Presence-Condition Simplification in Highly Configurable Systems

Alexander von Rhein*, Alexander Grebhahn*, Sven Apel*, Norbert Siegmund*, Dirk Beyer*, and Thorsten Berger**
*University of Passau, Germany **University of Waterloo, Canada

***Abstract*—For the analysis of highly configurable systems, analysis approaches need to take the inherent variability of these systems into account. The notion of presence conditions is central to such approaches. A *presence condition* specifies a subset of system configurations in which a certain artifact or a concern of interest is present (e.g., a defect associated with this subset). In this paper, we introduce and analyze the problem of *presence-condition simplification*. A key observation is that presence conditions often contain redundant information, which can be safely removed in the interest of simplicity and efficiency. We present a formalization of the problem, discuss application scenarios, compare different algorithms for solving the problem, and empirically evaluate the algorithms by means of a set of substantial case studies.**

I. Introduction

Highly configurable systems have become more and more complex in recent years [8, 37]. A *configurable system* provides configuration options (also known as *features*) to tailor the system according to a given set of requirements. A *configuration option* typically represents a choice to include a certain functionality in a system variant. As often not all combinations of configuration options are allowed or meaningful, additional constraints are defined between them, typically using a *variability model*, such as a feature model [26] or decision model [41]. The constraints of the variability model are enforced globally, that is, they must hold for all configurations.

A *presence condition* is an expression over a set of configuration options. The condition represents a subset of configurations in which a certain implementation artifact, such as a code fragment, is included in the corresponding system variants or in which a certain behavior can be observed. For many application scenarios, it is important that presence conditions are as simple as possible. For example, if the presence condition identifies defective system variants, a developer needs to understand the condition (i.e., which options need to be enabled or disabled) to fix the defect [4]. In this paper, we focus on Boolean configuration options and Boolean presence-conditions, which is sufficient to represent most presence conditions in practice [7, 8].

With presence conditions as a key concept, variability-aware analyses emerged for type checking [3, 20, 27], static analysis [32], model checking [12, 30], and variability-model analysis [5]. A *variability-aware analysis* analyzes a configurable system as a whole, incorporating its variability model. Typically, the analysis considers only valid combinations of options, and it presents its findings (e.g., a detected defect) annotated with presence conditions. For example, a reported presence condition could identify all system variants containing a certain type error. Even in small systems with few options, these conditions can be very complex and difficult to understand.

Consider the example of the tool SPLVERIFIER for verification of configurable systems [4], which reports the following defect in the E-MAIL system—a benchmark for interactions between options in configurable programs [23]:

```
1 Specification 11 violated on condition
2   encrypt && decrypt && keys &&
3   ((sign && verify && base && autoresponder) ||
4   (!sign && !verify && base && autoresponder))
```

The defect in this example is caused by an interaction of only two options, *encrypt* and *autoresponder*. All other parts of this presence condition have been introduced by the global variability model (e.g., *encrypt* requires *keys*). A desirable simplification of the presence condition is to identify *encrypt* and *autoresponder* as the sole cause of the defect and to "hide" the parts introduced by the variability model. Simplifying the condition and identifying the responsible options help in fixing the defect [4].

A straightforward way to simplify a presence condition (or make it more readable) is to find the smallest, but equivalent expression. This problem is known as the *minimum-equivalent-expression* problem [11, 24]. Although finding a minimal equivalent expression might reduce the size of the presence condition, the minimal equivalent expression is still larger than necessary. One reason for unnecessarily large expressions is that variability-aware analyses typically consider only configurations satisfying the variability model. Thus, the variability model is an integral part of every reported presence condition, even though the condition describes only a local situation or fact. Since the variability model must be satisfied globally, this information obfuscates the presence condition.

Our goal is to simplify a given presence condition such that it becomes smaller and can be used instead of the original presence condition. In the example above, we want to remove the constraints already enforced by the variability model from the presence condition and show only the rest to the user. This rest must be satisfied *in addition to* the variability model to reach the situation of interest (e.g., it identifies the source of the defect).

To this end, we introduce the *presence-condition-simplification problem* and present a formal definition of the problem. We are interested in a function $\text{simp}(p,m)$ such that the expression $p' = \text{simp}(p,m)$ is equivalent to the presence condition p under all assignments that satisfy the context m: $m \Rightarrow (p' \Leftrightarrow p)$. In addition to this invariant, the size of p' should be

as small as possible (we define a size measure in Section III-C).

There are many other application scenarios of presence-condition simplification, including (1) the simplification of preprocessor directives in system software (#ifdef conditions are simplified based on the conditions of surrounding #ifdef conditions in the nesting hierarchy) and (2) the simplification of cross-tree constraints in variability models (information that is already encoded in the feature/option hierarchy is removed from the cross-tree constraints).

As a solution for presence-condition simplification, we identify and adopt three algorithms from the circuit-optimization domain. To the best of our knowledge, the algorithms have not yet been applied to the simplification of presence conditions. The first algorithm, RESTRICT [14], is based on binary decision diagrams (BDDs) [10]. The second and third algorithms are solutions for two-level logic minimization: the QUINE-MCCLUSKEY algorithm [22, 40] and the ESPRESSO algorithm [9].

To compare the algorithms and to explore their feasibility and effectiveness for presence-condition simplification, we conduct a series of experiments on three application scenarios and 31 subject systems. We evaluate processing time and size reduction of presence conditions for the three algorithms. Our results show that presence-condition simplification can achieve substantial improvements in reasonable time for various realistic application scenarios. For example, in an experiment where we simplified analysis results (E1), one simplification algorithm (simp_{BDD}) reduced the size of presence conditions by 61%, on average. Furthermore, we analyze how the simplification algorithms scale with the increasing complexity of the input expressions.

In summary, we contribute:

- A formalization of the *presence-condition simplification problem* in the context of highly configurable systems, and a discussion of scenarios to which presence-condition simplification can be applied.
- A discussion of three algorithms solving the problem: RESTRICT, QUINE-MCCLUSKEY, and ESPRESSO.
- An evaluation of the algorithms on three application scenarios and 31 subject systems showing that simplification potential exists and the algorithms scale in realistic scenarios.

One of the presented algorithms (simp_{BDD}) has been integrated in the variability-aware analysis tool TYPECHEF, resulting from our research. We provide a replication package for our experiments and further detailed results on an accompanying website: www.fosd.de/pcsimp/.

