Reducing Configurations to Monitor in a Software Product Line

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 software *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, because a product line induces a number of programs that is potentially exponential in the number of features, it would be very expensive to use existing monitoring techniques: one would have to apply those techniques to every single program. Doing so would also be wasteful because many programs can provably never violate the stated property. We introduce a monitoring technique dedicated to product lines that, given a safety property, statically determines the feature combinations that cannot possibly violate the property, thus reducing the number of programs to monitor. Experiments show that our technique is effective, particularly for safety properties that crosscut many optional features.

1 Introduction

A *software product line ("SPL" or "product line" for short)* 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, SPLs help reduce both the time and cost of software development [17]. Unfortunately, SPLs also 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 SPLs for *safety property* [16] violation. We avoid monitoring every program of an SPL by statically identifying feature combinations (i.e., programs) that provably can never violate the stated property. These programs do not need to be monitored. Achieving this reduction is beneficial in at least two settings under which monitors are used. First, it can significantly speed up the testing process as these programs do not need to be run to see if the property can be violated. Second, if the

monitor is used in production, it can speed up these programs because they are not monitored unnecessarily.

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

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

To summarize, the contributions of this paper are:

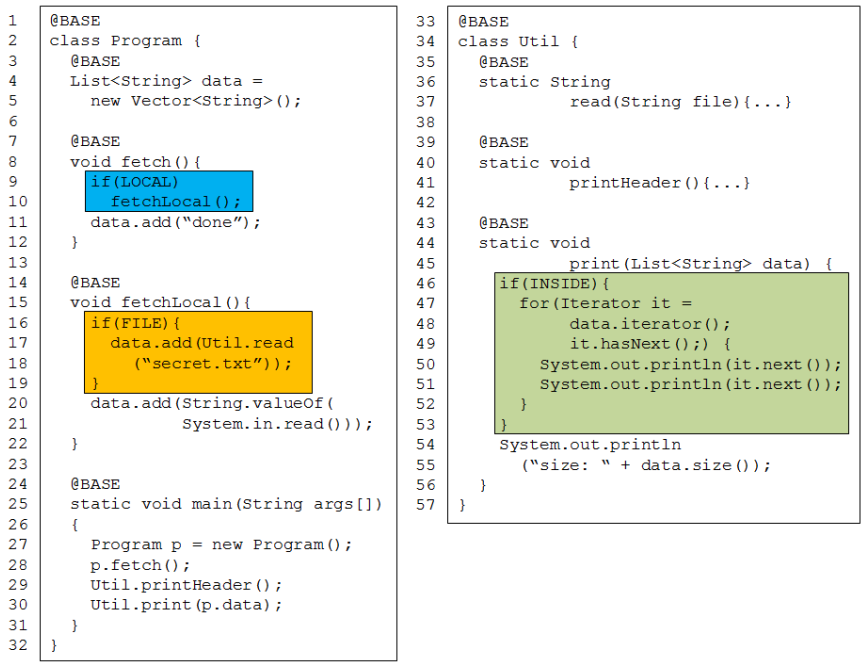
- A novel static analysis to determine, for a given SPL and runtime-monitor specification, the feature combinations (programs) that require monitoring,
- An implementation of this analysis within the CLARA framework for hybrid typestate analysis [4], as an extension to Bodden et al.'s earlier whole-program analysis [6], and
- Experiments that show that our analysis noticeably reduces the number of configurations that require runtime-monitoring and thus saves testing time and program execution time for the programs studied.

2 Motivating Example

Figure 1 shows a simple example SPL, whose programs fetch and print data. There are different ways of representing a product line. In this paper, we use the *SysGen program* representation [13], where an SPL is an ordinary Java program whose members are annotated with the name of the introducing feature 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. A program (also referred to as a *configuration* or *feature combination*) in SysGen is instantiated by assigning a Boolean value for each feature and statically evaluating feature-conditionals and feature-annotations.

Every SPL has a *feature model* [2] that defines the legal combinations of features. The feature model for our SPL is expressed below as a context-sensitive grammar. `Base` is a required feature. Optional features (`Inside`, `File`, and `Local`) are listed in brackets.

