Feature Diagrams and Logics: There and Back Again

Krzysztof Czarnecki
University of Waterloo, Canada
kczarnec@swen.uwaterloo.ca

Andrzej Wąsowski
IT University of Copenhagen, Denmark
wasowski@itu.dk

Abstract

Feature modeling is a notation and an approach for modeling commonality and variability in product families. In their basic form, feature models contain mandatory/optional features, feature groups, and implies and excludes relationships. It is known that such feature models can be translated into propositional formulas, which enables the analysis and configuration using existing logic-based tools. In this paper, we consider the opposite translation problem, that is, the extraction of feature models from propositional formulas. We give an automatic and efficient procedure for computing a feature model from a formula. As a side effect we characterize a class of logical formulas equivalent to feature models and identify logical structures corresponding to their syntactic elements.

While many different feature models can be extracted from a single formula, the computed model strives to expose graphically the maximum of the original logical structure while minimizing redundancies in the representation. The presented work furthers our understanding of the semantics of feature modeling and its relation to logics, opening avenues for new applications in reverse engineering and refactoring of feature models.

1 Introduction

Feature modeling is a family of notations and an approach for modeling commonality and variability in product families [15, 11, 4]. In their *basic* form, feature models contain mandatory/optional features, feature groups, and implies and excludes relationships [12]. As shown by Batory [4], such feature models can be translated into propositional formulas. The translation enables the analysis and configuration using existing logic-based tools, such as SAT solvers and Binary-Decision Diagram (BDD) libraries.

In this paper, we consider the problem of opposite translation, that is, the extraction of feature models from propositional formulas. We give an automatic and efficient procedure for computing a feature model from a formula. As a side effect we characterize a class of logical formulas equivalent to feature models and identify logical structures corresponding to their syntactic elements. While many different feature models can be extracted from a single formula, the computed model strives to expose graphically the maximum of the original logical structure while minimizing redundancies in the representation. The computed model can then be refactored to take into account additional concerns beyond the logical structure.

The presented work furthers the understanding of the semantics of feature diagrams and their relation to logic, enabling new applications in reverse engineering and refactoring of feature models. The intended audience of this paper are other researchers working with feature modeling and developers of feature modeling tools. The former shall benefit from deepened exploration of the relation between logics and feature models, the latter can learn a representation of the semantics of a model that is well suited to guiding the user during construction and refactoring of models.

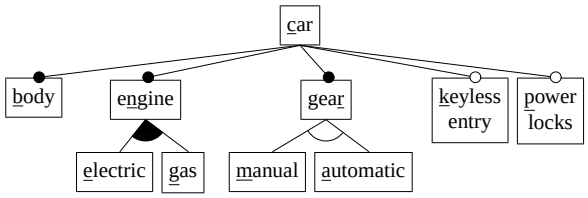
We proceed as follows. Section 2 provides background by describing basic feature models, their translation into logics, implication graphs and BDDs. In Section 3 we discuss the requirements and challenges of extracting feature models from formulas. We present an automatic extraction procedure in Section 5 and analyze it in Section 6. In Section 7 the applicability of the procedure and future work are discussed. Sections 8–9 survey related work and conclude.

2 Background

2.1 Basic Feature Models

A feature model is a tree of features. An example is shown in Figure 1. Table 1 summarizes the notation.

The root of the tree represents the *root feature* (*car*). Remaining nodes represent *grouped features* (e.g., *electric*) or *solitary features* (e.g., *body*). Solitary features can be *mandatory* (*body*) or *optional* (e.g., *keyless entry*). Grouped features can be contained by *xor-groups* (e.g., *manual* and *automatic*) or by *or-groups* (e.g., *electric* and *gas*). Additional constraints are specified as *propositional formu-*



keyless_entry → power_locks

Figure 1. A sample feature model

Table 1. Syntax of cardinality-based feature models

Symbol	Explanation
	Solitary feature with cardinality $[1..1]$, i.e., *mandatory* feature
	Solitary feature with cardinality $[0..1]$, i.e., *optional* feature
	Grouped feature
	Feature group with cardinality $\langle 1\text{–}1\rangle$, i.e. *xor*-group
	Feature group with cardinality $\langle 1\text{–}k\rangle$, where k is the group size, i.e. *or*-group

las, using $\vee$ (disjunction), $\wedge$ (conjunction), $\veebar$ (exclusive-or), $\barwedge$ (not-and or nand), $\rightarrow$ (implication), $\leftrightarrow$ (equivalence), and $\bar{\cdot}$ (negation). In our example, we additionally assume *keyless_entry*$\rightarrow$*power_locks*.

A feature model represents a set of *configurations*, each being a set of features selected from a feature model according to its semantics. The set of configurations represented by a feature model can be described by a *propositional formula* defined over a set of Boolean variables, where each variable corresponds to a feature. The propositional formula can be constructed as a conjunction of (i) implications from all subnodes to their parents, (ii) additional implications from parents to all their mandatory features, (iii) implications from parents to groups, and (iv) any additional constraints represented as propositional formulas.[1] An implication from a parent feature p to its subfeatures $f_1, \ldots, f_k$ that form an or-group has the following form:

$$p \rightarrow \bigvee_{i=1,\ldots,k} f_i \tag{1}$$

Similarly, a parent feature p to its subfeatures $f_1, \ldots, f_k$ that form an xor-group is defined as follows:

$$p \rightarrow \underline{\bigvee}_{i=1,\ldots,k} f_i \tag{2}$$

[1]We chose not to include the root feature as an additional argument in the conjunction, i.e., the empty configuration is always a correct one.

