Empir Software Eng (2017) 22:1763–1794
DOI 10.1007/s10664-016-9462-4

Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants

Wesley K. G. Assunção[1,2] · Roberto E. Lopez-Herrejon[3] · Lukas Linsbauer[4] · Silvia R. Vergilio[1] · Alexander Egyed[4]

Published online: 20 October 2016

Abstract Maintenance of many variants of a software system, developed to supply a wide range of customer-specific demands, is a complex endeavour. The consolidation of such variants into a Software Product Line is a way to effectively cope with this problem. A crucial step for this consolidation is to reverse engineer feature models that represent the desired combinations of features of all the available variants. Many approaches have been proposed for this reverse engineering task but they present two shortcomings. First, they use a single-objective perspective that does not allow software engineers to consider design trade-offs. Second, they do not exploit knowledge from implementation artifacts. To address

Communicated by: Mark Harman

✉ Wesley K. G. Assunção
wesleyk@inf.ufpr.br

Roberto E. Lopez-Herrejon
roberto.lopez@etsmtl.ca

Lukas Linsbauer
lukas.linsbauer@jku.at

Silvia R. Vergilio
silvia@inf.ufpr.br

Alexander Egyed
alexander.egyed@jku.at

[1] DINF, Federal University of Paraná (UFPR), CP: 19081, CEP: 81.531-980, Curitiba, Brazil

[2] COTSI, Federal University of Technology - Paraná (UTFPR), Cristo Rei Street, 19. CEP: 85.902-490, Toledo, Brazil

[3] Department of Software Engineering and IT, École de Technologie Supérieure, (ÉTS), Notre-Dame Street Ouest. 1100, H3C 1K3 Montreal, Canada

[4] ISSE, Johannes Kepler University Linz (JKU), Altenbergerstr. 69, 4040 Linz, Austria

these limitations, our work takes a multi-objective perspective and uses knowledge from source code dependencies to obtain feature models that not only represent the desired feature combinations but that also check that those combinations are indeed well-formed, i.e. variability safe. We performed an evaluation of our approach with twelve case studies using NSGA-II and SPEA2, and a single-objective algorithm. Our results indicate that the performance of the multi-objective algorithms is similar in most cases and that both clearly outperform the single-objective algorithm. Our work also unveils several avenues for further research.

Keywords Reverse engineering · Feature models · Multi-objective evolutionary algorithms · Empirical evaluation

1 Introduction

Software Product Lines (SPLs) are families of software products with focus on reuse of artefacts (Batory et al. 2004). SPLs have been receiving wide attention in the industry due to their advantages, among them, higher software quality and shorter time-to-market for new products (van d. Linden et al. 2007). However, there are some challenges on the adoption of SPLs in industrial settings. One of the main challenges is how to consolidate existing system variants into a SPL. To tackle such a challenge, reverse engineering strategies have been proposed to deal with different stages of the SPL development (Assunção and Vergilio 2014). The starting point to this reverse engineering process is the extraction of *Feature Models (FMs)*, the de facto standard for modeling variability and commonality in SPLs (Benavides et al. 2010), which denote the set of valid configurations of features that constitute the products of a SPL.

In recent years several approaches to reverse engineer FMs have been proposed. They are based on configuration scripts (She et al. 2011), propositional logic expressions (Czarnecki and Wasowski 2007; She et al. 2014), natural language (Weston et al. 2009), ad hoc algorithms (Acher et al. 2012; Haslinger et al. 2011, 2013), and search-based techniques (Linsbauer et al. 2014; Lopez-Herrejon et al. 2012, 2015; Thianniwet and Cohen 2015). However, they present two main limitations. They do not take a multi-objective perspective to capture the trade-offs that software engineers must make for the reverse engineering task, and do not exploit any knowledge on how the system variants are actually implemented.

A multi-objective perspective has advantage in situations where there are conflicts among the goals of the software engineer. For instance, obtaining an FM that represents a set of desired product configurations can lead to a model that also generates a surplus of configurations, which are not desired. Nevertheless, if we tweak the FM to avoid such additional configurations we might loose some desired configurations. Here a multi-objective algorithm aims to optimizing both goals and enables the engineer to make a decision based on an analysis of the different trade-offs of importance in the problem domain. In addition, the use of knowledge available in implementation artefacts is an important characteristic because we can generate FMs in accordance with the already existing software variants.

To cope with these limitations, our previous work presented an approach to reverse engineer FMs based on the multi-objective algorithm NSGA-II (Assunção et al. 2015). This work used a graph to represent dependencies in the source code artifacts of the existing system variants, and provided software engineers with sets of FMs with different trade-offs. In this paper we extend our previous work by including: *i)* a second multi-objective evolutionary algorithm, namely *Strength Pareto Evolutionary Algorithm (SPEA2)* (Coello et al.

2007), *ii)* a single-objective evolutionary algorithm that relies on a genetic programming representation for baseline comparison, *iii)* a detailed description of our problem representation and evolutionary operators, *iv)* seven new case studies, and *v)* a more thorough empirical evaluation and analysis. In summary, our current work addresses the following research questions:

- *RQ1: What are the benefits and advantages of using a multi-objective approach over a single-objective one to reverse engineer FMs?*
- *RQ2: How does the performance of NSGA-II and SPEA2 compare for reverse engineering FMs?*
- *RQ3: How could a multi-objective perspective be used in practice to support software engineers in the decision making process?*

The remainder of this paper is structured as follows: Section 2 presents the fundamental concepts of feature models, a running example, and the definitions regarding the dependency graphs. The details of the proposed approach are described in Section 3. The setup used in the evaluation of the proposed approach is found in Section 4. Section 5 has the results obtained in the evaluation and the corresponding analysis in order to answer the research questions. Related work is found in Section 6. The research avenues for future work and conclusions are presented respectively in Sections 7 and 8.

2 Background

In this section we present an overview of feature models, a running example to illustrate the details of our approach, and define more precisely the notions of dependencies and dependency graphs which we use to represent the source code dependencies. We rely on these definitions for describing our multi-objective approach in Section 3.

2.1 Feature Models

Feature Models (FMs) are widely used to model the different combinations of features in a SPL (Kang et al. 1990), and are key for supporting variability and commonality management (Benavides et al. 2010). These models follow a tree-like structure where features are depicted as boxes, labelled with the feature name, which are connected by lines with their children features.

An FM always has a *root* feature that is present in all products of the SPL. The other features can be of type *mandatory* or *optional*, or a set of features can be organized in groups as *alternative groups* or *or groups*. A mandatory feature is always selected when its parent feature is selected. An optional feature may or may not be selected when its parent is selected. The mandatory feature is represented with filled circle at the end of the relation line and the optional feature is represented with an empty circle. These two types of features are respectively presented in Fig. 1a and b. An *alternative group* indicates that when the parent feature of the group is selected then *exactly* one feature of the group must be selected. When the parent feature of an *or group* is selected then *at least* one feature of the group must be selected. The *alternative* group is represented by an empty arc and the *or* group is represented by a filled arc, as illustrated in Fig. 1c and d, respectively.

In addition to the hierarchical relation between the features, the valid configuration of features can also be restricted by some additional constraints, called *Cross-Tree Constraints*

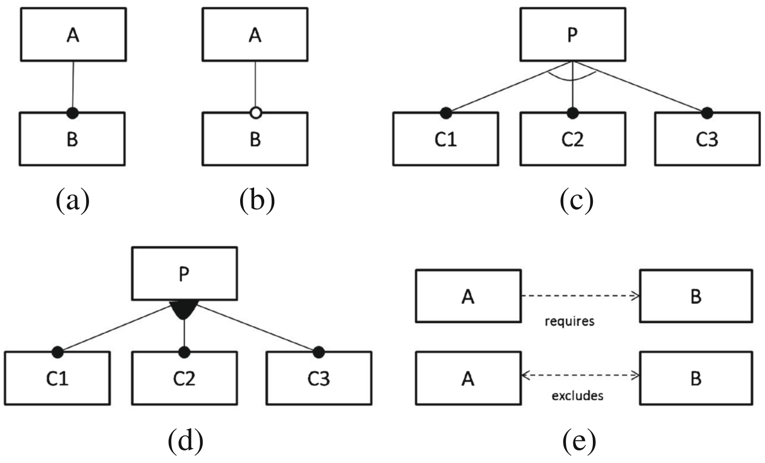


Fig. 1 Feature models graphical notation

(CTCs) (Benavides et al. 2010). The most common types of CTCs are *requires* and *excludes*. If a feature A is selected and requires feature B, then feature B must also be selected. If a feature A excludes feature B then these two features cannot both be selected in the same feature combination.

These two types of CTCs are usually respectively represented by single and double-arrow dashed lines as shown in Fig. 1e.

2.2 Running Example

In this subsection we present a running example to illustrate the details of our approach. We selected a set of variants of a drawing application. Our goal is to use these variants as a starting point to obtain a product line called *Draw Product Line (DPL)*. This application offers users the ability to handle a drawing area (feature BASE), draw lines (feature LINE), draw rectangles (feature RECT), draw filled rectangles (feature FILL), select a color for the line or rectangle (feature COLOR), and clean the drawing area (feature WIPE). With these six features we have a total number of 16 variants of the drawing application, presented in Table 1. The symbol ✓ indicates the selected features. We refer to each feature combination as a *feature set*, formally defined as Linsbauer et al. (2014):

Definition 1 Feature Set A *feature set* is a 2-tuple $[\texttt{sel},\overline{\texttt{sel}}]$ where 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 $\texttt{sel}, \overline{\texttt{sel}} \subseteq \texttt{FL}$, $\texttt{sel} \cap \overline{\texttt{sel}} = \emptyset$, and $\texttt{sel} \cup \overline{\texttt{sel}} = \texttt{FL}$.