II. BACKGROUND

To establish the terminology that we use throughout the paper, we give an overview of configurable systems and (Boolean) presence conditions. A *configurable system* provides a set of *configuration options* to be set by the user to derive a desired *system variant*. In what follows, we use the E-MAIL system of Hall [23] as a running example. The E-MAIL system simulates a network of e-mail hosts between which e-mails are sent.

```
1  struct email {
2    int id; char *from; char *to; char *subject; char *body;
3  #if (defined(encrypt) || defined(decrypt))
4    int isEncrypted; char *encryptionKey;
5  #endif
6  };
7  // incoming e-mails enter here
8  void incoming (struct client *client, struct email *msg) {
9  #if (defined(decrypt))
10   decrypt (client, msg); // decrypt encrypted incoming e-mails
11 #endif
12 #if (defined(forward))
13   forward (client, msg); // forward incoming e-mails automatically
14 #endif
15   ... // store e−mail in a mailbox
16 } ...
17 #if (defined(decrypt))
18 // decrypt a given e-mail if the key of the sender is known
19 void decrypt (struct client *client, struct email *msg) { ... }
20 #endif
```

Fig. 1: Excerpt of the configurable E-MAIL system

Figure 1 shows an excerpt of the system's code with variability expressed using #if preprocessor directives. In addition to the basic functionality (*base*), the E-MAIL system offers eight configuration options: *keys* enables support for private and public keys, *encrypt* and *decrypt* implement encryption and decryption, *sign* and *verify* implement signing of e-mails and verifying of signatures, *forward* implements automatic e-mail forwarding, *autoresponder* generates automatic response e-mails, and *addressbook* provides contact management. A *configuration* of a system is a total mapping from the configuration options to values. Here, we focus on Boolean configuration options (whose values can only be *true* or *false*), because they are an important backbone of configurations in practice [7]; non-Boolean configuration options can be represented by multiple Boolean variables, if the domain is discrete and small, such as for enumerations. We represent a mapping from options to choice values, such as $base \mapsto true, keys \mapsto true, forward \mapsto true, encrypt \mapsto false, \ldots$, with a Boolean expression such as $base \wedge keys \wedge forward \wedge \neg encrypt \wedge \ldots$ Typically, not all possible mappings are valid in a configurable system; a *variability model* defines the set of *valid* configurations. A possible representation of a variability model is again a Boolean expression, for example: $base \wedge (decrypt \Leftrightarrow encrypt) \wedge (sign \Leftrightarrow verify) \wedge (encrypt \Rightarrow keys) \wedge (sign \Rightarrow keys)$. All assignments of configuration options for which the variability model is satisfied correspond to the set of valid configurations.

A *presence condition* is a Boolean expression over the set of configuration options [16]. We use the term presence condition in a broad sense, but it always is a Boolean expression denoting the condition for presence of certain code or behavior in variants of the configurable system. For example, a presence condition can denote the condition for a defect in the system or for the presence of a statically conditional piece of code in a variant.

Finally, given two Boolean expressions f and g, and the fact that f implies g, we refer to f as an *implicant* (also known as *premise*) of g, and to g as an *implicate* (also known as *conclusion*) of f. A *prime implicant* f of g is an implicant of g

that is minimal—that is, the removal of any literal from f results in a non-implicant for g. For example, the expression $g = (x \wedge y) \vee w$ has the implicants $x \wedge y$, $x \wedge w$, and more. The term $x \wedge y$ is a prime implicant—the removal of x or y leaves a non-implicant.

III. Presence-Condition Simplification

We illustrate the problem of presence-condition simplification by means of three practical application scenarios (Section III-B), define invariants for a solution of the problem formally (Section III-C), and discuss algorithms satisfying these invariants (Section III-D).

A. Problem Overview

Presence-condition simplification aims at simplifying a presence condition p with respect to its context m. As an approximate measure for "simpleness" of a Boolean expression, we use the number of its literals, but other measures are possible as well (see Section III-C, for further details). p and m are given as Boolean expressions, and we require that p is embedded in m, which means that p is evaluated only if m is satisfied. A simplification function 'simp' receives two Boolean expressions p and m and returns a Boolean expression.

B. Application Scenarios

Although the application domain is much broader, we are particularly interested in presence-condition simplification in the context of developing and analyzing highly configurable systems. Next, we illustrate three interesting scenarios.

Reporting Analysis Results: Variability-aware analyses often report the condition under which certain events or states occur as presence conditions. Recall our example of SPLVERIFIER [4], which implements variability-aware model checking of configurable systems. The tool prints the following output after verifying Hall's E-MAIL system [4, 23]:[1]

```
1 Specification 11 violated on condition
2   encrypt && decrypt && keys &&
3   ((sign && verify && base && autoresponder) ||
4   (!sign && !verify && base && autoresponder))
```

The E-MAIL system can be configured with various options, and some combinations of options can lead to violations of certain specifications, as reported above by SPLVERIFIER. Such violations indicate either an incomplete variability model, which should be fixed to prevent defective configurations, or a bug in the system. Since SPLVERIFIER verifies only configurations that satisfy the variability model, which is standard in variability-aware analyses [47], the reported presence conditions contain parts that are already implied by the variability model. This mix of error condition with variability-model constraints hinders understanding and pinning down the source of an error. Even though the E-MAIL system has only nine configuration options, the reported defect conditions are often unnecessarily complicated. The defect conditions we encountered for the E-MAIL system have between 5 and 17 literals. By applying presence-condition simplification to the defect presence condition shown above, we yield the following result containing only 2 instead of 11 literals:

```
1 Specification 11 violated on condition
2   VariabilityModel && (encrypt && autoresponder)
```

[1]The specification states that e-mails must be decrypted before sending an automatic response.

```
32 ...
33 obj-$(LOCKDEP) \
34  += lockdep.o
35 ...
```

(a) Excerpt from kernel/Makefile

```
826  #if defined(PROVE_LOCKING)
827  ...
1301 #if defined(TRACE_IRQFLAGS)
1302  && defined(PROVE_LOCKING)
1303 ...
1674 #else
1675 ...
1688 #endif
1689 ...
2146 #endif
```

(b) Excerpt from kernel/lockdep.c

Fig. 2: Nested variability annotations with redundancy

Simplification of Variability Annotations: Variability annotations are directives in a system's source code that conditionally include or exclude parts of the code based on the choice of configuration options. For illustration, we focus on two implementation mechanisms for variability annotations: conditional inclusion of files in build scripts and #if preprocessor directives. Figure 2 shows an example of both mechanisms used together, taken from the LINUX kernel v3.4. Figure 2a shows an excerpt of the Makefile in the kernel directory. It states that the object file of lockdep.c is included if option LOCKDEP is enabled. Figure 2b shows an excerpt of file lockdep.c, which contains several #if directives.

Observe that, in the example, the innermost #if directive (Figure 2b, Line 1301) is enclosed by two conditions: the #if condition in Line 826 and the condition from the Makefile. The conjunction of both enclosing conditions is the context of the condition in Line 1301: $m = (\text{LOCKDEP} \wedge \text{PROVE\_LOCKING})$ and $p = (\text{TRACE\_IRQFLAGS} \wedge \text{PROVE\_LOCKING})$. Using presence-condition simplification, we can remove the redundant term PROVE_LOCKING from the condition of the inner preprocessor directive without changing the behavior of any variant of the LINUX kernel: $\text{simp}(p, m) = \text{TRACE\_IRQFLAGS}$.

Admittedly, the expressions involved in this example are relatively simple, so a developer might be aware of the redundancy and leave it for documentation. Still, in more complex cases, simplification can be more effectful, especially because it is also beneficial for tools working on the code to ease automatic reasoning.

