Scalable Analysis of Variable Software

Jörg Liebig, Alexander von Rhein
University of Passau, Germany

Christian Kästner
Carnegie Mellon University, USA

Sven Apel, Jens Dörre, Christian Lengauer
University of Passau, Germany

ABSTRACT

The advent of variability management and generator technology enables users to derive individual variants from a variable code base based on a selection of desired configuration options. This approach gives rise to the generation of possibly billions of variants that, however, cannot be efficiently analyzed for errors with classic analysis techniques. To address this issue, researchers and practitioners usually apply sampling heuristics. While sampling reduces the analysis effort significantly, the information obtained is necessarily incomplete and it is unknown whether sampling heuristics scale to billions of variants. Recently, researchers have begun to develop variability-aware analyses that analyze the variable code base directly exploiting the similarities among individual variants to reduce analysis effort. However, while being promising, so far, variability-aware analyses have been applied mostly only to small academic systems. To learn about the mutual strengths and weaknesses of variability-aware and sampling-based analyses of software systems, we compared the two strategies by means of two concrete analysis implementations (type checking and liveness analysis), applied them to three subject systems: Busybox, the x86 Linux kernel, and OpenSSL. Our key finding is that variability-aware analysis outperforms most sampling heuristics with respect to analysis time while preserving completeness.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*

General Terms

Experimentation

Keywords

Software Product Lines, C Preprocessor, Type Checking, Liveness Analysis, Variability-aware Analysis

ESEC/FSE'13, August 18–26, 2013, Saint Petersburg, Russia
Copyright 2013 ACM 978-1-4503-2237-9/13/08...$15.00
http://dx.doi.org/10.1145/2491411.2491437

1. INTRODUCTION

Generator-based approaches have proved successful for the implementation of variable software systems [1, 15]. For example, the Linux kernel can be configured by means of about 10 000 compile-time configuration options [39], giving rise to possibly billions of variants that can be generated and compiled on demand. While advances in variability management and generator technology facilitate the development of variable software systems with myriads of variants, this high degree of variability is not without cost. How could we analyze all possible variants for errors? Unfortunately, classic analyses look at individual variants and do not scale in the presence of the exponential number of variants that can be typically generated from a variable system. For systems such as the Linux kernel, it is not even possible to generate all variants to analyze them separately, because they are so many—more than the estimated number of atoms in the universe [40].

The idea of sampling is to select a reasonable set of variants to be analyzed using traditional analysis techniques. Many different sampling techniques have been proposed [22,41], and their application has proved useful in various scenarios [36,38]. Although analysis time can be reduced significantly, the information obtained is necessarily incomplete, since only a subset of all variants is checked.

Recently, researchers have begun to develop a new class of analyses that are *variability-aware* [43]. The key idea is to not generate and analyze individual variants separately, but to directly analyze the variable code base before variant generation, with the help of configuration knowledge. In the case of the Linux kernel, variable code is implemented by means of conditional-inclusion directives (a.k.a. `#ifdefs`), and variability-aware analyses analyze the variable code directly, instead of applying the generator (the C preprocessor) to generate the plain C code of individual kernel variants. Variability-aware analysis requires more effort than traditional analysis of a single system, because all local variations need to be considered; however, and this is the key success factor, variability-aware analysis takes advantage of the similarities among variants and avoids analyzing common code over and over again.

There are several proposals for variability-aware analyses in the literature, including parsing [25], type checking [2, 24, 26, 42], data-flow analysis [8, 9], model checking [3, 4, 14, 30], and deductive verification [44]. However, while this work is promising, variability-aware analyses (beyond parsing) have not been applied to large-scale, real-world systems so far; previous work concentrated mostly either on formal foun-

dations or is limited with respect to practicality (evaluated with academic case studies only), as we discuss in Section 6.

Despite the foundational previous work, it is unclear whether variability-aware analysis scales to large systems, as it considers all code and all variations of a system simultaneously. Since sampling is still a de-facto standard for analyzing variable software systems in practice, we explore the feasibility and scalability of both sample-based and variability-aware analysis in practice empirically. To this end, we have developed two fully-fledged variability-aware analyses for C: type checking and liveness analysis (a data-flow analysis). We applied each of them to three real-world, large-scale variable systems: the Busybox tool suite, the Linux kernel, and the cryptographic library OpenSSL. In terms of scalability, we compare the variability-aware analyses to state-of-the-art sampling strategies used in practice (generating all variants is not even possible in reasonable time in our case studies). We found that variability-aware analyses scale well—even outperform some of the sampling strategies—while still providing complete information on all system variants.

Beside quantitative results, we report on our experience with making variability-aware analyses ready for the real world, and we discuss insights into the development of variability-aware analyses in general. These insights subsume existing studies on variability-aware analysis techniques and they can guide the development of further analyses.

Overall, we make the following contributions:

- An introduction to the problem of analyzing variable software, including possible solutions such as sampling and variability-aware analysis.
- An experience report of how to implement scalable variability-aware analysis for preprocessor-based systems, based on an existing variability-aware parsing framework [25].
- A discussion of general patterns of variability-aware analyses that can guide the development of further analyses.
- A series of experiments that compare the performance of variability-aware analysis with the performance of state-of-the-art sampling strategies based on three real-world, large-scale subject systems.
- A reflection of our experience with applying variability-aware and sampling-based analyses in practice, and of challenges we encountered in our investigation.

The subject systems and all experimental data are available on a supplementary website: `http://fosd.net/vaa`; the analysis implementations are part of the TypeChef project: `http://ckaestne.github.com/TypeChef`.

2. PREPROCESSOR-BASED VARIABILITY

Before we get to sampled-based and variability-aware analyses and their comparison, we introduce the development of variable software using the C preprocessor CPP. CPP is a frequently applied tool for the development of variable software [31]. It provides several features to implement variable code fragments using conditional-inclusion macros (a.k.a. `#ifdef`s). For instance, our running example in Figure 1 contains three variable pieces of code: an alternative macro expression (Line 1-3), an optional function parameter (Line 6), and an optional statement (Line 13). The in-/exclusion of such annotated code is controlled by the values of *configuration options* (here: A and B) that can be combined using logical operators.

```
1  #ifdef A   #define EXPR (a<0)
2  #else      #define EXPR 0
3  #endif
4
5  int r;
6  int foo(int a #ifdef B , int b #endif) {
7    if (EXPR) {
8      return -b;
9    }
10   int c = a;
11   if (c) {
12     c += a;
13     #ifdef B c += b; #endif
14   }
15   return c;
16 }
```

Figure 1: Running example in C with variability expressed in the form of preprocessor directives (Lines 1–3, 6, and 13); for brevity, we underlined and integrated `#ifdef` directives inside single code lines.

