Extracting Variability-Safe Feature Models from Source Code Dependencies in System Variants

Wesley K. G. Assunção[1,2], Roberto E. Lopez-Herrejon[3], Lukas Linsbauer[3], Silvia R. Vergilio[1], Alexander Egyed[3]
[1]DINF - Federal University of Paraná, CP: 19081, 81531-980, Curitiba, Brazil
[2]COINF - Technological Federal University of Paraná, 85902-490, Toledo, Brazil
[3]ISSE - Johannes Kepler University Linz, 4040 Linz, Austria
{wesleyk, silvia}@inf.ufpr.br, {roberto.lopez, lukas.linsbauer, alexander.egyed}@jku.at

ABSTRACT

To effectively cope with increasing customization demands, companies that have developed variants of software systems are faced with the challenge of consolidating all the variants into a Software Product Line, a proven development paradigm capable of handling such demands. A crucial step in this challenge is to reverse engineer feature models that capture all the required feature combinations of each system variant. Current research has explored this task using propositional logic, natural language, and search-based techniques. However, using knowledge from the implementation artifacts for the reverse engineering task has not been studied. We propose a multi-objective approach that not only uses standard precision and recall metrics for the combinations of features but that also considers variability-safety, i.e. the property that, based on structural dependencies among elements of implementation artifacts, asserts whether all feature combinations of a feature model are in fact well-formed software systems. We evaluate our approach with five case studies and highlight its benefits for the software engineer.

Categories and Subject Descriptors

I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*Heuristic Methods*

Keywords

Reverse Engineering; Feature Models; Multi-Objective Evolutionary Algorithms

1. INTRODUCTION

Software Product Lines (SPLs) are families of related software systems where each product has a different combination of features [3]. SPL practices have extensive and proven technological and economical advantages [19]. However, the effective adoption of SPLs poses several challenges. Salient among them is the reverse engineering of SPLs from existing software system variants, the most prevalent scenario in industry. A first requirement to tackle this challenge is extracting *feature models* that effectively describe the combinations of features present in the existing system variants. Current approaches rely on configuration scripts [17], propositional logic expressions [5, 18], natural language [20], ad hoc algorithms [1, 8, 9], and even search-based algorithms [12, 13, 14]. However, to the best of our knowledge, none of them exploit information on how the system variants are actually implemented or take a multi-objective perspective whereby different quality aspects of the reverse engineering task are simultaneously considered.

The driving motivation of our work is to extract feature models that are *variability-safe*, which means that, based on a concrete set of implementation artifacts, *all* the feature combinations of the variants denoted by a feature model are structurally well-formed, e.g. they do not have unresolved references to undefined elements. In this paper, we take a multi-objective perspective for reverse engineering feature models where the objective functions not only consider the recall and precision of the feature combinations of the obtained feature models, but also if they are variability-safe. The latter is computed from dependency information among elements of the source code artifacts that implement the system variants. The multi-objective approach that we propose gives software engineers the capacity to detect inconsistencies between the desired feature combinations and the implementation artifacts (e.g. infeasible products), and to analyze different trade-offs among the three objectives (e.g. offering more well-formed products than currently existing variants). We evaluated our approach with five case studies of different domains and dimensions. The results clearly illustrate the advantage as well as the feasibility of a multi-objective perspective for this vital reverse-engineering task.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
GECCO '15, July 11 - 15, 2015, Madrid, Spain
© 2015 ACM. ISBN 978-1-4503-3472-3/15/07…$15.00
DOI: http://dx.doi.org/10.1145/2739480.2754720

2. BACKGROUND

In this section we briefly present the basic background knowledge on feature models and introduce our running example which we use to illustrate the source code dependencies we consider in our reverse engineering approach.

2.1 Features Models

Feature Models (FMs) are a de facto standard for modelling the different combinations of features desired in an SPL [10]. Features are depicted as labelled boxes connected

by lines to other features they relate with, all collectively forming a tree-like structure.

A feature can either be *mandatory*, which is selected in a system whenever its parent feature is also selected, or *optional*, which may or may not be selected whenever its parent feature is selected. They are respectively represented with filled and empty circles at the end of the feature relation as shown in Figure 1(a) and Figure 1(b).

Features can be grouped into *alternative groups* where if the parent feature of the group is selected then *exactly* one feature from the group must be selected, or into *or groups* where if the parent feature of the group is selected then *one or more* features from the group can be selected. Feature groups are depicted with lines connecting the parent feature (`P`) with the group features (`C1`, `C2`, and `C3`), crossed by an empty arc for *alternative* groups and by a filled arch for *or* groups as illustrated in Figure 1(c) and Figure 1(d).

Besides the hierarchical relations among features, features can also relate across different branches of the feature model with *Cross-Tree Constraints (CTCs)* [4]. The most common types are *requires* relation whereby if a feature `A` is selected a feature `B` must also be selected, and *excludes* relation whereby if a feature `A` is selected then feature `B` *must not* be selected, and vice versa. These relations are commonly depicted, respectively, with a single and double-arrow dashed line as illustrated in Figure 1(e).

