Feature Model Synthesis with Genetic Programming

Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed

Software Systems Engineering
Johannes Kepler University
Linz, Austria
lukas.linsbauer@jku.at, roberto.lopez@jku.at, alexander.egyed@jku.at
http://www.jku.at/isse

Abstract. Search-Based Software Engineering (SBSE) has proven successful on several stages of the software development life cycle. It has also been applied to different challenges in the context of Software Product Lines (SPLs) like generating minimal test suites. When reverse engineering SPLs from legacy software an important challenge is the reverse engineering of variability, often expressed in the form of Feature Models (FMs). The synthesis of FMs has been studied with techniques such as Genetic Algorithms. In this paper we explore the use of Genetic Programming for this task. We sketch our general workflow, the GP pipeline employed, and its evolutionary operators. We report our experience in synthesizing feature models from sets of feature combinations for 17 representative feature models, and analyze the results using standard information retrieval metrics.

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

1 Introduction

Search-Based Software Engineering (SBSE) is an emerging research area that focuses on the application of search-based optimization techniques to problems in software engineering [1]. Examples of these techniques are hill-climbing, simulated annealing, genetic algorithms, or swarm optimization [2]. SBSE has been applied at several stages of the software development life cycle, but most prominently for software testing [3].

Genetic Programming (GP) is a form of evolutionary computation that employs a tree-based representation of computer programs whose fitness is determined on how well the encoded programs solve a computational problem [4]. However, it is also used to solve mathematical problems like symbolic regression where the goal is to find a formula that best explains a set of sample points.

Software Product Lines (SPLs) are families of related software systems where each product has a different combination of features [5]. Most of the industrial applications of SPLs start from a set of system variants, each providing a different

C. Le Goues and S. Yoo (Eds.): SSBSE 2014, LNCS 8636, pp. 153–167, 2014.
© Springer International Publishing Switzerland 2014

set of feature combinations, that must be reverse engineered into a SPL [6]. A crucial step in this reverse engineering effort is obtaining a *feature model* [7] – the de facto standard to represent the valid feature combinations – that denotes all the desired feature combinations. Similarly to the use of GP for symbolic regression, we use GP to find a feature model that best explains a set of product variants.

Feature models are important to model the variability of software systems. They describe which features can be combined and which ones cannot in order to form products. However, often such information is not available, for example when companies maintain portfolios of legacy software product variants that are the result of ad hoc methods like clone and own, where a new product variant is created by copying an existing variant and adapting it to fit another customer's requirements, or by making existing variants highly configurable [8]. Only once the number of variants or possible configurations has become unmanagable companies decide to reverse engineer an SPL from their existing product variants [9]. The first step to this is often the reverse engineering of a feature model. She et al. provide two algorithms to solve this problem, a task that has been shown to be NP-hard [10].

A recent publication by Harman et al. summarizes the developments in the application of genetic programming and genetic improvement for reverse engineering tasks and proposes new research directions where both SBSE techniques could be employed [11]. Among these directions is SPLs for which the authors sketch some potential research venues. In this paper we make, to the best of our knowledge, the first application of genetic programming in the realm of SPLs. We extend our previous work [12] where we employed a genetic algorithm for reverse engineering feature models. We show that genetic programming provides a more accurate representation of the feature models which, provided with more specialized operators, can produce better reverse engineering results.

2 Feature Models and Running Example

Feature models have become the *de facto* standard for modelling the feature combinations for SPLs [7]. They depict features and their relationships collectively forming a tree-like structure. The nodes of the tree are the features denoted as labelled boxes, and the edges represent the relationships among them. Figure 1 shows the feature model of our running example, the *Graph Product Line (GPL)* [13], a standard SPL that has been extensively used as a case study. In GPL, a product is a collection of algorithms applied to directed or undirected graphs.

In a feature model, each feature (except the root) has one parent feature and can have a set of child features. A child feature can only be included in a feature combination of a valid product if its parent is included as well. The root feature is always included. There are four kinds of feature relationships: i) *Mandatory features* are selected whenever their respective parent feature is selected. They are depicted with a filled circle. For example, features `GraphType` and `Algorithms`,