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



```
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 (second line). In the last line, the feature model enforces additional implementation constraints that must hold for all programs in the product line to compile. For example, `File implies Base` because the code of the `File` feature references `data` (line 17, Figure 1) that belongs to `Base` (lines 3-5, Figure 1). A technique described elsewhere [18] can generate these implementation constraints automatically. In total, the feature model allows seven distinct programs (eight variations from three optional features then remove the case without any optional feature).

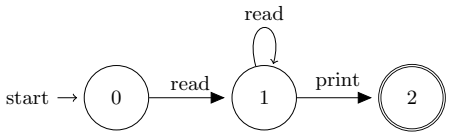
2.1 Example Monitor Specifications: ReadPrint and HasNext

Researchers have developed a multitude of specification formalisms for defining runtime monitors. As our approach extends the CLARA framework, it can generally apply to any runtime-monitoring approach that uses AspectJ aspects for monitoring. This includes popular systems such as JavaMOP [8] and tracematches [1]. For the remainder of this paper, we will use the tracematch 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` event follows one or more `read` events 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(``ReadPrint violation!'');
8      }
9    }
10 }
```

(a) ReadPrint Tracematch



(b) Finite-State Machine

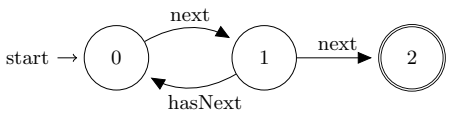
Fig. 2. ReadPrint Safety Property

Figure 3 shows another safety property, `HasNext` [6], which checks for iterators if `next()` is called twice without calling `hasNext()` in between. Note that this tracematch only matches if the two `next()` calls bind to the same `Iterator` object `i`, as shown in Figure 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 programs because these programs provably cannot trigger the runtime monitor.

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

(a) HasNext Tracematch



(b) Finite-state machine

Fig. 3. HasNext Safety Property [6]

2.2 Analysis by Example

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

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

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

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

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

2.3 The Need for a Dedicated Static Analysis for Product Lines

As mentioned earlier, there exist static analyses that improve the runtime performance of a monitor by reducing its instrumentation of a single program [5–7]. We will refer to these analyses as *traditional program analyses (TPA)*. There are two ways to apply such analyses to product lines. One way is inefficient, the other way imprecise. Running TPA against each instantiated program will be very inefficient because it will have to inspect every program of the product line separately. The other way is to run TPA against the product line itself. This is possible because a product line in a SysGen program representation can be treated as an ordinary program (recall that a SysGen program uses ordinary program constructs like if-conditionals, rather than pre-processor constructs like `#ifdef`s, to represent variability). However, this second way will be imprecise. For example, suppose we apply TPA on the `ReadPrint` and `HasNext` tracematches for our example SysGen program: both tracematches may match in the case in which all features are enabled. Being oblivious to the notion of features, the analysis will therefore report that the tracematches have to be present for every program of the product line. This shows that a static analysis, to be both efficient and effective on an SPL, has to be aware of the SPL's features.

3 Product Line Aware Static Analysis

Figure 4 displays an overview of our approach. First, for a tracematch, our analysis determines the symbols required for the tracematch to trigger ("Determine Required Symbols"). For each of these symbols, we use the aspect weaver to identify the statements that are matched ("Determine Symbol-To-Shadows"). We elaborate on these two steps in Section 3.1. Then, for each of the matched statements, we determine the feature combinations that allow the statement to be reachable from the program's `main()` method. This results in a set of *presence conditions*. We combine all these conditions to form the presence condition of the tracematch. We repeat the process for each tracematch ("Determine Presence Conditions") and add the tracematches and their presence conditions to the original feature model ("+"). We explain these steps in Section 3.2.

3.1 Required Symbols and Shadows

A safety property must be monitored for a feature configuration c if the code in c may drive the finite-state monitor from its initial state to its final (error) state. In earlier work [6], Bodden et al. described three different algorithms that try to determine, with increasing levels of detail, whether a single program can drive a monitor into an error state, and using which transition statements.