In most cases not all combinations of configuration options of a system are valid, so developers use *variability models* to express relations between configuration options and define which combinations of configurations options are valid. One widely used tool in practice to express variability models is Kconfig,[1] which is used for example in the development of Linux and Busybox. Variability models can be transformed into boolean formulas, which enables efficient reasoning about them using current SAT-solver technology [34].

3. SAMPLE-BASED ANALYSIS

Sample-based analysis has its origin in early approaches of testing software [35]. Due to the sheer size and complexity of real-world systems (the number of variants can grow exponentially with the number of configuration options), a brute-force approach of analyzing all variants in isolation is not feasible. Hence, developers typically analyze only a subset of variants, called the *sample set*, using off-the-shelf analysis tools. The idea is that, even though we cannot analyze all variants individually, we can still strive for analyzing a *representative* sample set to be able to draw informed conclusions about the entire set of variants (e.g., in terms of defect probability).

The sample set is selected by a *sampling heuristic*, either by a domain expert or by an algorithm. Researchers and practitioners have proposed different sampling heuristics, some of which require sophisticated upfront analyses. We selected four that are common in practice: single configuration, random, code coverage, and pair-wise coverage, as described below. For an overview of other sampling strategies, see a recent survey [35].

Single-conf heuristic. The simplest sampling heuristic (*single conf*) is to analyze only a single representative variant that enables most, if not all, of the configuration options of the variable system. Typically, the variant is selected manually by a domain expert. The strength of this heuristic is that one needs to analyze only a single variant, hence it is fast. By selecting many configuration options, the heuristic tries to cover a large part of the system's code, however, it cannot cover mutually exclusive code pieces or intricate interactions specific to individual combinations of configuration options [19]. For the code snippet in Figure 2, we can create a configuration that enables all configuration options:

[1] `https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt`

header.h

```
1  #ifdef A
2  int foo(int a) {...}
3  #else
4  int foo2(int a) {...}
5  #endif
6  #ifdef B
7  int bar(int i, int j) {
8    ...
9  }
10 #endif
```

main.c

```
11 #include "header.h"
12 int main() {
13 #ifdef A
14   bar(2,3);
15   foo2(3);
16 #endif
17 #ifdef C
18   print("done");
19 #endif
20 }
```

Figure 2: C code with preprocessor directives; the header file (left) contains one alternative and one optional definition; the C file (right) uses the definitions of the header file.

$\{A, B, C\}$. Since code fragments in Lines 2 and 4 are mutually exclusive, a single configuration will cover only one of them, leaving Line 4 uncovered, in our case.

According to Dietrich et al. [17], in the Linux development community, it is common to analyze only one predefined variant with most configuration options selected, called *allyesconfig*. Similarly, many software systems come with a default configuration that satisfies most users and that usually includes many configuration options.

Random heuristic. A simple approach to select samples is to generate them randomly. For example, in a project with n features, we could make n random, independent decisions whether to enable the corresponding configuration option. In projects with constraints between options, we would discard variants with invalid configurations and would keep the remaining variants as our sample. Random sampling is simple and scales to an arbitrary sample size. Alternatively testing can continue until time or money runs out. Random sampling does not adhere to any specific coverage criterion, though.

Code-coverage heuristic. The *code-coverage* heuristic is inspired by the statement-coverage criterion used in software testing [47]. In contrast to software testing, the code-coverage heuristic aims at variant generation not code execution [41]. The goal of this heuristic is to select a minimal sample set of variants, such that every lexical code fragment of the systems' code base is included in, at least, one variant. In contrast to the single-conf heuristic, the code-coverage heuristic covers mutually exclusive code fragments. However, note that including each code fragment at least once, does not guarantee that all possible combinations of individual code fragments are considered. For the code snippet in Figure 2, two configurations $\{A, B,\ C\}$ and $\{\}$—the first selecting all options and the second deselecting them all—would be sufficient to include every code fragment in at least one variant. However, it would not help to detect the compilation error, i.e., calling `bar` when A is selected but B is not.

Although there is an algorithm to compute an optimal solution (a minimal set of variants) by reducing the problem to calculating the chromatic number of a graph, this algorithm is NP-complete and by far too slow for our case studies.[2] Instead, we resort to the conservatively approximated solution of Tartler et al. [41], which speeds up the computation of the sample set significantly at the cost of producing a sample set that is possibly larger than necessary.

[2]For more details on the optimal algorithm, see `https://github.com/ckaestne/OptimalCoverage`.

A subtle problem of this heuristic arises from the issue of how to treat header files. When computing the sample set of two variants in our example, we have implicitly assumed that we analyze coverage in the main file *and* the included header file together. Due to the common practice of including files that themselves include other files, a single `#include` statement in the source code can bloat the code base of a single file easily by an order of magnitude, something we frequently observed in Linux, in which on average 300 header files are included in each C file [25]. In addition, header files often exhibit their own variability, not visible in the C file without expanding macros. Furthermore, some header files may be included only conditionally, depending on other `#ifdef` directives, such that, for a precise analysis of all header code, sophisticated analysis mechanisms become necessary (e.g., using symbolic execution of the preprocessor code) [20, 25, 29]. This explosion and cost can make precise analyses that include header files unpractical or infeasible, even with Tartler's approximate solution. Therefore, we distinguish two strategies for code coverage: (1) covering variability only in C files and (2) covering variability in C files and their header files. Analyzing only the main file above, a single configuration $\{A, C\}$ would be sufficient to cover all code fragments of the main file.

Pair-wise heuristic. The *pair-wise* heuristic is motivated by the hypothesis that many faults in software systems are caused by interactions of, at most, two configuration options [10, 28, 37, 38]. Using the pair-wise heuristic, the sample set contains a minimal number of samples that cover all pairs of configuration options, whereby each sample is likely to cover multiple pairs. For the code in Figure 2, with three optional and independent features, a pair-wise sample set consist of 4 configurations: $\{A, B\}$, $\{A, C\}$, $\{B, C\}$, and $\{\}$.