As an example, consider a formula corresponding to feature model of Figure 1. For brevity, we have abbreviated feature names to single letters, underlined in Figure 1.

$$\begin{aligned} q_{FM} = & \\ \textit{child-parent:}\quad & (b \rightarrow c) \wedge \\ & (n \rightarrow c) \wedge (e \rightarrow n) \wedge (g \rightarrow n) \wedge \\ & (r \rightarrow c) \wedge (m \rightarrow r) \wedge (a \rightarrow r) \wedge \\ & (k \rightarrow c) \wedge \\ & (p \rightarrow c) \wedge \\ \textit{mandatory:}\quad & (c \rightarrow b) \wedge (c \rightarrow n) \wedge (c \rightarrow r) \wedge \\ \textit{or-group:}\quad & (n \rightarrow e \vee g) \wedge \\ \textit{xor-group:}\quad & (r \rightarrow m \veebar a) \wedge \\ \textit{additional:}\quad & (k \rightarrow p) \end{aligned} \tag{3}$$

Any assignment of Boolean values to all features that makes the propositional formula satisfied represents a correct configuration of the feature model.

2.2 Conjunctive Normal Form and Implication Hypergraphs

A propositional formula is in *conjunctive normal form* (CNF) if it is a conjunction of *clauses*, which are disjunctions of *literals* (atoms or their negations). Any nonempty clause can be converted into an equivalent implication in one of the following three forms. A clause containing at least a negative and a positive literal is converted as follows:

$$\begin{aligned} & \bar{f}_1 \vee \ldots \vee \bar{f}_m \vee f_{m+1} \vee \ldots \vee f_{m+n} \\ \equiv\ & f_1 \wedge \ldots \wedge f_m \rightarrow f_{m+1} \vee \ldots \vee f_{m+n} \end{aligned} \tag{4}$$

A clause with one or more positive literals is converted as:

$$f_1 \vee \ldots \vee f_n \equiv \textit{true} \rightarrow f_1 \vee \ldots \vee f_m \tag{5}$$

Similarly for a clause with one or more negative literals:

$$\bar{f}_1 \vee \ldots \vee \bar{f}_m \equiv f_1 \wedge \ldots \wedge f_m \rightarrow \textit{false} \tag{6}$$

A conjunction of implications of the above form can be visualized as an *implication hypergraph*, in which nodes correspond to variables and constants (*true* or *false*) while directed hyperedges correspond to the implications. For every hyperedge all sources are conjoined together and their conjunction implies a disjunction of all the targets. As a special case a binary edge corresponds to the usual binary implication. Any formula can be visualized as an implication hypergraph in a sound and complete manner, using the above translation and the standard translation to CNF.

Figure 2 shows an implication hypergraph for the feature model of Figure 1, obtained directly from (3). Each implication of (3) is shown as a directed binary edge. The or-group implication from (3) corresponds to the hyperedge from *n* to *e* and *g*. The xor-group implication is represented by two hyperedges—observe that an xor-group is an or-group and a mutual exclusion for every pair of group members. Logically, mutual exclusion between features m and a is represented using the nand-operator $m \barwedge a$, which is equivalent to

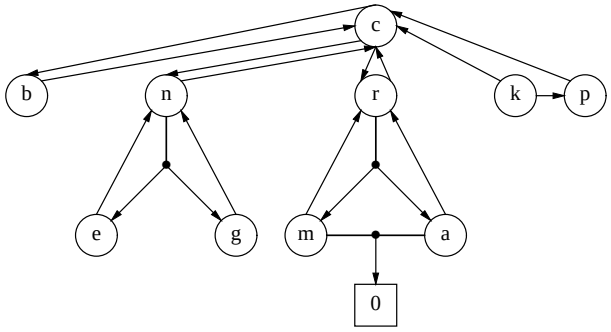


Figure 2. An implication hypergraph for the feature model in Figure 1

$\bar{m} \vee \bar{a}$. This, in turn, is represented by a hyperedge from m and a to the special node 0 representing *false*.

Another useful concept is that of an *implication graph*, which is similar to the implication hypergraph except that it only shows binary edges. More precisely, an implication graph of a formula φ over $f_1, \ldots, f_n$ is a directed graph $G(V, E)$ such that the set of vertices contains all variables, and there is an edge from variable f_i to variable f_j iff f_i implies f_j, assuming formula φ. Formally:

$$V = \{f_1, \ldots, f_n\} \quad E = \{(f_i, f_j) \mid \varphi \wedge f_i \rightarrow f_j\} \quad (7)$$

2.3 Binary Decision Diagrams

A propositional formula φ over n variables represents a binary function $\{0,1\}^n \rightarrow \{0,1\}$. Binary functions can be efficiently stored, analyzed, and manipulated using a class of data structures commonly referred to as *binary decision diagrams* (BDDs). BDDs are widely used in hardware synthesis and model checking. Since our algorithm, presented later, relies on BDDs, we introduce them briefly here. For more information see, e.g., [8, 3, 17].

