Optimal Feature Selection via Evolutionary Algorithms and Constraint Solving

Yibo Wang and **Lothar Hotz** [1]

Abstract. In software development, product lines especially with feature models are promising technologies to manage variability of software products. A new challenge in deriving software products from the product line is not only to select a set of features, which do not incur any feature inconsistencies, but to optimize multiple objectives (e.g. cost minimization and maximization of feature reuse) at the same time. This challenge is a constrained multi-objective optimization problem and has been proved to be difficult. In this paper, we approach this problem by utilizing the state-of-the-art method SATIBEA, which combines a multi-objective evolutionary algorithm with constraint solving techniques. The contribution of our approach is that we enhanced SATIBEA in two ways: by improving its mutation operator and by providing a novel crossover operator. Our empirical experiment results have shown that our approach SATIBEA+ improved SATIBEA noticeably, in providing more valid and more qualitative feature selections in terms of the standard measures such as hypervolume and Pareto front size.

1 INTRODUCTION

In many engineering fields, more and more emphasis is put on product line technology due to the need for customization with low efforts and in short terms. Planned variability, expressed in product models such as feature models [8], allows a smart configuration of products and services. However, products and services have to fulfill growing non-functional requirements, such as safety or cost, due to the increased competition on the markets. Such requirements more often demand for optimizations for achieving best fitted products to users' needs. Meanwhile, multiple non-functional requirements have to be considered together during optimization, although they might be non-commensurable or competing.

The field of product configuration is trying to cope with such problems. A configuration is a description of all parts with their appropriate parameters that are needed to build the product that hopefully will fulfill the given requirements. In product configuration, the configuration problem starts from certain user requirements and from a configuration model which implicitly describes all configurations of a certain domain. By using reasoning technology (e.g. a SAT-Solver [16]), the configuration problem is solved by automatically creating configurations. In the specific case of software product lines (SPL), configuration models are often expressed by feature models representing all features of a product and the configuration consists of selected features from such models. In this paper, this general task is further enhanced by taking optimization into account.

[1] Department of Informatics, University of Hamburg, Germany, email: wang@informatik.uni-hamburg.de

Technologies that approach this problems are SAT-solvers that allow the computation of valid feature configurations [7, 16] and evolutionary algorithms that are capable to compute solutions for multi-objective optimization problems [5, 10, 12, 14]. Evolutionary algorithms start with a set (*an initial generation*) of individuals (here, potential feature configurations), and modify them by mutation and crossover in the following generations. A mutation changes one individual (here, by selecting other features) and crossover combines a few individuals to a further individual (here, a new configuration generated by combination).

Syyad et al. [12] showed that for solving multi objective problems, especially with an increased number of objectives, indicator-based approaches (IBEA) outperform dominance-based ones. Dominance-based approaches rank solutions according to absolute dominance while indicator-based approaches provide an "amount" of dominance by computing a value which incorporates user preferences. Based on this, Henard et al. [5] introduced SATIBEA, which applies a SAT-solver in the mutation operator to correct a configuration for returning a valid mutation. However, we have found that SATIBEA often fails for complex feature models with thousands of features and constraints. In addition, the standard 1-point crossover used in SATIBEA generates in most cases worse offsprings (with more feature violations) than their parents, due to an arbitrary combination from parents. Thus, our work complements the existing work of SATIBEA (leading to SATIBEA+). The main contributions of our approach SATIBEA+ can be summarized as following:

- We improve mutation operator of SATIBEA so that it finds more valid configurations for complex feature models.
- We provide a novel crossover operator which reduces the number of feature violations for an invalid solution by learning from another configuration.
- We show that SATIBEA+ outperforms SATIBEA, in providing more valid and qualitative configurations in terms of hypervolume and Pareto front size.

The paper is organized as follows: Section 2 presents the underlying technologies which are used in our approach (Section 3). For evaluating this approach, we defined research questions (Section 4) and verify them through experiments (Section 5 and Section 6). The sections 7, 8, and 9 provide discussions, related work, and a conclusion.

2 BASIC TECHNOLOGIES

In SPL, the variability of products (i.e., the configuration space) is typically represented as feature models. They express all decision variables that are subject to select and optimize (Subsection 2.1). A main aspect of this paper is to support the combination of technologies that are capable to compute consistent configurations and that are capable to optimize those configurations. In our approach, we use SAT-solvers for consistency checking (Subsection 2.2) and evolutionary algorithms for computing multi-objective optimization problems (Subsection 2.3). The rest of the paper shows, how we incorporate both technologies, i.e., the SAT-solver used as a mutation method for the evolutionary algorithm.

2.1 Feature Models and Feature Constraints

In software product lines, *feature models* are often used to express *variability* of products. Features can be modeled with mandatory, optional, and alternative constraints, as well as attributes (*extended feature models* [2]). Furthermore, relations between features can be expressed such as *exclude* or *require* which are all considered here as *integrity constraints* or simply *constraints* (see [11]). Thus, constraints relate to features. In this paper, we divide a feature model into features and constraints on one side to form the *consistency part* of the feature model and feature attributes on the other side to form the *optimization part* of the feature model. In the general form, feature attributes might also belong to the consistency part. Figure 1 presents an example also later used in the paper.

The task is now to select features from the feature model given some preferred features so that a valid configuration is created (feature selection) that fulfills the constraints. For simplicity, in this paper we consider the special case where the set of preferred features is empty. This can be considered as a constraint satisfaction problem, defined as follows [7]:

Definition (Constraint Satisfaction Problem – CSP). A constraint satisfaction problem (CSP) is defined by a triple (V, D, C) where V represents a set of finite domain variables $V = \{v_1, v_2, ..., v_n\}$, D represents variable domains $D = \{dom(v_1), dom(v_2), ..., dom(v_n)\}$, and C represents a set of constraints defining restrictions on the possible combinations of variable values ($C = \{c_1, c_2, ..., c_m\}$).

