Optimizing Selection of Competing Features via Feedback-Directed Evolutionary Algorithms

Tian Huat Tan[†] Yinxing Xue[*] Manman Chen[*] Jun Sun[†] Yang Liu[‡] Jin Song Dong[*]
[†]Singapore University of Technology and Design, Singapore
[*]National University of Singapore, Singapore
[‡]Nanyang Technological University, Singapore

ABSTRACT

Software that support various groups of customers usually require complicated configurations to attain different functionalities. To model the configuration options, feature model is proposed to capture the commonalities and competing variabilities of the product variants in software family or Software Product Line (SPL). A key challenge for deriving a new product is to find a set of features that do not have inconsistencies or conflicts, yet optimize multiple objectives (e.g., minimizing cost and maximizing number of features), which are often competing with each other. Existing works have attempted to make use of evolutionary algorithms (EAs) to address this problem. In this work, we incorporated a novel feedback-directed mechanism into existing EAs. Our empirical results have shown that our method has improved noticeably over all unguided version of EAs on the optimal feature selection. In particular, for case studies in SPLOT and LVAT repositories, the feedback-directed Indicator-Based EA (IBEA) has increased the number of correct solutions found by 72.33% and 75%, compared to unguided IBEA. In addition, by leveraging a pre-computed solution, we have found 34 sound solutions for Linux X86, which contains 6888 features, in less than 40 seconds.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications

General Terms

Design, Performance, Algorithm

Keywords

Software product line, evolutionary algorithms, SAT solvers

1. INTRODUCTION

To reduce development costs, shorten development cycles, and improve flexibility and reusability, industries usually need to develop and maintain a set of similar products in a systematic and reuse-based way [18]. In software family or Software Product Line (SPL) [19], feature model is proposed to model commonalities and competing variabilities among similar yet different products. Based on the feature model, different features are carefully selected to meet the requirements of customers and to avoid possible conflicts or compatibility problems. In the era of a thriving market of mobile and serviced-based applications, vendors are required to continually reconfigure their applications promptly, to retain and extend their customer base. Therefore, it is desirable to automatically derive features that could meet the requirements of customers, and avoid all possible conflicts of features.

Feature model provides a representation of software product lines (SPLs), that could be used to facilitate the reasoning and configuration of SPLs [19]. Common SPLs consist of hundreds or even thousands of features. For instance, as reported in [30], the Linux X86 kernel contains 6888 features, and 343944 constraints. In addition, the features are usually associated with quality attributes such as cost and reliability. This complexity provides challenges for the reasoning and configuration of feature models. It is hard for the vendor to select a set of features that complies with the feature model, and meanwhile optimizes the quality attributes according to user preferences. This is called the *optimal feature selection* problem [14].

Existing works [14, 29, 27, 28] have adopted evolutionary algorithms (EAs) for feature selection with resource constraints and product generation based on the value of user preferences, respectively. Guo *et al.* [14] proposed a genetic algorithm (GA) approach for tackling the optimal feature selection problem. In their work, a repair operator is used to fix each candidate solution, so that it is fully compatible with the feature model after each round of crossover and mutation operations. This approach might be non-terminating, and furthermore, it does not take advantage of the automatic correction that brought by the GA. In addition, GA combines all objectives into a single fitness function with respective weights. This only gives users a solution that is specific to the weights used in the objective formula.

To address this problem, Sayyad *et al.* [29, 27] proposed an approach that uses EAs to support multi-objective optimization, and a range of optimal solutions (i.e., a Pareto front) is returned to the user as a result. They investigated seven EAs and discovered that the Indicator-Based Evolutionary Algorithm (IBEA) [36] yields the best results among the seven tested EAs in terms of time, correctness and satisfaction to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
© 2015 ACM. 978-1-4503-3620-8/15/07...$15.00
http://dx.doi.org/10.1145/2771783.2771808

user preferences. In [28], they made use of static method to prune features before execution of IBEA for reducing search space. They also introduced a "seeding method" by pre-computing a correct solution, which was subsequently used by IBEA to generate further correct solutions.

Our work complements existing works by introducing a novel feedback-directed mechanism to existing EAs. In our approach, the feature model is first preprocessed based on SAT solving to remove the prunable features, before the execution of an EA. We have shown that we always prune more features compared to the pruning method in [28]. During each round of executing EA, the violated constraints would be analyzed. The analyzed results are used as feedback to guide evolutionary operators (i.e., crossover and mutation) for producing offsprings for the next round. Our evaluation has shown that our method produces more promising offsprings (that have fewer violated constraints), which has led to faster convergence and resulted in more valid solutions in a significantly shorter amount of time.

We make use of both SPLOT [22] and LVAT [1] repositories to evaluate our work. SPLOT is a repository of feature models used by many researchers as a benchmark, and LVAT contains the real-world feature models which have large feature sizes, including the aforementioned Linux X86 kernel model which contains 6888 features.

Our main contributions are summarized below.

1. We introduce a feedback-directed mechanism into existing EAs. In a feedback-directed EA, solutions are analyzed by their violated constraints. The information is used as feedback for evolutionary operators to produce offsprings that are more likely to satisfy more constraints.
2. We evaluate benefits brought by the feedback-directed EAs using feature models that are available publicly. The feedback-directed EAs have shown a significant improvement on finding more optimized valid solutions, compared with the original unguided EAs used in [29, 28].
3. We make use of the seeding method as proposed in [28] for finding valid solutions in the Linux X86 feature model, which contains 6888 features. Our approach combining with the seeding method has shortened the search time for more than 200 times than the original seeding approach as proposed in [28].

Outline. Section 2 introduces the background of this work. Section 3 presents our feedback-directed EA. Section 4 provides the evaluation of our approach. Section 5 reviews related works. Finally, Section 6 concludes and outlines future work.

2. BACKGROUND

In this section, we provide the background knowledge on software product line, feature model, and multi-objective optimization problem.

2.1 Software Product Line

Software product line engineering (SPLE) is architecture-centric and feature-oriented, as SPLE adopts feature-oriented domain analysis [19] for requirements analysis and builds core assets architecture for reuse [8]. Technically, SPLE is a two-phase approach composed of domain engineering and application engineering. The task of domain engineering is to build the software product line (SPL) architecture consisting of a core-asset base and the variant features, while the application engineering focuses on derivation of new products by different customizations of variant features applied onto the core-asset base. Thus, automation of processing and verification of product derivation is a fundamental problem in SPLE. Exploring an efficient and scalable approach for the optimal feature selection problem is critical to the success of SPLE.

2.2 Feature Model and its Semantics

The concept of feature model in domain engineering is to represent the features within the product family as well as the structural and semantic (require or exclude) relationships between those features [19]. Since the proposal of SPL, feature model has even been characterized as "the greatest contribution of domain engineering to software engineering" [9].

A feature model is a tree-like hierarchy of features. The structural and semantic relationships between a parent (or compound) feature and its child features (or subfeatures) can be specified as:

- *Alternative* – If the parent feature is selected, exactly one among the exclusive subfeatures should be selected,
- *Or* – If the parent feature is selected, at least one of the subfeatures must be selected,
- *Mandatory* – A mandatory feature must be selected if its parent is selected,
- *Optional* – An optional feature is optional to be selected.

