Reverse Engineering Feature Models With Evolutionary Algorithms: An Exploratory Study

Roberto E. Lopez-Herrejon[1], José A. Galindo[2], David Benavides[2], Sergio Segura[2], and Alexander Egyed[1]

[1] Institute for Systems Engineering and Automation
Johannes Kepler University Linz, Austria
roberto.lopez@jku.at, alexander.egyed@jku.at
[2] Department of Computer Languages and Systems
University of Seville, Spain
jagalindo@us.es, benavides@us.es, sergiosegura@us.es

Abstract. Successful software evolves, more and more commonly, from a single system to a set of system variants tailored to meet the similiar and yet different functionality required by the distinct clients and users. Software Product Line Engineering (SPLE) is a software development paradigm that has proven effective for coping with this scenario. At the core of SPLE is variability modeling which employs Feature Models (FMs) as the de facto standard to represent the combinations of features that distinguish the systems variants. Reverse engineering FMs consist in constructing a feature model from a set of products descriptions. This research area is becoming increasingly active within the SPLE community, where the problem has been addressed with different perspectives and approaches ranging from analysis of configuration scripts, use of propositional logic or natural language techniques, to ad hoc algorithms. In this paper, we explore the feasibility of using Evolutionary Algorithms (EAs) to synthesize FMs from the feature sets that describe the system variants. We analyzed 59 representative case studies of different characteristics and complexity. Our exploratory study found that FMs that denote proper supersets of the desired feature sets can be obtained with a small number of generations. However, reducing the differences between these two sets with an effective and scalable fitness function remains an open question. We believe that this work is a first step towards leveraging the extensive wealth of Search-Based Software Engineering techniques to address this and other variability management challenges.

1 Introduction

Successful software evolves not only to adapt to emerging development technologies but also to meet the clients and users functionality demands. For instance, it is not uncommon to find academic, professional, or community variants of commercial and open source applications such as editing, modelling, programming or many other development tools, where each variant provides different functionality. Thus the distinction between variants is described in terms of their *features*, that we define as increments in program functionality [1].

In practice, the most common scenario starts with a first system and forks a new independent development branch everytime a new variant with different feature combinations is required. Unfortunately, this approach does not scale as the number of features and their combinations increases even slightly [2]. *Software Product Line Engineering (SPLE)* is an emerging software development paradigm that advocates a disciplined yet flexible approach to maximize reuse and customization in all the software artifacts used throughout the entire development cycle [2–5]. The driving goal of SPLE is to create *Software Product Lines (SPLs)* that realize the different software system variants in an effective and efficient manner. However, evolving SPLs from existing and invidividually-developed system variants is not an easy endeavor. A crucial requirement is accurately capturing the variability present in SPLs and representing it with *Feature Models (FMs)* [3, 6], the de facto standard for variability modeling. Current research has focused on extracting FMs from configuration scripts [7], propositional logic expressions [8], natural language [9], and ad hoc algorithms [10, 11].

Previous work from some of the authors has shown *Evolutionary Algorithms (EAs)* as an attractive alternative to synthesize FMs that are hard to analyze [12–14]. In this paper, we explore the feasibility of using EAs to reverse engineer FMs from the feature sets that describe the system variants and thus help coping with the evolution scenario – from system variants to SPLs – described above. Our study analyzed 59 representative feature sets from publicly available case studies of different sizes and complexity. For the implementation of our approach, we used a specific instantiation of the evolutionary algorithm ETHOM [13], integrated into the open source tool BeTTy [12].

We devised two fitness functions. With them we identified a trade-off between accuracy of the obtained feature model (the required feature sets vs of the obtained feature sets) and number of generations. That is, proper supersets of the desired feature sets can be obtained with a small number of generations. However, these supersets contain a large surplus of feature sets. In contrast, to reduce such surplus does require more generations. We believe our work is a first step towards leveraging the extensive wealth of Search-Based Software Engineering techniques to address many pressing variability management challenges.

2 Feature Models and Running Example

Feature models are the de facto standard to model the common and variable features of an SPL and their relationships [6], and thus represent the set of feature combinations that the products of the SPL can have. Features are depicted as labeled boxes and are connected with lines to other features with which they relate, collectively forming a tree-like structure.

