Large-Scale Variability-Aware Type Checking and Dataflow Analysis

Jörg Liebig,[*] Alexander von Rhein,[*] Christian Kästner,[†] Sven Apel,[*] Jens Dörre,[*] Christian Lengauer[*]

[*] Department of Informatics and Mathematics, University of Passau
`{joliebig,rhein,apel,doerre,lengauer}@fim.uni-passau.de`

[†] Institute for Software Research, Carnegie Mellon University
`kaestner@cs.cmu.edu`

Technical Report, Number MIP-1212
Department of Informatics and Mathematics
University of Passau, Germany
November 2012

Large-Scale Variability-Aware Type Checking and Dataflow Analysis

Jörg Liebig,* Alexander von Rhein,* Christian Kästner,† Sven Apel,* Jens Dörre,* Christian Lengauer*
*University of Passau, Germany
†Carnegie Mellon University, USA

***Abstract*—A software product line is a family of similar software products that share a common set of assets. The advent of proper variability management and generator technology enables end-users to derive individual products solely based on a selection of desired features. This gives rise to a huge configuration space of possible products. But the high degree of variability comes at a cost: classic analysis methods do not scale any more; there are simply too many potential products to analyze. Hence, researchers have begun to develop *variability-aware* analyses, which exploit the similarities of the products of a product line to reduce analysis effort. However, while being promising, variability-aware analyses have not been applied to real-world product lines so far. We close this gap by developing and applying two full-fledged analyses to two real-world, large-scale systems: the Busybox tool suite and the Linux kernel. We report on our experience with making variability-aware analysis ready for the real world, and with applying it to large-scale product lines. A key result is that variability-aware analysis can outperform even very limited sampling heuristics with respect to analysis time.**

I. Introduction

Software product lines have gained considerable momentum in academia and industry. A software product line is a set of similar software products tailored to the requirements of the stakeholders of a particular domain, with the goal of reusing assets among the individual products [16], [36]. Typically, the variabilities and commonalities of the products of a product line are expressed in term of features (i.e., units of behavior or other product characteristics visible to end-users) and feature dependencies are described in a feature model [22].

While there are different approaches to product-line engineering [16], [36], generator-based approaches have proved successful [2], [17]. For example, the Linux kernel can be configured by means of thousands of configurations options [39], thus giving rise to a product line whose products can be generated based on a user's feature selection.[1]

While advances in variability management and generator technology facilitate the development of product lines with myriads of products [2], [17], this high degree of variability is not without cost. What are the properties of the multitude of products that can be generated? Unfortunately, classic analyses do not scale to that "brave new world". For systems such as the Linux kernel, it is not even possible to generate all products in order to analyse them, because they have so many products.

Recently, researchers have begun to develop a new class of analyses that are *variability-aware* [43]. The key idea is not to generate and analyze individual products, but to directly analyze the variable code base with the help of configuration knowledge. In the case of the Linux kernel, one analyzes the source C code including preprocessor directives, instead of applying the preprocessor to generate individual kernel variants. The key idea behind variability-aware analysis is to take advantage of the similarities of the products of a product line.

There are several proposals for variability-aware analyses in the literature, including parsing [26], type checking [3], [25], [42], dataflow analysis [11], model checking [4], [15], [30], and deductive verification [44]. However, while this work is promising, variability-aware analyses have not been applied to real-world product lines so far; previous work concentrated either on the formal foundation or is limited with respect to practicality, as we discuss in Section II.

As variability-aware analysis considers all code and all variations of a product line simultaneously, it is unclear whether it really scales to large-scale systems, although experiments on small-scale case studies are promising. To explore the feasibility and scalability of variability-aware analysis in practice, we have developed two full-fledged variability-aware analyses for C: type checking and liveness analysis. We applied each of them to two real-world, large-scale systems: the Busybox tool suite and the Linux kernel. In terms of scalability, we compare the variability-aware analyses to state-of-the-art sampling strategies used in practice—generating all products is not even possible in reasonable time in our case studies. While sampling strategies can substantially reduce analysis effort and time (by analysing only a subset of products), it reduces the analysis coverage (i.e., not all valid feature combinations are considered). In a nutshell, we found that both variability-aware analyses scale well—even outperform some of the sampling strategies—while still attaining coverage of all products.

Beside quantitative results, we report on our experience with making variability-aware analyses ready for the real world, and we discuss observations that provide insights into the nature of variability-aware analysis, and that can guide the development of further analyses.

Overall, we make the following contributions:

- An experience report of how to implement scalable variability-aware analyses based on an existing variability-

[1]For simplicity, we use the terminology of the product-line community: features represent configuration options, products are configured programs, configurations are specified by feature selections, and a feature selection is valid if it conforms to the feature model's dependencies.

aware parsing framework [26].

- Practical, variability-aware type checking and dataflow analysis tools for product lines written in C, in which variability is expressed with preprocessor directives and the build system.
- A series of experiments that compare the performance of variability-aware analysis with the performance of corresponding state-of-the-art sampling strategies based on two real-world, large-scale case studies.
- A discussion of our experience with applying variability-aware analysis in practice, and of patterns we encountered in our investigation.

The sample systems and all experimental data are available on the supplementary website: http://fosd.net/vaa.

II. Background and Related Work

Early approaches of product-line analysis (mainly testing) rely on limited, well-scoped sets of products that are in the focus of development and maintenance [36]. Such a set, called *product map* [7] or *product portfolio* [36], is a subset of all possible products of a product line that targets specific customers and that is regularly analyzed.

With the advent of proper variability management and generator technology [2], [17], the vision of creating highly customized products, only based on a user's specification, without considerable overhead, came into reach. However, this push-button approach comes at the cost of an exploding space of possible products. To address this challenge, a whole new class of product-line analyses has been developed. These analyses can be classified according to the strategy of how they cope with the possibly huge variant space of a product line [43]. Since features can interact, it is usually not possible or sufficient to analyze features in isolation, but also their combinations need to be analyzed. In our experiments, we consider two strategies: product-based analysis with sampling and variability-aware analysis, which we discuss next.