Besides the above structure or parental relationships between features, cross-tree constraints (CTCs) are also often adopted to represent the mutual relationship for features across the feature model. There are three types of common CTCs:

- f_a *requires* f_b – The inclusion of feature f_a implies the inclusion of feature f_b in the same product.
- f_a *excludes* f_b – The inclusion of feature f_a implies the exclusion of feature f_b in the same product, and vice versa.
- f_a *iff* f_b – The inclusion of feature f_a implies the inclusion of feature f_b in the same product, and vice versa.

In Figure 1, the feature model of a Java Chat System (*JCS*) is illustrated. The root feature of the feature model is *Chat*, which has a mandatory subfeature (*Output*) and several optional subfeatures (e.g., *Encryption*). Since the feature *Output* is mandatory, exactly one of its subfeatures (*GUI*, *CMD*, and *GUI2*) must be selected. In addition, if the *Encryption* feature is selected, at least one of its subfeatures (*Caesar* and *Reverse*) needs to be selected. There is a CTC for *JCS* which is of the form f_a *iff* f_b – *Encryption_OR* is selected if and only if *Caesar* or *Reverse* is selected.

The feature model listed in Figure 1 can be captured by the constraints that are listed in Table 1. The constraints

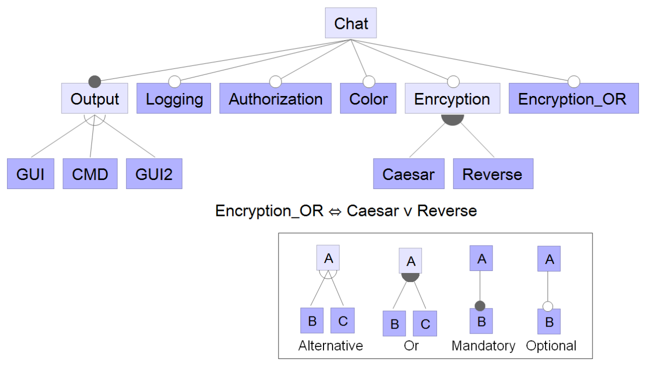

Figure 1: The feature model of *JCS*

Table 1: Constraints of *JCS*

$Chat$	c(1)
$Output \iff Chat$	c(2)
$Logging \implies Chat$	c(3)
$Authorization \implies Chat$	c(4)
$Color \implies Chat$	c(5)
$Encryption \implies Chat$	c(6)
$Encryption\_OR \implies Chat$	c(7)
$(GUI \vee CMD \vee GUI2) \iff Output$	c(8)
$\neg(GUI \wedge CMD)$	c(9)
$\neg(GUI \wedge GUI2)$	c(10)
$\neg(CMD \wedge GUI2)$	c(11)
$(Caesar \vee Reverse) \iff Encryption$	c(12)
$Encryption\_OR \iff (Caesar \vee Reverse)$	c(13)

are specified according to the semantics of feature model. Constraint c(1) specifies that the root feature must be present, to prevent a trivial feature model with no selected feature. Constraint c(2) specifies the mandatory feature *Output* and constraints c(3) – c(7) specify constraints on the other five optional subfeatures. The subfeatures of *Output* are in an *Alternative* relationship. This is specified using constraints c(8) – c(11). Constraint c(8) states that *Output* is selected, if and only if at least one of *CMD*, *GUI* and *GUI2* is selected. Constraints c(9) – c(11) specify that at most one feature from *CMD*, *GUI* and *GUI2* can be chosen. The subfeatures of *Encryption* are in *Or* relationship. The constraint c(12) denotes if *Encryption* is selected, then at least one feature from *Caesar* and *Reverse* needs to be selected, and vice versa. The only CTC of *JCS* is captured in the constraint c(13). Constraints c(1) – c(12) are called *tree constraints*, since they are related to the tree structure of the feature model. Henceforth, given a feature model M, we simply refer tree constraints and CTCs of the M, as the *constraints* of M. We denote the conjunction of constraints of M as $conj(M)$. We use $Fea(M)$ to denote the set of entire features of the feature model M. For the *JCS* example, $Fea(JCS) = \{Chat, \ldots\}$ and $|Fea(JCS)|$=12.

DEFINITION 1 (FEASIBLE FEATURE SET). *Given a feature model M, a* feasible feature set *for M is a non-empty feature set $F \subseteq Fea(M)$, such that F satisfies the constraints of M.*

We write $F \models M$ if $F \subseteq Fea(M)$ is a feasible feature set of the feature model M.

Example. We use *JCS* as an example. $F = \{Chat, Output, GUI\}$ is a feasible feature set of *JCS*, i.e., $F \models JCS$.

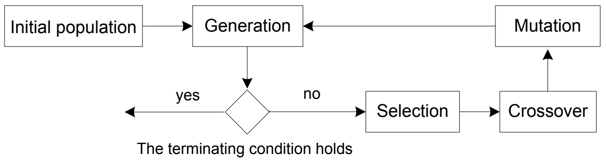

Figure 2: Typical flow of evolutionary algorithms

2.3 Multi-objective Optimization Problem

Many real-world problems have multiple objectives that need to be optimized simultaneously. However, these objectives usually conflict with each other, which prevents optimizing all objectives simultaneously. A remedy is to have a set of optimal trade-offs between the conflicting objectives.

A k-objective optimization problem could be written in the following form:

$$\begin{aligned} Minimize\ Obj(F) = (Obj_1(F), Obj_2(F), ..., Obj_k(F)) \\ subject\ to\ F \models M \end{aligned} \tag{1}$$

where $Obj(F)$ is a k-dimensional objective vector for F and $Obj_i(F)$ is the value of F for ith objective.

Given $F_1, F_2 \models M$, F_1 can be viewed as better than F_2 for the minimization problem in Equation (1), if Equation (2) holds.

$$\forall i : Obj_i(F_1) \leq Obj_i(F_2) \wedge \exists j : Obj_j(F_1) < Obj_j(F_2) \tag{2}$$

where $i, j \in \{1, \ldots, k\}$.

In such a case, we say that F_1 *dominates* F_2. F_1 is called a *Pareto-optimal solution* if F_1 is not dominated by any other $F \models M$. We denote all Pareto-optimal solutions as the *Pareto front*.

Many evolutionary algorithms (e.g., IBEA [36], NSGA-II [11], ssNSGA-II [13], MOCell [23]) are proposed to find a set of non-dominated solutions that approximate the Pareto front for solving the multi-objective optimization problem.

Problem Statement. Our work addresses the *optimal feature selection*, which aims at searching for feasible feature sets that approximate the Pareto front to solve the multi-objective optimization problem.

3. FEEDBACK-DIRECTED EVOLUTIONARY ALGORITHM

In this section, we elaborate our approach in addressing the optimal feature selection problem. First, we introduce a preprocessing method to filter out prunable features before the execution of an EA, in order to reduce the search space. Second, we illustrate feedback-directed evolutionary operators that are used in this work to guide an EA for the optimal feature selection.