A feature can be classified as: *mandatory* if it is part of a program whenever its parent feature is also part, and *optional* if it may or may not be part of a program whenever its parent feature is part. Mandatory features are denoted with filled circles while optional features are denoted with empty circles both at the child end of the feature relations. Features can be grouped into: *or* relation

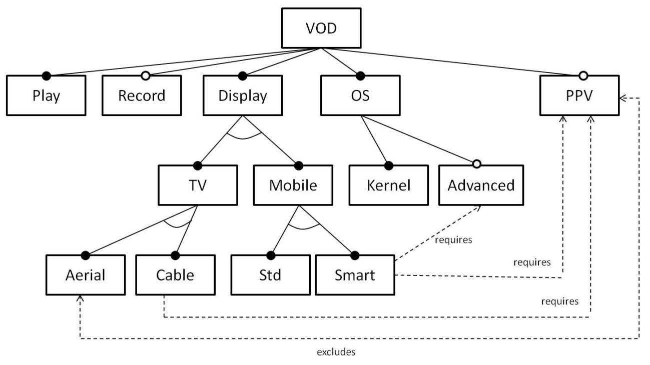

Fig. 1: Video On Demand SPL Feature Model

whereby one or more features of the group can be selected, and *alternative* relation where exactly one feature can be selected. These relations are depicted as filled arcs and empty arcs respectively.

Besides the parent-child relations, features can also relate across different branches of the feature model in the so called *Cross-Tree Constraints (CTC)* [15]. The typical examples of this kind of relations are: *i) requires* relation whereby if a feature `A` is selected a feature `B` must also be selected, and *ii) excludes* relation whereby if a feature `A` is selected then feature `B` *must not* be selected. In a feature model, these latter relations are depicted with doted single-arrow lines and doted double-arrow lines respectively. Additionally, more general cross-tree constraints can be expressed using propositional logic [15].

Figure 1 shows the feature model of our running example, a hypothetical SPL of Video On Demand systems. The root feature of a SPL is always included in all programs, in this case the root feature is `VOD`. Our SPL also has feature `Play` which is mandatory, in this case it is included in all programs because its parent feature `VOD` is always included. Feature `Record` is optional, thus it may be present or not in our product line members, the same holds for feature `PPV` (Pay Per View). Feature `Display` is also mandatory and like features `Play` and `OS` (Operating System) they are included in all our programs because their parent, the root, is always included. Features `TV` and `Mobile` constitute an alternative relation, meaning that our programs can have either one of them but only one. Similarly, features `Aerial` and `Cable`, and features `Std` (Standard) and `Smart` are respectively in alternative relations. Notice as well, `requires` relations between features `Smart` and `Advanced`, meaning that if a product contains feature `Smart` it must also contain feature `Advanced`. The same is the case between `Smart` and `PPV`, and between `Cable` and `PPV`. Finally, features `Aerial` and `PPV` are in an `excludes` relation meaning that both cannot be present in the same product. Next we provide a more formal definition of feature sets and their relation with products[3].

[3] Adapted from [15] where feature sets are referred to as configurations.

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

Definition 2. *A product is valid if the selected and not-selected features adhere to all the contraints imposed by the feature model.*

For example, the feature set `fs=[{VOD, Play, Display, TV, Aerial, OS, Kernel}, {Record, Cable, Mobile, Std, Smart, Advanced, PPV}]` is valid. As another example, a feature set with features `TV` and `Mobile` is not valid because it violates the constraint of the or relation which establishes that these two features cannot appear selected together in the same configuration.

Table 1 depicts the feature sets of our running example VOD SPL of Figure 1. The ticks represent the selected features whereas empty entries represent not selected features. The column headers are abbreviations of the feature names.

We reiterate that in this paper we address the problem of reverse engineering feature models out of a lists of feature sets such as that in Table 1. It should be stressed though that a list of feature sets can be denoted by different feature models. In other words, the mapping from feature sets to feature models is one-to-many. This characteristic makes EAs specially attractive as different potential feature models can be analyzed and ranked according to distinct criteria. Next we present a short description the EAs infrastructure we employed.

Table 1: Feature Sets of VOD Software Product Line

FSet	**VOD**	**Play**	**Rec**	**Disp**	**OS**	**TV**	**Mob**	**Sm**	**Std**	**Ker**	**Adv**	**Aer**	**Cab**	**PPV**
FS1	✓	✓	✓	✓	✓	✓				✓			✓	✓
FS2	✓	✓		✓	✓	✓				✓			✓	✓
FS3	✓	✓	✓	✓	✓	✓				✓		✓		
FS4	✓	✓		✓	✓	✓				✓		✓		
FS5	✓	✓	✓	✓	✓		✓		✓	✓				✓
FS6	✓	✓	✓	✓	✓		✓		✓	✓				
FS7	✓	✓		✓	✓		✓		✓	✓				
FS8	✓	✓		✓	✓		✓		✓	✓				✓
FS9	✓	✓	✓	✓	✓	✓				✓	✓		✓	✓
FS10	✓	✓		✓	✓	✓				✓	✓		✓	✓
FS11	✓	✓	✓	✓	✓	✓				✓	✓	✓		
FS12	✓	✓		✓	✓	✓				✓	✓	✓		
FS13	✓	✓	✓	✓	✓		✓		✓	✓	✓			✓
FS14	✓	✓	✓	✓	✓		✓		✓	✓	✓			
FS15	✓	✓		✓	✓		✓		✓	✓	✓			✓
FS16	✓	✓		✓	✓		✓		✓	✓	✓			
FS17	✓	✓	✓	✓	✓		✓	✓		✓	✓			✓
FS18	✓	✓		✓	✓		✓	✓		✓	✓			✓

