Toward Variability-Aware Testing

Christian Kästner
Philipps University Marburg

Alexander von Rhein
University of Passau

Sebastian Erdweg and Jonas Pusch
Philipps University Marburg

Sven Apel
University of Passau

Tillmann Rendel and Klaus Ostermann
Philipps University Marburg

ABSTRACT

We investigate how to execute a unit test for all products of a product line without generating each product in isolation in a brute-force fashion. Learning from variability-aware analyses, we (a) design and implement a variability-aware interpreter and, alternatively, (b) reencode variability of the product line to simulate the test cases with a model checker. The interpreter internally reasons about variability, executing paths not affected by variability only once for the whole product line. The model checker achieves similar results by reusing powerful off-the-shelf analyses. We experimented with a prototype implementation for each strategy. We compare both strategies and discuss trade-offs and future directions. In the long run, we aim at finding an efficient testing approach that can be applied to entire product lines with millions of products.

1. INTRODUCTION

Analysis of software product lines has attracted much attention by researchers [26]. The addressed key problem is that traditional analysis methods (type checking, static analysis, model checking, testing, and so forth) target only individual programs, whereas a product line with n optional compile-time features gives rise to $\mathcal{O}(2^n)$ distinct configurations, and thus $\mathcal{O}(2^n)$ distinct products. Traditionally, obtaining an analysis result for the entire product line (e.g., whether every product is well typed) would require to analyze each product in isolation, in a brute-force fashion. Since a brute-force approach does not scale due to the huge configuration space, practitioners resort to sampling strategies [5,20–22]: They analyze only a few products currently produced, they analyze a few randomly selected products, or they analyze a relatively small number of products selected by some coverage criterion, such as t-way feature coverage. However, sampling cannot yield reliable analysis results for the entire product line.

Recently, researchers have investigated alternative strategies to analyze entire product lines without looking at the generated code of each product. We call analyses following these strategies *variability-aware analysis* (or family-based analysis [26]), because they take the variability of the product-line implementation into account during analysis. Roughly speaking, the idea is to analyze a generator (the product-line implementation itself together with configuration knowledge) instead of analyzing the generated products. Variability-aware analysis exploits the fact that products in a product line typically are generated from a common code base and share a significant amount of common code [10, 22]. When using brute force or sampling, this common code is analyzed repeatedly. In contrast, variability-aware analyses usually perform analysis on common code only once, while only variable code that actually affects the analysis result causes additional effort.

Researchers have successfully developed variability-aware analyses for parsing, type checking, model checking, static analysis, and theorem proving (see Sec. 5). Although *testing* of product lines has received significant attention, researchers have concentrated on sampling strategies [5, 20, 21], on test suite reduction [15, 24], and on test generation [24, 28]. In all these approaches, though, individual tests are still executed on generated products, one by one. To the best of our knowledge, there is no notion of *variability-aware test execution*, where a test is run on an entire product line without generating individual products.

Our goal is to transfer experience from existing variability-aware analyses to product-line testing. *We want to execute a test case (e.g., a unit test) in all configurations of a product line, without actually generating a product for each configuration.* In this workshop paper, we explore early steps in this direction. In line with extended mechanisms used in variability-aware analyses, we build a *variability-aware interpreter* to execute a test case in all configurations of a product line in parallel (which resembles mixed concrete/symbolic execution). Additionally, we explore an alternative strategy based on *variability encodings* and off-the-shelf analysis tools, in our case, JavaPathfinder (JPF) [29] and the extension *jpf-bdd* [30].

Specifically, our contributions are: We generalize strategies to implement variability-aware analyses into white-box and black-box strategies, which was only implicit in prior work. We design and implement a variability-aware interpreter for a WHILE language (white box). We apply JPF for variability-aware testing (black box). Finally, while we cannot yet make claims about scalability to real-world problems, we discuss trade-offs and limitations, and we outline research directions.

We want to encourage researchers to investigate testing of whole product lines without the usual sampling strategies. We are still in an early exploration stage toward *variability-aware testing*. Here, we present initial ideas and early experiences with prototypes and cases studies. We appreciate any feedback and ideas.

2. VARIABILITY-AWARE ANALYSIS

Before we discuss test-case execution in product lines, we briefly introduce variability-aware analysis in general, from which we then adopt many concepts. We start with the general goal, outline how we represent variability, and discuss two common implementation

FOSD'12, September 24–25, 2012, Dresden, Germany.

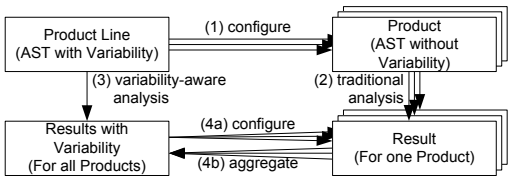

Figure 1: Variability-aware vs. brute-force analysis

```
1  case class Opt[T](pc: FeatureExpr, value: T)
2
3  abstract class Cond[T]
4  case class One[T](value: T) extends Cond[T]
5  case class Choice[T](pc: FeatureExpr, a: Cond[T], b: Cond[T])
       extends Cond[T]
6
7  def condFlatMap[T, U](a: Cond[T], vctx: FeatureExpr,
8        fun: (FeatureExpr, T) => Cond[U]): Cond[U] = a match {
9      case One(t) => fun(vctx, t)
10     case Choice(pc, a, b) =>
11         Choice(pc, condFlatMap(a, vctx∧pc, fun),
12                    condFlatMap(b, vctx∧¬pc, fun))
13 }
```

Figure 2: Variability structures and core utility functions implemented with Scala

strategies.