The computation of pair-wise sample sets is not trivial if constraints, such as A implies B or C, exist in a variability model; in fact it is NP-complete (similar to the minimum set cover problem) [21]. Hence, existing tools apply different conservative approximations to make the computation possible for large systems with many configuration options. For our experiments, we use SPLCATool[3] by Johansen et al., which computes pair-wise samples using covering arrays [21]. The computed sample covers all pair-wise interactions that occur in a given system, but is not guaranteed to be minimal.

4. VARIABILITY-AWARE ANALYSIS

Variability-aware analysis (also known as family-based analysis [43]) takes advantage of the similarities between the variants of a system in order to speed up the analysis process. Although individual variability-aware analyses differ in many details [43], an idea that underlies all of them is to analyze code that is shared by multiple variants only once. To this end, variability-aware analyses do not operate on generated variants, but on the raw code artifacts that still contain variability and available configuration knowledge, prior to the variant-generation step.

In our context, variability-aware analyses work directly on C code that still contains preprocessor directives. As code artifacts with preprocessor directives cannot be processed directly by standard analyses, an analysis has to be prepared—it has to be made variability-aware (plain preprocessing does

[3]`http://heim.ifi.uio.no/martifag/splcatool`

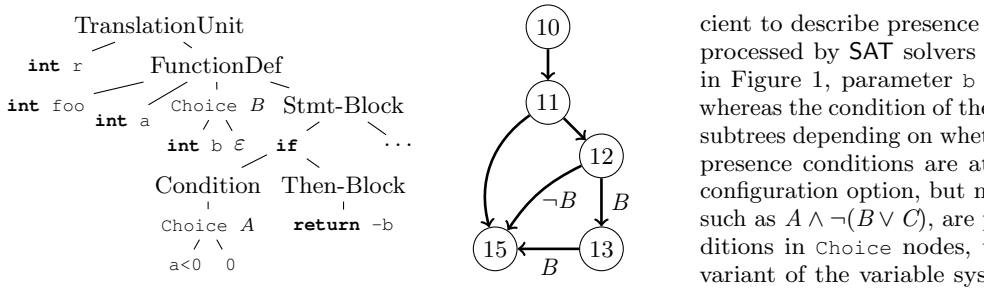

Figure 3: Excerpt of the corresponding variable AST (left) and CFG (right) for our running example of Figure 1.

not help, as it removes variability). Technically, one has to adapt existing analyses to empower them to work with variable code fragments. This approach has been pursued for adapting existing type-checking, model-checking, and testing techniques to variable systems [2, 14, 24, 26, 27, 30].

Although variability-aware analysis has been applied in academic projects, showing promising performance improvements by orders of magnitude, apart from parsing [25, 26], it has never been applied to real-world software systems at the scale of Linux. Since many industrial software systems are implemented in C and use `#define` and `#ifdef` directives (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.

For the purpose of presenting and discussing the results of our empirical study, we explain in the remaining section our design decisions in implementing variability-aware type checking and liveness analysis. Note that we implemented the liveness analysis for the purpose of our study. It is the first variability-aware data-flow analysis for C. It scales to real-world, large-scale systems such as Linux.

Variable abstract syntax trees. Many static analyses are performed on abstract syntax trees (ASTs). Since we want to analyze an entire variable software system, we have to construct an abstract syntax tree that covers all variants of a system and the corresponding configuration knowledge.

The desired *variable AST* is like a standard AST, but it contains additional nodes to express compile-time variation. A `Choice` node expresses the choice between two or more alternative subtrees (similar to ambiguity nodes in GLR parse forests [45]; explored formally in the choice calculus [18]). For example, `Choice(`A`,a<0,0)` (Figure 3, left) expresses the alternative of two expressions `a<0` and `0` that is controlled by configuration option A. The choice node is a direct representation of the variable expression in our running example (Figure 1; `#ifdef`s on Line 1 to 3 and their usage on Line 7). One alternative of a choice may be empty (ε; see Figure 3), which makes the other, in fact, optional. In principle, we could use a single `Choice` node on top of the AST with one large branch per variant; but a variable AST is more compact, because it shares parts that are common across multiple variants (e.g., in Figure 3, we store only a single node for the declaration of `r`, and a single node for the function name `foo`, which are shared by all variants). It is this sharing and keeping variability local, which makes variability-aware analysis faster than a brute-force approach (see the discussion at the end of this section).

To reason about variability, we need to represent configuration knowledge. To this end, we annotate subtrees with *presence conditions*. Propositional formulas are sufficient to describe presence conditions and can be efficiently processed by SAT solvers and BDDs [34]. As an example, in Figure 1, parameter `b` is included only if B is selected, whereas the condition of the `if` statement has two alternative subtrees depending on whether A is selected. In our example, presence conditions are atomic and refer only to a single configuration option, 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 code of every variant of the variable system, given the configuration for that variant. Compact representations of variable ASTs in this or similar forms are commonly used in variability-aware analyses [9, 18, 24, 25, 46].

The construction of a variable AST from a real-world software system such as Linux is not trivial. Whereas parsing preprocessed C code of an individual variant is well established, parsing a variable system with `#ifdef`s is challenging. To make matters worse, in the C preprocessor, conditional-compilation directives (`#ifdef`) interact with the build system, with macros (`#define`), and with file-inclusion facilities (`#include`), across file boundaries, in intricate ways. In previous work, we solved the parsing challenge and implemented a sound and complete parser as part of the TypeChef project [25], incorporating prior work on variability-model extraction and build-system analysis [6, 7]. Variability-aware parsing always considers a C file with all its header files. It is this recent breakthrough in parsing that now finally enables the analysis of real-world C code with `#ifdef` variability. For details on the parser, see the corresponding publication [25].

In the remainder of this paper, we use this parser framework as a black box and work on the resulting variable ASTs.

Variability-aware type checking. A standard type-checking algorithm for C traverses the AST, collects declarations in a symbol table, and attempts to assign proper types to all expressions (`getType:Map[Name,Type]`→`Expr`→`Type`). In principle, a variability-aware type checker works similar, but covers all variants; hence it must be aware of variability in each of the following three steps.

First, a symbol (variable, function, etc.) may only be declared in some variants, or it may even have alternative types in different variants. Therefore, we extend the symbol table (similar to the proposal of Aversano et al. [5]), such that a symbol is no longer mapped to a single type, but to a conditional type (a choice of types or ε; `VST = Map[Name,Choice[Type]]`). We illustrate a possible encoding of a conditional symbol table for our example in Table 1. If a symbol is declared in all variants, we do not need `Choice` nodes; however, if a symbol is declared in a subtree of the AST that is only reachable given a certain presence condition, we include the symbol and type in the symbol table only under that condition. Similarly, we may declare a symbol with different types in different variants. In our running example, function `foo` has two alternative types, depending on whether B is selected. Similarly, we made the table for structures and enumerations in C variability-aware.