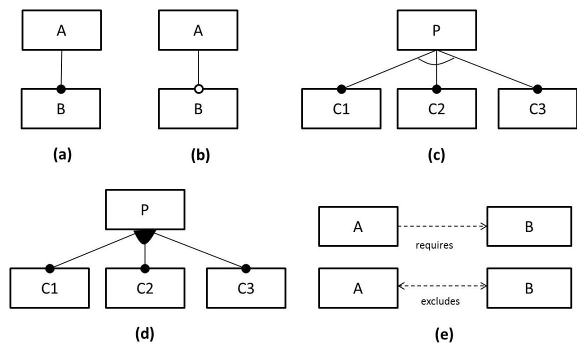


Figure 1: Feature Models Graphical Notation

2.2 Running Example

As our running example, let us consider a group of system variants of drawing applications which we would like to consolidate into a product line called *Draw Product Line (DPL)*. Each variant contains different combinations of the following features: the ability to handle a drawing area (`BASE`), draw lines (`LINE`), draw rectangles (`RECT`), select a color to draw with (`COLOR`), draw filled rectangles (`FILL`), and clean the drawing area (`WIPE`). Table 1 lists the 16 variants of DPL where the selected features are denoted with ✓, while the unselected features are left empty. We formally refer to each feature combination as a *feature set* defined as follows [12]:

Definition 1. ***Feature Set.*** *A feature set is a 2-tuple [*$\texttt{sel}$*,*$\overline{\texttt{sel}}$*] where* $\texttt{sel}$ *and* $\overline{\texttt{sel}}$ *are respectively the set of selected and not-selected features of a system variant. Let FL be the list of features of a feature model, such that* $\texttt{sel}$, $\overline{\texttt{sel}} \subseteq FL$, $\texttt{sel} \cap \overline{\texttt{sel}} = \emptyset$, *and* $\texttt{sel} \cup \overline{\texttt{sel}} = FL$.

Table 1: Feature Sets for DPL

Products	BASE	LINE	RECT	COLOR	FILL	WIPE
ProductX_1	✓	✓				✓
ProductX_2	✓	✓		✓		
ProductX_3	✓	✓	✓	✓		
ProductX_4	✓	✓	✓	✓	✓	
ProductX_5	✓	✓	✓			✓
ProductX_6	✓	✓				
ProductX_7	✓	✓		✓		✓
ProductX_8	✓	✓	✓			
ProductX_9	✓		✓			
ProductX_{10}	✓		✓			✓
ProductX_{11}	✓		✓	✓		
ProductX_{12}	✓		✓	✓	✓	
ProductX_{13}	✓		✓	✓		✓
ProductX_{14}	✓		✓	✓	✓	✓
ProductX_{15}	✓	✓	✓	✓		✓
ProductX_{16}	✓	✓	✓	✓	✓	✓

2.3 Source Code Dependency Graphs

Variability safety is the property of feature models that guarantees that *all* the variants that a feature model describes are structurally well-formed with respect to a set of concrete implementation artifacts, for example that they do not have unresolved references to undefined variables.

In this paper, we describe how information regarding dependencies among the code artifacts that implement the existing software variants can be exploited to reverse engineer variability-safe feature models. For this purpose, we rely on the terminology and tool support presented by Linsbauer et al. [7, 11]. Their work distinguishes two kinds of *modules* in the system variants:

Definition 2. ***Base Module.*** *A* base *module implements a feature regardless of the presence or absence of any other features and is denoted with the feature's name written in lowercase.*

Definition 3. ***Derivative Module.*** *A* derivative *module* $m = \delta^{\mathtt{n}}(\mathtt{c_0}, \mathtt{c_1}, ..., \mathtt{c_n})$ *implements feature interactions, where* c_i *is* `F` *(if feature* `F` *is selected) or* $\neg$`F` *(if not selected), and* n *is the order of the derivative.*

Linsbauer et al.'s extraction algorithm automatically produces a set of traces between both types of modules and the implementation artifacts (e.g. source code fragments) that implement them [11]. These code fragments can be of any granularity level, from entire classes down to individual statements. Let us consider the examples shown in Figure 2. This figure shows as comments some of the traces computed by our algorithm. Line 5 defines a field that traces to the base module of feature `Color`, which is defined in class `Canvas` regardless of the presence of other feature combinations in any product variant. As another example, consider method with signature $mousePressedLine(MouseEvent)$ in Line 7. The method header and most of its body trace to base module `Line`, meaning that these statements will be included in class `Canvas` whenever feature `Line` is included regardless of any other selected feature combination. Notice now that Line 10 traces to derivative module $\delta^1(Color, Line)$, which means this assignment statement will be part of its containing method whenever both features `Color` and `Line` are present in a system variant.

Definition 4. ***Module Expression.*** *A module expression is the propositional logic representation of modules. For*

```
1 public class Canvas ... {
2   List<Shape> shapes = new LinkedList<Shape>();
3   Point start;
4   Line newLine = null; // Line
5   Color color = Color.BLACK; // Color
6   ...
7   void mousePressedLine(MouseEvent e) { // Line
8    if (newLine == null) {
9       start = new Point(e.getX(), e.getY());
10       newLine=new Line(color,start);//δ^1(Color, Line)
11       shapes.add(newLine);
12    }
13   }
14 }
```

Figure 2: Source Code Snippet for DPL Example

a base module b, the module expression is its own literal b. For a derivative module $m = \delta^n(\mathtt{c_0}, \mathtt{c_1}, ..., \mathtt{c_n})$ *its module expression corresponds to* $c_0 \wedge c_1 \wedge ... \wedge c_n$.

For example, the module expression of base module `Line` is *Line*. As another example, consider module $\delta^1(\mathtt{Color}, \mathtt{Line})$, which expresses the interaction of features `Color` and `Line`, and is represented by the module expression $Color \wedge Line$.