3 Evolutionary Algorithms with ETHOM

We relied on the EA tool *ETHOM (Evolutionary algoriTHm for Optimized feature Models)* to implement our approach [13]. In this section, we present a basic overview of the main characteristics of ETHOM, and in next section we describe how it was applied for reverse engineering feature models.

3.1 Encoding

One of the most salient characteristics of ETHOM is its encoding of individuals (i.e chromosomes) which is specifically tailored to represent feature models. A feature model is thus represented as an array divided in two parts, one for the tree structure of the feature model and one for its cross-tree constraints. The enconding of the feature model of our running example in Figure 1 is shown in Figure 2.

The chromosomes of the structural part of the tree are tuples of the form `<PR,CN>` where:

- `PR` denotes the type of relationship a feature has with its parent. It can be `M` for mandatory, `Op` for optional, `Alt` for alternative, and `Or` for or relation.
- `CN` denotes the number of children of the feature.

In addition, the order of these tuples is determined by a Depth-First Traversal (DFT) starting from the root. It should also be noted that the root of the feature model is not encoded. For example, the chromosome at entry with DFT value 6, corresponds to feature `Mobile` that is an alternative (`PR` value is `Alt`) feature of its parent (feature `Display`), and has two children (`CN` value is 2), namely `Std` and `Smart`. As another example, chromosome at entry with DFT value 11. This chromosome encodes feature `Advanced`, with an optional relation with its parent feature `OS` (`PR` value is `Op`), and with no children (`CN` value is 0).

The chromosomes of the cross-tree constraints are tuples of the form `<TC,O,D>` where:

- `TC` encodes the type of cross-tree constraint. An `R` value denotes a requires constraint whereas an `E` value denotes an excludes constraint.
- `O` encodes the origin feature of the cross-tree constraint represented with the corresponding DFT value.
- `D` encodes the destination feature of the cross-tree constraint represented with the corresponding DFT value.

For example, the tuple `<E,4,12>` encodes the excludes cross-tree constraint between features `Aerial` (DFT value 4) and `PPV` (DFT value 12). As another example, the tuple `<R,8,11>` encode a requires cross-tree constraint between features `Smart` (DFT value 8) and `Advanced` (DFT value 11).

Tree

DFT	0	1	2	3	4	5	6	7	8	9	10	11	12
	M,0	Op,0	M,2	Alt,2	Alt,0	Alt,0	Alt,2	Alt,0	Alt,0	M,2	M,0	Op,0	Op,0

CTC

E,4,12	R,5,12	R,8,12	R,8,11

Feature DFT order
Play=0, Record=1, Display=2, TV=3, Aerial=4
Cable=5, Mobile=6, Std=7, Smart=8, OS=9
Kernel=10, Advanced=11, PPV=12

Fig. 2: ETHOM Encoding of Video On Demand feature model

3.2 Initial Population and Selection

There are different alternatives in the literature to randomly generate feature models [12,16]. ETHOM uses the following configuration parameters:

- Population size.
- Number of features.
- Percentage of cross-tree constraints.
- Maximum branching factor, defined as the maximum number of subfeatures of a feature, considering all the types of relationships.
- Percentage of mandatory relations.
- Percentage of optional relations.
- Percentage of `Alternative` relations.
- Percentage of `Or` relations.

There are multiple alternatives to implement selection in the EA literature [17]. The current version of ETHOM provides roulette wheel and tournament selection [13].

3.3 Crossover.

There are multiple alternatives to implement crossover in the EA literature [17]. The current version of ETHOM provides one-point and uniform crossover [13]. Fig. 3 depicts an example of the application of one-point crossover in ETHOM. The process starts by selecting two parent chromosomes to be combined. For each array in the chromosomes, the tree and CTC arrays, a random point is chosen (so-called crossover point). Finally, the offspring is created by copying the content of the arrays from the beginning to the crossover point from one parent and the rest from the other one. Notice that the characteristics of this encoding guarantee a fixed size for the individuals.