Product-based analysis and sampling: In a brute-force product-based strategy, every product of the product line is generated and analyzed individually. However, due to the sheer size of real-world product lines (the number of products can grow exponentially with the number of features), this is infeasible in practice. To handle this problem, product-line developers typically analyze only a subset of products, called the *sample set*. The idea is that, even though we cannot analyze all products individually, we can still strive for analyzing a representative sample set to be able to draw informed conclusions about the entire product set (e.g., in terms of defect probability and non-functional properties). The sample set is either selected by domain experts or automatically generated according to a *sampling heuristic*. Researchers and practitioners have proposed different sampling heuristics, of which we selected three that are most relevant in practice: single conf, pairwise, and code coverage, which we motivate and explain next.

The simplest sampling heuristic, called *single conf*, is to analyze only a single product that contains most, if not all, of the features of the product line. The benefit of this heuristic is that we need to analyze only a single product, hence it is fast. By selecting many features, the heuristic tries to cover a large part of the product-line's implementation, however, it cannot cover mutually exclusive features or intricate interactions specific to individual feature combinations [21]. According to Dietrich et al. [19], in the Linux development community, it is common to test only one predefined product with most features selected, called *allyes-config* in Linux.

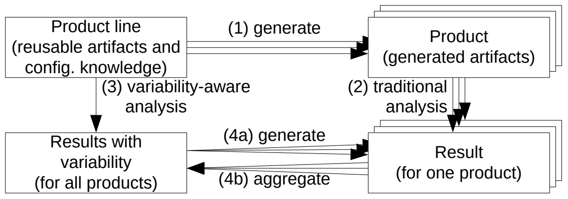


Fig. 1: The underlying pattern of variability-aware analysis.

The *pairwise* sampling heuristic is motivated by the observation that many faults in software systems are caused by interactions of, at most, two features [12], [29], [35], [38]. Using pairwise sampling, the sample set contains a minimal number of products that cover all pairs of features, whereby one product is likely to cover multiple feature pairs. For our experiments, we use the tool of Henard et al. to compute the sample set, as described in Section IV-C. Following the heuristic, we can discover first-order interactions (between two features, which are most common [29]), but may miss higher-order interactions (between more than two features).

Finally, *code-coverage* sampling is inspired by the statement-coverage criterion used in software testing [46], which states that a test suite must execute every statement of a program, at least, once. Code-coverage sampling in product-line engineering pursues a similar approach [41], but is targeted at product generation. It selects a minimal sample set of products, such that every lexical code fragment of the product line's code base is contained in at least one product (but not necessarily all possible combinations of individual code pieces are checked). For an overview of other sampling strategies, some of which require more sophisticated upfront analyses, see a recent survey [43].

Variability-aware analyses: Variability-aware analyses (also known as family-based analyses) take a different approach to cope with the possible huge variant space of a product line. The key idea is to take advantage of the similarities between the products of a product line in order to speed up the analysis process. Although individual variability-aware analyses differ in many details, an idea that underlies all of them is to analyze code that is shared by multiple products only once.

Variability-aware analyses do not operate on generated products, but on the reusable code artifacts and configuration knowledge, which are the input for the product generator, as illustrated in Figure 1. Depending on the implementation technology, reusable artifacts are some kind of components or modules that can be composed in different combinations or code fragments that are conditionally included from a common code base [24]. Our experiments are based on systems that express variability using the latter approach: C code with conditionally included or excluded code fragments, controlled by preprocessor directives such as #ifdef and #define.

As reusable artifacts such as C code with preprocessor directives cannot be processed directly by standard analyses, the code or the analysis has to be prepared—it has to be made variability-aware (preprocessing does not help, as it removes variability). There are different approaches to do so. First, one can adapt existing analyses to empower them to work at the level of variable and reusable artifacts such as individual components. This approach has been pursued for adapting existing type-checking, model-checking, and testing techniques to product lines [3], [5], [15], [25], [28], [30]. As an example, Apel et al. compared variability-aware model-checking with several sampling strategies; they found that variability-aware analysis outperforms existing sampling strategies both in terms of verification time and number of detected errors [5]. Second, one can encode variability in the artifact code (or representations thereof) to analyze. For example, for applying model checking and deductive verification to entire product lines, it has been proposed to create a product simulator for a given product line, which incorporates code and behavior of all individual products, as has been used in variability-aware model checking, deductive verification, and testing [4], [28], [37], [44]. Then, the analysis is applied to the simulator. The variability-aware parsing framework, on which we base our implementations, is a special instance of a lifted analysis [26]; given C code with preprocessor directives, it produces a single abstract syntax tree that covers the code of all products, as we will discuss in Section III. With the knowledge of which parts of the tree correspond to which feature combinations, a variability-aware analysis can simultaneously analyze all products.

Several instances of variability-aware analysis have been proposed in the literature; we discuss the analyses that influenced our work. In earlier work, we have developed two variability-aware type systems for Featherweight Java, a subset of Java [3], [25]. While the variability-aware type checker we present and evaluate here is inspired by this previous work, it targets a full-fledged language and real-world systems. Only this practical setting allows us to explore the tradeoffs and merits of variability-aware analysis, for example, with regard to specific analysis patterns (which we introduce later), which have not been considered (or only implicitly) in previous work. Other researchers have sketched similar strategies [6] and developed variability-aware type systems for the lambda calculus [13] and dialects of Java [18], [42]. A prior version of our type checker has been used in a study on variable module systems, but without empirical assessment and comparison to product-based analysis and sampling [27].

As with type checking, researchers proposed variability-aware approaches to a dataflow analysis. Closest to our work, Braband et al. compared three different algorithms for variability-aware, intra-procedural dataflow analysis for Java against a brute-force product-based approach [11]. Similarly, Bodden proposed an approach to extend an existing information-flow analysis framework to make it variability-aware [10]. Both approaches are limited to an academic setting in which the input Java programs contain #ifdef-like variability annotations managed by a research tool; to the best of our knowledge, there are no substantial product lines that use this technique. Furthermore, both approaches make limiting assumptions on the form of variability (in particular, type uniformity [25] and annotation discipline [23], [32]), which do not hold in real-world product lines [31], [32]. For a detailed overview of existing variability-aware analyses, see a recent survey [43].