2.3 Source Code Dependency Graphs

One of the main contributions of our approach is to use knowledge from implementation artefacts, besides the feature sets, to reverse engineer FMs. Based on this knowledge, we evaluate *variability safety* of FMs, a property that guarantees that the feature sets denoted by

Table 1 Feature sets for DPL

Products	BASE	LINE	RECT	COLOR	FILL	WIPE
Product_1	✓	✓				✓
Product_2	✓	✓		✓		
Product_3	✓	✓	✓	✓		
Product_4	✓	✓	✓	✓	✓	
Product_5	✓	✓	✓			✓
Product_6	✓	✓				
Product_7	✓	✓		✓		✓
Product_8	✓	✓	✓			
Product_9	✓		✓			
Product_{10}	✓		✓			✓
Product_{11}	✓		✓	✓		
Product_{12}	✓		✓	✓	✓	
Product_{13}	✓		✓	✓		✓
Product_{14}	✓		✓	✓	✓	✓
Product_{15}	✓	✓	✓	✓		✓
Product_{16}	✓	✓	✓	✓	✓	✓

an FM are structurally well-formed according to the source code. In particular, we exploit the information of existing dependencies between source code fragments, as described next.

To represent the dependencies existing in the source code of the different system variants we use a weighted directed graph. To create this graph we use the terminology and the tool from our previous work (Fischer et al. 2014; Linsbauer et al. 2013). We identify *modules* of two kinds:

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 name written in lowercase.

Definition 3 Derivative Module A *derivative* module $m = \delta^{n}(\texttt{c}_0, \texttt{c}_1, ..., \texttt{c}_n)$ 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.

These two types of modules are the basis of our extraction algorithm (Linsbauer et al. 2013). This algorithm computes traces from the modules to their implementing source code fragments and identifies dependencies between the modules that have dependencies in their implementations. The algorithm considers any granularity of the implementation artefacts, from class level to statement level. Figure 2 presents some examples of traces computed by the algorithm. The traces are indicated using comments at the end of the lines. For example, the field defined in Line 5 traces to the base module *Color* of the corresponding feature `Color`. This source code fragment is present in all variants with feature `Color`, regardless the existence of other features. Another example of a base module is observed in Line 7 of the example. This method header and most of its implementation will be included in class `Canvas` whenever feature `Line` is present, regardless of any other feature. However, in Line 10 of the method we can observe a derivative module $\delta^1(Color, Line)$, which means that the corresponding line of code will be part of its containing method only when the variant has both features `Color` and `Line`.

```
public class Canvas ... {
  List<Shape> shapes = new LinkedList<Shape>();
  Point start;
  Line newLine = null; // Line
  Color color = Color.BLACK; // Color
  ...
  void mousePressedLine(MouseEvent e) { // Line
   if (newLine == null) {
      start = new Point(e.getX(), e.getY());
      newLine=new Line(color,start);//δ^1(Color, Line)
      shapes.add(newLine);
   }
  }
}
```

Fig. 2 Source code snippet for DPL example

In order to more formally define dependencies and their representation as graphs, we first introduce the concept of module expression as follows:

Definition 4 Module Expression A module expression is the propositional logic representation of modules. For a base module b, the module expression is its own literal b. For a derivative module $m = \delta^{\mathrm{n}}(\mathrm{c}_0, \mathrm{c}_1, ..., \mathrm{c}_{\mathrm{n}})$ its module expression corresponds to $c_0 \wedge c_1 \wedge ... \wedge c_n$.

As an illustration, the module expression of base module `Color` is $Color$. For the derivative module δ^1(`Color`, `Line`), which indicates the interaction between features `Color` and `Line`, the module expression representation is $Color \wedge Line$.

The traces produced by our trace extraction algorithm are used to identify the dependencies between fragments of source code. These dependencies and the dependency graph they form are formally defined 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 refer to 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 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.

Now we recall our running example of the drawing application, Fig. 3 presents the dependency graph considering all its feature sets. To avoid clutter only the lowest order modules are presented, since they are the most relevant to our approach. Self-dependencies are removed from the graph for better readability. To make clear the different kinds of modules,

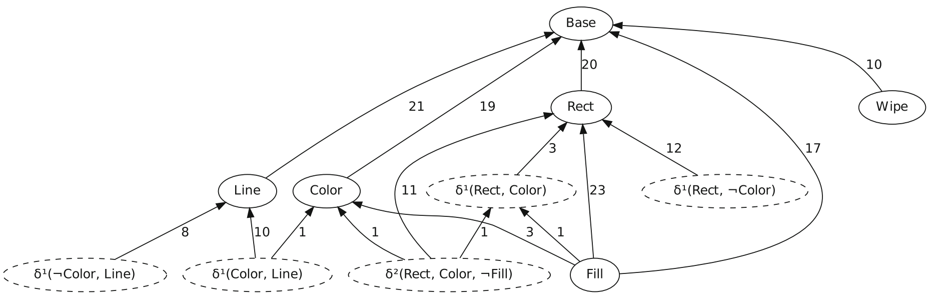


Fig. 3 Dependency graph for DPL

the base modules have solid borders and the derivative modules have dashed borders. In the figure we can observe that the strongest dependencies (i.e. those with the highest weights) are those that go to base modules, e.g. `Fill` to `Rect`, and the core feature `Base` has the largest number of incoming dependencies. As mentioned before, weights in the graph represent the number of structural dependencies between source code elements belonging to different features. For instance, in Fig. 3 there are 10 dependencies from source code elements of `WIPE` to source code elements of `BASE`. These dependencies are field accesses (4) and containment relationships (6), i.e a field belongs to a class.

Alternatively the dependency graph can be represented as a dependency matrix, as presented in Table 2. The rows are the dependencies, the first column is a dependency identification for easy reference, the second and third columns have the modules of the dependency in the order *from* to *to*, respectively. The weight of the dependency is displayed in the fourth column. In the fifth column the weight is normalized to keep the sum of all

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

weights of the graph equal to 1.0. This normalization enables a better interpretation of the values in the optimization process.

To illustrate the propositional logic representation of dependencies let us use again the source code shown in Fig. 2. Firstly consider the dependency that exists between the module $\delta^1(Color, Line)$ and the module $Color$. This dependency exists because the field `color` defined in Line 5 belongs to the module $Color$ and it is used by `newLine = new Line(color,start);` in Line 10 which belongs to the module $\delta^1(Color, Line)$. The propositional logic expression for this dependency is $(Color \wedge Line) \Rightarrow Color$. In the same code snippet of Fig. 2 we can see that module $\delta^1(Color, Line)$ depends on module $Line$, because the statement in Line 10 is contained in the method `void mousePressedLine(MouseEvent e)` which belongs to module $Line$. In this case, the propositional logic expression is $(Color \wedge Line) \Rightarrow Line$.

An important point is to observe that the propositional logic constraints of some dependencies are tautologies, hence they always hold. The two above examples illustrate this situation, since $(Color \wedge Line \Rightarrow Line) \Leftrightarrow TRUE$. This indicates that the implementation artefacts are consistent with the feature combinations represented in the FM.

3 Multi-Objective Approach to Reverse Engineer Feature Models

In this section we describe the details of our multi-objective search-based approach which relies on and extends upon our previous work (Linsbauer et al. 2014). We further describe the requirements identified by Harman et al. (2012) to implement a search-based solution: (i) an adequate representation of solutions, (ii) a set of operators to improve the solutions and explore the search space, and (iii) an adequate way to evaluate the quality of the solutions, the fitness functions.

3.1 Feature Model Representation

The feature model representation uses a simplified version of the SPLX metamodel.[1] This metamodel, presented in Fig. 4, defines both structure and semantic of the FMs. In the figure, the elements in the left part describe the tree-like structure of the FM and the elements in the right describe the CTCs between the features. The tree nodes *Root*, *Mandatory*, *Optional*, and *GroupedFeature* inherit from *Feature*. The tree is composed by exactly one *Root* feature. Features *Mandatory* and *Optional* have the cardinality of zero or more. The tree can also have an arbitrary number of *Alternative* and *Or* groups and each group must have at least one *GroupedFeature*. In the right part of the figure we can observe that an FM has exactly one *ConstraintSet*. This *ConstraintSet* describes the propositional formula in CNF. Zero or more *Constraint* are acceptable, each constraints is a clause in a CNF expression. Each *Constraint* has exactly one *OrClause* that has at least one *Literal*. A literal can either be an *Atom* which refers directly to a feature, or a *Not* which refers to an *Atom*.

Following this representation, the initial population is created by generating random feature trees and random CTCs. Some additional domain constraints are also taken into account, they are presented in the next subsection. The tools FaMa (Benavides et al. 2007) and BeTTy (Segura et al. 2012) were used to create the initial population.

[1] http://www.splot-research.org/.

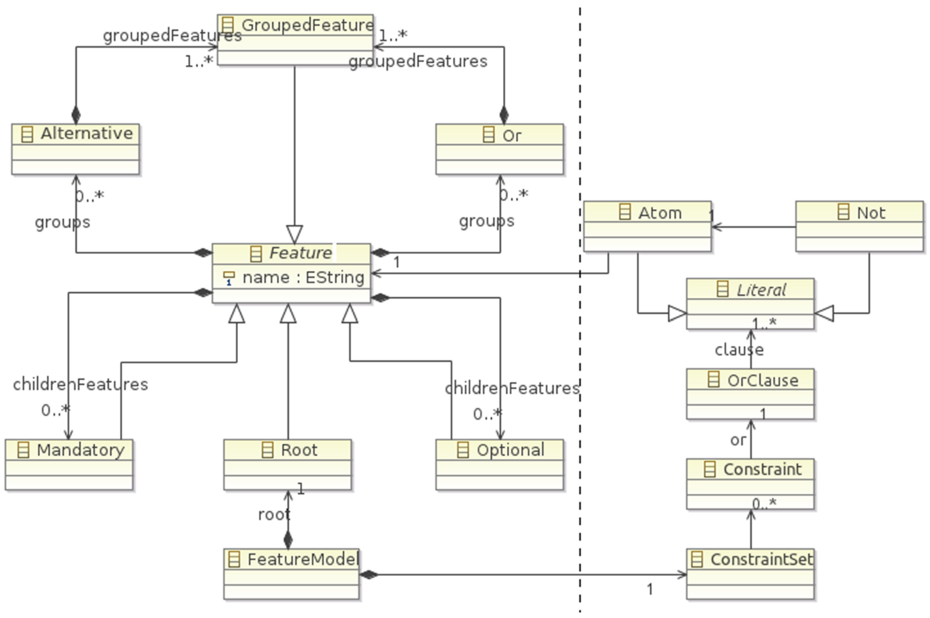


Fig. 4 Feature model metamodel, extracted from Linsbauer et al. (2014)

3.2 Evolutionary Operators

We adopted the evolutionary operators from our previous work (Linsbauer et al. 2014), and employ standard tournament selection as selection operator. There are some domain constraints that should be taken into account in the evolutionary process to guarantee the semantics of FMs and to avoid generating invalid solutions:

- Each feature is identified by its name, so every feature appears exactly once in the FM tree;
- All FMs have a fixed set of feature names, so in different FMs only the relations between features are different;
- CTCs can only be either *requires* or *excludes*, i.e. exactly two literals per clause with at least one being negated;
- CTCs must not contradict each other, i.e. the corresponding CNF of the entire constraint set must be satisfiable;
- There is a maximum number of CTCs (given as a percentage of the number of features) that must not be exceeded.