We can explain variability-aware analysis with the process pattern illustrated in Figure 1. Instead of repeatedly generating a product (Step 1) and analyzing each product with a traditional analysis (Step 2), we want to analyze the entire product line without generating individual products (Step 3). Variability-aware analysis should produce a result that describes the entire product line. The result explains in which configuration which specific property holds (e.g., "all configurations with feature *FOO* are ill typed, all other configurations are well typed"). From this analysis result, we are able to deduce the properties that we would establish for an individual product with the traditional analysis (Step 4a). Alternatively, by applying the traditional analysis in a brute-force fashion to all products, we could aggregate the individual properties to describe the entire product line (Step 4b). While the output should be equivalent, we expect the variability-aware analysis (Step 3) to be much faster than the brute-force strategy (repeating Steps 1, 2, and 4b). In this paper, we want to apply this concept also to testing.

2.1 Variability representation

To perform variability-aware analysis, we need a structural representation of the product-line implementation that contains all compile-time variability. In our work, we encode compile-time variability directly in abstract syntax trees (ASTs) with presence conditions. A *presence condition* is a propositional formula over features of the product line that yields *true* iff the AST element (i.e, the corresponding code fragment) should be included in the product for a given configuration.

We manage variability with two constructs, as illustrated with Scala code in Figure 2: First, program elements can be optional (`Opt[T]` for elements of type `T`). An optional element is guarded by a propositional presence condition, which is represented by type `FeatureExpr`. Second, type `Cond[T]` encodes conditional elements, that is, elements that differ between configurations. We have either one element (`One[T]`) or a choice between two elements (`Choice[T]`) depending on a presence condition. Since choices can be nested,

```
1  abstract class Stmt
2  case class Block(s: List[Opt[Stmt]]) extends Stmt
3  case class Assign(n: String, e: Cond[Expr]) extends Stmt
4  case class If(e: Cond[Expr], s: Stmt) extends Stmt
5  case class While(e: Cond[Expr], s: Stmt) extends Stmt
6
7  abstract class Expr
8  case class Var(name: String) extends Expr
9  case class Lit(value: Int) extends Expr
10 ...
```

Figure 3: Abstract syntax of a WHILE language with variability implemented in Scala

```
1  a = 1;
2  #ifdef FOO
3  b = true;
4  #endif
5  if (a < 3)
6    a = a + 1;
7  #ifndef FOO
8  b = false;
9  #endif
10 if (b)
11   a = 0;
12 #ifdef BAR
13 a = 0;
14 #endif
15
16 b =
17 #if BAR || FOO
18   true
19 #else
20   false
21 #endif
22 ;
```

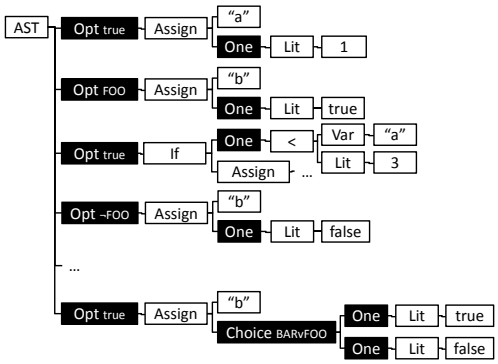

Figure 4: Example program with variability and corresponding AST (choice nodes shown with black background color)

we can express multiple alternative elements. For example, we can express that variable `v` has value 1 if feature X is selected, and value 2 if feature Y but not Z is selected, and value -1 in all other cases: `v=Choice(X, One(1), Choice(Y∧¬Z, One(2), One(-1)))`.[1] Optional elements are typically used inside lists when $0..n$ elements are supported (e.g., a list of optional statements can contain no, one, or multiple statements in each configuration), whereas conditional elements are used when exactly one element is required in each configuration (e.g., an assignment always has exactly one right-hand-side expression).

Using `Opt` and `Cond`, we can express variability directly in the declaration of abstract syntax, as illustrated with the WHILE language in Figure 3 (the WHILE language is a small but Turing-complete imperative language, standard in static-analysis research). To create an AST with variability from source code with #ifdef directives, we use our variability-aware TypeChef parser [14]. We show an example WHILE program that contains variability in the form of preprocessor directives and the corresponding AST with variability in Figure 4.

Based on our AST representation with variability, we can realize variability-aware analyses for entire product lines, including the interpreter we present in Section 3.

[1] Our current implementation allows arbitrary propositional formulas in choice nodes and uses a SAT solver to reason about variability. Instead of choice trees, we could alternatively store lists of optional entries, or encode conditional values similar to Boolean decision diagrams, or experiment with other representations, such as the Choice calculus [11].