3.1 Preliminaries of Evolutionary Algorithms

Evolutionary algorithms (EAs), inspired by the "survival of the fittest" principle of the Darwinian theory of natural evolution, are stochastic search methods based on principles of the biological evolution. By applying the EA, a problem is encoded into a simple chromosome-like data structure, and then evolutionary operators (e.g., selection, crossover, and mutation) are applied on these data structures to preserve "the fittest" information, which is analogous to "survival of the fittest" in the natural world. EAs often perform well in approximating solutions, and therefore EAs are typically

Algorithm 1: PrunableFeatures

```
  input  : Feature model $M$
  output : Common features $F_c \subseteq Fea(M)$
  output : Dead features $F_d \subseteq Fea(M)$
1 $F_c \leftarrow \emptyset$;
2 $F_d \leftarrow \emptyset$;
3 foreach $f \in Fea(M)$ do
4     if $\neg SAT(conj(M) \wedge \neg f)$ then
5         $F_c = F_c \cup f$;
6     else if $\neg SAT(conj(M) \wedge f)$ then
7         $F_d = F_d \cup f$;
8 return $(F_c, F_d)$;
```

suitable for the optimization problems especially if the search space of the problem is large and complex.

A typical workflow of EAs is described in Figure 2. An EA begins with an initial generation of chromosomes, which we denote as *initial population*. Typically, the initial population is generated randomly. Evolutionary operators are then applied on a generation to evolve into a new generation of chromosomes. Different EAs have different dominating criteria, which will be introduced in Section 4.1. The chromosomes that are ranked higher according to the dominating criteria of the EA have a higher chance to proceed to the next generation. The evolutionary process continues until the termination condition is met. An example of the termination condition might be that the number of generations exceeds a predefined upper bound $n \in \mathbb{Z}_{>0}$.

3.2 Preprocessing of Feature Model

In the following, we introduce the features that could be pruned from $Fea(M)$ before the execution of an EA. By doing this, the search space of the EA would be reduced, which could make the optimal feature selection more efficient.

Our approach of preprocessing is by exploiting the *commonalities* [6] of the products. Observed that some features must be present in all products derived from M. For example in *JCS*, the feature set $\{Chat, Output\}$ is shared by all derived products, and we call these features as *common features*. Similarly, we call the set of features that must not be used in all derived products as *dead features*. Dead features do not present in *JCS* but they are common in feature models of real systems (e.g., Linux X86 kernel and eCos operating system). Henceforth, we denote common features and dead features as F_c and F_d respectively, where $F_c, F_d \subseteq Fea(M)$, and $F_c \cap F_d = \emptyset$. The preprocessed features that are passed to the execution of EAs is $Fea(M) \setminus (F_c \cup F_d)$, and we denote $F_c \cup F_d$ as *prunable features*.

The function *PrunableFeatures* (Algorithm 1) is used to find common and dead features. Recall that $conj(M)$ represents the conjunction of all tree constraints and CTCs of feature model M, and SAT is a function that is used to check the satisfiability of the constraints. Note that SAT function is readily provided by many off-the-shelf SAT solvers (e.g., SAT4J [2]). We assume that $conj(M)$ is satisfiable, i.e., there exists at least a valid product from the feature model M. If $conj(M) \wedge \neg f$ is unsatisfiable (line 4), it implies that feature f must exist in all derived products of M. Therefore, feature f is added to common features F_c (line 5). This is similar to the detection of dead features in lines 6 and 7.

3.3 Genetic Encoding of the Feature Set

The selected features of a feature model is encoded using an array-based chromosome as shown in Figure 3. Given a chromosome of length n, array indices are numbered from 0 to $n-1$. Each feature is assigned with an array index starting from 0. Each value on the chromosome ranges over $\{0, 1\}$, where 0 (resp. 1) represents the absence (resp. presence) of the feature. Given a feature model M, we define a function $f_M : Fea(M) \to \{\mathbb{Z}, \perp\}$ that maps each feature f of the feature model M to an array index. $f_M(f_1) = \perp$ denotes that there is no array index that is assigned for the feature f_1. Similarly, we define ${f_M}^{-1} : \mathbb{Z} \to Fea(M)$ as a function that maps a given array index to the feature it represents.
Example. We show how a feature set on the *JCS* is encoded. Note that features *Chat* and *Output* have been pruned by the preprocessing algorithm in Algorithm 1; therefore, they are not contained in the chromosome (i.e., $f_M(Chat) = f_M(Output) = \perp$). The features are indexed level by level, and their indexes have been listed in Figure 3 (e.g., $f_M(Logging) = 0$). The chromosome in Figure 3 represents the feature set $\{Encryption, GUI, Caesar, Reverse\}$.

3.4 Feedback-Directed Evolutionary Operators

The violated constraints of a chromosome C_i provide an important clue on which features on the chromosome C_i need to be modified. If we focus on these features, we may converge faster on the optimal feature selection.

We incorporate this feedback into the crossover and mutation operations, which are the main evolutionary operators common for almost all EAs. The feedback-directed crossover and mutation operators provide an effective guidance for EAs to perform the optimal feature selection.

3.4.1 Feedback-Directed Mutation

The objective of *mutation* operator is to change some values in a selected chromosome leading to additional genetic diversity to help the search process escape from local optimal traps.

We introduce how the *feedback-directed mutation* operator works. Before the mutation, the feedback-directed mutation analyzes the selected chromosome on the violated constraints. We denote the corresponding positions on the chromosomes for the features that are contained in the violated constraints as *error positions*.
Example. We illustrate the feedback-directed mutation operator, using the *JCS* example shown in Figure 3. Given the values of the chromosome as shown in Figure 3, we can easily check that it violates the constraint c_{13}. The constraint c_{13} contains three features, which are *Encryption_OR*, *Caesar*, and *Reverse*. The corresponding array positions of these three features are shaded on the chromosome in Figure 3. These shaded positions are the *error positions*.

The algorithm *FMutation* for feedback-directed mutation are given in Algorithm 2. At line 1, an offspring chromosome C is initialized with values in the chromosome P, and $n \in \mathbb{Z}$ is initialized with the length of the chromosome P (line 2). At line 3, $Err \in \mathcal{P}(\mathbb{Z})$ is assigned with the set of integers that is returned from $ErrPos(C)$ (which will be introduced later). The set of integers returned by $ErrPos(C)$ represents the error positions on the chromosome C. Each position on the chromosome is iterated (line 4). The function $rand(a, b)$ (resp., $randInt(a, b)$), with $a > b$, chooses a real (resp., integer) number between numbers a and b. At

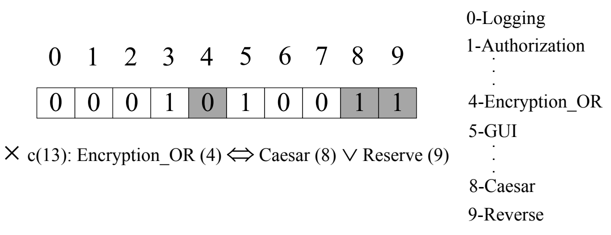

Figure 3: Feedback-directed mutation operator

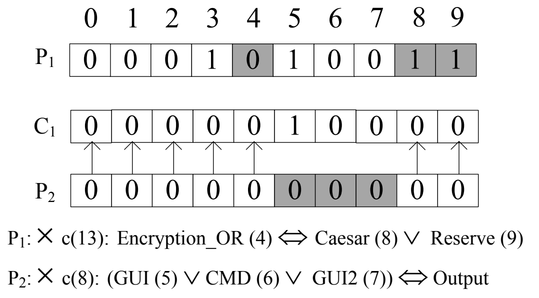
0 1 2 3 4 5 6 7 8 9
P1 0 0 0 1 0 1 0 0 1 1
C1 0 0 0 0 0 1 0 0 0 0
P2 0 0 0 0 0 0 0 0 0 0
P1: × c(13): Encryption_OR (4) ⟺ Caesar (8) ∨ Reserve (9)
P2: × c(8): (GUI (5) ∨ CMD (6) ∨ GUI2 (7)) ⟺ Output

