes quickly. Nevertheless, our results show that even this relatively pragmatic analysis approach can already considerably reduce the number of configurations that require monitoring. It should be simple to extend our work to the other analyses that Bodden et al. proposed but we consider this task less interesting because it does not fundamentally alter our technique.

3.1 Required Symbols

A symbol represents a set of transition statements with the same label. Given a tracematch, we determine the *required symbols*, i.e. the symbols required for the error state to be reached, by removing each symbol one at a time and seeing if removing the automaton edges with the symbol prevents the final state from being reached. A configuration that doesn't match a required symbol does not have to be monitored. For `ReadPrint` property, the symbols `read` and `print` are required because without even one of these, the final state in Figure 2(b) cannot be reached. For `HasNext` property, only the symbol `next` is required. Note that the final state can be reached without `hasNext`.

3.2 Symbol-to-Shadows

For each required symbol, we determine its *joinpoint shadows* (*shadows* for short), i.e. program statements that cause the symbol to match. The symbol will match — cause transition in the automaton — if even one of the shadows is executed. Each symbol is a conventional AspectJ pointcut and as such, determining its shadows requires asking the weaver where they are. The way `abc`, a compiler for tracematches that we are using, is built requires these shadows to

be actually woven into the SysGen product line first in order to determine the symbols associated with them. And because the weaving is done against bytecode, the SysGen product line has to be first compiled (as an ordinary program without partially evaluating feature-conditionals and feature-annotations). Note that the weaving against the SysGen product line is done solely to help determine the necessary configurations which are expressed as tracematch presence conditions (see Figure 4). The actual tracematch weaving has to be done for each configuration or program.

In our example, for `ReadPrint` tracematch, `read` symbol's shadow is `Util.read("secret.txt")` call in line 17 of Figure 1 and `print` symbol's shadows are `Util.printHeader()` call in line 29 and `Util.print(p.data)` call in line 30. For `HasNext` tracematch, `next` symbol's shadows are `Iterator.next()` calls in lines 50 and 51 and `hasNext` symbol's shadow is `Iterator.hasNext()` call in line 49.

4 Presence Conditions

A tracematch has to be inserted only on the configurations where each of the required symbols is present. Thus, a tracematch's presence condition (PC) is the conjunction of the presence condition of each of the required symbols. A symbol is present if even one of its shadows is present. Thus, a symbol's presence condition is the disjunction of the presence condition of each of its shadows. For a shadow to be present, features of its enclosing feature-conditionals and method and class must be present. In summary, a tracematch is computed in the following way:

```
pc(tracematch) = pc(reqdSymbol1) and pc(reqdSymbol2) and ... and pc(reqdSymbolN)
               = {pc(shadow11) or pc(shadow12) or ... or pc(shadow(1X))} and
                 {pc(shadow21) or pc(shadow22) or ... or pc(shadow(2Y))} and ... and
                 {pc(shadowN1) or pc(shadowN2) or ... or pc(shadow(NZ))}

pc(shadow) = feature(condition1) and feature(condition2) and ... and feature(conditionJ) and
             feature(classMember1) and feature(classMember2) and ... and feature(classMemberK)
```

For example, Figure 5 shows how `ReadPrint` tracematch is determined. As mentioned in Section 3.1, the tracematch's required symbols are `read` and `print` (Figure 5, line 1), `read` has one shadow in line 17 of Figure 1 and `print` has two shadows in lines 29 and 30. (Figure 5, line 2). For line 17 shadow to be syntactically present in a program, the program must have the `if(FILE)` conditional in line 16 and the `fetchLocal()` method definition (annotated with `BASE` in line 14), which explains how `pc(line17)` is expanded into `[File and Base]`. Similarly, `pc(line29)` and `pc(line30)` are each expanded into `[Base]` because each of the shadows just requires the `main` method definition, which is annotated with `BASE`.