2.2 Granularity, locality, and sharing

When specifying the abstract syntax of a language, we can decide where to inject variability in the AST. We can support variability at different levels of granularity, for example, allow conditional expressions inside assignments or merely allow optional elements at the statement level. We can always replace a fine-grained variability representation with a coarse-grained one at the cost of replication [11]. Usually, fine-grained granularity facilitates more sharing—sharing which we can potentially exploit to reduce analysis effort.

A key insight for variability-aware analysis is that, in all analysis steps, we want to keep variability as local as possible, to facilitate as much sharing as possible. For example, it is usually more efficient to store a map from names to conditional values than to store conditional maps from names to values: If we want to change a value in a single configuration in a representation of type `Cond[Map[A,B]]`, we would need to copy the entire map, whereas changing a value in representation `Map[A,Cond[B]]` has a local effect and preserves sharing for all other values.

2.3 White-box vs. black-box strategy

Researchers have explored different strategies for variability-aware analysis. We observed that two general implementation strategies emerge, which we call henceforth white-box and the black-box strategy. Note that these terms are orthogonal to white-box vs. black-box testing to describe tests with and without source code (we do only white-box testing), but they refer to how analysis is performed and implemented.

White-box strategy. One common strategy is to *extend* the internal algorithm and data structures of the analysis. The modified analysis works on a representation with explicit variability, such as the ASTs presented above. It reasons about variability in all steps of the analysis and keeps variability local. Since we need to understand and modify the internals of the analysis, we name the strategy the *white-box strategy*.

For example, most variability-aware type checkers described in the literature follow the white-box strategy [1, 7, 13, 25]. Such a variability-aware type checker takes an AST with explicit variability information and exploits variability during analysis. The type checker knows in which configurations (described by a presence condition) a method is declared, and may even reason about conditional types of an expression. The analysis returns a list of conditional type errors, describing exactly in which configurations each error occurs.

In a white-box strategy, we *extend* the analysis to reason about variability. We perform analysis on shared code only once and only split analysis where variability actually occurs locally (*late splitting*). Also, when the analysis yields the same subresult in different configurations, the remaining analysis may be performed only once on the common result (*early joining*). We present a variability-aware interpreter using the white-box strategy in Section 3.

Black-box strategy. The white-box strategy has the disadvantage that we need to modify an existing analysis (usually in a fundamental and crosscutting way, affecting interfaces and internal data structures). Several researchers have investigated how to use existing analyses out of the box instead [2, 23, 27]. They rewrite the product-line implementation or rephrase the specification such that it can be analyzed as a whole with an existing off-the-shelf tool. Typically, we need a powerful existing analysis (such as model checking) that can already deal with some form of variation. Since the analysis tool is reused as is, we name the strategy the *black-box strategy*.

A typical example of the black-box strategy is to encode an analysis as specification for a model checker. Since model checkers are already capable of dealing with different values of variables, we can encode compile-time variability (as the #ifdef variability from Figure 4 or the `Cond` and `Opt` elements in our AST) using normal control-flow mechanisms of the host language (as *if* statements). A model-checking tool then explores all feasible program paths (covering the paths of all configurations). As we encode compile-time variability merely as additional run-time paths, the model checker is able to reason about all configurations. If the model checker detects a violation of the specification, we can reconstruct the erroneous configuration from the problematic execution path. The efficiency of the approach depends on the efficiency of the reused analysis. Modern model checkers already contain sophisticated mechanisms to deal with variations and many paths.

After introducing the basic strategies, let us adapt them for variability-aware testing, first using a white-box strategy (Sec. 3), then with a black-box strategy (Sec. 4).

3. WHITE BOX: A VARIABILITY-AWARE INTERPRETER

As a first attempt to perform variability-aware testing, we implemented an interpreter that is explicitly aware of variability and represents variability locally in its data structures (white-box strategy). For implementing the interpreter, we adopt patterns from prior white-box variability-aware analyses.

A traditional textbook interpreter takes a code fragment, in the form of an AST (without variability), as well as a store; executes the code fragment; and returns an updated store with all variable assignments. In contrast, our variability-aware interpreter takes an AST with variability, a variability context, and a variable store; executes it (covering the entire configuration space); and returns an updated variable store. Let us go through these ingredients one by one:

- *AST with variability.* We execute programs and program fragments given as ASTs with variability, as described in Section 2.1.
- *Variability context.* The variability context (`vctx`) describes which part of the configuration space we are currently executing. Like presence conditions, we represent the variability context with a propositional formula. For example, *true* means that we are analyzing all configurations, and $X \vee Y$ means that we are analyzing all configurations in which feature *X* or feature *Y* is selected. If the variability context is not satisfiable, we do not need to execute that code fragment, because it cannot occur in any configuration. Typically, we aim at executing code within a large variability context (describing many products).
- *Variable store.* Where a traditional store maps names to values (`Map[String,Value]`), a variable store maps names to conditional values (`Map[String,Cond[Value]]`; so a variable can have different values in different configurations). We store variability as local as possible (cf. Sec. 2.2). If we were dealing with more complicated values, such as objects or functions, we would incorporate variability into the value representation, for example, fields of an object would store conditional values. We show the implementation of our variable store and corresponding access functions in Figure 5 (top).

3.1 Implementation

In Figure 5, we sketch a Scala implementation of our variability-aware interpreter. For illustration, we also show three example traces in Figure 6.

