Reverse Engineering Feature Models From Programs' Feature Sets

Evelyn Nicole Haslinger
Systems Engineering and Automation
Johannes Kepler University Linz,Austria
k0855482@students.jku.at

Roberto E. Lopez-Herrejon
Systems Engineering and Automation
Johannes Kepler University Linz,Austria
roberto.lopez@jku.at

Alexander Egyed
Systems Engineering and Automation
Johannes Kepler University Linz,Austria
alexander.egyed@jku.at

***Abstract*—Successful software is more and more rarely developed as a one-of-a-kind system. Instead, different system variants are built from a common set of assets and customized for catering to the different functionality or technology needs of the distinct clients and users. The Software Product Line Engineering (SPLE) paradigm has proven effective to cope with the variability described for this scenario. However, evolving a Software Product Line (SPL) from a family of systems is not a simple endeavor. A crucial requirement is accurately capturing the variability present in the family of systems and representing it with Feature Models (FMs), the de facto standard for variability modeling. Current research has focused on extracting FMs from configuration scripts, propositional logic expressions or natural language. In contrast, in this short paper we present an algorithm that reverse engineers a basic feature model from the feature sets which describe the features each system provides. We perform an evaluation of our approach using several case studies and outline the issues that still need to be addressed.**

***Keywords*-Feature; Feature Model; Feature Set; Software Product Line**

I. Introduction

An emerging trend in commercial software development is that successful products are more and more rarely developed and sold as a one-of-a-kind systems. Instead, different system variants are built from a common set of assets and customized for catering to the different functionality or technology needs of the distinct clients and users. Current market and technology drivers demand a disciplined yet flexible approach to maximize reuse and customization in all the software artifacts used throughout the entire development cycle. The *Software Product Line Engineering (SPLE)* paradigm has proven effective to cope with this demand as attested by extensive research and practice both in academia and industry [1]–[3]. The success of this paradigm lies at the effective management and realization of its *variability* – the capacity of software artifacts to vary [4].

At the core of SPLE is a *Software Product Line (SPL)* [1], [2], [5] which is a family of systems that share common functionality but also have variations tailored for distinct needs. In a SPL, each member product provides a different combination of *features* – increments in program functionality [6]. Each feature combination is called a *feature set.*

However, evolving a SPL from a family of systems is not a simple endeavor. A crucial requirement is accurately capturing the variability present in the family of systems and representing it with *Feature Models (FMs)* [1], [7], the de facto standard for variability modeling. Current research has focused on extracting FMs from configuration scripts, propositional logic expressions or natural language [8]–[10]. In contrast, we present an algorithm that reverse engineers a basic feature model from feature sets. We performed an evaluation using fourty five publicly available feature models. For these models, our algorithm produced a variability-correct feature model in the order of miliseconds. Finally, we describe the issues that need to be addressed.

II. Background and Running Example

In this section we provide the required background on variability modeling with feature models and related basic terminology.

A. Feature Models in a Nutshell

Feature models are the de facto standard to model the common and variable features of SPL and their relationships [1], [7]. Features are depicted as labeled boxes and are connected with lines to other features with which they relate, collectively forming a tree structure. A feature can be classified as: *mandatory* if it is part of a program whenever its parent feature is also part, and *optional* if it may or may not be part of a program whenever its parent feature is part. Mandatory features are denoted with filled circles while optional features are denoted with empty circles both at the child end of the feature relations denoted with lines. Features can be grouped into: *inclusive-or* relation whereby one or more features of the group can be selected, and *exclusive-or* relation where exactly one feature can be selected. These relations are depicted as filled arcs and empty arcs respectively.

Figure 1 shows the feature model of our running example, a hypothetical SPL of Video On Demand systems. The root feature of a SPL is always included in all programs, in this case the root feature is `VOD`. Our SPL also has feature `Play` which is mandatory, in this case it is included in all programs because its parent feature `VOD` is always included. Feature `Record` is optional, thus it may be present or not in our product line members. Features `Display` and `OS` are also mandatory. Features `TV` and `Mobile` constitute an exclusive-or relation, meaning that our programs can have either one

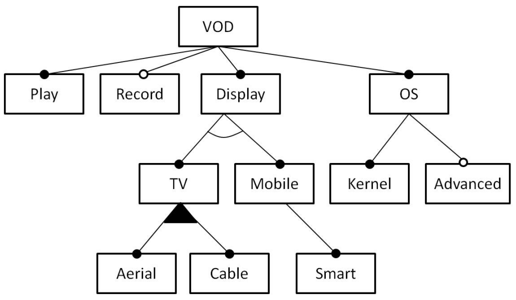


Fig. 1: Video On Demand SPL Feature Model

of them but only one. Features `Aerial` and `Cable` form an inclusive-or relation meaning that our systems can have either of the features or both of them, and feature `Smart` is mandatory with parent feature `Mobile`. Lastly, feature `Kernel` is mandatory while feature `Advanced` is optional with respect to their parent feature `OS`.