Based on the output of Linsbauer et al.'s algorithm, our work uses dependencies between code fragments, such as method calls and field references, as described next.

DEFINITION 5. ***Dependency.*** *A dependency establishes a requirement relationship between two sets of modules and it is denoted with a three-tuple* `(from, to, weight)`*, where* `from` *and* `to` *each are a set of modules (or module expressions) of the related modules, and* `weight` *expresses the strength of the dependency, i.e. the number of dependencies of structural elements in modules* `from` *on structural elements in modules* `to`*. We use the dot (.) operator to reference elements of a tuple, e.g. the weight of a dependency* `dep` *is denoted by* `dep.weight`*. A dependency's propositional logic representation is defined as:*

$$\bigvee_{mfrom \in dep.from} mfrom \Rightarrow \bigwedge_{mto \in dep.to} mto$$

DEFINITION 6. ***Dependency Graph.*** *A dependency graph is simply defined as a set of dependencies, where each node in the graph corresponds to a set of modules (or module expressions), and every edge in the graph corresponds to a dependency as defined above. Edges are annotated with natural numbers that represent the dependencies' weights.*

Figure 3 shows the dependency graph of our running example DPL considering all its variants. The nodes are labelled with the corresponding trace's lowest order modules as they are the most important ones, higher order modules are not depicted to avoid clutter. Also for readability self-dependencies are not shown. Base modules are depicted with solid border while derivative modules are depicted with dashed border. It is worth noting that edges with the highest values tend to be those that go to base modules (e.g. `Fill` to `Rect`, or `Line` to `Base`), and that the core feature `Base` has the highest incoming values.

Let us now illustrate the propositional logic representation of dependencies using code snippet in Figure 2. As a first example, consider the dependency between module $\delta^1(Color, Line)$ and module *Color* that comes from the fact that `newLine = new Line(color,start);` in Line 10 which belongs to module $\delta^1(Color, Line)$ accesses the field `color` (Line 5) which belongs to module *Color*. The corresponding propositional logic expression that must hold for this dependency is $(Color \wedge Line) \Rightarrow Color$.

Another example, consider module $\delta^1(Color, Line)$ which depends on module *Line* because the same statement in Line 10 is contained in the method `public void mousePressedLine(MouseEvent e)` (Line 7) which belongs to module *Line*. The statement requires the method to be present, because otherwise the statement could not exist in isolation. The propositional logic representation is then $(Color \wedge Line) \Rightarrow Line$.

The dependency graph of the DPL can also be represented as a dependency matrix as shown in Table 2. Every row in the matrix represents a dependency, where the first column of the matrix is its ID number. The second and third columns are respectively the modules that depend on each other, i.e. the modules in column *from* require (i.e. depend on) the modules in column *to*. The weight of each dependency is presented in the fourth column. The fifth column presents the normalized weight values. The weights of the edges are normalized so that the sum of all weights in the dependency graph is 1. We use these normalized values in our multi-objective approach as described in the next section.

Notice that some of the propositional logic constraints derived from the dependencies (e.g. the 2 examples above) are tautologies that will always hold, like $(Color \wedge Line \Rightarrow Line) \Leftrightarrow TRUE$. This is not a mistake, but merely an indication that the implementation is consistent with the variability represented in the feature model.

Table 2: Dependency Matrix for DPL

ID	From	To	Weight	Normalized
1	Line	Base	21	0.1304
2	Wipe	Base	10	0.0621
3	Color	Base	19	0.1180
4	Rect	Base	20	0.1242
5	Fill	Base	17	0.1056
6	Fill	Rect	23	0.1429
7	Fill	Color	3	0.0186
8	Fill	δ^1(Rect, Color)	1	0.0062
9	δ^1(Rect, ¬Color)	Rect	12	0.0745
10	δ^1(Rect, Color)	Rect	3	0.0186
11	δ^1(¬Color, Line)	Line	8	0.0497
12	δ^1(Color, Line)	Color	1	0.0062
13	δ^1(Color, Line)	Line	10	0.0621
14	δ^2(Rect, Color, ¬Fill)	δ^1(Rect, Color)	1	0.0062
15	δ^2(Rect, Color, ¬Fill)	Rect	11	0.0683
16	δ^2(Rect, Color, ¬Fill)	Color	1	0.0062
		Total:	161	1.0000

3. A MULTI-OBJECTIVE PERSPECTIVE

In this section we describe and illustrate the objective functions that our approach employs which rely on information retrieval measures and that also exploit the information provided in the dependency graphs.

3.1 Objective Functions Definitions

We start by defining some auxiliary functions. In the following definitions, let $\mathcal{FM}$ denote the universe of feature models, $\mathcal{SFS}$ the universe of sets of feature sets, and sfs the initial set of feature sets defined by the software engineer (e.g. as shown in Table 1 for our DPL running example).