Automatic code analysis of systems with #ifdef variability is difficult because the preprocessor directives can be interleaved with normal C code in complicated ways. The tool TYPECHEF [28] solves this problem by providing a variability-aware parser for C code with #if directives. It is used by many research projects [29, 32], which would benefit from presence condition simplification. TYPECHEF resolves preprocessor directives and macros, and it generates an abstract syntax tree (AST), preserving the variability induced by #if directives. Technically, nodes in the AST are annotated with the presence conditions that correspond to the #if directives. Due to difficulties in the

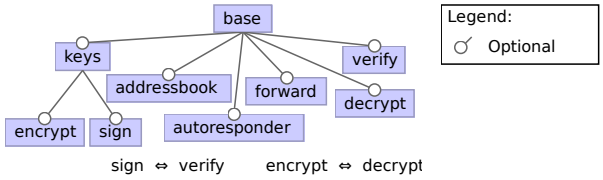


Fig. 3: Feature model of the E-MAIL system

parsing process (e.g., macro expansion) [28], these presence conditions are often an overapproximation of the actual presence conditions and contain redundancy. We can make the AST generated by TYPECHEF more concise by simplifying the presence conditions with their context (presence conditions of ancestors in the AST conjoined with the presence condition of the file), which improves the performance of subsequent analyses, such as type checking or data-flow analysis [32].

Variability-Model Generation: A variability model can be expressed in different formats. In this paper, we use Boolean expressions, but other scenarios require richer representations, such as feature models by Kang et al. [26]; Figure 3 shows a representation of the feature model of the E-Mail system (Section II). Such a model contains a hierarchy that shows dependencies between the configuration options (child–parent implication). For example, selecting option encrypt implies selecting its parent keys. Constraints that cannot be encoded in the hierarchy are written as separate *cross-tree constraints*. For example, the dependencies from encrypt to decrypt and vice versa are expressed as cross-tree constraints.

If a variability model is given as a Boolean expression (for instance, when extracted from source code [42]), it is sometimes desirable to transform it into a visual model for presentation. There are a number of approaches (e.g., [42]) that synthesize a hierarchy (shown as tree in Figure 3) and constraints between siblings in the hierarchy. All constraints that cannot be encoded in the hierarchy or as sibling constraints are added as cross-tree constraints.

If the cross-tree constraints are still complex, it is advisable to simplify them using the hierarchy and the sibling constraints as context. That is, the cross-tree constraints should not restate the dependencies covered by the hierarchy or the sibling constraints. Given the hierarchy constraints h, the sibling constraints s, and the cross-tree constraints ctc, this can be achieved with $\text{simp}(ctc, h \wedge s)$. The result of simplification can replace the original cross-tree constraints, because the context of hierarchy and sibling constraints always hold.

C. Problem Formalization

Function $\text{simp}(p,m)$ has two inputs: a presence condition p and a context m. The goal is to represent the *relevant* information in p as concise as possible. The input parameters and the result of the simplification are Boolean expressions. Even though the problem and the described algorithms (Section III-D) work on general Boolean expressions, we use them only in the configurable-systems context.

We assume that the context m is available and holds significant information on the situations in which p can be evaluated. If it is not available, or if it represents a tautology, then we have to assume that all information in p is relevant to identify the situation or fact that p represents. In this case, the only possibility to improve the presentation of p is to generate a *minimum equivalent expression* for p [11, 24]. However, in the scenarios that we focus on, usually a substantial, non-tautology context is available (e.g., a global variability model).

Presence conditions are meant to be evaluated only if the context m holds. The information encoded in a presence condition p is essentially the set of implicates of p. Elements of this set can be categorized as follows: An implicate of p is either (1) also an implicate of m or (2) no implicate of m. Implicates in group 1 are redundant and can be dropped. Some implicates in group 2 are implied by elements of group 2 conjoined with m and are therefore also redundant. If we can extract the essential, non-redundant elements of group 2 and present them as replacement for p, this would be sufficient, because the context m guarantees that the implicates in group 1 are satisfied. Hence, we do not search for an equivalence-preserving function, but we aim at removing implicates from p if they are redundant with respect to m and if they increase the size of p.

Figure 4 illustrates the relationship between p, m, and $\text{simp}(p, m)$ in terms of the configuration space of a configurable system. The white rectangle m represents all valid configurations. The rectangle p represents the space of configurations denoted by the presence condition. p encloses only configurations that are in m. Also, p is (often) smaller than m because it specifices a certain local condition within the global space of configurations. The rectangle $\text{simp}(p,m)$ represents the simplified presence condition. This rectangle encloses all configurations of p, but also configurations from out of m, if it helps to remove implicates from the expression (i.e., if it reduces the size of p). The objective is that the area of $\text{simp}(p,m)$ represents a more concise expression than p.

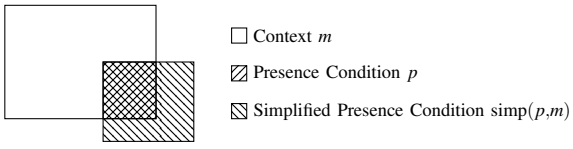


Fig. 4: Illustration of presence-condition simplification. Each point on the plane represents a configuration. The crosshatched area ⊠ denotes the overlapping of the area of p ▨ and $\text{simp}(p,m)$ ▧. The simplified presence condition can include configurations not allowed by the context (e.g., the variability model) if it helps reducing the size of the condition.[2]

Formally, the invariant for correctness of $\text{simp}(p,m)$ is:

$$m \Rightarrow (\text{simp}(p,m) \Leftrightarrow p) \quad (1)$$

This invariant states that, in the context of m, the expressions p and $\text{simp}(p, m)$ are logically equivalent. Therefore, we can use $\text{simp}(p,m)$ as replacement for p, provided that m holds.

Equation 1 is a sufficient condition for the correctness of replacing all occurrences of p in the context m by $\text{simp}(p,m)$.

[2] A condition that includes more configurations can be smaller in size than a condition with more implicates.

In the simplest case, $\text{simp}(p,m) = p$ would be a valid solution. However, our goal is to *simplify* p. So, we define an objective function stating that $\text{simp}(p,m)$ must be minimal according to a given measure size:

$$\forall x : \big(m \Rightarrow (x \Leftrightarrow p)\big) \Rightarrow \big(\text{size}(\text{simp}(p,m)) \leq \text{size}(x)\big) \quad (2)$$

Defining a general measure for the size of Boolean expressions is not reasonable as it depends on the application scenario. In the application scenarios we are interested in (cf. Section III-B), conciseness of expressions is most important, because they are usually presented to the user. In other cases, expressions are used to generate hardware circuits, for which other optimization goals are needed.

In practice, we have to compare formulas given in notations with different constraints (e.g., CNF, DNF, or BDD) because different simplification algorithms (described in Section III-D) have different input and output formats. To avoid bias of different notations, we focus on the complexity of the encoded formula. To this end, we convert all expressions to a canonical normal form before comparison. As canonical form, we choose a reduced if-then-else normal form (derived from BDDs) that contains only $\wedge$, $\vee$, and $\neg$ as operators.

After the expressions are converted to the same notation, there are several possible size measures for comparison. We choose the number of occurrences of literals as size measure because it represents the total expression length and is not influenced, for instance, by lengths of variable names. So, for the remaining sections, we define the measure $\text{size}(y)$ as the number of occurences of literals in the string representation of an expression y in canonical form. For expression $y = (A \wedge B) \vee (\neg A \wedge C)$, $\text{size}(y) = 4$ (B,C, and twice A).

We also evaluated the number of operators and the number of nodes in a BDD representation as alternative size measures, but observed no major deviations in our experiments (cf. Section IV). In their work on the minimum-equivialent-expression problem, Hemaspaandra and Schnoor [24] have also used the number of occurrences of literals and the number of operators. We decided against measures such as the depth of an AST of the formula, because a CNF/DNF representation would always have depth 2, which renders the measure useless for our purposes.