Second, during expression typing, we assign a variable type (choices of types) to each expression (`getType:VST`→`Expr`→`Choice[Type]`), where already looking up a name in a symbol table may return a variable type. For example, when checking that 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,

Table 1: Conditional symbol table at Line 6 of our running example of Figure 1.

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

we report a type error and pinpoint it to a subset of variants, as characterized by a corresponding presence condition. Similarly, an assignment is only valid if the expected (variable) type is compatible with the provided (variable) type in *all* variants. Therein, an operation on two variable types can, in the worst case, result in the Cartesian product of the types in either case of the choice, resulting in a variable type with many alternatives. All other type rules are essentially implemented along the same lines. In our running example, we would report a type error in Line 8, because symbol `b` cannot be resolved in variants without B (see Table 1).

Third, we can use the variability model of a system (if available) to filter all type errors that occur only in invalid variants. To this end, we simply check whether the presence condition of the type error is satisfiable when conjoined with the variability model (checked with a standard SAT solver).

Variable control-flow graphs. For most data-flow analyses, we need to construct a control-flow graph (CFG), which represents all possible execution paths of a program. Nodes of the CFG correspond to instructions in the AST, such as assignments and function calls; edges correspond to possible successor instructions according to the execution semantics of the programming language. 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 variants of systems. To create a CFG for a single program, we need to compute the successors of each node (`succ: Node→List[Node]`). In the presence of variability, the successors of a node may differ in different variants, so we need a variability-aware successor function that may return different successor sets for different variants (`succ: Node→Choice[List[Node]]`, or, equivalently, but with more sharing, `succ: Node→List[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 of Figure 1. In Figure 3 (right), we show an excerpt of the corresponding variability-aware CFG (node numbers refer to line numbers of Figure 1). The successor of the instruction `c += a` in Line 12 depends on the configuration: if B is selected, statement `c += b` in Line 13 is the direct successor; if B is not selected, `return c` in Line 15 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 variants, in which the source node is present (checked with a SAT solver or BDDs). By evaluating the presence conditions on edges, we can reproduce the CFG of each variant. [4]

[4]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 3, both nodes 12 and 15 may follow node 11). Such sound but incomplete approximation is standard practice to make static analysis tractable or decidable. However, we do not want to lose precision for static variability. 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, which is in line with prior work on CFG in variable Java programs [8, 9].

Table 2: Result of liveness computation of our running example of Figure 1; $\mathtt{b}_B$ is shorthand for `Choice(`B`,b,`ε`)`.

Line	Uses	Defines	In	Out
10	{a}	{c}	{a,$\mathtt{b}_B$}	{a,$\mathtt{b}_B$,c}
11	{c}	{}	{a,$\mathtt{b}_B$,c}	{a,$\mathtt{b}_B$,c}
12	{a,c}	{c}	{a,$\mathtt{b}_B$,c}	{$\mathtt{b}_B$,c}
13	{$\mathtt{b}_B$,$\mathtt{c}_B$}	{$\mathtt{c}_B$}	{$\mathtt{b}_B$,$\mathtt{c}_B$}	{$\mathtt{c}_B$}
15	{c}	{}	{c}	{}

Variability-aware liveness analysis. Liveness analysis is a classic data-flow analysis for the computation of variables that are live (that may be read before being written again) for a given statement. Its result can be used, for example, to conservatively detect dead code. In real-world systems, warnings about dead code that occurs only in specific variants are interesting for maintainers; corresponding problems are regularly reported as bugs.[5] So, again, our goal is to make liveness analysis variability-aware.

Liveness analysis is based on two functions: `uses` computes all variables read, and `defines` computes all variables written to. While, in traditional liveness analysis, both functions return sets of variables, in variability-aware liveness analysis, both return sets that may vary depending on the configuration (a choice of sets or a set with optional entries). The computation of liveness is a fixpoint algorithm that uses two functions, `in` and `out`, which compute variables that are live at respectively after the current statement. The result of `in` and `out` is variable again, and the signatures of both change from `Node→Set[ID]` to `Node→Set[Choice[ID]]`, where `ID` represents the identifier of a live variable.

In Table 2, 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 15. Considering the control flow from Line 10 to 13 ($10 \rightarrow 11 \rightarrow 12 \rightarrow_B 13$), in the declaration statement on Line 10, variable `a` is live, whereas `b` is only live if B is selected.

Principle: Keeping variability local. Although we have introduced how variability-aware analyses work, we have not explained why we expect that they can be executed efficiently even for real-world systems with myriads of possible variants. The key is *keeping variability local.* Parsing already preserves sharing in the AST and keeps variability local (code without `#ifdef` directives is represented only once, since it is common to all variants; choices are introduced only locally where code differs between variants). We preserve this sharing and locality throughout our analyses, as far as possible. Specifically, three patterns emerged that maximize sharing: *late splitting*, *local variability representation*, and *early joining*.

First, *late splitting* means that we perform the analysis without variability until we encounter it. For example, type

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

checking processes the declaration of symbol `r` in Line 5 only once, and adds it to the symbol table only once, whereas a brute-force strategy or a sampling-strategy would process this declaration multiple times. Also, when we use symbol `r` later, it has only one type. Variability-aware analyses only split and consider smaller parts of the variant space when they 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 performed only on demand [13].

Second, *local variability representation* aims at keeping 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 store a single type for `a` or `r`, independently of B.

Third, *early joining* attempts to *join* intermediate results as early as possible. For example, if we have a choice of two identical types `Choice(`A`,int,int)`, we can simply join them to `int` for further processing. 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 into 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 configuration options that occur in different parts of the AST if they actually produce different (intermediate) results when combined, otherwise the results remain orthogonal.

Note that the three patterns of late splitting, local variability representation, and early joining apply to any kind of variability-aware analysis; although not always made explicit, these patterns can also be observed in other variability-aware analyses [4, 9, 25, 27].

