Efficient Synthesis of Feature Models

Nele Andersen[†*] Krzysztof Czarnecki[‡] Steven She[‡] Andrzej Wąsowski[†]

[†]IT University of Copenhagen, Denmark
nele.andersen@gmail.com,
wasowski@itu.dk

[‡]Generative Software Development Lab
University of Waterloo, Canada
{kczarnec,shshe}@gsd.uwaterloo.ca

ABSTRACT

Variability modeling, and in particular feature modeling, is a central element of model-driven software product line architectures. Such architectures often emerge from legacy code, but, unfortunately creating feature models from large, legacy systems is a long and arduous task.

We address the problem of automatic synthesis of feature models from propositional constraints. We show that this problem is NP-hard. We design efficient techniques for synthesis of models from respectively CNF and DNF formulas, showing a 10- to 1000-fold performance improvement over known techniques for realistic benchmarks.

Our algorithms are the first known techniques that are efficient enough to be applied to dependencies extracted from real systems, opening new possibilities of creating reverse engineering and model management tools for variability models. We discuss several such scenarios in the paper.

1. INTRODUCTION

Variability models are central to development and management of software product lines (SPL). They comprise simple *problem space models* and usually quite complex *solution space models*. A problem space model describes major decisions made during customization—such as whether an Enterprise Resource Planning (ERP) system should include an e-commerce platform or not. The solution space model explains how the problem space decisions affect the realization—for example, how the e-commerce platform is woven into the implementation, by extending data models, user interfaces and services.

Variability models contain concepts referred to as decisions [36], features [27] or variation points [21], depending on the abstraction level. The abstract models tend to contain relatively few concepts (up to hundreds in the largest models[1]), while the low level concrete models can reach thousands of variation points. These concepts are typically organized hierarchically, and related to each other using constraints.

There exist multiple commercial (Pure Systems GmbH, Big Lever Software Inc.) and research [13, 23, 22, 36] tools for variability modeling. Recognizing the increasing significance of this market segment, The Object Management Group (OMG) has initiated [34] a standardization process for the Common Variability Language (CVL).

Feature models [27, 14] are one of the prominent notations used in variability modeling. Applications of feature modeling include automatic generation of product configurators, driving code generators [14] and build systems [6] to compose individual members of an SPL, and driving test and verification [29, 11]. Feature models will also be part of the CVL standard [34]. In this paper we use the term feature in the abstract unifying sense, meaning either a decision or a variation point. This simplification is justified, since we will be exploiting the combinatorial structure of features, which is similar in the solution space and in the problem space.

SPLs are typically large software projects, often resulting from a long lasting evolution, based on substantial legacy code. Industrial SPLs employ models containing thousands of features, especially if they mix the problem and the solution space. For instance, the Linux kernel project uses a model containing in excess of 5000 features to describe its x86 architecture [6]. At the same time, there exist SPLs, such as the FreeBSD kernel, that could benefit from having feature models, but presently no such models exist for them. Communications with Pure Systems indicate that similar models and situations are met in the industry.

Reverse engineering techniques for variability models, would ease adoption of product line practices, enabling more smooth migration of legacy code to systematic product line architectures and their subsequent evolution. This paper addresses the problem of synthesis of feature models, which is the core algorithmic part of reverse engineering: to synthesize a feature model from a given set of dependencies. We construct diagrams that contain a hierarchy of groups of binary features enriched by cross-hierarchy inclusion/exclusion constraints. Our algorithms assume a constraint system expressed in propositional logics as input. In practice, these constraints can be either specified by engineers, or automatically mined from the source code using static analysis [5]. Furthermore, effective management of large feature models requires model management operations such as merge, compare, diff, and project [1]. Such operations ease model evolution by allowing developers to compare models to assess the impact of model edits and build large models by composing smaller ones. The

[*]Presently at Configit A/S, Copenhagen

[1]Personal communications with Big Lever and Pure Systems

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

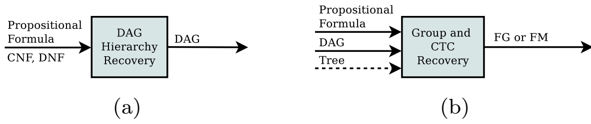


Figure 1: Components of FM synthesis

feature model synthesis problem is also at the core of several such model operations, which are defined via logical operators on formulas derived from the input models [1].

In this paper we formally define the problem of synthesis of feature models, discuss its complexity, derive semantic based algorithms and argue for their correctness. Technically, we synthesize not a feature model, but a *feature graph*, which is a symbolic representation of all possible feature models that could be sound results of the synthesis. Then we show that any of these models can be efficiently derived from the feature graph. Our contributions include:

- Definition of feature model synthesis as an algorithmic problem, an NP-hardness result, and a complexity driven analysis of suitable solution techniques.
- An algorithm for synthesis of feature models from *conjunctive normal form* (CNF) formulas, least 10-times faster than previously known algorithms.
- An efficient algorithm for synthesis of feature models from *disjunctive normal form* (DNF) formulas.
- An implementation and an evaluation of the above.

The above techniques produce feature models, but can be easily adjusted to other languages, such as the propositional part of variability specifications of the current CVL proposal. Importantly, the algorithm for synthesis of models from a constraint in CNF form, is the first known technique for this problem, which can be applied to data extracted from real systems. The previous work of the same authors [16] has shed light on the mathematical structure of the problem, but has failed to provide scalable algorithms.

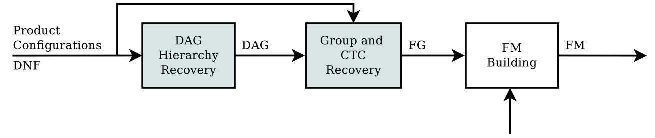
2. OVERVIEW AND MOTIVATION

Feature model synthesis takes as input a formula representing a set of feature dependencies or product configurations and outputs a feature graph (FG) or feature model (FM). We separate FM synthesis into two reusable steps (Fig. 1): (a) DAG hierarchy recovery—which reconstructs the hierarchy of the diagram, possibly with multiple parents for a feature, and (b) group and cross-tree constraint (CTC) recovery—which identifies feature groups and additional constraints that can not be represented by the hierarchy.

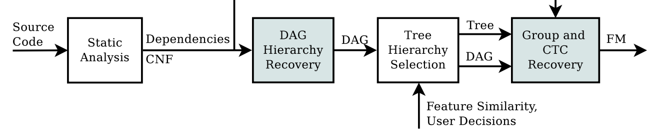
The first step, DAG hierarchy recovery, takes the input formula in either CNF or DNF, and produces a DAG that contains all possible FM tree hierarchies.

The second step, identifies all feature groups and CTCs given the propositional formula, DAG and an optional tree hierarchy. This step outputs a FM or a FG depending on whether a tree hierarchy is provided as input or is not.