A *feature selection problem* is a CSP where variables represent features defined in a feature model.

A solution to a given CSP = (V, D, C) can be defined as follows:

Definition (CSP Solution). A solution for a given CSP = (V, D, C) is represented by an assignment $A = \{ins(v_1), ins(v_2), ..., ins(v_k)\}$ where $ins(v_i) \in dom(v_i)$. We require solutions to be *complete*, i.e., to be represented by an assignment where each variable in the definition of the CSP is instantiated and *consistent* which means that the assignment A is consistent with the constraints in C.

Thus, a *feature selection* (or *configuration*) is a CSP solution of a feature selection problem.

2.2 SAT-based constraint resolving

A propositional logic formula consists of binary variables and the operators AND, OR, and NOT. A truth value ($TRUE$ or $FALSE$) can be assigned to each binary variable. By assigning a truth value to each variable a formula can be satisfied, i.e., results to $TRUE$. A boolean satisfiability problem (SAT) is given by the task to check whether a given formula is satisfiable. As an extension, a further task is to assign values to not pre-assigned variables to make a formula satisfiable.

A feature selection problem can be mapped to a SAT problem by introducing binary variables for each feature and map constraints to formulas. A SAT solver, such as SAT4J [1] can be used to check the constraint violations of a solution or to compute valid variable assignments for not pre-assigned variables.

2.3 Multi-objective Evolutionary Optimization Algorithms (MOEAs)

In multi-objective optimization, multiple objective functions have to be optimized at the same time. We define a feature optimization problem as follows (see [5]): Compute $min(F_1(x), F_2(x), \ldots, F_k(x))$ with k the number of objective functions and $x \in X$ is the set of possible feature configurations. Each $F_i(x)$ is an objective function based on feature attributes. We introduce the boolean attribute "selected" to indicate if a feature is selected in the configuration or not. Each $F_i(x)$ has to be minimized. In evolutionary algorithms, different $F_i(x)$s are combined to a single value, the *fitness value*, to evaluate a feature configuration.

Let x_1 and x_2 be two potential solutions to the problem. We say that x_1 dominates x_2, if and only if $\forall i \in \{1, \ldots, k\} : F_i(x_1) \leq F_i(x_2)$ and $\exists i \in \{1, \ldots, k\} : F_i(x_1) < F_i(x_2)$. Given $x_1, \ldots, x_n$ potential solutions to the multi-objective optimization problem, the Pareto front corresponds to the subset of these potential solutions that are non-dominated by the others.

Each solution in the Pareto front is optimal in the sense that it cannot be improved in one objective function without degrading another one. Furthermore, all solutions in a Pareto front are equally optimal. However, solutions obtained by MOEAs are not exactly the Pareto front, but an approximation of it. Two properties are used to evaluate the quality of the obtained solutions: convergence and diversity. The first one describes how near they are to the Pareto front, while the second one indicates how uniformly they distribute. A good solution would have good convergence and diversity at the same time.

As pointed out in the introduction, indicator-based evolutionary algorithms (IBEA) provide a mean for solving multi-objective optimization problems (MOP). The rest of the paper will explain how those technologies are applied for computing optimal feature configurations.

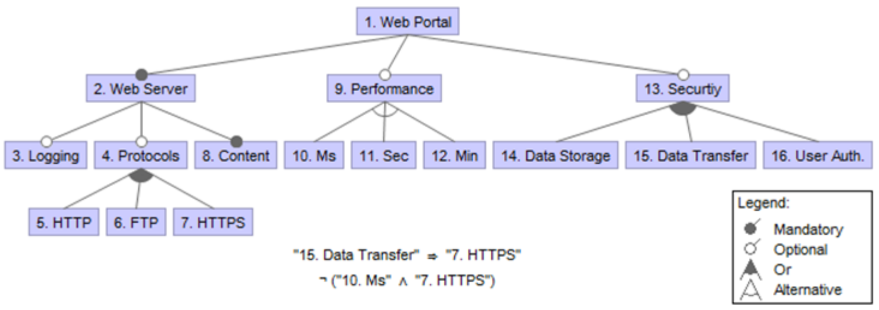
3 THE PROPOSED APPROACH - SATIBEA+

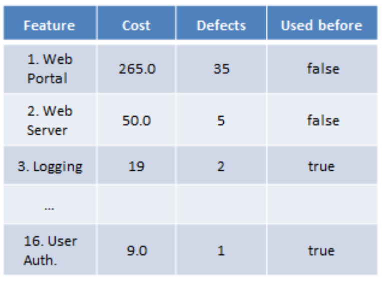
The main task of the automatic generation of configurations of SPLs considered here is to find a set of valid and optimal feature selections in consideration of multiple objectives. It means, at the end of the search process, the resulting configurations must be valid. In other words, the number of constraint violations should be zero in the final configurations. Although invalid configurations are permitted as intermediate results, minimization of the number of invalid constraints should be defined as an objective for the search process. Moreover, the more valid configurations at the end of the process are generated, the more useful is the search result, because more feasible configuration could be given to decision makers for the final selection.

In order to reduce the number of feature violations and increase the percentage of the valid configurations, the invalid configurations should be replaced by valid ones or at least by "better" ones (with less constraint violations) gradually during the search process. Our approach is based on the SATIBEA approach, so we name it "SATIBEA+". Similar to SATIBEA, we also use the SAT-solver to repair invalid configurations. In addition, SATIBEA+ can also change an invalid configuration into a valid or a "better" one by learning from an another configuration. In our approach, we extend the "smart" operators of "SATIBEA" to "smart+" operators to achieve this goal.

