Propagating Configuration Decisions with Modal Implication Graphs

Sebastian Krieter
University of Magdeburg, Germany
Harz University of Applied Sciences
Wernigerode, Germany
sebastian.krieter@ovgu.de

Thomas Thüm
TU Braunschweig, Germany
t.thuem@tu-braunschweig.de

Sandro Schulze
Reimar Schröter
Gunter Saake
University of Magdeburg, Germany
sandro.schulze@ovgu.de
reimar.schroeter@ovgu.de
saake@iti.cs.uni-magdeburg.de

ABSTRACT

Highly-configurable systems encompass thousands of interdependent configuration options, which require a non-trivial configuration process. Decision propagation enables a backtracking-free configuration process by computing values implied by user decisions. However, employing decision propagation for large-scale systems is a time-consuming task and, thus, can be a bottleneck in interactive configuration processes and analyses alike. We propose modal implication graphs to improve the performance of decision propagation by precomputing intermediate values used in the process. Our evaluation results show a significant improvement over state-of-the-art algorithms for 120 real-world systems.

CCS CONCEPTS

• **Software and its engineering → Software product lines**;

KEYWORDS

Software product line, Configuration, Decision Propagation

ACM Reference Format:
Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. 2018. Propagating Configuration Decisions with Modal Implication Graphs. In *ICSE '18: ICSE '18: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3180155.3180159

1 INTRODUCTION

Highly-configurable systems consist of thousands of configuration options also known as *features* [16, 18, 81]. This enormous and even growing amount of variability poses challenges for established algorithms used to analyze configurable systems [12, 89]. In particular, the variability analysis of large-scale systems, including their configuration, is still challenging as these tasks are computationally complex problems.

The features of a configurable system are typically connected by interdependencies that result from interactions within the system [64]. Examples of these dependencies are features that require another feature's functionality and features that are mutually exclusive [49]. In order to configure a working system variant, all dependencies of a configurable system must be considered. Thus, every decision a user makes in a configuration (i.e., selecting a feature) can imply to the inclusion or exclusion of other features.

During the configuration process it is often critical for users to immediately know the consequences of their decisions to avoid unwanted effects later on. For example, some users aim to configure a server system with a certain operating system and traffic monitoring. However, their chosen monitoring application is incompatible with their operating system. If they are unaware of such dependencies, their configured system variant is invalid. As real systems may contain thousands of interdependent configuration options, finding contradictions within a configuration manually is not feasible.

Decision propagation guarantees that users are informed about all consequences of their decisions at any point during the configuration process. Decision propagation determines the features that are implied or excluded by user decisions [42, 43, 59]. In an interactive configuration process, decision propagation prevents users from making contradictory decisions and reduces the amount of decisions a user has to make. By employing decision propagation in our example, users, who chose a particular monitoring application or operating system, can immediately notice the respective dependency and adjust their configuration accordingly (e.g., by choosing an alternative monitoring application).

Decision propagation is a computationally expensive task. In general, decision propagation is NP-hard as it involves finding valid assignments for interdependent boolean variables, also known as the boolean satisfiability problem (SAT), which is NP-complete [25]. With FeatureIDE, we have implemented decision propagation ten years ago and did not face scalability problems while using smaller feature models. However, when our industry partner used FeatureIDE with systems having more than 18,000 features, propagation of a single decision took over 20 seconds on average, summing up to 13 hours to create one configuration without even considering the time required to reason about decisions and to interact with the tool. While modern decision-propagation techniques can reduce this time to a feasible level for human interaction, decision propagation is still a bottleneck within automated configuration processes such as t-wise sampling [2].

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5638-1/18/05…$15.00
https://doi.org/10.1145/3180155.3180159

We aim to speed up decision propagation by means of *modal implication graphs*. We propose a graph-assisted decision propagation algorithm to reduce the number of satisfiability problems to be solved. The idea is to precompute implicit dependencies between features and to make this information available during configuration. In particular, we distinguish between two phases of the configuration process. In the *offline* phase a modal implication graph is constructed according to the feature dependencies of a configurable system, which only needs to be re-executed if these dependencies change. The actual decision propagation is part of the *online* phase, in which our proposed algorithm traverses the modal implication graph to determine the implied values. Our goal is to reduce the computational effort needed during the online phase and, thus, improve the response time of an interactive configuration process and analyses that derive configurations (e.g., t-wise sampling). We empirically evaluate the impact on the response time by comparing decision propagation with and without modal implication graphs. In short, our contributions are:

- We introduce modal implication graphs to represent and access feature dependencies efficiently.
- We propose an algorithm to propagate decisions using modal implication graphs.
- We provide an open-source implementation of our algorithm as part of the FeatureIDE framework.[1]
- We provide a large-scale evaluation of decision propagation, including existing algorithms, in terms of performance during the offline and online phase.

2 FEATURE MODELS AND CONFIGURATIONS

A feature model expresses the variability of a configurable system by defining distinct configuration options, called features, and their interdependencies [8, 11, 26]. A configuration defines a selection of features that are used to derive a particular system variant [8].

A common representation for feature models is a feature diagram, which represents feature dependencies in form of a tree structure [29]. A more general way to describe feature models are propositional formulas [11, 29]. Though propositional formulas lack structural information, they can express any arbitrary boolean constraint between any group of features [49].

In this paper, we use the representation of feature models as propositional formulas in conjunctive normal form (CNF). This representation can be applied to all feature model representations that use boolean constraints (e.g., feature diagrams can be converted into CNF) [1, 11, 29, 49]. An advantage of CNF is that it allows for automated reasoning about features and their valid configuration using satisfiability (SAT) solvers [1, 61].

In Fig. 1, we depict the feature diagram of our running example, the configurable *Server* system. The feature *Server* is the root of the feature tree and represents the common part of all variants of the configurable system. The features *File System (FS)* and *Operating System (OS)* are both mandatory children of *Server*, which means each variant that contains *Server* must also contain *FS* and *OS*. By contrast, the feature *Logging (Log)* is an optional child of *Server* and is not required if *Server* is part of a variant. The children of *FS* (i.e., *NTFS*, *HFS*, and *EXT*) are part of an or-group, which means

[1] https://github.com/FeatureIDE/FeatureIDE

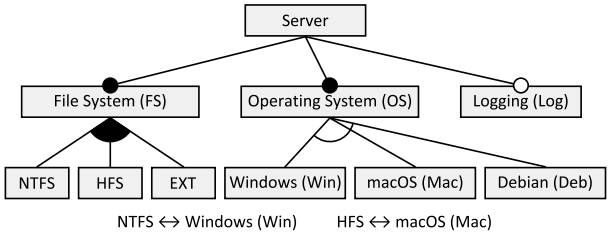

Figure 1: Feature model for configurable *Server* system.

that at least one of them must be part of a variant that contains *FS*. The children of *OS* (i.e., *Windows (Win)*, *macOS (Mac)*, and *Debian (Deb)*) are part of an alternative, which means that if *OS* is part of a variant, exactly one of them must be present too. Additionally, the feature model contains two cross-tree constraints, a bi-implication between *NTFS* and *Win* and a bi-implication between *HFS* and *Mac*.