Our domain constraints do not consider the rare case of contradictions between CTCs and the FM tree for which the detection and repair is computationally expensive. For individuals in that case, we let the evolutionary process itself weed them out because of their bad fitness value.

3.2.1 Mutation

The mutation operator applies small changes in randomly selected parts of the tree or in the CTCs of the feature model. The mutation probability is used to decide if the change is applied in the tree part, in the CTCs, or in both. The kind of change is randomly selected from the following lists:

- Mutations performed on the tree:
    - Randomly swaps two features in the feature tree;
    - Randomly changes an *Alternative* relation to an *Or* relation or vice-versa;
    - Randomly changes an *Optional* or *Mandatory* relation to any other kind of relation (*Mandatory*, *Optional*, *Alternative*, *Or*);
    - Randomly selects a subtree in the feature tree and puts it somewhere else in the tree without violating the metamodel or any of the domain constraints.
- Mutations performed on the CTCs:
    - Adds a new, randomly created CTC that does not contradict the other CTCs and does not already exist;
    - Randomly removes a CTC.

3.2.2 Crossover

Just like the mutation operator, the crossover must generate offspring in conformance to the metamodel and to the domain constraints. The steps of the crossover process are:

1. The offspring is initialized with the root feature of $Parent_1$. If the root feature of $Parent_2$ is a different one then it is added to the offspring as a mandatory child feature of its root feature.
2. Traverse the first parent depth first starting at the *root* node and add to the offspring a random number r of features that are not already contained by appending them to their respective parent feature already contained in the offspring using the same relation type between them (the parent feature of every visited feature during the traversal is guaranteed to be contained in the offspring due to the depth first traversal order).
3. Traverse the second parent exactly the same way as the first one.
4. Go to Step 2 until every feature is contained in the offspring.

The second child is obtained by performing the same process but with reverse parents, i.e. the position of the parents is swapped.

The CTCs offspring are obtained by merging all the constraints of both parents and then randomly selecting a subset of CTCs that are assigned to the first offspring and the remaining to the second offspring.

3.3 Multi-Objective Perspective

In this section we describe the three objective functions used in our approach and present an illustrative example.

3.3.1 Auxiliary Functions Definitions

In order to compute the objective functions of our approach we need some auxiliary functions. Let us consider $\mathcal{FM}$ as the universe of feature models, $\mathcal{SFS}$ the universe of set of feature sets, and sfs a set of feature sets defined by the software engineer. An example of sfs is the feature sets presented in Table 1 for the drawing application. Based on this terminology, we introduce the function *featureSets*:

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

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

To measure the variability safety of an FM we have to check if its feature sets are in conformance with the dependencies of the dependency graph. To check this we define the function *holds*:

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:

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

To illustrate this function recall our running example. Let us consider the dependency $Fill \Rightarrow Rect$ and a system variant with the features `Base` and `Line`. In this case the function *holds* returns 1, since the propositional logic formula $(Base \wedge Line \wedge \neg Fill \wedge \neg Rect \wedge \neg Wipe \wedge \neg Color) \Rightarrow (Fill \Rightarrow Rect)$ is true.

3.3.2 Fitness Functions Definitions

Considering the two auxiliary functions presented before we are able to introduce the three objective functions of our approach. The first two measures are based on information retrieval metrics, for further details refer to Manning et al. (2008), and the third measure is based on variability safety.

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.

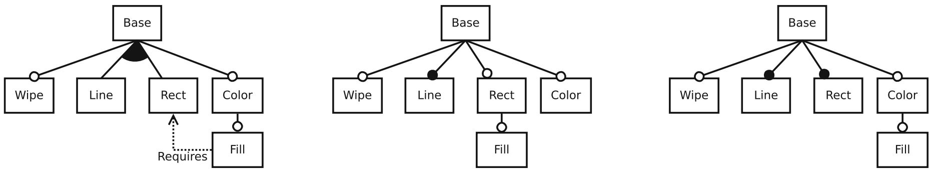
$$precision(sfs, fm) = \frac{|sfs \cap featureSets(fm)|}{|featureSets(fm)|}$$

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

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

Definition 11 Variability Safety (VS). Variability Safety expresses the degree of *variability-safety* of a reverse engineered feature model fm with respect to a dependency graph dg.

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



(a) FM1

$$\begin{aligned} P &= 16/16 = 1.000 \\ R &= 16/16 = 1.000 \\ VS &= 1.000 \end{aligned}$$

(b) FM2

$$\begin{aligned} P &= 10/12 = 0.833 \\ R &= 10/16 = 0.625 \\ VS &= 0.996 \end{aligned}$$

(c) FM3

$$\begin{aligned} P &= 6/6 = 1.000 \\ R &= 6/16 = 0.375 \\ VS &= 1.000 \end{aligned}$$

Fig. 5 Examples of extracted feature models for DPL

3.3.3 Fitness Functions Illustration

To illustrate our three objective functions let us consider the feature sets of sfs in Table 1, the dependency graph dg in Fig. 3, and the normalized weight values for dg from Table 2. Figure 5 presents three examples of FMs extracted from the drawing application variants. We computed the values of precision, recall and variability safety for these FMs.

In Fig. 5a the feature model FM1 is an ideal solution for the sfs in Table 1. This feature model has $|featureSets(FM1)| = |sfs| = 16$, leading to precision and recall equal to 1.000, which means that its valid configurations are exactly the same as the desired feature sets. The value of variability safety for this FM is also 1.000, indicating that the valid configurations of FM1 do not break any dependency.

The feature model FM2, presented in Fig. 5b, denotes $|featureSets(FM2)| = 12$ feature sets of which $|sfs \cap featureSets(FM2)| = 10$ are in the desired feature sets. With these values we have precision = 0.833 and recall = 0.625. Some feature sets denoted by this feature model do not satisfy all dependencies, leading to a value of variability safety = 0.996. To illustrate some broken dependencies we use the feature set $[\{Base, Line, Rect, Fill\}, \{Wipe, Color\}]$. For this feature set the dependency ID 7 (`Fill` $\Rightarrow$ `Color`), shown in Table 2, is not satisfied because the dependency indicates that when `Fill` is in the feature set the feature `Color` must also be included. When a dependency is not satisfied, its normalized weight is not added to the accumulated weight of the satisfied dependencies effectively decreasing the value of variability safety.

Let us now consider the feature model FM3 presented in Fig. 5c. This feature model denotes six feature sets and all of them are desired feature sets. Hence, $|featureSets(FM3)| = 6$ and $|sfs \cap featureSets(FM3)| = 6$, leading to precision = 1.000 and recall = 0.375. Furthermore, no dependency is broken by its feature sets, so the value of variability safety is 1.000. For instance, considering a single feature set $[\{Base, Line, Rect, Color, Fill\}, \{Wipe\}]$, the dependencies with IDs {1, 3, 4, 5, 6, 7, 8, 10, 12, 13} are satisfied because both the *from* modules and the *to* modules are contained. In addition, dependencies with IDs {2, 9, 11, 14, 15, 16} are also satisfied because the *from* modules are not part of the feature set. Once the *from* module is not included in the feature set, it is not required the *to* module be contained.

These illustrative examples show the possible diversity of values among the three objectives. From the decision maker point of view FM1 is the best one, since it has the best values for the three objectives. However, in most of the situations, ideal solutions like this do not necessarily exist, and hence many trade-offs must be considered.

Table 3 Algorithm's parameters

Parameter	GP	NSGA-II	SPEA2
Number of Generations	1000	1000	1000
Population Size	200	200	200
Archive Size	–	–	10
Crossover	0.7	0.7	0.7
Feature Tree Mutation	0.5	0.5	0.5
CTCs Mutation	0.5	0.5	0.5
Number of Elites	25 %	25 %	25 %
Selection Method	Tournament	Tournament	Tournament
Tournament Size	6	6	6
Maximium CTC Percentage for Builder[a]	0.1	0.1	0.1
Maximium CTC Percentage for Mutator[a]	0.5	0.5	0.5
Independent runs	30	30	30

[a]relative to number of features

4 Experimental Description

In this section we present our experimental setup and describe the case studies used for our evaluation. The implementation and data are available online for replication.[2]

4.1 Experimental Setup

In order to answer our research questions we perform an experiment, which is described as follows. As mentioned before, our focus is to solve the problem of reverse engineering of FMs using three objective functions: *Precision (P)*, *Recall (R)*, and *Variability Safety (VS)*. We designed these measures to be normalized values in the interval between 0 and 1, where the goal is to maximize the values of all objective functions. Hence, the ideal solution is P = 1.0, R = 1.0, and VS = 1.0.