3.1 "Smart" and "Smart+" Operators

We introduce two "smart+" operators as an extension of "mboxsmart" operators of [5]. The operators are called "smart", because they





Feature	Cost	Defects	Used before
1. Web Portal	265.0	35	false
2. Web Server	50.0	5	false
3. Logging	19	2	true
...			
16. User Auth.	9.0	1	true

Figure 1. Example of a feature model including feature attributes based on [9]

can turn an invalid configuration into a full valid one or at least makes it better. Furthermore, they change a configuration only slightly (the change will be as little as possible).

3.1.1 Smart and Smart+ Mutation

This type of mutation operator is an unary operator acting on a single configuration aiming to change it from invalid to valid. Thus, the output of this operator should be a configuration without any constraint violation.

Smart Mutation of SATIBEA: Before mutation, constraint violations are calculated for a configuration. Then, features in this configuration are divided into two groups: the "bad" ones and the "good" ones. The former refers to the features, which are involved in at least one of the constraint violations. The latter refers to the features, which do not incur in any constraint violation. Then the feature assignments of bad ones will be removed while the assignments for good ones remain unchanged. After that, smart mutation will inquire the SAT-solver for a valid configuration under this assumption.

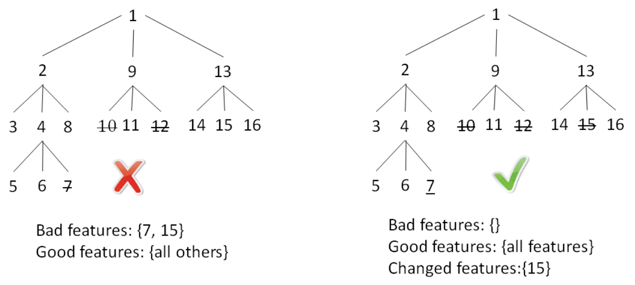
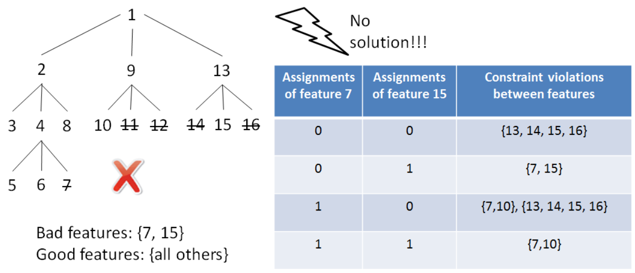


Figure 2. An positive example for the smart mutation in SATIBEA

Considering the configuration in the Figure 2 for the feature model defined in the Figure 1, it has a constraint violation of the *require* relation between 7 and 15 (the left side of Figure 2). In the following figures, the unselected features are struck out. If we remove the assignments of the features 7 and 15 and give the assignments of the rest features to the SAT-solver, then it will return a valid configuration without any violations (the right side). In this case, only the bad features will be changed (Feature 15). But what happens, if we apply this operator to repair the other invalid configuration shown in Figure 3 (the left side)? Like in Figure 2, the *exclude* relation between feature 7 and 15 is violated in this configuration.

According to all of the possible feature assignments of 7 and 15 (the right side), it is impossible to find a valid configuration. The more features and the more feature constraints a feature model has, the more likely no solution could be found by a SAT-solver. We have found that for the complex feature models (eCos, FreeBSD and Linux in Table 1) used in our experiment, the mutation operator of SATIBEA failed to find even just one valid configuration.



Assignments of feature 7	Assignments of feature 15	Constraint violations between features
0	0	{13, 14, 15, 16}
0	1	{7, 15}
1	0	{7,10}, {13, 14, 15, 16}
1	1	{7,10}

Figure 3. An negative example for the smart mutation in SATIBEA

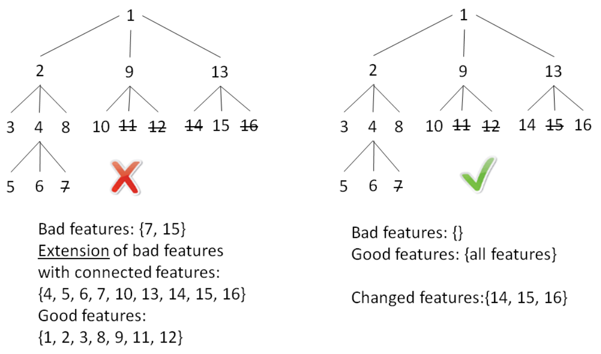


Figure 4. Using the new smart+ mutation to repair the configuration in the negative example

Smart+ mutation of SATIBEA+: In order to overcome this shortcoming, we introduce the concept of *connected features*. A feature x is "connected" with a feature y, if both features appear in the same feature constraint. For a feature x, we iterate all feature constraints and save all connected features with x in a set S. Then, we call S the *connected features* for the feature x. For example, the feature 7 has the connected features $\{4, 5, 6, 10, 15\}$ in the Figure 1, because there is at least one feature constraint defined between 7 and those. To search for an invalid configuration, we extended the set of the bad features with their connected features. It inquires the SAT-solver for a valid configuration while remaining the assignments of the good features (but without keeping the assignments of the connected features). Thus, we can now repair the invalid configuration in the Figure 3. As shown in Figure 4, not only the bad features, but also the their connected features, can be changed (features 14, 15 and 16 have been changed). With this extension, the failure rate of the repair operator for the Linux configuration model has been reduced up to 30% (see Section 6).

3.1.2 Smart+ Crossover

This operator is a binary operator acting on two configurations (called the parents in EA) aiming to reduce the constraint violations of the first one. Thus, the output of this operator should be a new configuration (called the *offspring* in EA) with a reduced number of constraint violations. The crossover operator uses a crossover point which splits a configuration (a list of features) into two parts, *before* and *after* the point.