Feature	P0	P1	P2	P3	P4
GPL	✓	✓	✓	✓	✓
Driver	✓	✓	✓	✓	✓
Benchmark	✓	✓	✓	✓	✓
GraphType	✓	✓	✓	✓	✓
Directed	✓		✓		
Undirected		✓		✓	✓
Weight		✓	✓	✓	✓
Search	✓	✓	✓	✓	✓
DFS	✓		✓		✓
BFS		✓		✓	
Algorithms	✓	✓	✓	✓	✓
Num	✓	✓	✓		
CC		✓		✓	✓
SCC			✓		
Kruskal		✓			
Prim				✓	✓
Cycle			✓		✓
Shortest			✓		

Table 1. Sample Feature Sets of GPL

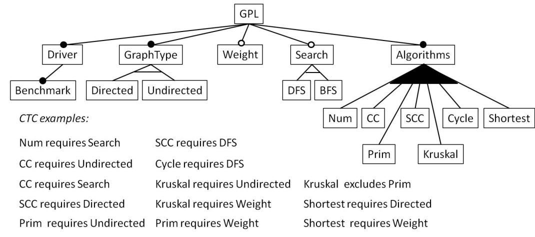

Fig. 1. GPL Feature Model

ii) *Optional features* may or may not be selected if their respective parent feature is selected. An example is feature `Weight`, iii) *Exclusive-or relations* indicate that exactly one of the features in the exclusive-or group must be selected whenever the parent feature is selected. They are depicted as empty arcs crossing over a set of lines connecting a parent feature with its child features. For instance, a graph can be either directed or undirected by selecting either feature `Directed` or `Undirected` respectively, iv) *Inclusive-or relations* indicate that at least one of the features in the inclusive-or group must be selected if the parent is selected. They are depicted as filled arcs crossing over a set of lines connecting a parent feature with its child features. As an example, when feature `Algorithms` is selected then at least one of the features `Num`, `CC`, `SCC`, `Cycle`, `Shortest`, `Prim`, and `Kruskal` must be selected.

Besides the parent-child relations, features can also relate across different branches of the feature model with the so called *Cross-Tree Constraints (CTCs)*. Figure 1 shows the CTCs of our feature model in textual form. These constraints as well as those implied by the hierarchical relations between features are usually expressed and checked using propositional logic in Conjunctive Normal Form (CNF) [14]. For instance, the CTC `Num requires Search` means that whenever feature `Num` is selected, feature `Search` must also be selected. In CNF this CTC is written as $\neg Num \vee Search$.

The following definitions are based on our previous work [12]:

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

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

For example, the feature set `fs=[{GPL, Driver, Benchmark, GraphType, Directed, Search, DFS, Algorithms, Num}, {Undirected, Weight, BFS, CC, SCC, Kruskal, Prim, Cycle, Shortest}]` is valid. In fact, it corresponds to feature set `P0` in Table 1. As another example, a feature set with features `DFS` and `BFS` is not valid because it violates the constraint of the exclusive-or relation which establishes that these two features cannot appear selected together in the same feature set. For GPL case study there are 73 different valid feature sets.

Please recall that the focus of this paper is on synthesizing feature models from feature sets. In other words, for our running example, starting from a table such as Table 1 that includes all the valid feature sets, our goal is to derive a feature model such as the one in Figure 1.

3 Feature Model Synthesis

This section describes the genetic programming pipeline we followed, the feature model representation we used, and the evolutionary operators that were developed.

3.1 Genetic Programming Pipeline

The genetic programming pipeline that we employed is shown in Figure 2. It consists of a set of operators. The gray operators are problem specific while the white ones are generic. It starts with a *Builder* that produces an initial population of randomized individuals. The *Selection* selects individuals from the current population and either passes them to the *Crossover* operator or to the *Reproduction* operator (depending on the crossover probability). The crossover produces offspring individuals that ideally maintain valuable traits of their parent individuals according to a fitness criterion. The reproduction operator just clones individuals. As a next step the individuals either pass through the *Mutation* operator which performs random mutations on the individuals or again just through the reproduction (based on the mutation probability). The part of the pipeline that produces a new population (i.e. the next generation of individuals) from an old one is called *Breeding*, shown as a box in our figure. Finally the fitness of the new individuals is evaluated and they are put back into the population to constitute the next generation. In most cases the new generation completely replaces the old one possibly with the exception of a select number of elite individuals (the ones with the best fitness) which survive and live on in the next generation.