Figure 4: Feedback-directed crossover operator

line 5, if the current position i is an error position, and the random number is less than the error mutation probablity P_{emut}, then the value in the ith-position on the chromosome is mutated by randomly choosing an integer between 0 and 1 (line 7). On the other hand, if the position does not belong to any error position, and the random number is less than P_{mut} (line 6), the value in the ith-position is mutated. Note that the probability P_{emut} will be set with a value that is far larger than P_{mut}, so that the mutation occurs more frequently on error positions. For P_{emut} and P_{mut}, example values could be 1.0 and 0.0000001. Note that we set P_{mut} much lower than classic mutation probability (e.g. 0.001-0.05 [31]). This is because lower P_{mut} with higher P_{emut} would lead to faster convergence, since it allows faster correction of constraint violations by minimizing the changes of non-error positions and focusing on the changes of error positions. This is demonstrated in Section 4.3.

We now introduce the *ErrPos* function described in Algorithm 3. At line 1, *ePos* is initialized with an empty set. The valuation function $\Pi : Fea(M) \to \{true, false\}$ (line 3) maps each feature f of the feature model M to a Boolean value that denotes whether the corresponding feature is selected. The mappings in Π are populated according to the values on the chromosome (line 5). Subsequently, common and dead features are added to the mappings in Π with values *true* and *false* respectively (lines 7–9). The reason is that common (dead resp.) features must (must not resp.) belong to any feasible feature set of feature model M as explained in Section 3.2. At line 11, $\Pi \not\models constraint$ holds iff replacing each feature f contained in the *constraint* with $\Pi(f)$ evaluates to false. In other words, $\Pi \not\models constraint$ means that the selection represented by chromosome C violates the constraint *constraint*. In such a case, the function $getFeatures(c)$ is used to get the features that are contained in the constraint c (line 12). For example, given the constraint $c(13)$ in Table 1 for *JCS* as an input, *getFeatures* will return $\{4, 8, 9\}$. These array indexes that represent the error positions will be included in *ePos*.

Algorithm 2: FMutation

```
input  : Chromosome $P$
input  : Error mutation probability $P_{emut}$
input  : Mutation probability $P_{mut}$
output : Chromosome $C$
1 $C \leftarrow P$;
2 $n \leftarrow |P|$;
3 $Err \leftarrow ErrPos(C)$;
4 for $i = 0$ to $n - 1$ do
5     if $(i \in Err \wedge rand(0,1) < P_{emut}) \vee$
6     $(i \notin Err \wedge rand(0,1) < P_{mut})$ then
7         $C[i] \leftarrow randInt(0,1)$;
8 return $C$;
```

Algorithm 3: ErrPos

```
input  : Chromosome $C$
input  : A set of constraints $constraints$
output : A set of integers $ePos$
1 $ePos \leftarrow \emptyset$;
2 $n \leftarrow |C_1|$;
3 $\Pi \leftarrow \emptyset$;
4 for $i = 0$ to $n - 1$ do
5     $\Pi \leftarrow \Pi \cup \{f_M{}^{-1}(i) \mapsto (C[i] \neq 0)\}$;
6 foreach $feature \in F_c$ do
7     $\Pi \leftarrow \Pi \cup \{F_c \mapsto true\}$;
8 foreach $feature \in F_d$ do
9     $\Pi \leftarrow \Pi \cup \{F_d \mapsto false\}$;
10 foreach $constraint \in constraints$ do
11     if $\Pi \not\models constraint$ then
12         $ePos \leftarrow ePos \cup getFeatures(constraint)$;
13 return $ePos$;
```

3.4.2 Feedback-Directed Crossover

The crossover operation is used to generate offsprings by exchanging values in a pair of chromosomes chosen from the population, and it happens with a probability P_{cross} (the crossover probability). The feedback-directed crossover operator uses values in the non-error positions to crossover. The objective for using values from non-error positions is to pass the “good genes” to offsprings.

Example. We demonstrate the feedback-directed crossover operator, using the *JCS* example shown in Figure 4. Suppose the chromosomes P_1 and P_2 have violated constraints $c(13)$ and $c(8)$ respectively. The offspring chromosome C_1 is first initialized as the same values with the chromosome P_1. We now show that how the feedback-directed crossover is performed. The values from non-error positions of the chromosome P_2 are copied to the chromosome C_1 (shown by the arrows). This results in the chromosome C_1 that is shown in Figure 4. The production of the chromosome C_2 (not shown in the graph) is symmetric to the production of the chromosome C_1.

The algorithm *FCrossover* of feedback-directed crossover operator is given in Algorithm 4. The chromosomes C_1 and C_2 are initialized with the values from chromosomes P_1 and P_2 respectively (lines 1, 2). If the generated random num-

Algorithm 4: FCrossover

```
input : Chromosome $P_1$
input : Chromosome $P_2$
input : Crossover probability $P_{cross}$
output: Chromosomes $C_1$, $C_2$
1  $C_1 \leftarrow P_1$;
2  $C_2 \leftarrow P_2$;
3  $n \leftarrow |P_1|$;
4  if $rand(0,1) < P_{cross}$ then
5      if $|ErrPos(P_1)| > 0 \wedge |ErrPos(P_2)| > 0$ then
6          for $i = 0$ to $n-1$ do
7              if $i \notin ErrPos(P_1)$ then
8                  $C_2[i] \leftarrow P_1[i]$;
9              if $i \notin ErrPos(P_2)$ then
10                 $C_1[i] \leftarrow P_2[i]$;
11     else
12         $crossIndex \leftarrow randInt(0, n-1)$;
13         for $i = crossIndex$ to $n-1$ do
14             $C_1[i] \leftarrow P_2[i]$;
15             $C_2[i] \leftarrow P_1[i]$;
16 return $(C_1, C_2)$;
```

ber is smaller than the crossover probability P_{cross} (line 4), then it will perform the crossover operation. First, it verifies whether there exists any error position in chromosomes P_1 and P_2, by checking whether the size of their error positions is greater than 0 (line 5). If it is, then the feedback-directed crossover will be performed. The algorithm iterates through the chromosome (line 6), and copies the values of non-error positions from chromosome P_1 (resp., P_2) to the corresponding positions in chromosome C_2 (resp., C_1) (lines 7–10).

Otherwise, if both chromosomes P_1 and P_2 do not have any error position, the classic single point crossover operator is applied. First, an array index, $crossIndex \in \{0, \ldots, n-1\}$, is randomly selected. Subsequently, all values starting from the position *crossIndex* are copied from chromosome P_2 (resp., P_1) to chromosome C_1 (resp., C_2) (lines 12–15).