III. Variability-Aware Analysis

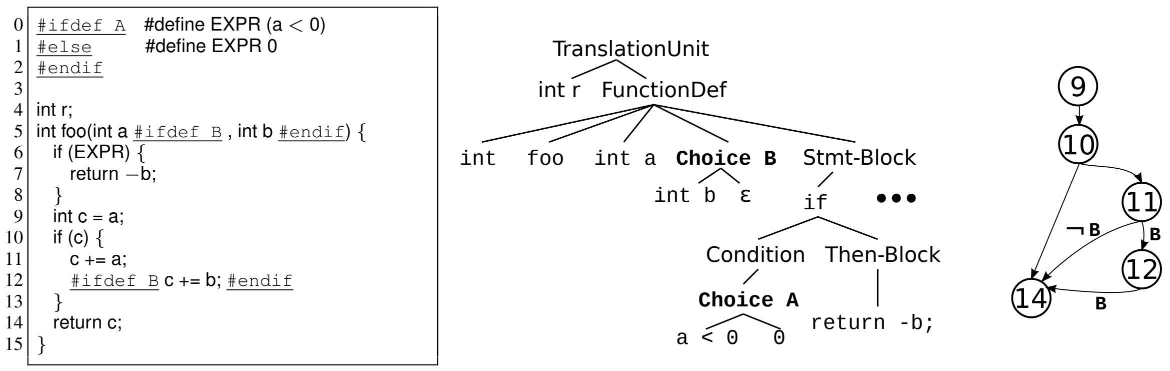
Although variability-aware analysis has been applied in academic projects, showing promising performance results, it has never been applied to real-world product lines. Since most industrial product lines are implemented in C and use #define and #ifdef directives of the C preprocessor (and a build system) to implement compile-time variability, we set the goal of implementing two variability-aware analyses *for C* and of applying them to *large-scale* projects. In this section, we describe our analyses and our experience before we get to our evaluation with Busybox and Linux in Section IV.

A. Variable Abstract Syntax Trees and Presence Conditions

Most static analyses are performed on an abstract syntax tree (AST) of the program to analyze. Since we want to analyze entire product lines, we construct an abstract syntax tree that contains all variability information of all products and the corresponding configuration knowledge.

The desired *variable AST* is like a standard AST, but it contains additional nodes to express variation. A Choice node expresses the choice between two or more alternative subtrees (similar to ambiguity nodes in GLR parse forests [45] and explored formally in the choice calculus [20]). One alternative of a choice may be empty (ε); this denotes an optional entry. We illustrate Choice nodes in Figure 2. In principle, we could use a single Choice node on top of the AST with one gigantic branch per product; but the variable AST is more compact, because it shares parts that are common across multiple products (e.g., in Figure 2, we store only a single node each for the declaration of *r* and the function name *foo*, which are shared by all products). It is this sharing and keeping variability local, which makes variability-aware analysis faster than a brute-force approach.

To reason about variability, we additionally need to incorporate configuration knowledge, more exactly, which subtrees of the variable AST are included in which products. To this end, we annotate subtrees with *presence conditions*. Propositional formulas are sufficient to describe presence conditions and efficient to reason about using SAT solvers in practice [34]. As an example, in Figure 2, parameter *b* is only included if feature *B* is selected, whereas the condition of the *if* statement has two alternative subtrees depending on whether feature *A* is selected. In our examples, presence conditions are atomic and refer only to a single feature, but more complex presence conditions, such as $A \wedge \neg(B \vee C)$ are possible. By storing presence conditions in Choice nodes, we can derive the abstract syntax of every product of the product line, given the feature selection for



```
 0 #ifdef A   #define EXPR (a < 0)
 1 #else      #define EXPR 0
 2 #endif
 3
 4 int r;
 5 int foo(int a #ifdef B , int b #endif) {
 6   if (EXPR) {
 7     return −b;
 8   }
 9   int c = a;
10   if (c) {
11     c += a;
12     #ifdef B c += b; #endif
13   }
14   return c;
15 }
```

Fig. 2: Code example (left), excerpt of the corresponding variable AST (middle), and excerpt of the corresponding variable CFG (right); for brevity, we underlined and integrated #ifdef directives within single code lines.

that product. Compact representations of variable ASTs in this or similar forms are commonly used in variability-aware analyses [11], [20], [25], [26].

B. Variability-Aware Parsing

Before we can actually analyze the AST, we need to create it from the source code by parsing. The parsing step has been the main obstacles causing variability-aware analysis being used only in academic environments such as CIDE [23] so far. Whereas parsing C code (and parsing most other languages) of a single product is well established, parsing an entire product line with its variability is problematic. To make matters worse, in the C preprocessor, conditional-compilation directives (#ifdef, etc.) interact with the build system, with macros (#define, etc.), and with file-inclusion facilities (#include), across file boundaries, in intricate ways.

It took us nearly two years to figure out a parsing process that preserves the variability from C preprocessor directives and build scripts and represents it in a variable AST, as shown in Figure 2.[2] Finally, we found and implemented a sound and complete solution as part of the TypeChef project (following the variability-awareness equivalence from Figure 1): We have built a variability-aware parser for C based on a special variability-ware lexer and parser-combinator framework [26]. By combining TypeChef with previous work on feature-model extraction [9], [40] and build-system analysis [8], we are able to produce variable ASTs for substantial systems such as Busybox and the Linux kernel, both of which use the C preprocessor and the Kbuild build system to express variability (and *Kconfig* to describe a variability model). For details on the parser, see the corresponding publication [26]; in the remainder of this paper, we simply use this parser framework as a black box and work on the resulting variable ASTs.

C. Variability-Aware Type Checking

A standard type-checking algorithm for C traverses the AST, collects declarations in a symbol table, and attempts to find types for all expressions. In principle, a variability-aware type checker works just the same, but covers all products, and it must be aware of variability in each of the following three steps (cf. Fig. 1).

First, a symbol may only be declared in some products or may even have alternative types in different products. Therefore, we extend the symbol table (similar to the proposal of Aversano et al. [6]), such that a symbol is no longer mapped to a single type, but to a conditional type (a choice of types or ε). We illustrate a possible encoding of a conditional symbol table in Table I. If a symbol is declared in all products, we do not need Choice nodes; however, if symbol is declared in a subtree of the AST that is only reachable given a presence condition, we include the symbol and type in the symbol table just under that condition. The same way, we may declare a symbol with different types in different products. In our running example, function *foo* has two alternative types, depending on whether feature *B* is selected. Similarly, we also made the table for structures and enumerations in C variability-aware.

