Reverse Engineering Feature Models

Steven She
University of Waterloo
shshe@gsd.uwaterloo.ca

Rafael Lotufo
University of Waterloo
rlotufo@gsd.uwaterloo.ca

Thorsten Berger
University of Leipzig
berger@informatik.uni-leipzig.de

Andrzej Wąsowski
IT University of Copenhagen
wasowski@itu.dk

Krzysztof Czarnecki
University of Waterloo
kczarnec@gsd.uwaterloo.ca

ABSTRACT

Feature models describe the common and variable characteristics of a product line. Their advantages are well recognized in product line methods. Unfortunately, creating a feature model for an existing project is time-consuming and requires substantial effort from a modeler.

We present procedures for reverse engineering feature models based on a crucial heuristic for identifying parents—the major challenge of this task. We also automatically recover constructs such as feature groups, mandatory features, and implies/excludes edges. We evaluate the technique on two large-scale software product lines with existing reference feature models—the Linux and eCos kernels—and FreeBSD, a project without a feature model. Our heuristic is effective across all three projects by ranking the correct parent among the top results for a vast majority of features. The procedures effectively reduce the information a modeler has to consider from thousands of choices to typically five or less.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*

General Terms

Design, Languages

Keywords

Feature models, feature similarity, variability modeling

1. INTRODUCTION

Software product lines (SPL) enable effective development of a range of related products with differing sets of features. The SPL paradigm is centered around a number of practices leading to systematic code reuse [9]. Large-scale SPLs, such as the Linux, eCos, and FreeBSD operating system kernels, contain thousands of features and many dependencies among them. These dependencies pose a challenge for both developers and users. For developers, adding or removing features or dependencies requires understanding the impact of such changes. For users that instantiate a product from the product line, intricate dependencies between features lead to errors during the configuration process.

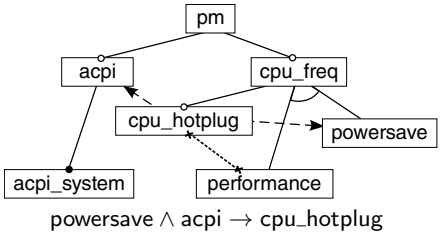

Figure 1: Power management feature model

Some projects, such as Linux and eCos, address these difficulties by providing feature models to describe their product line [7]. A *feature model* describes features—the common or variable characteristics of the products in a SPL—as a visual hierarchy with additional constraints between features [13]. Feature models offer a range of benefits from enabling automated analysis for verifying and resolving product line consistency to generating graphical configurators that guide users through the configuration process [5].

Figure 1, inspired by the Linux kernel model, shows a feature model of a power management sub-system. Features are represented as rectangles and may be *optional*—denoted by an empty circle—or *mandatory*—denoted by a filled circle. An edge from one feature to another denotes a dependency: a solid line denotes a *child-parent edge* of the feature tree, where the child implies the parent; a dashed line with an arrowhead represents a *cross-tree implies edge*; and a dotted line with x's is an *excludes edge*. Features may also belong to a group. In our example, performance and powersave are in an XOR-group meaning that one and only one may be selected. The feature tree, the group constraints, and the cross-tree edges form a *feature diagram*. A feature model consists of a feature diagram and, possibly, of a *cross-tree formula*—an additional cross-tree constraint expressing more complex dependencies.

Other projects, such as FreeBSD, do not have a feature model. These projects describe features and dependencies in an ad-hoc manner—features are scattered in documentation

ICSE '11, May 21–28, 2011, Waikiki, Honolulu, HI, USA

and dependencies are hidden in code. Such projects would benefit from having an explicit feature model instead. Unfortunately, constructing a model is both time and cost-intensive. Building the feature hierarchy in particular, requires substantial effort from a modeler. This task requires the modeler to review feature descriptions and dependencies to determine which dependencies to model in the hierarchy. FreeBSD has 1203 features—constructing a feature model for this project would require tremendous time and effort. Furthermore, the difficulty is compounded when the modeler lacks a complete set of dependencies. In this case, the dependencies might need to be uncovered by comparing feature descriptions. This may require the modeler to sift through hundreds of features in order to determine a parent for a single feature. Even if the modeler is given the complete set of dependencies, selecting the right parent for a feature is still challenging—a single feature may depend on over a hundred others as we have observed for Linux and eCos.

We present a tool-supported approach for reverse engineering feature models. The key challenge in this task is the construction of the feature diagram which reduces to the selection of a parent for each feature. We present heuristics for identifying the likely parent candidates for a given feature. Our heuristics significantly decrease the number of features that a user has to consider from potentially thousands to only a handful—typically five or less, as shown by our experiments. We also provide automated procedures for finding feature groups, implies and excludes edges. The final feature model is correct with respect to the input dependencies.

Our procedures require a list of feature names, descriptions and a propositional formula specifying dependencies. Feature names and descriptions can be extracted from documentation, preprocessor symbols or code comments. For the FreeBSD kernel, we extracted input data for our procedures by analyzing Makefiles, preprocessor declarations, and documentation, using a combination of generic and custom extraction tools.

Due to the complexity, size and nature of most software projects, it is likely that the extracted feature dependencies and descriptions are incomplete. Our heuristics accommodate this incompleteness by leveraging two sources of data that complement one another—when dependencies are incomplete, the feature descriptions are used to identify parent candidates and vice versa.

We evaluate the effectiveness of our procedures by comparing the results of our heuristics to the reference feature models of the Linux, eCos and FreeBSD kernels. Linux and eCos both have an existing reference feature model [7]. The input data for the procedures was extracted from the reference models themselves. For FreeBSD, we manually constructed a reference feature model for a subset of features after domain analysis. The evaluations show that, for 76% of features in Linux and 79% in eCos, the correct parent is in the top five parent candidates returned by our heuristics. In contrast to Linux and eCos, the input set of dependencies for FreeBSD is incomplete, and thus, we consider two separate results for FreeBSD: (1) for 84% of the features whose parent dependency is in the set, the correct parent is in the top two candidates; (2) for 75% of the remaining features, the correct parent is in the top or 3% of all 1203 features. Finally, our procedure automatically recovers all feature groups, as presented in the reference models for Linux and eCos, provided that the modeler settled on the same hierarchies as these

$$
\begin{aligned}
&(\mathsf{acpi} \rightarrow \mathsf{acpi\_system} \wedge \mathsf{pm}) \\
\wedge\ &(\mathsf{acpi\_system} \rightarrow \mathsf{acpi}) \\
\wedge\ &(\mathsf{cpu\_freq} \rightarrow \mathsf{pm}) \\
(1)\ \wedge\ &(\mathsf{cpu\_freq} \rightarrow \mathsf{powersave} \vee \mathsf{performance}) \\
(2)\ \wedge\ &(\mathsf{cpu\_hotplug} \rightarrow \mathsf{powersave}) \\
(3)\ \wedge\ &(\mathsf{cpu\_hotplug} \rightarrow \neg\mathsf{performance}) \\
\wedge\ &(\mathsf{cpu\_hotplug} \rightarrow \mathsf{acpi} \wedge \mathsf{cpu\_freq}) \\
\wedge\ &(\mathsf{powersave} \rightarrow \neg\mathsf{performance}) \\
\wedge\ &(\mathsf{powersave} \rightarrow \mathsf{cpu\_freq}) \\
\wedge\ &(\mathsf{performance} \rightarrow \mathsf{cpu\_freq}) \\
\wedge\ &(\mathsf{powersave} \wedge \mathsf{acpi} \rightarrow \mathsf{cpu\_hotplug})
\end{aligned}
$$