4. EVALUATION

We conducted experiments to evaluate our approach. Specifically, we attempted to answer the following questions:

RQ1. How is the *improvement* of the solutions that found by our method compared to the existing state-of-the-art methods in terms of the correctness of the solutions?

RQ2. What is the *runtime* of our method compared to the existing state-of-the-art methods?

RQ3. Can our method be *generalized* to different EAs?

RQ4. How *scalable* is our method in terms of the size of feature models?

4.1 Setup

4.1.1 Implementation

We have implemented our approach based on jMetal [12], which is a Java-based open source framework that supports multi-objective optimization with EAs. Sayyad *et.al* [29, 27] have made an extensive experiments to test how different EAs implemented on jMetal could contribute to the optimal feature selection. We use the EAs that are reported to work well in their experiments, and evaluate how the preprocessing and feedback-directed mechanisms affect these EAs. The EAs we are using for the evaluation are:

1. **IBEA**: Indicator-Based Evolutionary Algorithm [36]
2. **NSGA-II**: Nondominated Sorting Genetic Algorithm [11]
3. **ssNSGA-II**: Steady-state NSGA-II [13]
4. **MOCell**: A Cellular Genetic Algorithm for Multi-objective Optimization [23]

A brief overview of these EAs are provided in Table 3.

Table 2: Feature Models

Repo.	Model	Fea.	Cons.	F_p	F'_p	Ref.
–	JCS	12	13	2	–	–
SPLOT	Web Portal	43	36	4	–	[21]
	E-Shop	290	186	28	–	[35]
LVAT	eCos	1244	3146	54	19	[30, 35]
	uClinux	1850	2468	1244	1244	[5]
	Linux X86	6888	343944	156	94	[30]

4.1.2 Quality Indicators

To measure the quality of Pareto front, we make use of two indicators in this work: hypervolume [37] and spread [11].

a) ***Hypervolume (HV)***: Hypervolume of the solution set $S = (x_1, \ldots, x_n)$ is the volume of the region that is dominated by S in the objective space. In jMetal, although all objectives are minimized, but the Pareto front is inverted before the hypervolume is calculated. Therefore, the preferred Pareto front would be with the most hypervolume.

b) ***Percentage of Correctness* (%Correct)**: There might be solutions that violate some constraints in the Pareto front, since the correctness is an optimization objective that evolves over time. Solutions that are correct (i.e., without violating any constraint) are more useful to the user; therefore we are interested in the percentage of solutions that are correct in the Pareto front.

4.1.3 Feature Models and Attributes

The details of feature models used in the experiment are summarized in Table 2, with the repository information (*Repo.*), number of features (*Fea.*), number of constraints (*Cons.*), number of prunable features with the preprocessing method in Algorithm 1 (F_p), number of prunable features with the preprocessing method in [28] (F'_p), and literatures (*Ref.*) associated with each feature model.

JCS feature model is the feature model that we have used throughout the paper. Two feature models *Web Portal* and *E-Shop* are from SPLOT respository [22], which is a repository used by many researchers as a benchmark. The *Web Portal* model captures the configurations of Web portal product line, and the *E-Shop* model, which is one of the largest feature models in SPLOT, captures a B2C system with fixed priced products. These two models are chosen to facilitate

Table 3: Brief overview of EAs

Algorithm	Population	Operators	Criteria for Domination	Objective of the Criteria
IBEA	Main and Archive	Crossover, Mutation, Environmental Selection	The amount of domination are calculated based on quality indicator, e.g., hypervolume.	Favors user preferences.
NSGA-II	Main	Crossover, Mutation, Tournament Selection	Distances to closest point of each objective are calculated. Favors the point with greater distance from other objectives.	Favors more spread out solutions and absolute domination.
ssNSGA-II	Main	Crossover, Mutation, Tournament Selection	Similar to NSGA, with the exception that only one new individual inserted into population at a time.	Favors more spread out solutions and absolute domination.
MOCell	Main and Archive	Crossover, Mutation, Tournament Selection, Random Feedback	Similar to NSGA, a ranking and a crowding distance estimator is used, but bigger distance values are favored.	Favors more spread out solutions and absolute domination.

the comparison with [29]. To further evaluate the scalability of our methods, we make use of feature models from the Linux Variability Analysis Tools (LVAT) feature model repository [1]. The models in LVAT were reversed-engineered by making use of source code, comments and documentations of big projects such as Linux kernel and eCos operating system. Compared to the feature models in SPLOT, the feature models in LVAT contain a significant larger number of features and constraints, and have higher branching factors, but they have lower ratios of feature groups, and hence shallower tree structures in general.

Note that F_p always contains same number or more features than F'_p – this shows that our preprocessing method with Algorithm 1 has found more prunable features than [28]. In [28], their preprocessing method is based on static analysis. In particular, they detect disjunctions (rules) with only one feature, which means the feature is either a common feature or a dead feature. In addition, they investigate the disjunctions (rules) that include two features, if one of them is prunable in the first round, and the other one could be prunable as well. It is easy to see that our method based on SAT solving could detect all features that could be found by preprocessing method in [28], and it can be shown that F_p is always not fewer than F'_p.

4.1.4 Feature Attribute

Each feature in the feature models has the following attributes, which are the same as the attributes used in [29]:

1. ***Cost*** $\in \mathbb{R}$, records the number of cost incurred to use the feature. For each feature, the *Cost* value is assigned with a real number that is normally distributed between 5.0 and 15.0.
2. ***Used_Before*** $\in \{true, false\}$, indicates whether this feature was used before. The value of *Used_Before* is *true* if the feature has been used before, otherwise it is *false*. For each feature, the *Used_Before* value is assigned with a Boolean value that is distributed uniformly.
3. ***Defects*** $\in \mathbb{Z}$, records the number of defects known in the feature. For each feature, the *Defects* value is assigned with an integer number that is normally distributed between 0 and 10. However, if the feature has not been used before, the *Defects* value is set to 0.

4.1.5 Optimization Objectives

We introduce the five optimization objectives that we use in the experiment in the following. Note that since jMetal requires minimization of the objectives; all objectives listed here are objectives to be minimized.

Obj1. *Correctness*: minimize the number of violated constraints of the feature model.

Obj2. *Richness of features:* minimize the number of features that are not selected.

Obj3. *Cost:* minimize the total cost.

Obj4. *Feature used before:* minimize the number of features that have not been used before.

Obj5. *Defects:* minimize the number of known defects.

We specify correctness as an objective, rather than a constraint. The reason is that this allows EA to nudge the search towards feature models that contain fewer violated constraints, which eventually lead to valid feature models that do not contain violated constraints. Furthermore, note that some objectives are conflicting, e.g., Obj2 and Obj3, because the richness of features would imply a higher cost, but at the same time the cost needs to be minimized.

4.1.6 Configurations of EAs

Given an EA, we introduce the configurations for comparison.

1. **F+P**: This is the EA that makes use of feedback-directed crossover and mutation (Section 3.4) and preprocessing (Section 3.2) is applied before the execution of the feedback-directed EA.
2. **U+P**: The unguided version of EA with preprocessing (Section 3.2) applied before the execution of the unguided EA. We have demonstrated that, our method has found more prunable features than the preprocessing method of [28] in Section 4.1.3; therefore, U+P can be seen as an improved version of [28] with smaller search space.
3. **U**: The unguided version of EA without preprocessing, which is used by [29, 27].