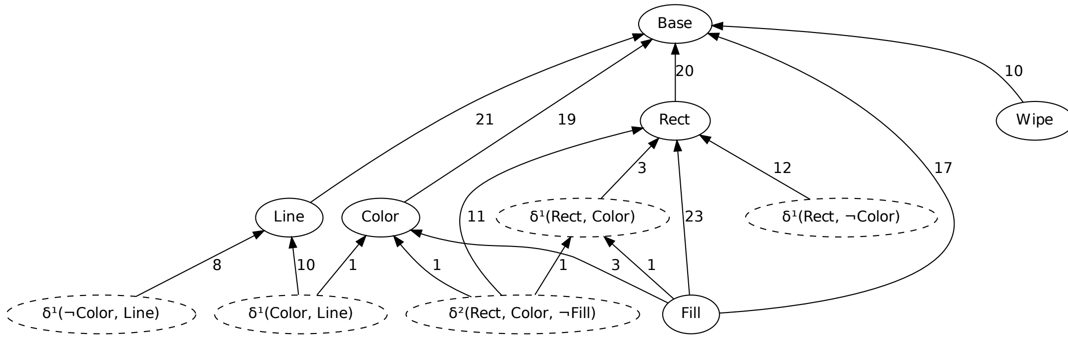


Figure 3: Dependency Graph for DPL

Definition 7. ***featureSets***. *Function featureSets returns the sets of feature sets denoted by a feature model.*

$$featureSets : \mathcal{FM} \rightarrow \mathcal{SFS}$$

The dependencies of a dependency graph must hold in order to guarantee variability safety of system variants. A system variant that violates at least one of the dependencies will not be well-formed because at least one of the source code dependencies is not fulfilled for at least one feature set.

Definition 8. ***holds***. *Function* $holds(dep, fs)$ *returns* 1 *if dependency dep holds on the feature set (of a system variant)* fs *and* 0 *otherwise. A dependency dep holds for a feature set* fs *if:*

$$(\bigwedge_{f \in fs.sel} f \wedge \bigwedge_{g \in fs.\overline{sel}} \neg g) \Rightarrow (dep.from \Rightarrow dep.to)$$

For example, consider dependency $Fill \Rightarrow Rect$, and the system with features `Base` and `Line` selected, i.e. with feature set $[\{Base, Line\}, \{Fill, Rect, Wipe, Color\}]$. Function `holds` returns 1 because the following propositional logic formula evaluates to true: $(Base \wedge Line \wedge \neg Fill \wedge \neg Rect \wedge \neg Wipe \wedge \neg Color) \Rightarrow (Fill \Rightarrow Rect)$.

Now we define the three objective functions that our work considers. The first two are based on information retrieval metrics (see [15]), and the third captures variability safety based on the source code dependencies.

Definition 9. ***Precision (P).*** *Precision expresses how many of the feature sets denoted by a reverse-engineered feature model* fm *are among the desired feature sets* sfs.

$$precision(sfs, fm) = \frac{|sfs \cap featureSets(fm)|}{|featureSets(fm)|}$$

Definition 10. ***Recall (R).*** *Recall expresses how many of the desired feature sets are denoted by the reverse-engineered feature model* fm.

$$recall(sfs, fm) = \frac{|sfs \cap featureSets(fm)|}{|sfs|}$$

Definition 11. ***Variability Safety (VS).*** *We express the degree to which a reverse-engineered feature model* fm *is* variability-safe *with respect to a dependency graph dg as follows:*

$$variabilitySafety(fm, dg) =$$

$$\sum_{dep \in dg} dep.weight \times \left(\frac{\sum_{fs \in featureSets(fm)} holds(dep, fs)}{|featureSets(fm)|} \right)$$

$VS = 0$ means every product variant in fm violates every dependency constraint in dg, and $VS = 1$ means every product variant in fm satisfies every dependency constraint in dg. Next we illustrate our three objective functions.

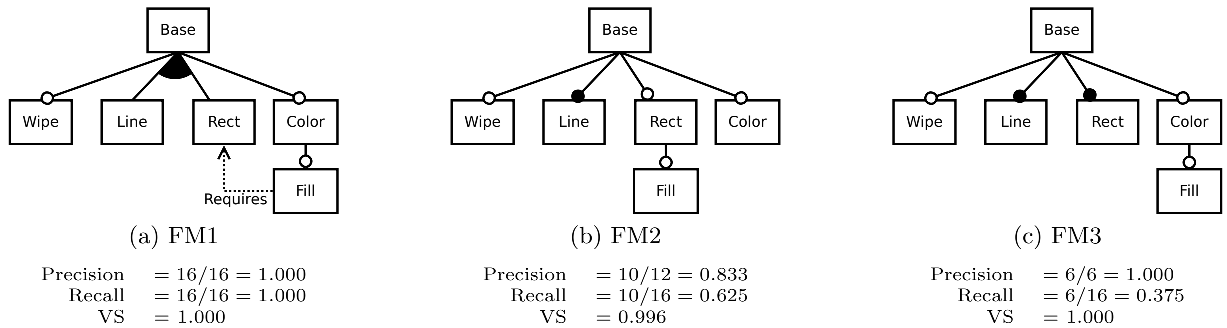
3.2 Objective Functions Illustration

In this section we illustrate our three objective functions, based on the feature sets of sfs in Table 1, the dependency graph dg from Figure 3, and the normalized weight values for dg from Table 2. Consider the three examples of extracted feature models shown in Figure 4 along with their respective values of precision, recall and variability safety.

The feature model FM1, Figure 4(a), is an example of an ideal feature model for the feature sets sfs we are considering. It has $|featureSets(FM1)| = |sfs| = 16$ valid configurations, and exactly the same as shown in Table 1. Hence, the values for both precision and recall are 1.000. In addition, the variability safety value is also 1.000, which means that the features sets of FM1 satisfied all the dependencies. Take for example the system with feature set $[\{Base, Rect, Color\}, \{Wipe, Line, Fill\}]$ and the dependency with ID 14 in Table 2, expressed as $\delta^2(Rect, Color, \neg Fill) \Rightarrow \delta^1(Rect, Color)$. The evaluation of function *holds* returns 1, meaning that this dependency is satisfied by the example feature set.