(a) Dependencies

pm Power management, CPU and ACPI options
acpi Advanced Configuration and Power Interface support
acpi_system Enable your system to shut down using ACPI
cpu_freq CPU frequency scaling
cpu_hotplug Allows turning CPU on and off
powersave This CPU governor uses the lowest frequency
performance This CPU governor uses the highest frequency

(b) Features and descriptions

Figure 2: Example input

models. With the incomplete dependencies of FreeBSD, we were still able to retrieve one of the three feature groups.

The contribution of this work is twofold. On the practical side, we present heuristics and procedures for reverse engineering feature models. Although reverse engineering feature models from logic formulas [11] and descriptions [2, 14] were considered before in separation, the main novelty of our approach is that it combines both sources of information together. This combination is desirable since, as our evaluation shows, the two sources are complementary. Also the procedures of [11] and [2, 14] are not complete, in the sense that the former cannot recover parents which are not direct dependencies, while the latter suggests only a single hierarchy that is unlikely the desired one. Moreover, in contrast to previous work, we evaluate reverse engineering of feature models on large-scale real-world systems showing that our approach and procedures scale. On the theoretical front, we expand our understanding of feature models by showing how both configuration semantics and ontological semantics relate to feature hierarchy.

We proceed as follows. Section 2 gives an overview of the procedure. Section 3 provides the background supporting the definition of the procedure (Section 4) and the evaluation (Section 5). We discuss threats to validity, relate to existing works and conclude in sections 6, 7 and 8 accordingly.

2. OVERVIEW

In this section, we demonstrate how our procedures assist the user in reverse engineering a feature model. Figure 2 shows a set of dependencies (given as a formula), feature names, and descriptions that we use as input data.

Our procedure reduces the reverse-engineering process of building the feature hierarchy, finding feature groups, and inserting implies and excludes edges in a sound and complete manner, to just the first step: building the feature hierarchy.

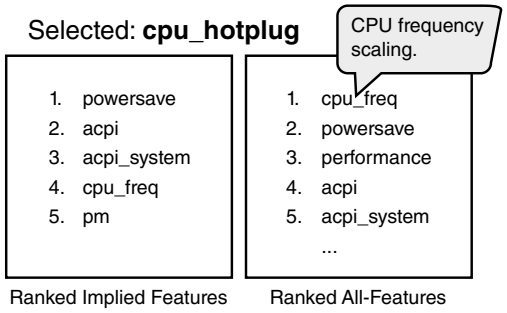

Figure 3: Parent candidates

The remaining steps are automated. Crucially, we support the user in building the hierarchy itself by providing tailored suggestions of parents, avoiding the need to sift through a multitude of candidates.

The key challenge of building the feature hierarchy is the selection of a parent for each feature. It requires understanding the meaning of the feature and its relationships to all other features. As a result, the hierarchy building process is inherently *interactive* and requires a *domain expert* modeler to review alternative choices for a feature's parent and to select the most suitable one. Our procedures present two lists of *parent candidates* to suggest the most appropriate parents for every feature, thus significantly reducing the total number of features to review at each step.

The first list is the *ranked implied features* (RIFs)—a sorted list of features that a given feature implies. Implied features are the primary criteria when deciding a parent—the semantics of feature models state that a child implies its parent. However, a feature may imply more than one other feature. This is where a ranking heuristic is applied to sort the implied features by their similarity to the selected feature, placing the most likely candidates at the very top.

The second list is the *ranked all-features* (RAFs)—all features sorted by their similarity to a given feature. The RAFs is a complete ranking, but is typically less accurate than the RIFs. It can be reviewed in the case the input dependencies are incomplete and the user cannot find an appropriate parent in the RIFs. In this case, the RAFs is useful for identifying potential parents where an implication from the selected feature may be missing due to incompleteness of available dependency information. We describe the details of the hierarchy building procedure in Section 4.

As an example, assume that the user is selecting a parent for cpu_hotplug. We would present its parent candidates as in Figure 3. There are five features here that are implied features of cpu_hotplug with powersave at the top position, and the actual parent cpu_freq is at the fourth position. If the dependencies are incomplete and the user cannot find an appropriate parent in the left list, the right list can be reviewed. Later, we show that the best candidates for parents are typically highly ranked in both lists.

Furthermore, once the feature hierarchy is decided, feature groups are detected. The user reviews the feature groups and select the ones that should be retained in the feature diagram. Any feature groups not retained in the diagram are kept as part of the cross-tree formula. For example, if the hierarchy in Figure 1 is chosen, our tooling will detect two feature groups: a MUTEX-group between cpu_hotplug and performance and an XOR-group between performance and powersave. The user selects one of the groups to keep in the diagram, relegating the other to the cross-tree formula.

Finally, mandatory features, implies and excludes edges are automatically discovered and added to the feature diagram. For example, assume that the user decides on the hierarchy in Figure 1 and implication (2) is omitted from the dependencies in Figure 2. Our procedure can still detect an implies edge from cpu_hotplug $\rightarrow$ powersave since it can be derived from implications (1) and (3). Now that the diagram is finished, further constraints are added to the cross-tree formula to make the resulting feature model sound—all legal configurations of the feature model are legal configurations with respect to the input dependencies. All these steps relies on SAT-based reasoners and thus, are independent from the syntactic structure of the dependency constraint.

Furthermore, if the user assumes the dependencies are complete, then only the RIFs needs to be reviewed by the user. The RAFs are no longer needed because all possible alternatives for parents are contained in the RIFs.

Our procedures can be integrated into existing feature model editors, in the style of [12], to equip them with reverse engineering capabilities and to allow modelers to make parent and group decisions in a graphical representation. However, we do not advocate any specific user interface design at this point. We leave this to future work.

3. BACKGROUND

3.1 Feature Modeling

We first define the abstract syntax of a feature diagram.