In Section 3.1 we described the genetic programming representation we used. In addition to our previous work (Assunção et al. 2015), where we applied only the *Non-Dominated Sorting Genetic Algorithm (NSGA-II)* (Deb et al. 2002), here we consider two additional algorithms. The single-objective *Genetic Programming (GP)* algorithm from related work (Linsbauer et al. 2014), and the *Strength Pareto Evolutionary Algorithm (SPEA2)* (Zitzler et al. 2001), which is characterized by its external archive used to create the fronts of non-dominated solution in each generation. The GP algorithm was applied to serve as a baseline comparison. Our GP algorithm uses a weighted measure of precision and recall with same weight for both values, called F_1 based on information retrieval theory (Manning et al. 2008). The implementation is the same as in our previous work, for further details, refer to Linsbauer et al. (2014). SPEA2 was added because it is widely used with NSGA-II (Coello et al. 2007) and commonly applied in search-based software engineering approaches (Harman et al. 2012). For the experimentation we used ECJ Framework.[3] The parameter

[2]http://www.inf.ufpr.br/gres/IS/MORevEngFMs.zip.

[3]http://cs.gmu.edu/~eclab/projects/ecj/.

Table 4 Case studies overview

System	#F	#P	LoC	#Nodes	#Edges
ArgoUML	11	256	264K–344K	49	114
DPL	6	16	282–473	12	27
GOL	15	65	874–1.9K	12	24
VOD	11	32	4.7K–5.2K	7	11
ZipMe	7	32	5K–6.2K	29	60
MM-V1	5	3	2.1K	5	4
MM-V2	6	6	2.3K–2.4K	6	6
MM-V3	7	12	2.3K–2.5K	9	12
MM-V4	8	24	2.6K–2.9K	10	14
MM-V5	9	48	2.7K–3.8K	14	25
MM-V6	10	96	2.8K–4.1K	22	43
MM-V7	13	240	2.9K–4.3K	31	70

#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

settings used to configure the algorithms are shown in Table 3. We performed 30 independent runs for each 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, and running on a Linux platform.

4.2 Case Studies

To evaluate the proposed approach we used the case studies presented in Table 4. ArgoUML is an open source tool for UML modelling (Couto et al. 2011). Draw Product Line (DPL), briefly presented in our running example, is a small drawing application. Game Of Life (GOF) is a customizable game. Video On Demand (VOD) implements video-on-demand

Table 5 Average runtime per run

System	GP			NSGA-II			SPEA2		
	min	sec	msec	min	sec	msec	min	sec	msec
ArgoUML		25	275	13	59	911	17	35	300
DPL		23	512		35	802	1	30	583
GOL		56	934	2	15	991	3	3	531
VOD			906		1	882		2	247
ZipME		1	131	1	50	576	2	57	495
MM-V1		4	816		10	468		20	258
MM-V2		8	502		15	321		38	451
MM-V3		17	212		22	325		45	628
MM-V4		29	327		30	624		58	208
MM-V5		52	962		59	342	1	44	310
MM-V6	1	27	442	3	28	598	4	18	747
MM-V7	2	35	35	10	53	762	11	2	444

min = minutes, sec = seconds, msec = milliseconds streaming. ZipMe is an application for files compression. MobileMedia (MM) is an application to manipulate media files, such as photo, music, and video, on mobile devices. In our evaluation we used seven versions of MobileMedia (Figueiredo et al. 2008).

5 Results and Analysis

The average runtime per run of each algorithm is presented in Table 5. GP is the fastest algorithm in all case studies, respectively followed by NSGA-II and SPEA2. As expected, the multi-objective algorithms performed slower than the single-objective GP because of the computation to compose the Pareto fronts in each generation. Nonetheless, next we elaborate on the advantages of a multi-objective approach for our reverse engineering task.

For the analysis of results obtained by the algorithms we computed two different sets of solutions. Our terminology is based on our previous work (see Assunção et al. 2014) and standard multi-objective optimization literature (Coello et al. 2007).

- *Pareto Front True* (*PF_{True}*): since we do not know the real Pareto Front True, we use PF_{True} as an approximation of the best solutions. This set consists of the best solutions reached by all algorithms for each case study. These best solutions are found by merging all the solutions of all runs of the three algorithms together, then leaving only the non-dominated solutions. The cardinality of PF_{True} for each case study is presented in the second column of Table 6.
- *Pareto Front Known* (*PF_{Known}*): this set contains the best solutions reached by each algorithm for each case study. To compute PF_{Known} we merged all the solutions of all runs for each algorithm and then we keep only the non-dominated solutions. The cardinality of PF_{Known} for each case study and each algorithm is presented in the fourth column of Table 6.

5.1 Answering RQ1

Recall that RQ1's purpose is to find what the benefits are of a multi-objective perspective for reverse engineering FMs. We do so by comparing solutions obtained from multi-objective algorithms against solutions from a single-objective algorithm. From Table 6 we can observe that in seven case studies there is only one single solution in PF_{True}, so all the three objectives could be optimized independently. For the other five case studies with more than one solution there are conflicts among the objectives. Taking into account the seven versions of Mobile Media, we can observe that the three objectives became conflicting in MM-V6. Besides, we can note that MM-V6 and MM-V7 have a large number of solutions in comparison with the other case studies with cardinality larger than one in PF_{True}. We found out that this happens because MM-V6 introduced big changes in the source-code of MobileMedia (Figueiredo et al. 2008). As explained in Figueiredo et al. (2008), MM-V6 and MM-V7 introduced the two alternative features `Music` and `Video`, and the mandatory `Photo` feature was made optional leading to a big impact on the whole system. These changes lead to a larger number of nodes and edges, see Table 4, which makes the dependency graph more complex and impacts the number of solutions found.

For the analysis on the ability of each algorithm to find non-dominated solutions we consider two values presented in Table 6: (i) the number of solutions found by each algorithm, i.e. PF_{known}, that are in PF_{True}, fifth column; and (ii) the average number of solutions found per run that are/were in PF_{True}, sixth column. To illustrate the meaning of these

Table 6 Pareto fronts

System	PF_{True} cardinality	Search Algorithm	PF_{known} cardinality	# of solutions in PF_{True}	Average in PF_{True} per run
ArgoUML	4	GP	2	2 (50 %)	0.966
		NSGA-II	3	3 (75 %)	1.9
		SPEA2	2	2 (50 %)	2.0
DPL	1	GP	1	1 (100 %)	0.033
		NSGA-II	1	1 (100 %)	0.166
		SPEA2	1	1 (100 %)	0.066
GOL	8	GP	6	0 (0 %)	0.0
		NSGA-II	8	8 (100 %)	0.966
		SPEA2	6	5 (62 %)	0.933
VOD	1	GP	1	0 (0 %)	0.0
		NSGA-II	1	1 (100 %)	1.0
		SPEA2	1	1 (100 %)	1.0
ZipME	4	GP	1	1 (25 %)	1.0
		NSGA-II	4	4 (100 %)	3.033
		SPEA2	3	3 (75 %)	3.0
MM-V1	1	GP	1	1 (100 %)	0.5
		NSGA-II	1	1 (100 %)	0.5
		SPEA2	1	1 (100 %)	0.566
MM-V2	1	GP	1	1 (100 %)	0.3
		NSGA-II	1	1 (100 %)	0.366
		SPEA2	1	1 (100 %)	0.433
MM-V3	1	GP	2	0 (0 %)	0
		NSGA-II	1	1 (100 %)	0.333
		SPEA2	1	1 (100 %)	0.333
MM-V4	1	GP	1	1 (100 %)	0.066
		NSGA-II	1	1 (100 %)	0.333
		SPEA2	1	1 (100 %)	0.3
MM-V5	1	GP	1	1 (100 %)	0.1
		NSGA-II	1	1 (100 %)	0.233
		SPEA2	1	1 (100 %)	0.166
MM-V6	42	GP	5	1 (2 %)	0.033
		NSGA-II	42	42 (100 %)	4.66
		SPEA2	20	13 (30 %)	1.033
MM-V7	801	GP	5	3 (0.37 %)	0.333
		NSGA-II	796	760 (92 %)	36.0
		SPEA2	155	61 (7 %)	3.833

two values we consider ArgoUML, which has the PF_{True} composed of four solutions. For this case study GP found two solutions that are in PF_{True}, NSGA-II found three solutions, and SPEA2 two. On average GP was able to find almost one solution in PF_{True} per run, what is expected because it is a single-objective approach. On the other hand NSGA-II found on average 1.9 solutions per run, and SPEA2 2.0 solutions. For this case study GP is able to find good solutions in comparison with the multi-objective algorithms, NSGA-II found the largest amount of solutions in PF$_{True}$ after the thirty runs, but SPEA2 finds more non-dominated solutions on average in each run.

Regarding the number of solutions in PF$_{True}$, fifth column of Table 6, we can observe that the three algorithms have the same results for five case studies: DPL, MM-V1, MM-V2, MM-V4, and MM-V5. NSGA-II and SPEA2 outperform GP for two case studies: VOD and MM-V3. NSGA-II found more solutions in PF$_{True}$ for five case studies: ArgoUML, GOL, ZipMe, MM-v6, and MM-V7. On the other hand, considering the average of solutions in PF$_{True}$ per run, sixth column of Table 6, the three algorithms have the same results for four case studies: ArgoUML, ZipMe, MM-V1, and MM-v2. NSGA-II and SPEA2 are better than GP in five case studies: GOL, VOD, MM-V3, MM-V4, and MM-V5. NSGA-II outperform GP and SPEA2 in three case studies: DPL, MM-V6, and MM-V7.

In summary, from Table 6 we can observe that GP can only have similar results of NSGA-II and SPEA2 in two case studies and never reached the best results. NSGA-II and SPEA2 reached the same results and outperform GP in four case studies, and in six case studies NSGA-II is better than GP and SPEA2.