Formally, we define a feature model $\mathcal{FM} = (F, R)$ as a pair of a set of features (i.e., variables) F and a set of constraints (i.e., clauses) R. The set of features F contains all features of the configurable system (i.e., $F = \{f_1, ..., f_n\}$ with n being the number of features). From F we infer the set of literals $L = \{l_1, \neg l_1, ..., l_n, \neg l_n\}$ that represent the selection states of each feature, where l_i indicates a selection and $\neg l_i$ a deselection of feature f_i. In particular, we assume that $\neg\neg l_i$ is equal to l_i. We define the set of constraints R as $R \subseteq 2^L$. Each element in R represents a clause from the CNF. Following our definition, the feature model depicted in Fig. 1 is defined as:

$$\begin{aligned}\mathcal{FM} = (&\{Server, OS, FS, Log, Win, Mac, Deb, NTFS, HFS, EXT\},\\ &\{\ \{Server\}, \{\neg\, Win, NTFS\}, \{Win, \neg\, NTFS\},\\ &\ \ \{Mac, \neg\, HFS\}, \{\neg\, Mac, HFS\}, \{OS\}, \{FS\},\\ &\ \ \{\neg\, Win, \neg\, Mac\}, \{\neg\, Win, \neg\, Deb\}, \{\neg\, Mac, \neg\, Deb\},\\ &\ \ \{NTFS, HFS, EXT\}, \{Win, Mac, Deb\}\ \})\end{aligned}$$

A configuration consists of a selection of features from a feature model. Formally, we define a configuration c for a feature model $\mathcal{FM} = (F, R)$ as a set of literals in L, such that $c \in C$, with $C = \{c \subset L \mid \forall l \in L : \{l, \neg l\} \nsubseteq c\}$. If a feature f_i is selected (i.e., $l_i \in c$) or deselected (i.e., $\neg l_i \in c$) in a configuration we call it *defined* and otherwise *undefined*. Thus, a feature can have one of three *selection states*, selected, deselected, or undefined. We call a configuration c *complete* if every feature is defined (i.e., $complete(c, \mathcal{FM}) := |c| = |F|$), otherwise the configuration is *partial*. If a configuration c satisfies all constraints in R it is *valid* (i.e., $valid(c, \mathcal{FM}) := \forall r \in R : r \cap c \neq \emptyset$), otherwise it is *invalid*. Additionally, we call a configuration *satisfiable*, if it is valid or can be made valid by defining more features (i.e., $satisfiable(c, \mathcal{FM}) := \exists c' \in C : c \subseteq c' \wedge valid(c', \mathcal{FM})$).

Feature models may contain anomalies [13] such that certain features have to or must not be selected. A feature f_i is *dead* iff it is deselected in any valid configuration (i.e., $\nexists c : l_i \in c \wedge valid(c, \mathcal{FM})$). Likewise, a feature f_i is *core* iff it is not deselected in any valid configuration (i.e., $\nexists c : \neg l_i \in c \wedge valid(c, \mathcal{FM})$). Iff a feature is neither dead nor core we refer to it as *variant*.

The definitions of dead and core features can be generalized by taken arbitrary partial configurations into account. Given a

satisfiable configuration c and a feature f_i with $\{l_i, \neg l_i\} \cap c = \emptyset$, f_i is *conditionally dead* iff $\neg$ *satisfiable*$(c \cup \{l_i\}, \mathcal{FM})$ and *conditionally core* iff $\neg$ *satisfiable*$(c \cup \{\neg l_i\}, \mathcal{FM})$. The set of conditionally dead and core features represents all features that are not explicitly defined in a satisfiable configuration, but are implied or excluded implicitly by the selection states of other features. Therefore, we use the definition of conditionally dead and core features as formal basis of decision propagation.

3 SAT-BASED DECISION PROPAGATION

Ideally, users create a complete and valid configuration in an interactive configuration process by successively adding new literals to an initially empty set. With interactive, we mean that users are getting feedback during the configuration process. Without further guidance, it is possible that users select contradictory features and later have to backtrack their steps to undo one or more of their decisions. Decision propagation can be used to avoid backtracking, meaning that users never have to revoke their decisions in order to obtain a valid configuration. Given a satisfiable partial configuration and a corresponding feature model, decision propagation determines all features that are implicitly defined (i.e, conditionally dead and core features) and adds them to the given configuration. For instance, consider the satisfiable partial configuration $c = \{Server, OS, FS, NTFS\}$ for our running example. Only four features are explicitly defined by the given configuration. However, given the feature model's dependencies some undefined features are conditionally dead or core. For example, if users select the feature *Mac*, the configuration will be invalid no matter what other features they select. Thus, they must backtrack their steps until they resolve this conflict. In contrast, if they apply decision propagation, the resulting configuration $c' = \{Server, OS, FS, NTFS, Win, \neg Mac, \neg Deb, \neg HFS\}$ includes all conditionally dead and core features.

A SAT-based algorithm for decision propagation reduces the problem to multiple instances of the SAT problem and solving these using optimized SAT solvers. In particular, the algorithm tests for each undefined feature f_i whether the given partial configuration c is still satisfiable if the f_i is selected (i.e., *satisfiable*$(c \cup \{l_i\}, \mathcal{FM})$) or deselected (i.e., *satisfiable*$(c \cup \{\neg l_i\}, \mathcal{FM})$).

In Alg. 1, we depict the general process of SAT-based decision propagation in pseudo code. The algorithm takes as input a feature model $\mathcal{FM}$ and a satisfiable configuration $c_{current}$, which includes the latest decision of the user. It returns a configuration c_{new} that contains all features that are conditionally dead or core. The main procedure decisionPropagation initializes c_{new} with the assignment from $c_{current}$ (cf. Line 2) and calls both sub-procedures getUnknown and test to determine the state of each undefined feature. getUnknown returns the set $L_{unknown}$ that contains all literals that must be checked using the SAT solver (cf. Line 3). test determines for each literal in $L_{unknown}$ whether it is conditionally dead or core (cf. Line 4–6). Both, c_{new} and $L_{unknown}$ are updated by test. Finally, the updated c_{new} is returned.

We show a straight-forward approach to implement both sub-procedures getUnknown and test in Line 9–17 of Alg. 1. getUnknown initializes $L_{unknown}$ by adding two literals for each currently undefined feature (cf. Line 10). test investigates a single literal l_{test} from $L_{unknown}$ by adding it to c_{new} and querying the SAT solver (cf. Line 13). If there is no satisfiable configuration corresponding to $c_{new} \cup \{l_{test}\}$, every configuration that contains all literals from c_{new} must also contain the complement of l_{test} (i.e., $\neg l_{test}$). Thus, the feature corresponding to l_{test} is either conditionally core (i.e., l_{test} is negative) or dead (i.e., l_{test} is positive). Consequently, $\neg l_{test}$ is added to c_{new} (cf. Line 15).

Algorithm 1 Naïve SAT-based testing algorithm

Require: $\mathcal{FM} = (F, R)$ – feature model
$c_{current}$ – current configuration
Return: c_{new} – new configuration
Global: $\mathcal{FM}, L_{unknown}, c_{new}$

```
1:  procedure decisionPropagation()
2:      c_new ← c_current
3:      L_unknown ← getUnknown()
4:      for all l_test ∈ L_unknown do
5:          test( l_test )
6:      end for
7:      return c_new
8:  end procedure
9:  procedure getUnknown()
10:     return {l ∈ L | c_current ∩ {l, ¬l} = ∅}
11: end procedure
12: procedure test( l_test )
13:     c_solution ← sat( FM, c_new ∪ {l_test} )
14:     if c_solution = ∅ then
15:         c_new ← c_new ∪ {¬l_test}
16:     end if
17: end procedure
```

From Alg. 1, we can see that for a single execution of decision propagation the naïve algorithm has to call the SAT solver twice for each undefined feature. Modern SAT solvers already apply techniques such as incremental solving, which aims to reduce the computing effort for repetitive SAT queries by learning and reusing clauses that are implicitly implied by the original formula [32, 52]. However, even when employing modern SAT solving techniques, the naïve algorithm does not scale to large configurable systems. Janota proposed a more efficient way to determine whether a feature is conditionally dead or core given a partial configuration [43]. His advanced SAT-based algorithm employs the same concept, but tries to reduce the number of tested literals. We show the relevant code changes in pseudo code in Alg. 2. The configurations found by the SAT solver are used to exclude literals from $L_{unknown}$ (cf. Line 3, 10). With this algorithm, our evaluation machine only required several minutes for configuring a system with more than 10,000 features. Janota's algorithm also uses a special selection strategy for the SAT solver, which determines the order in which the solver considers literals to find a solution [43]. For brevity, we exclude this selection strategy from the pseudo code.

4 GRAPH-ASSISTED DECISION PROPAGATION

We aim to enhance SAT-based decision propagation by reducing the number of necessary queries to the SAT solver. For this, we

Algorithm 2 Advanced SAT-based testing algorithm

```
1:  procedure getUnknown()
2:      c_solution ← sat( FM, c_new )
3:      return {l ∈ L | c_current ∩ {l, ¬l} = ∅} \ c_solution
4:  end procedure
5:  procedure test( l_test )
6:      c_solution ← sat( FM, c_new ∪ {l_test} )
7:      if c_solution = ∅ then
8:          c_new      ← c_new ∪ {¬l_test}
9:      else
10:         L_unknown ← L_unknown \ c_solution
11:     end if
12: end procedure
```

propose *modal implication graphs* (MIGs), which represent certain relationships between features in the feature model. In our approach, we differentiate between an offline phase, in which we compute a modal implication graph for a particular feature model and an online phase, in which we use it for decision propagation. The offline phase only needs to be executed when the feature model is modified, whereas the online phase is part of every configuration process. Our approach is based on two observations we made regarding decision propagation for large-scale feature models:

(1) In most cases the definition of a feature only affects a small set of other features.
(2) If other features are affected, it often results from binary *requires* and *excludes* constraints.