DEF. 1. *A* feature diagram *is a tuple* $\mathsf{FD} = (\mathcal{F}, E, I, X, G, C)$*, where* $\mathcal{F}$ *is a finite set of features,* $E \subseteq \mathcal{F} \times \mathcal{F}$ *is a set of directed child-parent edges;* $I \subseteq \mathcal{F} \times \mathcal{F}$ *is a set of* implies *edges,* $X \subseteq \mathcal{F} \times \mathcal{F}$ *is a set of* excludes *edges,* $G \subseteq 2^E$ *are non-overlapping sets of edges participating in feature groups. The final component,* $C \colon G \rightarrow \mathbb{N}_0 \times \mathbb{N}_0$ *is a mapping from a group to a pair denoting the cardinality of the group. The following well-formedness constraints hold in* FD*: (i)* $(\mathcal{F}, E)$ *is a rooted tree, (ii) all edges in a group share the same parent, so if* $g \in G$ *and* $(f_1, f_2), (f_3, f_4) \in g$ *then* $f_2 = f_4$ *and (iii)* $\forall\ (m, n) \in \text{range}(C)$*,* $m \leq n$*.*

A MUTEX-group is defined as a group m with cardinality $C(m) = (0, 1)$ and an XOR-group x is one where $C(x) = (1, 1)$. Furthermore, with Def. 1, mandatory features are syntactic sugar—they are represented by an *implies* edge from parent to child. Features not having such an edge are optional.

DEF. 2. *A feature model,* $\mathsf{FM} = (\mathsf{FD}, \phi)$*, where* FD *is the feature diagram and* ϕ *is a propositional formula over* $\mathcal{F}$*.*

The primary meaning of a feature model, known as its *configuration semantics*, is a set of *legal configurations*—sets of selected features that respect the dependencies entailed by the diagram and the cross-tree constraints. The configuration semantics can be specified via translation to logic [4]. Take the feature model in Figure 1 for example, the formula in Figure 2a defines its legal configurations.

In general, the function $\mathsf{p}(\cdot)$, defined below, translates a feature diagram or a feature model to propositional logic, interpreting features as variable names. For brevity, we use a Boolean predicate $\mathsf{choice}_{m,n}(\cdots)$ in its definition. Given Boolean variables $f_1, \ldots, f_k$ and $0 \leq m \leq n \leq k$,

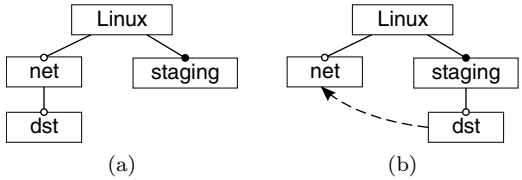

Figure 4: Same configurations, different ontological semantics

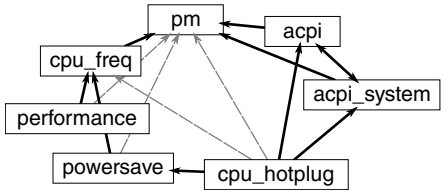

Figure 5: Implication graph where the transitively reduced subgraph is marked with thick edges

$\mathsf{choice}_{m,n}(f_1, \ldots, f_k)$ holds iff at least m and at most n of $f_1, \ldots, f_k$ are true.

DEF. 3. *For a diagram* $\mathsf{FD} = (\mathcal{F}, E, I, X, G, C)$ *define:*

$$p(\mathsf{FD}) = \bigwedge_{(c,p)\in E} (c \to p) \;\wedge\; \bigwedge_{(f,k)\in I} (f \to k) \;\wedge\; \bigwedge_{(f,k)\in X} (f \to \neg k) \;\wedge\; \bigwedge_{\substack{g\in G,\ (m,n)\in C(g),\\ g=\{(f_1,f),\ldots,(f_k,f)\}}} \big(f \to \mathsf{choice}_{m,n}(f_1, \ldots, f_k)\big)$$

Given $\mathsf{FM} = (\mathsf{FD}, \phi)$*, define* $p(\mathsf{FM}) = p(\mathsf{FD}) \wedge \phi$.

While the configuration semantics is most commonly associated with feature modeling, there exist other meanings of feature models. For example, Figure 4 depicts two feature models with identical configuration semantics, yet different hierarchies and meaning. The feature dst stands for distributed storage, a driver that allows accessing remote storage as a local device. The parent of dst affects its meaning. In (a), dst is nested under net meaning the feature is a product-quality device. In the diagram (b), dst is under staging, implying the feature is experimental and not ready for mainstream use. We call such semantics the model's *ontological semantics*, reflected in the meaning of the features and the hierarchy.

3.2 Linux, eCos and FreeBSD

We evaluate our procedures on Linux, eCos and FreeBSD kernels. Linux is an open-source general purpose OS created in the early 1990's. It has a feature model specified in the Kconfig language [20] for configuring features in its kernel. eCos is a real-time operating systems designed for embedded devices. It uses its own modeling language called the Component Definition Language (CDL), to specify its model. While Kconfig and CDL were developed independently from feature modeling, both models so closely resemble feature models that they can be interpreted as one [7, 16, 17]. FreeBSD is also an open-source OS. Unlike Linux and eCos, FreeBSD does not have a feature model, but only a flat list of features. The Linux feature model has over 5000 features, while eCos and FreeBSD have over 1200 features. We use the data from Linux to train and empirically build our procedures, and eCos and FreeBSD as tests subjects to evaluate our procedures.

4. THE PROCEDURES

The procedures assumes three kinds of inputs: a set of feature names $\mathcal{F}$, feature descriptions $\mathcal{D}$, and feature dependencies Φ. The ordering heuristics ignore the order of words in the descriptions, so $\mathcal{D}$ is defined as a mapping assigning a multiset of words to each feature in $\mathcal{F}$. A *multiset* is a pair (X, c), where X is a set and $c\colon X \to \mathbb{N}_1$. Given a feature f we write $\mathcal{D}_1(f)$ to refer to the set of words in f's description and for a word $w \in \mathcal{D}_1(f)$, $\mathcal{D}_2(f)(w)$ denotes the number of occurrences of w in f's description. Finally, dependencies are specified as a propositional formula Φ over $\mathcal{F}$.

The heuristics assume feature descriptions contain words from both the feature name itself and from any associated text. Often times, the feature names provides significant clues to its relation with other features. For example, cpu_hotplug and cpu_freq share the common word cpu. We apply a simple tokenization where we split the feature name by the underscore character and retain only alphanumeric words. We further assume that the descriptions have been stemmed, have stop words removed, and are case-insensitive. For example, using the descriptions in Figure 2 with stop words marked in gray, $\mathcal{D}(\mathsf{cpu\_freq}) = \{\mathsf{cpu}, \mathsf{cpu}, \mathsf{freq}, \mathsf{frequency}, \mathsf{scale}\}$.

4.1 Building Feature Hierarchy

Implication Graph.

First, we define several pre-requisites for the procedures. We assume that none of the input features are dead—features that no valid configuration can include. Dead features are automatically detected and removed.