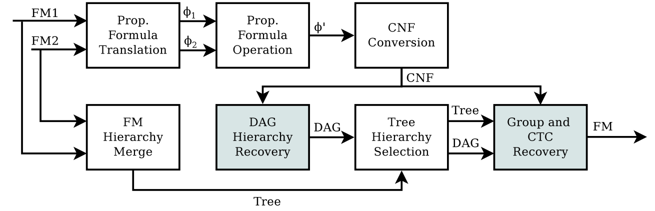
These two steps can be used in a variety of scenarios. The remainder of this section describes reverse engineering from product configurations or code and operations on feature models as examples (Fig. 2). The contribution of this paper is to provide efficient algorithms for these two steps.



Scenario 1. FM synthesis from product configurations—*late tree hierarchy selection.*



Scenario 2. Tool-assisted FM reverse engineering—*early tree hierarchy selection.*



Scenario 3. Binary FM merge operation—*early tree hierarchy selection.*

Figure 2: FM synthesis scenarios

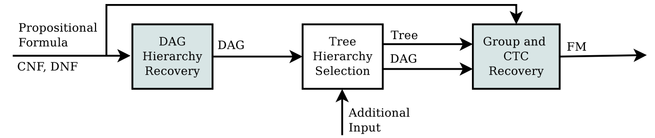


Figure 3: Abstract workflow for FM synthesis with early hierarchy selection

Scenario 1. This scenario describes the process of synthesizing a FG from a set of product configurations. Here, the product configurations are represented as a formula in DNF. In this conversion, a product is represented as a conjunction of positive literals representing features present in the product and negative literals representing features absent from the product; a set of products is then a disjunction of the conjunctions representing the individual products. The two synthesis steps are executed consecutively in this scenario yielding a feature graph. The final FM is built using an interactive FM building tool that uses the FG as a guide [26]. This scenario is an example of FM synthesis where the FM hierarchy is decided at a later time, after the FG is constructed.

Scenario 2. This scenario describes reverse engineering a FM from code. Variability rich software, such as the FreeBSD kernel, can benefit from having a FM. The FreeBSD operating system kernel is configured in the build system to derive variations of the kernel functionality. Unlike the Linux kernel [6], the FreeBSD kernel does not have a FM to make configuration of variants easier for users and management of variability easier for developers.

The dependencies among features can be extracted from source code using static analysis yielding a formula in CNF. This scenario differs from the first by introducing an intermediate step for deriving a tree hierarchy. In this scenario,

the tree is built by a user supported by a tool using a feature similarity measure operating on the DAG [39]. This paper describes a concrete realization of this scenario by reverse engineering dependencies from build systems [5] and code with conditional compilation directives, that is then used to reverse engineer an FM for FreeBSD. The work presented here allowed for the reverse engineering approach to scale to large FMs, with several thousands of features.

Scenario 3. Our third scenario describes binary operations on two FMs [1]. Examples of operations include merging, diffing, comparing, and slicing feature models. The two input models, FM_1 and FM_2 are first translated to their propositional formulas [3], then an operation is applied to merge the two models resulting in a single formula. This formula is converted to CNF then inputted into FM synthesis. In this scenario, a tree hierarchy is derived automatically based on merge heuristics applied to the tree hierarchies of the input FMs. Acher's FM management infrastructure [1] implements the operations using our previous BDD-based FM synthesis solution [16], which does not scale beyond small FMs, with few dozens of features. The algorithms presented here can be used to improve the scalability of that infrastructure.

Abstract workflow. In general, some form of additional input is required in order to derive a tree hierarchy from the DAG. In *Scenario 1*, the additional input came in the form of user decisions supported by an interactive FM building tool. *Scenario 2*, shifts tree hierarchy selection to before the group and CTC recovery steps and uses user decisions supported by a feature similarity measure to derive a tree. *Scenario 3* uses FM merge heuristics. *Scenarios 2* and *3* generalize to the workflow in Fig. 3. Since the workflow in *Scenario 1* builds all FMs that can be built from a formula, the workflow in Fig. 3 can be seen as a special, easier case that prunes alternative hierarchies from the DAG using a given tree hierarchy and thus builds only a single FM. Since our focus is on efficient algorithms, we will present only the computationally harder workflow from *Scenario 1*; the easier scenario is easily derivable from the presented one.

Methodology. We first analyze computational complexity of the individual steps in the synthesis of feature models, and of variations of the problems for different input representations. The complexity analysis allows us to decide what the promising reductions of the problem are; for example using SAT-based techniques for synthesis of or-groups from DNF formulae, and not using these techniques for CNF formulae. We exploit this in the design of algorithms, which are then implemented and evaluated experimentally.

3. BACKGROUND

We begin the technical development with basic terminology on propositional logics [9]. A clause is a disjunction of literals. A term is a conjunction of literals. Syntactically, clauses and terms are sets of literals. A clause C *subsumes* a clause C' iff $C \subseteq C'$. A propositional formula is in conjunctive normal form (CNF) iff it is a conjunction of clauses; in disjunctive normal form (DNF) iff it is a disjunction of terms.

An *implicate* D of a propositional formula φ is a clause such that (i) D is not a tautology and (ii) $\varphi \rightarrow D$ is a tautology. D is *prime* iff it is minimal: no literals can be removed from it without violating (ii). An *implicant* C of a

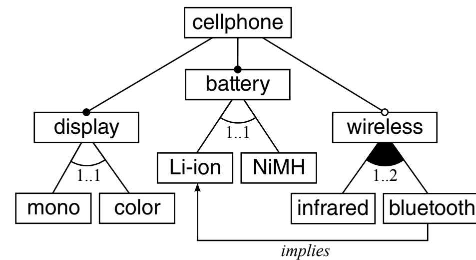


Figure 4: An example feature model

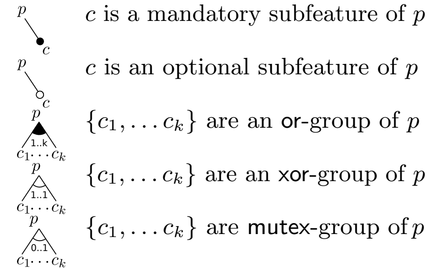


Figure 5: Concrete syntax of feature diagrams

propositional formula φ is a term such that C is consistent and $C \rightarrow \varphi$ is a tautology. C is a *prime* if it is minimal. A formula φ is *rooted* if it has at least one variable r such that for any other variable f: $\varphi \rightarrow (f \rightarrow r)$ is valid (in other words $\varphi \wedge f \rightarrow r$ is valid). We write $\varphi[x \mapsto 1]$ (resp. $\varphi[x \mapsto 0]$) meaning a formula created from φ by substituting all occurrences of variable x by the constant 1 (respectively 0). We lift this to sets of variables writing $\varphi[x \mapsto 0]_{x \in X}$.

We now switch to defining feature diagrams and feature models. Our definition largely follows the syntax of FODA [27].