First, the interpreter does not perform any computation if the variability context is not satisfiable, as determined with a SAT solver (Line 10).

```
1  type Store = Map[String, Cond[Value]]
2  def updateStore(store: Store, vctx: FeatureExpr,
3                  n: String, v: Cond[Value]): Store =
4      store + (n -> Choice(vctx, v,
5          store.getOrElse(n, One(VUndefined())) ).simplify)
6  def lookupStore(store: Store, n: String): Cond[Value] =
7      store.getOrElse(n, One(VUndefined()))
```

```
8  def executeStatement(stmt: Stmt, vctx: FeatureExpr,
9                       store: Store): Store =
10   if (!vctx.isSatisfiable()) store else stmt match {
11     case Assign(n, e) =>
12         val rhs: Cond[Value] = evalExpr(e, vctx, store)
13         return updateStore(store, vctx, n, rhs)
14     case Block(stmts) =>
15         for (Opt(fs, stmt) <- stmts)
16             store = executeStatement(stmt, vctx∧fs, store)
17         return store
18     case If(e, s) =>
19         val exprValue: Cond[Value] = evalExpr(e, vctx, store)
20         val x: FeatureExpr = whenTrue(exprValue)
21         return executeStatement(s, vctx∧x, store)
22     case While(e, s) =>
23         var exprValue: Cond[Value] = evalExpr(e, vctx, store)
24         var x: FeatureExpr = whenTrue(exprValue)
25         while (x.isSatisfiable()) {
26             store = executeStatement(s, vctx∧x, store)
27             exprValue = evalExpr(e, vctx, store);
28             x = whenTrue(exprValue)
29         }
30         return store
31 }
32 def whenTrue(v: Cond[Value]): FeatureExpr = v match {
33     case One(VInt(v)) if (v!=0) => True
34     case One(_) => False
35     case Choice(f, a, b) => (f∧whenTrue(a))∨(¬f∧whenTrue(b))
36 }
37 def evalExpr(ce: Cond[Expr], vctx: FeatureExpr,
38              store: Store): Cond[Value] =
39     condFlatMap(ce, vctx, (f, e) => evalExpr(e, f, store))
40 def evalExpr(e: Expr, vctx: FeatureExpr,
41              store: Store): Cond[Value] = e match {
42     case Var(n) => lookupStore(store, n)
43     case Int(v) => One(VInt(v))
44     case Neg(e) => condFlatMap(evalExpr(e, vctx, store), vctx,
45                         { case (_, VInt(v)) => One(VInt(-v)) })
46     ...
47 }
```

Figure 5: Variability-aware interpreter for the WHILE language, encoding variability in all execution steps (excerpt)

When interpreting an assignment (Line 11), we first evaluate the expression to a conditional value in the current variability context, then we store the value. If we execute the statement only in a restricted variability context, we also only store the value in that context.

The case for block statements (Line 14ff) illustrates how we restrict the variability context on optional statements. We execute each statement with a variability context restricted by the presence condition `fs` of that statement. If the statement has presence condition *true*, the variability context remains unchanged.

To evaluate a conditional expression (Lines 37ff), we evaluate every alternative expression separately in the corresponding variability context (Line 39; using auxiliary function `condFlatMap` defined in Figure 2). Variables are simply looked up in the store (Line 42), negations are applied to all alternative values (Line 44; also using auxiliary function `condFlatMap`). Notice, how we map over conditional values to preserve potential variability; if the AST does not contain variability, the interpreter behaves like a traditional interpreter.

As a novel concept, we use auxiliary function *whenTrue* when executing *if* and *while* statements (Lines 18–30). First, we evaluate the expression to a conditional value. Now, we need to decide when to execute the body. We want to execute it in all configurations in which the expression's value is *true*, but only once. To this end, with *whenTrue*, we determine a presence condition describing in which configurations the value is true. Subsequently, we execute the body only in the restricted variability context of those configurations in which the expression is *true*. Note that if the expression's value is *false* in all configurations, *whenTrue* will also return an unsatisfiable variability context *false*, so the body is never actually executed (Line 10).

Finally, the variability context makes it straightforward to deal with external specifications of valid feature combinations, as typically described in a *variability model*. We specify valid configurations as a propositional formula and simply pass the formula as the outermost variability context. As a consequence, the algorithm will not execute code related only to invalid feature combinations.

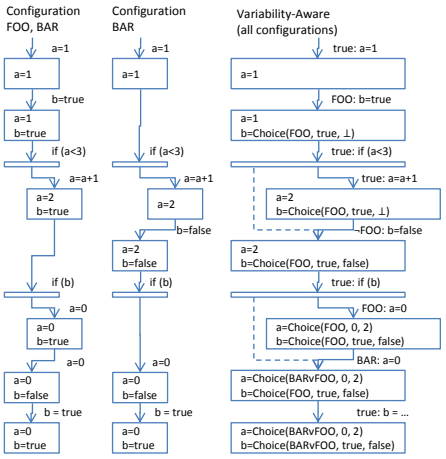

Figure 6: Trace of the example of Figure 4 for two configurations (left and middle) and variability aware (right). Indentation denotes scope; the edge labels denote the variability context; unchanged stores omitted

3.2 Discussion

As many existing white-box variability-aware analyses, our interpreter incorporates variability locally in internal data structures (e.g., the store and intermediate values), which facilitates late splitting and early joining (cf. Sec. 2.3).