3.2 Feature Model Representation

For the feature model representation we followed a *Model Driven Engineering (MDE)* approach whereby a *metamodel* defines the structure and semantics of the models that can be derived from it [15]. We choose a simplified version of

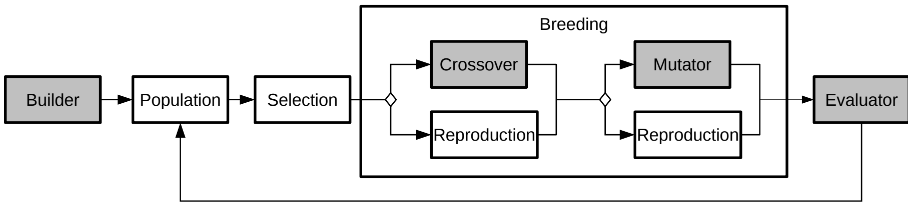

Fig. 2. Genetic Programming Pipeline Overview

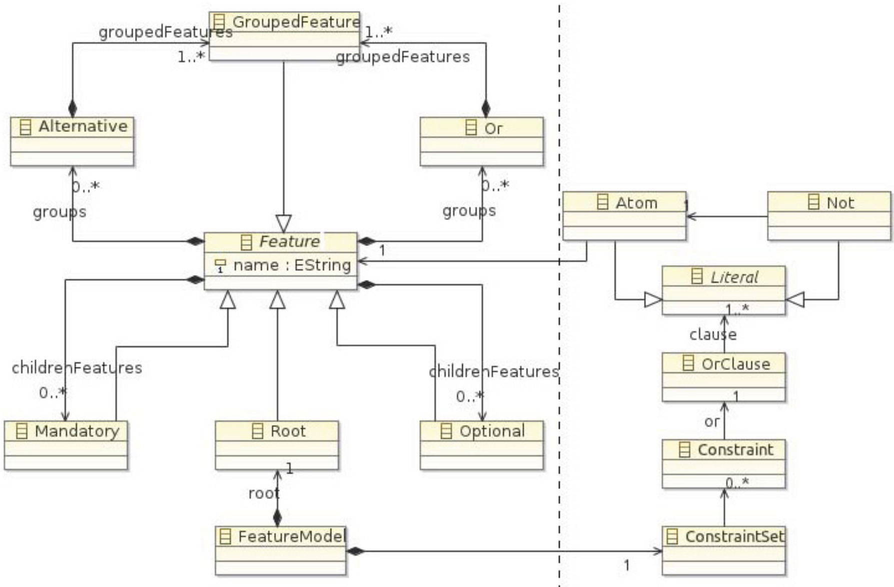

Fig. 3. Feature Model Metamodel

the SPLX metamodel[1], a common standard representation for feature models, which is shown in Figure 3.

This metamodel describes the structure of a feature model individual, represented by the *FeatureModel* meta class. The left part describes the *feature tree*. It has exactly one *Root* feature that, just like any other feature node (i.e. any node that inherits from *Feature*), can have an arbitrary number of *Mandatory* and *Optional* child features as well as an arbitrary number of *Alternative* (i.e. exclusive-or) and *Or* (i.e. inclusive-or) group relations which must have at least one *GroupedFeature* as a child. The right part of the metamodel describes the CTCs of a feature model individual. It has exactly one *ConstraintSet* which describes a propositional formula in CNF. It contains an arbitrary number of *Constraints* which correspond to clauses in a CNF expression. A *Constraint*

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

therefore contains exactly one *OrClause* which must have at least one *Literal*. A *Literal* can either be an *Atom* which refers to a feature directly or a *Not* which then refers to an *Atom*.

The tree structure for the genetic programming individuals reflects for the most part this metamodel and is mostly straightforward to derive. The only exceptions are the following:

- The abstract class *Feature* is represented by its own type of node even though it is an abstract class. This *Feature* node is placed as a child of the inheriting node (e.g. *Mandatory*, *Optional*, etc.). This decision was made to emphasize the importance of features in the domain of feature models so that changes to features in the tree are not just reflected in the change of a node's attributes (i.e. the name attribute) but also in the structure of the tree (i.e. the change of a feature node).
- The class *GroupedFeature* is not represented as a separate node because it does not hold any information and always appears at the same place in the tree: between *Or* or *Alternative* nodes and their child *Feature* nodes.

The tree structure for the GPL feature model as shown in Figure 1 is depicted in Figure 4. The *FeatureModel* node represents the whole *Individual*. It consists of two children: the *Root* node as the root of its *feature tree* and the *ConstraintSet* node containing its CTCs. For example the first *Constraint* node represents the CTC *Num requires Search*.