DEF. 1. *A* feature diagram *is a tuple* $\mathsf{FD}(F, E, (E_m, E_i, E_x), (G_o, G_x, G_m))$*, where F is a finite set of features, $E \subseteq F \times F$ is a set of directed child-parent edges; $E_m \subseteq E$ is a set of mandatory edges; $E_i \subseteq F \times F$ is a set of cross-tree* implies *edges, where $E_i \cap E = \emptyset$; $E_x \subseteq 2^F$ and for each $e \in E_x$, $|e| = 2$, is a set of cross-tree* excludes *edges; sets G_o, G_x, G_m contain non-overlapping subsets of E, participating in* or*-groups,* xor*-group and* mutex*-groups respectively: each member subset in any of G_o, G_x and G_m is disjoint from any other subset being a member of these sets.*

The following well-formedness constraints hold in FD*:*

i. (F, E) *is a rooted tree connecting all features in* F

ii. All edges in a group share the same parent, so if $g \in G_i$ *for* $i \in \{o,x,m\}$ *and if* $(f_1, f_2), (f_3, f_4) \in g$ *then* $f_2 = f_4$

iii. Sets E, E_i, *and* E_x *are pairwise disjoint.*

A feature model FM *is a pair* (FD, φ) *where* FD *is a feature diagram, and* φ *is a propositional constraint over the features of* FD*—a* cross-tree constraint*.*

Fig. 4 presents a feature model of a simple family of cellphones (inspired by an example available at fm.gsdlab.org). The root of the tree, or the *root feature*, represents the product family itself (cellphone). The remaining nodes represent mandatory or optional features of the products in the family. Display and battery are mandatory subfeatures of the root,

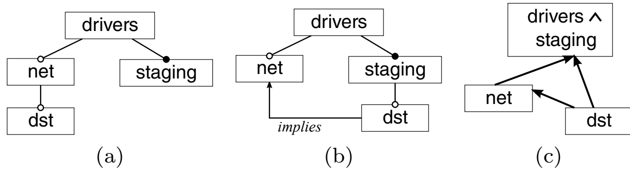


Figure 6: Two diagrams (a-b) and a feature graph (c), same configuration semantics

so in the abstract syntax of Def. 1, (display,cellphone) and (battery,cellphone) are members of E_m, whereas wireless is an optional child of root, so (wireless, root) $\in E \setminus E_\mathrm{m}$. Children of display form an xor-group, meaning that display can be either mono (i.e. monochrome) or color, but not both. In terms of abstract syntax {(mono, display), (color, display)} $\in G_\mathrm{x}$. The children of wireless form an or-group, meaning that a cellphone with local wireless support should include at least infrared or bluetooth communication, and possibly both. Finally, the diagram contains an implies edge, (bluetooth, Li-ion) $\in E_\mathrm{i}$, meaning that the bluetooth feature is only provided on the phone with Li-ion batteries.

The *configuration semantics* $[\![\mathsf{FD}, \varphi]\!]$ describes legal combinations of features in the products described by the model [3]:

$$
\begin{aligned}
&[\![(F, E,(E_\mathrm{m}, E_\mathrm{i}, E_\mathrm{x}), (G_\mathrm{o}, G_\mathrm{x}, G_\mathrm{m})), \varphi]\!] = \\
&\quad [\bigwedge_{(c,p)\in E\cup E_\mathrm{i}} c \to p] \wedge [\bigwedge_{(c,p)\in E_\mathrm{m}} p \to c] \wedge [\bigwedge_{(c,p)\in E_\mathrm{x}} c \to \neg p] \wedge \\
&\quad [\bigwedge_{\{(c_1,p),\dots,(c_k,p)\}\in G_\mathrm{o}\cup G_\mathrm{x}} p \to (c_1 \vee \cdots \vee c_k)] \wedge \\
&\quad [\bigwedge_{\substack{\{(c_1,p),\dots,(c_k,p)\}\\ \in G_\mathrm{m}\cup G_\mathrm{x}}} \bigwedge_{\substack{i,j=1..k\\ i\neq j}} c_i \to \neg c_j] \wedge \varphi
\end{aligned} \tag{1}
$$

This semantics does not subsume the entire meaning of feature diagrams. Different diagrams can have the same configuration semantics (see Fig. 6a-b). Other semantic aspects include structural dependencies between features, or conceptual proximity of features. Here we focus on the configuration semantics as the most central aspect of the models.

4. GENERIC SYNTHESIS ALGORITHM

The Problem. Our objective is to take an implicit description of the configuration semantics and synthesize a feature diagram out of it. Since FODA feature diagrams, as defined above, are not logically complete [37], for every formula φ there may not exist a diagram D such that $[\![D]\!] \equiv \varphi$. Instead, we seek a diagram that is *weaker* than φ and, accompanied by some cross-tree constraint ψ, coincides with φ: $[\![D, \psi]\!] \equiv \varphi$.

To enforce creation of interesting diagrams, we require that D is maximal, so that its hierarchy is connecting all the features, that no more cross-tree edges can be added, and that no group definition can be strengthened (no mutex- or or-group can become an xor-group). This way as much information as possible is represented in the diagram itself, without resorting to the cross-tree constraint (otherwise an empty diagram D and φ itself is a trivial answer to every instance of the problem).

Def. 2. *The feature model synthesis problem (FMS) is given a consistent rooted formula φ over a set of features F, synthesize a diagram D over F, such that $\varphi \to [\![D]\!]$ and D is maximal such, i.e. (i) no element can be added to the collections of mandatory edges (E_m), implies edges (E_i), excludes edges (E_x), and or-, xor-, and mutex- groups (G_o, G_x, G_m) without violating the above implication, (ii) no group can be moved from $G_\mathrm{o} \cup G_\mathrm{m}$ to G_x without violating the above implication.*

Recall that, by Def. 1, the above also implies that the diagram D must connect all the features in F.

Thm. 1. *The decision version of FMS is NP-hard.*[2]

In practice, the requirement that φ is rooted is not a limitation. This often follows from the way φ was obtained. Otherwise, a fresh variable r can always be added to φ with necessary implications to make it rooted. Consistency of φ can be checked using a SAT solver. An inconsistency normally indicates an error in software dependencies, which should be fixed before synthesizing a feature diagram.

Representing Many Diagrams Symbolically. As shown in Fig. 6a-b, there may be more than one solution to an FMS instance. The parts (a) and (b) show syntactically different diagrams that are equivalent in the sense of formula (1); both corresponding to the following input formula:

$$
\begin{aligned}
\varphi \equiv (\mathsf{net} \to \mathsf{drivers}) \wedge (\mathsf{dst} \to \mathsf{net}) \wedge \\
\wedge (\mathsf{staging} \to \mathsf{drivers}) \wedge (\mathsf{drivers} \to \mathsf{staging})
\end{aligned} \tag{2}
$$

Our algorithm synthesizes a diagrammatic representation of all possible feature diagrams that are compatible with the input constraints, delegating resolving the tree hierarchy to various usage scenarios as described in Sect. 2. This diagrammatic representation is known as a feature graph [16]:

Def. 3. *A tuple $\mathsf{FG}(F, E, E_\mathrm{x}, (G_\mathrm{o}, G_\mathrm{x}, G_\mathrm{m}))$ is a* feature graph *iff F is a set of features, $E \subseteq F \times F$ is a set of directed child-parent edges; $E_\mathrm{x} \subseteq 2^F$ is a set of undirected* excludes *edges, for each $e \in E_\mathrm{x}$, $|e| = 2$; sets G_o, G_x, G_m contain subsets of E, participating in or-groups, xor-group and mutex-groups respectively. The following constraints hold in FG:*

i. (F, E) is a connected DAG

ii. All edges in a group share the same parent, so if $g \in G_i$ for $i \in \{\mathrm{o,x,m}\}$ and $(f_1, f_2), (f_3, f_4) \in g$ then $f_2 = f_4$,

iii. E, E_x are disjoint (no implies edge is an exclude edge).