Besides the parent-child relations, features can also relate across different branches of the feature model in the so called *cross-tree constraints* [11]. The typical examples of this kind of relations are: *i) requires* relation whereby if a feature `A` is selected a feature `B` must also be selected, and *ii) excludes* relation whereby if a feature `A` is selected then feature `B` *must not* be selected. In a feature model, these latter relations are depicted with doted single-arrow lines and doted double-arrow lines respectively. Feature models without cross-tree constraints are refered to as *basic feature models.*

B. Basic definitions

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

Definition 2: Feature Set 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 (based on [11]).

Definition 3: Feature Set Table (FST) is a collection 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.

Table I shows the 16 possible feature sets described in our feature model in Figure 1. Throughout the paper we use as column labels the shortest distinguishable prefix of the feature names (e.g. `Ad` for `Advanced`). An example of a feature set is product `P2 =[{VOD, Play, Record, Display, OS, TV, Kernel, Aerial}, {Mobile, Smart, Advanced, Cable}]`.

Definition 4: Atomic set is a group of features that always appears together in all products [11]. 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.

TABLE I: Feature Sets of VOD Software Product Line

P	V	P	R	D	O	T	M	S	K	Ad	Ae	C
P1	✓	✓	✓	✓	✓	✓			✓	✓	✓	
P2	✓	✓	✓	✓	✓	✓			✓		✓	
P3	✓	✓		✓	✓	✓			✓	✓	✓	
P4	✓	✓		✓	✓	✓			✓		✓	
P5	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓
P6	✓	✓	✓	✓	✓	✓			✓	✓		✓
P7	✓	✓	✓	✓	✓	✓			✓		✓	✓
P8	✓	✓	✓	✓	✓	✓			✓			✓
P9	✓	✓		✓	✓	✓			✓	✓	✓	✓
P10	✓	✓		✓	✓	✓			✓	✓		✓
P11	✓	✓		✓	✓	✓			✓		✓	✓
P12	✓	✓		✓	✓	✓			✓			✓
P13	✓	✓	✓	✓	✓		✓	✓	✓	✓		
P14	✓	✓	✓	✓	✓		✓	✓	✓			
P15	✓	✓		✓	✓		✓	✓	✓	✓		
P16	✓	✓		✓	✓		✓	✓	✓			

For example, in the feature sets of Table I features `VOD`, `Display`, `Play`, `OS` and `Kernel` form an atomic set `atSet`. A representative can be $\overbracket{atSet}$ =`Display` and $\widetilde{atSet}$=`{VOD, Play, OS, Kernel}`. Another example of atomic set is formed by features `Mobile` and `Smart`, because both appear together in the last four products of Table I.

Definition 5: Smallest Common Product. Let S be a set of feature sets. Product p_i is the smallest common product of S iff $p_i.sel \subseteq p_j.sel$ for every product $p_j \in S$ with $p_i \neq p_j$.

III. Reverse Engineering Algorithm

In this section we sketch our reverse engineering algorithm. We start by providing a description of the auxiliary functions it relies on. Notice that we are assuming a pass-by-reference argument semantics.

- *splWideCommon(FST)*: computes from a Feature Set Table the list of features that are common to all the members of the product line, i.e. features `f` such that for all products $p_i \in$`FST`, and `f`$\in p_i$`.sel`.
- *selectRoot(FL)*: arbitrarily selects a feature from a Feature List to be the root of the feature model.
- *addMandRootFeatures(FL, root)*: creates a feature model with `root` as the feature model root and as mandatory children the features in FL.
- *reduceFeature(FL, FST)*: removes the features in the feature list FL from all the feature sets in FST. Pictorically, it removes from the table the columns of features FL.
- *computeAtomicSets(FL, FST)*: computes the atomic sets in the feature sets of FST involving features in feature list FL.
- *reduceAtomicSets(FST, atSets)*: eliminates the features in the feature sets of the FST that are non-representative elements of atomic sets in collection of atomic sets `atSets`.
- *dirChildren(FST, FL)*: selects features in feature list FL that are direct children of the current `parent`[1].