3.3 Evaluator Definition

The Evaluator uses a fitness function to describe the fitness of an individual. The fitness function we employed in the case of feature models is based on information retrieval metrics (see [16]). We start by defining two auxiliary functions. In the following definitions, let sfs be a set of feature sets (e.g. as denoted in Table 1) which represents our input, and let fm be a candidate feature model individual to be evaluated:

- $\#containedFeatureSets : \mathcal{SFS} \times \mathcal{FM} \rightarrow \mathbb{N}$, returns the number of feature sets received as first argument sfs that are valid according to a feature model fm.
- $\#featureSets : \mathcal{FM} \rightarrow \mathbb{N}$, returns the number of feature sets denoted by a feature model fm.

An ideal candidate feature model describes exactly the feature sets contained in sfs and no more. To express that we use the *precision* and *recall* metrics.

Definition 3. ***Precision***. *The fraction of the retrieved feature sets that are relevant to the search.*

$$precision(sfs, fm) = \frac{\#containedFeatureSets(sfs, fm)}{\#featureSets(fm)}$$

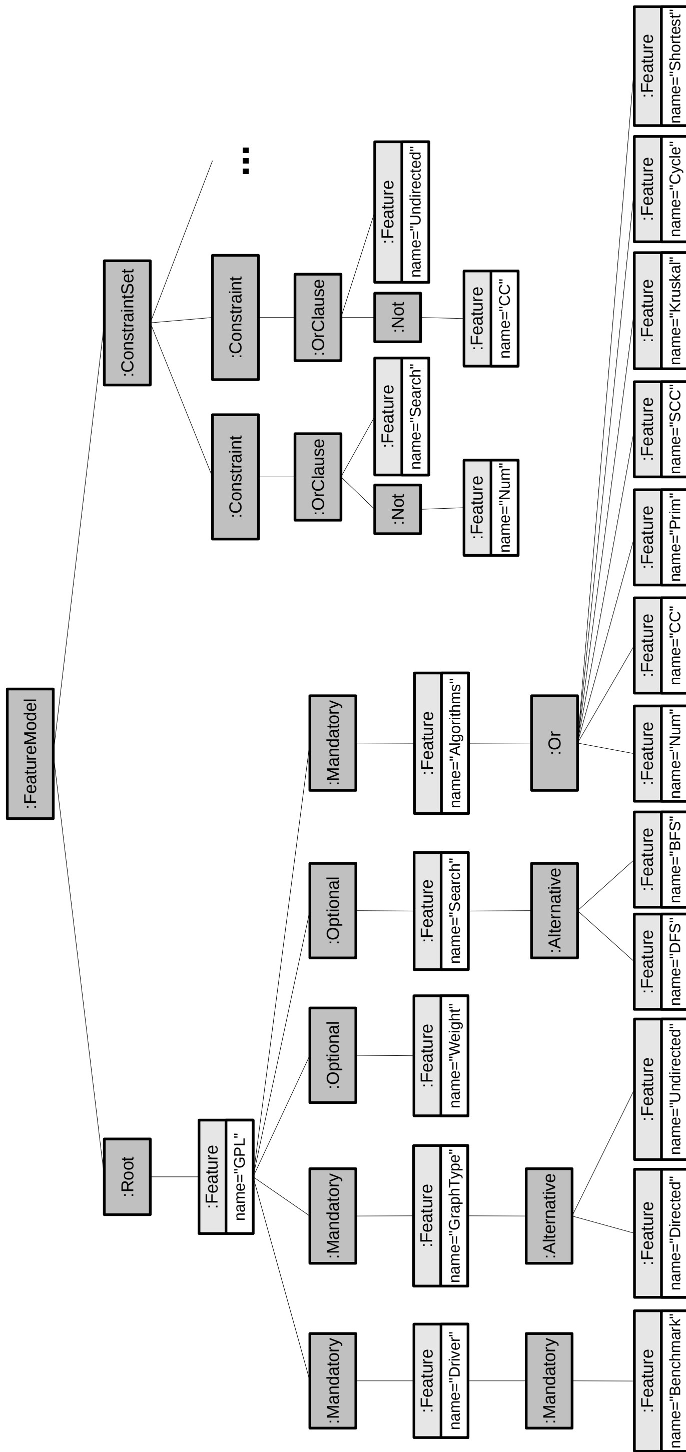

Fig. 4. GPL Feature Model Tree Structure

Definition 4. ***Recall.*** *The fraction of the feature sets that are relevant to the search that are successfully retrieved.*

$$recall(sfs, fm) = \frac{\#containedFeatureSets(sfs, fm)}{|sfs|}$$

Our Evaluator uses the F_β measure as fitness function which is defined as follows [16]:

Definition 5. ***F_β measure.*** *It is a weighted measure of precision and recall. The value of β indicates how many times the recall values weigh more in comparison with the precision values.*

$$F_\beta = \frac{(1+\beta^2) \times precision \times recall}{\beta^2 \times precision + recall}$$

To compute these metrics the feature model representation is executed in the sense that every node is implemented as a function that manipulates a set of feature sets in order to compute the final feature sets that are represented by the whole feature model. For example a *Mandatory* node adds to every feature set in the set its child feature, or a *Constraint* node removes certain feature sets from the set.