RQ1 Discussion From our set of case studies we identified that five of them have conflicting objectives. This means that when we get better values for one objective, then another objective is penalized, leading to a set of possible good solutions with different trade-offs. In such situation the multi-objective algorithms NSGA-II and SPEA2 are better than the single-objective GP. From the results we observed that on average the multi-objective algorithms found a set of solutions per run, on the other hand the single-objective algorithm is able to find only one. Furthermore, considering the set of solutions obtained after the 30 runs, GP still was not competitive against the multi-objective algorithms. GP tends to find in each run the same single solution, not exploring other parts of the search space, which is done by the multi-objective algorithms. For seven case studies the three objectives are not conflicting, however NSGA-II and SPEA2 found the same good solutions found by GP. In summary, we can conclude that a multi-objective approach is the best choice to reverse engineer FMs considering our case studies.

5.2 Answering RQ2

Recall that RQ2 aims at comparing the performance of algorithms NSGA-II and SPEA2 for the three selected objective functions. An interesting characteristic to be analysed is the position of the solutions on the search space. Figure 6 presents the three graphs for each case study with more than one solution in PF$_{Known}$. The first column of graphs presents the view of objectives precision and recall, the second column the objectives of variability safety and precision, and the third column presents variability safety and recall. As already analysed above, for all these case studies NSGA-II reached a larger amount of solutions than SPEA2. However, in the graphs we can observe that despite the smaller number of solutions, SPEA2 has solutions spread over a very similar area on search space explored by NSGA-II. On the previous analysis of Table 6 we observed that in six case studies NSGA-II reached better solutions than SPEA2, probably because of the larger number of solutions returned by the former algorithm. On the other hand, the smaller amount of solutions reached by SPEA2 can help in situations such as observed in the case studies MM-V6 and MM-V7, where a large number of solutions was returned and the decision maker has to choose only one to be used in practice.

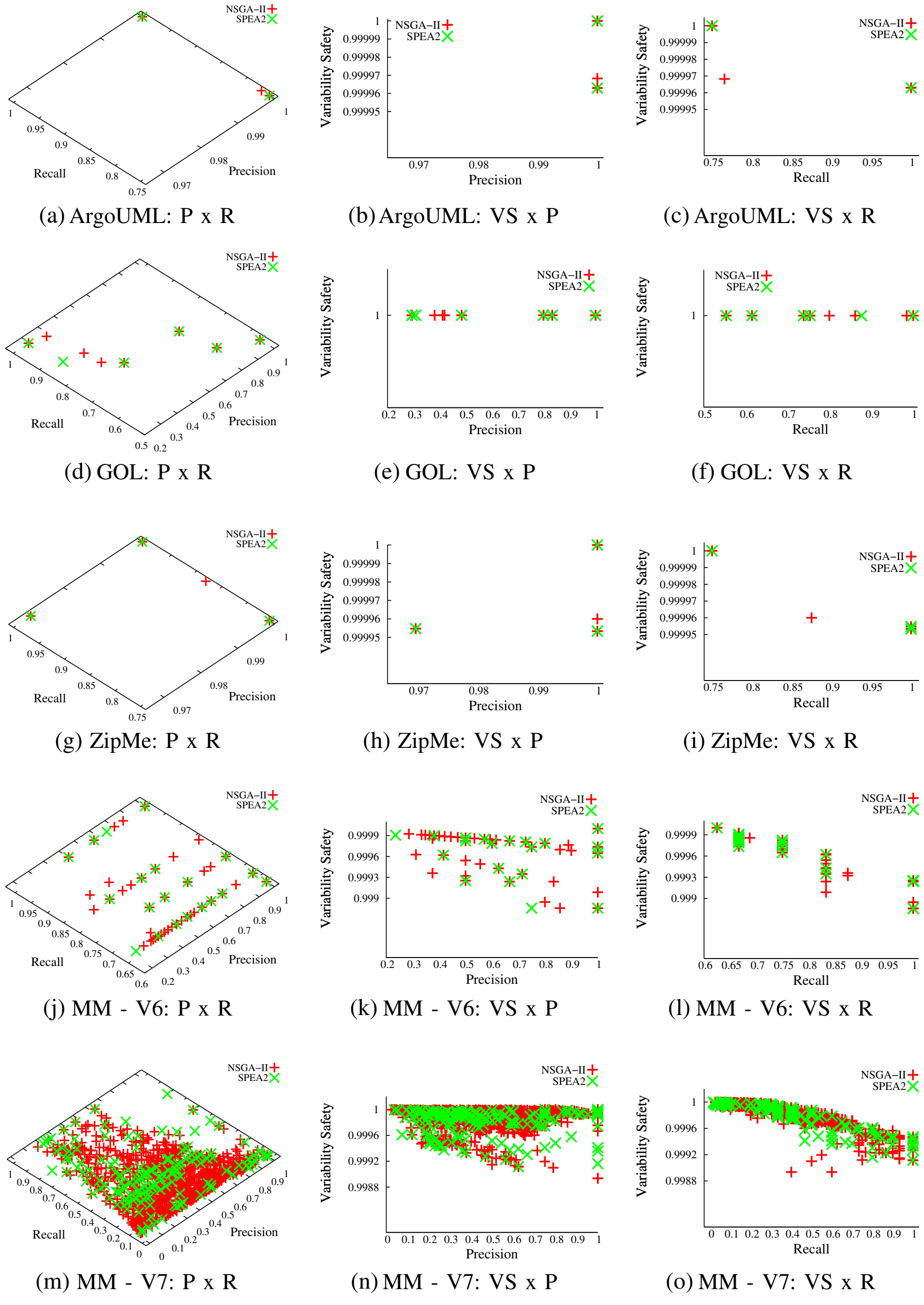


Fig. 6 Solutions on the search space

Statistical Analysis To reason about the differences between NSGA-II and SPEA2 we used the well known quality indicator called Hypervolume (Zitzler et al. 2003). Table 7 presents in the second and third columns the average of Hypervolume and standard

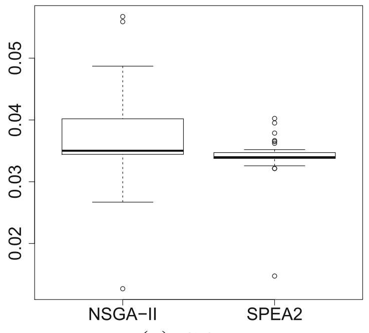
Table 7 Hypervolume and effect size

System	Hypervolume		Wilcoxon	$\hat{A}_{12}$ Effect Size	
	NSGA-II	SPEA2	p-value	NSGA-II	SPEA2
ArgoUML	0.0036 (0.0005)	0.0035 (0.0000)	1.61E-01	46.67	53.33
DPL	0.0064 (0.0025)	0.0071 (0.0017)	1.42E-01	56.56	43.44
GOL	0.0365 (0.0080)	**0.0340 (0.0041)**	**3.62E-02**	34.22	**65.78**
VOD	0.0010 (0.0000)	0.0010 (0.0000)	NA*	50.00	50.00
ZipME	0.0038 (0.0000)	0.0038 (0.0000)	NA*	50.00	50.00
MM-V1	0.0050 (0.0042)	0.0037 (0.0032)	1.69E-01	40.56	59.44
MM-V2	0.0097 (0.0075)	0.0083 (0.0073)	4.33E-01	44.50	55.50
MM-V3	0.0121 (0.0086)	0.0109 (0.0083)	4.33E-01	44.33	55.67
MM-V4	0.0124 (0.0089)	0.0104 (0.0083)	3.49E-01	43.22	56.78
MM-V5	0.0126 (0.0089)	0.0129 (0.0088)	5.06E-01	45.06	54.94
MM-V6	0.0399 (0.0128)	0.0435 (0.0123)	1.78E-01	60.17	39.83
MM-V7	0.1125 (0.0015)	**0.1038 (0.0059)**	**1.86E-09**	4.78	**95.22**

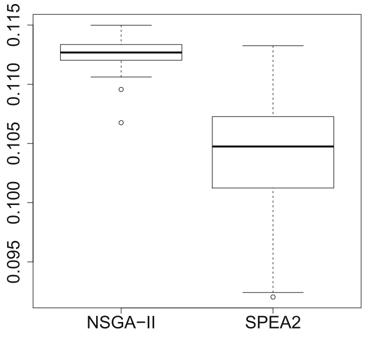
*NA = Not Available, because the two sets of values are identical

deviation, in parentheses, for the 30 runs. To compute the Hypervolume the reference point for all case studies was P=1.1, R=1.1, and VS=1.1. Since our problem is a maximization problem, then lower values of Hypervolume are better. To check statistical difference we applied the Wilcoxon test (Bergmann et al. 2000). The p-value obtained for each case study is presented on the fourth column of Table 7. To corroborate our analysis we also compute the effect size with the Vargha-Delaney's $\hat{A}_{12}$ statistic (Vargha and Delaney 2000), used for assessing randomized algorithms in Software Engineering (Arcuri and Briand 2014), presented on the last two columns of Table 7.

From the results on Table 7 we observe that there is a difference between both algorithms only for the case studies GOL and MM-V7. The boxplots for these two case studies are presented in Fig. 7. In these two systems the best algorithm was SPEA2. The results for the case studies VOD and ZipME are exactly the same. Despite some differences in the average Hypervolume for the other case studies, they are statistically similar.



(a) GOL



(b) MM-V7

Fig. 7 Hypervolume boxplots

RQ2 Discussion Regarding the number of good solutions found on average and after the 30 runs, NSGA-II outperformed SPEA2. However, both are able to widely explore the search space. Using the well-know Hypervolume indicator we observed that in ten case studies there are no significant differences. In the two case studies with significant difference, the algorithm SPEA2 was the best. In conclusion, both algorithms are good to solve the problem of reverse engineering of FMs using our three objectives.

5.3 Answering RQ3

Recall that the purpose of RQ3 was to explain how our approach could be used in practice. Let us now illustrate how. In Table 8 we present the values of the three objectives for all solutions that compose the sets PF_{known}. The exceptions are case studies MM-V6 and MM-V7. For these two case studies we selected those solutions with the value 1.0 for at least one of the objectives. Here the goal is to analyse qualitatively the solutions reached by the algorithms NSGA-II and SPEA2.