[1]No need to pass argument `parent` (input to Algorithm 2) because the feature sets in FST can only contain children of this parent

- *reduceProduct(FST)*: removes the repeated products in the feature set table FST. Two products $\texttt{p}_i$ and $\texttt{p}_j$ are repeated iff $\texttt{p}_i\texttt{.sel}$=$\texttt{p}_j\texttt{.sel}$ and $\texttt{p}_i.\overline{sel}$=$\texttt{p}_j.\overline{sel}$.
- *addXors(FST,FL, feature, atSets, FM)*: finds exclusive-or relations on the feature set table FST that involve features in feature list FL and have `feature` as the parent and adds them to feature model FM.
- *filterSmallestProducts(FST)*: divides features sets into disjoint subsets $\texttt{S}_1\texttt{..}\texttt{S}_k$ such that for every subset $\texttt{S}_i$ there exists a smallest common product $\texttt{p}_i$ with $\texttt{p}_i \in \texttt{S}_i$ and `k` is minimal (i.e. smallest number of subsets), and eliminates from the FST the products that are not the smallest common ones of their corresponding subset.
- *addOpts(FST, FL, feature, atSets, FM)*: finds optional features on the feature set table FST that involve features in feature list FL and have `feature` as the parent and adds them to feature model FM.
- *addOrs(FST, FL, feature, atSets, FM)*: finds inclusive-or relations on the feature set table FST that involve features in feature list FL and have `feature` as the parent and adds them to feature model FM.
- *descendants(FST, FL, feature)*: calculates the descendants of feature in a FST that involve features in feature list FL.

Let us explain now how our algorithm works. For simplicity, we have divided it in two parts. The first part, shown in Algorithm 1, sets up root and the features that are mandatory in all the products. The second part, shown in Algorithm 2, builds the feature model, top to bottom, in a recursive manner. For brevity, we omit details on the underlying data structures used for instance to represent the feature model.

Consider now our running example of VOD system. First, the features that are common to all products are identified (Line 5). In our example, these features are `VOD`, `Play`, `Display`, `OS`, and `Kernel`. A root is selected among this features (Line 6). To simplify our explanation, let us choose `VOD` also as root. Notice here that any other of these common features could be equally selected as root, yielding a different yet equivalent (i.e. same list of products [8]) feature model. Next, the root of the feature model and its mandatory children features are created (Line 9). Subsequently, the original feature set table (FST) is trimmed first to eliminate the common features already built into the feature model. A second reduction is made to remove the non-representative elements of the atomics sets (Lines 15-19). In our example, the only atomic set contains features `Mobile` and `Smart`. For simplicity, let us remove `Smart`. The last part (Lines 22-24) calls the recursive Algorithm 2 with the feature model so far constructed and the trimmed FST. Figure 2 summarizes the main execution steps of our running example. Figure 2(a) shows the FST and Figure 2(b) depicts the feature model constructed until this point. We also use the shortest distinguishable prefix of the feature names in the feature models and omit the surrounding boxes.

Algorithm 2 retrieves the relationships among the features of the basic feature model layer by layer. The idea is to extract at first all features which will be positioned directly beneath the current parent, which is the root during the first recursion

