On Extracting Feature Models from Sets of Valid Feature Combinations

Evelyn Nicole Haslinger, Roberto Erick Lopez-Herrejon, and Alexander Egyed

Systems Engineering and Automation,
Johannes Kepler University
Linz, Austria
{evelyn.haslinger,roberto.lopez,alexander.egyed}@jku.at
http://www.jku.at/sea/

Abstract. Rather than developing individual systems, Software Product Line Engineering develops families of systems. The members of the software family are distinguished by the features they implement and Feature Models (FMs) are the de facto standard for defining which feature combinations are considered valid members. This paper presents an algorithm to automatically extract a feature model from a set of valid feature combinations, an essential development step when companies, for instance, decide to convert their existing product variations portfolio into a Software Product Line. We performed an evaluation on 168 publicly available feature models, with 9 to 38 features and up to 147456 feature combinations. From the generated feature combinations of each of these examples, we reverse engineered an equivalent feature model with a median performance in the low milliseconds.

Keywords: Feature, Feature Models, Feature Set, Reverse Engineering, Software Product Lines, Variability Modeling.

1 Introduction

Commercial software systems usually exist in different versions or variants, this can be due, for instance, to requirement changes or different customer needs. *Variability* is the capacity of software artifacts to change [1] and *Software Product Line Engineering (SPLE)* is a software development paradigm which helps to cope with the increasing variability in software products. In SPLE the engineers develop families of products rather than designing the individual products independently. SPLE practices have shown to significantly improve productivity factors such as reducing costs and time to market [2]. At the core of SPLE is a *Software Product Line (SPL)* [2], which represents a family of software systems. These systems are distinguished by the set of features they support, where a *feature* is an increment in program functionality [3]. The de facto standard to model the common and variable features of an SPL and their relationships are *Feature Models (FMs)* [4], that express which feature combinations are considered valid products (i.e. SPL members).

However, companies typically do not start out building a feature model or SPL. Rather they build products and, if successful, variations of that product for different customers, environments, or other needs. A scenario, which is becoming more pervasive and frequent in industry, is to reverse engineer such product variations into an SPL. The first essential step in this process is obtaining a feature model that correctly captures the set of valid feature combinations present in the SPL. Constructing such feature model manually is time intensive and error prone. Our work is a complement to the current research in this area which assumes the existence of artifacts with variability already embedded from which an FM could extracted. In our earlier work [5], we presented an algorithm to address this problem on basic feature models. This paper extends this work by also considering feature models that can contain basic *Cross Tree Constraints (CTCs)*, that is, *requires* and *excludes* CTCs [6]. Furthermore, we also performed a more comprehensive evaluation. We used 168 publicly available feature models, with 9 to 38 features and up to 147456 feature combinations. From the generated feature combinations of each of these examples, we reverse engineered an equivalent feature model with a median performance in the low milliseconds.

2 Background and Running Example

This section provides the required background information about variability modeling with feature models and related basic technology.

2.1 Feature Models in a Nutshell

Feature models are commonly used in SPLE to define which feature combinations are valid products within an SPL. The individual features are depicted as labeled boxes and are arranged in a tree-like structure. There is always exactly one root feature that is included in every valid program configuration. Each feature, apart from root, has a single parent feature and every feature can have a set of child features. These child-parent relationships are denoted via connecting lines. Notice here that a child feature can only be included in a program configuration if its parent is included as well. There are four different kinds of relations in which a child (resp. a set of children) can interrelate with its parent:

- If a feature is *optional* (depicted with an empty circle at the child end of the relation) it may or may not be selected if its parent feature is selected.
- If a feature is *mandatory* (depicted with a filled circle at the child end of the relation) it has to be selected whenever its parent feature is selected.
- If a set of features forms an *inclusive-or* relation (depicted as filled arcs) at least one feature of the set has to be selected if their parent is selected.
- If a set of features forms an *exclusive-or* relation (depicted as empty arcs) exactly one feature of the set has to be selected if their parent is selected.

Besides the child-parent relations there are also so called *cross-tree constraints* (*CTC*), which capture arbitrary relations among features. *Basic CTCs* are the *requires* and *excludes* relation. If feature A *requires* feature B, then feature B has to be included whenever feature A is included. If two features are in an *excludes* relation then these two features cannot be selected together in any valid product configuration. The *requires* and *excludes* CTCs are the most commonly used in FMs; however, more complex CTCs can be expressed using propositional expression, for further details see [7].