Table 4: Evaluation with SPLOT

Model		IBEA			NSGAII			ssNSGAII			MOCell		
		F+P	U+P	U	F+P	U+P	U	F+P	U+P	U	F+P	U+P	U
E-shop	Time (ms)	6994	6369	7401	1906	2150	2548	15214	16863	17541	2822	3964	4463
	HV	0.3	0.18	0.19	0.26	0.2	0.17	0.24	0.22	0.22	0.24	0.19	0.22
	%Correct	**100.0**	0.0	0.0	**12.0**	0.0	0.0	**15.0**	0.0	0.0	**14.0**	0.0	0.0
Web Portal	Time (ms)	5678	4596	4646	433	483	546	8309	8315	8221	1033	1793	1857
	HV	0.32	0.2	0.23	0.3	0.24	0.21	0.3	0.21	0.24	0.31	0.21	0.22
	%Correct	**100.0**	1.0	0.0	**28.0**	0.0	0.0	**20.0**	1.0	0.0	**41.0**	0.0	0.0
JCS	Time (ms)	4681	4318	4735	271	269	301	7289	6834	6890	271	432	595
	HV	0.31	0.3	0.28	0.3	0.29	0.28	0.29	0.29	0.29	0.33	0.31	0.3
	%Correct	**96.0**	78.0	54.0	**27.0**	22.0	16.0	**31.0**	24.0	14.0	**34.0**	21.0	18.0

Table 5: Evaluation with LVAT

Model		IBEA		
		F+P	F'+P	U+P
eCos	Time (ms)	33245	51279	58561
	HV	0.25	0.21	0.18
	%Correct	**100.0**	61.45	0.0
	E50	6300	62400	–
uClinux	Time (ms)	50668	43876	46986
	HV	0.31	0.29	0.28
	%Correct	**100.0**	100.0	0.0
	E50	600	2100	–
Linux X86	Time (ms)	32758	31396	37472
	HV	0.2	0.2	0.22
	%Correct	0.0	0.0	0.0
	E50	–	–	–

4.1.7 *Parameter Settings*

For U, the same as [27], single-point crossover and bit-flip mutation are used as crossover and mutation operators, with crossover and mutation probabilities set to 0.1 and 0.01 respectively. These operators and probabilities also apply to $U + P$. For $F + P$, the feedback-directed crossover (Algorithm 4) and feedback-directed mutation (Algorithm 2) operators are used. The error mutation probability P_{emut}, mutation probability P_{mut}, and crossover probability P_{cross} are set to 1.0, 0.0000001, and 0.1 respectively. All other parameter settings for each EA are default settings of jMetal (e.g., population size is set to 100), and therefore are omitted here.

For SPLOT case study, we make use of 25000 evaluations using four EAs (IBEA, NSGAII, ssNSGAII, and MoCell). For the larger LVAT case study, we make use of 100000 evaluations using IBEA. For both case studies, we generate 10 sets of attributes. For each set of attributes, we run each EA repeatedly for 30 times, and report the medium values of the metrics. The evaluation results for SPLOT and LVAT are reported in Table 4 and Table 5 respectively.

We make use of Mann Whitney U-test [3] to test the statistical significant of $\%Correct$ indicator. We highlight the $\%Correct$ in **bold** for $F + P$, if the confidence level exceeds 95% when comparing $F + P$ and $U + P$.

The experiments were conducted on an Intel Core I7 4600U CPU with 8 GB RAM, running on Windows 7.

4.2 Evaluation with SPLOT

Table 4 demonstrates our results with SPLOT case study, where $Time(ms)$, HV, and $\%Correct$ represent execution time in milliseconds, hypervolume and percentage of correct solutions in the Pareto front.

Table 6: Improvement of EAs on SPLOT

	Time (ms)	HV	%Correct
IBEA	-690	0.08	72.33%
NSGA-II	97.33	0.05	15%
ssNSGA-II	400	0.04	13.67%
MOCell	687.67	0.06	22.67%

Table 7: Improvement of EAs on LVAT

	Time (ms)	HV	%Correct
IBEA	4290.5	0.02	50%

RQ1: We notice that the IBEA has outperformed other methods on the percentage of correctness. This is conformed to the observation in [29]. According to [29], this is because all EAs used in this case study (other than IBEA) use diversity-based selection criteria, which favor higher distances between solutions. For this reason, non-IBEA methods tend to remove solutions that crowded towards the zero-violation point, thus achieving lower scores on the percentage of correctness measure.

We also notice that for each EA, the configuration $U + P$ outperforms the configuration U on the percentage of correctness. This is because the preprocessing method has filtered away the prunable features, which makes the search space smaller. Hence EAs are more effective in the optimal feature selection. We also observe that $F + P$ outperforms $U + P$ constantly on the percentage of correctness. This is attributed to the feedback-directed crossover and mutation, which have effectively guided EAs to explore more promising region of the solution space for locating the optimal feature selection. The average improvement for the configuration $F + P$ over $U + P$ is summarized in Table 6, where the values are calculated by summing up the differences of $\%Correct$ between $F + P$ and U for all tested EAs, and divided by three (the number of test cases). Positive values mean improvements, while negative value mean the opposite. This has shown that our methods have provided an improvement on the percentage of correctness for all case studies using different EAs, especially in IBEA which has 72.33% improvement of correctness. These results answer research question RQ1.

RQ2: The runtime of configurations $F + P$, $U + P$, and U are comparable. There does not exist a configuration that has a clear advantage over the others in terms of the runtime. The reason is that all configurations go through the same number of evaluations. One might think that the configuration $F + P$ requires an extra calculation of the error position using Algorithm 3. In fact, the constraints also need to be enumerated for configurations $U + P$ and U during each round of evolution, in order to calculate the number of violated constraints. Therefore, the extra operation of $F + P$ is only the *getFeatures* function that is used in line 12 of Algorithm 3, which has a low complexity. On the other hand, $F + P$ and $U + P$ have shorter chromosome than U due to the preprocessing. However, these does not reflect much on the results, because the selection, mutation, and crossover for chromosomes could be done efficiently. These results answer research question RQ2.

RQ3: To answer research question RQ3, we notice that the percentage of correctness of all tested EAs (IBEA, NSGA-II, ssNSGA-II and MOCell) have been improved by using $F + P$. These results convey to us that, the preprocessing method and feedback-directed crossover and mutation have provided an advantage on the percentage of correctness and HV, regardless of the underlying EAs. The reason is that the preprocessing method effectively prunes the search space, and the feedback-directed crossover and mutation allow underlying EAs to use the feedback for faster finding of valid solutions. This also shows that the preprocessing method and feedback-directed crossover and mutation are general methods that could be applied for different EAs.

RQ4: To answer the research question RQ4, we make use of the E-shop model. E-shop model contains one of the largest set of features in the SPLOT repository [22]. The results show that, with $U + P$ and U, none of the EAs could locate a correct solution. On the other hand, with $F + P$, IBEA has achieved 100% of correctness, while NSGA-II, ssNSGA-II and MOCell have achieved 12–14% of correctness. We have also further evaluated for *50 millions* rounds of evolution for $U + P$ for IBEA. It has only achieved *46%* of correctness after 50 millions rounds which takes *3.25 hours*. In contrast, the configuration $F + P$ has achieved *100%* of correctness by just *6.9 seconds*.