5. EMPIRICAL STUDY

To evaluate feasibility and scalability of different analysis strategies, we attempt to analyze three real-world, and large-scale systems—a fact that substantially increases external validity, compared to previous work, which concentrated mostly on formal foundations, made limiting assumptions, or relied on comparatively small and academic case studies (see Section 6). We use both state-of-the-art sampling heuristics (*single conf*, *code coverage* with and without headers, and *pair-wise*), as introduced in Section 3. We apply both type checking and liveness analysis. We report our experience and perform rigorous performance measurements.

5.1 Hypotheses and Research Questions

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

1. *Variability-aware vs. single conf:* Analyzing all variants simultaneously using variability-aware analysis is likely slower than analyzing a single variant that covers most configuration options. The reason is that the variable program representation covering all variants is larger than the program representation of any single variant, including the largest possible variant.

   H$_1$ The execution times of variability-aware type checking and liveness analysis are *larger* than the corresponding times of analyzing the variants derived by single-conf sampling.

2. *Variability-aware vs. pair-wise:* While previous work has shown that pair-wise sampling is a reasonable approximation of the analysis of all variants [33], our previous experience is that it can still generate quite large sample sets. Hence, we expect a variability-aware analysis to outperform pair-wise sampling:

   H$_2$ The execution times of the variability-aware type checking and liveness analysis are *smaller* than the corresponding times of analyzing the variants derived by pair-wise sampling.

3. *Variability-aware vs. code coverage:* With respect to the comparison of variability-aware analysis and code-coverage sampling, we cannot make any informed guesses with respect to analysis time. Code-coverage sampling generates sample sets depending on the usage of configuration options in the analyzed C files. Since we do not know details about the code, we cannot predict how many variants will be generated and how large these will be. Therefore, we pose a research question instead. Specifically, the influence of variability that occurs in header files is unknown and therefore we look at two different variants of code coverage: one including header files and one without.

   RQ$_1$ How do the execution times of variability-aware type checking and liveness analysis compare to the times for analysis of the variants derived by code-coverage sampling (with and without header files)?

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

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

   The background for questioning scalability is that variability-aware analysis reasons about variability by solving SAT problems or by using BDDs during analysis, and so depends on the degree of sharing that is possible in practice (see Section 4). Generally, SAT is NP-complete, but previous work suggests that the problems that arise in variability-aware analysis are typically tractable for state-of-the-art SAT solvers [34] and BDDs, and that caching can be an effective optimization [2].

5.2 Subject Systems

To test our hypotheses and to answer our research questions, we selected three subject systems. We looked for publicly available systems (for replicability), that are of substantial size, actively maintained by a community of developers, used in real-world scenarios, and that implement substantial compile-time variability with the C preprocessor. The systems must provide at least an informal variability model that describes configuration options and their valid combinations [31]. In this context, we would like to acknowledge the pioneering work on variability-model extraction [7, 40] and build-system analysis [6], which enabled us, for the first time, to conduct variability-aware analysis on substantial, real-world systems that are an order of magnitude larger then previous work on Java-based subjects [4, 9, 24].

- The *Busybox* tool suite reimplements most standard Unix tools for resource-constrained systems. With 792 configuration options it is highly configurable; most of the options refer to independent and optional subsystems; the variability model in conjunctive normal form has 993 clauses. We use Busybox version 1.18.5 (522 files and 191 615 lines of source code).
- The *Linux kernel* (x86 architecture, version 2.6.33.3) is an operating-system kernel with millions of installations worldwide, from high-end servers to mobile phones. With 6 918 configuration options it is highly configurable. In a previous study, we identified the Linux kernel as one of the largest and most complex (w.r.t. variability) publicly available variable software systems [31]. It has 7 691 source code files with 6.5 million lines of code. Note that already the variability model of Linux is of substantial size: the corresponding extracted formula in conjunctive normal form has over 60 000 variables and nearly 300 000 clauses; a typical satisfiability check requires half a second on a standard computer.
- The cryptographic library *OpenSSL* implements different protocols for secure Internet communication. OpenSSL can be tailored to many different platforms, and it provides a rich set of 589 configuration options. We analyze OpenSSL version 1.0.1c with 733 files and 233 450 lines of code. Since OpenSSL does not come with a formal variability model like Busybox or Linux, we extracted a variability model based on manual analysis. The resulting variability model has 15 clauses.

5.3 Experience with Sampling

Before we discuss the performance measurements, we would like to share our experience with sampling approaches, which were surprising to us. We expected that contemporary sampling tools can quickly compute representative sample sets. However, we found that deriving samples at this scale is far from trivial, and that we even failed to compute sample sets for some sampling heuristics (code coverage with headers for Linux), and that the computation time already takes up to several hours (e.g., pair-wise: >20 h for Linux).

The single-conf heuristic worked well. Linux has a commonly used configuration *allyesconfig*, which is maintained by the community and frequently used for analysis purposes.[6] For Busybox and OpenSSL, we created large configurations by selecting as many configuration options as possible.

Random sampling already proved problematic. Both Busybox and Linux have variability models with many constraints. In 1 000 000 random configurations, there was not even a single configuration that fulfilled all variability-model constraints. Random sampling was only a possibility for OpenSSL, which has a comparably sparse variability model ($\sim 3\,\%$ of randomly generated configurations were valid). Busybox developers actually uses a skewed form of random sampling, in which, one by one, a random value is selected for every configuration option that is not yet decided by constraints of other options. This approach depends strongly on variable ordering and violates the developer intuition about random selection. Due to these sampling problems in the presence of constraints, we did not consider random sampling any further.

For the coverage-based and the pair-wise heuristics, we observed that the generation of samples took considerable time (times for sample generation are not part of our reported analysis times below) and was not possible in all cases (in particular for Linux).

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

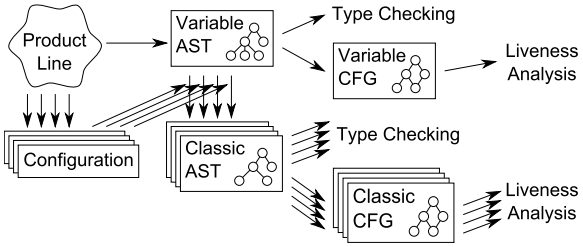

Figure 4: Experimental setup.

In contrast to all other heuristics, heuristics based on code coverage need to investigate every file separately (and optionally all their header files). We reimplemented the conservative algorithm of Tartler et al. for this task [41] in two variants: one including header files (code coverage) and one without (code coverage NH). When headers are included and macros are considered, the coverage analysis easily needs to process several megabytes of source code per C file [25]. Surprisingly, already the times needed for code-coverage sample computation exceeded the times for performing variability-aware type checking and liveness analysis.