First, as long as possible, we execute the program with a single variability context, even in conditionals and loops. We split the execution late, only when we actually encounter variability locally in the AST or store. In our example in Figure 4, we execute the first statements only once, even after conditional assignments in Lines 3 and 8, as long as those assigned values are not used. In Figure 6, we see that we never execute any statement of our example twice. In contrast, with a brute-force strategy, we would first generate all products and then execute the initial statements in every product. The local representation of variability ensures that we reason about variability only for variables that actually have different values.

Furthermore, we can *join* intermediate results (with auxiliary function `simplify`, not shown). For example, when we assign 0 to `a` again in Line 13 (Fig. 4), we store only distinct values of *a* and their corresponding conditions (i.e., we simplify `Choice(BAR, 0, Choice(FOO, 0, 2))` to `Choice(BAR∨FOO, 0, 2)`). If the variable is assigned to the same value in all configurations, we can join the

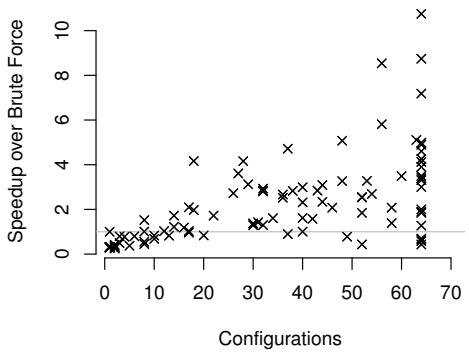

Figure 7: Speed-up of the variability-aware interpreter over a brute-force approach on 100 small, generated product lines (break even at the gray line)

```
1  bool FOO = randomBoolean(), BAR = randomBoolean();
2  int a; bool b;
3  a = 1;
4  if (FOO)
5    b = true;
6  if (a < 3)
7    a = a + 1;
8  if (!FOO)
9    b = false;
10 if (b)
11   a = 0;
12 if (BAR)
13   a = 0;
14 b = (BAR || FOO ? true : false);
15 a = 100 / a;
```

Figure 8: Code example with variability encoding

intermediate result and store only the single value. Joining can reduce effort in subsequent computations, but executing the join also requires computation effort, so there is a trade-off. However, we leave an empirical evaluation of how relevant joins are in practice for future work.

We have not explored limitations in detail yet. While reflection seems conceptually possible to support (operating on the variable structure of the program), I/O poses a problem. If we cannot provide a variability-aware test environment, we me might need to perform testing sequentially from the first occurrence of I/O. The WHILE language does not support I/O; hence, we leave also this problem for future work.

3.3 Experience

We have implemented a variability-aware interpreter for the WHILE language, with additional support for procedures. We can parse WHILE programs with preprocessor directives, like those in Figure 4, using the TypeChef variability-aware parser framework [14]. We are using this implementation to experiment with different strategies (e.g., granularity, different variability representation, when to attempt to join results), and to get a better understanding of which kinds of product-line implementations can be executed quickly and for which the execution resembles the brute-force approach (or is even slower due to the additional SAT solving).

We have developed a generator for random product lines written in the WHILE language and have implemented a testing framework following the pattern outlined in Figure 1. We generate all distinct products from our product line and compare the result of interpreting them without variability to the result of our variability-aware interpreter. Specifically, we do not generate unit tests, but, in a form of differential testing, we simply compare the stores following the equivalence in Figure 1 (4a, 4b). In Figure 7, we show how the variability-aware interpreter improves performance over the brute force approach for 100 generated product lines with at most 6 features (for larger product lines, we were unable to reliably generate random products that terminate, we leave this for future work). Absolute times are within few milliseconds; we gathered times as average from three runs. We can see an overhead for the variability-aware interpreter, but also that it mostly outperforms the brute-force analysis as the product-line size increases.

The implementation, which we currently extend with functions and objects, is available together with the test framework at `https://github.com/puschj/Variability-Aware-Interpreter`.

4. BLACK BOX: VARIABILITY ENCODING

In addition to implementing a variability-aware interpreter from scratch, we also experimented with performing variability-aware testing with existing tools (black-box strategy). We encoded variability such that we can use an off-the-shelf model checker—*JavaPathfinder* (JPF) and its extension *jpf-bdd* [30] in our case—to run test cases for all configurations. We use the model checker to execute the program paths of all valid configurations. This corresponds to seperate testing of all configurations in the brute-force approach.

Since model checkers are already capable of dealing with different values of variables, we encode compile-time variability using normal control-flow mechanisms of the host language. For example, we rewrite the code from Figure 4 as shown in Figure 8 (Lines 1–14). We replace preprocessor macros with global Boolean variables (called feature variables; non-deterministically initialized) and #ifdef directives with *if* statements or conditional expressions. Such rewrites can be performed mechanically; then, we can proceed with an existing analysis on traditional ASTs without variability. In the general case, the encoding can be trickier, but it is always possible to encode alternatives by renaming or code replication at statement level, as explored elsewhere [2, 13, 27]. Even a variability model can be encoded [2, 27]. We call the rewritten product a *product-line simulator* (a.k.a. meta-product [27]).