Next, a feature *implication graph* is a pair (V, E), where V is a set of features and $E \subseteq V \times V$ is a directed edges, such that $(s, t) \in E$ whenever Φ entails the implication $s \to t$. Figure 5 is the graph constructed from dependencies in Figure 2a. An implication graph is *transitively closed* due to the transitivity of implications. If $w \to u$ and $u \to v$ then $w \to v$.

The *transitive reduction* of an implication graph is the subgraph not containing the aforementioned transitive implications, except for cliques. The transitive reduction can be computed using known algorithms [1, 11]. We denote the transitive reduction of a graph G by G_{R} and edges remaining in G_{R} as *direct implications*. Figure 5 shows the transitively reduced subgraph in thick edges.

Next, $\mathrm{E}(G)$ denotes the set of edges in a graph G. For a feature f write $\mathrm{I}_f(G)$ to denote features implied by f (so heads of edges outgoing from f). Then $\mathrm{I}_f(G_R)$ gives the directly implied features of f in G.

Identifying Parents.

In essence, our parent ranking heuristics leverages two complementing forms of data: dependencies and descriptions. The dependencies describe the configuration semantics of our feature model and the descriptions are used to approximate its ontological semantics.

Given a parent candidate p and the selected feature s, we define the similarity function $\delta(p, s)$ to return the sum of the *inverse document frequency* (IDF) of the words shared between the descriptions of p and s, weighted by the number of occurrences of each shared word in p's description (Equa-

tion 1). The parent candidates whose descriptions share many words with feature s and with shared words often repeated are then ranked highly similar to the selected feature. Furthermore, we use the IDF to give less weight to common domain words such as 'Linux', 'eCos', 'choose', or 'select'. These words are not stop words according to standard natural language processing tools, but they do not contribute to finding commonalities between two feature descriptions.

$$\delta(p,s) = \sum_{w \in \mathcal{D}_1(p) \cap \mathcal{D}_1(s)} \mathrm{idf}(w) * \mathcal{D}_2(p)(w) \quad (1)$$

$$\text{where } \mathrm{idf}(w) = \log \frac{|\mathcal{F}|}{|\{f : w \in \mathcal{D}_1(f)\}|}$$

We use the similarity function δ to induce a ranking order on features. Given a selected feature s we define two strict partial orders: $>_s$ and $>_s^{\mathbf{P}}$. In the first, $>_s$, features are ranked strictly by their description similarity to s:

$$a >_s b \text{ iff } \delta(a,s) > \delta(b,s) \quad (2)$$

The second partial order $>_s^{\mathbf{P}}$ prioritizes directly implied features of s over all other implied features. This prioritization is based on our observation that in Linux 88% of parents in the model are directly implied features. The partial order is defined as:

$$a >_s^{\mathbf{P}} b \text{ iff } \begin{cases} a \in \mathrm{I}_s(G_R) \wedge b \notin \mathrm{I}_s(G_R) & \text{or} \\ a \in \mathrm{I}_s(G_R) \wedge b \in \mathrm{I}_s(G_R) \wedge a >_s b & \text{or} \\ a \notin \mathrm{I}_s(G_R) \wedge b \notin \mathrm{I}_s(G_R) \wedge a >_s b \end{cases} \quad (3)$$

Finally, we can define the two lists that make the parent candidates: The RIFs ranks only the implied features of f using the prioritizing order while the RAFs ranks all features using the non-prioritizing order.

DEF. 4. *Given a feature s and an implication graph G, $\mathsf{RIF}(s)$ is the list created by sorting features in $\mathrm{I}_s(G)$ in decreasing order with respect to $>_s^{\mathbf{P}}$ (largest rank first). $\mathsf{RAF}(s)$ is the list of features in $\mathcal{F}$ sorted in decreasing order with respect to $>_s$. The orders are made total (any ties broken) by applying alphabetical ordering, to ease browsing.*

As an example consider determining the RIFs of cpu_hotplug. Feature cpu_hotplug has five implied features, three of which are directly implied (Figure 5). Examining their descriptions in Figure 2b, we see that cpu_hotplug shares the word 'cpu' with cpu_freq, performance, pm and powersave. The RIFs of cpu_hotplug are:

$$\mathsf{RIF}(\text{cpu\_hotplug}) = \langle \text{powersave}, \text{acpi}, \text{acpi\_system}, \text{cpu\_freq}, \text{pm} \rangle$$

The RAFs of cpu_hotplug uses the set of all features instead:

$$\mathsf{RAF}(\text{cpu\_hotplug}) = \langle \text{cpu\_freq}, \text{performance}, \text{pm}, \ldots \rangle$$

The user chooses parents for every feature by examining RIFs and RAFs, forming a set of directed child-parent edges $E \subseteq \mathcal{F} \times \mathcal{F}$. These edges may not form a single tree when there is no common top-level ancestor. In this scenario, we insert an additional root feature to join together the forest to form a single tree.

4.2 Groups and Cross-Tree Constraints

After the hierarchy is built, we detect the remaining components of a feature diagram—namely, feature groups and implies and excludes edges.

A mutex graph is used to detect feature groups and excludes edges. A *mutex graph* is an undirected graph consisting of vertices being features and edges denoting a mutual exclusion between two features u and v such that Φ entails $u \to \neg v$. Unlike in the implication graphs, edges in the mutex graph are not transitive. The mutex graph constructed from the dependencies in Figure 2a consists of two edges: cpu_hotplug *excludes* performance and performance *excludes* powersave.

Feature Groups.

A MUTEX-group defines a $[0\,..\,1]$ cardinality among its members. MUTEX-groups are recovered by finding all maximal cliques in the mutex graph M among sets of children features. Given a hierarchy E consisting of child-parent edges, $\mathrm{C}_p(E)$ returns the children of p in E:

$$\mathrm{C}_p(E) = \{c \mid (c,p) \in E\} \quad (4)$$

We now define G_M as the MUTEX-groups represented as sets of child-parent edges given the hierarchy E:

$$G_M = \bigcup_{p \in \mathcal{F}} \big\{ g \times \{p\} \mid g \in \text{max-cliques}\big(\text{subgraph}(M, \mathrm{C}_p(E))\big) \big\} \quad (5)$$

where $\text{subgraph}(G,V)$ returns the subgraph containing the vertices V and any edges between elements of V in G, and $\text{max-cliques}(G)$ returns the set of maximal cliques in the undirected graph G, ignoring trivial cliques of size one since they do not contribute to feature groups.

In our example, we get two overlapping MUTEX-groups: $\{(\text{cpu\_hotplug}, \text{cpu\_freq}), (\text{performance}, \text{cpu\_freq})\}$ and $\{(\text{performance}, \text{cpu\_freq}), (\text{powersave}, \text{cpu\_freq})\}$.