Fig. 6c shows a feature graph embedding the feature diagrams of (a) and (b). Feature graphs do not distinguish features with mandatory relationships (here drivers and staging), because the configuration semantics does not distinguish them. This is why in Fig. 6c there is a node labelled with the conjunction of the two: drivers ∧ staging. Such sets of always co-occurring features are sometimes called and-groups. To preserve information about and-groups, we use sets of features as nodes in the algorithms below (so in practice F in the above definition is a power-set, where each node is an equivalence class with respect to φ).

A feature graph is essentially a feature diagram in which some conditions have been relaxed: sharing is allowed (it is

[2] Theorems are available in the appendix, to be accessed at the discretion of referees at http://www.itu.dk/~wasowski/doc-appendix.pdf. The appendix does not extend the paper but merely provides evidence for the interested reader.

a DAG, not a tree), and feature groups can overlap. Feature graphs do not have implication edges as they are part of the hierarchy now that sharing is allowed.

For a given formula a feature graph is potentially not unique, but two special cases are unique: the transitively reduced graph and the transitively closed graph. In this paper we work with transitively closed graphs, like the one in Fig. 6c (the child-parent relation E is transitively closed). One can extract any maximal feature diagram from a transitively closed feature graph FG in polynomial time. This is achieved by the following steps:

1. Find a spanning tree over FG and move all the edges not in the spanning tree to cross-tree implications (E_i);
2. Select greedily non-overlapping subsets from G_o, G_x, and G_m to form syntactically correct groups;
3. Choose one element from each equivalence class of features, and create mandatory edges (E_m) from it to all other members of the class;
4. Remove from E_x all edges that participate in selected mutex- or xor-groups.

So the actual algorithmic hardness of synthesizing a maximal feature diagram can be addressed by synthesizing a feature graph and then applying the above linear time procedure (the latter step is very fast in practice). Using a feature graph has one more advantage: since a suitably constructed feature graph encompasses all possible solutions to an FMS instance, one can optimize against additional (extra-logical) objectives to select a useful diagram out of many possible, as described in Sect. 2.

Extracting The Feature Graph. Figure 7 presents the *generic* algorithm for retrieving a feature graph from a propositional formula (FGE). It follows the design proposed in [16], extended with mutex-groups and excludes edges. Our contribution is to show how FGE can be implemented very efficiently for input represented in CNF and DNF. The implementation in [16] could not scale beyond couple dozens of features.

Mendonca [33, 32] has shown that large feature models can be analyzed efficiently using a SAT solver. The analysis is usually feasible, because even though complex constraints are present, the hierarchical tree constraint imposed over all features by the diagram, significantly simplifies the task of a SAT-solver. This hints that exploring a SAT-based algorithms for synthesis may be beneficial. The difficulty however, lies in the fact that not all the queries used in the algorithm can be directly answered by a SAT solver, thus we have to resort to additional techniques. The final outcome demonstrates a dramatic performance improvement, making the FMS problem practically tractable. Below, we briefly summarize the main steps of the generic algorithm, while we will address the details of CNF and DNF oriented implementations in the upcoming sections.

Let us walk through the steps of FGE presented in Fig. 7. The algorithm takes two parameters: a formula φ and its root r. We begin by detecting dead features in φ (lines 1–2). A feature is *dead* if it is not present in any configuration. FGE produces a feature graph containing only live features; dead features are either irrelevant, or they manifest errors. The binary implication graph G is computed next where live features are vertices in G and an edge (u, v) exists whenever φ entails $u \to v$. See lines 3–4.

FEATURE-GRAPH-EXTRACTION
(φ : formula over F rooted in r, $r \in F$)

▷ Find and remove all dead features
1 $D = \{f \in F \mid \varphi \wedge r \to \neg f\}$
2 $\varphi = \varphi[d \mapsto 0]_{d \in D}$

▷ Compute the implication graph $G(V, E)$
3 $V = F \setminus D$
4 $E = \{(u, v) \in V \times V \mid \varphi \wedge u \to v\}$

▷ Compute strongly connected components
5 $V' = \{S \subseteq V \mid S \text{ is a SCC of } G\}$

▷ Make edges between SCCs creating a DAG
6 $E' = \{(u, v) \in V' \times V' \mid u \neq v$ and
7 $\exists u' \in u, v' \in v.\ (u', v') \in E\}$

▷ Compute the mutex graph $M(V, E_\mathrm{x})$
8 $E_\mathrm{x} = \{\{u, v\} \subseteq V' \mid \exists u' \in u, v' \in v.\, \varphi \wedge u' \to \neg v'\}$

▷ Compute mutex-groups
9 $G_\mathrm{m} = \{\{(f_1, p), \ldots, (f_k, p)\} \mid \{f_1, \ldots, f_k\}$ is
10 a maximal clique in M and $\forall f_i.\, (f_i, p) \in E'\}$

▷ Compute or-groups
11 $G_\mathrm{o} = \{\{(f_1, p), \ldots, (f_k, p)\} \mid f'_1 \vee \cdots \vee f'_k$ is
12 a prime implicate of $\varphi \wedge p'$ and
13 $p' \in p$ and $\forall f_i.\ f'_i \in f_i \wedge (f_i, p) \in E'\}$

▷ Compute xor-groups
14 $G_\mathrm{x} = \{\{(f_1, p), \ldots, (f_k, p)\} \in G_\mathrm{o} \mid \forall\, i \neq j.\ (f_i, f_j) \in E_\mathrm{x}\}$

15 **return** $\mathsf{FG}(V', E', E_\mathrm{x}, (G_\mathrm{o} \setminus G_\mathrm{x}, G_\mathrm{x}, G_\mathrm{m} \setminus G_\mathrm{x}))$

Figure 7: The generic algorithm, mostly after [16]

And-groups—features that always co-occur—are identified as the strongly connected components (SCCs) in the implication graph G. We lift the implication graph to its SCCs: vertices V' are sets of co-occurring features (line 5). There is an edge (u, v) in E' between two and-groups iff there exists an implication edge between any member of u and any member of v in the implication graph G (lines 6–7). The resulting graph (V', E') is a DAG rooted in a vertex containing r. Since r is the root feature, every feature co-occurring with r is in the same and-group, and if there are two or more roots in φ, then they would also belong in the same and-group (a rooted formula can have more than one root, according to the definition given in Sect. 3; r is one of these roots).

The mutex graph M is an undirected graph where the vertices are and-groups. An edge exists between u, v iff φ entails a mutual exclusion, $u \to \neg v$ (line 9). The edges of the mutex graph become the excludes edges of the resulting feature graph. Mutex-groups are computed by finding all maximal cliques [8] in M. A mutex-group is created for each clique and common ancestor p (line 8). Or-groups are computed by identifying prime implicates among variables and their common ancestor (line 11). Finally, to find xor-groups either check for each or-groups if its children are mutually exclusive (line 14), or for each mutex-group if disjunction of its members is implied by parent (more efficient).