1-point Crossover of SATIBEA: This crossover operator combines two configurations in a new one by applying the feature assignments of the first one before the randomly selected crossover point and applying the feature assignments of the second one after the crossover point. Because the crossover point is selected randomly and feature values are simply copied without consideration of constraints, the number of constraint violations will be reduced also randomly. The more features and the more feature constraints a feature model has, the more likely no reduction of constraints could be achieved. We have found that for the complex feature models used in our experiment, the constraint violations has been even increased in most cases.

Algorithm 1 smart+ crossover

```
 1: Input: p1, p2
 2: offspring = p1.copy()
 3: violatedConstraintsP1 = getViolatedConstraints(p1);
 4: violatedConstraintsP2 = getViolatedConstraints(p2);
 5: for each constraint in violatedConstraintsP1 do
 6:     if constraint not in violatedConstraintsP2 then
 7:         for each featureX in constraint do
 8:             valueOf(offspring, featureX) = valueOf(p2, featureX);
 9:         end for
10:     end if
11: end for
12: return offspring
```

Smart+ crossover in SATIBEA+: Instead of generating an offspring by exchanging values of configurations arbitrarily, we correct feature violations in one configuration by learning "good" feature assignments from the other one. We introduce the smart+ crossover operator in Algorithm 1. The inputs are two configurations *p1* and *p2* as parents (line 1) and the output is the resulting configuration *offspring* (line 12). The algorithm begins with copying *p1* as the prototype of the *offspring* (line 2). Then it calculates the violated constraints of both parent configurations (line 3-4). For each violated constraint in *p1* (also in offspring, because it is the copy of *p1*), if we could find feature assignments in *p2*, which do not violate this constraint, then we use these the "good" feature assignments of *p2* to replace the "bad" ones in *offspring* (5-11). Please note that smart crossover is not aiming to resolve all invalid feature constraints, just as smart mutation does. It will only "improve" a configuration. It could generate an offspring without any invalid feature constraint, but not necessarily.

Suppose the configuration *Parent 1* in the Figure 5, which has 4 feature invalid constraints. Its counterpart *Parent 2* has only 2 feature constraints. Because the first three constraints of *Parent 1* are not violated in *Parent 2*, *Parent 1* could replace their problem features 4, 6, 7, 13, 14 with the "good" feature assignments of Parent 2. Thus, the generated *Offspring* has only one feature violation, which could not be fixed anyhow (because it is violated in both configurations). With this extension, the number of constraint violations for the Linux configuration model can be reduced in almost 70% of the cases (see Section 6).

3.2 Other changes against SATIBEA

3.2.1 *Mating Selection*

It is a selection operator which acts on a set of configurations aiming to generate parents for crossover. In evolutionary algorithms, solutions with better fitness values should have better opportunities to be selected as parents for the crossover.

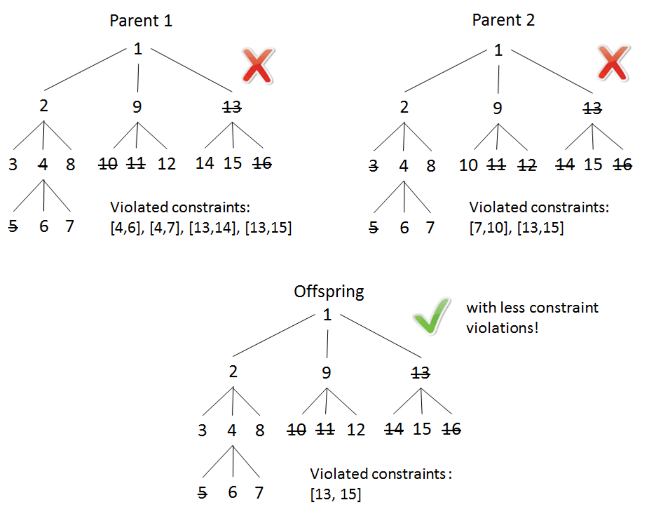


Figure 5. An example by using smart+ crossover to improve a configuration

Mutation selection of SATIBEA: Binary tournament strategy [14] has been applied in SATIBEA. From two randomly selected configurations, the better one (with the better fitness value) will be selected as the parent.

Changes in SATIBEA+: Instead of selecting parents by considering only fitness values, we make additional limitations on the selection process. Recalling the smart crossover defined in the last section, the "bad" configuration will get improved by learning from the "good" one. Thus, we limit a "good" configuration only on the ones without any constraint violations. It can be selected only from the set of valid configurations. In contrast, a "bad" configuration can be selected from any configurations, as in SATIBEA.

3.2.2 *Smart Replacement*

This operator is aiming to add new solutions randomly in the current population.

Smart Replacement of SATIBEA: In SATIBEA, it picks up a configuration from the current population randomly and replace it with a new valid configuration, which is also generated randomly.

Changes in SATIBEA+: Because this operator adds valid configurations into the population arbitrarily and periodically, it could produce uncontrolled influence on the final result so that effectiveness of smart operators could not be measured properly. In this paper, we investigated the influence of this operator.

3.3 The SATIBEA+ approach

The simplified activity diagram of SATIBEA+ can be seen in Figure 6. It augments SATIBEA with the new smart operators (steps 5-7). The other activities (steps 1-4 and 8) are described in detail in [12] and [5]. Like other MOEAs, it evolves a population of configurations from generation to generation (circles from 3 to 7 and back to 3) aiming to optimize given objectives. The difference is that features constraints are taken into account in the following activities:

- Steps 2 and 3: The number of constraint violations is considered as an extra objective to minimize. Its value is integrated in the calculation of fitness value by IBEA.
- Step 5: Smart mating selection selects a "good" parent (without constraint violations) and a "bad" parent (possibly with constraint violations) for crossover.