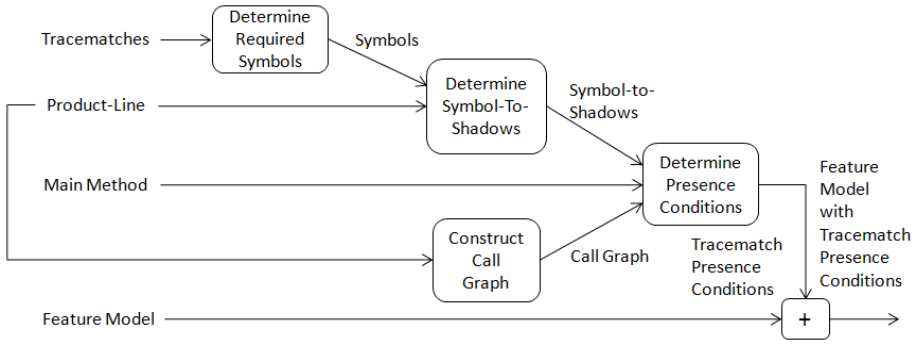


Fig. 4. Overview of Our Technique

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

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

Required Symbols A symbol represents a set of transition statements with the same label. Given a tracematch, we determine the *required symbols*, i.e., the symbols required to reach the error state, by fixing one symbol s at a time and checking whether removing all automaton edges labeled with s prevents the final state from being reached. For any given program p, if there exists a required symbol s for which p contains no s-transition, then p does not have to be monitored. For the `ReadPrint` property, the symbols `read` and `print` are required because without one of these, the final state in Figure 2(b) cannot be reached. For the `HasNext` property, only the symbol `next` is required. This is because one can reach the final state without seeing a `hasNext`-transition. If a tracematch has no required symbol, e.g. $a \mid b$ (either symbol will trigger the monitor, meaning that neither is required), it has to be inserted in all programs of the product line.[4]

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

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

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

3.2 Presence Conditions

A tracematch monitor must be inserted into a configuration when each of the tracematch's required symbols is present in the configuration. The *presence condition (PC)* of a tracematch is thus the conjunction of the presence condition of each of its required symbols. In turn, a symbol is present if any one of its shadows is present. Thus, the PC of a symbol is the disjunction of the PC of each of its shadows. The PC of a shadow is the conjunction of features that are needed for that shadow to appear in an SPL program. A first attempt to computing the PC of a tracematch is therefore:

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

For example, Figure 5 shows how we determine the PC of the `ReadPrint` tracematch. The required symbols of this tracematch are `read` and `print`. `read` has one shadow in line 17 of Figure 1 and `print` has two shadows in lines 29 and 30. For the shadow in line 17 to be syntactically present in a program, the `if(FILE)` conditional in line 16 must be `true` and the `fetchLocal()` method definition (annotated with `BASE` in line 14) must be present. That is, `pc(line17) = [File and Base]`. Similarly, `pc(line29)` and `pc(line30)` are each expanded into `[Base]` because each of the shadows just requires `BASE`, which introduces the `Program` class and its `main`-method definition.

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

Fig. 5. Computing ReadPrint's Presence Condition

The solution in Figure 5 is imprecise in that it allows configurations where a shadow is syntactically present, but not necessarily reachable from the `main` method. For example, according to the algorithm, the `read(..)` shadow (line 17) is "present" in configurations {`Base=true`, `Local=false`, `File=true`, `Inside=DONT_CARE`} even though it is not reachable from `main` due to `Local` being turned off. Based on this observation, the algorithm that we implemented can take into account the shadow's callers in addition to its syntactic containers. The algorithm therefore conjoins a shadow's imprecise PC with the disjunction of precise PC of each of its callers, recursively. For the line 17 shadow, which is called by line 10, which is in turn called by line 28, this precise algorithm would return:

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

Substituting this in Figure 5, we get `ReadPrint iff (File and Local and Base)`, which is optimal for our example and, as mentioned in Section 2.2, is what we set out to construct. Similarly, `HasNext`'s presence condition is:

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

Note that, even though `HasNext` is more localized than `ReadPrint`, i.e., in one optional feature (`Inside`) as opposed to two optional features (`File` and `Local`), it is required in more configurations (4 out of 7) than `ReadPrint` (2 out of 7).[5] This is because the feature model allows fewer configurations with both `Local=true` and `File=true` than configurations with just `Inside=true`.