Any binary function can be represented using a *binary decision tree* (BDT). An example of BDT representing the binary function described by the formula $(k \rightarrow p) \wedge (p \rightarrow c)$ is shown Figure 3. The formula corresponds to the fragment of the implication hypergraph from Figure 2 involving the features c, k, and p. Each non-terminal node in a BDT has a *low edge* (dashed line) and a *high edge* (solid line). Any variable assignment corresponds to a path from the root to a terminal node in a BDT. If a given variable is assigned 0, the low edge of the corresponding node is taken. Similarly, if a variable is assigned 1, the high edge is taken. The resulting value for the assignment is given by the terminal node at the end of the path, which is either 0 or 1. A BDT essentially represents the entire *truth table* of a binary function and its size is exponential in the number of variables.

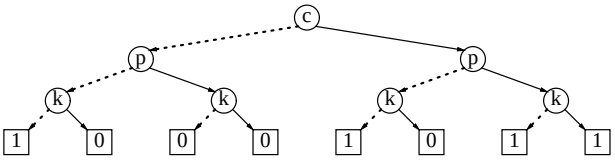


Figure 3. A sample BDT for a fragment of the feature model in Figure 1

Looking at the BDT in Figure 3, we realize two sources of redundancies, which bloat the representation:

1. Some nodes have no influence on the outcome of the function; e.g., the value of k in the rightmost node labeled with that variable from the left in Figure 3 has no influence on the function outcome since both its low and high edges point to the terminal 0.

2. The tree contains many duplicate subtrees; e.g., the subtrees rooted in the first and the third node labeled with k from the left are identical; also, the terminal nodes are duplicated multiple times.

Removing the above redundancies by introducing node sharing results in the graph shown in Figure 4, which is an example of a *reduced ordered binary decision diagram* (ROBDD) [8]. Although several types of BDDs exist, the term *BDD* is usually just an abbreviation for *ROBDD*, which also applies to the rest of this paper.

Thus, a BDD can be thought of as a compressed representation of a binary function. While the size of a BDT only depends on the number of variables, the size of a BDD usually depends both on the binary function and the *variable ordering* (which is $k < p < c$ in Figure 4) in a rather unpredictable way. Still, the BDD size never (asymptotically) exceeds the size of the corresponding BDT. While computing an optimal ordering for a given function is an NP-hard problem, efficient heuristics exist. As a result, many practically important boolean functions can be efficiently represented as BDDs, which has been amply demonstrated in hardware synthesis, software verification, fault tolerance analysis and constraint programming.

Canonicity is another important property of BDDs (shared with BDTs): equivalence of two formulas can be established by checking that their BDD representations for a fixed variable ordering are identical. If properly implemented, such check only takes constant time. Checks of many other properties, such as satisfiability, tautology, or counting the number of solutions, and many operations like computing the conjunction or disjunction of two BDDs, can be performed in at most polynomial time with respect to the sizes of the BDDs involved. Moreover efficient optimized implementations are provided by off-the-shelf BDD libraries (e.g. Buddy, JavaBDD, JDD, CUDD).

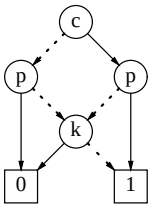


Figure 4. A sample BDD corresponding to the BDT in Figure 3

3 Feature Model Extraction: Challenges and Requirements

We have already given semantics to feature diagrams using boolean logic. Let us now begin discussing challenges and requirements for the reverse process, namely translation of formulas to feature diagrams.

The sample feature model in Figure 1 and its implication hypergraph in Figure 2 have a clear structural correspondence. This correspondence suggests the possibility of extracting a feature model from a formula by first extracting an implication hypergraph and then recognizing the various feature relationships in that hypergraph. For example, an optional subfeature will have a binary implication edge to its parent; a mandatory subfeature will have both a binary implication edge to and from its parent; and each member of an or-group will have a binary implication edge to its parent, and all the members will be targets of a single implication hyperedge from the parent.

The general idea of extracting a feature model from a formula faces several challenges and these challenges become apparent when we consider the extraction process via implication hypergraphs.

- *Challenge 1: Many different feature models can be extracted from one formula.* As an example, consider the implication hypergraph in Figure 5(a). Looking at the graph, we immediately recognize the pattern for an or-group. As a result, we can draw the equivalent feature model in Figure 5(b). Since the implication from b to c cannot be accommodated by the feature tree, it is shown as an additional constraint. However, further analysis of this feature model reveals that c is part of every correct configuration. Consequently, feature models in Figures 5(c) and 5(d) are also equivalent to the implication hypergraph in Figure 5(a).

- *Challenge 2: Showing all implied relationships and groups can be overwhelming to the user.* The example in Figure 5 demonstrated that we might not want to show all the groups that are implied by the implication hypergraph since the same information may already be visualized by a simple feature model (e.g., Figure 5(d)). Similarly, while binary edges in the implication graph may be shown as subfeature relationships in a feature model, the transitivity of implication results in many more binary edges that we might want to show in a feature model (e.g., see Figure 6). An approach to extracting feature models from formulas will have to be selective about which potential edges and groups to show.

- *Challenge 3: Logical structure is not the only structuring criterion for feature models.* While the propositional formula produced from a feature model as described in Section 2.1 captures all the correct feature configurations of the feature model, it does not contain all the information about the structure of the diagram. For example, since a mandatory feature is logically equivalent to its parent, both the feature model in Figure 5(c) and the one in Figure 5(d) are logically equivalent, in the sense of denoting the same set of configurations. However, we would often insist on a particular feature being the mandatory subfeature of another feature in order to convey additional ordering and grouping information. For example, as both *car* and *body* are logically equivalent in Figure 1, either one could be the parent of the other without affecting the set of correct configurations. However, we clearly want *body* as the subfeature of *car* in order to express the part-of relationship. Consequently, the automatic extraction can only be partial in the sense that the user will likely need to do further restructuring in order to account for the additional information.