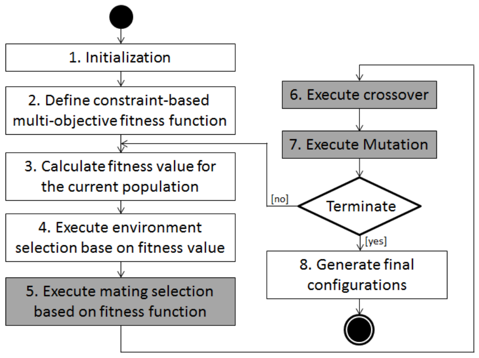

1. Initialization
2. Define constraint-based multi-objective fitness function
3. Calculate fitness value for the current population
4. Execute environment selection base on fitness value
5. Execute mating selection based on fitness function
6. Execute crossover
7. Execute Mutation
Terminate
[no]
[yes]
8. Generate final configurations

Figure 6. Activity diagram of SATIBEA+. Extensions compared to SATIBEA are marked in gray.

- Step 6: Smart+ crossover reduces the number of constraint violations in generating an offspring. It repairs the "bad" parent by learning from the "good" one.
- Step 7: Smart+ mutation eliminates constraint violations in changing an invalid configuration into a valid one.
- Step 8: Only the valid configurations will be considered as result, because configurations with any constraint violations are useless for the end user.

4 RESEARCH QUESTIONS

We conducted experiments to empirically compare the performance of SATIBEA+ with that of SATIBEA. Specially, we attempted to find answers for the following questions:
RQ1. How are the results found by SATIBEA+ compared to the results found by SATIBEA? Is the improvement repeatable on feature models with different sizes?
RQ2. How well does the new mutation operator affect the performance of search techniques?
RQ3. Is the new crossover operator actually more effective than the classic 1-point crossover used by SATIBEA?
RQ4. How much does the smart replacement operator affect the results?
RQ5. How does the algorithm implemented by SATIBEA+ work according to performance compared to SATIBEA?

5 EXPERIMENTAL DESIGN

5.1 Experimental Subjects

The study uses 4 feature models from the public feature model repositories SPLOT[2] and LVAT[3]. The characteristics of these feature models used in the experiment are summarized in Table 1 with the repository information (Repository), the name of the feature model (Model), the number of features (#Fea.), and the number of constraints (#Cons.).

Web Portal is a small demo feature model, which we are using to illustrate our approach (Figure 1). eCos and Linux X86 are examples of middle and big-sized feature models. They were reverse engineered by analyzing source codes, comments, and documentations of Linux kernel and eCos operation systems. Following the experiments used by [5, 9, 12, 15], each feature of used feature models is augmented by 3 attributes: *cost* $\epsilon\ \mathbb{R}_{\geq 0}$, *defects* $\epsilon\ \mathbb{Z}_{\geq 0}$ and *used_before* $\epsilon\ Boolean$. The values for these attributes have been set arbitrarily in the feature model with an uniform distribution (*cost* in (5.0, 15.0), *defects* in (0, 10) and *used_before* in (true, false)). There is only one dependency which should be considered by the generation of attribute values. It can be formulated as: if (not *used_before*) then *defects*=0.

Repository	Model	#Fea.	#Cons.
SPLOT	Web Portal	16	26
LVAT	eCos	1244	3146
	FreeBSD	1369	62183
	Linux X86	6888	343944

Table 1. Feature models used in the experiment

5.2 The optimization problems

For the feature models introduced above, we are optimizing the following objectives formulated as minimization problems uniformly. They are calculated by the formula defined in Table 2.

- *Correctness*: Any constraint violation is not allowed in the final configurations. But as an objective in the optimization framework, we intend to minimize it.
- *Richness of features*: In a configuration, we want to have as many features selected as possible. It implies that the number of unselected features should be minimized.
- *Cost*: The total cost for a configuration should be minimized.
- *Defects*: The number of defects, which are caused by selected features, should be also minimized.
- *Feature used before*: In order to reduce the product risk, we are seeking to find the configurations that have minimized number of unused features.

Objective	Calculation formula for objectives
Correctness	$\sum Cons.(violated = \textbf{true})$
Richness of features	$\sum Fea.(selected = \textbf{false})$
Cost	$\sum Fea.(selected = \textbf{true}).cost$
Defects	$\sum Fea.(selected = \textbf{true}).defects$
Feature used before	$\sum Fea.(selected = \textbf{true}\&used\_before = \textbf{true})$

Table 2. Objectives used in the experiment

5.3 Implementation and Experimental Settings

1. Implementation:
We used jMetal[3], an open-source Java framework for multiobjective optimization and SAT4j[1], a open-source library of SAT solvers to implement SATIBEA+. In addition, we compared SATIBEA+ with the original SATIBEA algorithm[4].
2. Parameter settings:
All the experiments were performed on a computer with Quad Core@2.90 GHz CPU and 16 GB RAM, running on Windows 7. In order to compare with SATIBEA under fair conditions, we used exactly the same parameters for the evolutionary algorithm as the ones of [5]. Thus, SATIBEA+ differentiates SATIBEA only from EA-operators. Table 3 lists the used parameter settings and gives for each parameter a short description. To evaluate the influence of the smart replacement, we execute our experiment in two variations, namely with smart replacement and without smart replacement (last row of Table 3). In order to avoid the problem of genetic drift (diversity loss) described in [14], the smart crossover and smart mutation is only executed with low frequencies. Please

[2] Software Product Line Online Tools, `http://splot-research.org/`
[3] Linux Variability Analysis Tools, `http://code.google.com/p/linux-variability-analysis-tools/`

[4] Implementation of SATIBEA, `http://research.henard.net/SPL/ICSE_2015/`

note, the focus of this paper, was on comparing SATIBEA+ with SATIBEA, rather than tuning the parameters to achieve the best performance, which could be explored in future work.