3.4 Mutation.

ETHOM defines four mutation operators that are applied with the probability set in the configuration. The mutation operators available are:

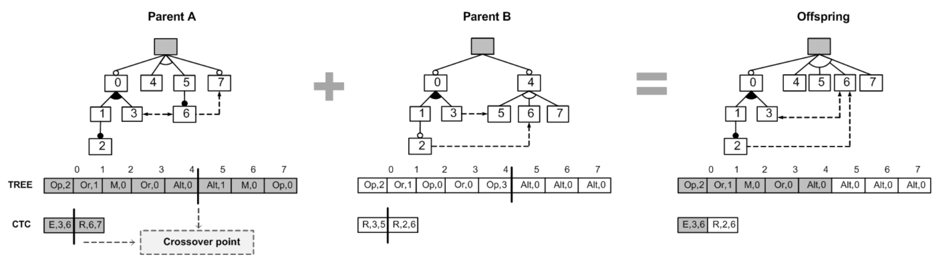
Parent A
Parent B
Offspring
TREE
CTC
Op,2 Or,1 M,0 Or,0 Alt,0 Alt,1 M,0 Op,0
E,3,6 R,6,7
Crossover point
Op,2 Or,1 Op,0 Or,0 Op,3 Alt,0 Alt,0 Alt,0
R,3,5 R,2,6
Op,2 Or,1 M,0 Or,0 Alt,0 Alt,0 Alt,0 Alt,0
E,3,6 R,2,6

Fig. 3: Example of one-point crossover in ETHOM [13].

- Operator 1. Changes randomly a relation between two features from one kind to any other kind. For example, from mandatory (`M`) to optional (`Op`) or from `Op` to `Alternative` (`Alt`).
- Operator 2. Changes the number of children `CN`, to a number selected from 0 to a maximum branching factor parameter set up in ETHOM.
- Operator 3. Changes the type of cross-tree constraint, from excludes to requires and vice versa.
- Operator 4. Changes either the origin or destination feature (with equal probability) of a cross-tree constraint. It is checked that the resulting CTC does not have the same origin and destination values.

It should be noted that with application of cross-over and mutation operators there is possibility of creating feature models that are not semantically correct. ETHOM provides mechanisms for their identification and repair. For further details, please consult [13].

4 Applying ETHOM for Extracting Feature Models

In this section, we describe the concrete instantiation of ETHOM developed for reverse engineering feature models and how it was used in the experimental setting of our exploratory study.

4.1 ETHOM Configuration Parameters and Fitness Functions

Based on our previous experience using ETHOM to generate hard-to-analyze feature models, we set up ETHOM remaining configuration parameters as shown in Table 2. Notice that, as mentioned in Section 3.4, infeasible individuals (i.e. semantically incorrect feature models), are replaced.

A crucial decision in our study was selecting an adequate fitness function. Overall, our goal is to obtain features models that produce exactly the set of products desired. Unfortunately, existing work on formal analysis of feature models indicates that finding the relations between feature models and their related product specifications (e.g. logic representation) can be a hard and expensive computational task [8, 15, 16]. For this reason, we decided to analyze two fitness

Table 2: ETHOM configuration parameters

Parameter	Value selected
Selection strategy	Roulette-wheel
Crossover strategy	One-point
Crossover probability	0.7
Mutation probability	0.01
Initial population size	100
Infeasible individuals	Replace
Maximum generations	25

functions, one that focuses on obtaining the desired feature sets disregarding any surplus of feature sets and one that does penalize surplus. We argue that studying both alternatives could provide some insights as to whether the extra cost of computing the penalty would yield a faster (i.e. less number of generations) and more accurate result.

Relaxed Fitness Function. The relaxed fitness function `FFRelaxed` maximizes the number of desired feature sets contained in a feature model disregarding any surplus feature sets that the model could denote. `FFRelaxed` is defined as follows:

$$FFRelaxed(sfs, fm) = |\{fs : sfs \mid validFor(fs, fm)\}|$$

Where `sfs` is the set of desired feature sets, `fs` is an individual feature set in `sfs`, `fm` a feature model, and `validFor(fs,fm)` is a function that receives a feature model and determines if a given feature set is contained in the set of feature sets represented by the feature model `fm`[4].