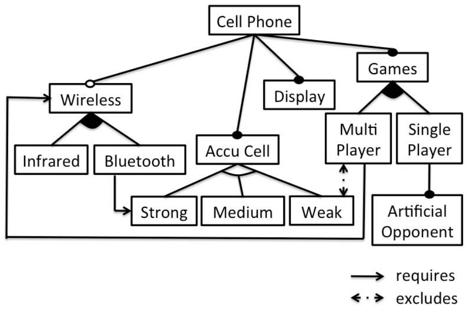


Fig. 1. Cell Phone SPL Feature Model

Table 1. Feature Sets of Cell Phone Software Product Line

P	C	W	I	B	A	S	M	We	D	G	Mu	Si	Ar
p1	✓				✓	✓			✓	✓		✓	✓
p2	✓				✓		✓		✓	✓		✓	✓
p3	✓				✓			✓	✓	✓		✓	✓
p4	✓	✓		✓	✓	✓			✓	✓		✓	✓
p5	✓	✓		✓	✓	✓			✓	✓	✓		
p6	✓	✓		✓	✓	✓			✓	✓	✓	✓	✓
p7	✓	✓	✓		✓	✓			✓	✓		✓	✓
P8	✓	✓	✓		✓	✓			✓	✓	✓		
p9	✓	✓	✓		✓	✓			✓	✓	✓	✓	✓
p10	✓	✓	✓		✓		✓		✓	✓		✓	✓
p11	✓	✓	✓		✓		✓		✓	✓	✓		
p12	✓	✓	✓		✓		✓		✓	✓	✓	✓	✓
p13	✓	✓	✓		✓			✓	✓	✓		✓	✓
p14	✓	✓	✓	✓	✓	✓			✓	✓		✓	✓
p15	✓	✓	✓	✓	✓	✓			✓	✓	✓		
p16	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓

Figure 1 shows the feature model of our running example, an SPL of cell phones inspired by a model of the SPLOT homepage [8]. Feature `Cell Phone` is the root feature of this feature model, hence it is selected in every program configuration. It has three mandatory child features (i.e. the features `Accu Cell`, `Display` and `Games`), which are also selected in every product configuration as their parent is always included. The children of feature `Accu Cell` form an exclusive-or relation, meaning that the programs of this SPL include exactly one of the features `Strong`, `Medium` or `Weak`. The features `Multi Player` and `Single Player` constitute an inclusive-or, which demands that at least one of these two features is selected in any valid program configuration. `Single Player` has `Artificial Opponent` as a mandatory child feature. Feature `Wireless` is an optional child feature of root, hence it may or may not be selected. Its child features `Infrared` and `Bluetooth` form an inclusive-or relation, meaning that if a program includes feature `Wireless` then at least one of its two child features has to be selected as well. The Cell Phone SPL also introduces three CTCs. While feature `Multi Player` cannot be selected together (`excludes`) with feature `Weak`, it cannot be selected without feature `Wireless`. Lastly feature `Bluetooth` `requires` feature `Strong`.

2.2 Basic Definitions

Definition 1. *Feature List (FL) is the list of features in a feature model.*

The FL for the Cell Phone FM is `[Cell Phone, Infrared, Bluetooth, Strong, Medium, Weak, Multi Player, Single Player, Display, Games, Artificial Opponent, Accu Cell, Wireless]`.

Definition 2. *Feature Set (FS) is a 2-tuple [sel,$\overline{sel}$] where sel and $\overline{sel}$ are respectively the set of selected and not-selected features of a member product. Let FL be a feature list, thus sel, $\overline{sel} \subseteq FL$, $sel \cap \overline{sel} = \emptyset$, and $sel \cup \overline{sel} = FL$. The terms p.sel and p.$\overline{sel}$ respectively refer to the set of selected and not-selected features of product p*[1].

Definition 3. *Feature Sets Table (FST) is a set of feature sets, such that for every product p_i we have that $p_i.sel \cup p_i.\overline{sel}$=FL, where FL is a feature list of the corresponding SPL. Let $FST_{FL'}$ denote the clipping of FST that only contains features in feature list FL′, i.e. $FST_{FL'} = \{FS \mid \exists\, FS' \in FST : FS'.sel \cap FL' = FS.sel \wedge FS'.\overline{sel} \cap FL' = FS.\overline{sel}\}$ and let FST_f denote the subset of FST that contains only feature sets in which feature f is selected.*

Table 1 shows the 16 valid feature sets defined by the feature model in Figure 1. Throughout the paper we use as column labels the shortest distinguishable prefix of the feature names (e.g. `We` for `Weak`). An example of a feature set is product `p1 =[{C, A, S, D, G, Si, Ar}, {W, I, M, We, Mu}]`. `p1` is a valid product because none of the constraints imposed by the Cell Phone FM is violated.

Definition 4. *Atomic set is a group of features that always appears together in all products [6]. That is, features f_1 and f_2 belong to an atomic set if for all products p_i, $f_1 \in p_i.sel$ iff $f_2 \in p_i.sel$ and $f_1 \in p_i.\overline{sel}$ iff $f_2 \in p_i.\overline{sel}$. Let atSet be an atomic set, we denote $\overbracket{atSet}$ as an arbitrarily chosen representative feature of the atomic set, and $\widetilde{atSet}$ the remaining non-representative features in atSet.*

For example, in the feature sets of Table 1, features `Si` and `Ar` form an atomic set `atSet`. A representative can be $\overbracket{atSet}$ =`Si` and $\widetilde{atSet}$={`Ar`}. Both features always appear together in the products of Table 1, e.g. while product `p1` includes features `Si` and `Ar`, product `p15` does include neither of them.