Let us now consider the feature model FM2, Figure 4(b), which has $|featureSets(FM2)| = 12$ feature sets of which $|sfs \cap featureSets(FM2)| = 10$ are contained in the input feature sets sfs. Hence the value for precision is 0.833 and for recall is 0.625. For this feature model, some of its denoted feature sets do not satisfy all dependencies. For instance, for the feature set $[\{Base, Line, Rect, Fill\}, \{Wipe, Color\}]$, the dependencies ID 7 (`Fill` ⇒ `Color`) and ID 8 (`Fill` ⇒ δ^1(`Rect`, `Color`)) are not satisfied. According to dependency ID 7, when the feature `Fill` is included in a feature set, feature `Color` must also be included. Similarly for dependency ID 8, when feature `Fill` appears, features `Rect` and `Color` must also be included. This feature set hence breaks these two dependency constraints. In other words, FM2 is not fully variability-safe with final value of VS of 0.996.

For feature model FM3, Figure 4(c), there are 6 configurations possible, and all are contained in the input feature sets sfs, in other words, $|featureSets(FM3)| = 6$ and $|sfs \cap featureSets(FM3)| = 6$. Hence, the value of precision is 1.000 and the value of recall is 0.375. Now let us consider the variability safety objective that has a value of 1.000. As mentioned before for FM1, this value indicates



(a) FM1

Precision = 16/16 = 1.000
Recall = 16/16 = 1.000
VS = 1.000

(b) FM2

Precision = 10/12 = 0.833
Recall = 10/16 = 0.625
VS = 0.996

(c) FM3

Precision = 6/6 = 1.000
Recall = 6/16 = 0.375
VS = 1.000

Figure 4: Examples of Extracted Feature Models for DPL

that for all denoted feature sets, no dependency constraint is broken. For example, let us consider the feature set: $[\{Base, Line, Rect, Color, Fill\}, \{Wipe\}]$. For this feature set the dependencies with IDs {1, 3, 4, 5, 6, 7, 8, 10, 12, 13} in Table 2 are satisfied because both the depending modules (*from* column) as well as the modules they depend on (*to* column) are contained. For instance, according to dependency with ID 1, when feature `Line` is selected in a feature set, feature `Base` must also be in the feature set, which is the case because `Base` is the root feature and `Line` is its mandatory child. Furthermore, the dependencies with IDs {2, 9, 11, 14, 15, 16} are also satisfied because the depending modules (*from* column) are not contained. For instance, dependency with ID 2 states that when feature `Wipe` is selected, feature `Base` must also be selected, but since in our feature set example feature `Wipe` is not selected, it is not required that feature `Base` be selected.

Considering the three feature models presented in Figure 4 it is also possible to observe trade-offs among the objectives. It is clear that FM1 is the optimal solution for the given input, but this ideal solution is not known in the beginning of the evolutionary process. If we consider FM2 and FM3, it is not simple to decide which one is better. They have different values for the three objectives. FM2 has a better value for recall than FM3, but on the other hand, FM3 has better values for precision and variability safety than FM2. These types of trade-offs give the software engineer the flexibility to decide which solution(s) work(s) best for his/her concrete reverse engineering task.

4. EVALUATION

In this section we provide details of the case studies used in our evaluation, our experimental set up, the results obtained with their corresponding analysis, and the threats to validity we identified. The implementation and data are available online for replication[1].

4.1 Case Studies

Table 3 presents the systems used in the evaluation of the proposed approach. Draw Product Line (DPL), briefly presented in our running example, is a simple drawing application. Video On Demand (VOD) implements video-on-demand streaming. ArgoUML is an open source UML modelling application. ZipMe is a system for files compression. Game Of Life (GOF) is a customizable game.

[1]http://isse.jku.at/tools/gecco15

Table 3: Case Studies Overview

System	#F	#P	LoC	#Nodes	#Edges
DPL	6	16	282 - 473	12	27
VOD	11	32	4.7K - 5.2K	7	11
ArgoUML	11	256	264K - 344K	49	114
ZipMe	7	32	5K - 6.2K	29	60
GOL	15	65	874 - 1.9K	12	24

#F: Number of Features, #P: Number of Products, LoC: Lines of Code, #Nodes: Number of Nodes in the Dependency Graph, #Edges: Number of Edges, i.e. Dependencies, in the Dependency Graph

4.2 Experimental Setup

As explained before, we used three objective functions precision, recall, and variability safety and normalized their values in the interval between 0 and 1. The goal is to maximize the values of all objective functions. Hence, the best solution is *precision*=1.0, *recall*=1.0, and *variability safety*=1.0.

For our implementation, we relied on the same genetic programming representation and evolutionary operators proposed by Linsbauer et al. [12], whose parameter settings are replicated in Table 4. In contrast with their single-objective work, we not only use recall and precision but also add variability-safety as objective function and use the *Non-Dominated Sorting Genetic Algorithm (NSGA-II)* [6], implemented in the ECJ Framework[2]. We performed 30 independent runs of our algorithm for each case study. The runs were performed on a machine with an Intel® Core^{TM} i7-4900MQ CPU with 2.80 GHz, 16 GB of memory running a Linux platform.

Table 4: Algorithm's Parameters based on [12].