Second, during expression typing, we need to lookup and compare variable types (choices of types). For example, looking up a name in a symbol table may return a variable type. When checking that an expression (for example, the condition of an *if* statement) has a scalar type, we need to check that all alternative choices of the variable type are scalar. If the check fails only for some alternative results, we can report a type error for the product line and pinpoint the error to a subset of products, as characterized by the corresponding presence condition. Similarly, an assignment is only valid if the expected (variable) type is compatible with the provided (variable) type in all products. Therein, an operation on two variable types can, in the worst case, result in the Cartesian product of the types in either case, resulting in a variable type with many alternatives. All other type checks are essentially implemented along the same lines. In our running example, we would report a type error in Line 7, because variable *b* cannot be resolved in products without feature *B* (see also the symbol table in Table I).

Third, we can use the feature model of the product line (if available) to filter all type errors that occur only in invalid products, according to the feature model. To this end, we simply check whether the presence condition of the type error

[2]Note how we resolve macro *EXPR* with two alternative expansions during parsing.

Symbol	(Conditional) Type	Scope
r	int	0
foo	Choice(B, int → int → int, int → int)	0
a	int	1
b	Choice(B, int, ε)	1

TABLE I: Conditional symbol table at Line 5 of our running example.

is satisfiable when conjoined with the feature model (checked with a standard SAT solver).

D. Variable Control-Flow Graphs

To perform dataflow analysis, we first need to construct a control-flow graph (CFG). A CFG is a graph that represents all possible execution paths of a program. Nodes of the CFG correspond to instructions in the AST, such as expressions, assignment statements, or function calls; edges correspond to possible successors according to the execution semantics of the programming language used. A CFG is a conservative static approximation of the actual behavior of the program. As with type checking, we need to make CFGs variable to cover all products of a product line.

To create a CFG for a single program, we need to compute the successors of each node (succ: Node→Set[Node]). In a product line, the successors of a node may differ in different products, so we need a variability-aware successor function that may return different successor sets for different products (succ: Node→Choice[Set[Node]], or, equivalently but with more sharing, succ: Node→Set[Choice[Node]]). Using the result of this successor function, we can determine for every possible successor a corresponding presence condition, which we store as annotation of the edge in the variable CFG.

Let us illustrate variable CFGs by means of the optional statement in Line 12 of our running example. In Figure 2 (right), we show an excerpt of the corresponding variability-aware CFG (node numbers refer to line numbers of the code listing). The successor of the instruction c += a in Line 11 depends on the feature selection: if feature B is selected, statement c += b in Line 12 is the direct successor; if feature B is not selected, return c in Line 14 is the (only) successor. Technically, we add further nodes to the result set of the successor function, until the conditions of the outgoing edges cover all possible products, in which the source node is present (checked with a SAT solver). By evaluating the presence conditions on edges, we can reproduce the CFG of each product, as we would have computed them in the brute-force approach.[3]

[3]Alternatively, we could have dropped the presence conditions on edges and express variations of the control flow with *if* statements. On an *if* statement, a normal CFG does not evaluate the expression, but conservatively approximates the control flow by reporting both alternative branches as possible successor statements (e.g., in Figure 2, both nodes 11 and 14 may follow node 9). Such sound but incomplete approximation is standard practice to make static analysis tractable or decidable. However, for static variability, we do not want to lose precision, to preserve the equivalence of Figure 1. Furthermore, we have only propositional formulas to decide between execution branches, which makes computations decidable and comparably cheap, so we decided in favor of presence conditions on edges.

Line	Uses	Defines	In	Out
9	$\{a\}$	$\{c\}$	$\{a, b_{\mathtt{B}}\}$	$\{a, b_{\mathtt{B}}, c\}$
10	$\{c\}$	$\{\}$	$\{a, b_{\mathtt{B}}, c\}$	$\{a, b_{\mathtt{B}}, c\}$
11	$\{a, c\}$	$\{c\}$	$\{a, b_{\mathtt{B}}, c\}$	$\{b_{\mathtt{B}}, c_{\mathtt{B}\vee\neg\mathtt{B}}\}$
12	$\{b_{\mathtt{B}}, c_{\mathtt{B}}\}$	$\{c_{\mathtt{B}}\}$	$\{b_{\mathtt{B}}, c_{\mathtt{B}}\}$	$\{c_{\mathtt{B}}\}$
14	$\{c\}$	$\{\}$	$\{c\}$	$\{\}$

TABLE II: Result of liveness computation of code example in Figure 2; $a_{\mathtt{B}}$ means that variable a is in the result set if feature B is selected.

E. Variability-Aware Liveness Analysis

Based on the variable CFG, we have implemented variability-aware liveness analysis, a standard dataflow analysis [1]. Liveness analysis computes all variables that are live (that may be read before written again) for a given statement. Its result can be used, for example, to conservatively detect or eliminate dead code. In a product line, warnings about dead code that occurs only in specific products are interesting for maintainers; corresponding problems are regularly reported as bugs.[4] So, again, our goal is to make liveness analysis variability-aware to cover entire product lines (cf. Fig. 1).

Liveness analysis is based on two equations: uses computes all variables read, and defines computes all variables written to. While, in a traditional liveness analysis, both equations return sets of variables, in variability-aware liveness analysis, both return variable sets (a choice of sets or a set with optional entries). The computation of liveness is a fixpoint algorithm that consists of two equations, in and out, which denote variables that are live before respectively after the current statement. The result of in and out is variable again.

In Table II, we show the results of variability-aware liveness analysis for our running example. We show the result of each equation as a set of variables together with their presence condition as subscript. For example, only c is live in the return statement on Line 14. The variables a and $b_{\mathtt{B}}$ are live in the declaration statement on Line 9 because, considering the control flow from Line 9 to 12 ($9 \rightarrow 10 \rightarrow 11 \rightarrow_{B} 12$), both variables can be reached.