D. Implementation

We introduce four algorithms solving the presence-condition-simplification problem: BRUTE-FORCE (simp_{BF}), RESTRICT (simp_{BDD}), ESPRESSO (simp_{E}), and QUINE-MCCLUSKEY (simp_{QC}). simp_{BF} finds an optimal solution, but it iterates over all possible solutions. The other three algorithms employ heuristics to improve computational complexity while still satisfying the invariant of Equation 1.

Naive Solution: The BRUTE-FORCE (simp_{BF}) algorithm enumerates all implicates of p. Technically, it uses the clauses of the canonical conjunctive normal form (CCNF) of p. Then, it builds the powerset of these clauses. For each element of the powerset, the algorithm tests whether it satisfies Equation 1 and therefore qualifies as a solution. From all possible solutions, the algorithm selects an optimal solution according to the size measure.

The CCNF has 2^n clauses for n configuration options. Therefore, the size of the powerset of the set of clauses is 2^{2^n}, and we have to iterate through the entire set. Due to its computational complexity, we cannot use simp_{BF} in our experiments.[3]

BDD Simplification: The second algorithm was first described by Coudert and Madre [14] in 1989 as the RESTRICT algorithm (simp_{BDD}). The RESTRICT algorithm takes two expressions p and m represented as BDDs and generates a third BDD $c = \text{simp}_{BDD}(p,m)$ that satisfies the invariant of Equation 1 [14]. The algorithm is intended to minimize the number of nodes in the BDD representation of $\text{simp}_{BDD}(p,m)$. This is in line with our optimization goal, but as the algorithm uses heuristics, it does not always generate optimal results. For further details, we refer to the original publication [14] and to our supplementary website.

Like many other BDD operations, simp_{BDD} is a polynomial-time graph-manipulation algorithm (if caching is used). In the worst case, the size of the graph may be exponential in the number of the variables, which renders the algorithm also exponential in the number of variables. However, in practice, the worst case is unlikely, which is part of the reason for the success of BDDs.

Two-Level Logic Minimization: The third solution is to transform the problem of presence-condition simplification into a *two-level-logic-minimization* problem [15], which can be solved with the QUINE-MCCLUSKY and ESPRESSO algorithms (simp_{QC} and simp_{E}). The attribute "two-level" arises from the fact that input expressions are expected in DNF, and a DNF has two levels (the global level with $\vee$ operations and the clause level with $\wedge$ operations). Two-level logic minimization receives a Boolean expression f and a second expression dc, which represents a *don't care set*, called *DC set*. The expressions divide the entire space of option assignments into three partitions: (1) the set of assignments for which $f \wedge \neg dc$ is satisfied, called the *ON set*, (2) the set of assignments for which $\neg f \wedge \neg dc$ is satisfied, called the *OFF set*, and (3) the *DC set* for which dc is satisfied. The result of two-level logic minimization is a simplified version of f.

Mapped to our problem, expression f represents the presence condition p. DC describes variable assignments for which the result $\text{simp}_{E}(p,m)$ need not be equivalent to p. In our case, these are all variable assignments that are not valid in the context ($\neg m$). That is, DC is the piece of information needed for minimization. So, the setup $f \equiv p$ and $dc \equiv \neg m$ satisfies Equation 1.

Two-level logic minimization can be exact or heuristics-based. An exact algorithm determines the minimal set of prime implicants needed to represent f without respecting dc. It can be solved with the QUINE-MCCLUSKY algorithm, which is NP-complete. In a nutshell, the algorithm starts with computing all prime implicants for the union of the ON and DC sets. Finding the smallest set of these prime implicants that still cover f is basically a set-covering problem, which is also NP-complete. The algorithm uses reduction techniques and a branch-and-bound strategy to solve this problem [15].

[3] Our implementation works for up to four configuration options.

For performance, several heuristics have been developed. The most prominent heuristic-based algorithm is the ESPRESSO algorithm [15], which utilizes a local search without generating all prime implicants. It is composed of three main operations: *expand*, *reduce*, and *irredundant*. The operations *expand* and *reduce* are applied to improve the current term during optimization, and the operation *irredundant* is used to get out of a local minimum. In our experiments, we evaluate the ESPRESSO algorithm, denoted with simp_E, and the QUINE-MCCLUSKEY algorithm, denoted with simp_{QC}. For further details on the algorithms, we refer to an overview paper [15].

Input-Format Conversion: Different application scenarios of presence-condition simplification require different input formats for the parameters p and m. So, the input expressions need to be converted in formats suitable for the different algorithms. This is mainly a technical issue that we describe in the Appendix.

IV. EVALUATION

We evaluate the different algorithms for presence-condition simplification guided by two research questions:

RQ1 We expect that presence conditions with a known context are often too complex and can be simplified. In which application scenarios does this hold, and are the resulting expressions substantially smaller?

RQ2 How does the processing time of the algorithms simp_{BDD}, simp_E, and simp_{QC} scale to complex simplification tasks?

We evaluate these research questions on the application scenarios described in Section III-B on, overall, 31 example configurable systems. As a measure of simplification (RQ1), we compare the number of occurences of literals in the expression before and after simplification. To ensure a fair comparison, we transform the results generated by the simp_E and simp_{QC} algorithms to BDDs after the algorithms have terminated. This step ensures that the compared result strings are compact and have the same variable order. We do not include the time needed for this transformation. The processing time (RQ2) is measured per simplification task. To ensure fairness we did not call simp_{BDD} on in-memory BDDs, but wrote them to a file, invoked simp_{BDD} in a new process, and measured the time for that process to terminate. This time includes parsing the presence condition and context, and writing the result expression.

In total, we designed five experiments labelled E1 through E5. To evaluate research question RQ1, we needed sets of Boolean presence conditions and contexts from different application scenarios and configurable systems. We obtained these sets from different research projects and did one experiment per project. E1 and E2 represent variations of the "Reporting Analysis Results" application scenario, E3 and E4 apply the "Simplification of Variability Annotations" scenario to source code and to the internal code representation in TYPECHEF, respectively.

To evaluate research question RQ2, we needed a setting where we can flexibly control the size of the problem. We chose to evaluate this question with the "Variability-Model Generation" scenario and used a variability-model generator [34] to create simplification tasks. In the generator, we can increase the number of generated variables and therefore generate harder problems. We used these generated tasks in E5 to evaluate the processing-time performance of the algorithms.

TABLE I: SUBJECT SYSTEMS AND APPLICATION SCENARIOS (MORE DETAILS ARE AVAILABLE ON THE SUPPLEMENTARY WEBSITE)

APACHE (E1)	LINUX v2.6.33.3 (E3, E4)	SPLOT models (E5)
E-MAIL (E1, E2)	LINUX v3.4 (E3)	SQLITE (E1,E3)
ELEVATOR (E2)	LLVM (E1)	ZIPME (E1)
H264 (E1)	PKJAB (E1)	18 other
LINKED LIST (E1)	SNW (E1)	systems (E3)

A. Subject Systems and Experiments

We use a diverse set of subject systems from various sources to evaluate the different applications of presence-condition simplification. For each system, we ensured that a variability model was available. Table I gives an overview of the systems and in which experiments they are used. The experiments E1–E5 are described next.