Figure 5 is imprecise in that it allows configurations where a shadow is syntactically present, but is not reachable from the `main` method of the program. For example, according to the algorithm, `Util.read(..)` shadow is "present" in configurations {`Base=true`, `Local=false`, `File=true`, `Inside=DONT_CARE`} even though it is not reachable from `main` due to `Local` being turned off. A

```
1 ReadPrint = pc(read) and pc(print)
2           = {pc(line17)} and {pc(line29) or pc(line30)}
3           = {File and Base} and {Base or Base}
```

Fig. 5. Computing ReadPrint's Presence Condition

more precise algorithm could require not only the shadow's syntactic containers to be present, but also its transitive callers. The full algorithm for precisely computing a shadow's presence condition is not shown for space reasons, but it basically takes a shadow's imprecise presence condition and conjoins it with the disjunction of the callers' precise presence conditions. For example, for the line 17 shadow, which is called by line 10 that is in turn called by line 28, the precise algorithm would return:

```
pc(line17) = [enclosingFeatures and (pc(caller1) or pc(caller2) or ... or pc(callerM))]
           = [enclosingFeatures and (pc(line10))]
           = [enclosingFeatures and (enclosingFeaturesLine10 and (pc(line28)))]
           = [File and Base and (Local and Base and (Base))]
           = File and Local
```

Substituting this in Figure 5, we get `ReadPrint = File and Local`, which is what we seeked to achieve as outlined in Section 2.2. Similarly, `HasNext`'s presence condition is:

```
HasNext = pc(next)
        = {pc(line50) or pc(line51)}
        = {[Inside and Base and (Base)] or [Inside and Base and (Base)]}
        = Inside
```

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

4.1 Technical Subtleties

There are subtle, but important factors to take into account when computing a shadow's precise presence condition, which we discuss using a more involved example shown in Figure 6. Suppose that `Util.read(..)` is the shadow whose presence condition we're computing. For simplicity, recursive edges in the call-graph are not included in the presence condition. It is safe to do so because the shadow can be still be reached by ignoring the feature controlling the recursive edge. To see why, consider the recursion between `a()` and `c()` in the call-graph. The shadow `Util.read(..)` can actually be reached only through the recursive edge being present, i.e. `X=true`. If we do not include this constraint on `X`, we will just end up inserting the monitor for both values of `X`, which is imprecise but sound.

Also, `main`'s callers (e.g. `D`) is not included because `main` can be executed through the execution environment, without having a caller. And `main` must be

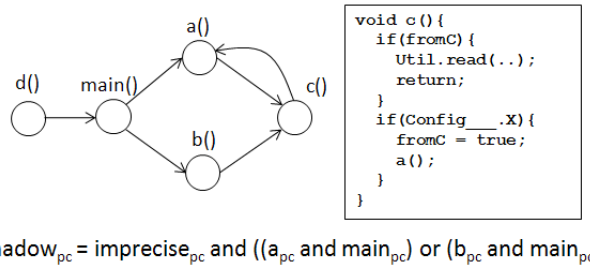


Fig. 6. Example of Computing a Precise Shadow Presence Condition

present at least once in a shadow's presence condition. Otherwise, the shadow is not reachable and we can simply return `false` as the shadow's presence condition.

4.2 Precision

Conjunctions strengthen the presence condition of the callee, but they are entirely optional. For example, we can return the imprecise presence condition of the callee in this example and the result will still be sound, albeit less precise. This nice property allows us, through user's preference for example, to control the size of the presence condition by simply not going any farther in the call-graph, thereby trading off precision for a more feasible analysis. Also, our technique works with either a context-sensitive or -insensitive call-graph, although we use a context-insensitive one constructed from *Spark* [11] for evaluation.

5 Evaluation

We implemented our analysis by extending the *Clara* static analysis framework [3] and evaluated it on the following product lines: *Graph Product Line (GPL)*, a set of programs that implement different graph algorithms [12] and *Notepad*, a Java Swing application with functionalities similar to Windows Notepad. We considered three safety properties for each product line. For each property, we report the number of configurations on which the property has to be monitored and the execution time to derive the tracematch presence condition. We ran our tool on a Windows 7 machine with Intel Core2 Duo CPU with 2.2 GHz and 2 GB of RAM.

Note that, although the product lines were created in-house, they were created long before this paper was conceived (GPL over 5 years ago and Notepad 2 years ago). Our tool, the examined product lines and monitors, as well as the detailed evaluation results are available for download [9].

5.1 Case Studies

Graph Product Line (GPL) Table 1 shows the results for GPL, which has 1713 LOC with 17 features and 156 congurations. Variations arise from algorithms and structures of the graph (e.g. directed/undirected and weighted/unweighted).

Table 1. Graph Product Line (GPL) Results

Lines of code	1713
No. of features	17
No. of configurations	156
DisplayCheck	
No. of configurations	67 (43%)
Duration	95.5 sec. (1.6 min.)
SearchCheck	
No. of configurations	58 (38%)
Duration	225.3 sec. (3.8 min.)
KruskalCheck	
No. of configurations	13 (8%)
Duration	86.0 sec. (1.4 min.)