Parameter	Explanation	Setting
Population size	Number of new configurations in the current population	300
Archive size	Number of configurations from the last population	300
Crossover probability	Probability that a crossover is executed	0.05
Probability for using standard mutation	Probability that a standard mutation is executed	0.98
Probability for bit flipping in the standard mutation	Probability that a changeable feature (not mandatory and dead feature) is flipped	0.001
Probability for using smart mutation	Probability that a smart mutation is executed	0.01
Probability for using smart replacement	Probability that a smart replacement is executed	0.01 or 0

Table 3. Parameter settings

5.4 Performance metrics

To evaluate the studied approach, we measure the calculated Pareto front in three directions: convergence, diversity, and computation time. Convergence metrics evaluate the effectiveness of the solutions in terms of their closeness to the optimal Pareto front, while diversity metrics measure the distribution of the solution set. Computation time is the length of time required to perform an algorithm, which represents its computational complexity. In our approach, they are represented by the following three indicators:

1. Hypervolume (HV)
   The hypervolume indicator, associated with a solution set S is given by the volume of the objective space portion that is weakly dominated by the set S [14]. It combines convergence and diversity measurement in a single indicator. In jMetal, all objectives are to be minimized and the Pareto front is inverted before the HV is calculated. Thus, the more HV value a solution set has, the more qualitative it is.
2. Pareto front size (PFS)
   Like HV, it is a combined indicator for the measurement of convergence and diversity. Although correctness is defined as a separate objective to be minimized in Table 2, there might be still some invalid configurations (with constraint violations) in the results. We use PFS to measure the number of unique and valid solutions in the obtained Pareto front. Because duplicated configurations are treated as a single one, it also gives a hint about the diversity of the solution set. A higher PFS value is preferred, because more valid configurations can be presented to the user.
3. Execution time (ET)
   This run-time indicator calculates the duration of the evolutionary algorithm for a given number of iterations. In each iteration, operations such as crossover and mutation will be executed. A higher ET value denotes a higher time complexity of an algorithm.

6 EXPERIMENTAL RESULTS

This section presents the results when applied to the 4 feature models. The performance metrics achieved by SATIBEA+ and by SATIBEA have been compared to each other. To investigate the contribution of the smart operators independently, three combinations are designed as followings:

1. The original SATIBEA with Smart Mutation: **the original approach**
2. SATIBEA+ with Smart Mutation+ but without Smart Crossover: **the filtered SATIEBA+ approach**
3. SATIBEA+ with Smart Mutation+ and with Smart Crossover: **the SATIBEA+ approach**

For each combination, we run the algorithm 30 times and for each run with a given number of objective evaluations. The number of objective evaluations equals the execution times of crossover and mutation in jMetal. Then we reported the medium values of the metrics.

6.1 With smart replacement vs. without smart replacement

As described above, the smart replacement will add a valid configuration to the population with a probability of one percent. If it had a strong influence on the result, then it would "flood" the contribution of the other smart operators. Thus, we executed our experiment with and without smart replacement separately. Each run is executed with 25000 objective evaluations (default value set by jMetal). The results are recorded in Table 4 and 5. When interpreting the results, we make the following observations:

Model	Performance indicators	SATIBEA	Filtered SATIBEA+	SATIBEA+
Web Portal	HV	0.066	0.067	0.067
	PFS	24	27	27
	ET (ms)	12909	12938	12980.3
eCos	HV	0.244	0.236	0.227
	PFS	82	203	215
	ET (ms)	16161	15925	15869
FreeBSD	HV	0.257	0.254	0.258
	PFS	41	47	148
	ET (ms)	64558	67647	67639
Linux X86	HV	0.237	0.241	0.238
	PFS	40	112	151
	ET (ms)	247084	256192	257110

Table 4. Evaluation results *with* smart replacement (*25000* objective evaluations)

Model	Performance indicators	SATIBEA	Filtered SATIBEA+	SATIBEA+
Web Portal	HV	0.069	0.066	0.069
	PFS	24	26	26
	ET (ms)	13073	13809	13385
eCos	HV	0	0.203	0.209
	PFS	0	196	190
	ET (ms)	16971	17832	16184
FreeBSD	HV	0	0.255	0.254
	PFS	0	55	148
	ET (ms)	70034	66574	71213
Linux X86	HV	0.010	0.245	0.243
	PFS	0	138	177
	ET (ms)	224468	264858	270617

Table 5. Evaluation results *without* smart replacement (*25000* objective evaluations)

Answering RQ4 (compare Table 4 and Table 5): The result of the experiment is "glamorized" with smart replacement. Although

SATIBEA was not able to find any valid solutions for complex FMs (eCos, FreeBSD and Linux) without using smart replacement (Table 5), a couple of valid configurations could be found by using it (Table 4). A main reason is that more conflicting features can be adapted by the SAT-solver in consideration of connected features. In addition, no big difference in terms of HV can be seen by using smart replacement (Table 4). It is obvious that the result has been drastically affected by using smart replacement, because this operation adds valid solutions to the population periodically. Thus, we treated the result in Table 4 as "invalid" and only used the results of Table 5 for further analysis.

Answering RQ1 (compare column 3 with 5 in Table 5): SATIBEA+ outperforms SATIBEA in terms of HV and PFS significantly, particularly for the middle-complex (eCos and FreeBSD) and high-complex FMs (Linux X86). For the simple FM (Web Portal), the difference is less notable. In addition, it is notable that for the complex FMs (eCos, FreeBSD and Linux), no valid configurations could be found by using SATIBEA. In contrast, SATIBEA+ could find many valid configurations for them.