F. Keeping Variability Local

Over the years, we have gained considerable experience with making various analyses variability-aware (see Section II). We learned that it is essential to preserve sharing during analysis and to keep variability as local as possible in order to be able to scale the analysis as we have done here. Specifically, two patterns emerged that we have used in many of these analyses: *late splitting* and *early joining*. We illustrate them using our type checker and liveness analysis; this will also explain why the variability-aware solution typically outperforms a brute-force approach, as we will demonstrate in Section IV.

In contrast to a brute-force approach, variability-aware analyses are conceptually much faster because they do not recheck common parts of the source code over and over

[4]e.g., https://bugzilla.kernel.org/show_bug.cgi?id=1664.

again. In our running example, there are two features and four products. A brute-force approach would recheck all common parts, such as int c = a, in every product. In variability-aware analyses, we attempt to check common parts only once. Of course, where variability occurs we need to check all variations. But variability tends to be local already after parsing in the variable AST; it leaks only moderately into other parts of the program and remains largely orthogonal between features. So, variability-aware analysis does not need to consider all possible feature combinations as the brute-force strategy does.

The first pattern of *late splitting* means that we perform the analysis without additional variability until we encounter variability. For example, we process the declaration of symbol *r* in Line 4 of our running example only once. Also, when we use symbol *r* later, it has just one type. We only split and consider smaller parts of the product space when we actually encounter variability, for example, in the declaration of parameter *b*. Late splitting is similar to path splitting in "on-the-fly" model checking, where splitting is also only performed on demand [14].

However, we also keep variability local in intermediate results. For example, instead of copying the entire symbol table for a single variable entry, we have only a single symbol table with conditional entries (technically, we use Map[String,Choice[Type]] instead of Choice[Map[String,Type]] to achieve this locality). Therefore, even after the conditional declaration of parameter *b*, we only look up a single type for *a* or *r*, independent of feature *B*.

Finally, due to locality, we can *join* intermediate results early in many cases. For example, when determining the type of the expression in Line 5 of our running example, we get a choice of two scalar types (Choice(A, int, int)). For further processing, we can join this variable type to a single type int. So, even if we need to compute the Cartesian product on some operations with two variable types, the result can often be joined again to a more compact representation. This way, variability from parts of the AST leaks to other parts if and only if variability *actually makes a difference* in the internal representations of types, names, or other structures. Also, we need to consider only combinations of features that occur in different parts of the AST if they actually produce different (intermediate) results when combined, otherwise the results remain orthogonal.

We can observe the same local variability with late splitting and early joining in CFG construction and dataflow analysis as well. We need to split only when we actually encounter variability, otherwise we compute invariable successor sets, in sets, out sets, and so forth. Again, we encode variability locally in intermediate data structures, which allows us to join intermediate results. For example, if a node has the same successor with presence conditions B and $\neg B$, we can return a single successor with presence condition $B \vee \neg B = true$ (cf. computation of out in Line 11 of our running example). Our experience shows that joining is even more common in CFG construction than in dataflow analysis and type checking, because we can often join variable intermediate results again after processing an optional statement, so the effect of a variable statement typically remains quite local.

IV. Empirical Study

To evaluate its feasibility and scalability, and to learn about the merits and challenges of large-scale variability-aware analysis, we compared our variability-aware implementations of type checking and liveness analysis with corresponding product-based analyses. Specifically, we used three different state-of-the-art sampling heuristics to generate products for analysis (single conf, code coverage, and pairwise; cf. Sec. II). A brute-force generation of all possible products was infeasible as it would have required excessive execution time.

Our evaluation is based on two real-world, large-scale software systems—a fact that increases external validity compared to previous work, which concentrated on formal foundations and which was backed on comparatively small and academic case studies (cf. Sec. II).

A. Hypotheses and Research Questions

Based on the goals and properties of variability-aware and product-based analyses, we pose two hypotheses and two research questions.

1) *Variability-aware vs. single conf:* Analyzing all products simultaneously using variability-aware analyses is slower than analyzing a single product that contains all features. One reason is that the variable AST covering all products is larger than the AST of any single product, including the largest possible product.

   H_1 The execution times of variability-aware type checking and liveness analysis are *larger* than the times to analyze the products derived by the single configuration sampling heuristic.

2) *Variability-aware vs. pairwise:* The pairwise heuristics assumes that most unwanted interactions in product lines occur between two features. Therefore, it is reasonable to select a test suite that covers as many feature pairs as possible, while keeping the test suite relatively small. While previous work has shown that pairwise sampling is a reasonable approximation to analysis of all products [33], our experience is that such sampling heuristic still generates quite large test suites. Hence, we expect a variability-aware analysis to outperform the corresponding product-based approach based on pairwise sampling:

   H_2 The execution times of the variability-aware type checking and liveness analysis are significantly *smaller* than the times to analyze all the products derived by pairwise sampling.

3) *Variability-aware vs. code coverage:* With respect to the comparison of variability-aware analyses and product-based analyses based on code-coverage sampling, we cannot make any informed predictions with respect to analysis time. The code-coverage sampling algorithm generates configuration sets depending on the usage of features in the analyzed C-files. Because we do not know details about feature usages, we cannot predict how many products will be generated and how large these will be.

Therefore we cannot state a hypothesis and pose a research question instead.

RQ$_1$ How do the execution times of variability-aware type checking and liveness analysis compare to the times for analysis of the products derived by code-coverage sampling?

4) *Scalability:* Finally, we pose the question of the scalability of variability-aware analysis.

RQ$_2$ Does variability-aware analysis scale to systems with thousands of features?

The background for questioning scalability is that variability-aware analysis reasons about variability by solving SAT problems (cf. Sec. III). Generally, solving SAT problems is NP-hard, but previous work has shown that the problems that arise in variability-aware analysis are typically tractable for state-of-the-art SAT solvers [34], and that caching can be an effective optimization [3].

B. Sample Systems

To test our hypotheses and to answer our research questions, we selected two sample systems for evaluation. We looked for publicly available systems (for reproducibility) that are of substantial size, actively maintained by a community of developers, used in real-world scenarios, and that implement compile-time variability with the C preprocessor. As a prerequisite, the system must be developed as a product line or, at least, provide a variability model that describes features and their valid combinations (a requirement which excluded several systems from our previous study [31]). Eventually, we selected Busybox and Linux. In this context, we would like to acknowledge the pioneering work on feature-model extraction [9], [40] and build-system analysis [8], which enabled us, for the first time, to conduct two substantial, real-world case studies on variability-aware analysis.