As mentioned before, the solutions of NSGA-II and SPEA2 are the same for seven case studies, namely DPL, VOD, MM-V1, MM-V2, MM-V3, MM-V4, and MM-V5. For the remaining case studies it is possible to observe how conflicting the objectives can be. For example, in ArgoUML we can observe that the conflict occurs only between recall and variability safety, since in all solutions have the value 1.0 for precision. A similar situation happens for GOL, but in this case study the variability safety is 1.0 in all solutions. For the case studies ZipMe, MM-V6 and MM-V7 there are solutions with different trade-offs among the three objectives. These solutions, with different values for each objective, help the software engineers in the decision making. They can decide which measure is more important and then select the corresponding solution.

To illustrate this process of selecting one solution, in Fig. 8 we present four FMs from MM-V7 obtained by SPEA2, their tree-like structure and cross-tree constraints. We selected these FMs based on two criteria: *i)* solutions with value equal to 1.0 for at least one objective, and *ii)* solutions with the best value for a second objective. For example, The solution presented in Fig. 8a has P = 1.0, satisfying the first criteria, and R = 0.8000000119 the best value of recall among solutions with P = 1.0, satisfying the second criteria.

In the FM presented in Fig. 8a we have 192 valid configurations and all of them are desired products, so the value of precision is 1.0. However, the total number of desired products for this case study is 240, so the recall of this FM is not 1.0. The value of variability safety equal to 0.999162264 means that there are 324 broken dependencies from the dependency graph on the valid configurations. The FMs in Fig. 8b and c have the best values for variability safety, however with very low value of recall. In the FM of Fig. 8b only four valid configurations are possible mainly because of the constraint `"Include_CopyMedia EXCLUDES Capture_Photo"` what makes all the configurations after the third level of the tree invalid, because the feature `Include_CopyMedia` is mandatory of `Capture_Photo`. In Fig. 8c only 16 valid configurations are possible, because the constraint `"Include_Video EXCLUDES Include_Music"` makes invalid all the configurations below the second level of the tree, since the feature `Include_Music` is mandatory child of feature `Include_Video`. In Fig. 8d we have an FM with recall equal to 1.0, which means that all desired products are valid configurations here; however, it denotes more valid configurations than the input, hence decreasing the value of precision. In this same FM, the number of broken dependencies is 624. These are four examples of multi-objective solutions that are given to the software engineers, so that they can analyse the trade-offs and select the one that best meets their needs.

Table 8 Non-dominated solutions

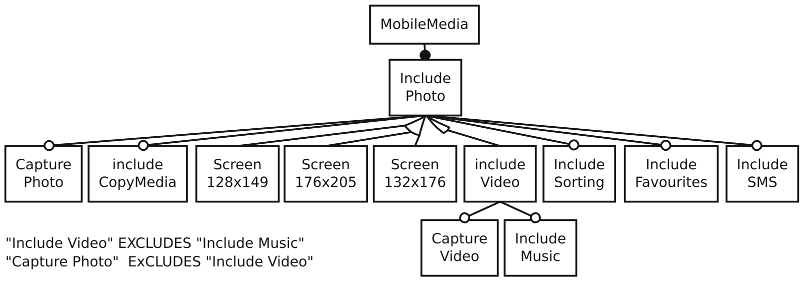
System	NSGA-II			SPEA2		
	P	R	VS	P	R	VS
ArgoUML	1.0	1.0	0.9999629741.0	1.0	1.0	0.9999629741.0
	1.0	0.765625	0.9999682636	1.0	0.75	1.0
	1.0	0.75	1.0			
DPL	1.0	1.0	1.0	1.0	1.0	1.0
GOL	1.0	0.5538461804	1.0	1.0	0.5538461804	1.0
	0.8333333135	0.6153846383	1.0	0.8333333135	0.6153846383	1.0
	0.8000000119	0.7384615541.0	1.0	0.8000000119	0.7384615541.0	1.0
	0.4851485193	0.7538461685	1.0	0.4851485193	0.7538461685	1.0
	0.4193548262	0.8000000119	1.0	0.3081081212	0.8769230843	1.0
	0.4117647111.0	0.8615384698	1.0	0.2941176593	1.0	1.0
	0.380952388	0.9846153855	1.0			
	0.2941176593	1.0	1.0			
VOD	1.0	1.0	1.0	1.0	1.0	1.0
ZipMe	1.0	1.0	0.9999533669	1.0	1.0	0.9999533669
	1.0	0.875	0.9999600288	1.0	0.75	1.0
	1.0	0.75	1.0	0.9696969986	1.0	0.99995478
	0.9696969986	1.0	0.99995478			
MM-V1	1.0	1.0	1.0	1.0	1.0	1.0
MM-V2	1.0	1.0	1.0	1.0	1.0	1.0
MM-V3	1.0	1.0	1.0	1.0	1.0	1.0
MM-V4	1.0	1.0	1.0	1.0	1.0	1.0
MM-V5	1.0	1.0	1.0	1.0	1.0	1.0
MM-V6	1.0	1.0	0.9988592489	1.0	1.0	0.9988592489

Table 8 (continued)

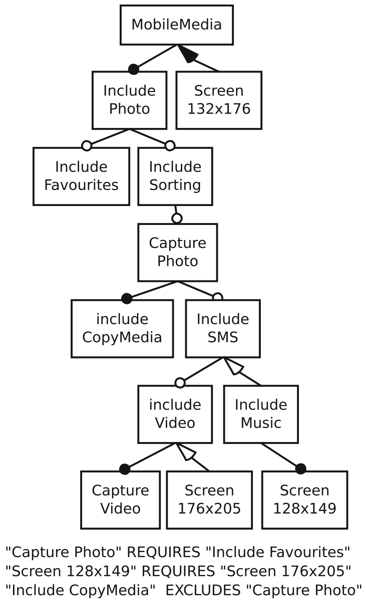
System	NSGA-II			SPEA2		
	P	R	VS	P	R	VS
	1.0	0.8333333135	0.9990873991.0	1.0	0.75	0.9996489996
	1.0	0.75	0.9996489996	1.0	0.6666666865	0.9997367497
	1.0	0.6666666865	0.9997367497	1.0	0.625	1.0
	1.0	0.625	1.0	0.75	1.0	0.9988592489
	0.8571428657	1.0	0.9988592489	0.6666666865	1.0	0.9992394992
	0.8000000119	1.0	0.9989469989	0.5	1.0	0.9992541243
	0.6666666865	1.0	0.9992394992		...	
	0.5	1.0	0.9992541243			
		...				
MM-V7	1.0	0.6000000238	0.9989360141.0	1.0	0.8000000119	0.999162264
	1.0	0.400000006	0.9989360141.0	1.0	0.6000000238	0.9993559012
	1.0	0.3625000119	0.999666002	1.0	0.4666666687	0.9994112262
	1.0	0.2333333343	0.9997554055	1.0	0.400000006	0.9997431758
	1.0	0.1000000015	0.9998858559	1.0	0.200000003	0.9999021622
	1.0	0.0666666701.0	0.9999266217	1.0	0.0833333358	0.9999217298
	1.0	0.0500000007	0.9999347748	1.0	0.0500000007	0.9999347748
	1.0	0.0333333351.0	0.9999510811.0	1.0	0.0333333351.0	0.9999510811.0
	1.0	0.0250000004	1.0	1.0	0.0250000004	0.9999673874
	0.625	1.0	0.9991153834	1.0	0.020833334	0.9999804324
	0.46875	1.0	0.9992723315	1.0	0.0166666675	1.0
	0.375	1.0	0.9993762841.0	0.625	1.0	0.9991153834
	0.25	1.0	0.9994374327	0.46875	1.0	0.9992723315
	0.1666666716	1.0	0.9994700453	0.375	1.0	0.9993347031.0

Table 8 (continued)

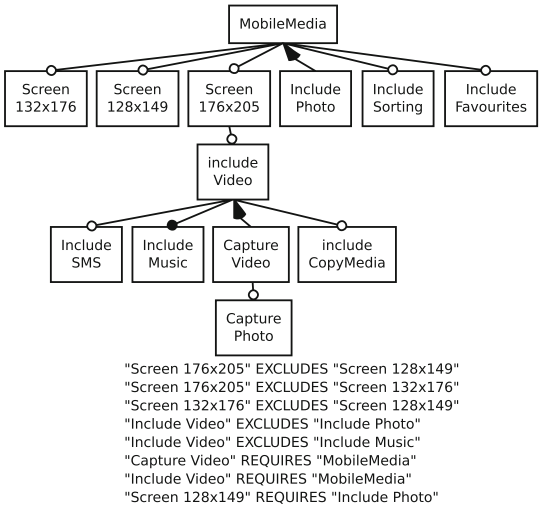
System	NSGA-II			SPEA2		
	P	R	VS	P	R	VS
	0.8000000119	0.0333333351.0	1.0	0.2727272809	1.0	0.9993751723
	0.75	0.0500000007	1.0	0.25	1.0	0.9994374327
		...		0.2307692319	1.0	0.9994393142
				0.2083333284	1.0	0.9994700453
				0.75	0.0500000007	1.0
					...	