Thus, we introduce modal implication graphs in Section 4.1 to (1) identify the set of affected features and (2) determine some of their selection states in the configuration. In Section 4.2, we present an algorithm to derive a modal implication graph and describe the graph's usage for graph-assisted decision propagation in Section 4.3.

4.1 Modal Implication Graphs

An implication graph is a directed graph that describes a propositional formula consisting of a conjunction of implications between single literals (i.e., binary relations). Each node represents a literal of a variable and a directed edge from one node to another represents a binary relation. A feature model's binary dependencies can be transformed into an implication graph by representing each feature by a positive and a negative node and each requires and excludes constraint by an edge [29].

Analogous to an implication graph, the modal implication graph for a feature model consists of nodes that represent the literals of each feature and directed edges that represent the relations between these literals. We extend implication graphs by introducing an additional edge type to express n-ary relations, such as or-groups with three or more features.

We differentiate between two types of edges, *weak* and *strong*. A strong edge indicates a binary relation (e.g., requires and excludes) between two features. If the literal of the source node is element of a configuration, then also the literal of the destination node must be element of the configuration. In contrast, a weak edge indicates that two literals are part of an n-ary constraint, which involves at least one other literal. For a partial configuration these weak edges could become strong edges due to the selection and deselection of other features.

A modal implication graph for our running example can be seen in Fig. 2. Due to the alternative group, there is a strong edge from *Mac* to ¬ *Win*, which implies that if feature *Mac* is selected feature *Win* must be deselected (i.e., $Mac \in c \rightarrow \neg\, Win \in c$). In addition, there is a weak edge from ¬ *Win* to *Mac*, also resulting from the alternative group, which means that under certain conditions (i.e., if feature *Deb* is deselected) *Mac* must be selected if *Win* is deselected. In other words, this edge will become strong, if *Deb* is deselected.

Formally, we define a modal implication graph $\mathcal{G} = (L, S, W)$ to be a triple consisting of a set of nodes L, which is equal to the set of literals, a set of strong edges S, and a set of weak edges W, such that $S, W \subseteq \{(l_{start}, l_{end}) \mid l_{start}, l_{end} \in L, l_{start} \neq l_{end}\}$ and $S \cap W = \emptyset$. As implications are transitive, we are interested not only in edges, which directly connect two literals, but also in paths within a graph. Analogous to edge types, we also differentiate between *strong* and *weak paths*. We call a path from one node to another *strong*, if it consists solely of strong edges and *weak*, if it contains at least one weak edge. If there exists at least one strong path from a node l_s to a node l_t we denote this with $l_s \blacktriangleright_{\mathcal{G}} l_t$ and call l_t *strongly connected* to l_s. Complementary to this, if there exist only weak paths from l_t to l_s, we denote this with $l_s \triangleright_{\mathcal{G}} l_t$ and call l_t *weakly connected* to l_s. Within the context of a modal implication graph $\mathcal{G}$, we define $\blacktriangleright_{\mathcal{G}}$ and $\triangleright_{\mathcal{G}}$ as:

$$\blacktriangleright = \{(l_s, l_t) \in L^2 \mid (l_s, l_t) \in S \vee \exists\, l_m \in L : l_s \blacktriangleright l_m \blacktriangleright l_t\}$$

$$\triangleright = \{(l_s, l_t) \in L^2 \mid l_s \not\blacktriangleright l_t, (l_s, l_t) \in W \vee \exists\, l_m \in L : (l_s \triangleright l_m \triangleright l_t) \vee (l_s \triangleright l_m \blacktriangleright l_t) \vee (l_s \blacktriangleright l_m \triangleright l_t)\}$$

Strongly connected literals are directly dependent, while weakly connected literals also depend on other literals. Moreover, non-connected literals are completely independent of each other. Thus, we can use the modal implication graph to understand the relationship between any two features by looking at the paths between their respective nodes.

For instance, in the modal implication graph from the Server system, ¬ *NTFS* is strongly connected to *Mac* (via ¬ *Win*). Hence, if *Mac* is selected in a satisfiable configuration, *NTFS* must be deselected. In contrast, *Deb* is weakly connected to ¬ *NTFS* (via ¬ *Win*). Thus, if *NTFS* is deselected in a satisfiable configuration, *Deb* must be selected for certain conditions, which, in this example, is the deselection of *Mac*. Finally, no node is connected to *Log*, which means that it is independent of all other literals.

4.2 Deriving Modal Implication Graphs (Offline Phase)

Before using a modal implication graph in decision propagation (i.e., online phase), we need to derive it from a feature model (i.e., offline phase). If the corresponding model evolves, the graph must be recreated, before the next decision propagation. A modal implication graph is constructed by creating and analyzing a feature model's CNF. The CNF clauses can be categorized by their number of literals. Clauses containing just one literal (i.e., unit-clauses) can be ignored, as they describe no relationship between features, but simply make the respective features core or dead. Clauses with exactly two literals (i.e., two-literal clauses) are used to derive strong

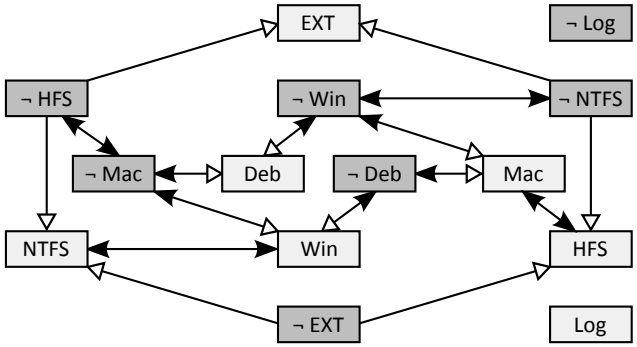

Figure 2: Modal implication graph for *Server* system.

edges. Weak edges are derived from the remaining clauses, which contain more than two literals (i.e., n-literal clauses). We propose the following four steps to derive a graph with minimal number of nodes and edges.

Step 1 – Adding Nodes. In the first step, we initialize the graph by adding two literal nodes for each variant feature. As dead and core features are not relevant for decision propagation, we only add variant features, to reduce the graph's memory space consumption and save unnecessary traversal during the online phase. We compute the set of variant features by running Alg. 2 with an empty configuration. In our running example the features *Server*, *OS*, and *FS* are core and, thus, excluded from the graph (cf. Fig. 2).

Step 2 – Adding Edges. In the second step, we add edges according to the following pattern. For each two-literal clause, we add two strong edges to the graph, as each two-literal clause can be transformed into two equivalent implications. For instance, the clause $\neg\, Win \vee \neg\, Mac$ can be transformed into $Win \Rightarrow \neg\, Mac$ and $Mac \Rightarrow \neg\, Win$. Thus, we add two strong edges from *Win* to $\neg\, Mac$ and from *Mac* to $\neg\, Win$. Likewise, an n-literal clause can be transformed into n equivalent implications. In this case, we add a weak edge for each pairwise combination of literals (i.e., in total $n \cdot (n-1)$). For example, the clause $NTFS \vee HFS \vee EXT$ can be transformed into $\neg\, NTFS \Rightarrow (HFS \vee EXT)$, $\neg\, HFS \Rightarrow (NTFS \vee EXT)$, and $\neg\, EXT \Rightarrow (NTFS \vee HFS)$, resulting in six weak edges (e.g., from $\neg\, NTFS$ to *HFS* and to *EXT*).

Before transforming a clause, we test it for obvious redundancies. In particular, we remove literals that are duplicates or that can never be true (i.e., core and dead features) and remove clauses that are tautologies. For instance, in our example the clause $(\neg\, Win \vee OS)$ is a tautology, as *OS* is a core feature. Likewise, we could remove $\neg\, OS$ from $(\neg\, OS \vee Win \vee Mac \vee Deb)$, since it can never be true.