Parameter	Value
Number of Generations	1000
Population Size	100
Crossover	0.7
Feature Tree Mutation	0.5
CTCs Mutation	0.5
Number of Elites	1
Selection Method	Tournament
Tournament Size	6
Maximium CTC Percentage for Builder*	0.1
Maximium CTC Percentage for Mutator*	0.5

* relative to number of features

[2]http://cs.gmu.edu/˜eclab/projects/ecj/

4.3 Results and Analysis

For the analysis we composed the approximated *Pareto Front (PF)* by merging all the solutions of each run and leaving only the non-dominated solutions. The PF for each case study is presented in Table 5. The average runtime of the algorithm NSGA-II in each run for the case studies were: ArgoUML = 7m 16s 337ms, DPL = 9s 854ms, GOL = 45s 036ms, VOD = 574ms, and ZipMe = 40s 263ms.

Table 5: Non-Dominated Solutions

System		Precision	Recall	Variability Safety
ArgoUML		1.0	1.0	0.9999630
		1.0	0.9375	0.9999654
		1.0	0.875	0.9999683
		1.0	0.75	1.0
DPL		1.0	1.0	1.0
GOL		1.0	0.5538462	1.0
		0.8333333	0.6153846	1.0
		0.8000000	0.7384616	1.0
		0.4851485	0.7538462	1.0
		0.4827586	0.8615385	1.0
		0.4571429	0.9846154	1.0
		0.2826087	1.0	1.0
VOD		1.0	1.0	1.0
ZipMe	(A)	1.0	0.75	1.0
	(B)	1.0	0.875	0.9999600
	(C)	0.9696970	1.0	0.9999548
	(D)	1.0	1.0	0.9999534

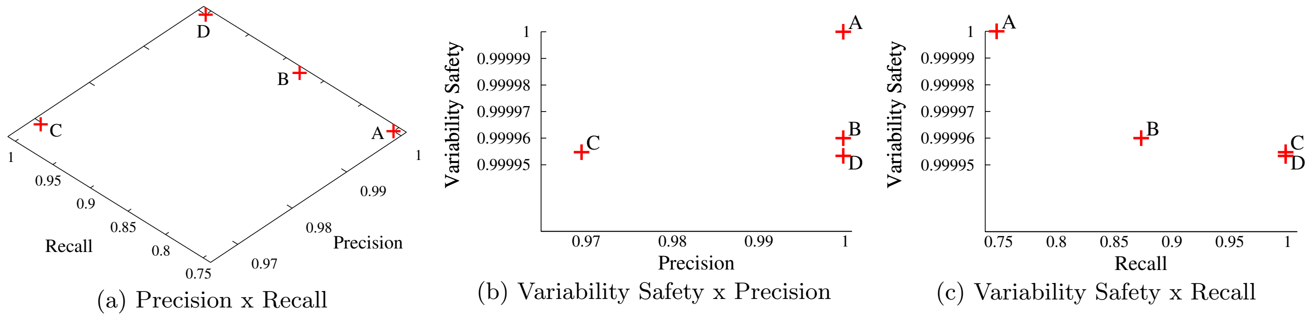
For the case studies DPL and VOD the PF is composed by only one solution. This single solution has the best values for the three objectives, therefore it dominates any other existing solution. The reverse-engineered feature model denotes all the feature sets used as input (recall=1.0), the feature sets denoted are exactly those used as input (precision=1.0) and none of the feature sets break any of the dependency constraints (variability safety=1.0). During the analysis of the reason why these systems have only one solution, we figured out that their features are mainly connected by optional and mandatory relationships, the depth of the feature model tree has only two levels, and the input feature sets could be achieved by different FMs. These characteristics allow the values of precision and recall to reach 1.0 in the same solution. Furthermore, these two systems have few dependencies (see Table 3), making it easier to reach 1.0 for variability safety. In other words, these two systems impose less constraints on the search space which makes finding a solution easier.

For the systems ArgoUML, GOL and ZipMe the approximated PF has four, seven and four solutions, respectively. This indicates that there are some conflicts to optimize the three measures simultaneously. This is an important benefit of using a multi-objective approach, since no single solution is the best one, but a set of acceptable solutions exists and their respective trade-offs can be considered by the software engineer. We now illustrate in more detail the decision making support that our approach can provide to software engineers. We use the ZipMe case study because it has a good trade-off among the solutions and because of its size, only seven features, which allows us to depict some of the feature models that were reverse-engineered.

Figure 5 shows the solutions of ZipMe for each combination of two dimensions in the search space. The solutions are labelled with letters for easy reference, see Table 5 last row. Figure 5(a) presents the measures of precision and recall. In an initial analysis, considering only these two objectives, solution *D* is the best one, with value 1.0 for precision and recall. However, these solutions have different values of variability safety, as depicted in the other graphs. In Figure 5(b), it can be seen that solution *D* has the worst value for variability safety with the value 0.9999534. This value indicates that some dependencies are broken for some feature sets, in particular one dependency is broken by eight feature sets. Solution *A* has the best value for variability safety, i.e. 1.0, which indicates that there are no feature sets that break any dependency. In this same solution, the value of precision is still excellent (precision=1.0), but the value of recall of this solution is the worst (recall=0.75), as presented in Figure 5(c). In this figure, we can also observe that the value of recall increases while the value of variability safety decreases. The solutions *C* and *D* have the best value of recall (recall=1.0) and a slight difference of variability safety with values of 0.9999548 and 0.9999534 respectively. Solution *C* has better value of variability safety than solution *D*, but on the other hand has the worst value of precision (Figure 5(b)). Solution *B* has the best value of precision, as well as solutions *A* and *D*, has a slightly better value of variability safety than solutions *C* and *D*, but on the other hand has a worse value of recall than solutions *C* and *D*.