3.4 Operators Definitions

Not all semantic constraints can be implicitly conveyed by a metamodel. In the case of our metamodel for feature models there were additional constraints that also apply:

- A feature is identified by its name.
- There is a fixed set of feature names in each feature model.
- Every feature appears exactly once in the feature tree part of a feature model individual.
- CTCs must not contradict each other, i.e. the corresponding CNF of the entire constraint set must be satisfiable.
- CTCs can only be either *requires* or *excludes*, i.e. exactly two literals per clause with at least one being negated.
- There is a maximum number of CTCs (given as a percentage of the number of features) which must not be exceeded.

Based on the tree structures derived from the metamodel and on these domain constraints the necessary operators for genetic programming, namely *Builder*, *Crossover*, and *Mutator* were developed.

Builder. The *Builder* creates random feature trees and random CTCs that conform to the metamodel and also adhere to the additional domain constraints. We implemented it using the tools FaMa [17] and BeTTy [18], which are frameworks written in Java for managing and reasoning about feature models.

Mutator. The *Mutator* makes small random changes to a feature model individual. One of the following mutations is performed randomly on the feature tree with equal probability:

- 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.

The mutations performed on the CTCs, applied with equal probability, are:

- Adds a new, randomly created CTC (i.e. clause) that does not contradict the other CTCs and does not already exist.
- Randomly removes a CTC (i.e. a clause).

Crossover. The *Crossover* takes two individuals from the current population, the parents, and creates two new individuals from them, the offspring. The offspring should maintain desirable traits from both their parents. Just like the other operators the crossover also has to make sure that every offspring still conforms to the metamodel and does not violate any of the additional domain constraints. The following describes how our crossover for feature model individuals works.

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 offspring is obtained the exact same way only that the parents are reversed (i.e. the process starts with the second parent $Parent_2$) and the same sequence of random numbers is used.

The crossover for CTCs is performed by building the union of CTCs of both parents and then assigning a random subset to the first offspring and the remaining to the second offspring.

4 Evaluation

This section first presents the process followed for our evaluation and then analyzes its results.

4.1 Process

We implemented the presented approach using ECJ[2], a generic framework for evolutionary computation written in Java. For the evaluation of the approach we used 17 feature models of actual SPLs that are publicly available[3]. They are shown in Table 2.

Table 2. Feature Models Summary

Feature Model Name	NF	NP	Domain
Apache	10	256	web server
argo-uml-spl	11	192	UML tool
BDBFootprint	9	256	database
BDBMemory	19	3,840	database
BDBPerformance	27	1,440	database
Curl	14	1024	data trasfer
DesktopSearcher	22	462	file search
fame_dbms_fm	20	320	database
gpl	18	73	graph algorithms
LinkedList	27	1,344	data structures
LLVM	12	1,024	compiler library
PKJab	12	72	messenger
Prevayler	6	32	object persistence
SensorNetwork	27	16,704	networking
Wget	17	8,192	file retrieval
x264	17	2,048	video encoding
ZipMe	8	64	data compression

NF: Number of Features, **NP**: Number of Products,
***BDB**: prefix stands for Berkeley DataBase.

We computed for each of these feature models the respective sets of valid feature sets which we used as input to our GP pipeline. The parameter values we employed are shown in Table 3. Note that as a fitness function in our Evaluator we use the F_1 measure, putting equal weight on recall and precision. As a base line to compare our results to we used a *Random Search* (RS) that just randomly creates feature models in hopes of finding a good solution. Details can be found in [19]. The number of random tries is set to the product of the maximum number of generations and the population size of our genetic programming problem so that the number of evaluated candidate feature model individuals is the same for both: $maxGenerations \times populationSize = 100 \times 100 = 10000$ performed evaluations. For the generation of random feature models again the tools FaMa [17] and BeTTy [18] were used. Additionally we used the *Genetic Algorithm* (GA)