Step 3 – Adding Implicit Strong Edges. The initially constructed graph can contain implicit strong relationships between literals that are only weakly connected [7]. For instance, in the graph of our running example $\neg\, Deb$ is weakly connected to $\neg\, EXT$ (cf. Fig. 2). However, due to the feature model's or-group and cross-tree constraints, $\neg\, EXT$ directly implies $\neg\, Deb$ and, thus, there could be a strong edge from $\neg\, EXT$ to $\neg\, Deb$.

To find implicit strong edges, we employ a depth-first search that investigates all pairs of weakly connected nodes. We start the search with an arbitrary node A and consider each node B that is weakly connected to it. Using the SAT solver, we check, whether A implies B. In that case, we add a strong edge from A to B and recursively continue the search with B as new starting node. Due to this depth-first search, we are able to deduce strong edges by transitivity without querying the SAT solver (e.g. from $A \blacktriangleright B$ and $B \blacktriangleright C$, we can deduce that $A \blacktriangleright C$). We repeat the search with different starting literals until we checked every pair of weakly connected nodes.

Step 4 – Removing Redundant Weak Edges. Redundant weak edges are weak edges that can be removed from a graph without changing its reachability. They can occur due to the inclusion of implicit strong edges in step three and redundant constraint in the CNF. The removal of redundant edges saves memory and decreases the number of weak paths that must be traversed in the online phase.

To remove redundant weak edges, we consider their corresponding clauses in the CNF. We begin with ordering the list of all clauses by their number of literals in descending order, as larger clauses are more likely to be redundant. For each n-literal clause, we query the SAT solver to test whether the clause is redundant regarding the current list of clauses and remove it from the list if it is redundant. For example, we would remove the clause $(NTFS \vee HFS \vee EXT)$, if there would be an additional clause $(NTFS \vee HFS)$, as this clause subsumes the former. If a weak edge is no longer represented by any clause in the CNF, we remove it from the graph.

Alternative Construction Process. Step 3 and 4 are optional and can be considered a trade-off between offline and online time. As Step 3 is the most time consuming one, we evaluate its impact in our evaluation, regarding both, offline and online time. When step three was applied during the construction process, we call the resulting modal implication graph *complete* (because it contains all possible strong edges). In contrast, if the step was skipped, we call the resulting graph *incomplete*.

4.3 Using Modal Implication Graphs for Decision Propagation (Online Phase)

We employ modal implication graphs in our graph-assisted decision propagation algorithm to reduce the number of SAT queries during the online phase. The algorithm is based on Alg. 1 (cf. Section 3), but makes two major changes. First, it traverses strong paths for known literals to find implied literals without querying the SAT solver. Second, it only tests literals that can be reached via weak paths from the starting literal, which excludes features unaffected by a given decision.

In Alg. 3, we present the relevant differences over Alg. 1. In addition to a feature model $\mathcal{FM}$ and the current configuration $c_{current}$, the graph-assisted algorithm requires the modal implication graph $\mathcal{G}$ and the most recent decision in form of a literal l_{new}.

The procedure getUnknown (cf. Line 1–9) traverses the modal implication graph to find potentially implied literals. First, the algorithm checks whether there are any weak edges in the graph that can be transformed into strong edges according to the current configuration (cf. Line 2, 22–23). Second, the outgoing strong paths from l_{new} are traversed (cf. Line 3). Each literal that can be reached via a strong path is added to c_{new} (i.e., without any SAT queries).

Algorithm 3 Graph-assisted testing algorithm

Require: $\mathcal{FM} = (F, R)$ – feature model
$\mathcal{G} = (L, S, W)$ – modal implication graph
l_{new} – literal of most recent decision
Return: c_{new} – new configuration
Global: $\mathcal{FM}, \mathcal{G}, l_{new}, c_{new}, L_{unknown}$

```
1:  procedure getUnknown()
2:      G' ← updateGraph( G )
3:      c_new ← c_new ∪ {l ∈ L | l_new ▶_G' l}
4:      L_unknown ← {l ∈ L | {l, ¬l} ∩ c_new = ∅ ∧ l_new ▷_G' l}
5:      if L_unknown ≠ ∅ then
6:          L_unknown ← L_unknown \ sat( FM, c_new )
7:      end if
8:      return L_unknown
9:  end procedure
10: procedure test( l_test )
11:     c_solution ← sat( FM, c_new ∪ {l_test} )
12:     if c_solution = ∅ then
13:         G' ← updateGraph( G )
14:         L_strong ← {l ∈ L | ¬l_test ▶_G' l}
15:         c_new ← c_new ∪ {¬l_test} ∪ L_strong
16:         L_unknown ← L_unknown \ {¬l | l ∈ L_strong}
17:     else
18:         L_unknown ← L_unknown \ c_solution
19:     end if
20: end procedure
21: procedure updateGraph( G_old )
22:     R' ← {r \ {¬l | l ∈ c_new} | r ∈ R, r ∩ c_new = ∅}
23:     return updateEdges( G_old, R' )
24: end procedure
```

Third, all outgoing weak paths are traversed for l_{new}. All literals that belong to undefined features and can be reached via a weak path are added to L_{unkown} (cf. Line 4).

The procedure test is similar to the procedure of the advanced SAT-based algorithm. In addition, test utilizes the modal implication graph whenever an implied literal is found by the SAT solver (cf. Line 13–16). For each found literal the graph's weak edges are updated again and the outgoing strong paths are traversed (cf. Line 13–14). All reachable literals are added to c_{new} and removed from $L_{unknown}$ (cf. Line 15–16).

5 EVALUATION

With modal implication graphs (MIGs), we aim to speed-up the online phase of decision propagation by doing further computations in the offline phase. Therefore, we evaluate the performance of different decision-propagation algorithms. We compare the offline and online execution time of graph-assisted decision propagation using modal implication graphs (cf. Section 4) against SAT-based decision propagation (cf. Section 3). In detail, we aim to answer the following research questions:

- RQ1 Does the choice of a decision propagation algorithm affect the execution time of the *offline* phase?
- RQ2 Does the choice of a decision propagation algorithm affect the execution time of the *online* phase?
- RQ3 Given a number of configuration processes (i.e., online phases), which decision propagation algorithm is superior to others in terms of overall execution time and memory consumption?

5.1 Experimental Setup

To answer our research questions, we perform two experiments with one factor and four treatments. In summary, we compare four different algorithms for decision propagation using 120 real-world systems. In addition to execution time, we measure the memory consumption of the derived modal implication graph for each feature model. In the following, we describe which configurable systems and decision-propagation algorithms we consider, how we designed the individual experiments, and what values we measure during a single experiment. All computations during the evaluation were run on a notebook with the following specifications: CPU: Intel Xeon E3-1505Mv5 (2.80 GHz), RAM: 64 GB, OS: Windows 7, Java Version: 1.8.0_121 (64-Bit). Memory consumption was measured using the Java Instrumentation package.

Configurable Systems (Subjects). In our evaluation, we use the feature models of 120 real-world configurable systems with varying sizes and complexity, which have been used in prior studies [17, 49]. The majority of these feature models (117) contain between 1,166 and 1,397 features. Of these 117 models, 107 comprise between 2,968 and 4,138 cross-tree constraints, while one has 14,295 and the other nine have between 49,770 and 50,606 cross-tree constraints. The remaining three models contain an even higher number of features. The feature models from the systems Automotive01, Automotive02, and Linux contain 2,513, 18,616, and 6,889 features and 2,833, 1,369, and 80,715 constraints, respectively.

Decision-Propagation Algorithms (Treatments). In our evaluation, we compare the following algorithms:

(1) Naïve SAT-based (NSAT)
(2) Advanced SAT-based (ASAT)
(3) Graph-assisted using an *incomplete* MIG (IMIG)
(4) Graph-assisted using a *complete* MIG (CMIG)

To ensure a fair comparison of all algorithms, we employ a white-box evaluation, where each algorithm uses the same base implementation as described in Section 3. We provide all algorithms as part of the open-source framework FeatureIDE[2] Our implementation uses Java and the default SAT solver of Sat4J (Version 2.3.5) [52]. With Sat4J, we are able to employ incremental SAT solving. For each feature model we create a separate solver, which is able to deduce new clauses while solving a query and later reuse these clauses in subsequent queries.