Answering RQ2 (compare column 3 with 4 in Table 5): Filtered SATIBEA+ outperforms SATIBEA in terms of HV and PFS significantly, particularly for the middle-complex and high-complex FMs. For the simple FM, the difference is less notable. Fitlered SATIBEA+ found also valid configurations for the complex FMs by using the enhanced smart mutation.

Answering RQ3 (compare column 4 with 5 in Table 5): SATIBEA+ outperforms filtered SATIBEA+ in terms of PFS significantly, particularly for the middle-complex and high-complex FM (except for the FM of FreeBSD). For the simple FM, the difference is less notable. In addition, there is no remarkable performance improvement in terms of HV by using smart crossover.

Answering RQ5 (compare ETs in Table 5): For each feature model, there is no significant differences in terms of ET. The execution time of all algorithms are in a comparable range.

Other findings: For the simple feature model, there is no notable differences between SATIBEA, filtered SATIBEA+ and SATIBEA+ for all performance indicators. Thus, this feature model was not considered in further experiments.

6.2 Further Runs

In order to analyze the development of performance metrics by increased objective evaluations (also by increased execution times), we performed further runs with 12500 and 50000 objective evaluations on the 4 FMs separately. Because smart replacement changed the results too radically, it was not applied by the further executions. Considering the results in Table 5 and Table 6, we make the following observations:

- PFSs get improved significantly by the increased times of objective evaluations. One exception is the feature model FreeBSD. It generates less valid solutions with more objective evaluations.
- HVs also get improved by the increased times of objective evaluations, but slightly.
- ETs are proportional to the times of objective evaluations.
- For the feature model eCos, SATIBEA+ performs a little bit worse in terms of PFS than filtered SATIBEA+.

7 DISCUSSION

Comparison of SATIBEA+, filtered SATIBEA+, and SATIBEA In this section, we reason about our findings and discuss their implications. First of all, we ask why SATIBEA+ performs much better than SATIBEA? The performance improvement is essentially achieved by the smart+ operators used by SATIBEA+. Firstly, the smart+ mutation beats the native smart mutation in repairing an invalid configuration. The reason for that is that we "relax" the scope of the mutation. In SATIBEA+, a feature and its related (by constraints) features will be considered as a whole mutable-able unit. Thus, the SAT-solver searches for a valid solution in an expended range. Secondly, the smart+ crossover repairs an invalid configuration much better than the standard 1-point crossover, because it follows the motto "learn from the best". In SATIBEA+, a configuration will get repaired (at least partly) by learning "good" feature assignments from a valid configuration. Thus, the number of feature violations of a configuration will be reduced and finally more valid configurations will be produced. To sum up, we embed the specific problem information (feature configuration) into an evolutionary algorithm, together with a problem-dependent local search method (with smart+ operators); therefore, the performance of the algorithm will be improved according to the No Free Lunch theorem [18].

Model	Objective evaluations	Performance indicators	SATIBEA	Filtered SATIBEA+	SATIBEA+
eCos	12500	HV	0	0.200	0.199
		PFS	0	158	141
		ET (ms)	8113	8393	8358
	50000	HV	0	0.231	0.194
		PFS	0	228	209
		ET (ms)	32281	31768	31894
FreeBSD	12500	HV	0	0.248	0.248
		PFS	0	83	135
		ET (ms)	34887	36768	41682
	50000	HV	0	0.260	0.267
		PFS	0	28	135
		ET (ms)	132204	134914	134129
Linux X86	12500	HV	0	0.242	0.241
		PFS	0	122	126
		ET (ms)	116686	144918	145726
	50000	HV	0	0.247	0.244
		PFS	0	138	183
		ET (ms)	457070	492015	451761

Table 6. Evaluation results *without* smart replacement (*12500* and *50000* objective evaluations)

The results of experiments reveals also some abnormality. One is for the feature model eCos, less valid configurations could be found by SATIBEA+ than by filtered SATIBEA+. The other is for the feature model FreeBSD, the performance indicator PFS could not get converged with increased number of runs. It indicates that the performance is sensible to the characteristics of a feature model (such as the number of cross-tree constraints). We are planing to investigate this phenomena in the future work.

Threats to Validity A prerequisite for applying optimization technologies, such as SATIBEA+ is that the objectives have to be calculated from the feature attributes. For attributes such as *cost* this is given by the trivial sum, however, other attributes might be more complex to evaluate. The test set for the Web-Portal is a theoretical one, i.e., the attributes and their values are set randomly, however, this will not influence our experimental results. Moreover, the execution time of our experiments are limited to 10 minutes, because SATIBEA+ could already outperform the original algorithm SATIBEA during this short time.

More interesting is the choice of the connected features (see Section 3). In our case, we only select the directly connected features (i.e., depth 1). Hence, in very complex domains it might be the case that no solution will be found. In future work, we will investigate in features connected indirectly through multiple constraints.

8 RELATED WORK

A key challenge in the Software Product Line community is to determine how to select a set of features from a feature model, which not only satisfies feature constraints but also optimize the different objectives of customers. [17] uses Filtered Cartesian Flattening to calculate the optimal feature sets subject to the given resource constraints. They transform a feature selection problem as a Multi-Dimensional, Multiple-Choice (MDMC) knapsack problem and introduce a heuristic to filter choices.

Using the task planning technique HTN (Hierarchical Task Network), [13] proposes a framework to select suitable features that satisfy both the stakeholders' functional requirements (FRs) and non-functional requirements (NFRs). For optimization, they aggregate qualitative and quantitative properties into a single object value. [4] presents a genetic algorithm GAFES to optimize feature selections in the face of resource constraints. In the fitness function, they use weighting to reflect the different importance of resources. However, it is not trivial to find a proper utility function to change a multi-criteria problem to a single-criterion problem. Moreover, only one "best" solution will be calculated, while a set of solutions (Pareto set) is expected for multi-criteria problems.