After this rewrite, we use JPF to execute test cases. Where the test case on a single product would run deterministically, we introduce nondeterminism through feature variables. Still, JPF explores all feasible program paths of the simulator and gives warnings if one of the paths would result in runtime errors. To illustrate this behavior, we introduced a division-by-zero bug that only occurs when features *FOO* or *BAR* are selected (Fig. 8, Line 15). The model checker finds this bug in paths that assign *true* to *FOO* or *BAR*.

Using a model checker for the verification of the simulator is rewarding, because in model checkers "unknown" values for variables are a common concept and model checkers provide out-of-the-box support. However, by using model checking, we limit the set of product lines that can be verified with the approach. For example, we are not able to verify product lines that contain (potentially) endless loops, need user interaction, or need file or network access. For most of these issues, there is advanced research, but we leave those for future work.

4.1 Gray-box extensions: jpf-bdd

Using an off-the-shelf model checker, such as JPF, ensures that errors in all configurations are found. However, in its standard configuration, JPF does not take advantage of the variability information in the product simulator. In the white-box approach, we knew that

variability was always expressed in propositional formulas, and we could reason about it with SAT solvers and attempt joins. Ideally, also for model checking, we want an exploration strategy that executes a path until it encounters variability; then it should split the path, execute both alternatives, and join the paths again as soon as possible. In the standard configuration, JPF splits paths quite early (when the variable is assigned to the "unknown" value). Also, standard JPF never joins paths after variability-related splits, because, once it has chosen a value for a feature variable, that value is part of the program state. Because each path has a different choice of feature values, all paths have at least one difference in their states, and different states can never be joined. That is, we split late, but we never join. In the worst case, this results in one execution path per configuration, much like in the brute-force approach.

Fortunately, JPF is extensible. For product-line verification, we developed *jpf-bdd* [30], which enables joining by separating feature variables from the remaining program state. Feature variables are stored in separate binary decision diagrams (BDDs). Because the program states do not contain the feature values any more, JPF can split paths later and join more states (the extension joins the BDDs accordingly), so potentially fewer program paths are executed.

In addition, a late splitting optimization in *jpf-bdd*, which is also common in other model checkers, chooses the value for feature variables at the last possible point of (execution) time. In our example, this means to store an *unknown* value for *BAR* in Line 1 and to choose the concrete value (*true* or *false*) only in Line 14. Lines 4–13 do not depend on *BAR*, so they only have to be executed twice (once for every assignment of *FOO*). This simple optimization (late splitting) saves nearly half of the analysis time compared to a brute-force approach. Still, JPF always splits the entire state, which corresponds to a store of the form `Cond[Map[String,Value]]`, and cannot take advantage of sharing between contexts as we do in our interpreter (using `Map[String,Cond[Value]]`). Similarly, *jpf-bdd* can join stores, but only if they are identical, except for feature variables.

For more information on *jpf-bdd* and on performance improvements, we refer to a recent workshop paper [30].

By extending JPF, we diverge from the pure black-box strategy and actually extend an existing tool. We still reuse most existing work. Hence, we call this a *gray-box strategy*. Actually, *jpf-bdd* was developed independently of and prior to our testing efforts and is not specific to product lines. Put differently, we reused the existing tool *jpf-bdd* as black-box without further modifications. However, the fact that the extension was developed by the second author gives us some perspective on the effort of specific extensions.

4.2 Experience

To gain experience with JPF for variability-aware testing, we rewrote the Graph Product Line [19] as a product-line simulator (as explained above). The Graph Product Line is a frequently used benchmark for product-line technology, a product line with 15 features, giving rise to 42 configurations, written in about 1000 lines of Java code, and (slightly) more realistic than the generated WHILE programs above. We attempted to detect 10 bugs carefully introduced by Cohen et al. for prior work on testing with sampling strategies [5]. One of the defects introduces an endless loop, so it cannot be found with JPF. Of the remaining defects, two defects already showed up with exceptions; for the others, we encoded corresponding specifications using runtime assertions, analogue to how xUnit unit tests indicate a failed test with an exception. We executed tests with two provided test graphs.

We built 10 variants of the product-line simulator (9 variants with one defect each, and 1 variant without defects). As a baseline, we tested each of the 42 configurations of each variant in a brute-force fashion in a standard Java execution environment. Next, we executed JPF (henceforth called *jpf-core*) and our extension *jpf-bdd* on all 10 variants. We report the arithmetic mean of three executions and the corresponding standard deviation, with 2 GB RAM on two 1 GHz cores of an Opteron QuadCore machine.

Running tests in the brute-force strategy with Java took 13 ± 0 seconds per product line. In contrast, *jpf-core* needs 167 ± 50 seconds, *jpf-bdd* 14 ± 1 seconds per product line.

First, surprisingly, *jpf-core* is much slower than the brute-force approach. However the difference can be explained because the standard Java virtual machine is more optimized than the virtual-machine part of JPF (which runs a custom byte-code interpreter written in Java). Executing the brute-force approach with the JPF virtual machine (deterministic, without performing additional model-checking overhead) requires 230 ± 7 seconds per product line, which indicates a conceptual speed-up. As the brute-force approach behaves exponentially, we expect higher speed-ups in larger product lines.