An XOR-group defines a $[1\,..\,1]$ cardinality among its members and thus, impose a stronger constraint than MUTEX-groups. XOR-groups can be recovered by checking an additional condition on a MUTEX-group. Taking the set of MUTEX-groups, we check each if given the presence of its parent, at least one element in the group must also be present. The XOR-groups G_X is defined:

$$G_X = \big\{ \{(f_1,p), \ldots, (f_k,p)\} \in G_M \mid \Phi \models p \to (f_1 \vee \cdots \vee f_k) \big\} \quad (6)$$

Now let $G_{M'} = G_M - G_X$ be groups that are strictly MUTEX and not XOR. Given a group $g \in G_{M'} \cup G_X$, the cardinality mapping C is defined:

$$C(g) = \begin{cases} (0,1) & \text{if } g \in G_{M'} \\ (1,1) & \text{if } g \in G_X \end{cases} \quad (7)$$

$G_{M'}$ and G_X are maximal (no group subsumed by other) and complete (none missing) with respect to the constructed hierarchy and input dependencies. $G_{M'}$ contains all maximal MUTEX-groups since it uses the maximal cliques algorithm. G_X inherits the maximality of MUTEX-groups but further constrains it by enforcing a lower bound. Thus, G_X contains all possible XOR-groups—a feature cannot be added or removed without violating the $[1\,..\,1]$ cardinality constraint.

However, the maximality and completeness of $G_{M'}$ and G_X may cause groups to overlap (i.e. a feature may belong to one or more feature groups), a property disallowed by the well-formedness rules of a feature diagram. Similar to the hierarchy building, we present users with all detected groups so that they can decide which to maintain in the feature diagram. Groups that are not overlapping are kept in the diagram. Any groups that are not kept will remain as excludes edges or cross-tree constraints.

Implies and Excludes Edges.

The final components of the feature diagram—implies and excludes edges—describe the remaining implications and exclusions that are not represented in the feature hierarchy or as a feature group.

The implies edges can be thought of as the edges from child to parent candidates that were not selected to be in the hierarchy. Let G be the implication graph and E be the constructed hierarchy, the implies edges I are $I = \mathrm{E}(G) - E$.

Similarly, excludes edges are the edges that were not chosen to be represented as a MUTEX or XOR-group. We define members$(\cdot)$ to return the members of a group. For example, let $g = \{(f_1, p), \ldots, (f_k, p)\}$ then members$(g) = \{f_1, \ldots, f_k\}$. Given a mutex graph M, feature groups G, the excludes edges X is defined: $X = \mathrm{E}(M) - \bigcup_{g \in G} \text{members}(g) \times \text{members}(g)$.

The Cross-Tree Formula.

Using the procedures described in this section yields a feature model that is complete—all valid configurations in the input dependencies are valid configurations of our feature model. However it may still allow some configurations that were disallowed by initial dependencies. To address this we can add our input dependencies as a cross-tree formula to make our feature model sound. In practice, to reduce redundancy in the cross-tree formula, we add only the clauses in Φ that are not already entailed by the diagram.

Incomplete Dependencies.

A MUTEX-group with members $f_1, \ldots, f_k$ is detected if there exists a clique between $f_1, \ldots, f_k$ in the mutex graph. A clique in the mutex graph requires each member to have an exclusion to every other member. If any exclusions between the members are missing in the case the dependencies are incomplete, a MUTEX-group with k members will be detected as one with less than k members.

XOR-groups, on the other hand, contain two dependencies. First requirement is that an underlying MUTEX-group exists between the group members. Second, the XOR-group requires an implication from the group's parent to its members (Equation 6). If either the first dependency is incomplete or if the second dependency is missing, then the XOR-group will be detected simply as a MUTEX-group of equal or lesser size.

4.3 Implementation

A prototype implementation is available as an open-source project[1]. Computation of the implication and mutex graphs, which can be done offline, took about 3 and 7 hours for Linux and eCos respectively on a 2.40GHz Intel Core2 Duo machine. The computation of RIFs and RAFs (needed for user interaction) is instantaneous. The graph algorithms use SAT4J for checking implications and exclusions. We used the Bron-Kerbosch algorithm [8] for finding all maximal cliques in the mutex graph.

5. EVALUATION

We evaluate our procedures on input from Linux, eCos and FreeBSD. For Linux and eCos, we extract our input data from their existing reference feature models [7], which gives us two samples with complete dependencies and extensive descriptions. FreeBSD, on the other hand, does not have a reference feature model. For this project, we evaluate our procedures against data extracted from the FreeBSD codebase, giving us a sample with incomplete dependencies and partial descriptions. We believe that FreeBSD is representative of the projects that will require our procedures for reverse engineering. Since FreeBSD lacks a reference model, we created one manually for a subset of features.

Our evaluation criterion is to check if for every feature, its parent in the reference feature model is one of the top parent candidates in the RIFs and RAFs. We consider the parent-child relations in the reference feature models to be the best choices possible because these models were built over ten years by their respective development communities.

We also measure the effect of incomplete dependencies and descriptions by progressively removing dependencies and descriptions from the Linux and eCos data. We see that prioritizing direct implications has a significant impact on the effectiveness of our procedures with incomplete descriptions.

Finally, we evaluated our procedure for recovering feature groups in the presence of complete and incomplete data.

5.1 Experiment Input Data Characteristics

We construct our input data using the reference models of the x86 Linux 2.6.28.6 and the i386pc feature model for eCos 3.0. Dependencies were extracted by applying a translation of their formal semantics to propositional formulas [6, 15]. Feature names and descriptions were extracted directly from the reference models themselves. We have placed the translation tools online as open-source projects[2,3].

Figure 6 characterizes the inputs for Linux, eCos, and FreeBSD. Linux has 5321 features, while eCos has 1245, and as shown in Figure 6, the distribution of number of words are only slightly different. The majority of features in eCos have 10 to 40 words. Linux, on the other hand, has a large number of features with no descriptions, while the rest have roughly 20 to 60 words. The distributions of direct and transitive implications are significantly different. Almost all features in Linux have from 60 to 80 transitive implications, while in eCos the variation is much larger. For direct implications, Linux's features have from 0 to 10 implications, and again, eCos has a larger variety. In Section 5.2, we will see if and how the distributions affect the results of our procedures.

Unlike Linux and eCos that have a configuration tool, configuring the FreeBSD's kernel involves creating a large text file that lists selected features—such as devices and CPU options—and their values. Various boilerplate templates are available to the user as default configurations. There is no explicit description of legal combinations of features.

We extracted the input for our procedure from the FreeBSD 8.0.0 codebase. We included the codebase for all nine supported hardware architectures (unlike for Linux and eCos, where we used architecture specific models). The 1203 features were mined by hand-crafted parsers from LINT templates—maximal configurations that contain all options supported by the kernel. Since these templates often contain feature descriptions in a semi-structured way, we also created heuristics-based fuzzy parsers to extract them.