5. COMPLEXITY DISCUSSION FOR FGE

Before we move on to describing how FGE can be implemented for CNF and DNF inputs, we want to clarify, which are the hard steps in the FGE algorithm. Observe that all steps except computing or-groups reduce to establishing en-

tailment of binary implications between literals, and it is well known that this can be often efficiently done using a SAT-solver (at least if the input is a CNF formula [32]).

It remains to discuss the complexity of the most difficult step in the algorithm—the computation of prime implicates in lines 11–13. Observe that if π is a prime implicate of φ, then $\neg\pi$ is a prime implicant of $\neg\varphi$. So the prime implicate problem for CNF is as hard as the prime implicant problem for DNF, and, dually, the prime implicant problem with CNF is equi-difficult with the prime implicate problem of DNF. We now define decision versions of these problems:

DEF. 4. CNF-Shortest-Implicant *Problem: given a formula φ in CNF and an integer k, is there an implicant of φ that contains k or fewer literals?*

DNF-Shortest-implicant*: given a DNF formula φ and an integer k, is there an implicant of φ of at most k literals?*

THM. 2. *The* DNF-Shortest-Implicant *problem is coNP-hard, so it is not in NP unless NP = coNP.*

Since NP = coNP is an important and long outstanding open problem, it is unlikely that a SAT solver, an efficient solving technique for NP-complete problems, can be used as the main part of the solution for the problem of finding prime implicants of a DNF formula, or equivalently computing prime implicates of a CNF. Consequently, we will seek other techniques for finding or-groups in a CNF formula.

THM. 3. CNF-Shortest-Implicant *problem is NP-complete.*

Since CNF-shortest-implicant is just as hard as the satisfiability problem, there exists a polynomial reduction between SAT and CNF-Shortest-Implicant. Likely finding prime implicants can be realized by solving SAT, and thus, given efficiency of current SAT solvers, it will likely be beneficial to use them to find or-groups within a DNF formula (or-groups are prime implicates; prime implicates of a DNF formula, are prime implicants of its negation, a CNF formula).

6. SYNTHESIS WITH FGE-CNF

Even though synthesis of or-groups is harder for CNF than for DNF, studying algorithms assuming CNF on input remains relevant. As sketched in Sect. 2, applications of FGE-CNF include synthesizing feature diagrams from declarative constraints specified by engineers and reverse engineering a model from existing code artefacts (*Scenario 2*). In the latter case it is normally natural to generate CNF representation of dependencies. Similarly, since semantics of a feature diagram is also expressed as a CNF formula (equation (1)) one can use FGE-CNF to reason about existing diagrams (*Scenario 3*). CNF clauses can be reinterpreted as implications from conjunctions to disjunctions of literals, naturally expressing properties like x *requires* y, or x *excludes* y. Furthermore, clauses are very naturally combined using conjunction.

Let the version of FGE, assuming CNF input be called FGE-CNF. The structure of FGE-CNF is the same as of FGE (Fig. 7). We detail how to implement the individual parts of the algorithm, assuming that the φ is in CNF.

Lines 1–2 Dead Features: To detect whether a feature f is dead, check if $\varphi \wedge f$ is consistent. Now $\varphi \wedge f$ is a CNF formula; a single SAT call establishes consistency. Further, a positive answer comes with a witness, which proves liveness of all variables to which it assigns true, not just f. No further SAT calls are made for these. Also, the SAT solver is tuned to prefer witnesses with multiple true values, over those with many zeroes, to allow learning about many features in one call. Still, in the worst-case, detecting dead features performs $O(|F|)$ SAT calls.

Lines 3–4 Implications: Detecting binary implications requires proving validity for formulas of the $\varphi \wedge f_i \to f_j$ kind, or, equivalently, checking if $\varphi \wedge f_i \wedge \neg f_j$ is inconsistent. Thus one implication edge is detected by one SAT call. Detecting all implications requires $O(|F|^2)$ calls. In practice, again, a single witness can be used to disprove all implications between variables f_l and f_k, whenever f_l is assigned true, and f_k is assigned false.

Line 8 Mutual Exclusions: Detecting mutexes resembles detecting positive implications and is done by checking if $\varphi \wedge f_i \wedge f_j$ is inconsistent. Like above, finding all exclusions requires a quadratic number of SAT checks on a formula which is (essentially) the same size as φ. Again this number can be decreased by learning about more than one pair of features from a single witness.

Line 11–13 Or Groups: To identify or-groups we need to find prime implicates. We will rely on the following lemma:

LEMMA 1. *Let φ be a formula in CNF and C a clause, then $\varphi \to C$ if and only if there exist a clause C' such that $C' \subseteq C$ and C' is derivable from φ by resolution.*

See [9] for a proof. The idea is to perform consecutive resolutions of clauses of φ discarding subsumed resolvents, otherwise adding them to φ and removing clauses that are subsumed by them. If the fixpoint is reached with a result other than the empty clause, the result is the set of all prime implicates of φ. Completeness of this procedure was shown by Quine in the fifties, his proof is rephrased in [10, p. 24].

We synthesize or-groups using the PIG algorithm [25, 24], which orders the resolutions in the above scheme heuristically. Proofs for the completeness and soundness of PIG are outlined in [24]. However PIG itself is not sufficient. It generates all possible resolvents, and it is unlikely that it can be optimized to find or-groups efficiently. In our case we are only interested in implicants containing features that share the same parents in the (E', V') graph, and these features should only appear in positive form. Our brief experiments show that it is not feasible to generate all implicants and then filter out the uninteresting ones. Thus we apply variable elimination. For a given parent p, we eliminate from φ all features that are not its children, before proceeding to search for prime implicates of p in this smaller formula. This leads to significant performance gains.

We use VER [41] to eliminate variables. The output of $\text{VER}(\varphi, x)$ is a CNF formula ψ not containing the variable x, but *equisatisfiable* to φ. It turns out that formulas presented by VER are not only equisatisfiable, but also the set of prime implicants of φ over the kept variables is preserved:

THM. 4. *Let φ be a formula in CNF over the set of variables X, $x \in X$ and let $\psi = \text{VER}(\varphi, x)$. Let π be a clause consisting only of variables in $X \setminus \{x\}$. Then π is a prime implicate of φ if and only if π is a prime implicate of ψ.*

Incremental Computation of or-groups. The above way of identifying or-groups appears to do a lot of redundant work.

We first find implicates of $\varphi \wedge f$ for some parent feature f, and then seek for implicates of $\varphi \wedge f'$ for the next parent f'. But these two formulas are very similar. Alternatively, one can use an algorithm computing the prime implicates of $\varphi \wedge f$, assuming the prime implicates of φ are already known.