To compute the pair-wise sample set we only found one research tool (SPLCATool), which is able to compute a complete set of pair-wise configurations for a given feature model (see Section 3). SPLCATool did reasonably well for Busybox and OpenSSL, but the larger configuration space of Linux made the computation of the sample set very expensive. Also in this case, the computation time exceeded the times for performing variability-aware liveness analysis.

5.4 Experimental Setup

We use TypeChef as underlying parsing framework. As explained in Section 4, 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 these variable ASTs, and they are integrated into and deployed as part of the TypeChef project.

To avoid bias due to different analysis implementations, we use our infrastructure for variability-aware analysis also for the sample-based analyses. To this end, we generate individual variants (ASTs without variability) based on the sampling heuristics. We create an AST for a given configuration by pruning all irrelevant branches of the variable AST, so that no `Choice` nodes remain. As 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*.

As liveness analysis is intraprocedural, it would have been possible and more efficient to apply sampling to individual functions and not to files, as done by Brabrand et al. for Java product lines [9]. 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 of Figure 1, we would not even have noticed that function `foo` is affected by A, because variability comes from variable macros defined outside the function. Variable macros defined in header files are very common in C code [25].

Furthermore, we implemented only an imprecise liveness analysis, since the analysis is performed without a real analysis question (e.g., which code is dead). The strategy of abstraction is a common approach in model checking [12], to handle the complexity of a system and to make the analysis feasible in the first place. In particular, during liveness computation, our algorithms perform SAT checks without taking the variability model of the analyzed system into account. This way, the computation is faster and still complete, though, false positives may occur. False positives can be eliminated easily after a refinement step (i.e., using the variability model in SAT checks), so that only valid execution paths are taken into account [12].

In Figure 4, we illustrate the experimental setup. Depending on the sampling heuristics, one or multiple configurations are checked. For each file of the three subject systems, we measured the time spent in type checking and liveness analysis, each using the variability-aware approach and the three sampling heuristics (the latter consisting of multiple internal runs)—in total, four analyses per subject system: one variability-aware + three respectively four sampling (with and without header files for code coverage).

We ran all measurements on Linux machines (Ubuntu 12.04) with Intel Core i7-2600, 3.4 GHz, and 16/32 GB RAM. We configured the Java JVM with upto 8 GB RAM for memory allocation. To reduce measurement bias, we minimized disk access by using a ramdisk and warmed up the JVM by running an example task before the actual measurement run. However, due to just-in-time compilation and automatic garbage collection inside the Java JVM, measurements of analysis times might slightly differ for similar inputs. We could not mitigate this problem with repetitive analysis runs, since the setup already takes weeks to finish, but we believe that the large number of files produces still a reliable result.

Analysis procedure and reporting. We report performance measurements as total times for the entire analysis and additionally graphically as the distribution of analysis times for individual files in the project, using notched boxplots on a logarithmic scale. We highlight the median (over all files) of the variability-aware analyses with a vertical line, to simplify comparison with the medians of the sample-based analyses.

Furthermore, we provide the number of analyzed configurations for each of the sample-based analyses (below the name of the analysis, 'configs per file' or short 'c.p.f.'). Single conf requires the same number of variants for each file (because they are based on global knowledge of the variability model only), whereas code coverage and pair-wise[7] require different numbers of variants in different files, which we provide in terms of mean±standard deviation. We evaluate all research hypotheses with paired t-tests at a confidence level of 95 %.

5.5 Results

In Table 3, we show the measurement results for each analysis and subject system. We report sequential times, though parallelization would be possible in all cases, because all files are analyzed in isolation. In Figures 5, 6, and 7, we plot the distributions of analysis times for Busybox, Linux, and OpenSSL, as described in Section 5.4.

Table 3: Total times for analyzing the subjects with each approach (time in seconds, with three significant digits).

		Type checking	Liveness analysis
Busybox	Single conf	40.3	19.4
	Code coverage NH	107	67.1
	Code coverage	2 030	985
	Pair-wise	1 110	544
	Variability-aware	223	97.3
Linux	Single conf	5 060	2 560
	Code coverage NH	33 000	22 500
	Pair-wise	569 000	380 000
	Variability-aware	73 500	13 900
OpenSSL	Single conf	64.1	24.5
	Code coverage NH	86.9	37.6
	Code coverage	388	145
	Pair-wise	1 110	455
	Variability-aware	228	44.7

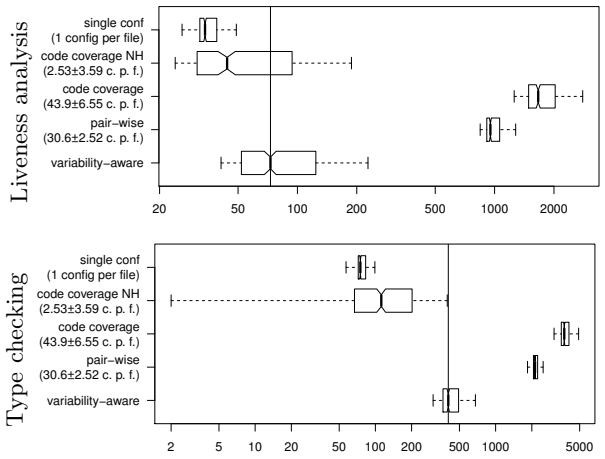

Figure 5: Distribution of analysis times for Busybox (times in milliseconds; logarithmic scale).

In all subject systems and for both type checking and liveness analysis, the variability-aware approach is slower than single-conf sampling (H_1; statistically significant), and it is faster than pair-wise sampling (H_2; statistically significant). The results regarding code-coverage sampling (H_3) are mixed: variability-aware analysis is faster for liveness analysis in Linux, slower for liveness analysis in Busybox and OpenSSL and for type checking of Linux and OpenSSL (statistically significant). We observe that code coverage without header files (NH) is often faster than with header files and sometimes it even outperforms single-conf sampling. The reason for this is that many `#ifdef`s occur in header files, something that is neglected in code-coverage sampling NH. Single-conf sampling considers variability in header files such that it may select a larger configuration with additional header code, which is potentially unnecessary, and is, therefore, slower than code-coverage sampling NH. Table 4 summarizes the actual speedups of all comparisons.