- The *Busybox* tool suite is a medium-sized product line that reimplements most standard Unix tools for resource-constraint systems, especially in the embedded-systems domain. With 792 features it is highly configurable; most of the features refer to independent and optional subsystems. We analyze Busybox version 1.18.5 (522 files and 206 815 lines of source code, measured with cloc (http://cloc.sf.net)).
- The *Linux kernel* (x86 architecture, version 2.6.33.3) is a large-sized product line of configurable operating-system kernels with millions of installations worldwide, from high-end servers to mobile phones. It is highly configurable with 6 918 features. In a previous study, we identified the Linux kernel as one of the largest and most complex publicly available product lines [31]. It has 7 665 source code files with 6.7 million lines of code. Note that already the feature model of Linux is of substantial size: the extracted formula in conjunctive normal form has over 60 000 variables and nearly 300 000 clauses; answering a single satisfiability question requires over half a second on our machines.

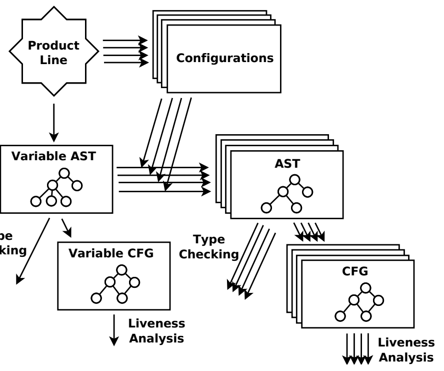


Fig. 3: Experimental setup.

C. Experimental Setup

We use TypeChef as underlying parsing framework. As explained in Section II, TypeChef generates a variable AST per file, in which choice nodes represent optional and alternative code. Our implementations of variability-aware type checking and liveness analysis are based on variable ASTs, and they are integrated into and deployed as part of the TypeChef project. The entire project is implemented in Scala.

To avoid bias due to different implementations of the analyses, we compare different executions of our own implementation. To analyze an individual product, we run the same analysis on an AST without variability. We create this AST for a given configuration by pruning all irrelevant branches of the variable AST, so that no Choice nodes remain. Since there is no variability in the remaining AST, the analysis never splits and there is no overhead due to SAT solving, because the only possible presence condition is *true*.

Due to the sheer size of Busybox and Linux, we could not analyze all possible products individually in a brute-force fashion.[5] Instead, we used the three sampling strategies of Section II: single conf, pairwise, and code coverage.

For single-conf sampling, we used Linux's *allyes* configuration, as provided by the Linux configuration tool KConfig.[6] For Busybox, we created a large configuration using the configuration toolkit (we selected as many features as possible).

For the generation of pairwise product configurations, we employed Henard's tool,[7] which uses a genetic algorithm to generate product configurations that cover as many pairs of features as possible. To evaluate the coverage of the resulting set of products, the tool also computes the number of feature pairs that are valid in at least one product. For Busybox, we need only 20 product configurations to achieve a coverage rate of 97 % of all valid feature pairs. For Linux, we were not able to compute how many valid feature pairs exist due to the size of the feature model; so based on recommendations of Henard et al., we set the number of products for sampling to 50.

For code-coverage sampling, we computed a set of product

[5]In fact, in an attempt to compute the number of possible products of Busybox, we encountered an integer overflow, which means that the number exceeds 2^{32}.

[6]http://www.kernel.org/doc/Documentation/kbuild/kconfig.txt

[7]http://research.henard.net/SPL/

configurations per file, such that every piece of code is present in at least one product of the sample. We reimplemented the conservative algorithm of Tartler et al. for this task [41].[8]

For intraprocedural liveness analysis, it would have been more efficient to apply brute-force or sampling strategies to individual functions, not to files, as done by Braband et al. for Java product lines [11]. Unfortunately, preprocessor macros in C rule out this strategy, as we cannot even parse functions individually without running the preprocessor first or without performing full variability-aware parsing. In our running example, we would not even have noticed that function *foo* is affected by feature *A*, because variability comes from variable macros defined outside the function. Unfortunately, variable macros defined in header files are quite common in C code [26]. Hence, since there is no practical solution for function-level analysis, we used samples at file level.

In Figure 3, we illustrate the experimental setup. Depending on the sampling strategies one or multiple configurations are checked. For each file of the two sample systems, we measured the time spent in type checking and liveness analysis, each using the variability-aware approach and the three sampling strategies (the latter consisting of multiple inner runs)—in total 4 analyses per case study (1 variability-aware + 3 sampling).

We ran all measurements on a Linux machine (Ubuntu 12.04) with Intel Core i7-2600, 3.4 GHz, and 16 GB RAM. To remove measurement bias, we minimized disk access by using a ramdisk and warmed up the JVM by running each experiment once before the actual measurement run. We cleaned caches after each measurement.

D. Results

In Table III, we list the sum over all measurement results for each analysis strategy and sample system. (We report sequential times, though massive parallelization is possible in all cases, because all files are analyzed in isolation).

In Figures 4 and 5, we plot the distribution of analysis times for each strategy for Busybox and Linux (using notched boxplots). Each data point represents the measurement of one file in the respective project. We highlight the median of the variability-aware analyses to simplify comparison with the medians of the sample-based analyses. In addition, we provide the number of analyzed products for each of the sample-based analyses (below the name of the analysis). Because the number of necessary product configurations differs between files for code-coverage sampling, we provide the mean±standard deviation over all files.

With a paired t-test at confidence level 95 %, we performed tests corresponding to our research hypothesis. In both sample systems and for both analysis, variability-aware analysis is significantly slower than single-conf sampling (H_1) and significantly faster than pairwise sampling (H_2). The results regarding code-coverage sampling (H_3) are mixed: variability-aware analysis is significantly faster for liveness analysis in Linux, significantly slower for liveness analysis in Busybox and type checking of Linux, and there is no statistically significant difference for type checking Busybox. Table IV summarizes the actual speedups of all comparisons.