- *Challenge 4: Brute force search for an optimal feature model will not scale.* Since there are many equivalent feature models that can be extracted from a formula, the extraction algorithm will need to favor some form of minimality of the result. The latter is clearly connected to the problem of minimization of propositional formulas, which is known to be NP-complete. Thus, the algorithm will have to be carefully designed to avoid potential exponential blow-up.

Based on the first three challenges above, we conclude that a practical approach to feature model extraction from propositional formulas will have to be a combination of an automatic procedure and interactive user involvement. In the following sections, we give an automatic extraction procedure with the following properties:

- **Maximality property:** *The resulting feature model graphically exposes maximum of logical structure.* For example, while the feature model in Figure 5(b) still needs an additional textual constraint, the equivalent feature model in Figure 5(e) visualizes the complete

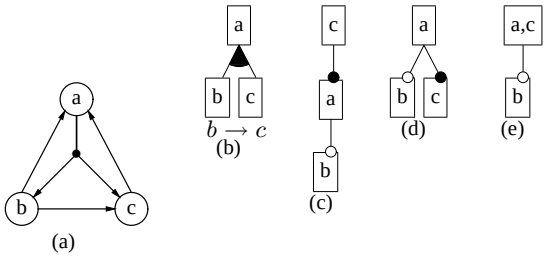


Figure 5. An implication hypergraph and its equivalent feature models

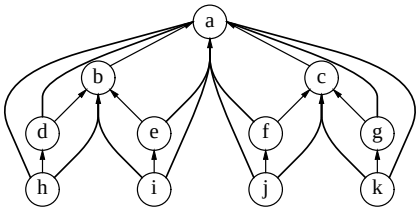


Figure 6. A transitively closed implication graph

logical structure without any additional constraints. In general, as shown in Section 2.2, additional constraints may be unavoidable since a feature model expressed in the notation from Section 2.1 cannot capture all hyperedges (only binary edges arranged hierarchically and the limited forms of hyperedges in groups can be captured). However, as much logical structure as possible should be visualized.

- **Minimality property:** *The resulting feature model avoids redundancy in the representation.* For example, in Figure 5(d), although b and c are clearly in an or-group relationship with respect to a, showing that group in addition to what is already in the diagram would not add any new information.

- **Uniqueness property:** *Logically equivalent formulas lead to the same feature diagram.* We want the procedure to be deterministic in order to minimize surprises.

The automatically produced feature model is intended as a starting point for further refactoring by the user.

4 Traits of Feature Models in Formulas

Before giving the actual feature model extraction algorithm in Section 5, we discuss the key ideas behind the algorithm. In particular, we describe how to detect the traits of

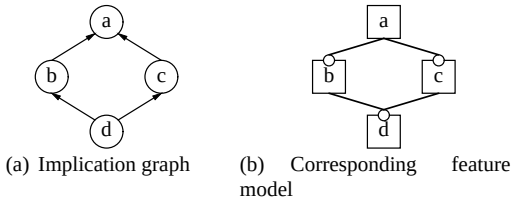


Figure 7. Node sharing in the generalized notation

feature model syntax, such as the feature hierarchy, mandatory and optional features, and or- and xor-groups, given a logical formula φ over variables $f_1, \ldots, f_n$.

Existence of configurations Let us begin with a simple observation that an unsatisfiable formula, or a formula that describes no legal configurations, cannot possibly be represented as a feature model. So the most fundamental characteristics of a formula φ that can embed a feature diagram is that φ is satisfiable. Any algorithm for extraction of feature trees from φ should establish its satisfiability, as otherwise it could create an ill-formed feature model that does not allow any configurations (not even the empty configuration).

Dead features A given feature f is *dead* [5] if it is not present in any configurations of the feature diagram. Dead features are usually caused by modeling errors and should not be present in well formed feature models. Dead features have their counterpart in the logic formulas—they correspond to variables that are always assigned false:

$$\{f \mid \varphi \rightarrow \bar{f}\} \tag{8}$$

All the variables in the above set should not participate in any of the well-formed features models derived from φ.

Feature hierarchy As we have seen in Section 2.1, a parent–child relationship in a feature tree corresponds to a child–parent implication in logics. Consequently the edges of a feature tree overlap with implications among the variables of φ, while the implication graph $G(V, E)$ of φ (see (7)) overapproximates the edges of a feature diagram in the sense that a feature tree is its spanning tree. If φ describes a feature model, then G will only have one *sink* (a vertex with no outgoing edges) and there will be a path from any feature to this sink. For an arbitrary formula the graph may be disconnected and may have multiple sinks. Because of the transitivity of implication, G will have potentially a large number of implied edges, many of which unlikely candidates for feature tree arcs. Because of this is the minimality property from Section 3, we will only consider the *transitive reduction* of G.