To confirm the scalability of feedback-directed IBEA, we conduct the evaluation using LVAT in the next section.

4.3 Evaluation with LVAT

Table 5 demonstrates our results with LVAT case study with IBEA, where *E50* represents the number of executions required to obtain 50% of correct solutions in the Pareto front. Configuration $F' + P$ is the same as $F + P$, with the exception that the mutation probability P_{mut} is set to 0.01 (for $F + P$, $P_{mut} = 0.0000001$). The average improvement for the configuration $F + P$ over $U + P$ is summarized in Table 7. We notice that for *eCos* and *uClinux*, $F + P$ achieves 100% correctness for all cases, while $U + P$ does not find any correct solution after 100000 executions. Although $F + P$ achieves overall better runtime, it does not has clear advantage over $U + P$ for all models. These results have confirmed for the better percentage of correctness (RQ1) and comparable runtime (RQ2) of $F + P$ over $U + P$. For *Linux X86* which contains 6888 features, none of the methods ($F + P$, $U + P$, and U) have found a correct solution. Therefore, we resort to the "seeding method" proposed by [28].

In [28], the authors make use of two methods, i.e., SMT solver and IBEA of two objectives, for finding a correct solution (the "seed"), and plant the seed in the initial population of IBEA with the hope to find more valid solutions. We run two seeding methods proposed by [28] with $F + P$, and compare the results with the improved version of method proposed by [28], i.e., $U + P$.

First, Microsoft Z3 SMT solver [10] is used to find a seed. In our case, Z3 successfully finds a valid solution in around three seconds (we repeat for 30 times, and medium of the number of selected features is 1455). With the seed, $F + P$ successfully find 34 correct solutions using no longer than 30 seconds. In contrast, $U + P$ does not find any new solution after 30 minutes.

Second, IBEA with two objectives is used to generate the seed. In our case, $F + P$ uses less than *40 seconds* to get 36 correct solutions. While for $U + P$, it spends a total of *3.5 hours* of execution time for 30 correct solutions and *4 hours* of execution time for 36 correct solutions. $F + P$ has shortened the search time of $U + P$ for more than 200 times. In particular, $U + P$ spends 3 hours to generate the seed, and spends half an hour to obtain 30 correct solutions. And given another half an hour, $U + P$ finally obtains 36 correct solutions.

These results have shown that $F + P$ outperformed $U + P$ given both seeding methods in [28]. Note that the seed generated by IBEA with two objectives, is better than the seed generated by the Z3 SMT solver. Both $F + P$ and $U + P$ find more solutions using seed generated from IBEA with two objectives. This is conformed to the observation in [28]. According to [28], it is because the seed generated by IBEA with two objectives has more selected features, and the "feature-rich" seed allows the effective search of other valid solutions.

We also compare how the mutation parameter P_{mut} affects feedback-directed IBEA. Out of five models, $F' + P$ only performs poorer than $F + P$ in *eCos*. To better observe the effect, we make use of *E50*. It shows that $F + P$ obtained 50% of correct solutions in Pareto front in a smaller number of evaluations for all models, except *LinuxX86*. The results show that smaller P_{mut} leads to faster convergence of correct solutions in Pareto front. This is because smaller P_{mut} minimizes the modification of non-error positions; therefore, it allows IBEA to focus more on the correction of constraint violations.

4.4 Threats to Validity

There are several threats to validity. The first threat of validity is due to the fact that values for the feature attributes (i.e., *Cost*, *Defects*, and *Used_Before*) were randomly generated. This is due to difficulty in obtaining the attributes that are associated with real-world products since many of them are proprietary. To mitigate the effect of randomness, we generate 10 set of attributes for each case study. Furthermore, for each set of attributes, we run each EA repeatedly for 30 times, and report the medium values of the metrics. Future work should involve the use of real data for the evaluation.

The second threat of validity stems from our choice of using an exemplar parameter set (e.g., for crossover and mutation probability), which comes with the default setting of jMetal, in order to cope with the combinatorial explosion of options. To address these threats, it is clear that more experimentations with different feature models and experimental parameters are required, so that we could investigate effects that have not been made explicit by our dataset and experimental parameters.

5. RELATED WORK

Our work is related to the feasible feature selection. In [34], White *et al.* reduced the feature selection problem in

SPL to a multidimensional multi-choice knapsack problem (MMKP). They proposed a polynomial time approximation algorithm, called Filtered Cartesian Flattening (FCF), to derive an optimal feature configuration subject to resource constraints. Their evaluation showed that FCF can stably achieve the optimality above 90% even when the number of resources increases up to 91, while the optimality of Constraint Satisfaction Problem (CSP) based Feature Selection in [4] drops down to 30% when there are 91 resources.

Although FCF in [34] can achieve a highly optimal solution, but it requires significant computing time. To address the problem of scalability, Guo *et al.* [14] presented GAFES (a genetic algorithm based approach). The rationale is that GAs are quite suitable for the highly constrained problems, such as the feature selection (product derivation) problem. GAFES integrated a new *repair* operator for feature selection and also defined a *penalty* function for resource constraints. The evaluation showed GAFES may not beat the FCF and CSP in optimality, but it scaled up to large-scale models with a reasonable optimality.

Genetic algorithm only allows single objective function, and in addition, the method proposed in [14] repairs each solution explicitly, and does not take advantage of the evolution of GA algorithm for repairing. To address this problem, Sayyad *et al.* [29] investigated the use of different types of EAs that support multi-objective function for the optimal feature selection. They adopted 7 types of EAs, such as IBEA, NSGA-II and MOCell, to search for the optimal product. The results have shown that IBEA performs much better than other 6 EAs in terms of time, correctness and satisfaction to user preferences. In [27], Sayyad *et al.* improved [29] by turning down the crossover probability from 0.9 to 0.1 and mutation probability from 0.05 to 0.01, and they reported HV-mean and spread mean may increase by 5% to 10% in most cases. In [28], Sayyad *et al.* proposed the use of EA with simple heuristic in larger product lines from LVAT repository. They proposed the use of static analysis to identify prunable features for reducing search space, and the use of seeded techniques to find more correct products from Linux X86 Kernel. Our method has improved the method proposed in [29, 27, 28] by incorporating feedback-directed mechanism for EAs (cf. Section 4 for the evaluation). We also show that our method for finding prunable feature with Algorithm 1 is always not fewer than the method proposed in [28] (cf. Section 4.1.3 for the explanation and evaluation).

Our method is relevant to searching valid features for a feature model. In [25], an experiment for measuring the efficiency of BDD, SAT, and CSP solvers is conducted using feature models from SPLOT repository. They reported long run times for certain operations, and certain runs are cancelled if exceeded three hours. They also reported an exponential runtime increase with the number of features for non-BDD solvers on the "valid" operations. In [26], the state-of-art solvers, e.g., JavaBDD BDD solver, the JaCoP CSP solver and the SAT4J SAT solver, were used to answer the questions such as "derive one valid product from a feature model" and "number of products". They found that CSP and SAT solvers have exponential runtime increase as the feature size of feature model increases, and BDD requires a maximum of 28 seconds to derive a valid product for web-portal, even without considering the quality of feature attributes. Thus, these automated reasoning techniques can be precise, but generally not scalable for large feature models. Our work complements with their work by using feedback-directed evolutionary algorithm that scales well for large feature models. In [15] introduces five novel parallel algorithm for Multi-Objective Combinatorial Optimization (MOCO) to allow parallel processing. Our work complements with them by considering feedback-directed mechanism for MOCO problem using feedback-directed EAs.