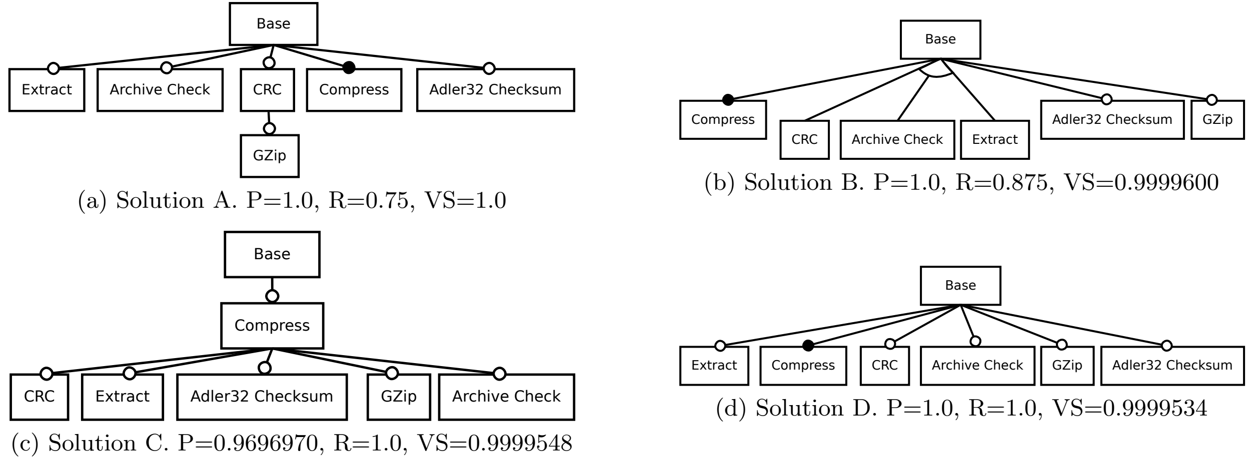
Let us now analyze the meaning of the values for each objective function using the solutions of ZipMe shown in Figure 6. It is important to point out that we identified the dependency from feature `GZip` to feature `CRC` as the only broken dependency in solutions *B*, *C*, and *D*. In the FM of solution *A*, shown in Figure 6(a), we observe that all dependencies are satisfied (variability safety = 1.0). The dependency broken in the other FMs is not possible here because feature `GZip` is child of `CRC`. But this feature model does not denote the same feature sets used as input, for example, those with feature `GZip` and without feature `CRC`. From the 32 feature sets used as input only 24 are denoted, hence the value of recall decreases to 0.75. The FM of solution *B*, shown in Figure 6(b), denotes 28 feature sets belonging to the input (recall=0.875). The feature sets missing are because of the *or group* that forces the selection of at least one of the features `CRC`, `ArchiveCheck` and `Extract`. From these 28 feature sets, six break the dependency between features `GZip` and `CRC`, i.e. variability safety = 0.9999600. The FM of the solution *C*, shown in Figure 6(c), denotes all the 32 input feature sets (recall=1.0). However, one more additional feature set is also denoted, namely, a feature set with only the feature `Base`. This surplus decreases the value of precision to 0.9696970. Out of these 33 feature sets, the dependency between `GZip` and `CRC` is broken in eight of them. The FM in solution *D*, Figure 6(d), denotes all the 32 feature sets used as input (recall=1.0), there are no additional feature sets (precision=1.0), and eight feature sets break the dependency between `GZip` and `CRC`. Even though the number of broken dependencies is the same as in solution *C*, the value of variability safety of solution *D* (i.e. 0.9999534) is worse because solution *C* (i.e. 0.9999548) has more feature sets.

In summary, based on the obtained solutions, the software engineer can decide which combinations of objective functions and trade-offs are more relevant to his/her task.



(a) Precision x Recall (b) Variability Safety x Precision (c) Variability Safety x Recall

Figure 5: ZipMe Non-Dominated Solutions



(a) Solution A. P=1.0, R=0.75, VS=1.0

(b) Solution B. P=1.0, R=0.875, VS=0.9999600

(c) Solution C. P=0.9696970, R=1.0, VS=0.9999548

(d) Solution D. P=1.0, R=1.0, VS=0.9999534

Figure 6: ZipMe Feature Models

For instance, if the recall is more important, solutions C and D are the best options. If he/she has precedence for variability safety, solution A is the best. On the other hand, if the goal is a solution with good trade-off considering all objectives, solution D can be a better option. Regarding the practical use of the variability safety, if the software engineer chooses a solution with variability safety less than 1.0 (e.g. solution B), he/she must resolve the broken dependencies (e.g. applying refactorings) so that the affected systems can be actually realized if so desired.

In contrast with ZipMe, the case studies ArgoUML and GOL only exhibit trade-offs between two objective functions, see Table 5. Regarding ArgoUML, the values of precision for all solutions are the best (precision=1.0), so the software engineer should decide among recall or variability safety values. For GOL, the value of variability safety is the best for all solutions (variability safety=1.0), with diversity of values for precision and recall to choose from.