This procedure is strongly inspired by the PIGLET algorithm [24], which computes the prime implicates of a formula $\varphi \wedge \psi$, where φ is an arbitrary formula and ψ a formula in CNF, assuming the prime implicates of φ are known. Let Π_φ denote the set of prime implicates of the formula φ. Then the prime implicates $\Pi_{\varphi\wedge f}$ of $\varphi \wedge f$ can be computed as follows:

1. Let $\Pi = \Pi_\varphi$. Add f to Π and remove all clauses from Π subsumed by f.
2. Let $S = \{\text{Resolve}(\pi, f) \mid \pi \in \Pi_\varphi, \neg f \in \pi\}$. Add the clauses in S to Π and remove all $\pi \in \Pi$ with $\neg f \in \pi$ from Π as they are subsumed by a clause in S, since $\text{Resolve}(\pi, f) = \pi \backslash \{\neg f\}$.

This procedure can be used to compute the prime implicates of $\varphi \wedge r \wedge f$ efficiently. First, compute implicates of $\varphi \wedge r$ and then reuse the results to find implicates of $\varphi \wedge r \wedge f$ for each parent feature f. The resolution steps needed to compute the implicates of $\varphi \wedge r$ are only performed once.

7. SYNTHESIZING WITH FGE-DNF

FGE-DNF is a variant of FGE assuming DNF as input. In Sect. 2 we have shown that it is applicable to scenarios where models are to be synthesized from a list of existing variants of a product (*Scenario 1*).

FGE-DNF shares the structure with FGE (Fig. 7). We describe the details of the computation for DNF below. We assume that the DNF formula only contains satisfiable terms. A term is satisfiable if it does not contain a literal and its negation. Unsatisfiable terms can be removed in linear time.

Lines 1–2 Dead Features: A variable f is dead iff it appears negated in every term of φ. This can be checked in linear time in $|\varphi|$. So the step runs in $O(|\varphi||F|)$ time.

Lines 3–4 Implications: Since φ is in DNF, checking if $\varphi \wedge f_i \to f_j$ is valid can be done by checking if $\varphi \wedge f_i \wedge \neg f_j$ is satisfiable, which takes time linear in $|\varphi|$. Check if each term contains $\{\neg f_i, f_j\}$. Thus the detection of all implications can be done in $O(|\varphi||F|^2)$ time.

Line 8 Mutual Exclusions: Similarly, the satisfiability of $\varphi \wedge f_i \wedge f_j$ can be computed in linear time. So detection of exclusions also takes $O(|\varphi||F|^2)$ time.

Line 11–13 Or Groups: Recall that synthesizing or-groups requires identifying prime implicates of φ and, since φ is in DNF, this is equivalent to finding prime *implicants* of its negation. We will use a procedure based on Binary Integer Programming (BIP), a special case of Integer Linear Programming (ILP) that assumes binary domain for variables, to address this problem. BIP is an NP-complete problem [20] and thus strongly related to the NP-complete CNF-Shortest-Implicant problem (see Section 5).

We outline a straightforward polynomial reduction from finding implicants to BIP [40, 31]. It translates a CNF formula, here $\neg\varphi$, into a BIP problem P, with the property that any optimal solution to P corresponds to a shortest implicant of $\neg\varphi$. Let L be the set of literals occurring in $\neg\varphi$.

```
SOLVE(f : objective function, S : set of constraints,
      k : upper bound)
1  while (k ≥ 0)
2      S = S ∪ {f(x) ≤ k}
3      status = SAT(S)
4      if (status == satisfiable) then k = f(x') − 1
5      else return k = k + 1
6  return k
```

Figure 8: A SAT-based solver for BIP

1. For each $l \in L$ introduce a Boolean variable x_l. The objective is to minimize $\sum_{l\in L} x_l$
2. For each clause $l_1 \vee \cdots \vee l_m$ in $\neg\varphi$ add the linear inequality $x_{l_1} + \cdots + x_{l_m} \geq 1$ to the set of constraints.
3. As a literal l and its negation $\neg l$ cannot both be *true* in the same assignment of φ, a constraint of the form $x_l + x_{\neg l} \leq 1$ is added to the set of constraints.

As every feasible solution must satisfy all constraints in the BIP, at least one literal in each clause of φ corresponds to a variable assigned the value 1 (second constraint). Constraint 3 ensures that none of these literals are conflicting, i.e. the variables x_l and $x_{\neg l}$ do not both occur with the value 1 in a feasible solution. It follows that a conjunction of literals corresponding to the set of variables assigned the value 1 in any feasible solution is an implicant of φ. Moreover an optimal solution to the BIP corresponds to a prime implicant of φ, since the number of literals in the solution is minimal and therefore cannot be subsumed by another implicant.

In [31] two SAT-based algorithms MIN_PRIME and BSOLO for finding a shortest implicant by solving the corresponding BIP problem are presented. Experimental results comparing these algorithms to other BIP/ILP solvers show that SAT-based algorithms are preferable when computing minimal prime implicants, and that BSOLO tends to be more efficient than MIN_PRIME. Despite this conclusion, we have chosen to implement the MIN_PRIME algorithm because it can be implemented easily on top of any SAT solver that allows BIP input. The MIN_PRIME algorithm transforms a CNF formula $\neg\varphi$ into a BIP as described above and subsequently calls the subprocedure SOLVE (Fig. 8).

The MIN_PRIME algorithm can be extended to incrementally enumerate all prime implicants of $\neg\varphi$ [19]. Details about this extension, called PRIME, can be found in [2]. Instead of calling SOLVE just once, PRIME calls SOLVE iteratively. In each iteration, a prime implicant $l_1 \wedge \cdots \wedge l_k$ is returned by SOLVE. By adding a new constraint $x_{l_1} + \cdots + x_{l_k} < k$ to the set of constraints, the PRIME algorithm ensures that the same prime implicant will not be returned again in the following iterations.

Recall that given a formula φ over variables $\{f_1, \ldots, f_n\}$ an or-group of a feature f in φ corresponds to a prime implicate $(f_1 \vee \cdots \vee f_k)$ of $\varphi \wedge f$ containing only positive literals corresponding to children of f in the implication graph. By negation of this implication, it follows that an or-group corresponds to a prime implicant $(\neg f_1 \wedge \cdots \wedge \neg f_k)$ of $\neg\varphi \vee \neg f$ [3] that contains only negative literals corresponding

[3]Note that $\neg\varphi \vee \neg f$ is not in CNF, this can however easily be achieved since the operators $\vee$ and $\wedge$ are distributive.

to children of f. Thus every prime implicant of interest corresponds to an optimal solution of the BIP program, where each variable corresponding to a positive literal in $\neg\varphi \vee \neg f$ is assigned 0. To avoid the computation of prime implicants containing positive literals, we modify the BIP program by removing every variable corresponding to a positive literal in $\neg\varphi\vee\neg f$. Furthermore, if a variable in the BIP corresponding to a non-child of f in the implication graph is assigned the value 1 in a solution, this solution cannot correspond to an or-group of f. Consequently, we can also remove all variables in the BIP corresponding to non-children of f.