Our work is also related to the feedback-directed methods in software engineering. Pacheco *et al.* [24] proposed RANDOOP, a feedback-directed mechanism for performing random test. It uses erroneous results of previous method invocation to generate a better random test. Clarke *et al.* [7] proposed CEGAR, which uses spurious counterexamples as a feedback to guide the refinement process. Our method is on feedback-directed methods in EAs for the optimal feature selection. This work is also related to using evolutionary algorithms and SMT solvers in tackling software engineering problems. In [33], we make use of genetic algorithm in calculating the optimal recovery plan during service failure. In [32, 20], we calculate the local time requirements of individual components given the global time requirement of the system with Z3 SMT solver [10]. In this work, our focus in on making use of evolutionary algorithms and Z3 SMT solver in tackling optimal feature selection.

In addition to the SPL domain, multi-objective evolutionary optimization algorithms (MEOAs) have also been applied to various software engineering problems. In [17], Harman *et al.* proposed the term Search-Based Software Engineering (SBSE), and reported that the surveyed and proposed optimization techniques for SE problems by 2001 were all single-objective based. Seeing the potential of using multi-objective optimization, Harman [16] discussed about the possible usage of the meta-heuristic search techniques such as: simulated annealing and genetic algorithm. Harman considered it insensible combination of multiple metrics into an aggregate fitness in the way of assigning coefficients, and further suggested to use Pareto optimality rather than aggregate fitness.

6. CONCLUSION AND FUTURE WORK

In this work, we have presented a novel technique by introducing a feedback-directed mechanism into various EAs. Our approach is based on analyzing violated constraints, and uses the analyzed results as a feedback to guide the process of crossover and mutation operators. In addition, we also introduce a preprocessing technique to reduce the search space, by filtering away the prunable features in all feasible feature sets. Our evaluation shows that both the preprocessing technique and the feedback-directed mechanism have improved over existing unguided EAs on the optimal feature selection. Without compromising on running time, the feedback-directed IBEA successfully found 72.33% and 75% more correct solutions for case studies in SPLOT and LVAT repositories, compared to the unguided IBEA. In addition, with "seeding method" proposed by [28] and feedback-directed IBEA, we have reduced the running time from about *3.5 – 4 hours* to less than *40 seconds* for finding 34 correct solutions.

As future works, first, we plan to find other types of feedback that could be incorporated in EAs, to address the scalability problem of large feature models, such as Linux X86. Second, we would investigate extensibility of our method to other software engineering problems. Lastly, we plan to further evaluate the method using different case studies.

7. REFERENCES

[1] Linux Variability Analysis Tools (LVAT) Repository. https://code.google.com/p/linux-variability-analysis-tools/source/browse/?repo=formulas.

[2] SAT4J – The boolean satisfaction and optimization library in Java. http://www.sat4j.org/.

[3] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, pages 1–10, 2011.

[4] D. Benavides, P. T. Martín-Arroyo, and A. R. Cortés. Automated reasoning on feature models. In *CAiSE*, pages 491–503, 2005.

[5] T. Berger, S. She, R. Lotufo, K. Czarnecki, T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the systems software domain. Technical report, University of Waterloo, 2012.

[6] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. H. Obbink, and K. Pohl. Variability issues in software product lines. In *PFE*, 2001.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, pages 154–169, 2000.

[8] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Professional, 3rd edition, Aug. 2001.

[9] K. Czarnecki and U. W. Eisenecker. *Generative programming - methods, tools and applications.* Addison-Wesley, 2000.

[10] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[11] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-II. *IEEE Trans. Evolutionary Computation*, 6(2):182–197, 2002.

[12] J. J. Durillo and A. J. Nebro. jmetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, 2011.

[13] J. J. Durillo, A. J. Nebro, F. Luna, and E. Alba. On the effect of the steady-state selection scheme in multi-objective genetic algorithms. In *EMO*, pages 183–197, 2009.

[14] J. Guo, J. White, G. Wang, J. Li, and Y. Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, 84(12):2208–2221, 2011.

[15] J. Guo, E. Zulkoski, R. Olaechea, D. Rayside, K. Czarnecki, S. Apel, and J. M. Atlee. Scaling exact multi-objective combinatorial optimization by parallelization. In *ASE*, 2014.

[16] M. Harman. The current state and future of search based software engineering. In *FOSE*, pages 342–357, 2007.

[17] M. Harman and B. F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, 2001.

[18] I. Jacobson, M. L. Griss, and P. Jonsson. *Software reuse - architecture, process and organization for business.* Addison-Wesley-Longman, 1997.

[19] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, November 1990.

[20] Y. Li, T. H. Tan, and M. Chechik. Management of time requirements in component-based systems. In *FM*, pages 399–415, 2014.

[21] M. Mendonça, T. T. Bartolomei, and D. D. Cowan. Decision-making coordination in collaborative product configuration. In *SAC*, pages 108–113, 2008.

[22] M. Mendonça, M. Branco, and D. D. Cowan. S.P.L.O.T.: software product lines online tools. In *OOPSLA Companion*, pages 761–762, 2009.

[23] A. J. Nebro, J. J. Durillo, F. Luna, B. Dorronsoro, and E. Alba. Mocell: A cellular genetic algorithm for multiobjective optimization. *Int. J. Intell. Syst.*, 24(7):726–746, 2009.

[24] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84. IEEE Computer Society, 2007.

[25] R. Pohl, K. Lauenroth, and K. Pohl. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *ASE*, pages 313–322, 2011.

[26] R. Pohl, V. Stricker, and K. Pohl. Measuring the structural complexity of feature models. In *ASE*, pages 454–464, 2013.

[27] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Optimum feature selection in software product lines: Let your model and valuesguide your search. In *CMSBSE*, pages 22–27, 2013.

[28] A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable product line configuration: A straw to break the camel's back. In *ASE*, 2013.

[29] A. S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: a case study in software product lines. In *ICSE*, pages 492–501, 2013.

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

[31] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667, 1994.

[32] T. H. Tan, É. André, J. Sun, Y. Liu, J. S. Dong, and M. Chen. Dynamic synthesis of local time requirement for service composition. In *ICSE*, pages 542–551, 2013.

[33] T. H. Tan, M. Chen, É. André, J. Sun, Y. Liu, and J. S. Dong. Automated runtime recovery for qos-based service composition. In *WWW*, pages 563–574, 2014.

[34] J. White, B. Dougherty, and D. C. Schmidt. Selecting highly optimal architectural feature sets with filtered cartesian flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.

[35] Y. Xue, Z. Xing, and S. Jarzabek. Understanding feature evolution in a family of product variants. In *WCRE*, pages 109–118, 2010.

[36] E. Zitzler and S. Künzli. Indicator-based selection in multiobjective search. In *PPSN*, pages 832–842, 2004.

[37] E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Trans. Evolutionary Computation*, 3(4):257–271, 1999.