Classification of Variants (E1): The first application scenario is based on an approach that estimates non-functional properties (footprint, response time, etc.) of the variants of a configurable system [44]. Experiments evaluating the approach typically generated huge datasets. For E1, we use the following systems from previous studies [43, 44, 45]: APACHE, E-MAIL, H264, and LLVM (prediction of response time per variant), and LINKED LIST, PKJAB, SNW, SQLITE, and ZIPME (prediction of binary footprint per variant). Presence conditions in this scenario indentify system configurations for which the prediction accuracy is low, possibly due to unknown interactions among configuration options. Presence-condition simplification is useful for pinpointing these to a smaller number of options such that further investigation is possible. We have presence conditions for seven different levels of prediction accuracy and simplify all of them seperately using the corresponding variability model as context.

Reporting Defect Locations (E2): For our second experiment, we use data from a study evaluating the performance of variability-aware model checking [4]. During experiments, the authors found many defects in the subject systems that occur only under certain presence conditions. We use the E-MAIL and ELEVATOR systems, which are standard benchmarks for interaction detection [4, 13, 23]. Presence conditions of defects and the variability model are given as textual Boolean expressions. An example for the defect location scenario is the output of the SPLVERIFIER tool given in Section III-B. We simplify the defect presence conditions and evaluate the performance of the simplification algorithms.

Code-Annotation Simplification (E3): To evaluate the simplification potential for code annotations, we use several configurable software systems with #if directives and apply simplification to the #if conditions (see Section III-B). For our experiments (see Section IV-C), we use 21 configurable systems, including the LINUX kernel.

The context of #if conditions in these projects has two components: the conditions of enclosing #if directives and the condition under which the respective file will be included in

the project, as described in Section III-B. In projects that use KCONFIG, we used the tool KBUILDMINER [6, 7] to extract the conditions under which source files are used. For the others, we assumed that each file is used in all configurations. We extracted #if conditions in source files with the PREDATOR tool [46]. PREDATOR also provides the hierarchy of #if conditions, such that we can generate for each #if condition a context consisting of the conjunction of the enclosing #if conditions and the file condition. Given these pairs of #if conditions and contexts, we apply the simp_{BDD}, simp_{E} and simp_{QC} algorithms and measure how often the conditions could be improved to evaluate the potential for presence-condition simplification. We skip pairs for which #if conditions or contexts are tautologies or contradictions because then simplification is impossible.

AST-Annotation Simplification (E4): In this experiment, we analyze the variability-aware ASTs generated by TYPECHEF [28]. Each generated AST node has a presence condition. Due to difficulties in parsing C code with #if directives (e.g., undisciplined annotations and macro expansion), the resulting presence conditions are often larger than the conditions written in the source code [28]. For simplification, we generated a context for each presence condition p by building the conjunction m of all presence conditions on the path from p to the root node of the AST. Then, we applied simplification of p in the context m and evaluated the reduction in the size of the presence conditions. Again, we do not simplify if p or m is a tautology or a contradiction. Even though we optimize only an internal representation here, it can affect processing time. Furthermore, presence conditions are visible to users (1) as part of reports, (2) as debugging info, and (3) if the AST is printed again after some modification (e.g., automatic code refactoring).

Cross-Tree-Constraint Simplification (E5): To evaluate the scalability potential of presence-condition simplification, we used the variability-model generator from the SPLOT repository [34] for generating test variability models. Each model comprises hierarchy, grouping, and cross-tree constraints given in CNF. This is the same setup as in the final step of the variability-model generation scenario (Section III-B).

As scaling factor, we used the number of configuration options of the generated models. We have generated sets of 10 variability models with 20/30/40/50/60 configuration options (50 models in total). For each model, we simplify the cross-tree constraints using the hierarchy and sibling constraints as context. All constraints are given in CNF, so we apply the FORCE algorithm [2] to optimize the BDD variable ordering. The more compact representation is beneficial for simp_{BDD}, but also for simp_{E} and simp_{QC}.

B. Experiment Setup

For our experiments, we use existing algorithm implementations: simp_{BDD} is available as function net.sf.javabdd.BDD.simplify(BDD) in the JAVABDD library, simp_{E} is available in the ESPRESSO tool, and simp_{QC} is also implemented in ESPRESSO as a revised version of the original QUINE-MCCLUSKEY algorithm. We provide links to the tools on our supplementary website. We also tried to use SCHERZO, a newer tool for two-level logic minimization, however, we were not able to apply it to presence-condition simplification because of technical problems and missing documentation.

All experiments have been executed on an Intel Xeon machine (8 cores with 2.93 GHz) with Ubuntu 12.04. Regarding parallelization, we have not observed that more than one core was used in the experiments. In all experiments, simplification has been executed in a JVM with 4 GB of RAM. We set the timeout for the simplification algorithms in all experiments to 60 seconds (the usual response time was much less). In the experiments E1–E4, we encountered only 12 timeouts, (7 with simp_{E} and 5 with simp_{QC}). All timeouts occured while simplifying presence conditions of the SQLITE system (E1). For code simplification (E3), we used scripts to call the external analysis tools (e.g., TYPECHEF). The tool output was aggregated and later simplified.

During the experiments, we measured the processing time of the algorithms and the number of literals in the expressions before and after simplification. We use the number of occurences of literals as size measure (Section III-C). We also evaluated other measures (number of operands and node count in the BDD), but they do not change the overall picture. For each simplification, we compare the size of the original presence condition p and the simplified presence condition $\text{simp}(p, m)$. For these comparisons, we define the *reduction factor* as $\text{size}\big(\text{simp}(p,m)\big)/\text{size}(p)$.

In some cases, the size of the supposedly simplified expression was larger than the size of the original expression. This can happen because some of the algorithms rely on heuristics. Such cases are easy to detect and we just use the original expression instead of the generated expression. In such cases, we logged that simplification did not improve the expression size (the reduction factor is 1). To provide a ground truth, we would need to iterate over all solutions (i.e., apply the BRUTE-FORCE algorithm). However, due to the complexity of the problem, BRUTE-FORCE does not scale for any of our experiments.

C. Results

Classification of Variants (E1): Figure 5 shows the reduction factors we observed for the classification of variants per subject system. A lower reduction factor indicates a better simplification result. Each boxplot covers all experiments per algorithm and subject system. Figure 5 shows that (1) the number of literals is generally much lower after simplification and that (2) simp_{BDD} generates slightly better results, on average, than simp_{E} and simp_{QC}, as confirmed by paired Mann-Whitney tests (p-values below 0.001 for both tests). These results confirm RQ1: For this application scenario and the considered systems, there is significant simplification potential, and the algorithms are able to simplify the presence conditions substantially.