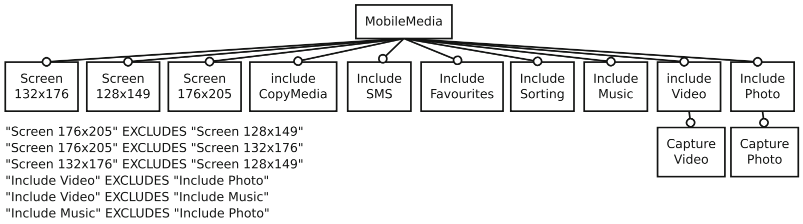
(a) P=1.0, R=0.8000000119, VS=0.999162264



(b) P=1.0, R=0.0166666675, VS=1.0


MobileMedia
Screen 132x176
Screen 128x149
Screen 176x205
Include Photo
Include Sorting
Include Favourites
include Video
Include SMS
Include Music
Capture Video
include CopyMedia
Capture Photo
"Screen 176x205" EXCLUDES "Screen 128x149"
"Screen 176x205" EXCLUDES "Screen 132x176"
"Screen 132x176" EXCLUDES "Screen 128x149"
"Include Video" EXCLUDES "Include Photo"
"Include Video" EXCLUDES "Include Music"
"Capture Video" REQUIRES "MobileMedia"
"Include Video" REQUIRES "MobileMedia"
"Screen 128x149" REQUIRES "Include Photo"

(c) P=0.75, R=0.0500000007, VS=1.0


MobileMedia
Screen 132x176
Screen 128x149
Screen 176x205
include CopyMedia
Include SMS
Include Favourites
Include Sorting
Include Music
include Video
Include Photo
Capture Video
Capture Photo
"Screen 176x205" EXCLUDES "Screen 128x149"
"Screen 176x205" EXCLUDES "Screen 132x176"
"Screen 132x176" EXCLUDES "Screen 128x149"
"Include Video" EXCLUDES "Include Photo"
"Include Video" EXCLUDES "Include Music"
"Include Music" EXCLUDES "Include Photo"

(d) P=0.625, R=1.0, VS=0.9991153834

Fig. 8 MM-V7 feature models

The use of dependency graphs to compute variability safety helps to identify another characteristic in the implementation artefacts, the existence of compilation errors or incoherences in the source code. For example, looking at the FM presented in Fig. 8a, the precision is 1.0, this means that all feature sets denoted by the FM belong to the set of desired feature sets, but there are configurations that even though are valid, they do have broken dependencies. In the dependency graph of this case study there is the dependency $Include\_SMS \Rightarrow Capture\_Photo$, which means that in the source code the feature `Include_SMS` depends on `Capture_Photo`; however, in the set of desired configurations there are 72 configurations that have the feature `Include_SMS` but do not have `Capture_Photo`, decreasing the value of variability safety.

RQ3 Discussion As discussed before, the main advantage of a multi-objective approach is to find a set of good solutions regarding different trade-off among the objectives. In a practical point of view, these solutions allow the software engineer to reason about the different measures used to evaluate the FMs. At the end, the software engineers can select the solution that best fits his/her needs. In addition, multi-objective algorithms can return solutions that can call the attention of the decision maker to characteristics that are not considered in the beginning of the optimization process. We discussed this point about the possible identification of inconsistencies between the source code and the feature sets used as input, e.g. invalid references. To identify such inconsistencies was not the goal of the software engineers when starting the optimization process, however our approach further enables software engineers to reason about inconsistencies by looking at the solutions with different values among the objectives.

5.4 Threats to Validity

The first threat to validity identified was the parameter settings for the algorithms. We addressed this threat by using conventional values for our parameters as we have done in our previous work (Assunção et al. 2015); however, we increased the population size to allow the algorithms to generate more individuals.

The second threat regards to the used case studies. Despite using only twelve case studies, these systems are from different domains and have different sizes. We argue that they are representative to evaluate our approach.

A third threat concerns the baseline comparison. To the best of our knowledge, our approach is the first to use information from implementation artefacts and a multi-objective perspective to reverse engineer feature models. Then as baseline we use the genetic programming algorithm from our previous work (Assunção et al. 2015). Since this single-objective algorithm returns a single solution per run we used averages per run and a set composed with the 30 runs to reason about the differences in comparison with the multi-objective algorithm.

The fourth threat is related to the set of measures we used to evaluate the reverse engineered FMs. Different measures and metrics could produce FMs with other characteristics. However, we believe the three measures we considered are well designed for our goals of representing the actual set of product variants and take into account the source-code structure.

The last threat refers to the validation of the reversed engineered feature models by developers. Certainly, because there are no canonical representations of feature models, domain knowledge can play a significant role when choosing among different feature models. However, this threat is mitigated by considering real case studies that have been extensively studied by us an others for different purposes.

6 Related Work

Search-based techniques have been applied in a wide range of SPL activities such as feature selection, architectural improvement, SPL testing, and feature model construction (Harman et al. 2014; Lopez-Herrejon et al. 2015). To reverse engineer FMs, search-based algorithms were explored in the work of Lopez-Herrejon et al. (2012, 2015). In this work an evolutionary algorithm uses as input a set of desired feature sets and, as objective, a function that maximizes the number of the desired feature sets contained in a feature model disregarding any surplus feature sets that the model could denote (Lopez-Herrejon et al. 2012, 2015). This work was extended by Thianniwet and Cohen Thianniwet and Cohen (2015) to reverse engineer complex features models. The authors designed a fitness function to balance both additional and missing products and create a representation and evolution operators to support complex cross-tree constraints. But none of these works includes the information from the implementation artefacts or a multi-objective perspective.

Feature sets were also used in the work of Haslinger et al. (2011, 2013). The authors used an ad hoc algorithm to identify patterns in the selected and not selected features mapped in parent-child relations of feature models (Haslinger et al. 2011). An extension was done to consider the CTCs requires and excludes (Haslinger et al. 2013). Again the authors do not consider the implementation artefacts.

Czarnecki and Wasowski used a set of propositional logic formulas as input to the reverse engineering task (Czarnecki and Wasowski 2007; She et al. 2014). They propose an ad hoc algorithm to extract multiple feature models from single propositional logic formula preserving the original formulas and reducing redundancy (Czarnecki and Wasowski 2007). Recently, the algorithm has been improved based on CNF and DNF constraints (She et al. 2014). In contrast with our work their starting point are configuration files, documentation files, and constraints expressed in propositional logic instead of feature sets and source code.

Acher et al. proposed an interactive process to map each feature into a feature model, and then all the feature models are merged in a single feature model (Acher et al. 2012). In the work of Sannier et al. they consider product matrices from Wikipedia as start point (Sannier et al. 2013). These matrices can have other values besides select or not select. From these matrices an analysis is performed to identify variability patterns. The authors only mention about the benefits of exploiting that information to extract models such as feature models.

Genetic Programming is also the focus of other pieces of work on software development. An example is the paper of Chan et al. where the authors deal with the problem of product planning and customer satisfaction using a method based on GP (Chan et al. 2011).

There is extensive work using multi-objective optimization algorithms in the field of search-based software engineering. For instance, software module clustering, integration testing, testing resource allocation, protocol tuning, software project scheduling, software project effort estimation, and software defect prediction (Harman et al. 2012; Yao 2013). However, they do not address reverse engineering of FMs.

7 Future Directions

In this section we describe some research opportunities and trends on the task of reverse engineering of feature models.

Note that the variability safety measure is not restricted to implications obtained from a dependency graph. A similar metric can be computed on arbitrary constraints, as long as it can be determined whether they hold or not. This is important because constraints may not only be provided in the form of a dependency graph as a result of a code analysis tool, but also from other sources like constraints defined by domain experts. Generally speaking this measure expresses the conformance of the found candidate feature models to a set of given constraints.

In this sense, the exploration of different sources of information is a future direction. For instance, the use of test cases is a possible target, since they are usually available in most projects. The comments in the source code are also a possible target for new research. Design models, like UML diagrams, are another interesting starting point to be considered.

Besides the use of different artefacts, another research opportunity is to design new measures and metrics to evaluate the feature models. For example, the use of graph similarity could be applied in dependency graphs of each variant. Perhaps the use of semantic similarity measures among elements is a research opportunity. Non-functional characteristics of systems can be another measure to be included in the reverse engineering process. We also identified a lack of complementary metrics to evaluate and validate the reengineered SPL artefacts, these metrics could facilitate the process of comparison between obtained results and expected ones.

There is a lack of tools that support the reverse engineering of feature model in practice. Recently, tools like BUT4Reuse (Martinez et al. 2015), a framework that provides technologies for leverage commonality and variability of software artefacts, have appeared. However, more tools with different purpose and focus are needed. For example, they should cover different programming languages, different artefact types, etc.

Interactive approaches that include the user in the loop are an alternative strategy to extract information that sometimes is only present in the user's mind. Furthermore, interactive approaches allow the user to guide the reverse engineering process to some preferred directions. Hence performing empirical studies that involve users providing feedback on the generated models is an avenue for future research.

From the identified related work we did not find any strategy to deal with ambiguity in the input artefacts. In other words, the strategies work only with well defined and structured inputs. To deal with incomplete or unsound input is an open challenge.

8 Concluding Remarks

This paper presents an approach to reverse engineer feature models from a set of feature sets and a dependency graph, extracted from the source code. The set of feature sets is used to compute the precision and recall of the feature models. The dependency graph is used to compute the variability safety, i.e. how well-formed the feature model is regarding the implementation artefacts. Furthermore, the approach takes a multi-objective perspective to deal with the three different measures independently, supporting the software engineers in the decision making process.

To evaluate the proposed approach we designed an experiment to answer questions regarding the benefits and advantages of using our multi-objective approach, the performance of two multi-objective evolutionary algorithms for reverse engineering of FMs, and about the practical use of a multi-objective perspective by the software engineers. The experiment was conducted with twelve case studies and used NSGA-II and SPEA2, and a single-objective GP algorithm.

The results indicate that the multi-objective algorithms are better than the single-objective one, which was expected, because in five case studies the three measures are in conflict. From the number of solutions obtained after 30 runs and the average of good solutions found per run, we observe that the algorithm NSGA-II outperforms the algorithm SPEA2. However, the statistical test using the Hypervolume values indicates they do not have a difference in ten case studies, and in the other two the algorithm SPEA2 has better Hypervolume values.

Reverse engineering of feature models has recently received an increasing attention from the research community on SPLs; however, our work also revealed several research avenues worthy of further investigation. Among them, we can mention: using different types of constraints in addition to the source code dependencies, exploiting new sources of information like non-functional properties, devising new metrics as objective functions, dealing with ambiguity in input artefacts, and developing more robust and interactive tools and techniques.

Acknowledgments This work was supported by Austrian Science Fund (FWF): P 25289-N15, and the Brazilian Agencies CAPES: 007126/ 2014-00 and CNPq: 453678/2014-9 and 305358/2012-0.