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

[3] `http://www.fosd.de/fh`,`http://spl2go.cs.ovgu.de/`,`http://fosd.de/SPLConqueror`

Table 3. Genetic Programming Parameters

Parameter	**Value**
Fitness Function	F_1 measure
Crossover Probability	0.7
Feature Tree Mutation Probability	0.5
CTCs Mutation Probability	0.5
Population Size	100
Maximum Number of Generations	100
Number of Elites	1
Selection Method	Tournament Selection
Tournament Size	6
Maximum CTC Percentage* ...	
... for Builder	0.1
... for Mutator	0.5

*(relative to number of features)

approach to feature model reverse engineering from our previous work [12] and extended it to use F_1 as its fitness function to allow for a comparison of the results. Other than the fitness function nothing was changed on that approach.

For every feature model we did 30 independent runs. Table 4 shows for every feature model the average and the best F_1 value as well as the variance for each our Genetic Programming (GP) approach, the Random Search (RS) and our previous Genetic Algorithm (GA) approach. All the runs were performed on a machine with an Intel® Core™ i5 processor with 3.1 GHz and 8 GB of main memory. The total execution time of our genetic programming approach for all the runs (17 feature models times 30 runs = 510 runs) was at around 13 hours with an average time per run of around 1.5 minutes. For our GPL running example a run took on average only roughly 6 seconds.

4.2 Statistical Analysis

We performed the statistical analysis using R[4], an environment for statistical computing.

The *Wilcoxon Signed-Rank Test* [20] determines whether the difference of two data samples is statistically significant (alternative hypothesis) or due to chance (null hypothesis). We performed the test on the average F_1 values to compare our genetic programming approach against the random search baseline which yielded a p-value of 0.00001526 which leads to rejecting the null hypothesis and accepting the alternative hypothesis that there is a significant difference between our genetic programming approach and the random search. Applying this same test to compare our genetic programming approach against the genetic algorithm

[4] `http://www.r-project.org/`

Table 4. F_1 Values per Feature Model over 30 runs for Genetic Programming (GP), Random Search (RS) and Genetic Algorithm (GA)

	F_1 for GP			F_1 for RS			F_1 for GA		
FM Name	Mean	Best	Variance	Mean	Best	Variance	Mean	Best	Variance
Apache	1.00	1.00	0.0000	0.73	0.95	0.0112	0.72	1.00	0.0160
argo-uml-spl	1.00	1.00	0.0000	0.62	0.98	0.0096	0.67	1.00	0.0188
BDBFootprint	1.00	1.00	0.0000	0.78	0.98	0.0103	0.70	1.00	0.0103
BDBMemory	0.29	0.40	0.0048	0.05	0.13	0.0004	0.16	0.32	0.0051
BDBPerformance	0.22	0.33	0.0029	0.02	0.04	0.0000	0.16	0.23	0.0020
Curl	0.77	0.89	0.0148	0.25	0.35	0.0015	0.46	0.87	0.0170
DesktopSearcher	0.29	0.34	0.0010	0.05	0.08	0.0002	0.21	0.42	0.0061
fame_dbms_fm	0.21	0.38	0.0078	0.04	0.08	0.0001	0.13	0.22	0.0019
gpl	0.47	0.57	0.0086	0.09	0.19	0.0009	0.22	0.41	0.0044
LinkedList	0.28	0.34	0.0028	0.02	0.04	0.0001	0.20	0.32	0.0018
LLVM	1.00	1.00	0.0000	0.53	0.70	0.0081	0.70	1.00	0.0080
PKJab	0.97	1.00	0.0048	0.50	0.66	0.0033	0.66	0.80	0.0115
Prevayler	1.00	1.00	0.0000	0.96	1.00	0.0011	0.69	1.00	0.0072
SensorNetwork	0.26	0.33	0.0016	0.02	0.05	0.0001	0.14	0.21	0.0009
Wget	0.72	0.89	0.0164	0.16	0.23	0.0008	0.40	0.61	0.0041
x264	0.47	0.68	0.0093	0.11	0.20	0.0008	0.23	0.45	0.0054
ZipMe	1.00	1.00	0.0000	0.88	1.00	0.0057	0.72	1.00	0.0160

approach resulted in a p-value of 0.0003204 also indicating a significant difference between the two approaches.