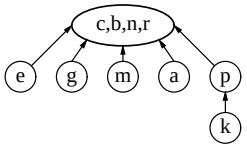


Figure 8. The reduced implication graph for the feature model of Figure 1

Furthermore, both G and its transitive reduction may not be a tree, but a directed graph, and, thus, the transitively reduced version of G may have multiple spanning trees. Since there is no clear criteria for favoring one spanning tree over another and because of the desired uniqueness property (Section 3), we avoid the problem by assuming the entire transitively reduced G after cycle removal (described in the next paragraph) to be the feature hierarchy. Consequently, the feature hierarchy of the computed feature model may not be a tree, but it is guaranteed to be a directed acyclic graph (DAG), and thus the original feature modeling notation from Section 2.1 needs to be slightly generalized by allowing node sharing. Figure 7 gives an example how an implication graph being a DAG can be visualized as a feature model using the generalized notation.

And-groups According to Section 2.1 an and-group with parent f and children $f_1, \ldots f_l$ entails implications from the parent f to children f_i and vice-versa. Due to transitivity of implication we have that for any $i, j = 1 \ldots l$ variable f_i implies f_j. And-groups thus manifest themselves as *cliques* in the implication graph of φ (a clique is a subgraph in which any two vertices are connected by an edge). Candidates for maximal and-groups can be found by identifying maximal cliques. These can then be replaced by and-groups. Figure 8 shows the transitive reduction of G after cycle removal. The graph contains one and-group gathering the features c, b, n, and r. Following the convention introduced in Figure 5(e), an and-group is visualized using a single node.

Mandatory features Mandatory features are logically indistinguishable from and-group members. All mandatory subfeatures of a feature collapse into a single and-group.

Or-groups An or-group with parent f and members $f_1, \ldots f_k$ gives rise to the implication

$$f \rightarrow f_1 \vee \cdots \vee f_k \tag{9}$$

Recall that adding new terms to a disjunction weakens the formula (the set of solutions increases). Thus, if the above implication holds, so do *many* other implications with disjunctions comprising supersets of $f_1, \ldots, f_k$ enriched with *arbitrary* literals, e.g, $f \rightarrow f_1 \vee \cdots \vee f_k \vee f_{k+1}$. Consequently, we want to find all the *minimal* disjunctions of features implied by a parent.

More precisely, given the parent feature f and all its children $f_1, \ldots, f_l$, we want to find all minimal disjunctions $f_1 \vee \cdots \vee f_k$ of a subset of $f_1, \ldots, f_l$ for which the following formula is a tautology:

$$\varphi \rightarrow (f \rightarrow f_1 \vee \cdots \vee f_k) \tag{10}$$

Each of the disjunctions should be minimal in the sense that removing any literal from the disjunction would invalidate the above formula. Furthermore, $f_1, \ldots, f_l$ are children of f in G after cycle removal but before transitive reduction, so that all potential groups are detected.

It turns out that such minimal disjunctions can be computed by computing so called *prime implicants* of a formula. An *implicant* of a formula $\varphi\prime$ is a conjunction of literals $l_1 \wedge \ldots \wedge l_n$ such that $l_1 \wedge \ldots \wedge l_n \rightarrow \varphi\prime$ is a tautology. A prime implicant is an implicant that cannot be reduced by removing literals from it, in the sense that the resulting conjunction would not be an implicant of $\varphi\prime$. As prime implicants are widely used in reliability analysis and in hardware synthesis, several efficient methods for computing them exist [10, 16] and can be used in tools to identify or-groups.

In order to see how finding prime implicants identifies or-groups, consider a slight transformation of formula (10):

$$\begin{aligned} & \varphi \rightarrow (f \rightarrow f_1 \vee \cdots \vee f_k) \\ \equiv\ & \varphi \rightarrow (\bar{f} \vee f_1 \vee \cdots \vee f_k) \\ \equiv\ & \bar{\varphi} \vee \bar{f} \vee f_1 \vee \cdots \vee f_k \\ \equiv\ & \bar{f_1} \wedge \cdots \wedge \bar{f_k} \rightarrow \bar{\varphi} \vee \bar{f} \end{aligned} \tag{11}$$

where the last line above is equivalent to

$$\bar{f_1} \wedge \cdots \wedge \bar{f_k} \rightarrow (\varphi \rightarrow \bar{f}) \tag{12}$$

Thus, we can find all (minimal) or-groups $f_1, \ldots, f_k$ of f by computing all prime implicants of $\varphi \rightarrow \bar{f}$ and selecting those that contain only the negated children of f. We additionally need to filter out the prime implicants that are inconsistent with φ, i.e., those for which $\bar{f_1} \wedge \cdots \wedge \bar{f_k} \wedge \varphi$ is not satisfiable, since they are actually or-groups of *true*. These groups could also be visualized if we choose to include an additional node corresponding to *true* in the computation of the implication graph.

Since several minimal or-groups implied by φ can overlap and we have no good way to automatically prefer one over another, the generalized feature notation used for showing the extracted feature model allows for overlapping groups (see Figure 9).

Xor-groups Every xor-group is an or-group, which can be characterized by prime implicants as described above,

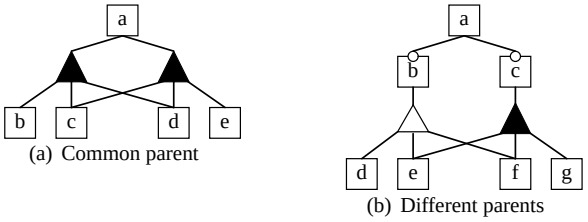


Figure 9. Group overlap in the generalized notation

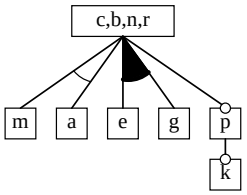


Figure 10. Extracted feature model

and that additionally requires that its members are mutually exclusive. Thus, if a prime implicant identifies an or-group with members $f_1, \ldots, f_k$, and it also holds that

$$\forall i = 1..k. \forall j = 1..k, i \neq j. \varphi \rightarrow f_i \bar{\wedge} f_j \qquad (13)$$

then f_i's form an xor-group.