Figure 6 shows the time needed for simplification in a quantile plot. For example, the point $(150, 110)$ in graph simp_{E} in the plot states that the 150-th fastest simplification with simp_{E} took 110 ms and there were 149 simplification tasks that took 110 ms or less with simp_{E}. The plot shows how the algorithms scale when tasks are harder to solve using the

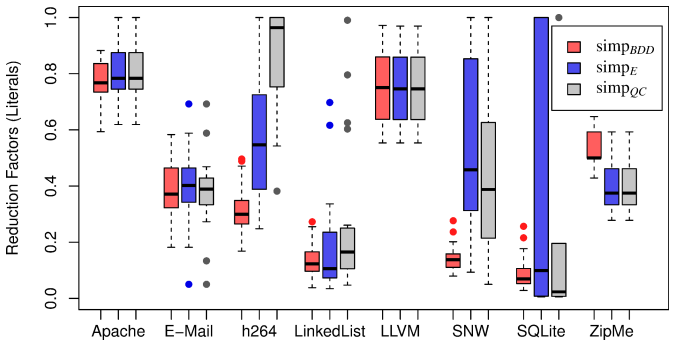


Fig. 5: Reduction factors for the classification of variants (E1)

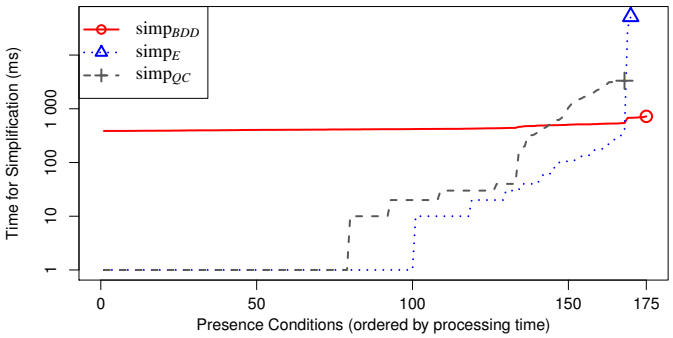


Fig. 6: Time for simplification in a quantile plot (E1); a point (x,y) in the plot states that the x-th fastest simplification with the respective algorithm took y milliseconds; the right-most x value indicates the number of solved tasks; the y axis has a logarithmic scale

TABLE II: REDUCTION FACTORS FOR DEFECT LOCATION REPORTING (E2)

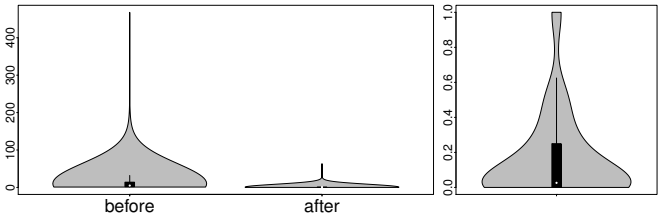
	simp$_{BDD}$	simp$_{E}$	simp$_{QC}$
ELEVATOR	0.39	0.37	0.37
E-MAIL	0.22	0.14	0.14

same simplification tasks as in Figure 5; the time for simp$_{QC}$ and simp$_{E}$ is negligible for easy tasks but increases with harder tasks; simp$_{BDD}$ needs between 400 ms and 500 ms in most cases, however, it can solve more problems than simp$_{QC}$ and simp$_{E}$. Note that in our setup simp$_{BDD}$ requires startup time for the JVM, which dominates the processing time.

Reporting Defect Locations (E2): In E2, we consider verification of E-MAIL and ELEVATOR [4] as application scenario. Note that E-MAIL and ELEVATOR are the same systems as in E1, but the considered presence conditions represent very different facts: In E1, the presence conditions represent the prediction accuracy of the non-functional property prediction approach [43, 45]. In E2, the considered presence conditions point to configurations in which specifications of the configurable systems are violated, as identified by SPLVERIFIER [4].

Table II shows the average reduction factor per case study and algorithm. All three algorithms achieve significant improvements of the simplified expressions in terms of the reduction factors, providing further evidence for RQ1. Overall, the reduction factors are very similar for all algorithms. The maximum time measured for simplification was 472 ms, which is negligible.

Code-Annotation Simplification (E3): In E3, we evaluate the potential for simplification of #if conditions in source code.



(a) Presence-condition size (b) Reduction factor

Fig. 7: Experiment results for the TYPECHEF AST simplification on LINUX (E4)

Overall, we found only few situations where our approach could improve the presence conditions. For detailed results of the experiment we refer to the supplementary website. With all three algorithms and in all systems except gnuplot (3.3%) and libxml2 (3.3%), we could improve only less than 2% of the parsable, non-trivial presence conditions. If we could not parse the conditions, this was usually due to non-Boolean configuration options. Most situations for which we could improve the conditions are rather simple, similar to the example shown in Figure 2.

The experiment E3 shows that our approach is feasible in the application scenario, but there is only little potential for simplification in the considered systems. #if conditions in the analyzed systems do not contain much redundancy, which indicates a good code quality. So the expectation stated in RQ1 does not hold in this application scenario, for these systems.

AST-Annotation Simplification (E4): To evaluate the simplification potential in ASTs as generated by TYPECHEF, we modified TYPECHEF such that it applies simplification to all presence conditions generated as AST annotations. In particular, we analyzed the AST conditions generated for LINUX 2.6.33.3 (the actual subject of E4 is TYPECHEF, not LINUX, so one version is sufficient). Figure 7 shows the results by means of violin plots. A violin plot contains a boxplot and shows also the probability density of the data.

In total, we found 6 115 774 non-trivial presence conditions in LINUX's AST. Figure 7a shows that most of these have less than 100 literals. However, there is a substantial number of presence conditions that have an extremely large number of literals. After simplification (shown data generated with simp$_{BDD}$), the conditions have less than 50 literals. Figure 7b shows the reduction factors observed with simp$_{BDD}$ (the results are similar for simp$_{E}$ and simp$_{QC}$). For most presence conditions, we achieved extreme improvements, which leads us to two conclusions confirming RQ1: (1) TYPECHEF introduces many redundancies during parsing, because we did not observe similar sizes for LINUX in E3, and (2) simplification can remove those redundancies from the AST.

Cross-Tree-Constraint Simplification (E5): To evaluate the scalability of simp$_{BDD}$, simp$_{E}$, and simp$_{QC}$ (RQ2), we ran experiments with synthetic feature models of different sizes (see scenario "Cross-Tree-Constraint Simplification"). Figure 8 shows the time needed for simplification of the cross-tree constraints in a quantile plot. It supports the general result

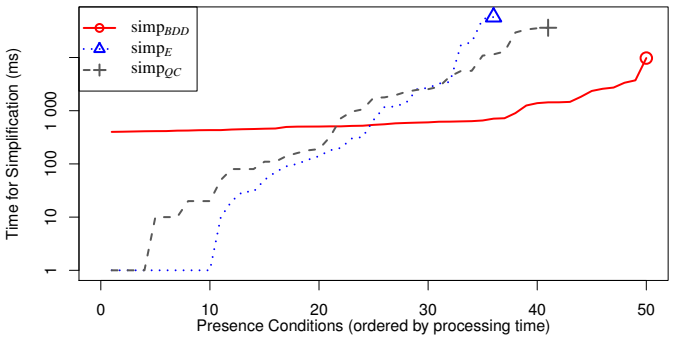


Fig. 8: Scalability of simplification algorithms shown in a quantile plot (E5), similar to the quantile plot in Figure 6

of E1: simp_{BDD} has a higher processing time for simple tasks (again, consider JVM startup time), but it can solve more tasks and is faster than simp_E and simp_{QC} when it comes to harder tasks. simp_{QC} performs better than simp_E, which was not to be expected, because it is an earlier algorithm solving the same problem. We were not able to run a complete evaluation for larger problem sizes, because the computation of the input files for simp_E and simp_{QC} is very expensive for harder problems. When evaluating presence-condition simplification on 10 problem instances with 150 options, simp_{BDD} still needs only 9 ms, on average (when the BDD is already loaded in memory). In summary, the answer to RQ2 is that simp_{BDD} has a high base processing-time (500 ms) but scales better than the other algorithms.