8. EXPERIMENTAL EVALUATION

We implemented the algorithm using the core SAT4J library (http://sat4j.org/). SAT4J is a widely used open-source Java interface to SAT solvers that implements the initial Minisat specification [18]. An advantage of SAT4J is the support of cardinality constraints, which allows a straightforward implementation of the PRIME algorithm (cf. Sect. 7). The performance of the PIG algorithm is heavily dependent on the expense of forward an backward subsumption and we have implemented the algorithms presented by Zhang [42].

We will now evaluate the efficiency of our techniques. Note that quality of the produced models does not need to be evaluated, since by design we create a compact representation of *all possible* diagrams consistent with the input. Evaluation of quality of the derived models belongs to work on tools that help deriving them (see Section 2 for possible scenarios).

To imitate a realistic usage of the algorithm our evaluation used input formulae representing dependencies amongst features, which were obtained from feature models translated into CNF, DNF and BDDs. We took a subset of models available at SPLOT and the feature model repository (splot-research.org) and (fm.gsdlab.org) with sizes ranging from 9 to 287 features. We further generated an additional 20 random 3-CNF feature models having 100 or 200 features using the Feature Model Generator on the SPLOT website. The 3-CNF feature models are tougher benchmarks than the real models since they tend to induce harder problems [32]. The experiments were run using Java 1.6 on a 2.0 GHz Intel Core 2 Duo processor. We used JavaBDD 1.0b2 for the BDD-based implementation. The memory available to the JVM was set to to 1.6GB and a timeout was recorded after one hour for all models (excluding the Linux kernel, see below)

Linux is an operating system kernel with an explicit variability model used to configure features in the kernel prior to compilation. In contrast to the other models, here we followed the early FM hierarchy selection workflow described in *Scenario 2*. Group and CTC recovery was performed after a hierarchy was selected. We used a propositional translation of the variability model in the version 2.6.28.6 of the Linux kernel [39, 38]. The model, with 5701 features, was too large for computing or-groups, however we used an alternative method of computing xor-groups that does not rely on or-groups as shown in Fig. 7. The alternative method first finds the set of mutex-groups and then checks for each of them whether at least one group member must be present:

$$G'_x = \{\{(f_1,p),\ldots,(f_k,p)\} \in G_m \mid \varphi\wedge p \rightarrow f_1\vee\cdots\vee f_k\} \quad (3)$$

Table 1 shows the total running time of the BDD-based implementations and of FGE-CNF with non-incremental or-group computation. The times are broken down into three components: the computation time of the implication graph (IG), mutex graph (MG) and or-groups (OR). The total time includes the computation of and-groups, mutex-groups, and xor-groups. We only show results for models with 43 or more features since the two implementations show no notable difference for smaller models.

$\|F\|$	DNF-terms	Model name	BDD	SAT-DNF
67	6400	Home-Integration-System	270 s	2.9 s
44	80658	Thread-Domain	•	35 s

• timeout

Table 2: Running time of the BDD-based vs the SAT-based implementation for DNF input

The computation times for the implication and mutex graphs are similar for FGE-BDD. However, the mutex graph computation for FGE-CNF takes significantly longer for large models—3.5 times longer for the Linux kernel. BDD methods are generally slightly faster if they succeed (but we are talking about differences in miliseconds). Unfortunately, they run out of memory for some cases. For the E-Shop model, the FGE-CNF performs better by computing the implication graph roughly 6 times faster and the mutex-graph 2 times faster than FGE-BDD. On the computation of or-groups, the SAT-based implementation is significantly faster than the BDD-based implementation for all models. 7 of the models did not terminate within an hour (timeout) while the other 3 ran out of memory while building the BDD. The BDD-based implementation managed to compute or-groups for only 2 models, (Documentation-Generation and Home-Integration-System) and was roughly 1000 times slower than the SAT-based implementation. With the randomly generated 3-CNF models, the results are similar where the FGE-CNF completed the computation and FGE-BDD timed out during the or-group calculation. See the online appendix.

We also evaluated FGE-DNF, using formulas obtained by enumerating all legal configurations for small models (below 67 features). FGE-DNF was at least 100 times faster than the BDD-based results when computing or-groups. See Table 2.

The main threat to validity in the above experiment lies in selection of instances for experiments. Since we are dealing with NP-hard problems it is always possible to tune the instances that are favourable for one technique and adversarial for the other. We avoid this bias by using realistic examples from public repositories and randomly generated models.

An internal threat is a possible incorrectness of our implementation that could affect performance. To mitigate this problem we have tested the algorithms against each other, primarily the CNF and the BDD version, but also the DNF-CNF-BDD triple for smaller examples.

9. RELATED WORK

The present work is described in greater detail in [2]. In [16], we show a BDD-based algorithm for synthesis of feature models from formulae. The algorithms presented here are based on SAT solving, resolution and binary integer programming, achieving improved performance. Essentially, the procedure presented in [16] was of theoretical interest—it explained how to identify semantic traits of feature diagrams in propositional constraints. The present paper provides an executable scalable technique. *Scenario 1* scales to medium size models

$\|F\|$		Fge-BDD[16] IG	MG	OR	total	Fge-CNF IG	MG	OR	total
43	Web-Portal	2 ms	2 ms	•	•	8 ms	38 ms	28 ms	89 ms
44	Documentation-Generation	18 ms	11 ms	220 s	220 s	30 ms	52 ms	159 ms	261 ms
44	Thread-Domain	6 ms	2 ms	•	•	28 ms	31 ms	711 ms	945 ms
46	Dell-Laptop-Notebook	13 ms	11 ms	•	•	24 ms	27 ms	360 s	360 s
58	GG4	65 ms	30 ms	•	•	24 ms	27 ms	21 s	21 s
61	Arcade-Game	20 ms	17 ms	•	•	55 ms	72 ms	711 ms	875 ms
67	Home-Integration-System	16 ms	3 ms	270 s	270 s	108 ms	17 ms	195 ms	347 ms
88	Model-Transformation	19 ms	8 ms	•	•	181 ms	264 ms	342 ms	858 ms
94	BerkleyDB	○	○	○	○	103 ms	153 ms	438 ms	1.0 s
170	Violet	○	○	○	○	125 ms	866 ms	2.9 s	4.0 s
287	E-Shop	19 s	15 s	•	•	3.0 s	7.2 s	110 s	120 s
5701	Linux kernel 2.6.28.6	○	○	○	○	1.7 h	6.1 h	•	7.8 h†

• timeout ○ out of memory

Table 1: Performance of the BDD-based and the SAT-based method for input formulas in CNF on real models

(a few hundred features). *Scenario 2* scales to very large models of thousands of features (without or-groups).

As discussed in Sect. 2, the presented synthesis problem is an essential subproblem in many usage scenarios. We have shown how the algorithm can be used with additional information coming from feature name similarity and user decisions to reverse engineer FMs from the build system and code with conditional compilation [39]. That work targets creating FMs for systems such as FreeBSD, which have a build system exposing several hundred variable features as compile options as a flat list. Other reverse engineering scenarios may require additional steps, such as feature identification and feature location [17], before the feature dependencies can be identified and fed into the presented algorithms. Acher [1] presents a model management framework for FMs based on [16]. His framework would experience a significant performance boost if employing our new algorithms. Janota et al. [26] propose an interactive tool for building feature models from propositional formulas. The tool uses a feature graph synthesized using [16] to determine the editing operations that create valid feature diagrams. Fge-CNF can be used to drastically improve the scalability of that tool.