There may be shadows that can only be reached through a cyclic edge in a call-graph. Rather than including the features controlling the cyclic edge in the presence condition of such a shadow, for simplicity, we ignore the cyclic edge. This is not optimally precise but sound. For example, `Util.read(..)` call in Figure 6 is actually only present in an execution if the execution traverses the cyclic edge from `c()` to `a()`, which is possible only if `X=true`. Instead of adding this constraint on `X` to the presence condition of `Util.read(..)`, we simply insert the monitor for both values of `X`.

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

While considering the callers of a shadow makes its presence condition more precise, doing so is entirely optional for the following reason: without considering the callers, a shadow will simply be considered to exist both when a caller is

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

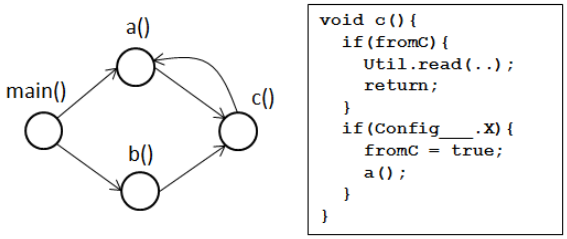


Fig. 6. Example of Computing a Presence Condition with Cycles in the Call-Graph

present and when a caller is not present, which will insert a monitor even if a required symbol's shadow cannot be reached. For example, it would be sound, although not optimally precise, to return the imprecise presence condition of the shadow at line 17. But users of our approach can even go beyond that. Our analysis is pessimistic, i.e., starts from a sound but imprecise answer that ignores the call graph and then gradually refines the answer by inspecting the call graph. Therefore, our analysis can report a sound intermediate result at any time and after a certain number of call sites have been considered, we can simply stop going farther in the call-graph, trading precision for less computation time and resources. Being able to choose the degree of precision is useful especially because the call graph can be very large, which can make computing the presence condition expensive both time-wise and memory-wise. Our technique works with any kind of call graph. In our evaluation, we found that even simple context-insensitive call graphs constructed from *Spark* [14] are sufficient.

4 Evaluation

We implemented our analysis as an extension of the CLARA framework for hybrid typestate analysis [4] and evaluated it on the following SPLs: *Graph Product Line (GPL)*, a set of programs that implement different graph algorithms [15] and *Notepad*, a Java Swing application with functionality similar to Windows Notepad. We considered three safety properties for each SPL. For each property, we report the number of configurations on which the property has to be monitored and the time taken (duration) to derive the tracematch presence condition. We ran our tool on a Windows 7 machine with Intel Core2 Duo CPU with 2.2 GHz and 1024 MB as the maximum heap size.

Note that, although the product lines were created in-house, they were created long before this 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 [12].

4.1 Case Studies

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

Table 1. Graph Product Line (GPL) Results

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

The `DisplayCheck` safety property checks if the method for displaying a vertex is called outside of the control flow of the method for displaying a graph: a behavioral API violation. Instead of monitoring all 156 configurations, our analysis reveals that only 55 configurations, or 35% of 156, need monitoring. The analysis took 1.2 minutes to complete. The tracematch presence condition that represents these configurations is available on our website [12].

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

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

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

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

Table 2. Notepad Results

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

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

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

4.2 Discussion

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

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

Ideal (Product Line, Tracematch) Pairs. Our technique works best for pairs where the tracematch can only be triggered on few configurations of the product line. Ideally, a tracematch would crosscut many optional features or touch one feature that is present in very few configurations. This is evident in the running example, where the saving for `ReadPrint`, which requires two optional features, is greater than that for `HasNext`, which requires one optional feature. It is also evident in the case studies, where `KruskalCheck` and `UndoRedoCheck`, which are localized in a small number of features but requires other

features due to the feature model, see better saving than their counterparts. Without any constraint, a tracematch requiring x optional features needs to be inserted on $1/(2^x)$ of the configurations (`PersistenceCheck` requires one optional feature, hence the 50% reduction). A general safety property, such as one involving library data structures and algorithms, is likely to be applicable to many configurations of a product line (if a required feature uses it, then it must be inserted in all configurations) and thus may not enable our technique to eliminate many configurations. On the other hand, a safety property crosscutting many optional features makes an ideal candidate.