D. Threats to Validity

A threat to internal validity is that we have no exact measure for the simplicity of Boolean expressions. This is a problem that is not specific to our work; in general, it is difficult to define such a measure. We have tried a number of different measures and observed similar results, so we expect our observations to hold with other sensible measures. In addition, we tried to use a ground truth for the minimal size measure in our experiments. However, even small problems require an infeasible amount of computations. Hence, we focus on the comparison between the algorithms.

Another threat to internal validity is that we used existing tools to compare the algorithms, so we rely on that the tools actually implement the algorithms correctly. Still, we verified that each simplification result satisfies Equation 1.

A threat to the external validity—as always—is the selection of subject systems and application scenarios. To mitigate this threat, we selected a diverse set of application scenarios and subject systems. Our assumption that simplification can significantly improve presence-condition size holds in all these scenarios (except for code-annotation simplification).

A possible threat to the external validity is that we can only handle Boolean options. However, it has been shown that the majority of presence conditions in large configurable systems (e.g., kernels of LINUX and FREEBSD) can be expressed using only propositional expressions [7], and that the large majority of options in configurable systems software is Boolean [8]. Even if parts of the context cannot be expressed with Boolean options, a partial context can be used for simplification.

Even though the majority of our subject systems and application scenarios are real, our approach is certainly limited with respect to very large presence conditions. We tried larger problem instances in experiment E5. However, for one variability model with 100 options, we had to generate an input file for simp_E with 637 GB, which is not feasible. For very large presence conditions, the simplified presence condition will probably still be quite large, so it is questionable whether simplification is even useful in such cases. We argue that even if the result expression is still large, every bit of size reduction helps if the result is used as input to an analysis tool. If a user interprets the (still large) result, the user might use tools such as dependency graphs. Such graphs are simpler once redundant information is removed by simplification.

V. Related Work

Simplification of presence conditions in the context of highly configurable systems has not been investigated before. However, work on variability-model reasoning and the extraction of presence conditions is related.

Variability-model reasoning aims at analyzing properties of variability models (e.g., consistency) or of sets of models (e.g., relationships between models)—to assure correctness, and to support evolution and configuration of systems. A common reasoning operation is to calculate differences between two variability models. This problem is closely related to ours and has been explored before [1, 17, 48]. In general, computing differences (diffs) between two entities a and b involves two tasks: stripping the information of a from b and vice versa. Thus, a diff is a pair of sub-comparisons. The main difference to our simplification problem is that a diff has to be exact. That is, *all* the information of b is removed from a and *only* the remainder is presented to the user. Therefore, applying model-differencing methods on presence conditions has fewer means to influence the size of the presence condition. Our problem formulation gives us more leverage: Information that is contained in p and m can either remain in $\text{simp}(p,m)$ or be removed. We use this leverage to make $\text{simp}(p,m)$ smaller and more readable.

Off-the-shelf reasoners, such as BDD libraries or SAT solvers, are used for reasoning about Boolean expressions. Scalability experiments [33, 35, 36] show that SAT solvers are more scalable than BDDs for most analyses on feature models. However, BDDs are efficient for analyses that rely on enumerating configurations; they are known to scale up to models with 2000 features [35]. In our experiments, simp_{BDD} exhibited a better scalability than simp_E and simp_{QC}, but for smaller models and presence conditions, and for a very different analysis, which has not been investigated before. Good news is that the algorithms we tested scaled on generated models, which are usually more complex and harder to reason about than real-world models [39]. We are not aware of any SAT-based algorithm applicable to our problem. However, investigating the feasibility of using a SAT solver would be valuable future work.

Various researchers have extracted and analyzed presence conditions in the context of highly configurable systems. Presence conditions have been extracted using static analysis from build systems [6, 38], and using dynamic analysis by compiling individual system variants [21]. All these pieces of work show the importance of complex presence conditions to realize the mapping between the variability model and implementation. In fact, presence conditions in source code and other artifacts are means to maintain the variability model by establishing a balance between constraints residing in the model and in other artifacts. Our experiments (E3) have shown that the size of presence conditions in real systems is moderate, suggesting that such systems are relatively well maintained.

In our search for simplification algorithms, we have also looked at several research areas related to BDDs and two-level-logic minimization. In particular decomposable negation normal forms [19, 25], semantic tableaux [18], and minimization of propositional formulae [31] appeared promising at first sight, but in the end, we have not found algorithms applicable to our problem.

VI. Conclusion

We formally defined the problem of *presence-condition simplification*: A presence condition p is simplified with respect to its context m, in which the presence condition is used. It is not necessary (and even obstructive) for p to include information that is already guaranteed by m, which offers simplification potential. We provide solutions to this problem by mapping presence-condition simplification to related problems developed in the 1980/90s for different purposes, namely two-level logic minimization and the RESTRICT algorithm.

We discussed a variety of application scenarios for presence-condition simplification in the area of development and analysis of highly configurable systems. The application scenarios include the simplification of (1) presence conditions representing defects found by variability-aware analysis tools, (2) #if conditions in source code and in variability-aware ASTs, and (3) cross-tree constraints in variability models.

In a series of experiments, we evaluated the different algorithms concerning their effectiveness and processing time as well as the different application scenarios concerning their potential for simplification. In our experiments, we found substantial potential for simplification of presence conditions in many real use cases (e.g., performance prediction or defect reporting), and the algorithms are suited well for simplification. This suggests that our approach may produce good results in other contexts, too. Our experiments have shown that reduction factors are usually better with simp_{BDD} than with simp_{QC} or simp_{E}. Concerning scalability, we have shown that simp_{BDD} can handle larger problems than simp_{QC} or simp_{E}.

In future work, we plan to include presence-condition simplification in a variety of analysis tools for configurable systems. Furthermore, our work can serve as a basis for researchers to further investigate algorithms for presence-condition simplification.

Acknowledgements

We thank Olivier Coudert for information on the relationship of the RESTRICT algorithm to two-level logic minimization, and Ilia Polian for suggesting two-level logic minimization as a solution. This work has been supported by the DFG grants AP 206/4, AP 206/6, and AP 206/7.

Appendix

TABLE III: Example for a presence condition given as configuration table. p_1 is the DNF expression for r_1. For brevity, we omit conjunctions ($\wedge$) and denote negations with overlines.

o_1	o_2	o_3	o_4	r_1	r_2	$p_1 =$	$m =$
0	0	1	0	✓	X		$(\overline{o_1}\overline{o_2}o_3\overline{o_4})$
0	0	1	1	✓	X		$\vee(\overline{o_1}\overline{o_2}o_3o_4)$
1	1	0	0	X	X	$(o_1o_2\overline{o_3}\overline{o_4})$	$\vee(o_1o_2\overline{o_3}\overline{o_4})$
1	1	0	1	✓	✓		$\vee(o_1o_2\overline{o_3}o_4)$
1	1	1	0	X	✓	$\vee(o_1o_2o_3\overline{o_4})$	$\vee(o_1o_2o_3\overline{o_4})$
1	1	1	1	X	✓	$\vee(o_1o_2o_3o_4)$	$\vee(o_1o_2o_3o_4)$

Input-Format Conversion: Different application scenarios of presence-condition simplification require different input formats for the parameters p and m. Conversion between these formats is a non-trivial technical issue.