`FFRelaxed` is maximized to have as many feature sets from `sfs` as possible. Thus its maximum is the size of the set of desired feature sets `sfs`. However this function has a shortcomming, namely, that a feature model can contain more feature sets besides those in `sfs`. An ideal solution would then include all the feature sets in `sfs` and no more additional feature sets. However, reaching the maximum would not guarantee this. For instance, consider the feature sets of Table 1 and the feature model shown in Figure 1. If we apply `FFRelaxed`, we will get a value of 18 which is the maximum in this case since this is the number of feature sets we want the feature model to include and in this case this is exactly the set of feature sets represented by the feature model. However, if we change, for instance, the relationship between the root feature (`VOD`) and the feature `Play` from mandatory to optional, the number of feature sets represented by the feature model quickly raises from 18 to 36 but the value of the fitness function would remain the same since all the 18 desired feature sets are included but more

[4] For more details on the implementation of function `valid` please refer to [15].

are also included. The opposite could also eventually happen, i.e. that the feature model includes less feature sets than the ones expected. For instance, imagine that we add an excludes relationship between features `Record` and `Mobile` in the feature model of Figure 1. The total number of feature sets represented by the model will be reduced from 18 to 13 (feature sets FS5, FS6, FS13, FS14, and FS17 would not be included). If we apply now `FFRelaxed`, the result will be 13 not reaching the maximum of the fitness function in this case.

Strict Fitness Function. The strict fitness function `FFStrict` penalizes having more feature sets than those desired. `FFStrict` is maximized and its values can range from 0 up to the concrete number of feature sets we want the feature model to have. It is defined as follows:

$$FFStrict(sfs, fm) == \begin{cases} 0 & : \#products(fm) \neq |sfs| \\ FFRelaxed(sfs, fm) & : \#products(fm) = |sfs| \end{cases}$$

Where `sfs` is the set of desired feature sets and `#products(fm)` is a function that receives a feature model and returns the number of feature sets it represents [5]. `FFStrict` returns 0 in the case the number of feature sets represented by the feature model is not equal to the number of desired feature sets and the value of applying `FFRelaxed` otherwise. A potential shortcomming of this function is that it equally penalizes all feature models that have more feature sets than the number desired, irrespective of how big the differences are.

4.2 Experimental Setting

In order to assess our approach we analyzed case studies from the SPLOT website [18], a publicly available repository of feature models that contains both academic and real-life examples. We selected 59 representative feature models based on their number of features and number of products. The number of products ranged from 1 to 896 and the number of features from 9 to 27. We chose these thresholds so that the fitness functions analysed yield results within a reasonable amount of time using the available tooling support.

Figure 4 sketches the overall control flow for each selected feature model. First, we used FAMA [19], a tool for the analysis of feature models, to generate the feature sets represented by a given feature model. ETHOM follows a standard EA control flow. Using information gathered from FAMA (e.g. number of features and feature names) the initial population is created. The evaluation takes into account the feature sets of the products we are looking for and a fitness function, either `FFRelaxed` or `FFStrict`. If the EA finds an individual with the feature sets desired, it records the number of generations along with time elapsed to obtain them and stops. Otherwise, ETHOM then proceeds with the standard selection, crossover and mutation steps before continuing with the evaluation of the new generation. As shown in Table 2, there is also a stop criteria according to the maximum number of generations, 25 generations in our study.

[5] Implemented based on function *number of products* defined in [15].

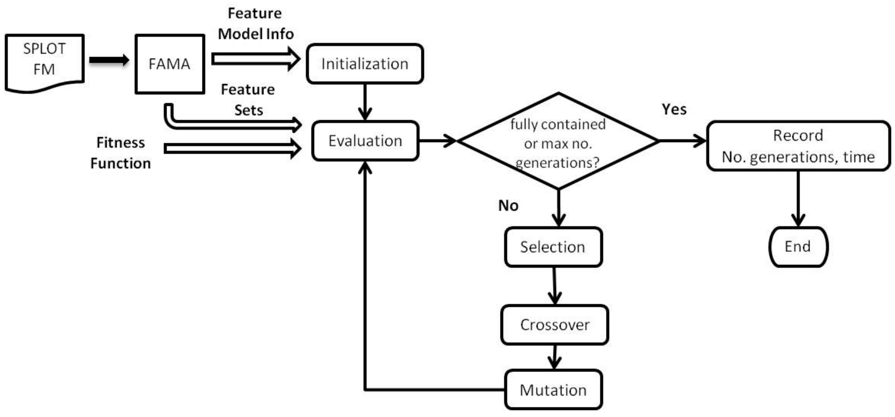

Fig. 4: Experiment Control Flow

4.3 Evaluation Results and Analysis

We performed 10 runs for each of the two fitness functions for each of the 59 feature models we selected in order to get average values. The evaluation of both functions started with the same initial populations. All the experiments were performed on a Intel Xeon E5620 ©with 16 cores running at 2.40GHz. This machine has 25 GB of shared RAM with other 5 machines inside a cloud, and was running CentOS 5 and Sun Java 1.6. Note that for our evaluation only one core was used per model. Execution time was measured using Java utilities.