Second, *jpf-bdd* outperforms *jpf-core* by an order of magnitude, because it can join many paths. In the Graph Product Line, joins are particularly effective, because several features have no persistent influence on the program state. For example, feature *Cycle* executes searches for cycles in the graph, prints the result, but does not change any variables shared with other features; so, *jpf-bdd* joins where *jpf-core* can not.

Though we are at an early stage, our experiment is encouraging to look at variability-aware testing with (extended) model checkers.

5. RELATED WORK

Product-line testing. As in all other domains, testing has been recognized as a crucial topic during product-line development. General strategies, such as those discussed by Pohl et al. [22], emphasize testing features in isolation (for example, unit tests on plug-ins) and preparing test cases that should be run on each generated product. Testing the integration of features remains hard, though. Pohl et al. distinguish a brute-force strategy from a sampling strategy and an application-only strategy (only products generated for customers are tested). They encourage reuse of test artifacts, but they have no means of testing all configurations of the product line, other than brute-force.

Along these lines, many researchers have investigated suitable sampling strategies according to some coverage criteria [5, 8, 18, 20, 21, 24]. A typical strategy is sampling with n-way feature coverage, such that each n-tuple of features appears in at least one tested product [20]. Especially, 2-way feature coverage is frequently used, since it seems to strike a good balance between number of products that need to be tested and detection of interaction problems [16]. Nonetheless, sampling prevents establishing properties about the *entire* product line.

Another strategy to scale product-line testing is to determine which test cases need to be run in which configurations, to reduce the number of test executions. Kim et al. have used static analysis to conservatively approximate which test cases are influenced by which features [15]. Shi et al. have used symbolic execution to analyze the product line to reduce the number of products that need to be tested [24]. Cichos et al. explore a strategy to generate tests to achieve coverage for an entire product line [8], and Lochau et al. explore test case generation such that products can be tested incrementally [18]. All these approaches analyze the whole product line (or its test model) in a variability-aware fashion to reduce the number of tests, but the tests themselves are still executed on individual products. In contrast, by construction, our interpreter and our

encoding with model checking cover the entire product line and split *test execution* only when needed, without dedicated prior analysis.

Variability-aware analysis. Although a rather recent research topic, many researchers have investigated strategies for variability-aware analysis for parsing (white-box [14]), type checking (white-box [1,7,13,25] and black-box [23]), model checking (white-box [9, 17] and black-box [2, 23]), static analysis (white-box [4] and black-box [3]), and theorem proving (black-box [27]). For a detailed overview of that field, we defer the interested reader to a recent survey [26].

The specific style of writing a variability-aware analysis by mapping over conditional data structures was inspired by *variational programming* by Erwig and Walkingshaw [11, 12]. They also presented and formalized a type system for the lambda calculus in this style [7]. Our encoding differs from theirs in that we encode choices and feature models with arbitrary propositional formulas, instead of using atomic feature names defined within the conditional data structure. This difference makes our approach potentially simpler and more flexible, but also more expensive to compute (we rely on SAT solvers or BDDs).

Our interpreter implements a form of mixed concrete/symbolic execution—see [6] for an overview of that field. Conceptually, in the variability-encoded version, we consider all feature variables as symbolic and execute the remaining program with concrete values. We have not yet experimented with existing tools for symbolic execution. They seem promising as black-box tools for the variability-encoding strategy. There is a rich and advanced collection of tools to explore for product-line testing in future work.

6. DISCUSSION AND CONCLUSIONS

We have investigated variability-aware testing with a white-box strategy (variability-aware interpreter), a black-box strategy (variability encoding for JPF), and even a gray-box strategy (variability encoding for *jpf-bdd*). In all cases, we run a test case on all configurations of a product line at once, as opposed to a brute-force or sampling strategy. Although it is too early to draw sound conclusions, we want to share our observations and encourage feedback at this early stage. We have gained interesting insights into the spectrum between white-box, gray-box, and black-box analyses regarding implementation effort and flexibility.

Effort. The white-box strategy obviously requires more effort to implement than the black-box strategy. We need to write our own interpreter from scratch or significantly rewrite an existing interpreter, because variability pervades all data structures and execution steps. While writing interpreters is well understood, writing an interpreter for a full language such as Java, C, or JavaScript requires significant effort. In contrast, reusing existing and optimized tools in the black-box strategy allowed us to experiment directly with Java code with much less effort.

Flexibility. The white-box strategy is more flexible than the black-box strategy. The black-box strategy depends very much on the power of the existing analysis and how efficiently it deals with variability. We have to 'hope' that their optimizations fit to our use cases (test case execution despite variability in our case). The variability encoding does not necessarily have the shape of typical programs for which general-purpose analysis may be optimized.