Definition 5. *Smallest Common Product. Let S be a set of feature sets. Product $p_i \in S$ is a smallest common product of S iff $\forall\, p_j \in S : |p_i.sel| \leq |p_j.sel|$.*

Product `p1` includes seven features, it is a smallest common product as there exists no product in Table 1 that includes less features.

For our reverse engineering algorithm two different kinds of graphs are constructed to represent information contained in the input FST, the implication and mutex graph, both are specializations of feature graphs.

Definition 6. *A Feature Graph FG is an ordered pair (`V,E`) where `V` is a set of features (i.e. `V` $\subseteq$ `FL`) representing the vertices of the graph and `E` is a set of tuples of the form $(f_1,f_2) \in V \times V$, where $(a,b) \in E$ denotes that there is an edge from feature a to feature b.*

[1] Definition based on [6].

We say *feature `f` implies feature `f'`* in `FST`, whenever the proposition holds that `f'` is included whenever `f` is selected. An implication graph is used to summarize which features imply which other features in the input FST.

Definition 7. *Implication Graph IG. Let fst be an FST where all non-representative features have been removed and let fl be the feature list of fst. The IG of fst and fl is a feature graph that contains an edge from feature f_1 to feature f_2 if f_1 implies f_2 in FST, where IG does not contain any transitive connections between f_1 and f_2 i.e. $\forall f_1, f_2 \in fl : imp(f_1, f_2, fst) \wedge \neg path((f_1, f_2), IG \setminus \{(f_1, f_2)\}) \Rightarrow (f_1, f_2) \in IG$ holds for IG, where:*

- $imp((t_1, t_2), fst) \equiv \forall fs \in fst : t_1 \notin fs.sel \vee t_1 \in fs.sel \wedge t_2 \in fs.sel$
- $path((x, y), G) \equiv (x, y) \in G \vee (\exists x' : (x, x') \in G \wedge path((x', y), G)$.

The mutex graph stores which features are not selected together in any valid product configuration of the input FST. Section 3 provides algorithms to extract the mutex and implication graph from an FST.

Definition 8. *Mutex Graph. Let fst be an FST and let fl be the FL of fst. A Mutex Graph MG of fst and fl is a feature graph that contains an edge from feature f_1 to feature f_2 iff all feature sets in fst select at most one of these features i.e. $\forall f_1, f_2 \in fl : (\forall fs \in fst : \neg f_1 \in fs.sel \vee \neg f_2 \in fs.sel) \Leftrightarrow (f_1, f_2) \in MG$ holds for MG.*

Definition 9. *A feature map $M \subseteq FL \times \mathcal{P}(FL)$ maps a single feature to a set of features, i.e. $\forall (f_1, features_1), (f_2, features_2) \in M : f_1 = f_2 \Rightarrow features_1 = features_2$ holds. Let f be a feature, M.f denotes the set of features that f maps to.*

3 Reverse Engineering Algorithm

This section describes our reverse engineering algorithm that extracts from an input FST and its FL the corresponding feature model. We start by outlining the challenges introduced by also considering CTCs then we proceed by describing the overall procedure of our algorithm and its core auxiliary function `buildFM`.

3.1 Challenges Created by Considering CTCs

Our previous work proposed an algorithm to reverse engineer feature models without CTCs (i.e. basic feature models) from FSTs [5]. By also considering CTCs the extraction process gets more complex, because many of the observations used for extracting basic feature models no longer hold. The reason being that CTCs do not introduce new valid feature combinations, but instead they reduce their number.

Essentially there are three core issues that need to be resolved. While the IG of an FST that corresponds to a basic feature model always yields the correct child parent relations, this is not necessarily the case for an FST that corresponds to a feature model with basic CTCs. Also the extraction of exclusive-or relations gets more complex, because two sibling features that are never selected together in any valid product configuration can either be in an exclusive-or relation or in an *excludes* CTC. The third challenge is the extraction of optional features. Previously, we used the observation that all features that are not selected in any of the smallest common products formed by features with the same parent in IG are optional. This observation is not strong enough as soon as *requires* CTCs are considered, because a feature that is not selected in any of the smallest common products might be optional but it could as well be a feature that is just in a *requires* CTC with its parent in IG.

Algorithm 1. Feature Model Extraction

```
1: Input: A Feature Sets Table (FST), and Feature List (FL).
2: Output: A feature model FM.
3:
4: {Start building FM from common features}
5: $splCF := splWideCommon(FST)$
6: $f \in splCF$
7: $root := [f, splCF - \{root\}, \{\}, \{\}, \{\}]$
8: $FM := [root, \{\}, \{\}]$
9:
10: {Prunes FST by removing common features}
11: $FL' := FL - splCF$
12: $FST' := FST_{FL'}$
13:
14: {Computes atomic sets}
15: $atSets := compAtomicSets(FST', FL')$
16: {Prunes FST by removing atomic sets}
17: $FL'' := FL' - \widetilde{atSets}$
18: $FST'' := FST'_{FL''}$
19:
20: {Build Mutex and Implication Graph}
21: $IG := buildImplGraph(FST'', FL'')$
22: $MG := buildMutexGraph(FST'', FL'')$
23:
24: {Build Feature Model}
25: $FL''' := FL'' - \{root\}$
26: $FST''' := FST''_{FL'''}$
27: $buildFM(FST''', FL''', atSets,$
        $root, FM, IG, MG)$
28:
29: {Extract EXCLUDES CTCs}
30: $excludes = extractExclCTC($
        $FM, MG)$
31: $addConstraints(FM, excludes)$
32: return $FM$
```