Dependencies were extracted from various sources. We manually derived around 100 dependencies that were present in feature descriptions, but the majority of dependencies were extracted from source code. We examined the codebase to

[1] gsd.uwaterloo.ca/reverse-engineering-feature-models

[2] code.google.com/p/linux-variability-analysis-tools

[3] code.google.com/p/variability/wiki/CDLTools

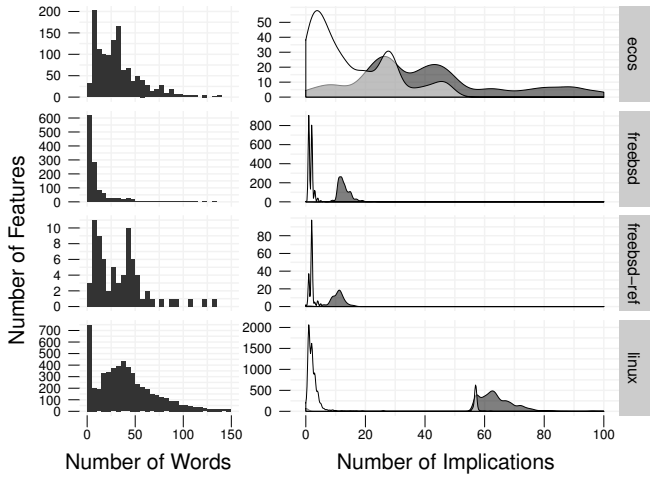

Figure 6: Characterization of descriptions, transitive implications (dark grey) and direct implications (white) for eCos, FreeBSD, the FreeBSD reference model, and Linux

find statements relevant for extracting feature dependencies. For example, dependencies between device drivers were specified by FreeBSD-specific preprocessor macros, making such dependencies easy to extract. However, most constraints had to be extracted by using a more comprehensive static analysis infrastructure. This infrastructure is one of our ongoing projects, but we use its early results for this work. The analysis infrastructure derives constraints by analyzing C source files and exploiting three types of information: #error preprocessor macros (*build error analysis*), the location of feature references (*feature liveness analysis*) and the definition and use of identifiers (*def/use analysis*).

From our experience with extracting dependencies from FreeBSD, we learned that although part of the process can be implemented as an automatic tool that works across different projects, there was a significant amount of manual work required to capture project-specific patterns from the build system. However, the project-specific component once constructed, can be beneficial for other applications such as iteratively checking the dependencies of a project against its feature model as the project evolves. We estimate it took one person, one month of effort to build our project-specific component for FreeBSD.

FreeBSD Reference Model.

In order to be able to assess the quality of the model synthesized by our tools, we created a reference feature model of 90 features by manually analyzing the FreeBSD kernel artifacts. We started by performing domain analysis on documentation and architecture to produce an ontology[4]. The ontology comprises of 192 features with several domain-specific relationships. The reference feature model was derived by taking the parts of the ontology that we felt were most developed and correct. The feature hierarchy was built by traversing generalization and composition relationships (cf. [10]). The resulting feature model mainly covers technical aspects of the kernel, such as tracing, monitoring and debugging. Structurally, 21% of its features are mandatory; 24% participate in cross-tree constraints; and three XOR-groups that bundle 12% of the features. Creating the ontology and deriving the feature model took about two person weeks.

Figure 6 shows that the number of implications per feature in the FreeBSD reference model is very representative of the number of implications of all features in FreeBSD. However, the reference model has a larger proportion of long descriptions (over 10 words) to short descriptions (less than 10 words) than all features in FreeBSD; in this aspect, it is more similar to Linux and eCos.

[4]code.google.com/p/variability/wiki/FreeBSDOntology

5.2 Effectiveness of Parent Heuristics

We evaluate our parent heuristics on complete input with Linux and eCos and on incomplete input with Linux, eCos and FreeBSD.

Our evaluation criterion for the RIFs is whether the reference parent as defined in the reference model, appears in the top five positions of our RIFs list. We feel the top five results are a reasonable number for a user to review and to select the correct choice.

To evaluate the RAFs, we calculate the percentage of all features users will need to examine to have a 75% chance of finding the feature's parent. This measure the effectiveness of only the ranking heuristic, in the absence of any dependencies.

Complete input.

For each feature, we record the position of the reference parent in the RIFs list. We find that 76% of features in Linux and 79% of features in eCos have their reference parent within the top 5 results.

We consider these results show that the heuristics are typically successful at identifying the correct parent. However, in our evaluation subjects, a significant number of features have root as their parent. Since root is not in the RIFs list, it can never be found within the top five results, skewing the statistics to our disadvantage. Still we have decided not to include the root in the RIFs. Nesting under root (or on top-level) is qualitatively a different decision than nesting features anywhere else. Our tools do not guide the reader in any way to nest under root, but we consider this decision to be much easier to make then detailed nesting of small granularity features deep in the hierarchy. If we only consider features that are not children of root, we observe that as many as 90% of them in Linux and 81% in eCos have their reference parent within the top five results. For this reason, the following diagrams (Figures 7, 8) omit top-level features.

Incomplete input.

When reverse engineering feature models in real-world situations, we cannot assume completeness of the input dependencies and descriptions. The RAFs list, which ranks all-features, can be used to identify potential parents when the RIFs are incomplete. Common to both the RIFs and RAFs is our ranking heuristic that depends on the descriptions of features—as descriptions shrink in size, so does the effectiveness of our heuristic.

We evaluate the effects of incomplete data and the robustness of our RIFs by randomly selecting subsets of implications and words from descriptions for Linux and eCos. For RAFs, we randomly select just words from the descriptions since implications are not used.

Figure 7 shows how the RIFs performs as we reduce the

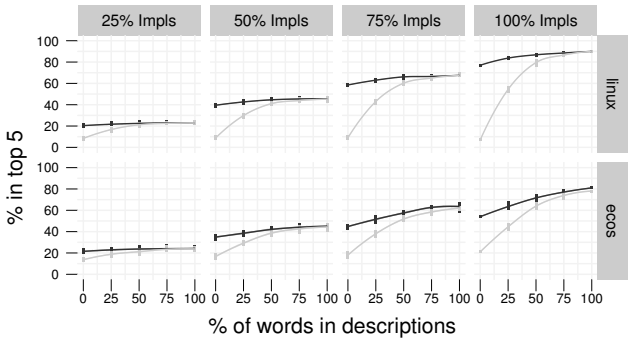

Figure 7: Robustness of RIFs for the *prioritizing* (black) and *non-prioritizing* (gray) orders under complete and incomplete data

number of implications and words of descriptions for the Linux and eCos data. We created sets of samples where we progressively removed random implications and words, forming sets with 25%, 50%, 75% and 100% of implications and words from descriptions. We repeated the experiment 10 times for each combination to assure robustness of results.