4.4 Threats to Validity

The first threat to validity that we identified was the selection of parameter settings for our algorithm. To address this threat, we relied on the standard values employed by Linsbauer et al. [12]. Certainly, other parameter values could yield different results. A detailed analysis is outside of the scope of this paper and part of our future work. The second threat concerns the selection of case studies. To address this threat, we selected five case studies from different application domains and dimensions. These case studies have been substantially used for SPL research for multiple purposes [2]. Based on this we argue that they are representative of the reverse engineering scenario targeted by our approach. A third threat regards the comparison against related work. We should remark that, to the best of our knowledge, our work is the first to propose a multi-objective perspective and include information from artifacts such as dependency graphs for reverse engineering feature models. Therefore a comparison against such approaches is not adequate.

5. RELATED WORK

There is increasing literature in reverse engineering feature models. In this section, we summarize those pieces of work closest to ours. The closest related work is Linsbauer et al. [12], on which we based our feature model representation and evolutionary operators. In contrast with our work, they propose a single-objective algorithm for F1 — a measure that considers recall and precision with equal preference — and do not consider any information from implementation artifacts for their reverse engineering of feature models.

Work by Haslinger et al. proposes an ad hoc algorithm to reverse engineer feature models from feature sets that works by identifying occurrence patterns in the selected and not selected features that are mapped to parent-child relations of feature models [8]. This work has been extended to consider requires and excludes CTCs [9]; however, it does not support more general types of CTCs. In contrast with our work, they

do not consider any domain knowledge information, just the input feature sets, for the reverse engineering task.

Work by Czarnecki and Wasowski reverse engineers feature models from sets of propositional logic formulas by means of an ad hoc algorithm that can potentially extract from a single propositional logic formula multiple feature models while trying to preserve the original formulas and reduce redundancies [5]. A recent extension of this work provides improved algorithms based on CNF and DNF constraints [18]. In contrast with our work, their starting point are configuration files, documentation files, and constraints expressed in propositional logic.

Acher et al. also tackle the reverse engineering of feature models from feature sets by mapping each feature set into a feature model which are later merged into a single feature model [1]. Besides this difference, they do not consider any other domain knowledge information. Sannier et al. perfomed an analysis of matrices for product comparison available at Wikipedia [16]. Their product matrices can contain other values, rather than selected or not selected, as in our case. Based on these matrices, their work identifies variability patterns in the values of the cells, and portrays the challenges and potential benefits of exploiting that information, among other things, to extract models such as feature models. However, to the best of our knowledge, at present there is no algorithm, let alone tool support, that capitalizes on this information to aid the reverse engineering effort.

6. CONCLUSIONS AND FUTURE WORK

This paper presented a multi-objective approach to reverse engineering FMs. We introduced a measure to evaluate variability safety in addition to recall and precision. For computing variability safety we considered dependency graphs representing the source code dependencies. Our evaluation showed that a multi-objective approach can indeed offer a set of solutions with good trade-offs. From this set of solutions the software engineer has the power to select which objective function to favour based on his/her needs.

Note that variability safety is not restricted to dependencies of source code. A similar objective function can be computed on other types of implementation artifacts. Furthermore this type of objective function expresses a conformance of a candidate FM to a set of constraints, which can also include for instance those given by domain experts based on their knowledge of the legacy system variants. Along these lines, in our future work we plan to extract and use dependency information from other sources besides code. In addition, we plan to apply and study other multi-objective algorithms, analyze parameter settings, and use our approach on more case studies.

Acknowledgments

This work was supported by Brazilian Agencies CAPES and CNPQ and the Austrian Science Fund (FWF): P 25289-N15.

7. REFERENCES

[1] M. Acher, A. Cleve, G. Perrouin, P. Heymans, C. Vanbeneden, P. Collet, and P. Lahire. On extracting feature models from product descriptions. In *VaMoS*, pages 45–54, 2012.

[2] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, and A. Egyed. A systematic mapping study on migrating software systems to software product lines. 2015. submitted.

[3] D. S. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[4] D. Benavides, S. Segura, and A. R. Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[5] K. Czarnecki and A. Wasowski. Feature diagrams and logics: There and back again. In *SPLC*, pages 23–34. IEEE Computer Society, 2007.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *Trans. on Evol. Comp.*, 6(2):182–197, 2002.

[7] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Enhancing clone-and-own with systematic reuse for developing software variants. In *ICSME*, 2014.

[8] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. Reverse engineering feature models from programs' feature sets. In *WCRE*, pages 308–312, 2011.

[9] E. N. Haslinger, R. E. Lopez-Herrejon, and A. Egyed. On extracting feature models from sets of valid feature combinations. In *FASE*, pages 53–67, 2013.

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, CMU, 1990.

[11] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Recovering traceability between features and code in product variants. In *SPLC*, pages 131–140, 2013.

[12] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. Feature model synthesis with genetic programming. In *SSBSE*, pages 153–167, 2014.

[13] R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *SSBSE*, pages 168–182, 2012.

[14] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, and A. Egyed. An assessment of search-based techniques for reverse engineering feature models. *Journal of Systems and Software*, 103(0):353 – 369, 2015.

[15] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to information retrieval.* CUP, 2008.

[16] N. Sannier, M. Acher, and B. Baudry. From comparison matrix to variability model: The wikipedia case study. In *ASE*, pages 580–585. IEEE, 2013.

[17] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *ICSE*, pages 461–470. ACM, 2011.

[18] S. She, U. Ryssel, N. Andersen, A. Wasowski, and K. Czarnecki. Efficient synthesis of feature models. *Inf. & Softw. Techn.*, 56(9):1122–1143, 2014.

[19] 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.

[20] N. Weston, R. Chitchyan, and A. Rashid. A framework for constructing semantically composable feature models from natural language requirements. In *SPLC*, pages 211–220, 2009.