It is worth noting that we did not find any confirmed defects during our experiments. For Linux, we found a defect already fixed in subsequent releases; for Busybox, we found and reported several defects in earlier versions that have been fixed in the current version, which we used for our experiments.[8]

[7] In addition to the given variability model, the build system of Linux defines presence conditions for individual files. So, as for Linux, each file has its own variability model. Nevertheless, we use the global variability model for the computation of pair-wise sample sets. Since, the sample may contain configurations that are not valid for a file, the overall number of analyzed configurations for a file decreases.

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

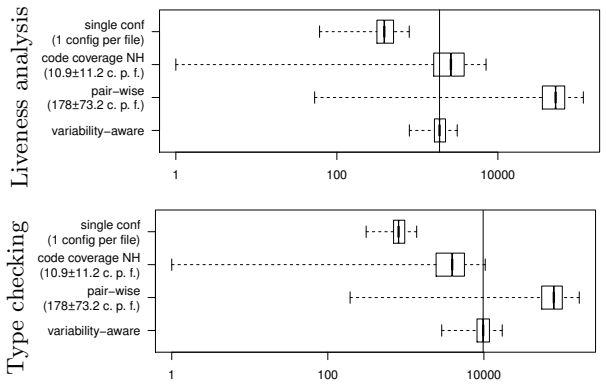

Figure 6: Distribution of analysis times for Linux (times in milliseconds; logarithmic scale).

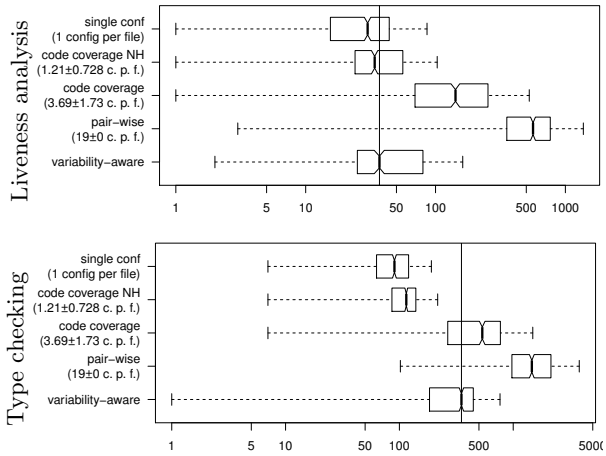

Figure 7: Distribution of analysis times for OpenSSL (times in milliseconds; logarithmic scale).

5.6 Discussion

Our experiments confirm hypotheses H_1 and H_2: in all three subject systems, variability-aware analysis is faster than sampling-based analysis using the pair-wise heuristics, but slower than using the single-conf heuristics. With respect to research question RQ_1, there is no clear picture. The performance of code-coverage sampling depends on the variability implementations in the respective files; the number of sampled variants and performance results differ considerably between files inside each subject system (see Figure 5, 6, and 7). So, performance of the code-coverage heuristic is hard to predict and depends strongly on implementation patterns.

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, whereas type checking considers entire compilation units. Exploring the performance of variability-aware inter-procedural analyses of large scale systems is an interesting avenue of further work.

The experimental results for Busybox, Linux, and OpenSSL demonstrate that variability-aware analysis is in the range of the execution times of sampling with multiple samples (code coverage and pair-wise). So, with regard to question RQ_2, we conclude that variability-aware analysis is practical for large-scale systems. An important finding is that the overhead induced by solving SAT problems during analysis is not a bottleneck, not even for large systems such as the Linux kernel. Overall, variability-aware type checking (compared to single conf) in Busybox takes as much time as checking 6 variants (15 variants in Linux and 4 variants in OpenSSL). For liveness analysis, the *break-even point* is after 5 (Busybox), 6 (Linux), and 2 (OpenSSL) variants. That is, if a sampling heuristic produces a sampling set larger than that (or if continuing random sampling for more than this number of samples), variability-aware analysis is faster and also complete. All values are very low compared with the number of the possible variants of the respective system, showing that a complete analysis is already possible at the costs of an incomplete sampling heuristic.

Table 4: Speedup of variability-aware analysis; a speedup < 1.0 means that sampling is faster and a speedup > 1.0 means that variability-aware analysis is faster (non-significant result in parentheses).

	Variability-aware vs.	Type checking	Liveness analysis
Busybox	Single conf	0.18	0.20
	Code coverage NH	0.48	(0.69)
	Code coverage	9.10	10.12
	Pair-wise	5.01	5.59
Linux	Single conf	0.07	0.18
	Code coverage NH	0.45	1.61
	Pair-wise	7.74	27.24
OpenSSL	Single conf	0.28	0.54
	Code coverage NH	0.38	0.84
	Code coverage	1.70	3.24
	Pair-wise	4.88	10.17

Threats to validity. A threat to internal validity is that our implementations of variability-aware type checking and liveness analysis support ISO/IEC C, but not all GNU C extensions used in the subject systems (especially Linux). Our analyses simply ignore corresponding code constructs. Also due to the textual and verbose nature of the C standard, the implementation does not align entirely with the behavior of the GNU C compiler. Due to these technical problems, we excluded 4 files of Busybox, and 470 files of Linux from our study. All numbers presented in this paper have been obtained after excluding the problematic files. Still, the comparatively large numbers of 518 files for Busybox, 7 221 files for Linux, and 733 files for OpenSSL deem the approach practical and our evaluation representative.

Second, the variants generated by the sampling heuristics represent only a small subset of possible variants (which is the idea of sampling). But, for pair-wise sampling, it may happen that some variants of a file are very similar, as the difference in the respective variant configurations affect the content of a file only to a minor or no extent. However, we argue that our conclusions are still valid, as this lies in the nature of the sampling heuristics, and all heuristics we have used are common in practice.

Finally, a (standard) threat to external validity is that we considered only three subject 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 these systems.

6. RELATED WORK

Our implementations of variability-aware type checking and liveness analysis are inspired by earlier work in two fields.

First, we and others have developed variability-aware type systems for academic languages such as Featherweight Java [2, 24], Lightweight Java [16], the lambda calculus [11], and other dialects of Java [24, 42]. First conceptual sketches even reach back 10 years [5]. Although a prior version of our C type checker has been used to study a variability-aware module systems and has been applied to Busybox, it has not been evaluated in an empirical assessment and comparison to sample-based type checking [26].