Algorithm 1 Feature Model Extraction

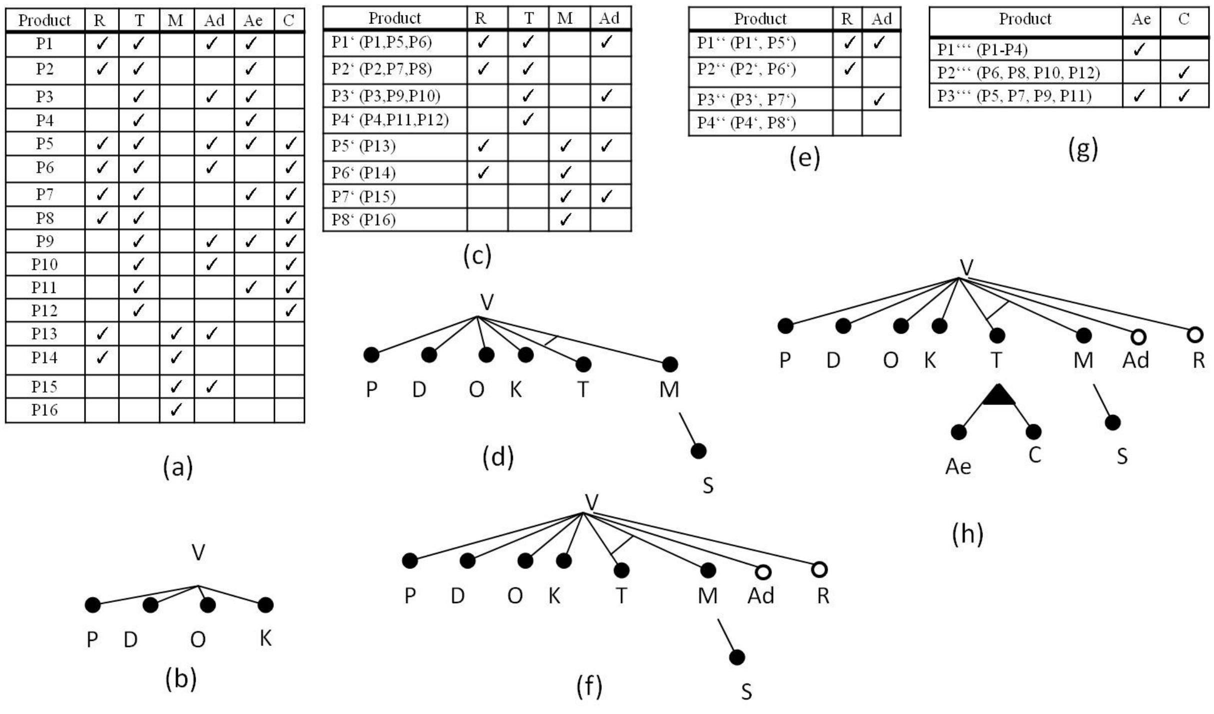
```
1:  Input: A Feature Sets Table (FST), and Feature List (FL).
2:  Output: A basic feature model FM.
3:
4:  {Computes SPL-wide common features and root}
5:  $splCF := splWideCommon(FST)$
6:  $root := selectRoot(splCF)$
7:
8:  {Starts building FM from common features}
9:  $FM := addMandRootFeatures(splCF, root)$
10:
11: {Prunes FST by removing common features}
12: $FST' := reduceFeature(splCF, FST)$
13:
14: {Computes atomic sets}
15: $FL' := FL - splCF$
16: $atSets := computeAtomicSets(FL', FST')$
17:
18: {Prunes FST by removing atomic sets}
19: $FST'' := reduceAtomicSets(FST', atSets)$
20:
21: {Build Feature Model}
22: $FL'' = FL' - atSets$
23: $buildFM(FST'', FL'', atSets, root, FM)$
24: return $FM$
```

step. Subsequently the relationships among these features are computed. The last step of Algorithm 2 is to make a recursive call, for all those direct children which are non-leaf features. Before we proceed, please notice that: *i)* Line 10 `copy` makes a fresh new copy of the direct children list, and *ii)* Line 27 a call to `reduceFeature` has its first argument indicated in brackets, this is a special case of this method that also deletes those products of FST that do not have `child` as feature.

During the first recursion the computation of the direct children (Line 4) delivers: `Record`, `TV`, `Mobile` and `Advanced`. Next, a copy of the given feature set table is produced which only contains direct children and no duplicate entries (Line 7). Figure 2(c) shows the outcome of this computation. Then the xor relations among the direct children are calculated (Line 11). Notice that `TV` and `Mobile` are inserted into the basic feature model FM during this step. As `Mobile` is one of the representatives of the atomic sets the non-representative feature `Smart` will be inserted as child mandatory of the feature `Mobile`, see Figure 2(d).

Line 12 again produces a reduced copy of the current feature sets table. `FST''` contains only four different products, namely the different combinations of `Record` and `Advanced` which appear in Figure 2(e). Then the computation of the smallest common products is performed (Line 15). In our example, there is only one smallest common product `psp=[{},{Record, Advanced}]`.