Each algorithm performs certain tasks during its offline phase. Both SAT-based algorithms (cf. Section 3) determine the core and dead features (i.e., initial decision propagation). In addition to computing core and dead features, both graph-assisted algorithms (cf. Section 4) derive a modal implication graph. While CMIG derives a complete graph (i.e., containing all explicit and implicit strong edges), IMIG derives an incomplete graph (cf. Section 4.2). The modal implication graph is implemented as an adjacency list, due

[2] https://github.com/skrieter/MIG-Evaluation

to reasons of memory efficiency. It is stored to and loaded from persistent memory using Java's serialization mechanism.

During their online phase, all algorithms calculate implied and excluded features, as described in Section 3 and 4. While the SAT-based algorithms solely query the SAT solver, the graph-assisted algorithms additionally traverse through a modal implication graph to avoid SAT queries.

Offline Phase (Experiment 1). To answer RQ1, we measure the execution time for each algorithm's offline phase. As stated above, the offline phase of each algorithm consists of all tasks after receiving the CNF of a feature model and before starting the actual configuration process. As the process of creating a CNF is independent from the chosen algorithm, we do not include it as part of the offline phase, but use the CNF as initial parameter for each algorithm. To avoid computational bias and calculate a representable mean value for each feature model and algorithm, we repeat the experiment 200 times. Furthermore, we compensate for the warm-up effect of the Java virtual machine (JVM) by performing an initial execution without any measurement.

Online Phase (Experiment 2). To answer RQ2, we measure the execution time for each algorithm's online phase. We simulate a configuration process by using random decisions, as we cannot know which decisions users would make in their configurations and want to avoid relying on false assumptions. The simulated configuration process consists of the following steps:

(1) Start with an empty configuration
(2) Randomly choose an undefined feature
(3) Randomly define the feature (i.e., select or deselect)
(4) Apply decision propagation
(5) Repeat 2–4 until all features are defined

We measure the execution time for each individual application of decision propagation. In the experiment, we neglect the time that a user would need to make configuration decisions (i.e., reasoning and input), as these values highly depend on the user and, thus, would bias our results. Furthermore, this is not an issue for automated configuration processes, such as t-wise sampling [2, 45].

We use a pseudo random generator, which has the advantage that we can fix the random seed for each iteration of the experiment. Therefore, we ensure that all algorithms get the same series of random decision and, thus, that the resulting configurations are equal. To get meaningful results, we repeat the experiment 200 times with different random seeds. Analogous to Experiment 1, we compensate for the JVM warm-up effect.

Hypotheses. In order to be able to draw meaningful conclusions, we formulate the following null hypotheses from our research question RQ1 and RQ2, respectively:

H_0^{RQ1} The execution time of the offline phase is the same for all investigated decision-propagation algorithms.

H_0^{RQ2} The execution time of the online phase is the same for all investigated decision-propagation algorithms.

We conduct two experiments with one factor (i.e, execution time) and four treatments (i.e, algorithms) using the same subjects (i.e, feature models). Hence, we test our hypotheses using a paired Wilcoxon-Mann-Whitney test. We choose 95% as our confidence interval. Our expectation is that the offline phase of both SAT-based algorithms is faster than the offline phase of the graph-assisted algorithms, but that they perform worse during the online phase. This is due to the difference in effort of the algorithms during the offline phase. NSAT and ASAT do the least amount of precomputations, while IMIG additionally derives an incomplete modal implication graph and CMIG even derives a complete modal implication graph. This means that, during the online phase, CMIG can access more information than IMIG, while IMIG has more information than ASAT and NSAT. Hence, we expect that for the offline phase NSAT and ASAT are faster than IMIG, which is again faster than CMIG. For the online phase, we expect that the fastest algorithm is CMIG, followed by IMIG, ASAT, and NSAT, in that order.

5.2 Results and Interpretations

In the following, we present and analyze our evaluation results and answer our research questions. In Table 1, we give an excerpt of the aggregated evaluation results for a selection of our subject systems. For brevity, we do not list the results from all feature models.[3] For each feature model, we list its number of features and constraints, memory consumption of the modal implication graph, and aggregated measurements of our experiments. We show the execution time that each algorithm needs during its offline phase. Additionally, we show the execution time of the online phase when 3%, 10%, and 100% of variant features were defined. The number of defined features includes the features defined by decision propagation. All shown results represent the mean value over the 200 conducted experiments. We also show the mean of all values over all feature models and conducted experiments at the bottom of the table. However, we omit the results of NSAT, as its offline phase is equal to ASAT and its online phase execution time is orders of magnitude larger than all other algorithms (e.g., over 100 times larger compared to ASAT). In the following, we discuss the results in more detail.

Offline Phase (Experiment 1). We display the results of our first experiment in Fig. 3 in the first diagram for most feature models. We excluded the four largest feature models (i.e., Automotive01, Automotive02, FreeBSD, and Linux) from the diagram, as they are visually hard to compare to the other models due to their size. Nevertheless, we state the results for these models in Table 1. Each data point represents the offline time of a particular algorithm and feature model. On the y-axis, we show the execution time in milliseconds and on the x-axis, the number of features in the feature model. Our data reveals that the execution times from the different algorithms differ in orders of magnitude. For instance, for the feature model Automotive01, ASAT required 67 ms, IMIG 696 ms, and CMIG 11,596 ms for the offline phase. In terms of memory consumption, the additional memory required to store a modal implication for IMIG and CMIG lies between 0.25 MB for the feature model FreeBSD and 5.1 MB for Automotive02 with a mean value of 0.9 MB over all 120 feature models.

In Table 2, we show the p-values of the Wilcoxon-Mann-Whitney test for all pairwise combinations of algorithms. In all cases, we received a p-value of less than 10^{-15} and, thus, we can reject our null

[3] A complete table can be found here: https://github.com/skrieter/MIG-Evaluation/blob/master/ICSE2018_Evaluation/execution_times.pdf

Table 1: Offline and online time of evaluated algorithms for a selection of feature models (mean value over 200 experiments).

Feature Model	#Features	#Clauses	MIG Memory (Byte)	Offline time in s (∅)			∑ Online time in s for relative number of defined features (∅)								
							3%			10%			100%		
				ASAT	IMIG	CMIG	ASAT	IMIG	CMIG	ASAT	IMIG	CMIG	ASAT	IMIG	CMIG
FreeBSD 8.0.0	1,397	14,295	243,168	0.04	0.42	6.89	0.21	0.05	0.04	0.44	0.07	0.05	1.82	0.10	0.08
Automotive01	2,513	2,833	1,098,248	0.07	0.70	11.60	1.54	0.36	0.35	3.10	0.56	0.54	6.05	0.76	0.74
Linux 2.6.28.6	6,889	80,715	2,157,320	0.27	16.75	399.98	11.78	5.75	4.24	26.98	8.39	5.63	80.67	10.07	6.66
Automotive02	18,616	1,369	5,088,720	2.30	42.47	296.73	329.29	38.08	37.84	535.70	58.77	58.45	821.48	68.03	67.68
All models (∅)	–	–	–	0.03	0.62	6.99	2.98	0.38	0.36	4.96	0.58	0.55	8.10	0.68	0.64

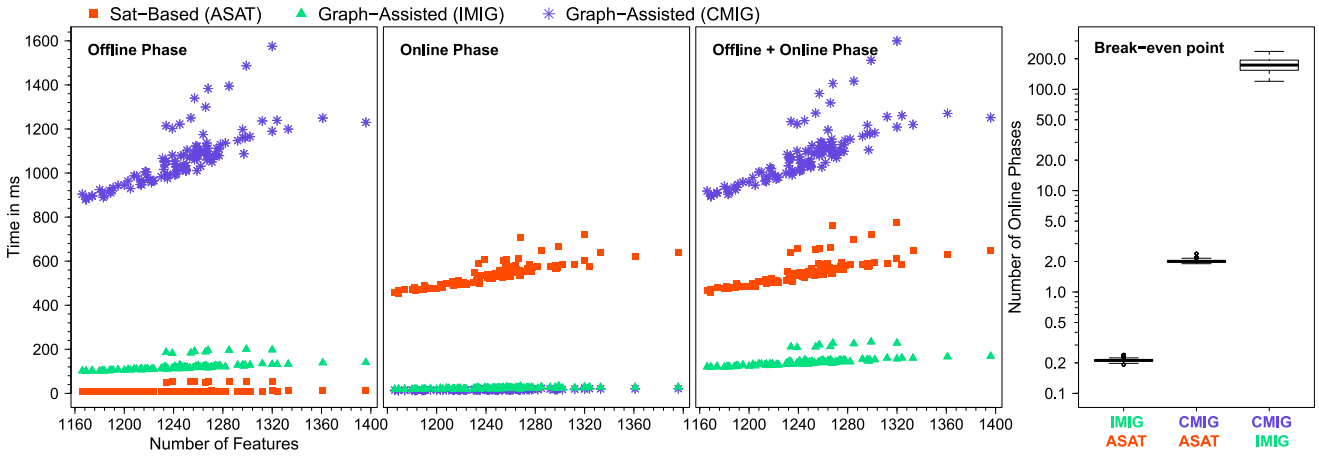

Figure 3: Execution time of offline (1), online (2), and combined offline and online phase (3) of all algorithms for multiple feature models (sorted by size). Break-even point (4) of two algorithms indicates the number of online phases one algorithm's overall execution time becomes faster than another (e.g., CMIG is faster than ASAT for two or more online phases).