Optional features A feature g is an optional subfeature of f iff $g \rightarrow f$ holds and $f \rightarrow g$ does not.

5 The Algorithm

Characterizations of previous section are used in the algorithm reconstructing feature models from formulas (see Figure 11). For a formula φ over a set of boolean variables $F = \{f_1, \ldots, f_n\}$ the algorithm creates an augmented implication graph G over live features in F. This graph combined with a suitable left-over constraint is equivalent to φ. If the original formula represented a simple feature model, then the feature graph is a tree corresponding to the model.

The algorithm follows the scheme of Section 4. We begin with checking satisfiability of φ (line 1), and removing all dead features (2–4). Then an implication graph is constructed (5–7). Subsequently cliques in the graph need to be identified in order to extract the and-groups. In general finding maximal cliques is an NP-hard problem, as a graph can contain exponentially many of them [19, Section 6.1]. Fortunately due to transitivity of implication, maximal cliques are actually strongly connected components in the implication graph, which can be found efficiently by graph traversal, for example breadth-first search (lines 8-9).

FEATURE-GRAPH(φ : formula)

```
1  if not SAT(φ) then quit with an error.

2  ▷ Find and remove all dead features
3  D = {d_1, ..., d_m} ← all f_i ∈ F such that φ → f̄_i
4  φ ← ∃d_1, ..., d_m. φ

5  ▷ Compute the implication graph G(E, V)
6  V ← F − D
7  E ← {(u, v) ∈ V × V | φ ∧ u → v}

8  ▷ Compute strongly connected components of G
9  C ← a set of SCCs in G

10 ▷ Contract SCCs in G creating and-groups
11     do for c ∈ C do let f be a fresh node
12        V ← V ∪ {f} − c
13        E ← {(u, v) ∈ E | u ∉ c ∧ v ∉ c} ∪
14            ∪ {(u, f) | ∃v.(u, v) ∈ E} ∪
15            ∪ {(f, v) | ∃u.(u, v) ∈ E}

16 ▷ G is acyclic (a DAG) at this point.
17 ▷ Compute OR-groups and XOR-groups
18 for each f ∈ V
19     do for each prime f̄_1 ∧ ... ∧ f̄_k of φ → f̄
20           do if SAT(f̄_1 ∧ ··· ∧ f̄_k ∧ φ)
21                 then Let f be a fresh node
22                      V ← V ∪ {f}
23                      E ← E ∪ {(f_1, f)}
24                              ∪ {(f, f_i) | i = 1...k}
25                      if f_2, ..., f_l satisfy (13)
26                      then mark f as an XOR node

27 Compute the unique transitive reduction of G.
```

Figure 11. Extraction of a feature model

In lines 10–15 the connected components are contracted to single nodes. These new nodes represent groups of features that always need to be present together. A tool can later decide to render such nodes as and-groups.

In lines 16–26 or-group candidates are identified by searching for prime implicants. For each or-group a fresh node representing it is created, with implications from parent features to the node and from the node to the group members. A modeling tool can later decide to render such nodes as or-groups. All or-group nodes are tested whether they are not xor-groups and marked accordingly (25–26).

Finally standard transitive reduction techniques [1, 18] are applied to the graph. The result is uniquely determined as our graph is acyclic at this point, after the cliques had

been contracted. Even though visualizations of reduced graphs are much easier to comprehend, this step is optional. Some non visual tools may need to access all implications.

Figure 8 presents an implication graph for the model of Figure 1, after the clique contraction, but before or-groups have been identified. Only a transitive reduction of the graph is shown, to avoid clutter. Figure 10 shows the final graph rendered as a feature model.

Observe that there is no need to detect optional features. All features in singleton groups are optional. Otherwise they would be identified as parts of cliques and contracted.

5.1 BDD-based Implementation

Our algorithm utilizes several operations on logical formulas: satisfiability and tautology checks, compositions with basic connectives and the computation of prime implicants. In order to implement it, one needs a representation for logical formulas that can support these operations efficiently. Binary decision diagrams, introduced in section Section 2.3, are one such representation. Once a formula is translated into a BDD using standard techniques [8, 3, 17] a SAT and TAUTOLOGY checks can be performed in constant time. A check whether an implication of a literal holds takes linear time in the size of the BDD.

On top of the above operations we use two non-standard algorithms on BDDs to implement lines 3, 7, 19 and 25. Consider line 7 as an example—finding implications from every feature. The most direct solution would be to use linear time to compute BDDs representing $\varphi \wedge f_i \rightarrow f_j$ for all pairs of variables in F, which would cost $O(|F|^2 \cdot |\varphi|)$ time. Instead we construct BDDs representing $\varphi \wedge f_i$ for each of the linearly many f_i's and compute VALID-DOMAINS [13, 14] for variables in these BDDs.