[12] shows that Indicator-Based Evolutionary Algorithm (IBEA) works better than other dominance-based EAs in solving feature selection problems for large models and many optimization objectives. [9] propose a multi-objective evolutionary algorithm IVEA to optimize the selection of features with FRs and NFRs. They treat constraint violations in a feature model as a separated dimension besides the optimization of NFRs. They define a violation-dominance function to guide the environment selection and mating selection. However, only the standard crossover and mutation operators are applied in these approaches and therefore the performance improvements are limited.

[15] incorporate feedback-directed mechanism into the EA for multi-objective feature selection problem. Similar as the smart crossover of SATIBEA+, an invalid configuration gets repaired by copying all of the non-error features from an another configuration. The difference is that our approach is mini-invasive which copies only a minimal set of feature assignments (only for "bad" features). For configuration optimization, [10] propose a two tasks approach. Firstly, a rough approximation of Pareto front is searched and presented to the user. Then the user indicates the areas which he interested in. In [6], optimal products are chosen from a feature model by using SIP, a two steps multi-objective evolutionary algorithm. They concentrate primarly on the number of constraints that hold and then on the other objectives.

9 CONCLUSION AND FUTURE WORK

In this work, we have demonstrated that our approach SATIBEA+, for the multi-objective feature selection problem, outperforms the state-of-the-art algorithm SATIBEA [5] in the quality and amount of found configurations. In addition, we have also show that our approach scales to the complex feature models with thousands of features and constraints. Still, the question may arise regarding the effectiveness of SATIBEA+ in consideration of other characteristics of feature models (such as number of cross-tree constraints, which may be the subject of further investigation). Other directions for future work regarding optimal feature selection problems could be:
1. Comparison of performance of constraint-based approaches and constraint-first approaches [6].
2. Incorporate customer requirements during the search in all phases of the optimization process.
3. Further experiments with complex feature models with real attribute values.

REFERENCES

[1] Daniel Le Berre and Anne Parrain, 'The sat4j library, release 2.2, system description', *Journal on Satisfiability, Boolean Modeling and Computation*, **7**(2010), 59–64, (2010).

[2] K. Czarnecki, S. Helsen, and U. Eisenecker, 'Formalizing Cardinality-based Feature Models and their Specialization', *Software Process: Improvement and Practice*, **10**(1), 7–29, (2005).

[3] Juan J Durillo and Antonio J Nebro, 'jMetal: A Java framework for multi-objective optimization', *Advances in Engineering Software*, **42**(10), 760–771, (October 2011).

[4] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang, 'A genetic algorithm for optimized feature selection with resource constraints in software product lines', *Journal of Systems and Software*, **84**(12), 2208–2221, (December 2011).

[5] Christopher Henard and Mike Papadakis, 'Combining multi-objective search and constraint solving for configuring large software product lines', in *ICSE' 15 Proceedings of the 37th International Conference on Software Engineering*, pp. 517–528, (2015).

[6] Robert M Hierons, Miqing Li, Xiaohui Liu, Sergio Segura, and Wei Zheng, 'SIP: Optimal Product Selection from Feature Models Using Many-Objective Evolutionary Optimization', *ACM Transactions on Software Engineering and Methodology*, **25**(2), 1–39, (April 2016).

[7] L. Hotz, A. Felfernig, M. Stumptner, A. Ryabokon, C. Bagley, and K. Wolter, 'Configuration Knowledge Representation & Reasoning', in *Knowledge-based Configuration – From Research to Business Cases*, eds., A. Felfernig, L. Hotz, C. Bagley, and J. Tiihonen, chapter 6, 59–96, Morgan Kaufmann Publishers, (2013).

[8] K C Kang, J Lee, and P Donohoe, 'Feature-Oriented Product Line Engineering', *IEEE Software*, **19**(4), 58–65, (2002).

[9] Xiaoli Lian and Li Zhang, 'Optimized feature selection towards functional and non-functional requirements in Software Product Lines', in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 191–200. IEEE, (March 2015).

[10] Paul Pitiot, Michel Aldanondo, Elise Vareilles, Thierry Coudert, and Paul Gaborit, 'Improving configuration and planning optimization: Towards a two tasks approach ', in 15^{th} *International Configuration Workshop*, eds., M. Aldanondo and A. Falkner, pp. 35–40, Vienna, Austria, (2013).

[11] *Handbook of Constraint Programming*, eds., F. Rossi, P. van Beek, and T. Walsh, Elsevier, 2006.

[12] AS 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 '13 Proceedings of the 2013 International Conference on Software Engineering*, pp. 492–501, (2013).

[13] Samaneh Soltani, Mohsen Asadi, and D Gašević, 'Automated planning for feature model configuration based on functional and non-functional requirements', in *SPLC'12*, pp. 56–65, (2012).

[14] El-Ghazali Talbi, *Metaheuristics: From Design to Implementation*, John Wiley & Sons, Inc., Hoboken, NJ, USA, June 2009.

[15] Tian Huat Tan, Yinxing Xue, Manman Chen, Jun Sun, Yang Liu, and Jin Song Dong, 'Optimizing selection of competing features via feedback-directed evolutionary algorithms', in *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSTA 2015*, pp. 246–256, New York, New York, USA, (2015). ACM Press.

[16] Edward Tsang, *Foundations of Constraint Satisfaction*, Academic Press, London, San Diego, New York, 1993.

[17] Jules White, Brian Dougherty, and Douglas C. Schmidt, 'Selecting highly optimal architectural feature sets with Filtered Cartesian Flattening', *Journal of Systems and Software*, **82**(8), 1268–1284, (August 2009).

[18] Xinjie Yu and Mitsuo Gen, *Introduction to Evolutionary Algorithms*, Decision Engineering, Springer London, London, 2010.