Table 2: Pairwise comparison of algorithms.

Attribute	ASAT / IMIG	ASAT / CMIG	IMIG / CMIG
p-value H_0^{RQ1}	$< 10^{-15}$	$< 10^{-15}$	$< 10^{-15}$
p-value H_0^{RQ2}	$< 10^{-15}$	$< 10^{-15}$	$< 10^{-15}$

hypothesis H_0^{RQ1}. For all feature models ASAT needs significantly less time for its offline phase than the two graph-assisted algorithms. Likewise, IMIG needs significantly less time than CMIG.

Therefore, we can answer RQ1: Yes, there is a significant difference in the time required for the offline phase of the different algorithms. These results are expected, as the algorithms' offline phases differ in the amount of precomputations. While ASAT only detects core and dead features, IMIG has to derive an incomplete modal implication graph in addition. Moreover, CMIG does all of the above and also computes implicit strong edges within the modal implication graph to make it complete.

Online Phase (Experiment 2). We depict the aggregated results of our second experiment in Fig. 3 in the second diagram for most feature models. We provide the results for the remaining feature models in Table 1. Analogous to the diagram for the offline time, each data point represents the mean execution time over 200 experiments for a particular algorithm to define 100% of the variant features of one feature model. On the y-axis, we show the execution time in milliseconds and, on the x-axis, the number of features in the feature model. From our data we can see that for every feature model ASAT requires more online time than both graph-assisted algorithms. In contrast, there is no big difference in the time required by both graph-assisted algorithms. Nevertheless, using CMIG indicates slight improvements over IMIG.

To illustrate the results of the second experiment in more detail, we depict the execution time for each individual decision propagation for the feature model of Linux in Fig. 4. Each data point originates from *one* of the 200 conducted experiments and represent the execution time of decision propagation by a particular algorithm. On the y-axis, we depict the time in milliseconds and, on the x-axis, the number of defined variant features before decision propagation was executed. The regression curves indicate the mean execution time over 200 experiments. For ASAT the data points for a particular x-value spread wide around the regression curve. However, most data points lie above the data points from CMIG and IMIG. While CMIG shows slightly better results than IMIG, the difference between both is mostly in the range of a few milliseconds. It is also notable that, for both graph-assisted algorithms, there are many data points that are close to zero.

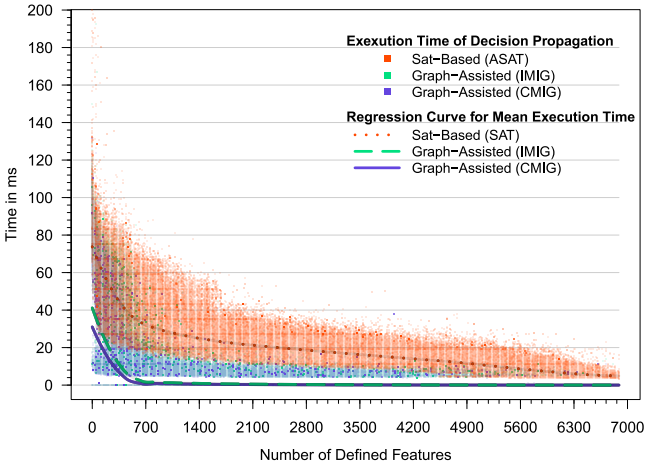

Figure 4: Execution time during online phase with ASAT, IMIG, and CMIG for the feature model of Linux.

We list the results from the Wilcoxon-Mann-Whitney test for Experiment 2 in Table 2. Again, we can reject our null hypothesis H_0^{RQ2} as we received a p-value of less than 10^{-15} for each algorithm combination. ASAT needs significantly more time than IMIG and CMIG, whereas CMIG is faster than IMIG to a significant, but rather small degree.

Therefore, we can answer RQ2: Yes, there is a significant difference in the execution time of decision propagation for the different algorithms. These results correspond to our expectations, as all algorithms can access a different amount of information during their online phase. CMIG traverses a complete modal implication graph, while IMIG uses a graph that might lacks some strong edges. By contrast, ASAT does not use any additional data structure and has to infer all needed feature dependencies on-the-fly. The low performance increase of CMIG over IMIG is due to the small number of implicit strong edges that can be derived in the offline phase.

Comparison of Offline and Online Phase. As the number of configuration processes and changes to the feature model might differ for each configurable system, we are interested in the combined cost of offline and online phase. For a better comparison of execution times for offline and online phase, we present the diagrams for both results side by side in Fig. 3 with both diagrams sharing the same y-axis. Moreover, we visualize the combined cost of offline and online phase in the third diagram of Fig. 3. In this diagram, we add the time needed for the offline phase to the time required to execute the online phase once. Again, we depict the results of the algorithms for all but the largest feature models. We visualize the number of online phases necessary for each algorithm to break even with the other algorithms in the fourth diagram of Fig. 3.

The visualization clearly indicates that IMIG needs less time than ASAT even for just one iteration of the online phase. Regarding CMIG, we see that its higher offline time compared to ASAT is amortized after two iterations of the online phase. In our evaluation, we experienced only two exceptions of this observation for the feature models FreeBSD (three iterations) and Automotive02 (five iterations) (cf. Table 1). As the online time for both graph-assisted approaches only differs slightly, CMIG needs many more online phases in order to amortize its initial costs when compared to IMIG. In detail, we measured between 112 and 801 necessary iterations, with a mean value of 185 over all feature models.

Considering the observations, we made from the evaluation results, we can answer RQ3: In most cases IMIG is superior to both ASAT and CMIG in terms of overall execution time. Only when considering an incomplete online phase (i.e., creating only a partial configuration) ASAT outperforms IMIG due to its efficient offline Phase. On the other hand CMIG outperforms IMIG for a high number of online phases (i.e., 112 in our experiments). Thus, in general IMIG seems to be preferable over the other three algorithms, as it provides a good trade-off between the time required for offline and online phase. ASAT and CMIG are preferable over other algorithms only in some extreme cases. When the feature model evolves more frequently than the configurations, ASAT can be superior. In case that configurations are updated frequently while the feature model does not evolve for a longer period of time, CMIG can be superior as its online phase requires less time than ASAT and IMIG. Regarding memory consumption, in our experiments we found that the memory required to store a modal implication graph was at maximum 5.1 MB, which is relatively small compared to the available main memory on modern hardware. Thus, the additional memory consumption can be neglected for most applications.

5.3 Threats to Validity

Internal. A number of issues might threaten the internal validity of our results. First, bugs in the implementation might cause wrong results. We mitigate this issue by deploying unit tests to test each algorithm individually. Furthermore, we compared the resulting configurations of all algorithms and found no difference during all conducted experiments. Additionally, we use matured open-source tools such as Sat4J to further reduce the possibility of bugs.

Second, the results could be biased in favor of our proposed algorithms. This is due to the fact, that we implemented all evaluated algorithms by ourselves. However, we use the same base implementation for all algorithms and, for each algorithm, we only do the necessary modifications as described in Section 3 and Section 4.

Third, random input data might lead to unrepresentative results. To simulate a configuration process, we used a series of random decisions, which might not correspond to a real-world configuration. However, a randomized approach gave us the capability to efficiently do multiple iterations with distinct random seeds and, thus, gather more data. To avoid random bias, we evaluate each setting in 200 iterations.