5 Related Work

Statically Evaluating Monitors. Our work is most closely related to [6]. As mentioned in Section 2.3, this traditional static analysis is not suitable for product lines because it is oblivious to features. As mentioned in Section 3.1, 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 on it, which could yield optimizations that were not possible in the SysGen program.

Testing Product Lines. The idea of reducing configurations for product line monitoring originated from our work on product line testing [13], which finds "sandboxed" features, i.e. features that do not modify other features' control-flow or data-flow, and treats such features as don't-cares to determine configurations that are identical from the test's perspective. But the two works are different both in setting and technique. In setting, in [13], only one of the identical configurations needs to be tested. In this paper, even if a hundred configurations are identical in the way they trigger a monitor (e.g. through the same feature), all hundred configurations need to be monitored because all hundred can be used by the end-user. In testing mode, it would be possible to run just one of the hundred configurations if our technique could determine that the configurations are identical in the way they trigger the monitor. However, this would require a considerably more sophisticated analysis and is beyond the scope of this paper. In technique, the static analysis employed in [13] is not suitable for our work because a sandboxed feature can still violate safety properties and cause a monitor to trigger. Thus the two works are complementary.

Model-Checking Product Lines. Works in model-checking product lines [9, 10] are similar in intent to ours: using these techniques, programmers can apply model checking to a product line as a whole, instead of applying it to each program of the product line. In the common case, these approaches yield a far smaller complexity and therefore have the potential for speeding up the model-checking process. However, these approaches do not model-check concrete product lines. Instead, they assume a given abstraction, such as a transition sys-

tem, of a product line. Because our technique works on *SysGen* and Java, we need to consider issues specific to Java such as the identification of relevant events, the weaving of the runtime monitor and the static computation of points-to information. Also, model-checking answers a different question than our analysis: model-checking a product line can only report the configurations that may violate the given temporal property. Our analysis further reports a subset of instrumentation points (joinpoint shadows) that can, in combination, lead up to such a violation. As we showed in previous work [3], identifying such shadows requires more sophisticated algorithms than those that only focus on violation detection.

Safe Composition. [18, 11] collect implementation constraints in a product line that ensure that every feature combination is compilable or type-safe. Our work can be seen as a variant of safe composition, where a tracematch is treated as a feature itself that "references" its shadows in the product line and requires features that allow those shadows to be reached. However, our analysis checks a much stronger property, i.e. reachability to the shadows, than syntactic presence checked by the existing safe composition techniques. Also, collecting the referential dependencies is much more involved in our technique because it requires evaluating pointcuts that can have wildcards and control-flow constraints.

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

6 Conclusion

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

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. The work of Bodden was supported by CASED (www.cased.de). The work of Khurshid was supported by NSF CCF-0845628 and IIS-0438967.

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. Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. In *ICSE 2010*. ACM Press.
4. E. Bodden. Clara: a framework for implementing hybrid typestate analyses. Technical Report Clara-2. Available from http://www.bodden.de/pubs/tr-clara-2.pdf, 2009.
5. E. Bodden, F. Chen, and G. Rosu. Dependent advice: a general approach to optimizing history-based aspects. In *AOSD 2009*. ACM.
6. E. Bodden, L. J. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. In *ECOOP 2007*.
7. E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *SIGSOFT 2008/FSE-16*. ACM.
8. F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *OOPSLA 2007*, pages 569–588. ACM Press.
9. 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. In *ICSE 2010*. IEEE.
10. A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *FMOODS 2008*, pages 113–131. Springer-Verlag.
11. C. Kästner and S. Apel. Type-checking software product lines - a formal approach. In *Automated Software Engineering (ASE)*, 2008.
12. C. H. P. Kim. Reducing Configurations to Monitor in a Software Product Line: Tool and Results. Available from `http://userweb.cs.utexas.edu/~chpkim/splmonitoring`, 2010.
13. 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, January 2010. Available from `http://userweb.cs.utexas.edu/~chpkim/chpkim-productline-testing.pdf`.
14. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *Compiler Construction*, volume 2622 of *LNCS*, pages 153–169. Springer, 2003.
15. 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.
16. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.
17. Software Engineering Institute, CMU. Software product lines. `http://www.sei.cmu.edu/productlines/`.
18. 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.