3.2 Overall Procedure

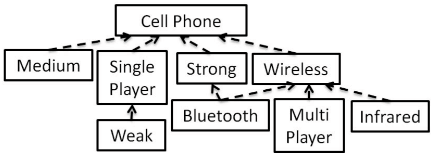
Algorithm 1 shows the overall procedure to extract feature models from an input FST and its corresponding FL. The data structure used to store the extracted model is a three-tuple of the form `[root, requires, excludes]`, where `root` is a feature model node, and `requires` and `excludes` are feature maps that respectively represent requires and excludes CTCs.

Definition 10. *A feature model node is a five-tuple of the form `[f, mand, opt, or, xor]`, where f is the feature represented by the node, mand is the set of mandatory child features of f, opt is a set of feature model nodes representing* *the optional child features of f and or (resp. xor) is a set of sets of feature model nodes representing the inclusive-or (resp. exclusive-or) relations among child features of feature f.*

Auxiliary Function 1. *`splWideCommon(fst)` computes from a Feature Sets Table the set of features that are common to all the members of the product line, i.e. features `f` such that for all products p_i∈`fst`, `f`∈p_i`.sel` holds.*

Line 5 calls `splWideCommon(fst)` which yields for our example the features `C`, `D`, `A` and `G`, as they are selected in every single product of Table 1. Subsequently Line 6 arbitrarily selects one of these features to be the root feature. Then Lines 7 to 8 initialize the feature model data structure.

Auxiliary Function 2. *`compAtomicSets(fl, fst)` computes the atomic sets in the feature sets of Feature Sets Table `fst` involving features in Feature List `fl`.*



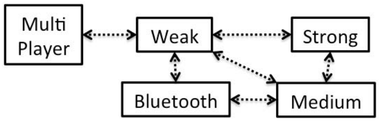


Fig. 2. Implication Graph of Cell Phone SPL

Fig. 3. Mutex Graph of Cell Phone SPL

For our running example $\overbracket{atSet}$ =`Si` and $\widetilde{atSet}$={`Ar`} is extracted as atomic set in Line 15. Lines 21 and 22 extract the implication and mutex graph as they are shown in Figure 2 and 3. To build up the implication graph the auxiliary function `buildImplGraph(fst, fl)` is used (see Algorithm 2). This function takes an FST where all non-representative features have been removed and its FL. Using these two inputs it extracts an implication graph as defined in Definition 7. The two auxiliary functions used by Algorithm 2 are defined as follows.

Auxiliary Function 3. *`independent(fst,fl)` returns all features in FL `fl` that do not imply any other feature in FST `fst`.*

Auxiliary Function 4. *`imply(fst,fl, f)` returns all features in FL `fl` that imply feature `f` in FST `fst`.*

The function `buildMutexGraph(fst, fl)`, that is described in Algorithm 3, extracts the mutex graph corresponding to an FST and its FL as defined in Definition 8.

Function `buildFM`, shown in Algorithm 4, traverses the implication graph `IG` from bottom to top and determines the types of relationships among sibling features, at the same time it also extracts the *requires* CTCs and inserts them into FM. Section 3.3 describes this function in more detail.

Auxiliary Function 5. *`extractExclCTC(FM, MG)` returns a feature map representing the excludes CTC. The function traverses the feature model `FM` to extract*

Algorithm 2. Build Implication Graph buildImplGraph(FST, FL)

```
1: Input: A Feature Sets Table (FST), and Feature List (FL).
2: Output: An Implication Graph IG.
3:
4: $IG := \{\}$
5: $f \in splWideCommon(FST)$
6: $FL' = FL - \{f\}$
7: $FST' = (FST_f)_{FL}$
8: $directChildren = independent(FST', FL')$
9:
10: for $f_{in}$ in $directChildren$ do
11:     $IG = IG \cup \{[f_{in}, f]\}$
12:     $descendants = imply(FST', FL', f)$
13:     $FST'' = (FST'_f)_{descendants}$
14:     $IG = IG \cup buildImplGraph(FST'', descendants)$
15: end for
16: return $IG$
```

Algorithm 3. Build Mutex Graph buildMutexGraph(FST,FL)

```
1: Input: A Feature Sets Table (FST), and Feature List (FL).
2: Output: A Mutex Graph MG.
3:
4: $MG = \{\}$
5: for $f$ in $FL$ do
6:     $mutex = \bigcap_{fs \in FST \wedge f \in fs.sel} fs.\overline{sel}$
7:     if $| mutex | > 0$ then
8:         $MG = MG \cup (\{f\} \times mutex)$
9:     end if
10: end for
11:
12: return $MG$
```

a mutex graph `MG'`*. Tuples that are elements of* `MG` *(which has been extracted from the input FST) but not element of* `MG'` *are extracted as excludes CTCs.*