Second, researchers proposed variability-aware approaches for data-flow analysis. Closest to our work, Brabrand et al. compared three different algorithms for variability-aware, intra-procedural data-flow analysis for Java against a brute-force approach [9]. Similarly, Bodden proposed an approach to extend an existing inter-procedural, data-flow analysis framework to make it variability-aware [8]. Both approaches are limited to an academic environment in which the input Java programs contain `#ifdef`-like variability annotations managed by a research tool; there are no substantial variable real-world systems that use this technique. Furthermore, both variability-aware analysis approaches make frequently limiting assumptions on the form of variability (in particular, type uniformity [24] and annotation discipline [23, 32]), which do not hold in real-world software systems [25, 32].

7. CONCLUSION

In this paper, we reported on our experience with the implementation and performance of practical, scalable, variability-aware and sampling-based analyses for real-world, large-scale systems written in C, including preprocessor directives. In a series of experiments on three real-world, large-scale subject systems, including the Linux kernel, we compared the performance of variability-aware type checking and liveness analysis with the performance of corresponding state-of-the-art sampling heuristics (single conf, pair-wise, and code coverage).

In our experiments, we found that the performance of variability-aware analysis scales to large software systems, such as the Linux kernel, and even outperforms some of the sampling heuristics, while still being complete. In contrast to previous work on sampling, we faced many problems and found several limiting factors that render state-of-the-art sampling heuristics, such as pair-wise, infeasible.

In future work, we aim at the development of further analyses, at experimenting with other sampling heuristics and with more case studies, and at setting up an automated and incremental checking system for producing bug reports.

8. ACKNOWLEDGMENTS

We thank Tillmann Rendel for fruitful discussions on patterns in variability-aware analysis, and Klaus Ostermann for pushing us to generalize the underlying concepts of variability-aware analysis. This work has been supported in part by the German Research Foundation (AP 206/2, AP 206/4, AP 206/5, and LE 912/13) and ERC grant #203099.

9. REFERENCES

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

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

[3] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, pages 372–375. IEEE, 2011.

[4] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 482–491. IEEE, 2013.

[5] L. Aversano, L. Di Penta, and I. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proc. Working Conf. Source Code Management and Manipulation (SCAM)*, pages 83–92. IEEE, 2002.

[6] T. Berger, S. She, K. Czarnecki, and A. Wasowski. Feature-to-Code Mapping in Two Large Product Lines. In *Proc. Int. Software Product Line Conference (SPLC)*, pages 498–499. Springer, 2010.

[7] 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. Int. Conf. Automated Software Engineering (ASE)*, pages 73–82. ACM, 2010.

[8] E. Bodden, M. Mezini, C. Brabrand, T. Tolêdo, M. Ribeiro, and P. Borba. SPL$^{\text{LIFT}}$ — Statically Analyzing Software Product Lines in Minutes Instead of Years. In *Proc. Int. Conf. Programming Language Design and Implementation (PLDI)*, pages 355–364. ACM, 2013.

[9] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *Proc. Int. Conf. Aspect-Oriented Software Development (AOSD)*, pages 13–24. ACM, 2012.

[10] 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.

[11] S. Chen, M. Erwig, and E. Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *Proc. Int. Conf. Functional Programming (ICFP)*, pages 29–40. ACM, 2012.

[12] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided Abstraction Refinement for Symbolic Model Checking. *Journal of the ACM*, 50(5):752–794, 2003.

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

[14] 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. Conf. Software Engineering (ICSE)*, pages 335–344. ACM, 2010.

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

[16] B. Delaware, W. Cook, and D. Batory. Fitting the Pieces Together: A Machine-Checked Model of Safe Composition. In *Proc. Int. Symp. Foundations of Software Engineering (FSE)*, pages 243–252. ACM, 2009.

[17] C. Dietrich, R. Tartler, W. Schröder-Preikshat, and D. Lohmann. Understanding Linux Feature Distribution. In *Proc. AOSD Workshop Modularity in Systems Software (MISS)*, pages 15–20. ACM, 2012.

[18] 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.

[19] B. Garvin and M. Cohen. Feature Interaction Faults Revisited: An Exploratory Study. In *Proc. Int. Symp. Software Reliability Engineering (ISSRE)*, pages 90–99. IEEE, 2011.

[20] Y. Hu, E. Merlo, M. Dagenais, and B. Lagüe. C/C++ Conditional Compilation Analysis using Symbolic Execution. In *Proc. Int. Conf. Software Maintenance (ICSM)*, pages 196–206. IEEE, 2000.

[21] M. Johansen, Ø. Haugen, and F. Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proc. Int. Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 638–652. Springer, 2011.

[22] M. Johansen, O. Haugen, and F. Fleurey. An Algorithm for Generating t-Wise Covering Arrays from Large Feature Models. In *Proc. Int. Software Product Line Conference (SPLC)*, pages 46–55. ACM, 2012.

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

[24] 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.

[25] 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. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 805–824. ACM, 2011.

[26] C. Kästner, K. Ostermann, and S. Erdweg. A Variability-Aware Module System. In *Proc. Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 773–792. ACM, 2012.

[27] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *Proc. Int. Workshop Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2012.

[28] D. Kuhn, D. Wallace, and A. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. Software Engineering*, 30(6):418–421, 2004.

[29] M. Latendresse. Fast Symbolic Evaluation of C/C++ Preprocessing using Conditional Values. In *Proc. European Conf. Software Maintenance and Reengineering (CSMR)*, pages 170–179. IEEE, 2003.

[30] K. Lauenroth, S. Toehning, and K. Pohl. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. Int. Conf. Automated Software Engineering (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. Int. Conf. Software Engineering (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. Int. Conf. Aspect-Oriented Software Development (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*, 20(3-4):567–604, 2012.

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

[35] C. Nie and H. Leung. A Survey of Combinatorial Testing. *ACM Comput. Surv.*, 43(2):1–29, 2011.

[36] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. Int. Software Product Line Conference (SPLC)*, pages 196–2010. Springer, 2010.

[37] 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*, 20(3-4):605–643, 2012.

[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. Int. Conf. Software Engineering (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. Int. Workshop Opens Source Software and Product Lines (OSSPL)*, 2007.

[40] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Feature Consistency in Compile-Time Configurable System Software. In *Proc. EuroSys Conf.*, 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. Int. Conf. Generative Programming and Component Engineering (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. Int. Conf. Generative Programming and Component Engineering (GPCE)*, pages 11–20. ACM, 2012.

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

[46] A. von Rhein, S. Apel, and F. Raimondi. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *Java Pathfinder Workshop*, 2011. co-located with ASE'11.

[47] H. Zhu, P. Hall, and J. May. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.