Figure 5 summarizes the results obtained for `FFRelaxed`. In total, the algorithm reached the maximum of the fitness function in 94,64% of all the runs with an average execution time of 11 minutes. The average number of generations to reach a maximum was 5 with a standard deviation of 3,39. We found that there was only a model where the fitness function did not reach the maximum (within 25 generations) in any of the runs, shown in histogram of Figure 5(a). The percentage of cases where the maximum of the fitness function was reached within the first 2 generations was around 30%. This would suggest that our algorithm

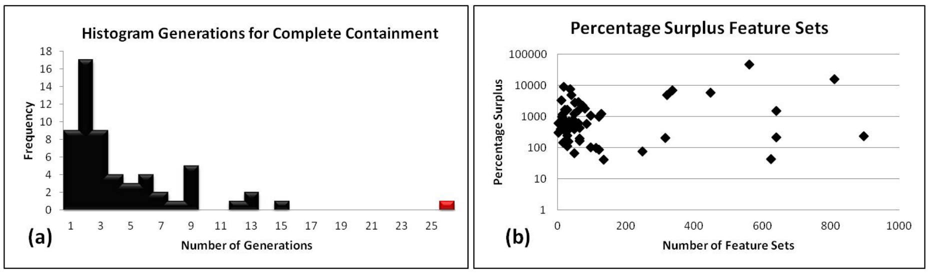

Fig. 5: FFRelaxed Results

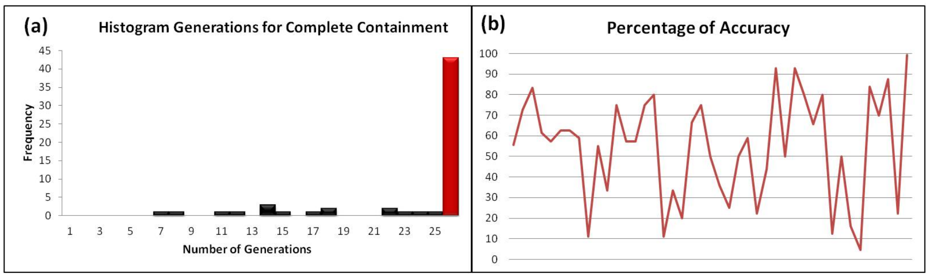

Fig. 6: FFStrict Results

is performing quite effectively. However, we found that the number of denoted feature sets in the evolved feature models was far from the expected number of feature sets. To gauge at this difference, we defined the following surplus metric:

$$Surplus(sfs, fm) = \frac{\#products(fm) - |sfs|}{|sfs|} \times 100 \tag{1}$$

This metric thus shows the percentage of increment or reduction of feature sets denoted by an evolved feature model. The differences obtained in our runs are shown in Figure 5(b). Please do note the logarithmic scale of the percentage surplus axis. On average, the value was 2401,24% of surplus feature sets with respect to the value expected. We did not find any feature model with the best score of the fitness function with less number of feature sets than `sfs`. We did not find any feature model with exactly the same number of feature sets either. With these data, we conjecture that our algorithm using `FFRelaxed` would get similar results to those of a random search because the number of generations to get a maximum of the fitness function is small and the surplus of the resulting feature model is high. It might also suggest that the fitness function is not guiding too much the algorithm and thus using random search of feature models would offer similar results. To confirm this conjecture, a proper and dedicated experiment comparing both approaches is called for, which is part of our future work.

Figure 6 summarizes our findings for `FFStrict`. With this fitness function 16 out of the 59 feature models reached a maximum within the limit of 25 generations, with an average of number of generations of 16,66 and a standard deviation of 5,56, see histogram in Figure 6(a). This would suggest that our algorithm is performing worse; however, it is important to highlight the accuracy obtained by the 43 feature models that did not reach the maximum. The accuracy here was defined as follows:

$$Accuracy(sfs) = bestFitnessAchieved(sfs)/|sfs| \times 100 \tag{2}$$

Where `sfs` is the set of desired feature sets, and function `bestFitnessAchieved` collects the best fitness value obtained in *all* the generations of the 10 runs evaluated for a sfs (i.e. each feature model of our study). Figure 6(b) shows the accuracy percentages (y-axis) plotted arbitrarily by increasing number of feature sets of the corresponding feature model (x-axis). On average, the accuracy was 54,81% but with a wide standard deviation value of 25,4%. This result shows that even though penalty of not having the exact number of expected feature sets may appear harsh, it contributed to hone in the search to yield good accuracy percentages.