The last step of our reverse engineering algorithm is to extract the *excludes* CTCs (see Lines 30 to 31 in Algorithm 1). Consider Figure 7, it depicts the FM that has been extracted by `buildFM` and that is shown in the bottom part of Figure 7. Feature `Strong` is in an exclusive-or relation with `Medium` and `Weak`, hence the tuples $\{$`(Strong, Medium)`, `(Strong, Weak)`$\}$ are added to MG'. Also *requires* CTCs have to be considered while `MG'` is built up, i.e. *"Bluetooth requires Strong"* in `FM` as `Strong` excludes `Medium` and `Weak`, feature `Bluetooth` cannot be selected together with these two features either. Figure 4 shows the complete mutex graph `MG'` extracted by `extractExclCTC(FM, MG)`. `(Multi Player, Weak)` is an element of `MG` (see Figure 3) but not of `MG'`, therefore the *excludes* CTC *Multi Player excludes Weak* is added to `FM`. Figure 5 shows the final feature model that has been extracted by our reverse engineering algorithm. Notice that in this case the extracted FM is different from the one used as our running example in Figure 1. Nonetheless they are both equivalent in the sense that both denote the same set of feature sets.

3.3 Build Feature Model

Algorithm 4 builds up the feature model tree using the extracted graphs (i.e. the implication and mutex graph), the atomic sets and an FST. The implication

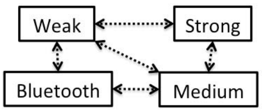


Fig. 4. Mutex Graph extracted from the FM shown in Figure 7

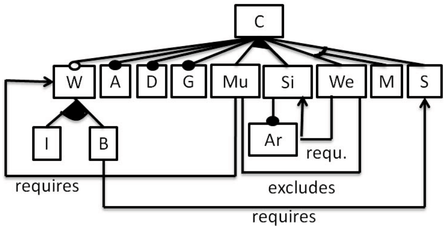


Fig. 5. Extracted FM

graph `IG` gives hints on the tree-structure of the feature model to extract, i.e. a feature is guaranteed to reach its parent in the implication graph, it is not guaranteed though that a feature is directly connected with its parent. For instance feature `Multi Player` has a direct connection to feature `Wireless` (which is not its parent feature) in IG due to the CTC *Multi Player requires Wireless*. BuildFM traverses `IG` from bottom to top (see Lines 5 to 11).

Algorithm 4. Build Feature Model buildFM

```
1: Input: A Feature Sets Table (FST), a Feature List (FL), atomic sets (atSets), a feature (parent), a feature model (FM) and two graphs (IG and MG).
2: Output: The modified feature model (FM).
3:
4: {Bottom to top traversal of implication graph}
5: for $f$ in $directChildren(parent, IG)$ do
6:    if $|\ descendants(f, IG)\ | > 0$ then
7:       $FST' := (FST_{node.f})_{descendants(f,IG)}$
8:       $stack.push(f)$
9:       $buildFM(FST',$
              $descendants(f, IG), atSets,$
              $f, FM, IG, MG)$
10:   end if
11: end for
12:
13: {Keep only columns that are direct children of parent}
14: $FL' = directChildren(parent, IG)$
15: $FST' := FST_{FL'}$
16:
17: {Add XOR relations and compute reduced FST}
18: $xors = insertXors(FST', FL', parent,$
       $FM)$
19: $FL'' = FL' - xors$
20: $FST'' := getSmallestProduct(FST'_{FL''})$
21:
22: {Add optional relations and compute reduced FST}
23: $opts = insertOptionals(FST'', FL'',$
       $parent, FM, IG)$
24: $FL''' = FL'' - opts$
25: $FST''' := getSmallestProduct$
       $(FST_{FL'''})$
26:
27: {Add OR relations}
28: $insertOrs(FST''', FST', FL, parent,$
       $FM, IG)$
29:
30: $stack.pop()$
31: return
```

The feature list `FL` contains all features that are descendants of `parent` in `IG`, meaning that these features are either descendants of `parent` in the feature model to extract or they are in a *requires* CTC with `parent`.

Auxiliary Function 6. *`descendants(f, IG)` returns all features `f'` that can reach feature `f` in `IG`, i.e. path((f',f), IG) holds.*

Auxiliary Function 7. *`directChildren(f, IG)` returns all features `f'` that are connected to feature `f` in IG i.e.* $(f', f) \in IG$.

One of our core observations is that the relationships among sibling features can be extracted by only considering the valid combinations among them, these combinations are calculated in Line 15. Next the exclusive-or relations are extracted (see Line 18) using the auxiliary function `insertXors`.

Auxiliary Function 8. *`insertXors(FST, FL, MG, parent, FM, IG)` inserts distinct subsets* $xor_i \subseteq FL$ *as to be in an exclusive-or relation for which holds that all members of* xor_i *exclude each other and at least one of them is selected in every feature set of FST, i.e.:* $\forall f_1 \in xor_i, f_2 \in xor_i : (f_1, f_2) \in MG \wedge \forall fs \in FST : xor_i \cap fs.sel \neq \{\}$.