It is worth noting that we did not find any confirmed defects during our experiments. Nevertheless, for Linux, we found a defect, but it is not confirmed so far; for Busybox, we found several defects in earlier versions that have been fixed in the current version, which we used for our experiments.[9]

		Type checking	Liveness analysis
Busybox (521 files)	Single conf	15.5	8.82
	Code coverage	71.6	33.7
	Pairwise	344	187
	Variability-aware	70.2	20.1
Linux (6985 files)	Single conf	4050	1040
	Code coverage	19800	9340
	Pairwise	140000	44100
	Variability-aware	71200	2610

TABLE III: Total times for analysis of the case studies with each strategy (time in seconds, with three significant digits).

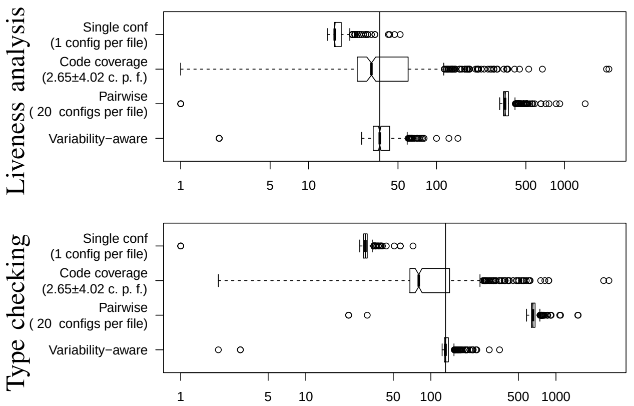


Fig. 4: Distribution of analysis times for Busybox (times in milliseconds; logarithmic scale).

E. Discussion

	Variability-aware vs.	Type checking	Liveness analysis
Busybox	Single conf	0.23	0.44
	Code coverage	(0.61)	0.86
	Pairwise	4.99	9.56
Linux	Single conf	0.07	0.39
	Code coverage	0.26	2.91
	Pairwise	2.27	16.60

TABLE IV: Speedup of variability-aware analysis based on median analysis times (non-significant result in parentheses).

The results confirm hypotheses H_1 and H_2: variability-aware analysis is faster than product-based analysis using

[8]Although there is an algorithm to compute an optimal solution (a minimal set of sample products) by reducing the problem to determining the chromatic number of a graph, we use the heuristics-based approach of Tartler et al. Unfortunately, determining the optimal solution is NP-complete and too slow for practical usage for instances of the required size for Linux. For more details on the optimal algorithm, see https://github.com/ckaestne/OptimalCoverage.

[9]Bug reports: https://bugs.busybox.net/show_bug.cgi?id=4994 and http://lists.busybox.net/pipermail/busybox/2012-April/077683.html

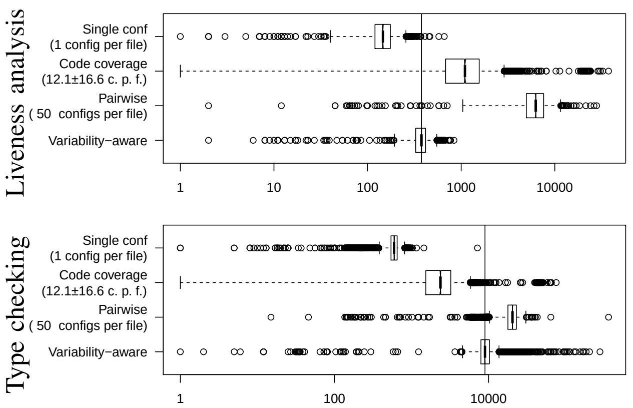


Fig. 5: Distribution of analysis times for Linux (times in milliseconds; logarithmic scale).

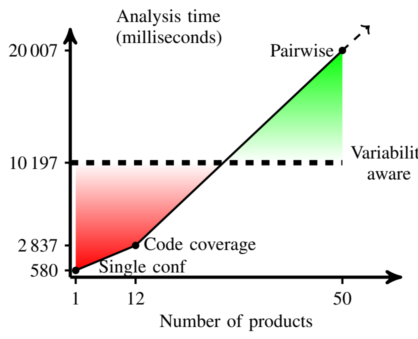


Fig. 6: Number of products versus mean analysis time for type checking of Linux.

pairwise sampling, but slower than product-based analysis using single-conf sampling on both case studies. With respect to research question RQ_1, there is no clear picture. The performance of the code-coverage strategy depends on the variability implementations in the respective files; number of sampled products and performance differ strongly between files inside each project (cf. Fig. 4 and 5). That is, performance of the code-coverage heuristic is hard to predict and depends strongly on the implementation in the individual files.

A further observation is that the speedup of variability-aware liveness analysis in relation to sampling is higher than the speedup of variability-aware type checking. This can be explained by the fact that liveness analysis is intra-procedural and type checking considers entire compilation units.

The experimental results for Busybox and Linux demonstrate that variability-aware analysis is in the range of the execution times of sampling strategies with multiple samples. So, with regard to question RQ_2, we conclude that variability analysis is practical for large-scale systems. The overhead induced by solving SAT problems for variability reasoning is not a bottleneck, not even for large systems such as the Linux kernel. Overall variability-aware type checking in Linux takes about as much time as checking 27 products (4 products in Busybox). Both values are very low compared with the number of possible products of the product lines. For liveness analysis, the break-even point is even only after two products (in both case studies).

To put variability-aware analysis into perspective, we illustrate the trade-offs compared to sampling by means of the example of type checking Linux in Figure 6. The x-axis shows the average number of products sampled for the respective sampling heuristics (per file); the y-axis shows the average analysis times for the products analyzed (per file). Clearly, the more products are analyzed, the longer that analysis takes, which illustrates the trade-off between analysis coverage and analysis time. The interesting point is the break-even point at which variability-aware analysis becomes faster (dashed line). But recall that, using sampling, one can be fast but at the price of losing information due to a limited analysis coverage.

Threats to validity: A key threat to validity is that our implementations of variability-aware type checking and liveness analysis support only ISO/IEC C, but not all GNU C extensions used by the sample systems (especially Linux). Our analyses simply ignore such code constructs. Also due to the textual and verbose nature of the C standard, the implementation does not yet align entirely with the behavior of the GNU C compiler. Due to such technical problems, we excluded one file of Busybox and 680 files of Linux from our analysis. All numbers presented here have been obtained after excluding the problematic files. Still, we argue that the large numbers of 521 files for Busybox and 6 985 files for Linux deem the approach practical and our evaluation representative.