Figure 7 shows that results linearly degrade as we remove dependencies. We also observe that larger descriptions significantly improve results, particularly from 0 to 50% descriptions where the gain is most significant. The figure also shows the effect of our order where direct implications are prioritized (in black) over the non-prioritizing variant (grey). We see that the prioritizing order is more effective than the non-prioritizing variant overall. Furthermore, the prioritizing order is particularly effective on Linux where the prioritized results are still relatively high even as the descriptions approach zero.

On FreeBSD, we find that 35% of features are missing an implication to their reference parent in its dependencies. As a result, these features do not have their reference parent in their RIFs. For the remaining 65%, the reference parent is contained in the top 5 results of the RIFs for all features. 84% of features have their reference parent in the first or second positions of the RIFs.

Regarding RAFs, Figure 8 shows the top fraction of all features users will need to examine to have a 75% likelihood of finding the reference parent on Linux and eCos. For this experiment, we formed datasets containing no descriptions, 25%, 50%, 75% and 100% of randomly selected words from descriptions. We measure the robustness of our ranking heuristic with respect to the size of descriptions.

If we look at the 50% mark, the user needs to examine approximately 10% of all features in Linux and eCos. As descriptions increase in size, we see that the user needs to examine fewer and fewer features, reaching only 3% of all features in Linux and 6% in eCos when we have complete descriptions (100%). FreeBSD has results similar to Linux, with the user needing to examine 3% of all features to attain a 75% likelihood of finding the reference parent.

5.3 Feature Groups

The reference Linux feature model had 5 MUTEX-groups, a total of 24 XOR-groups and 1 OR-group with a total of 526 features participating in groups. We identified all 5 MUTEX-groups and found 23 of the XOR-groups. One XOR-group

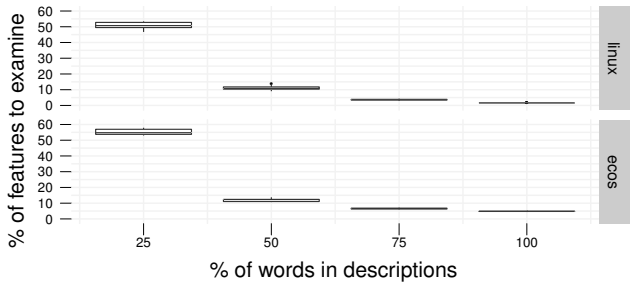

Figure 8: The top RAFs needed for a user to have a 75% chance of finding the reference parent under complete and incomplete descriptions

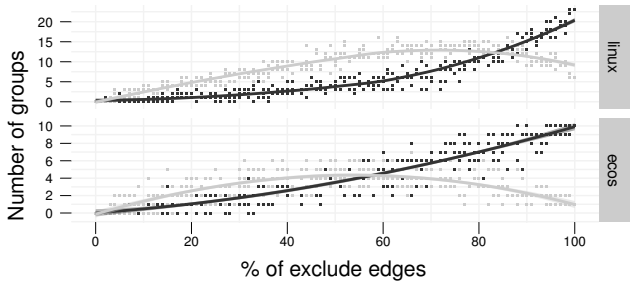

Figure 9: Reference groups detected as *xor*- (black) or *mutex*-groups (grey) under complete and incomplete exclusions

was not found due to the presence of a dead feature in its members. We were unable to find the OR-group since our procedure lacks support for finding such groups.

The reference feature model of eCos had only a single MUTEX-group, 11 XOR-groups and no OR-groups, with 44 features participating in groups. The MUTEX-group and 10 of the XOR-groups were discovered by our procedure. The XOR-group that was not detected was in fact, dead. The group required a package that was not present in the eCos i386 model that we analyzed.

The reference FreeBSD feature model had three XOR-groups. We correctly identified one group in its entirety. Parts of a second group were detected as MUTEX-groups and a third group was not detected at all due to the incompleteness of our dependency data. The FreeBSD data was incomplete as we saw in the evaluation for the hierarchy building—roughly a third of features did not imply their parents of the reference model.

Incomplete Input.

Feature groups rely on the presence of exclusions in the mutex graph to determine its size and members. As we have seen in our FreeBSD evaluation, incomplete dependencies affects the detection of feature groups. Here, we evaluate the effect of incomplete exclusions on the Linux and eCos data. Figure 9 shows the number of feature groups from the respective reference models that are detected as an XOR-group, or as a MUTEX-group of equal or smaller size depending on the completeness of exclusions. We randomly selected exclusions ranging from 0 to 100%, with 100% being all present. Each point is an individual sample and the lines indicate the fitted curve across all the sample points.

As the number of exclusions increases, the number of MUTEX and XOR-groups detected increases, with the former growing at a much faster rate. In the presence of incomplete dependencies, an XOR-group may be detected as a MUTEX-group of equal or smaller size. Only in the case that all exclusions among members of an XOR-group are present is it actually detected as one. We observe this effect at roughly 70% of exclusions in Linux and 50% in eCos—the number of MUTEX-groups starts a downward trend and the XOR-groups begin to grow at an increasing rate. This means that the input for our procedure should contain at least about 50% of exclusion dependencies to warrant that group synthesis reasonably precisely distinguishes the two kinds of groups.

We also observe different curves between the two systems due to their characteristics—Linux has 30 groups over 526 features while eCos has 12 groups over 44 features. With the larger number of features participating in groups in Linux, we see it takes more exclusions in Linux before XOR-groups are detected when compared to eCos.

6. THREATS TO VALIDITY

External threats.

Our procedures assume that feature names, descriptions, and dependencies can be extracted from a project. Our evaluation on FreeBSD shows that it is possible to extract such input from existing software projects. Crafting the project-specific component for FreeBSD required significant work. However, the effort required for this component may vary significantly depending on the project.

As we apply our procedures to Linux, eCos and FreeBSD, one might perceive that our procedures are tailored for the OS domain. However, our procedures are general and can be applied to systems in other domains. Notably, all three systems are of considerable complexity and size and differ significantly in their input characteristics.

As stated before, feature names in Linux and eCos often reflect the hierarchy of the reference models. Thus, one could conjecture that we obtain good results for these systems because the reference hierarchy is re-discovered by our similarity metric from the feature names. However, our similarity measure performs similarly well on FreeBSD, even though the system does not come with a feature model and their feature names follow a different convention.

Internal threats.