The $\hat{A}_{12}$ *Effect Size Measure* [20, 21] represents the probability that using one algorithm (our genetic programming approach) yields better results than using another algorithm (random search and genetic algorithm approach). An $\hat{A}_{12}$ measure of 0.5 would mean both algorithms perform equally well. The value we obtained based on the average F_1 values of our genetic programming approach and the random search is $\hat{A}_{12} = 0.7750865$ which means that the genetic programming approach clearly outperforms the random search, as the probability of achieving better results with it is at 77.5%. Computing this measure for our genetic programming approach and the genetic algorithm approach yielded $\hat{A}_{12} = 0.7474048$ which again means that our genetic programming approach outperforms the genetic algorithm approach.

4.3 Threats to Validity

Following the guidelines in [22] we identified three threats to validity that are relevant to our work. The first is the parameter settings that were used during the evaluation, all of which are given in Table 3. Mostly standard values for genetic programming were used except for the mutation probability where we followed the example of [23] and used an above standard value. The second threat is the correctness of the implementation. To address this threat we provide an overview of our genetic programming pipeline (Figure 2), we used ECJ as

a proven framework for evolutionary computation to implement our approach with, and we make the full implementation and data available for replication[5]. The third threat is the selection of the corpus of feature models on which the evaluation was performed. These feature models stem from actual SPLs and we thus believe that they are good representatives of the feature models domain.

5 Related Work

In this section, we briefly summarize the pieces of work that are closest to ours.

Our previous work studied reverse engineering feature models using a genetic algorithm [12]. We encoded feature models based on a depth-first traversal order. The key limitations of that approach were the relative ordering that features should have between them and the heavy performance penalty of detecting and fixing incorrect individuals after mutation and crossover.

The work by Haslinger et al. presents an ad hoc algorithm also to reverse engineer feature models [24]. The main distinction with our work is that it only reverse engineers one feature model, as opposed to potentially many equivalent feature models in our work. The work by She et al. also provides ad hoc algorithms for reverse engineering feature models, however, in contrast with our work, they start from a set of constraints expressed either in CNF or DNF. Work by Acher et al. relies on user-defined domain knowledge to help structure the hierarchy between features [25]. This knowledge helps to eliminate semantically correct (i.e. correct feature combinations) feature models that are hierarchically incorrect (i.e. parent-child relation swapped). These two pieces of work could be respectively leveraged to seed the initial population and guide the search in our approach. These are two issues we plan to explore as part of our future work.

Recent work by Acher et al. presents several feature model composition operators [26]. They provide their semantics and analyze their properties. We believe their work could help our approach to both define other crossover operators as well as put them in a more formal footing. Doing this is part of our future work.

6 Conclusions and Future Work

In this paper we applied genetic programming to the problem of reverse engineering feature models in the context of SPLs. We showed the workflow that we followed along with the resulting representation of feature models and the evolutionary operators used in our genetic programming pipeline. We reported our encouraging experience synthesizing 17 feature models and compared our results to a random search baseline as well as to previous work in the area of feature model reverse engineering and showed that our approach outperforms both.

As future work we plan to investigate the impact of seeding knowledge derived from ad-hoc reverse engineering algorithms into our GP pipeline, as well as other operators for crossover based on feature model composition. Currently the fitness

[5] http://www.sea.uni-linz.ac.at/sbse4vm/data/ssbse.zip

of individuals is based on whether feature sets are contained or not. This is rather coarse grain. We want to employ a more fine-grain fitness metric that works on the level of single features instead of complete feature sets. Also we plan to evaluate our approach using more feature models.

Acknowledgments. This research is partially funded by the Austrian Science Fund (FWF) project P25289-N15 and Lise Meitner Fellowship M1421-N15.