Using logics and reasoners to analyze feature models is now well established [4]. Some of the steps in Fge, like dead feature detection, are known as separate analyses listed in [4].

Probabilistic feature models (PFM) extend feature models with soft constraints, expressing preference among legal configurations [15]. In [15] we have presented a method for extracting a PFM from a set of configurations using Bayesian statistics. Fge-DNF achieves a specialized result where all constraints have 100% probability. The group detection methods in [15] were based on [16], so [15] would considerably benefit from the current performance improvement.

Loesch and Ploedereder [30] extract variability from a sample set using formal concept analysis. The extracted variability is used to construct a concept lattice, exposing and-groups, mutually exclusive and dead features. Unlike the feature graph constructed by Fge, their lattice does not include or-groups. Ryssel et. al. [35] also exploit concept analysis to synthesize FMs including or- and xor–groups, from product matrices that correspond to our DNF representation. The complexity of this problem depends primarily on the number of configurations in the input. While we handle models of up to 80 thousand products, in up to 35 seconds, they report up to 63 products with times from 120s to 3 days. On the other hand, their technique synthesizes new abstract concepts (features), which do not. It would be interesting to investigate whether that technique could enhance our method, without significant loss of performance.

Coudert and Madre give two prime implicant algorithms [12]. Other methods for CNF input are found in [28, 25].

10. CONCLUDING REMARKS

We have presented algorithms for synthesis of feature models from propositional constraints by deriving symbolic representations of all candidate diagrams, and deriving instances from this diagrams.

We have designed and implemented the algorithms for the input expressed as a CNF or DNF formulae. We have shown experimentally that both techniques outperform the old BDD implementation by a factor of 10 to 1000 times, enabling the use of synthesis techniques in tools. The biggest tractable model for the BDD technique had 67 features (as opposed to 287 for Fge-CNF). More importantly, the BDD technique was extremely unpredictable failing for many smaller models as soon as they exceed 30 features. We also know that Fge-CNF scales to up to above 5000 features, if the or-group computation is switched off, whereas for the BDD-technique it is usually not even possible to build BDD-representations for feature model instances exceeding 2000 features [32].

Once a diagram FD is derived, one still needs to construct a textual cross-tree constraint ψ such that the entire feature model is equivalent to the input formula φ. Obviously, one choice for ψ is φ itself. However, normally we would like to simplify the formula, seeking a (syntactically) minimal ψ such that $[\![\mathsf{FD}]\!] \wedge \psi \equiv \varphi$. Unfortunately, finding minimal representations is difficult. Another possibility is to use an almost optimal approach such as the Espresso-II [7], known to efficiently produce close-to-minimal representations in practice. The efficiency of Espresso-II for our particular problem still needs to be investigated.

11. REFERENCES

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

[2] N. Andersen. Automatic synthesis of feature models based on satisfiability checking. Master's thesis, IT University of Copenhagen, 2009.

[3] D. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.

[4] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 2010.

[5] T. Berger, S. She, R. Lotufo, K. Czarnecki, and A. Wąsowski. Feature-to-code mapping in two large product lines. In *SPLC*, 2010.

[6] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *ASE*, 2010.

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

[8] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Commun. ACM*, 1973.

[9] H. K. Büning and T. Lettman. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, 1999.

[10] V. Chandru and J. Hooker. *Optimization Methods For Logical Inference*. Wiley, 1999.

[11] A. Classen, P. Heymans, P. Y. Schobbens, A. Legay, and J. F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *ICSE*, 2010.

[12] O. Coudert and J. C. Madre. Implicit and incremental computation of primes and essential primes of boolean functions. In *29th ACM/IEEE Conference on Design Automation*. IEEE Computer, 1992.

[13] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, and K. Pietroszek. fmp and fmp2rsm: Eclipse plug-ins for modeling features using model templates. In *OOPSLA Companion*, pages 200–201. ACM, 2005.

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

[15] K. Czarnecki, S. She, and A. Wąsowski. Sample spaces and feature models: There and back again. In *SPLC '08*. IEEE, 2008.

[16] K. Czarnecki and A. Wąsowski. Feature models and logics: There and back again. In *SPLC '07*. IEEE, 2007.

[17] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[18] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, LNCS. Springer, 2003.

[19] B. Errico, F. Pirri, and C. Pizzuti. Finding prime implicants by minimizing integer programming problems. In *Australian Joint Conference on AI*, 1995.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[21] M. L. Griss, J. Favaro, and M. d. Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98. IEEE Computer, 1998.

[22] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *SPLC*, pages 139–148. IEEE Computer, 2008.

[23] F. Heidenreich, J. Kopcsek, , and C. Wende. FeatureMapper: Mapping Features to Models. In *ICSE'08*, 2008.

[24] P. Jackson. Computing prime implicates incrementally. In *CADE*, 1992.

[25] P. Jackson and J. Pais. Computing prime implicants. In *CADE*. Springer, 1990.

[26] M. Janota, V. Kuzina, and A. Wąsowski. Model construction with external constraints: An interactive journey from semantics to syntax. In *MoDELS*, 2008.

[27] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, CMU, 1990.

[28] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *Journal of Symbolic Computation*, 9(2), 1990.

[29] T. Kishi and N. Noda. Formal verification and software product lines. *Commun. ACM*, 49(12), 2006.

[30] F. Loesch and E. Ploedereder. Optimization of variability in software product lines. In *SPLC '07*. IEEE Computer, 2007.

[31] V. M. Manquinho, P. F. Flores, J. P. M. Silva, and A. L. Oliveira. Prime implicant computation using satisfiability algorithms. In *ICTAI*. IEEE Computer, 1997.

[32] M. Mendonca, A. Wąsowski, and K. Czarnecki. SAT-based analysis of feature models is easy. In *SPLC '09*. IEEE, 2009.

[33] M. Mendonca, A. Wąsowski, K. Czarnecki, and D. Cowan. Efficient compilation techniques for large scale feature models. In *GPCE '08*, 2008.

[34] OMG. ad/09-12-03. Common Variability Language RFP with ab changes, 2009.

[35] U. Ryssel, J. Ploennigs, and K. Kabitzsch. Extraction of feature models from formal contexts. In *FOSD*, 2011.

[36] K. Schmid, R. Rabiser, and P. Gruenbacher. A comparison of decision modeling approaches in product lines. In *VAMOS*, 2011.

[37] P. Y. Schobbens, P. H., J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2), 2007.

[38] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the linux kernel. In *VaMoS*, 2010.

[39] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE*, 2011.

[40] J. P. M. Silva. On computing minimum size prime implicants. In *International Workshop in Logic Synthesis*, 1997.

[41] S. Subbarayan and D. Pradhan. Niver: Non-increasing variable elimination resolution for preprocessing sat instances. In *SAT*. Springer, 2004.

[42] L. Zhang. On subsumption removal and on-the-fly CNF simplification. In *SAT*, 2005.