Notice here that the proposition $xor_i \cap fs.sel \neq \{\}$ in Auxiliary Function 8 ensures that two features that are only in an *excludes* CTC are not extracted as to form an exclusive-or relation.

Auxiliary Function 9. *`getSmallestProducts (FST)` returns an `FST'` that contains only the smallest common products of `FST`.*

An example of the use of this function is in Line 20 of Algorithm 4.

Auxiliary Function 10. *`insertOptionals(FST, FL, parent, FM, IG)` processes possible optional features, i.e. all features `f` that are not selected in any FS of FST and do not imply their sibling feature or descendant, i.e.* $(\forall fs \in FST : f \in fs.\overline{sel}) \wedge descendants(parent, IG) \cap FM.requires.f = \varnothing$ *holds, where FST contains only smallest common products. If `f` is a true optional feature it is inserted as such, otherwise it is pushed one level upwards in IG.*

Connections between two features exist in `IG` either due to *requires* CTC or child-parent relations. Features that do only have a connection in `IG` to the current `parent` due to a *requires* CTC are possible optional features. For a true optional feature `f` holds that each valid feature combination containing `f` is still valid if `f` is deselected not considering any features that imply `f`, i.e. $\forall fs \in fst : f \in fs.sel \Rightarrow \exists fs' \in fst : (fs.sel \setminus \{f\}) \setminus implies = fs'.sel \setminus implies$, where `implies` is the set of features that imply feature `f` in the input FST. Function `insertOptionals` checks this property and is used in Line 23, note here that this is the only auxiliary function that operates on the complete input FST of Algorithm 1. The last kind of relation that need to be extracted are inclusive-or relations. Before we do this we again change the considered clipping of the input FST (see Lines 24 to 25).

Auxiliary Function 11. *`insertOrs(FST, FST', FL, parent, FM, IG)` extracts disjoint subsets of `FL` as to form inclusive-or relations and inserts them into feature model `FM`.*

Let `n` be the number of included features in a smallest common product ($fs \in FST, n = |fs.sel|$), then `n` yields the number of inclusive-or relations to extract.

Features that are selected together in the products of FST have to be in different inclusive-or relations. The features in FL that are selected in the products of FST, are grouped into n disjoint subsets, where two features that are selected together in any of the products in FST are put into different subsets.[2] Each of these subsets ss_i represents a possible inclusive-or relation, but only those ss_i are inserted as to be in an inclusive-or relation for which holds: $\forall fs \in FST' : fs.sel \cap ss_i \neq \varnothing$.[3]

The remainder of this section describes how buildFM proceeds on the running example. During the first call of buildFM the variable parent is equal to C, Line 5 yields the set {M, Si, S, W} as direct children of C in IG. Lets choose feature W as the first direct child that is processed by the loop in Line 5 As the direct children of W (i.e. I, B and Mu) do not have descendants there are no further recursive calls for these features.

Line 15 reduces the input FST, in our example the set of descendants of W is equal to its direct children, hence FST is equal to FST'. The features I, B and Mu do not exclude each other, hence Line 18 does not extract any exclusive-or relations. Line 20 extracts the smallest common products in FST' yielding FST''= $\{[\{B\}, \{I, Mu\}], [\{I\}, \{B, Mu\}]\}$. Line 23 pushes feature Mu one level upwards in IG inserting the CTC *"Multi Player requires Wireless"*. Mu is a possible optional feature, i.e. it is not selected in any of the products in FST'' and there exists no *requires* CTC with any of its sibling features, but Mu is not a true optional feature. Consider for instance product p11 in Table 1, *p11.sel* = *{C, W, I, A, Me, D, G, Mu}*. If W was an optional feature then a product $p11'.sel = p11.sel \setminus \{Mu\}$ should also exist in Table 1 as this is not the case W cannot be optional.

Auxiliary function insertOrs extracts features I and B as to be in an inclusive-or relation. Figure 6 depicts the extracted Feature Model and the modified feature graph IG after the first recursive call of buildFM.

The second recursive call is performed for feature Si and its descendant feature We. Once again auxiliary function insertOptionals yields that We is not a true optional feature, hence it pushes We one level upwards in IG and inserts the *requires* CTC *"Weak requires Single Player"*.

As no further recursive calls are required, buildFM now extracts the relations among the direct children of root, i.e. features W, Si, Mu, S, M and We.

Function insertXor extracts the features S, M and We as to form an exclusive-or relation. Subsequently Line 20 calculates FST''= $\{[\{Si\}, \{W, Mu\}]\}$ that contains only the smallest common products formed by features W, Mu and Si.

Feature W is extracted as optional feature because it is neither selected in any of the products in FST'' nor does it require one of its sibling features. As feature Mu implies feature W it cannot be an optional feature. FST''' is equal to

[2] Please refer to [9] for an explanation how we deal with features that are not selected in any of the products in FST.

[3] If there are features that could not be inserted, this means that possibly no equivalent model is extracted. Features for which the correct place in the feature model could not be found will be pushed one level upwards in the implication graph, if the current parent is the root feature then these features are inserted as optional child features.