The `DisplayCheck` safety property checks if the method for displaying a vertex is called not in the control flow of the method for displaying a graph, which would be a behavioral API violation. Instead of monitoring all 156 configurations, our analysis reveals that only 67 configurations, or 43% of 156, need monitoring. Our analysis took 1.6 minutes to complete. (The tracematch presence constraint that represents these configurations is available on our website [9].)

`SearchCheck` checks if the search method is called without first calling the initialize method on a vertex, which would make the search erroneous. Our analysis shows that only 38% of the 156 configurations need monitoring and took 3.8 minutes to complete. It should be noted that 2.3 minutes out of the 3.8 minutes is actually the time taken to count the configurations described by the complex tracematch presence condition generated for this property, not to generate the presence condition itself. We believe that the long duration is due to the unoptimized implementation of our feature model solver, not due to the complexity of the new feature model with the tracematch presence condition.

`KruskalCheck` checks if the method that runs the Kruskal's algorithm returns an object that was not created in the control-flow of the method, which would mean that the algorithm is not functioning correctly. In 1.4 minutes, our analysis showed that only 8% of the GPL product line needed monitoring.

Notepad Table 2 shows the results for Notepad, which has 2074 LOC with 25 features and 144 configurations. Variations arise from permuting largely independent functionalities, such as saving/opening files, printing, and user interface support (e.g. menubar or toolbar). The analysis, for all safety properties, takes notably longer than that for `GPL` because Notepad uses the Java Swing framework, which heavily uses call-back methods that increase by large amounts the size of the call graph that our analysis needs to construct and to consider.

Table 2. Notepad Results

Lines of code	2074
No. of features	25
No. of configurations	144
PersistenceCheck	
No. of configurations	72 (50%)
Duration	331.8 sec. (5.5 min.)
CopyPasteCheck	
No. of configurations	64 (44%)
Duration	372.0 sec. (6.2 min.)
UndoRedoCheck	
No. of configurations	32 (22%)
Duration	304.5 sec. (5.1 min.)

`PersistenceCheck` checks if `java.io.File*` objects are created outside of persistence-related functions, which should not happen. In 5.5 minutes, our analysis reduces the configurations to monitor to 50%.

`CopyPasteCheck` checks if a paste can be performed without first performing a copy, an obvious error with the product line. In 6.2 minutes, our analysis reduces the configurations to monitor to 44% of the original number.

`UndoRedoCheck` checks if a redo can be performed without first performing an undo. In 5.1 minutes, our analysis reduces the configurations to monitor to 22%.

5.2 Effectiveness

Cost-Benefit Analysis. As the duration for each product-line/tracematch pair shows, our analysis introduces a small cost. Even for product lines considerably larger than our case studies, we believe that the duration will remain small. Most of the duration is from the weaving that is required to determine the required shadows and from constructing the inter-procedural call-graph that we then traverse to determine the presence conditions. The one-time cost of duration is worth incurring if it is less than the time it takes to test-run each saved configuration with complete path coverage (complete path coverage is required to see if a monitor can be triggered). Consider `Notepad` and `PersistenceCheck` pair, for which our technique is least effective as it takes the longest time, 4.65 seconds, to save a configuration (144 minus 64, or 80, configurations are saved in 372.0 seconds, or 4.65 seconds to save a configuration).

is least effective as it takes the longest time, 287.4 seconds (4.8 minutes), to achieve the smallest saving (72 configurations or 50%). In absolute numbers, the duration of the analysis is spread out to 4 seconds per configuration. The only way our technique would not be worth employing is if one could test-run a configuration of `Notepad` with complete path coverage in less than 4 seconds. Achieving the test-run within this time is unrealistic, especially in a UI-driven application like `Notepad`.

Ideal (Product-line, Tracematch) Pairs. Our technique works best for pairs where the tracematch can only be triggered on few configurations of the product

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

6 Related Work

To the best of our knowledge, we are the first to propose using static analysis to reduce the number of configurations to be monitored. Never-the-less, there are direct connections to prior work.

Statically Evaluating Monitors. Our work is most closely related to [6]. As mentioned in Section 1 and Section 2, this traditional static analysis is not suitable for product lines because it is oblivious to features. As mentioned in Section 3, the traditional static analysis proposes three stages of precision. Although we took only the first stage and extended it, there is no reason why the other stages cannot be extended in a similar fashion. Whether further optimization should be performed after running our technique remains an open question. Namely, it may be possible to take a configuration or a program that our technique has determined to require a monitor and apply the traditional program analysis [6] on it, which could possibly yield optimizations that wasn't possible in the more complex SysGen product line.