The E-Mail example of Figure 1 illustrates a simple scenario: The presence condition is aggregated to a single expression that covers multiple configurations. In other scenarios, a presence condition is given as a table, in which a row represents a configuration and the corresponding analysis result. In Table III, we illustrate this format by means of a system with configuration options o_1, o_2, o_3, and o_4 and two results r_1 and r_2 (e.g., for two different specifications). As an example, assume that configurations of the system are tested, and we mark failed configurations with X. Configurations that are not valid with respect to the variability model are not listed (e.g., $\neg o_1 \wedge \neg o_2 \wedge \neg o_3 \wedge \neg o_4$).

We can build the expressions p and m by forming the disjunction of the respective configurations in the table. For example, to build p_1 (representing the failed configurations in r_1), we form the disjunction of all rows for which column r_1 has an X. To build the variability model m, we form the disjunction of all rows (shown in Table III).

To merge DNF clauses, if possible, we use BDDs, because BDDs can automatically optimize the representation. Besides a reduction in BDD size and an improvement for simp_{BDD}, this also benefits simp_{QC} and simp_{E}, because we build the input sets for simp_{QC} and simp_{E} using the reduced BDD. Each path from the root node to the *true* terminal in the BDD translates to one row in the input table for simp_{QC} and simp_{E}. Often these paths do not use all configuration options (e.g., (o_1o_2)). In the input of simp_{QC} and simp_{E}, we would use "-" as value for o_3 and o_4 to indicate that this DNF clause does not depend on o_3 and o_4. This decreases the size of the ON and OFF sets to be considered by two-level logic minimization. The DC set needs not be stated explicitly, because it can be inferred as complement of ON- and OFF-set. In CNF-based scenarios, we apply the FORCE algorithm [2] to optimize the variable ordering in the BDD and improve the input for simp_{BDD}, simp_{QC}, and simp_{E}.

References

[1] M. Acher. *Managing Multiple Feature Models: Foundations, Language and Applications*. PhD thesis, Université Nice-Sophia Antipolis, 2011.

[2] F. Aloul, I. Markov, and K. Sakallah. FORCE: A Fast and Easy-to-Implement Variable-Ordering Heuristic. In *Proc. GLSVLSI*, pages 116–119. ACM, 2003.

[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, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proc. ICSE*, pages 482–491. IEEE, 2013.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6), 2010.

[6] T. Berger, S. She, K. Czarnecki, and A. Wąsowski. Feature-to-Code Mapping in Two Large Product Lines. Technical report, Department of Computer Science, University of Leipzig, 2010.

[7] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski. Feature-to-Code Mapping in Two Large Product Lines. In *Proc. SPLC*, pages 498–499. ACM, 2010.

[8] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, 2013.

[9] R. Brayton, A. Sangiovanni-Vincentelli, C. McMullen, and G. Hachtel. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer, 1984.

[10] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[11] D. Buchfuhrer and C. Umans. The Complexity of Boolean Formula Minimization. *Journal of Computer and System Sciences*, 77(1):142–153, 2011.

[12] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.

[13] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *Proc. ICSE*, pages 321–330. ACM, 2011.

[14] O. Coudert, C. Berthet, and J. Madre. Verification of Synchronous Sequential Machines Based on Symbolic Execution. In *Proc. AVMFSS*, pages 365–373. Springer, 1990.

[15] O. Coudert and T. Sasao. Two-level Logic Minimization. In *Logic Synthesis and Verification*, pages 1–27. Kluwer, 2002.

[16] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. GPCE*, pages 422–437. Springer, 2005.

[17] K. Czarnecki and A. Wąsowski. Feature Diagrams and Logics: There and Back Again. In *Proc. SPLC*, pages 23–34. IEEE, 2007.

[18] M. D'Agostino. Tableau Methods for Classical Propositional Logic. In *Handbook of Tableau Methods*, pages 45–123. Springer, 1999.

[19] A. Darwiche and P. Marquis. A knowledge compilation map. *Artificial Intelligence Research*, 17:229–264, 2002.

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

[21] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A Robust Approach for Variability Extraction from the Linux Build System. In *Proc. SPLC*, pages 21–30. ACM, 2012.

[22] E. McCluskey. Minimization of Boolean functions. *The Bell System Technical Journal*, 35(5):1417–1444, 1956.

[23] R. Hall. Fundamental Nonmodularity in Electronic Mail. *Automated Software Engineering*, 12(1):41–79, 2005.

[24] E. Hemaspaandra and H. Schnoor. Minimization for Generalized Boolean Formulas. In *Proc. IJCAI*, pages 566–571. AAAI, 2011.

[25] J. Huang and A. Darwiche. On Compiling System Models for Faster and More Scalable Diagnosis. In *Proc. AAAI*, pages 300–306. MIT, 2005.

[26] 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.

[27] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-based Product Lines. *ACM Transactions on Software Engineering and Methodology*, 21(3):14:1–14:39, 2012.

[28] 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. OOPSLA*, pages 805–824. ACM, 2011.

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

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

[31] P. Liberatore. Redundancy in logic i: {CNF} propositional formulae. *Artificial Intelligence*, 163(2):203–232, 2005.

[32] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proc. ESEC/FSE*, pages 81–91. ACM, 2013.

[33] M. Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009.

[34] M. Mendonça, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proc. OOPSLA*, pages 761–762. ACM, 2009.

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

[36] M. Mendonça, A. Wąsowski, K. Czarnecki, and D. Cowan. Efficient Compilation Techniques for Large Scale Feature Models. In *Proc. GPCE*, pages 13–22. ACM, 2008.

[37] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proc. ICSE*, pages 140–151. ACM, 2014.

[38] S. Nadi and R. Holt. The Linux Kernel: A Case Study of Build System Variability. *J Softw.*, 2013.

[39] R. Pohl, V. Stricker, and K. Pohl. Measuring the Structural Complexity of Feature Models. In *Proc. ASE*, pages 454–464. IEEE, 2013.

[40] W. Quine. *The Problem of Simplifying Truth Functions*. Mathematical Association of America, 1952.

[41] K. Schmid, R. Rabiser, and P. Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proc. VaMoS*, pages 119–126. ACM, 2011.

[42] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse Engineering Feature Models. In *Proc. ICSE*, pages 461–470. IEEE, 2011.

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

[44] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines: Footprint and Memory Consumption. *Information and Software Technology*, 55(3):491–507, 2013.

[45] N. Siegmund, A. von Rhein, and S. Apel. Family-Based Performance Measurement. In *Proc. GPCE*, pages 95–104. ACM, 2013.

[46] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *Proc. USENIX*, pages 421–432. USENIX Association, 2014.

[47] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.*, 47(1):6:1–6:45, 2014.

[48] T. Thüm, D. Batory, and C. Kästner. Reasoning About Edits to Feature Models. In *Proc. ICSE*, pages 254–264. IEEE, 2009.