5 Related Work

There is extensive and increasing literature in reverse engineering mostly from source code. The novelty of our work lies on the application of EAs for reverse engineering of variability. In this section, we shortly describe those works that focus on reverse engineering feature models from specifications and those that employ search-based techniques.

Recent work by Haslinger et al. proposes an ad hoc algorithm to reverse engineer feature models from feature sets [10]. It works by identifying occurrence patterns in the selected and not selected features that are mapped to parent-child relations of feature models. Currently, they do not address general feature models that can contain any type of cross-tree constraints. The main distinction with our work is that only one feature model can be reversed engineered, whereas in our approach we could provide different feature models (if they exist) as alternatives for the designers to choose from.

Work by Czarnecki and Wasowski study reverse engineering of feature models but from a set of propositional logic formulas [8]. They provide an ad hoc algorithm that can potentially extract from a single propositional logic formula multiple feature models but that tries to preserve the original formulas and reduce redundancies. Subsequent work by She et al. highlighted the limitations of this approach, namely problems selecting the parents of features and incompleteness [7]. They proposed a heuristic to address these two issues that complements dependency information with textual feature description. In contrast with our work, their starting point are configuration and documentation files.

Closer to our work is Acher et al. that also tackle the reverse engineering of feature models from feature sets [11]. The salient difference between our approaches is that their work maps each feature set into a feature model which are later merged in to a single feature model. This mapping and merge operation rely on propositional logic techniques and tools which can be computationally expensive. A more detailed comparison and analysis of the advantages and disadvantages of both approaches is part of our future work.

Finally, our exploratory study was inspired by and builds upon the previous work on BeTTy, a benchmark and testing framework for feature model analysis [12]. This framework is geared to generate feature models that are hard to analyze, be it in execution time or memory footprint.

6 FutureWork

We argue that our exploratory study has opened up several research venues on the application of SBSE techniques for variability management. The following are some areas we plan to pursue as future work.

Improvement of fitness functions. The cornerstone of our work is devising an adequate fitness function that contains the set of products required, as tight as possible, but still remains scalable. A possibility is to experiment with functions that consider feature model metrics [20].

Parameter landscape analysis. Currently, ETHOM is equally configured for all the feature models we used in our runs. A detailed analysis of the parameter landscape of the problem is duly called for. We plan to experiment with ETHOM's configuration parameters. The ultimate goal is to see whether any particular parameter configurations can yield better results and how they would scale for larger feature models and feature sets.

Variability-aware mutation operators. Currently ETHOM blindly applies mutation operators without any considerations of any potential variability implications, that is, how it could impact the set of products denoted by a feature model. We want to extend the set of mutation operators so that they consider the impact they may have on variability. We could then set up different probabilities so that they could be applied distinctly perhaps depending on the nature of the required set of products. Integrating the work on analysis of feature model changes is a starting point [16, 21].

Quality of feature models. So far the emphasis of our work has been on obtaining a feature model that denotes the required set of feature sets. However, as we mentioned, more than one feature model can denote the same set of feature sets. The question is now, towards which equivalent feature model should the search be directed to? We believe that quality metrics for feature models [20] as well as quantification of developers feedback could also be integrated [22] to help answer this question.

Novel applications. Software Product Line Engineering covers the entire development life cycle [4], from early design to deployment and maintenance. Thus, there are plenty of areas where SBSE could be potentially applied. A salient one is testing, where EA approaches could be tailored to consider variability implications in their search. A solid first step in that direction is the work by Segura et al. [21]. As another example, the area of fixing inconsistencies in models with variability [23, 24]. These new applications will in turn require more general feature model and problem encodings and effective fitness functions.

Comparative studies. As mentioned throughout the paper, we plan to perform a comparative study of basic search techniques (e.g. hill climbing), other EAs, and feature model composition (e.g. [11]) with our approach.

7 Conclusions

In this paper we explored the feasibility of EAs for reverse engineering feature models from feature sets. We devised two fitness functions that respectively focused on: *i)* getting the desired feature sets while disregarding any surplus (`FFRelaxed`), *ii)* getting the desired number of feature sets and then on the desired feature sets (`FFStrict`).