References

Acher M, Cleve A, Perrouin G, Heymans P, Vanbeneden C, Collet P, Lahire P (2012) On extracting feature models from product descriptions. In: International workshop on variability modelling of software-intensive systems (vamos), pp 45–54

Arcuri A, Briand L (2014) A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability 24(3):219–250. doi:10.1002/stvr.1486

Assunção WK, Lopez-Herrejon RE, Linsbauer L, Vergilio SR, Egyed A (2015) Extracting variability-safe feature models from source code dependencies in system variants. In: Genetic and evolutionary computation conference (GECCO). ACM, New York, NY, USA, pp 1303–1310. doi:10.1145/2739480.2754720

Assunção WKG, Colanzi TE, Vergilio SR, Pozo A (2014) A multi-objective optimization approach for the integration and test order problem. Inf Sci 267:119–139. doi:10.1016/j.ins.2013.12.040

Assunção WKG, Vergilio SR (2014) Feature location for software product line migration: a mapping study. In: 18th software product line conference - 2nd international workshop on REverse variability engineering (REVE), pp 1–8. doi:10.1145/2647908.2655967

Batory DS, Sarvela JN, Rauschmayer A (2004) Scaling step-wise refinement. IEEE Trans Softw Eng 30(6):355–371

Benavides D, Segura S, Cortés AR (2010) Automated analysis of feature models 20 years later: a literature review. Inf Syst 35(6):615–636

Benavides D, Segura S, Trinidad P, Cortés AR (2007) FAMA: tooling a framework for the automated analysis of feature models. In: Pohl K, Heymans P, Kang KC, Metzger A (eds) International workshop on variability modelling of software-intensive systems (VaMoS), Lero Technical Report, vol 2007-01, pp 129–134

Bergmann R, Ludbrook J, Spooren WPJM (2000) Different outcomes of the Wilcoxon-Mann-Whitney test from different statistics packages. Am Stat 54(1):72–77. doi:10.2307/2685616

Chan KY, Kwong CK, Wong TC (2011) Modelling customer satisfaction for product development using genetic programming. J Eng Des 22(1):55–68. doi:10.1080/09544820902911374

Coello CAC, Lamont G, van Veldhuizen D (2007) Evolutionary algorithms for solving multi-objective problems, 2nd edn. Genetic and Evolutionary Computation. Springer, Berlin

Couto MV, Valente MT, Figueiredo E (2011) Extracting software product lines: a case study using conditional compilation. In: Conference on software maintenance and reengineering (CSMR), pp 191–200. doi:10.1109/CSMR.2011.25

Czarnecki K, Wasowski A (2007) Feature diagrams and logics: there and back again. In: International software product line conference (SPLC). IEEE Computer Society, pp 23–34

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

Figueiredo E, Cacho N, Sant'Anna C, Monteiro M, Kulesza U, Garcia A, Soares S, Ferrari F, Khan S, Castor Filho F, Dantas F (2008) Evolving software product lines with aspects: an empirical study on design stability. In: International conference on software engineering (ICSE). ACM, New York, NY, USA, pp 261–270. doi:10.1145/1368088.1368124

Fischer S, Linsbauer L, Lopez-Herrejon RE, Egyed A (2014) Enhancing clone-and-own with systematic reuse for developing software variants. In: International conference on software maintenance and evolution (ICSME)

Harman M, Jia Y, Krinke J, Langdon WB, Petke J, Zhang Y (2014) Search based software engineering for software product line engineering: a survey and directions for future work. In: 18Th international software product line conference - volume 1, SPLC '14. ACM, New York, NY, USA, pp 5–18. doi:10.1145/2648511.2648513

Harman M, Mansouri SA, Zhang Y (2012) Search-based software engineering: trends, techniques and applications. ACM Comput Surv 45(1):11:1–11:61. doi:10.1145/2379776.2379787

Haslinger EN, Lopez-Herrejon RE, Egyed A (2011) Reverse engineering feature models from programs' feature sets. In: Working conference on reverse engineering (WCRE), pp 308–312

Haslinger EN, Lopez-Herrejon RE, Egyed A (2013) On extracting feature models from sets of valid feature combinations. In: International conference fundamental approaches to software engineering (FASE), pp 53–67

Kang K, Cohen S, Hess J, Novak W, Peterson A (1990) Feature-Oriented Domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, SEI CMU

van d. Linden FJ, Schmid K, Rommes E (2007) Software product lines in action: the best industrial practice in product line engineering. Springer

Linsbauer L, Lopez-Herrejon RE, Egyed A (2013) Recovering traceability between features and code in product variants. In: International software product line conference (SPLC), pp 131–140

Linsbauer L, Lopez-Herrejon RE, Egyed A (2014) Feature model synthesis with genetic programming. In: International symposium on search based software engineering (SSBSE), pp 153–167

Lopez-Herrejon RE, Galindo JA, Benavides D, Segura S, Egyed A (2012) Reverse engineering feature models with evolutionary algorithms: an exploratory study. In: International symposium on search based software engineering (SSBSE), pp 168–182

Lopez-Herrejon RE, Linsbauer L, Egyed A (2015) A systematic mapping study of search-based software engineering for software product lines. J Inf Softw Technol. doi:10.1016/j.infsof.2015.01.008

Lopez-Herrejon RE, Linsbauer L, Galindo JA, Parejo JA, Benavides D, Segura S, Egyed A (2015) An assessment of search-based techniques for reverse engineering feature models. J Syst Softw 103(0):353–369. doi:10.1016/j.jss.2014.10.037

Manning CD, Raghavan P, Schütze H (2008) Introduction to information retrieval. Cambridge University Press

Martinez J, Ziadi T, Bissyandé TF, Klein J, Traon YL (2015) Bottom-up adoption of software product lines: a generic and extensible approach. In: International conference on software product line (SPLC), pp 101–110. doi:10.1145/2791060.2791086

Sannier N, Acher M, Baudry B (2013) From comparison matrix to variability model: The wikipedia case study. In: International conference on automated software engineering (ASE). IEEE, pp 580–585

Segura S, Galindo J, Benavides D, Parejo JA, Cortés AR (2012) BeTTy: benchmarking and testing on the automated analysis of feature models. In: Eisenecker UW, Apel S, Gnesi S (eds) International workshop on variability modelling of software-intensive systems (VaMoS). ACM, pp 63–71

She S, Lotufo R, Berger T, Wasowski A, Czarnecki K (2011) Reverse engineering feature models. In: International conference on software engineering (ICSE). ACM, pp 461–470

She S, Ryssel U, Andersen N, Wasowski A, Czarnecki K (2014) Efficient synthesis of feature models. Inf Softw Technol 56(9):1122–1143

Thianniwet T, Cohen M (2015) Splrevo: optimizing complex feature models in search based reverse engineering of software product lines. In: North american search based software engineering symposium (NasBASE)

Vargha A, Delaney H (2000) A critique and improvement of the cl common language effect size statistics of mcgraw and wong. J Educ Behav Stat 25(2):101–132

Weston N, Chitchyan R, Rashid A (2009) A framework for constructing semantically composable feature models from natural language requirements. In: International software product line conference (SPLC), pp 211–220

Yao X (2013) Some recent work on multi-objective approaches to search-based software engineering. Springer, Berlin, pp 4–15. doi:10.1007/978-3-642-39742-4_2

Zitzler E, Laumanns M, Thiele L (2001) SPEA2: improving the strength pareto evolutionary algorithm. Tech. Rep. 103, Gloriastrasse 35, CH-8092 Zurich, Switzerland

Zitzler E, Thiele L, Laumanns M, Fonseca CM, da Fonseca VG (2003) Performance assessment of multiobjective optimizers: an analysis and review. IEEE Trans Evol Comput 7:117–132

Wesley K. G. Assunção received a bachelor's degree in Information Systems from Faculdade Sul Brasil in 2006 and the MSc degree in 2012 from Federal University of Paraná (UFPR), Brazil. He is currently PhD candidate at the Post-graduation Program in Informatics of Federal University of Paraná (UFPR), being a member of Research Group on Software Engineering - GRES. His areas of interest are: Software Testing, Software Product Lines, Search Based Software Engineering and Multi-Objective Evolutionary Algorithms.

Roberto E. Lopez-Herrejon is an Associate Professor at the Department of Software Engineering and Information Technology of the École de Technologie Supérieure of the University of Quebec in Montreal, Canada. Prior he was a senior postdoctoral researcher at the Johannes Kepler University in Linz, Austria. He was an Austrian Science Fund (FWF) Lise Meitner Fellow (2012–2014) at the same institution. From 2008 to 2014 he was an External Lecturer at the Software Engineering Masters Programme of the University of Oxford, England. From 2010 to 2012 he held an FP7 Intra-European Marie Curie Fellowship sponsored by the European Commission. He obtained his Ph.D. from the University of Texas at Austin in 2006, funded in part by a Fulbright Fellowship sponsored by the U.S. State Department. From 2005 to 2008, he was a Career Development Fellow at the Software Engineering Centre of the University of Oxford sponsored by Higher Education Founding Council of England (HEFCE). His main expertise is in software customization, software product lines, and search based software engineering.

Lukas Linsbauer is currently a PhD student at the Institute for Software Systems Engineering at the Johannes Kepler University (JKU) in Linz, Austria under the supervision of Prof. Alexander Egyed and Dr. Roberto Erick Lopez-Herrejon. He received his master's degree in computer science from the JKU after only four years of study for each of which he received a merit scholarship. His research interests are in traceability, software product lines, variability modeling and management, and highly variable and configurable systems.

Silvia R. Vergilio received the MS (1991) and DS (1997) degrees from University of Campinas, UNICAMP, Brazil. She is currently at the Computer Science Department at the Federal University of Paraná, Brazil, where she has been a faculty member since 1993. She has been involved in several projects and her research interests are in the areas of Software Engineering, such as: software testing, software quality and software metrics.

Alexander Egyed heads the Institute for Software Systems Engineering (ISSE) at the Johannes Kepler University, Austria. He received his Doctorate from the University of Southern California, USA and previously worked many years in industry before joining academia. Dr. Egyed was recognized among the Top 10 scholars in software engineering and his work has received numerous awards.