It is worth noticing that smallest common products are computed because all direct children which are not inserted



Product	R	T	M	Ad	Ae	C
P1	✓	✓		✓	✓	
P2	✓	✓			✓	
P3		✓		✓	✓	
P4		✓			✓	
P5	✓	✓		✓	✓	✓
P6	✓	✓		✓		✓
P7	✓	✓			✓	✓
P8	✓	✓				✓
P9		✓		✓	✓	✓
P10		✓		✓		✓
P11		✓			✓	✓
P12		✓				✓
P13	✓		✓	✓		
P14	✓		✓			
P15			✓	✓		
P16			✓			

(a)

Product	R	T	M	Ad
P1‘ (P1,P5,P6)	✓	✓		✓
P2‘ (P2,P7,P8)	✓	✓		
P3‘ (P3,P9,P10)		✓		✓
P4‘ (P4,P11,P12)		✓		
P5‘ (P13)	✓		✓	✓
P6‘ (P14)	✓		✓	
P7‘ (P15)			✓	✓
P8‘ (P16)			✓	

(c)

Product	R	Ad
P1‘‘ (P1‘, P5‘)	✓	✓
P2‘‘ (P2‘, P6‘)	✓	
P3‘‘ (P3‘, P7‘)		✓
P4‘‘ (P4‘, P8‘)		

(e)

Product	Ae	C
P1‘‘‘ (P1-P4)	✓	
P2‘‘‘ (P6, P8, P10, P12)		✓
P3‘‘‘ (P5, P7, P9, P11)	✓	✓

(g)

Fig. 2: Algorithm Main Execution Steps.

yet and do not appear in one of the smallest common products have to be optional features. In this example, the remaining direct children `Record` and `Advanced` are inserted into `FM` as optional features (Line 18), see Figure 2(f). Line 21 will not insert anything into FM, because there are no direct children left to process.

At last the recursive calls (Line 24-30) are made. In this example, only `TV` has descendants namely `Aerial` and `Cable`. The feature set table `FST′` (Line 27), with which the function is called recursively, contains three products as shown in Figure 2(g).

During the second call of Algorithm 2 the direct children are `Cable` and `Aerial` (Line 4). There are no xor relations inserted (Line 11) and the smallest common products are `psp1=[{Cable},{Aerial}]` and `psp2=[{Aerial},{Cable}]` (Line 15). There are no optional features inserted (Line 18). Finally the two features are inserted in an or relation (Line 21), Figure 2(h).

This example also illustrates that even though the feature model we obtained in Figure 2(h) is not identical to our original in Figure 1, on closer inspection, both models are equivalent in the sense that they describe the same collection of feature sets (i.e. Table I).

IV. Evaluation

We performed an evaluation of the execution time of our algorithm using 45 feature models publicly available from the SPLOT website [12]. These feature models range from 9 to 67 features with a median of 17 features. Using the FAMA tool suite [13], we computed the list of feature sets for each

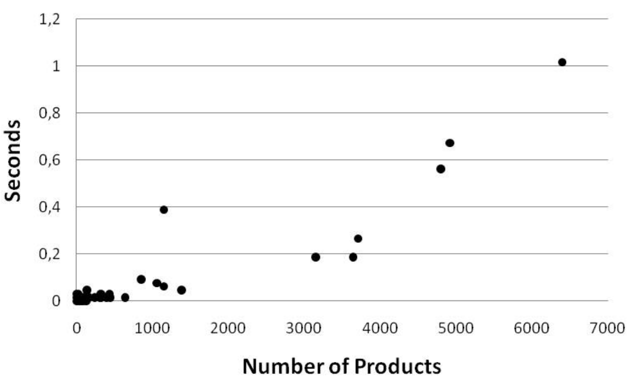


Fig. 3: Algorithm Execution Times

feature model (Operation `Products` in FAMA as defined in [11]) and use it as input to our algorithm. The number of feature sets in these models ranges from 1 to 6400 with a median of 135 feature sets. We executed our examples on a MS Windows XP system, running at 1.5Ghz, and with RAM of 2GB. The results are summarized in Figure 3. It can be seen that even for the largest model the performance falls within an acceptable range, about 1 second. For correctness, we performed a straightforward test. We computed the list of products of our reversed-engineered feature models, again using the FAMA tool suite, and compared them with the list of products originally used as input. In all cases, our algorithm produced an equivalent (i.e. same feature set list) feature model.