With these two functions we were able to identified a trade-off between accuracy of the obtained feature model (the required feature sets vs of the obtained feature sets) and number of generations. That is, proper supersets of the the desired feature sets can be obtained with a small number of generations. However, these supersets contain a large surplus of feature sets. In contrast, reducing such surplus does require more generations but still can yield good accuracy results. Despite this encouraging results, devising a fitness function that can reduce, if not eliminate, this trade-off is still an open question. We hope that this work has highlighted some of the the many potential areas where SBSE techniques can help tackle many open challenges in the realm of variability management.

The sources of our exploratory study can be downloaded from:
`http://www.lsi.us.es/~dbc/material/ssbse2012`

Acknowledgments

This research is partially funded by the Austrian Science Fund (FWF) project P21321-N15 and Lise Meitner Fellowship M1421-N15, and Marie Curie Actions - Intra-European Fellowship (IEF) project number 254965. It is also supported by the European Commission (FEDER) and Spanish Government under CICYT project SETI (TIN2009-07366) and by the Andalusian Government under THEOS project and Talentia scholarship. We would also like to thank Jules White and the MAGNUM research group for its support running the required experiments.

References

1. Zave, P.: Faq sheet on feature interaction http://www.research.att.com/ pamela/-faq.html.
2. Krueger, C.W.: Easing the transition to software mass customization. In van der Linden, F., ed.: PFE. Volume 2290 of Lecture Notes in Computer Science., Springer (2001) 282–293
3. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
4. Pohl, K., Bockle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer (2005)
5. van d. Linden, F.J., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer (2007)
6. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University (1990)
7. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In Taylor, R.N., Gall, H., Medvidovic, N., eds.: ICSE, ACM (2011) 461–470

8. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: There and back again. In: SPLC, IEEE Computer Society (2007) 23–34
9. Weston, N., Chitchyan, R., Rashid, A.: A framework for constructing semantically composable feature models from natural language requirements. In Muthig, D., McGregor, J.D., eds.: SPLC. Volume 446 of ACM International Conference Proceeding Series., ACM (2009) 211–220
10. Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: Reverse engineering feature models from programs' feature sets. In Pinzger, M., Poshyvanyk, D., Buckley, J., eds.: WCRE, IEEE Computer Society (2011) 308–312
11. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., Lahire, P.: On extracting feature models from product descriptions. [25] 45–54
12. Segura, S., Galindo, J., Benavides, D., Parejo, J.A., Cortés, A.R.: BeTTy: benchmarking and testing on the automated analysis of feature models. [25] 63–71
13. Segura, S., Parejo, J.A., Hierons, R.M., Benavides, D., Ruiz-Cortés, A.: ETHOM: An Evolutionary Algorithm for Optimized Feature Models Generation. Technical Report ISA-2012-TR-01, Applied Software Engineering Research Group. Department of Computing Languages and Systems. University of Sevilla., ETSII. Avda. de la Reina Mercedes s/n (2 2012)
14. Segura, S., Parejo, J., Hierons, R., Benavides, D., A.Ruiz-Cortés: Automated generation of hard feature models using evolutionary algorithms. Journal Submission (2012) Under review.
15. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. **35**(6) (2010) 615–636
16. Thüm, T., Batory, D.S., Kästner, C.: Reasoning about edits to feature models. In: ICSE, IEEE (2009) 254–264
17. Eiben, A., Smith, J.: Introduction to Evolutionary Computing. Springer Verlag (2003)
18. Generative Software Development Lab. Computer Systems Group, University of Waterloo, C.: Software Product Line Online Tools(SPLOT). http://www.splot-research.org/ (2012)
19. : FAMA Tool Suite. http://www.isa.us.es/fama/ (2012)
20. Bagheri, E., Gasevic, D.: Assessing the maintainability of software product line feature models using structural metrics. Software Quality Journal (2010)
21. Segura, S., Hierons, R.M., Benavides, D., Cortés, A.R.: Automated metamorphic testing on the analyses of feature models. Information & Software Technology **53**(3) (2011) 245–258
22. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P.: Reverse engineering architectural feature models. In Crnkovic, I., Gruhn, V., Book, M., eds.: ECSA. Volume 6903 of Lecture Notes in Computer Science., Springer (2011) 220–235
23. Lopez-Herrejon, R.E., Egyed, A.: Fast abstract. Searching the variability space to fix model inconsistencies: A preliminary assessment. In: Third International Symposium Search Based Software Engineering SSBSE 2011.
24. Lopez-Herrejon, R.E., Egyed, A.: Towards fixing inconsistencies in models with variability. [25] 93–100
25. Eisenecker, U.W., Apel, S., Gnesi, S., eds.: Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings. In Eisenecker, U.W., Apel, S., Gnesi, S., eds.: VaMoS, ACM (2012)