Second, the products generated by the sampling heuristics represent only a small subset of possible products (which is the idea of sampling). But, for pairwise sampling based on Henard's tool, it may happen that some variants of a file are very similar, as the difference in the respective product configurations does not affect the content of the file. However, we argue that our conclusions are still valid, as this lies in the nature of the sampling heuristics and all heuristics we used are common in practice.

Finally, an obvious threat to external validity is that we considered only two sample systems. We argue that this threat is largely compensated by their size and the fact that many different developers and companies contributed to the development of both systems.

V. Conclusion

Variability-aware analysis is a promising approach to cope with the complexity induced by feature combinatorics in product-line engineering. We reported of our experience with the implementation of practical, scalable, variability-aware analyses for real-world systems written in C, including preprocessor directives. In a series of experiments on two real-world, large-scale case studies, we compared the performance of variability-aware type checking and liveness analysis with the performance of corresponding state-of-the-art sampling strategies. We found that the performance of variability-aware analysis is well within the spectrum of state-of-the-art sampling strategies, while, in contrast to sampling, maintaining full analysis coverage. Especially, the experiments with the Linux kernel suggest the practical potential of variability-aware analysis at large scale (despite the use of SAT-solving technology), which we were able to demonstrate for the first time.

In further work, we aim at exploring variations of the patterns of late splitting and early joining, at experimenting with other sampling heuristics and with more case studies, and at setting up an automated and incremental checking system that semiautomatically produces bug reports.

Acknowledgments: We thank Christopher Henard for help with the generation of pairwise configurations for Linux, Tillmann Rendel for many fruitful discussions on patterns in variability-aware analysis, and Klaus Ostermann for pushing us to generalize the underlying concepts.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Priniciples, Techniques, and Tools*. Pearson, 2006.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *J. Object Technology*, 8(5):49–84, 2009.

[3] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[4] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proc. of ASE*, pages 372–375. IEEE, 2011.

[5] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. of ICSE*. IEEE, 2013. to appear.

[6] L. Aversano, L. Di Penta, and I. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proc. of SCAM*, pages 83–92. IEEE, 2002.

[7] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In *Proc. Symp. Software Reusability (SSR)*, pages 122–131. ACM, 1999.

[8] T. Berger, S. She, K. Czarnecki, and A. Wasowski. Feature-to-Code Mapping in Two Large Product Lines. In *Proc. of SPLC*, pages 498–499. Springer, 2010.

[9] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability Modelling in the Real: A Perspective from the Operating Systems Domain. In *Proc. of ASE*, pages 73–82. ACM, 2010.

[10] E. Bodden. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *Proc. of SOAP*, pages 3–8. ACM, 2012.

[11] C. Braband, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proc. of AOSD*, pages 13–24. ACM, 2012.

[12] M. Calder and A. Miller. Feature Interaction Detection by Pairwise Analysis of LTL Properties: A Case Study. *Formal Methods in System Design*, 28(3):213–261, 2006.

[13] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. Technical report (draft), School of EECS, Oregon State University, 2012.

[14] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. The MIT Press, 1999.

[15] 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. of ICSE*, pages 335–344. ACM, 2010.

[16] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[17] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proc. of FSE*, pages 243–252. ACM, 2009.

[19] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. Understanding Linux Feature Distribution. In *Proc. of MISS*, pages 15–19. ACM, 2012.

[20] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *ACM Trans. Software Engineering and Methodology*, 21(1):6:1–6:27, 2011.

[21] B. Garvin and M. Cohen. Feature Interaction Faults Revisited: An Exploratory Study. In *Proc. of ISSRE*, pages 90–99. IEEE, 2011.

[22] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[23] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. of ICSE*, pages 311–320. ACM, 2008.

[24] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. of GPCE*, pages 157–166. ACM, 2009.

[25] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Trans. Software Engineering and Methodology*, 21(3):1–39, 2012.

[26] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proc. of OOPSLA*, pages 805–824. ACM, 2011.

[27] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *Proc. of OOPSLA*. ACM, 2012.

[28] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *Proc. of FOSD*, pages 1–8. ACM, 2012.

[29] D. Kuhn, D. Wallace, and A. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. Software Engineering*, 30:418–421, 2004.

[30] K. Lauenroth, S. Toehning, and K. Pohl. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. of ASE*, pages 269–280. IEEE, 2009.

[31] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. of ICSE*, pages 105–114. ACM, 2010.

[32] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code. In *Proc. of AOSD*, pages 191–202. ACM, 2011.

[33] M. Lochau, S. Oster, U. Goltz, and A. Schürr. Model-based Pairwise Testing for Feature Interaction Coverage in Software Product Line Engineering. *Software Quality Journal*, pages 1–38, 2011. Online first.

[34] M. Mendonça, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proc. of SPLC*, pages 231–240. ACM, 2009.

[35] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal*, 2011. Online first.

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

[37] H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proc. of ASE*, pages 347–350. IEEE, 2008.

[38] N. Siegmund, S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake. Predicting Performance via Automated Feature-Interaction Detection. In *Proc. of ICSE*, pages 167–177. IEEE, 2012.

[39] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the Linux Kernel a Software Product Line? In *Proc. of OSSPL*, 2007.

[40] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Feature Consistency in Compile-Time Configurable System Software. In *Proc. of EuroSys*, pages 47–60. ACM, 2011.

[41] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-scale System Software. *SIGOPS Oper. Syst. Rev.*, 45(3):10–14, 2012.

[42] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. of GPCE*, pages 95–104. ACM, 2007.

[43] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis Strategies for Software Product Lines. Technical Report FIN-004-2012, University of Magdeburg, 2012.

[44] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-Based Theorem Proving for Deductive Verification of Software Product Lines. In *Proc. of GPCE*. ACM, 2012. To appear.

[45] M. Tomita. LR Parsers for Natural Languages. In *Proc. Int. Conf. Computational Linguistics (COLING)*, pages 354–357. ACL, 1984.

[46] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29:366–427, 1997.