Product-line analysis is special in that variability follows only few restricted patterns, reducible to propositional formulas and Boolean satisfiability problems. Those specifics are usually not considered by the black-box tools or might even get lost in the encoding (i.e., analyzing arbitrary expressions in *if* statements is much harder than analyzing presence conditions in choice nodes). By extending existing tools (gray-box strategy; *jpf-bdd* in our case), we can attempt to add some product-line specific optimizations. In the white-box strategy, however, we have full control over the execution and how to store variability internally. We can weigh where and how to encode variability (e.g., `Cond[Map[T, U]]` vs. `Map[T, Cond[U]]`), when to join results, and so forth. We exploit that variability is always expressed with propositional formulas, allowing more specific analyses, such as the one we performed with *whenTrue*.

```
1  int a, c, res;
2  #ifdef FOO
3  a = 3;
4  #else
5  a = 2;
6  #endif
7  c = 1;
8  res = 1;
9  while (c < a) {
10   c = c + 1;
11   res = res * c;
12 }
13 assert(res < 10);
```

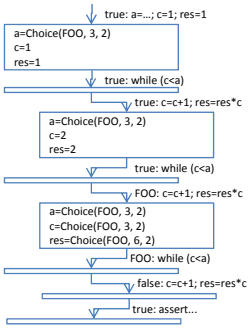

Figure 9: Example program calculating the factorial of *a* and the corresponding execution trace of our variability-aware interpreter (unchanged stores omitted)

We illustrate the difference in internal behavior between our variability-aware interpreter and the strategy of JPF with a constructed favorable example of the factorial function in Figure 9. The interpreter attempts to execute the body of the *while* loop three times. The first time with variability context *true*, that is, all values are updated together. Only in the second iteration, the body is executed in a restricted context; so, all values are updated conditionally. The final iteration then has variability context *false* and is not executed at all. This is an instance of storing variability locally and splitting as late as possible. In the same example, JPF (and *jpf-bdd*) separately computes the *while* loop 1 and 2 times without any sharing. This constructed example can demonstrate significant performance differences between both strategies, when using larger values for `a`.

We are still exploring different strategies within the spectrum between pure white-box and pure black-box approaches. The gray-box strategy appears promising, although extending existing black-box tools depends on predefined interfaces. Also experimenting further with white-box implementations should yield useful insights in the specifics of product-line testing. As next step, we want to grow our interpreter to support a real language. We are still at the beginning of the road to variability-aware testing and encourage others to join this path.

Acknowledgments. This work is supported by ERC grant ScalPL #203099 and the DFG grants AP 206/2, AP 206/4, and LE 912/13. We thank Myra Cohen for sharing the GPL bug scenarios.

7. REFERENCES

[1] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[2] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of feature interactions using feature-aware verification. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.

[3] E. Bodden. Position paper: Static flow-sensitive & context-sensitive information-flow analysis for software product lines. In *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2012.

[4] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural dataflow analysis for software product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 13–24. ACM,

2012.

[5] I. Cabral, M. B. Cohen, and G. Rothermel. Improving the testing and testability of software product lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 6287 of *LNCS*, pages 241–255. Springer, 2010.

[6] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: Preliminary assessment. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1066–1071. ACM, 2011.

[7] S. Chen, M. Erwig, , and E. Walkingshaw. Extending type inference to variational programs. Technical report (draft), School of EECS, Oregon State University, 2012.

[8] H. Cichos, S. Oster, M. Lochau, and A. Schürr. Model-based coverage-driven test suite generation for software product lines. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS)*, volume 6981 of *LNCS*, pages 425–439. Springer, 2011.

[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 *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344. ACM, 2010.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, New York, 2000.

[11] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1):Article 6, 2011.

[12] M. Erwig and E. Walkingshaw. Variation programming with the choice calculus. In *Proc. Int'l Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, 2011.

[13] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(3), 2012.

[14] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, 2011.

[15] C. H. P. Kim, D. S. Batory, and S. Khurshid. Reducing combinatorics in testing product lines. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 57–68. ACM, 2011.

[16] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Softw. Eng. (TSE)*, 30:418–421, 2004.

[17] K. Lauenroth, K. Pohl, and S. Toehning. Model checking of domain artifacts in product line engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280. IEEE, 2009.

[18] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental model-based testing of delta-oriented software product lines. In *Proc. Int'l Conf. Tests and Proofs (TAP)*, volume 7305 of *LNCS*, pages 67–82. Springer, 2012.

[19] R. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *Proc. Int'l Conf. Generative and Component-Based Software Engineering (GCSE)*, volume 2186 of *LNCS*, pages 10–24. Springer, 2001.

[20] S. Oster, F. Markert, and P. Ritter. Automated incremental pairwise testing of software product lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, volume 6287 of *LNCS*, pages 196–210. Springer, 2010.

[21] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proc. Int'l Conf. Software Testing, Verification, and Validation*, pages 459–468. IEEE, 2010.

[22] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin/Heidelberg, 2005.

[23] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350. IEEE, 2008.

[24] J. Shi, M. Cohen, and M. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Proc. Int'l Conf. Fundamental Approaches to Software Engineering*, volume 7212 of *LNCS*, pages 270–284. Springer, 2012.

[25] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007.

[26] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical Report FIN-004-2012, School of Computer Science, University of Magdeburg, 2012.

[27] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-based deductive verification of software product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM, 2012.

[28] E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. A specification-based approach to testing software product lines. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 525–528. ACM, 2007.

[29] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[30] A. von Rhein, S. Apel, and F. Raimondi. Introducing binary decision diagrams in the explicit-state verification of Java code. In *Proc. Java Pathfinder Workshop*, 2011.