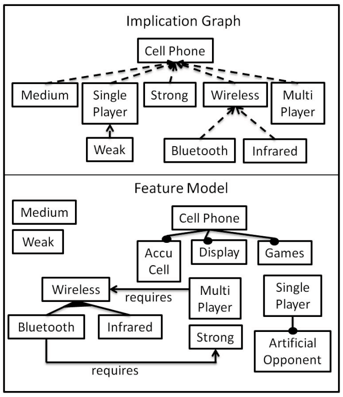


Fig. 6. `IG` and `FM` after first recursive call of `buildFM`

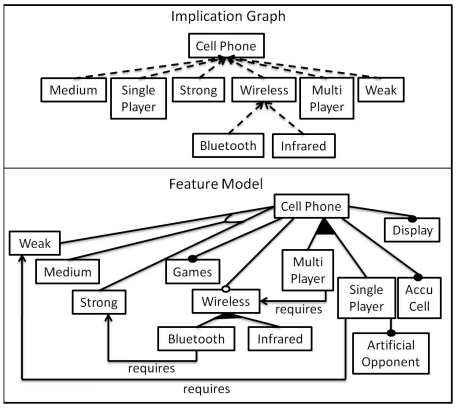


Fig. 7. `IG` and `FM` after call of `buildFM`

$\{[\{Si\}, \{Mu\}], [\{Mu\}, \{Ri\}]\}$, it contains the smallest common products formed by `Mu` and `Si`. Using `FST''' insertOrs` inserts an inclusive-or relation among these two features. Figure 7 shows the extracted feature model and the modified implication graph after the call of `buildFM`.

4 Evaluation

We evaluated our approach with 168 FMs publicly available from the SPLOT Homepage [8], for which the FAMA tool suite [10] was able to generate the corresponding FSTs that were then used as input to our reverse engineering algorithm[4]. The size of the FSTs described by one of these models ranges from 1 to 147456 products with an average of 1862.2 and a median of 62 products. The number of features is between 9 and 38. Of these 168 models, 69 have *requires* or *excludes* CTCs. The average number of CTCs of these 69 models is 3.12, the median is 2. The model with the most CTCs has 6 *excludes* and 11 *requires* CTCs and describes 1042 products. We executed our examples on a Windows 7 Pro system, running at 3.2Ghz, and with 8 GB of RAM. Figure 8 depicts the execution times of this evaluation. Our timing analysis shows that the average execution time is 1862.2ms and the median is 62ms, for the largest FST the execution of our reverse engineering algorithm takes only 12s. In each of these 168 cases our reverse engineering algorithm generated an equivalent model. To determine the equivalence of the input model and the reverse engineered model we used a procedure very similar to the one presented in [11].

[4] The code and feature models samples are available at: `http://www.jku.at/sea/content/e139529/e126342/e188736/`

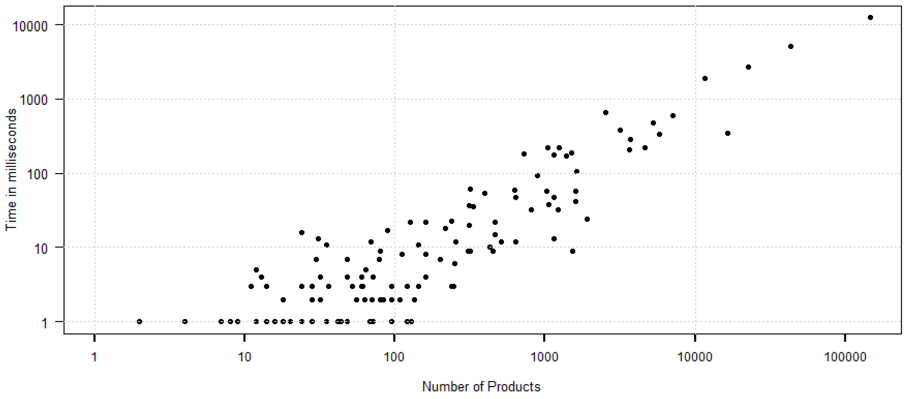


Fig. 8. Execution times of test runs

4.1 Algorithm Limitations

Each additional non-redundant CTC reduces the set of valid product configurations, however at some point it may no longer be possible to extract the hierarchy of the FM tree as too much information may be missing. Our algorithm does not cover circular *requires* CTCs, (e.g. if a feature `A` *requires* feature `B` and feature `B` *requires* feature `A`) or structural feature constructs which are equivalent to them [9]. In such case our reverse engineering algorithm produces an FM that is not equivalent to the input FST. However, we argue that such cases may indicate potential design flaws in the feature models that the software engineers should address. Our algorithm assumes that the input FST is complete, that is it contains *all* possible product configurations the FM should denote. Dealing with incomplete FSTs and incremental adaptation of existing FMs is part of our future work.