A *valid domain* for a variable f given a formula φ is the set of values that f can assume in satisfiable assignments of φ. Using the VALID-DOMAINS algorithm we can decide all the variables that are implied by f by just computing valid domains of $\varphi \wedge f$ once. All variables whose valid domain only contains *true* are implied by f. For a formula encoded as a BDD valid domains can be computed in time linear in the size of the BDD [13]. The entire graph can be built in $O(|F| \cdot |\varphi| + |F|^2)$ time, where $|\varphi|$ denotes the size of the BDD representing φ. A similar trick is used to increase efficiency in lines 3 and 25.

Line 19 is concerned with computing prime implicants. Already in 1992 Coudert and Madre [10] have proposed an efficient symbolic algorithm for computing prime implicants of a formula represented as a BDD. This algorithm is easily modified to only return implicants containing negative terms. The implicants are themselves stored symbolically in another BDD, which allows the procedure to terminate even though the number of the implicants could be prohibitive to represent them explicitly. This is particularly useful for our algorithm as one can compute the BDD of prime implicants and query it for the number of implicants (this a linear time operation) and only after that decide whether she wants to visualize all of them as potential or-groups. If the amount of information passed to the user exceeds certain size limits it is probably wiser not to present it at all. Obviously this can only happen if the formula we analyze does not represent a feature model and has a very complex structure.

The decision to use BDDs in our implementation was grounded in our previous experience, availability of good BDD libraries, and easily available literature, and *not* inherently in the algorithm. Any other representation for formulas supporting the required operations could have been used instead. Alternatively search-based techniques of SAT-solving could likely be applied. All our operations, including computing prime implicants [16] can in principle be implemented using a SAT-solver and several standard techniques for manipulating CNF representations. We have not tried this possibility though, and thus we cannot comment on difficulty or efficiency of such a solution.

6 Analysis

We have implemented our algorithm using the open source JavaBDD library and the popular Graphviz package. Implementation and testing cost less than a week of work of a full-time programmer. With this prototype we have successfully reconstructed several feature models, confirming practically that the algorithm works and can be used to produce successful visualizations. In the rest of the section, we analyze some of the key properties of the algorithm.

Reconstruction of Feature Models Our algorithm successfully constructs feature models for formulas that actually describe them. More precisely it constructs a tree with additional nodes for feature groups that can be straightforwardly translated into a regular feature model syntax.

If the output of the algorithm is not a feature tree with semantics equivalent to φ (but a directed acyclic graph with semantics *implied* by φ) then φ is a formula that cannot be directly represented as a feature tree. The relation between feature models and combinatorial logical formulas has been known for a long time now, but to our best knowledge this is the first attempt ever to describe a subclass of formulas directly equivalent to feature diagrams.

The graph constructed by the algorithm has several pleasant properties itself, and we think that, accompanied with a BDD, it can in itself be useful as an internal representation of the problem in feature modeling and refactoring tools. Having such a graph at hand the tool can propose refactorings that go beyond the simple catalog of [2].

Quality of the constructed graph The graph constructed by our algorithm is *complete* in the sense that it contains all and-group, or-group and xor-group candidates. It indicates all possible parent-child relationships and all possible optional subfeature relationships. Thus a tool using the algorithm can provide a maximum guidance to the user constructing a feature tree.

Our graph is also a safe overapproximation of the original formula, meaning that it is never ruling out any configurations allowed by satisfiable assignments of φ. The graph can always be constrained further using a propositional constraint, so that the resulting graph allows the same amount of configurations as the feature tree allowed (the translation of the graph to logics is done using the same rules as for feature models, mentioned in Section 2.1). This additional formula can be obtained automatically by simplifying φ with respect to a formula ψ resulting from translating the graph back to logics. One way to obtain this simplification is to use the algorithm for simplifying BDDs [9, 3], translating them back to a propositional formula and using a generic formula simplifier.

Consequently the interactive editor for feature models can use the computed graph to guide user safely, without risking that some originally planned configurations are removed. The tool can either synthesize the left-over constraint ψ automatically or let the user do it, verifying only whether conjoined with the semantics of the model it is equivalent to the input formula.

Performance Assume that the formula φ is represented as a BDD of size $|\varphi|$. In the worst case $|\varphi|$ is exponentially larger than the syntactic representation of φ, however BDDs are known to handle many classes of formulas extremely well, in particular also those that occur in configuration problems [14]. With BDDs the satisfiability check in line 1 is constant time [8]. Lines 3-4 can be implemented in $O(|\varphi| + |F|)$ time using a VALID-DOMAINS algorithm [13]. We use the same algorithm to compute the implication graph in line 7–8, obtaining complexity of $O(|F|\varphi + |F|^2)$ as described in Section 5. Strongly connected components are found using graph search, typically in $O(|E|+|V|)$ time, so in our case at most $O(|F|^2)$ time if the graph is very dense. Typically product configuration graphs are sparse, giving an algorithm that is linear in $|F|$. La Poutré and van Leeuwen [18] give an incremental algorithm for computing transitive reduction in $O(|E| \cdot |V|)$ time. Which is $O(|F|^2)$ for sparse graphs.

The most expensive step in our algorithm is the computation of prime implicants. Coudert and Madre [10] give two procedures for this problem, labeled IP1 and IP2. IP2 is said to be polynomial in $|\varphi|$, which allows us to make a claim that the entire algorithm of Fig. 11 *can be implemented in time polynomial* in $|\varphi|$. However IP1 is reported to perform better on practical problems, so our prototype is actually using it.