Algorithm 2 Build Feature Model buildFM

```
1: Input: A Feature Sets Table (FST), a Feature List (FL), atomic sets (aSets), a parent feature (parent), and a basic feature model (FM).
2: Output: The modified basic feature model FM.
3: {Computes direct children features}
4: $directChildren := dirChildren(FST, FL)$
5:
6: {Removes columns not in dirChildren}
7: $FST' := reduceProduct($
       $reduceFeature(FL{-}directChildren, FST))$
8:
9: {Adds Xor relations}
10: $directChildren' := copy(directChildren)$
11: $addXors(FST', directChildren, parent, atSets, FM)$
12: $FST'' := reduceProduct(reduceFeature($
       $directChildren' - directChildren, FST'))$
13:
14: {Retain smallest products}
15: $FST''' := filterSmallestProduct(FST'')$
16:
17: {Adds optional relations}
18: $addOpts(FST''', directChildren, parent, atSets, FM)$
19:
20: {Adds or relations}
21: $addOrs(FST''', directChildren, parent, atSets, FM)$
22:
23: {Recursive feature model building}
24: for $child\ in\ directChildren'$ do
25:     $FL' := descendants(FST, FL, child)$
26:     if $|FL'| > 0$ then
27:         $FST' := reduceProduct(reduceFeature($
               $< FL - FL', child >, FST))$
28:         buildFM(FST',FL', atSets, child, FM)
29:     end if
30: end for
```

V. Related Work

For sake of brevity, we describe only the work that most closely relate to ours. Czarnecki and Wasowski propose an algorithm to transform back and forth feature models and propositional logic formulas that relies on using Binary Decision Diagrams [8]. She et al. extend this work by taking into account not only the feature configuration dependencies but also the ontological (domain) knowledge of the feature descriptions [9]. A similar approach is taken by Weston et al. that uses natural language processing techniques to extract feature models from requirements documents [10].

VI. Conclusions and Future Work

In this paper we presented an algorithm to reverse engineer features models from feature sets, and performed a preliminary evaluation to assess its performance. Encouraged by our promising results, there are several issues that still need to be addressed. First and foremost, a thorough formal analysis of our algorithm. We are currently working on providing a formal footing to our algorithm along the lines of Czarnecki et al. [8]. A formal representation, would allow us to better assess the correctness of the derived feature models. We believe that works like Thüm et al. [14] and Batory's [15] could prove useful for this assessment.

Our current algorithm focuses on basic feature models, those that do not have cross-tree constraints. We are currently investigating how to integrate arbitrary cross-tree constraints. We have seen that a collection of feature sets can yield more than one equivalent feature model. This fact poses the question of what is role of the human guidance in this reverse engineering process. Like others ([9], [10]), we believe that reverse engineering feature models requires an iterative process that is lead by domain expertise (human guidance) informed with a variability analysis such as our algorithm. We plan to further explore this question with an actual case study.

Acknowledgment

This research is partially sponsored by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965.

References

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

[2] K. Pohl, G. Bockle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[3] F. J. van d. Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[4] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Softw., Pract. Exper.*, vol. 35, no. 8, pp. 705–754, 2005.

[5] D. S. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Trans. Software Eng.*, vol. 30, no. 6, pp. 355–371, 2004.

[6] P. Zave, "Faq sheet on feature interaction," http://www.research.att.com/ pamela/faq.html.

[7] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[8] K. Czarnecki and A. Wasowski, "Feature diagrams and logics: There and back again," in *SPLC*. IEEE Computer Society, 2007, pp. 23–34.

[9] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki, "Reverse engineering feature models," in *ICSE*, R. N. Taylor, H. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 461–470.

[10] N. Weston, R. Chitchyan, and A. Rashid, "A framework for constructing semantically composable feature models from natural language requirements," in *SPLC*, ser. ACM International Conference Proceeding Series, D. Muthig and J. D. McGregor, Eds., vol. 446. ACM, 2009, pp. 211–220.

[11] D. Benavides, S. Segura, and A. R. Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.

[12] "Software Product Line Online Tools(SPLOT)," 2011, http://www.splot-research.org/.

[13] "FAMA Tool Suite," 2011, http://www.isa.us.es/fama/.

[14] T. Thüm, D. S. Batory, and C. Kästner, "Reasoning about edits to feature models," in *ICSE*. IEEE, 2009, pp. 254–264.

[15] D. S. Batory, "Feature models, grammars, and propositional formulas," in *SPLC*, ser. Lecture Notes in Computer Science, J. H. Obbink and K. Pohl, Eds., vol. 3714. Springer, 2005, pp. 7–20.