5 Related Work

This section outlines other approaches to reverse engineer feature models. Lopez-Herrejon et. al published an exploratory study on reverse engineering feature models using evolutionary algorithms [12], where *ETHOM (Evolutionary algoriTHm for Optimized feature Models)* was used for their implementation. Like our approach they also use a set of valid program configurations as an input to their algorithm. We see the advantage that they are theoretically able to reverse engineer more than one feature model that represents the input FST, which makes it possible to choose the model that contains the most meaningful feature hierarchy. Their evaluation showed though that they extracted in most cases feature models that are not equivalent to the input FST.

Acher et. al proposed a procedure to extract feature models from product descriptions [13]. Their reverse engineering algorithm operates on a slightly different perspective than ours does. They use semi-structured product descriptions instead of the mere set of valid feature combinations. These product descriptions are given in the form of tables where they produce a feature model for every row in the input tables. This is done through interpreting the values of the individual cells, e.g. if a cell contains the value "Plugin" then the corresponding feature is extracted as optional. Subsequently they merge the extracted feature models into a single model that their algorithm returns as output. Weston et. al introduce a framework that supports SPL engineers in constructing FMs from natural language requirements [14]. Their framework is used to determine the features of the SPL, to extract the tree structure among the extracted features and to differentiate between *mandatory* and *variant* features. She et. al present procedure to simplify the reverse engineering process of feature models [15]. To do this they use logic formulas as well as textual descriptions to make proposals to the user who then guides the extraction process. To extract the hierarchy of the FM tree they use an implication graph obtained from the logic formulas and similarity measures among the features of the SPL that have been extracted from the textual descriptions. Andersen et. al propose algorithms to reverse engineer feature models from propositional logic formulas, these formulas are either in disjunctive normal form (DNF) or conjunctive normal form (CNF) [16]. Note here that while an FST can be viewed as a propositional logic formula in DNF, our approach is very different from Andersen et. al's. While we use set operations to reverse engineer an FM from an FST their algorithms heavily rely on BDDs or rather SAT solvers. Moreover we want to emphasize here that Andersen et. al extracted for their evaluation feature graphs which need to be converted into feature models at a later time, either with the help of user input or an automated procedure described in their paper.

6 Conclusions and Future Work

Our evaluation shows that the proposed algorithm is able to reverse engineer feature models from input FSTs with a median execution time in the order of milliseconds. Our future work will address the following three issues. First, our current algorithm does not support circular *requires* CTCs. We plan to assess in practice if this scenario can indeed be characterized as a design error. Alternatively, we would extend the algorithm to cope with such cases. The second issue is a thorough scalability assessment. Our current evaluation is limited by FAMA's capability to generate feature sets. We expect to overcome this limitation with an approach sketched in [9]. FMs are not canonical, so there might be several non-identical FMs that represent the same FST. The third issue we plan to address is assessing the understandability of our reverse engineered models.

Acknowledgments. This research is partially funded by the Austrian Science Fund (FWF) project P21321-N15 and Lise Meitner Fellowship M1421-N15, and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965.

References

1. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw., Pract. Exper. 35(8), 705–754 (2005)
2. Pohl, K., Bockle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)
3. Zave, P.: Faq sheet on feature interaction, `http://www.research.att.com/pamela/faq.html`
4. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
5. Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: Reverse engineering feature models from programs' feature sets. In: Pinzger, M., Poshyvanyk, D., Buckley, J. (eds.) WCRE, pp. 308–312. IEEE Computer Society (2011)
6. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35(6), 615–636 (2010)
7. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
8. Software Product Line Online Tools, SPLOT (2010), `http://www.splot-research.org/`
9. Haslinger, E.N.: Reverse engineering feature models from program configurations. Master's thesis, Johannes Kepler University (2012), `http://www.sea.jku.at/ms/evelyn.haslinger`
10. FAMA Tool Suite (2012), `http://www.isa.us.es/fama/`
11. Thüm, T., Batory, D.S., Kästner, C.: Reasoning about edits to feature models. In: ICSE, pp. 254–264. IEEE (2009)
12. Lopez-Herrejon, R.E., Galindo, J.A., Benavides, D., Segura, S., Egyed, A.: Reverse Engineering Feature Models with Evolutionary Algorithms: An Exploratory Study. In: Fraser, G., Teixeira de Souza, J. (eds.) SSBSE 2012. LNCS, vol. 7515, pp. 168–182. Springer, Heidelberg (2012)
13. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., Lahire, P.: On extracting feature models from product descriptions. In: Eisenecker, U.W., Apel, S., Gnesi, S. (eds.) VaMoS, pp. 45–54. ACM (2012)
14. Weston, N., Chitchyan, R., Rashid, A.: A framework for constructing semantically composable feature models from natural language requirements. In: Muthig, D., McGregor, J.D. (eds.) SPLC. ACM International Conference Proceeding Series, vol. 446, pp. 211–220. ACM (2009)
15. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) ICSE, pp. 461–470. ACM (2011)
16. Andersen, N., Czarnecki, K., She, S., Wasowski, A.: Efficient synthesis of feature models. In: de Almeida, E.S., Schwanninger, C., Benavides, D. (eds.) SPLC, vol. (1), pp. 106–115. ACM (2012)