7 Suggestions for future work

We see two main categories for future work: (i) applications of the presented concepts and (ii) extensions of the concepts.

Applications The original motivation for this work was the desire to support potentially complex refactorings such as merging of feature models and extraction of views on a feature model. During that work we have realized that while the semantics of refactorings can naturally be captured in terms of operations on logical formulas, we did not have a way to translate the results back to feature models. While the presented work provides a step towards such a support for refactoring, several challenges remain. In particular, ways to take additional structuring information such as preferred feature ordering and grouping into account need to be investigated.

Another application for this work is reverse engineering of feature models from formulas that were not obtained from other feature models, but, for example, reverse engineered from code. At this point however it is unclear how satisfying the structure computed by the algorithm would be for the users of such applications. We envision that reverse engineering tools will need to cast the reconstruction as an interactive process using the computed graph as a knowledge base for a guidance heuristics.

Also, the combination of the extraction procedure with the refactorings described in [2] needs to be investigated.

Extensions In this paper, we only consider a basic notation similar to the original FODA notation [15]. We still need to investigate how general n-to-m groups can be efficiently detected. An even more ambitious project would attempt the treatment of models with feature replication.

8 Related work

The relation between feature models and propositional logic has been studied by several authors, including Batory [4], Bontemps et al. [7], and Wei Zhang and colleagues [21]. The connection between feature models and logic has led to the application of existing logic-based tools to support feature configuration and feature model debugging. For example, Batory [4] explores the use of SAT solvers for that purpose. Van der Storm [20] considers BDD packages, and Benavides with colleagues [6] uses constraint solvers for the same purpose.

All these consider the translation of feature models into logic and we are not aware of any work on extracting feature models from formulas.

9 Conclusion

We have furthered the understanding of the connection between feature models and logics by studying the problem of extracting feature models from logical formulas. In particular, we have identified the challenges of feature model extraction and presented an algorithm for extracting a feature model that visualizes the maximum of the formula's logical structure while avoiding redundancy. The resulting feature model is identical for any equivalent formulas. We have also characterized the limited expressiveness of feature models compared to propositional logic via the concept of implication hypergraphs. Finally, we have suggested several directions for future work, such as exploring the application of the presented concepts in feature model refactoring and reverse engineering, considering additional aspects beyond logical structure, and treating more general feature model notations.

References

[1] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

[2] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring product lines. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 201–210. ACM Press, 2006.

[3] H. R. Andersen. *Binary Decision Diagrams*. Department of Information Technology, Technical University of Denmark, Lyngby, Denmark, 1997. Lecture notes for 49285 Advanced Algorithms E97, `http://www.itu.dk/people/hra/notes-index.html`.

[4] D. S. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[5] D. Benavides, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A survey on the automated analyses of feature models. In J. Riquelme and P. Botella, editors, *JISBD 2006: XV Jornadas de Ingeniería del Software y Bases de Datos*, Barcelon, 2006.

[6] D. Benavides, P. Trinidad, and A. Ruiz-Cortés. Automated reasoning on feature models. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, 2005*, LNCS. Springer, 2005.

[7] Y. Bontemps, P. Heymans, P. Schobbens, and J. Trigaux. Generic semantics of feature diagrams variants. In *Features Interactions in Telecommunications and Software Systems VIII(ICFI'05)*, pages 58–77, Leicester, UK, Jun 2005.

[8] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, Aug. 1986.

[9] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 365–373. Springer-Verlag, 1989.

[10] O. Coudert and J. C. Madre. Implicit and incremental computation of primes and essential primes of boolean functions. In *Proceedings of the 29th ACM/IEEE Conference on Design Automation*, pages 36–39. IEEE Computer Society Press, 1992.

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1), 2005. Special issue on Software Variability: Process and Management, `http://swen.uwaterloo.ca/~kczarnec/spip05a.pdf`.

[12] K. Czarnecki, C. H. P. Kim, and K. Kalleberg. Feature models are views on ontologies. In *Proceedings of 10th International Software Product Line Conference (SPLC 2006)*, pages 41–51. IEEE, 2006.

[13] T. Hadzic, R. Jensen, and H. R. Andersen. Notes on calculating valid domains. Manuscript online `http://www.itu.dk/~tarik/cvd/cvd.pdf`, 2006.

[14] T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO Conference*, pages 131–138. DTU-tryk, June 2004.

[15] 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, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[16] V. M. Manquinho, P. F. Flores, J. P. M. Silva, and A. L. Oliveira. Prime implicant computation using satisfiability algorithms. In *9th IEEE International Conference on Tools with Artificial Intelligence Newport Baach, CA, USA*, pages 232–239. IEEE Computer Society, 1997.

[17] C. Meinel and T. Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag, 1998.

[18] J. A. L. Poutré and J. van Leeuwen. Maintanance of transitive closures and transitive reductions of graphs. Technical Report RUU-CS-87-25, Rijksuniversiteit Utrech, 1987.

[19] G. Valiente. *Algorithms on Trees and Graphs*. Springer-Verlag, 2002.

[20] T. van der Storm. Variability and component composition. In *Proceedings of ICSR8*, LNCS. Springer, 2004.

[21] W. Zhang, H. Zhao, and H. Mei. A propositional logic-based method for verification of feature models. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004. Proceedings*, volume 3308 of *Lecture Notes in Computer Science*, pages 115–130, Heidelberg, Germany, 2004. Springer-Verlag.