External. There are some threats that may affect the generalizability of our results. First, our results might not transfer to real configuration applications. Our simulated configuration process is likely to be different from a manual configuration by a user with domain knowledge. In addition, starting with a partial configurations may mitigate the problem of slow initial decision propagations (cf. Fig.4). However, a manual configuration process strongly depends on the particular user, which could bias the results as well.

Second, the tested feature models might not be representative of feature models used in practice. To mitigate this issue, we tested 120 real-world feature models with a varying number of features and

constraints that have been used in prior studies [17, 75]. Furthermore, we included the largest real-world feature models referenced in literature at the moment.

Third, in our implementation, we only employ Sat4J as SAT solver. However, we use Sat4J as a black-box, such that other solvers (e.g., SAT, CSP, BDD, MDD) could also be plugged-in. As we shift SAT calls from the online to the offline phase, faster solvers should improve both, online and offline computation.

6 RELATED WORK

Interactive Configuration. Many tools already provide decision propagation for an interactive configuration process. Among those tools are GEARS [50, 51], GUIDSL [11], S2T2 Configurator [21], S.P.L.O.T. [60], and VariaMos [58]. Our work is based on SAT-based decision propagation as proposed by Janota [43]. Others proposed to propagate decisions using binary decision diagrams (BDDs) [36, 62] and constraint satisfaction problem (CSP) solvers [6, 15]. BDDs are somewhat similar to our modal implication graphs, as both avoid some effort during decision propagation (i.e., online phase) by more effort for their creation (i.e., offline phase). A problem with BDDs is that they typically do not scale for feature models larger than 1,000 features. For instance, there is no BDD for Linux so far. While CSP solvers can handle constraints beyond boolean formulas as needed for extended feature models [15], they often reduce inputs to satisfiability problems internally. Hence, we have little hope that they could improve performance compared to our graph-assisted algorithm. Similar to CSP, satisfiability modulo theories (SMT) extend the boolean SAT problem to first-order logic [20]. As far as we know, SMT solvers, such as Z3 [30], have not been applied to decision propagation yet. However, as our approach is independent from the actual solver, but instead tries to reduce the number of required SAT queries, we assume that approaches that employ SMT solvers can also benefit from model implication graphs.

In our evaluation, we use configurable systems such as Linux and eCos that provide their own feature modeling language and corresponding configuration tools (i.e., KConfig [90] and CDL [85]). Although, these languages allow for multi-valued logic, they can be translated into boolean feature models [18, 49, 78]. The KConfig language differentiates between *select* and *depends* constraints. In terms of modal implication graphs, select can be considered a strong edge, as it directly implies other features, while depends can be seen as a set of weak edges. In contrast, modal implication graphs do not rely on manual specification of select and depends constraints, but can compute the respective relationships, which avoids mistakes by users and represents feature dependencies more efficiently.

Another technique to avoid contradictions within a configuration is *error resolution*, which automatically detects and tries to resolve conflicts [14, 66, 87]. Configuration tools that support this kind of technique are, for example, pure::variants [19, 72] and FaMa [14]. Contrary to decision propagation, error resolution can be applied at any point during or after the configuration process. To support error resolution, modal implication graphs can be combined with cycle detection algorithms.

Decreasing the configuration time can also be achieved by considering only a subset of system's configuration options. Users might be interested in certain partial configurations or only need to configure a sub-tree of the feature model. An example of this are staged configurations [27] and the configuration of decomposed or sliced feature models [74, 75]. While, in our evaluation, we only focused on complete configuration, modal implication graphs could also speed-up partial configuration processes (cf. Fig. 4).

An interactive configuration process is not limited to ensure valid configurations, but can also provide other useful information. For instance, users can be supported by recommender systems [69], or visual feedback [57, 65]. All techniques that consider feature dependencies can potentially benefit from modal implication graphs, as they provide a fast and complete access to binary feature relations.

Automated Configuration. Besides manual configuration by a user, configurations can also be created automatically by certain algorithms. A use-case for automated configuration is product-based testing, which requires the generation of a representative sample of configurations [2–5, 9, 22–24, 31, 33–35, 37–41, 44–48, 53–56, 67, 68, 70, 71, 73, 76, 79, 80, 82]. Another use case is the product generation via optimization of non-functional properties [15, 66, 77, 86, 88]. Similar to manual configuration, decision propagation can be used in an automated configuration process to query the feature model and derive valid configurations. Thus, modal implication graphs could also be applied in automated configuration processes.

Product-Line Analyses. There exist many analyses for configurable systems that reason about feature-model dependencies [10, 13, 28, 53, 63, 83, 84]. For instance, in our work, we use the analysis of finding core and dead features. Similar to decision propagation, most analyses can be implemented using SAT solvers [13, 84]. Therefore, modal implications graphs could be used to speed up those analyses as well.

7 CONCLUSION AND FUTURE WORK

Decision propagation is a useful method for an interactive configuration process. It prevents users from defining contradictions during the configuration process. However, current implementations do not scale well for large-scale configurable systems. In this work, we introduced modal implication graphs, an extension of implication graphs for feature models to support the application of decision propagation. We presented the concept of modal implication graphs, their derivation from feature models, and how they are employed during decision propagation. Based on our open-source implementation, we evaluated the benefits of using graph-assisted decision propagation for a complete configuration process and reasoned about its trade-offs. Compared to a SAT-based approach, using modal implication graphs can significantly speed up decision propagation and, thus, make its application feasible for large-scale configurable systems.

In future work, we will apply modal implication graphs to applications beyond decision propagation. We are convinced that many existing approaches could profit from modal implication graphs. This includes analyses such as atomic sets, visualization of feature model dependencies, and product-based testing.

Acknowledgments: Thorsten Berger, Alexander Knüppel, Christian Kästner. This work is partially funded by DFG (LE 3382/3-1).

REFERENCES

[1] K. Z. Ahmed, Bestoun S.and Zamli, W. Afzal, and M. Bures. Constrained Interaction Testing: A Systematic Literature Study. *IEEE Access*, 5:25706–25730, 2017.

[2] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 144–155. ACM, 2016.

[3] M. Al-Hajjaji, S. Lity, R. Lachmann, T. Thüm, I. Schaefer, and G. Saake. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. pages 34–40. IEEE Computer Science, 2017.

[4] M. Al-Hajjaji, J. Meinicke, S. Krieter, R. Schröter, T. Thüm, T. Leich, and G. Saake. Tool Demo: Testing Configurable Systems with FeatureIDE. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 173–177. ACM, 2016.

[5] M. Al-Hajjaji, T. Thüm, M. Lochau, J. Meinicke, and G. Saake. Effective Product-Line Testing Using Similarity-Based Product Prioritization. 2018. To appear.

[6] J. Amilhastre, H. Fargier, and P. Marquis. Consistency Restoration and Explanations in Dynamic CSPs-Application to Configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[7] S. Ananieva, M. Kowal, T. Thüm, and I. Schaefer. Implicit Constraints in Partial Feature Models. In *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*, pages 18–27. ACM, 2016.

[8] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation.* Springer, 2013.

[9] P. Arcaini, A. Gargantini, and P. Vavassori. Generating Tests for Detecting Faults in Feature Models. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10. IEEE Computer Science, 2015.

[10] L. Aversano, M. D. Penta, and I. D. Baxter. Handling Preprocessor-Conditioned Declarations. pages 83–92. IEEE Computer Science, 2002.

[11] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005.

[12] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated Analyses of Feature Models: Challenges Ahead. *Communications of the ACM*, 2006.

[13] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010.

[14] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 129–134. Technical Report 2007-01, Lero, 2007.

[15] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 677–682, 2005.

[16] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 7:1–7:8. ACM, 2013.

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

[18] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 73–82. ACM, 2010.

[19] D. Beuche. Modeling and Building Software Product Lines with Pure::Variants. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 255–255, 2012.

[20] A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability*, volume 185. IOS press, 2009.

[21] G. Botterweck, M. Janota, and D. Schneeweiss. A Design of a Configurable Feature Model Configurator. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 165–168. Universität Duisburg-Essen, 2009.

[22] I. D. Carmo Machado, J. D. McGregor, Y. a. C. Cavalcanti, and E. S. De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. 56(10):1183–1199, 2014.

[23] M. B. Cohen, M. B. Dwyer, and J. Shi. Coverage and Adequacy in Software Product Line Testing. pages 53–63. ACM, 2006.