Testing Product Lines. The idea of reducing configurations originated from our work on product line testing [10], which determines features that are incremental, a notion similar to *sandboxing*, and uses this information to reduce the combinatorics in running a product line test. The two works are different both in setting and technique. In terms of setting, that work requires a test to be written as a unit test, i.e. a program that exercises a subset of the program under test, while this work requires a test to be written as a monitor against the program under test. In terms of technique, [10] employs a static analysis that checks if a feature does not alter the control-flow or data-flow of another feature, which is not sufficient for our work because such a sandboxed feature can still violate safety properties. Thus the two works are complementary.

Model-Checking Product Lines. Classen et al.[8] propose a feature-aware model-checking technique. The technique is similar in intent to ours: using the

authors' technique, programmers can apply model checking to a product line as a whole, instead of applying it to each program of the product line. The authors show, that in the common case this approach yields a far smaller complexity and therefore has the potential for speeding up the model-checking process. Classen et al. do not, however, model-check concrete product lines. Instead they assume a given abstraction of a product line, given as a *Feature Transition Systems (FTS)*, a linear transition system annotated with feature conditions. Because our technique works on *SysGen* and Java, we need to consider issues specific to Java such as the identification of relevant events, the weaving of the runtime monitor and the static computation of points-to information. Also, model checking answers a different question than our analysis: model checking can only tell whether or not a given program (or product line) may violate a given temporal property. Our analysis further reports a subset of instrumentation points (joinpoint shadows) that can, in combination, lead up to such a violation. As we showed in previous work [4], identifying such shadows requires more sophisticated algorithms than those that only focus on violation detection.

Safe Composition. [15] collects referential dependencies in a product line to determine the so-called "implementation constraints" that ensure that every feature combination produces a compilable program. Our work can be seen as a variant of safe composition, where a tracematch is treated as a feature itself that "references" its shadows in the product line and requires features that allow those shadows to be reached. Our analysis checks a much stronger property, i.e. reachability to the shadows, than syntactic presence checked by the existing safe composition technique. Also, collecting the referential dependencies is much more involved in our technique because it requires evaluting pointcuts that can have wildcards and control-flow constraints.

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

7 Conclusion

A product line enables systematic development of related programs, but it also introduces the challenge of analyzing its large number of programs, which can be exponential in the number of features. For safety properties that are enforced through an execution monitor, conventional wisdom tells us that every configuration must be monitored. In this paper, we presented a static analysis technique that minimizes the configurations on which an execution monitor has to be inserted. The technique determines the required instrumentation points and determines what features need to be present for those points to be reachable.

The execution monitor is inserted only on the configurations with such features. Our technique incurs a small overhead and achieves high reduction when a safety property crosscuts optional features.

As the importance of product lines grows, so too will the importance of analyzing and testing product lines, especially in a world where reliability and security are its first and foremost priorities. This paper takes one of the many steps needed to make analysis and testing of product lines an effective technology.

Acknowledgement. The work of Kim and Batory was supported by the NSF's Science of Design Project CCF 0724979 and NSERC Postgraduate Scholarship.

References

1. C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Adding trace matching with free variables to aspectj. In *OOPSLA*, pages 345–364, 2005.
2. D. Batory. Feature models, grammars, and propositional formulas. Technical Report TR-05-14, University of Texas at Austin, Texas, Mar. 2005.
3. E. Bodden. Clara: a framework for implementing hybrid typestate analyses. Technical Report Clara-2, 2009.
4. E. Bodden. Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. In *International Conference of Software Engineering (ICSE)*. ACM Press, 2010. To appear.
5. E. Bodden, F. Chen, and G. Rosu. Dependent advice: a general approach to optimizing history-based aspects. In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2009. ACM.
6. E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 525–549. Springer, 2007.
7. E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47, New York, NY, USA, 2008. ACM.
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. H. P. Kim. Enforcing safety properties in product lines: Tool and results. Available from `http://userweb.cs.utexas.edu/~chpkim/splmonitoring`, 2010.
10. C. H. P. Kim, D. Batory, and S. Khurshid. Reducing combinatorics in product line testing. Technical Report TR-10-02, University of Texas at Austin, Austin, Texas, USA, January 2010.
11. 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.

12. 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.
13. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
14. Software Engineering Institute, CMU. Software product lines. `http://www.sei.cmu.edu/productlines/`.
15. 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.