We evaluate results for FreeBSD on a reference feature model that the third author created. This creates a threat of potential bias, since the author knew the procedures that were to be evaluated against this model. Also, it is possible that the reference model is different from what a domain expert would create. To address these problems, we used an entirely different approach to build the feature model [10], which required building an ontology first. The ontology represents the domain in more detail than a feature model would represent, effectively forcing the modeler to become an expert in the modeled fragment of the domain. Another threat is that the subset of features in the reference model may not be representative of the entire system. We compared both in Figure 6, observing that the subset of features had a similar distribution in its number of implications to the distribution for all features. However, the features in the subset tend to have longer descriptions than the rest of FreeBSD's features; still, the distribution is similar to that of Linux and eCos. Thus, while applying our procedures on all features of FreeBSD would likely produce worse results than for the subset, the results for the subset—in particular, the need to review 3% of RAFs to have a 75% chance of finding the right parent—are consistent with those for Linux.

We have not run user experiments to evaluate the effort saved by our procedures. Such experiments involve usability while we focus on the reverse engineering algorithms in this paper. These evaluations and incorporating our procedures into modeling tools is future work.

7. RELATED WORK

The procedures presented in this paper build on our previous work on synthesizing feature diagrams from logical formulas [11] that produces a non-standard feature diagram with a DAG structure instead of a tree. We reuse implication graphs and their transitive reductions for hierarchy building and also the basic concept for group detection from that work. However, the present approach addresses several significant shortcomings. First, given a feature, [11] effectively proposes all directly implied features as parent candidates, without any ranking. This approach has two problems. First, 683 features in Linux have more than 40 directly implied features, meaning that the user has to review 30 features to have 75% chance of finding the parent. We address this problem by using similarity to rank the candidates, reducing 30 to five (6-fold improvement). Further, features may imply their parents indirectly, as is the case for cpu_hotplug in Figure 1. We found 677 features in Linux imply their parents indirectly, which makes the approach of [11] inherently incomplete. We address the problem by including all implied features in RIFs. Interestingly, 82% of features in Linux with indirectly implied parents have three or less directly implied features, thus allowing our approach to include them in the top five results. Second, the approach relies on dependencies alone and performs poorly if the dependencies are incomplete.

Unlike [11], our procedures do not detect OR-groups. Unfortunately, the algorithms of [11] do not scale to models as large as in our evaluation. Furthermore, OR-groups were rare in our systems, with only Linux having just one. We continue to work on more efficient techniques for identifying OR-groups [3], and will report progress on these in future.

Janota et al. [12] propose an interactive editing environment that takes the DAG structure from the synthesis work [11] and allows the user to decide interactively among parent and group candidates. The key difference is that our approach also incorporates textual descriptions in order to rank the potential choices presented to the user, addressing the shortcomings stemming from relying only on dependencies. Furthermore, our procedures are significantly more scalable, handling upwards of 5000 features, while the previous approach is reported to handle at most 200–300 features.

Snelting applied formal concept analysis to extract concept lattices from C code representing the logical dependencies among code blocks enclosed in `#ifdef` statements [18]. These lattices were then used to guide refactoring of source code. A concept lattice is equivalent to a reduced implication graph in our setting. Our approach has several significant differences. First, we aim to select a single tree from the implication graph rather than visualizing entire lattice (which was the case in our previous work [11]). Second, we also detect the

other components of a feature model—namely, feature groups, implies and exclude edges. Third, Snelting's approach relies on the logical dependencies only.

Alves et al. [2] and Niu et al. [14] apply information retrieval techniques to abstract requirements from existing specifications of a given domain into a feature model. As such, these works are related to our use of a text similarity to rank parent candidates. In particular, Alves et al. use text similarity, specifically Latent Semantic Analysis (LSA) and Vector Space Model (VSM) with cosine function as the similarity metric. Our approach resembles the VSM, but uses a special similarity metric that can handle very short descriptions where already one shared word is significant. Niu et al. define similarity over attributes of abstract functional requirements called functional requirements profiles (FRPs). The FRPs and their attributes needs to be extracted manually from the requirements, incurring significant effort.

One key difference from our work is that both related works use clustering techniques—over the similarity metrics they define—to generate a single feature hierarchy. We have experimented with clustering to produce a single hierarchy, coming to the conclusion that there is simply not enough information in the input descriptions and dependencies to decide a single, desirable hierarchy—such as the one from a reference model—without additional expert input.

Another key difference is that our approach incorporates precise logical dependencies from other sources of information such as a build system or source code. This increases the quality of the final outcome and allows for higher level of automation, such as the automatic detection of feature groups and cross-tree dependencies. Interestingly, Alves' approach has been further extended by Weston et al. [19] with identification of grammatical patterns, like alternative enumerations or conjunctive enumerations, to distinguish optional and mandatory dependencies. However, the modeler has to incorporate the new variability information manually into the resulting feature model.

Finally, both related approaches have been used to create feature models with dozens rather than thousands of features.

8. CONCLUSIONS

Software product line developers and users benefit from having feature models that describe the variability of software systems. However, building such a model from an existing system requires considerable effort where the key challenge is the selection of a parent for every feature. We have presented a ranking heuristic for identifying parent candidates and also automated procedures for identifying mandatory features, feature groups and implies/excludes edges.

We show that the procedures work well for Linux and eCos where the input data is complete, and also for FreeBSD, a system with incomplete descriptions and dependencies. By leveraging both dependencies and descriptions, the RIFs list contains the correct parent for 76% of features in Linux and 79% in eCos in its first five positions. If we omit dependencies and rely strictly on our ranking heuristic (RAFs), the user only needs to examine 3% to 6% of all features to find the parent in most cases. We also recovered all MUTEX and XOR-groups for Linux and eCos assuming the reference hierarchy was selected by the user.

This work contribute towards the reverse-engineering of feature models from large-scale projects. However, in order for these procedures to be incorporated into practical tools, there are many open problems in the areas of feature location and dependency mining that remain to be solved.

9. 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, C. Schwanninger, L. Barbosa, A. Rashid, P. Sawyer, P. Rayson, C. Pohl, and A. Rummler. An exploratory study of information retrieval techniques in domain analysis. In *SPLC*, 2008.

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

[4] D. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6), 2010.

[6] T. Berger and S. She. Formal semantics of the CDL language. Technical Note. Available at www.informatik.uni-leipzig.de/~berger/cdl_semantics.pdf.

[7] 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.

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

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[10] K. Czarnecki, C. H. P. Kim, and K. Kalleberg. Feature models are views on ontologies. In *SPLC*, 2006.

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

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

[13] 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, 1990.

[14] N. Niu and S. M. Easterbrook. On-demand cluster analysis for product line functional requirements. In *SPLC*, 2008.

[15] S. She and T. Berger. Formal semantics of the Kconfig language. Technical Note. Available at eng.uwaterloo.ca/~shshe/kconfig_semantics.pdf.

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

[17] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. Is The Linux Kernel a Software Product Line? In *SPLC-OSSPL*, 2007.

[18] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *TOSEM*, 1996.

[19] N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC*, 2009.

[20] R. Zippel and contributors. kconfig-language.txt. available in the kernel tree at kernel.org, seen 2009-11/23.