[24] M. B. Cohen, M. B. Dwyer, and J. Shi. Interaction Testing of Highly-Configurable Systems in the Presence of Constraints. pages 129–139. ACM, 2007.

[25] S. Cook. The Complexity of Theorem-Proving Procedures. In *In Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.

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

[27] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[28] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM, 2006.

[29] K. Czarnecki and A. Wąsowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 23–34. IEEE Computer Science, 2007.

[30] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *In Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008.

[31] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans. Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 59:59–59:66. ACM, 2015.

[32] N. Eén and N. Sörensson. An Extensible SAT-Solver. In *In Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518. Springer, 2003.

[33] A. Ensan, E. Bagheri, M. Asadi, D. Gasevic, and Y. Biletskiy. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proceedings of the International Conference on Information Technology:New Generations (ITNG)*, pages 291–298. IEEE Computer Science, 2011.

[34] F. Ensan, E. Bagheri, and D. Gasevic. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 7328, pages 613–628. Springer, 2012.

[35] S. Fischer, R. E. Lopez-Herrejon, R. Ramler, and A. Egyed. A Preliminary Empirical Assessment of Similarity for Combinatorial Interaction Testing of Software Product Lines. pages 15–18. ACM, 2016.

[36] T. Hadzic, S. Subbarayan, R. Jensen, H. Andersen, J. Møller, and H. Hulgaard. Fast Backtrack-Free Product Configuration Using a Precompiled Solution Space Representation. In *In Proceedings of the International Conference on Economic, Technical and Organizational Aspects of Product Configuration Systems*, pages 131–138, 2004.

[37] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, and B. Baudry. Test them all, is it worth it? A ground truth comparison of configuration sampling strategies. *CoRR*, (arXiv:1710.07980), 2017.

[38] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 16:1–16:6. ACM, 2013.

[39] C. Henard, M. Papadakis, and Y. Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In C. Le Goues and S. Yoo, editors, *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer International Publishing, 2014.

[40] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. L. Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering (TSE)*, 40(7):650–670, 2014.

[41] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Multi-Objective Test Generation for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 62–71. ACM, 2013.

[42] A. Hubaux, Y. Xiong, and K. Czarnecki. A User Survey of Configuration Challenges in Linux and eCos. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 149–155, 2012.

[43] M. Janota. Do SAT Solvers Make Good Configurators? In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–195, 2008.

[44] M. F. Johansen, Ø. Haugen, and F. Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. pages 638–652. Springer, 2011.

[45] M. F. Johansen, Ø. Haugen, and F. Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55. ACM, 2012.

[46] M. F. Johansen, Ø. Haugen, F. Fleurey, A. G. Eldegard, and T. Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. pages 269–284. Springer, 2012.

[47] C. H. P. Kim, D. Batory, and S. Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 57—68. ACM, 2011.

[48] C. H. P. Kim, E. Bodden, D. Batory, and S. Khurshid. Reducing Configurations to Monitor in a Software Product Line. pages 285–299. Springer, 2010.

[49] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, and I. Schaefer. Is There a Mismatch Between Real-World Feature Models and Product-Line Research? In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 291–302. ACM, 2017.

[50] C. Krueger. BigLever Software Gears and the 3-tiered SPL Methodology. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 844–845, 2007.

[51] C. Krueger and P. Clements. Systems and Software Product Line Engineering with BigLever Software Gears. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 136–140, 2013.

[52] D. Le Berre and A. Parrain. The Sat4j Library, Release 2.2, System Description. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010.

[53] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 81–91. ACM, 2013.

[54] M. Lochau, I. Schaefer, J. Kamischke, and S. Lity. Incremental Model-Based Testing of Delta-oriented Software Product Lines. pages 67–82. Springer, 2012.

[55] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. Comparative Analysis of Classical Multi-Objective Evolutionary Algorithms and Seeding Strategies for Pairwise Testing of Software Product Lines. pages 387–396. IEEE Computer Science, 2014.

[56] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu. Practical Pairwise Testing for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 227–235. ACM, 2013.

[57] J. L. Martinez, T. Ziadi, R. Mazo, T. F. Bissyandé, J. Klein, and Y. Le Traon. Feature Relations Graphs: A Visualisation Paradigm for Feature Constraints in Software Product Lines. In *Conference on Software Visualization (VISSOFT)*, pages 50–59. IEEE, 2014.

[58] R. Mazo, C. Salinesi, and D. Diaz. VariaMos: A Tool for Product Line Driven Systems Engineering with a Constraint Based Approach. In *In Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 147–154. CEUR-WS.org, 2012.

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

[60] M. Mendonça, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 761–762. ACM, 2009.

[61] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-Based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240, 2009.

[62] M. Mendonça, A. Wąsowski, K. Czarnecki, and D. Cowan. Efficient Compilation Techniques for Large Scale Feature Models. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 13–22. ACM, 2008.

[63] A. Metzger, K. Pohl, P. Heymans, P.-Y. Schobbens, and G. Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Requirements Engineering*, pages 243–253. IEEE Computer Science, 2007.

[64] S. Nadi, T. Berger, C. KÃd'stner, and K. Czarnecki. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering (TSE)*, 41(8):820–841, 2015.

[65] D. Nestor, S. Thiel, G. Botterweck, C. Cawley, and P. Healy. Applying Visualisation Techniques in Software Product Lines. In *In Proceedings of the Symposium on Software Visualization (SoftVis)*, pages 175–184. ACM, 2008.

[66] L. Ochoa, O. González-Rojas, and T. Thüm. Using Decision Rules for Solving Conflicts in Extended Feature Models. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, pages 149–160. ACM, 2015.

[67] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 196–210. Springer, 2010.

[68] S. Oster, M. Zink, M. Lochau, and M. Grechanik. Pairwise Feature-interaction Testing for SPLs: Potentials and Limitations. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 6:1–6:8. ACM, 2011.

[69] J. A. Pereira, P. Matuszyk, S. Krieter, M. Spiliopoulou, and G. Saake. A Feature-Based Personalized Recommender System for Product-Line Configuration. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 120–131, 2016.

[70] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journa l (SQJ)*, 20(3-4):605–643, 2012.

[71] G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE Computer Science, 2010.

[72] pure::systems. pure::variants. Website, 2017. Available online at http://www.pure-systems.com/products/pure-variants-9.html; visited on May 10th, 2017.

[73] D. Reuling, J. Bürdek, S. Rotärmel, M. Lochau, and U. Kelter. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 131–140. ACM, 2015.

[74] J. Schroeter, M. Lochau, and T. Winkelmann. Multi-Perspectives on Feature Models. pages 252–268. Springer, 2012.

[75] R. Schröter, S. Krieter, T. Thüm, F. Benduhn, and G. Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 667–678. ACM, 2016.

[76] J. Shi, M. B. Cohen, and M. B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 270–284. Springer, 2012.

[77] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. SPL Conqueror: Toward Optimization of Non-Functional Properties in Software Product Lines. *Software Quality Journal*, 20(3-4):487–517, 2012.

[78] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is the Linux Kernel a Software Product Line? In *Proceedings of the International Workshop on Open Source Software and Product Lines (OSSPL)*, pages 9–12. IEEE Computer Science, 2007.

[79] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. pages 421–432. USENIX Association, 2014.

[80] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, 2012.

[81] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem. In *In Proceedings of the European Conference on Computer Systems*, pages 47–60, 2011.

[82] M. H. ter Beek, H. Muccini, and P. Pelliccione. Assume-Guarantee Testing of Evolving Software Product Line Architectures. pages 91–105. Springer, 2012.

[83] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 95–104. ACM, 2007.

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

[85] B. Veer and J. Dallaway. The eCos Component Writer's Guide. Manual, 2017. Available online at http://ecos.sourceware.org/ecos/docs-3.0/pdf/ecos-3.0-cdl-guide-a4.pdf; visited on May 10th, 2017.

[86] J. White, B. Dougherty, and D. C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software (JSS)*, 82(8):1268–1284, 2009.

[87] J. White, D. C. Schmidt, D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 225–234. IEEE Computer Science, 2008.

[88] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating Product-Line Variant Selection for Mobile Devices. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 129–140, 2007.

[89] H. Ye and H. Liu. Approach to Modelling Feature Variability and Dependencies in Software Product Lines. *IEE Proceedings - Software*, 152(3):101–109, 2005.

[90] R. Zippel. KConfig Documentation. Website, 2017. Available online at http://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt; visited on May 10th, 2017.