References

1. Harman, M., Mansouri, S.A., Zhang, Y.: Search-based software engineering: Trends, techniques and applications. ACM Comput. Surv. 45(1), 11 (2012)
2. Harman, M., McMinn, P., de Souza, J.T., Yoo, S.: Search based software engineering: Techniques, taxonomy, tutorial. In: Meyer, B., Nordio, M. (eds.) Empirical Software Engineering and Verification. LNCS, vol. 7007, pp. 1–59. Springer, Heidelberg (2012)
3. de Freitas, F.G., de Souza, J.T.: Ten years of search based software engineering: A bibliometric analysis. In: Cohen, M.B., Ó Cinnéide, M. (eds.) SSBSE 2011. LNCS, vol. 6956, pp. 18–32. Springer, Heidelberg (2011)
4. Poli, R., Langdon, W.B., McPhee, N.F.: A Field Guide to Genetic Programming. Lulu (2008)
5. Batory, D.S., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Software Eng. 30(6), 355–371 (2004)
6. Lopez-Herrejon, R.E., Egyed, A.: Sbse4vm: Search based software engineering for variability management. In: Cleve, A., Ricca, F., Cerioli, M. (eds.) CSMR, pp. 441–444. IEEE Computer Society (2013)
7. 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)
8. Laguna, M.A., Crespo, Y.: A systematic mapping study on software product line evolution: From legacy system reengineering to product line refactoring. Sci. Comput. Program. 78(8), 1010–1034 (2013)
9. Krueger, C.W.: Easing the transition to software mass customization. In: van der Linden, F.J. (ed.) PFE 2002. LNCS, vol. 2290, pp. 282–293. Springer, Heidelberg (2002)
10. She, S., Ryssel, U., Andersen, N., Wsowski, A., Czarnecki, K.: Efficient synthesis of feature models. Information and Software Technology (in press, 2014)
11. Harman, M., Langdon, W.B., Weimer, W.: Genetic programming for reverse engineering. In: Lämmel, R., Oliveto, R., Robbes, R. (eds.) WCRE, pp. 1–10. IEEE (2013)
12. Lopez-Herrejon, R.E., Galindo, J.A., Benavides, D., Segura, S., Egyed, A.: Reverse engineering feature models with evolutionary algorithms: An exploratory study. In: Fraser, G., Teixeira de Souza, J. (eds.) SSBSE 2012. LNCS, vol. 7515, pp. 168–182. Springer, Heidelberg (2012)
13. Lopez-Herrejon, R.E., Batory, D.: A standard problem for evaluating product-line methodologies. In: Bosch, J. (ed.) GCSE 2001. LNCS, vol. 2186, pp. 10–24. Springer, Heidelberg (2001)

14. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. 35(6), 615–636 (2010)
15. Stahl, T., Völter, M., Bettin, J., Haase, A., Helsen, S.: Model-driven software development - technology, engineering, management. Pitman (2006)
16. Manning, C.D., Raghavan, P., Schütze, H.: Introduction to information retrieval. Cambridge University Press (2008)
17. Benavides, D., Segura, S., Trinidad, P., Cortés, A.R.: Fama: Tooling a framework for the automated analysis of feature models. In: Pohl, K., Heymans, P., Kang, K.C., Metzger, A., eds.: VaMoS. Volume 2007-01 of Lero Technical Report, 129–134 (2007)
18. Segura, S., Galindo, J., Benavides, D., Parejo, J.A., Cortés, A.R.: BeTTy: Benchmarking and testing on the automated analysis of feature models. In: Eisenecker, U.W., Apel, S., Gnesi, S. (eds.) VaMoS, pp. 63–71. ACM (2012)
19. Luke, S.: Essentials of Metaheuristics. Lulu (2009) Available for free at `http://cs.gmu.edu/~sean/book/metaheuristics/`
20. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification & Reliability (2012)
21. Vargha, A., Delaney, H.D.: A critique and improvement of the "cl" common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)
22. de Oliveira Barros, M., Neto, A.C.D.: Threats to Validity in Search-based Software Engineering Empirical Studies. Technical Report 0006/2011, Universidade Federal Do Estado Do Rio de Janeiro. Departamento de Informatica Aplicada (2011)
23. Faunes, M., Cadavid, J.J., Baudry, B., Sahraoui, H.A., Combemale, B.: Automatically searching for metamodel well-formedness rules in examples and counterexamples. In: [27], pp. 187–202
24. Haslinger, E.N., Lopez-Herrejon, R.E., Egyed, A.: On extracting feature models from sets of valid feature combinations. In: Cortellessa, V., Varró, D. (eds.) FASE 2013 (ETAPS 2013). LNCS, vol. 7793, pp. 53–67. Springer, Heidelberg (2013)
25. Acher, M., Baudry, B., Heymans, P., Cleve, A., Hainaut, J.L.: Support for reverse engineering and maintaining feature models. In: Gnesi, S., Collet, P., Schmid, K. (eds.) VaMoS, pp. 20:1–20:8. ACM (2013)
26. Acher, M., Combemale, B., Collet, P., Barais, O., Lahire, P., France, R.B.: Composing your compositions of variability models. In: [27], pp. 352–369
27. Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P. (eds.): MODELS 2013. LNCS, vol. 8107. Springer, Heidelberg (2013)