Noname manuscript No.
(will be inserted by the editor)

Reparation in Evolutionary Algorithms for Multi-objective Feature Selection in Large Software Product Lines

Takfarinas Saber * · David Brevet ·
Goetz Botterweck · Anthony Ventresque

Received: date / Accepted: date

Abstract *Purpose:* Software Product Lines Engineering is the area of software engineering that aims to systematise the modelling, creation and improvement of groups of interconnected software systems by formally expressing possible alternative products in the form of Feature Models. Deriving a software product/system from a feature model is called Feature Configuration. Engineers select the subset of features (software components) from a feature model that suits their needs, while respecting the underlying relationships/constraints of the system–which is challenging on its own. Since there exist several (and often antagonistic) perspectives on which the quality of software could be assessed, the problem is even more challenging as it becomes a multi-objective optimisation problem. Current multi-objective feature selection in software product line approaches (e.g., SATIBEA) combine the scalability of a genetic algorithm (IBEA) with a solution reparation approach based on a SAT solver or one of its derivatives.

Methods: In this paper, we propose MILPIBEA, a novel hybrid algorithm which combines IBEA with the accuracy of a mixed-integer linear programming (MILP) reparation.

Results: We show that the MILP reparation modifies fewer features from the original infeasible solutions than the SAT reparation and in a shorter time. We also demonstrate that MILPIBEA outperforms SATIBEA on average on various multi-

T. Saber
Lero, School of Computer Science, University College Dublin, Ireland
E-mail: takfarinas.saber@ucd.ie,

A. Ventresque
Lero, School of Computer Science, University College Dublin, Ireland
E-mail: anthony.ventresque@ucd.ie,

D. Brevet
Monogramm, Clermont-Ferrand, France
E-mail: david.brevet@monogramm.io

G. Botterweck
Lero, School of Computer Science and Statistics, Trinity College Dublin, Ireland
E-mail: goetz.botterweck@tcd.ie

* Corresponding Author

objective performance metrics, especially on the largest feature models. The other major challenge in software engineering in general and in software product lines, in particular, is evolution. While the change in software components is common in the software engineering industry, the particular case of multi-objective optimisation of evolving software product lines is not well-tackled yet. We show that MILPIBEA is not only able to better take advantage of the evolution than SATIBEA, but it is also the one that continues to improve the quality of the solutions when SATIBEA stagnates.

Conclusion: Overall, IBEA performs better when combined with MILP instead of SAT reparation when optimising the multi-objective feature selection in large and evolving software product lines.

Keywords Software Product Line, Feature Selection, Multi-Objective Optimisation, Evolutionary Algorithm, Reparation, Mixed-Integer Linear Programming.

1 Introduction

Software Engineering brings multiple domains together to help with the process of modelling, building, monitoring, testing and debugging software systems [1] and Software Product Lines (SPL) is one of them. SPL deals with related software systems as sets rather than considering each of them separately [2]. This strategy simplifies software reuse [3], permits better reliability, and drives important cost reductions [4], thus allowing SPL to attract more interest from the software engineering industry in recent years.

Software Product Lines with their respective products and characteristics are prevailingly represented using Feature Models (FMs). In an FM, every feature represents a component of a software product that might be interesting to some company or customer. Every FM depicts a list of all possible feature configurations, thus the FM describes the set of all possible software products. In real-life software product lines, FMs can expand in size (number of features and their relationships) to become very large (e.g., in our paper, we deal with an FM which has $\sim$7k features and $\sim$350k constraints).

To configure/create a specific product out of the product line, we need to operate a *feature selection*. Given that an SPL depicts multiple software products, we need to *optimise the feature selection* to find the best possible product. In other words, we need to select the features that could be combined to make the 'best' software product [5]. Given that there are different perspectives on what constitutes a good software product, different objectives have to be considered at the same time (e.g., technical feasibility, reliability, feature cost). Therefore, searching for the 'best' feature selection is often represented as a *multi-objective optimisation problem* [6].

One of the best performing algorithms to address the *multi-objective feature selection in SPL* is SATIBEA [6]: a hybrid algorithm that combines an Indicator-Based Evolutionary Algorithm (IBEA) and a SAT solver as a solution reparation procedure in its mutation operator. IBEA faces a difficult challenge: the search space is so large and constrained that mutation and crossover operators generate a large number of infeasible solutions. SATIBEA uses a SAT solver to fix infeasible solutions and obtains a number of viable solutions at each generation of the genetic algorithm. While SATIBEA achieves large performance improvements, we have shown in our previous work that SATIBEA's improvement is slow and plateaus/stagnates after some duration [7]. SATIBEA's process has two major issues: (i) we have shown in an empirical study that SATIBEA is time-consuming and that the vast majority of its execution time is spent on fixing faulty solutions; and (ii) SATIBEA also defies the inheritance property from parents to their offsprings as it substantially modifies the faulty solutions.

In this paper, we extend our previous work [7] as we thoroughly investigate our proposed MILPIBEA approach; a hybrid algorithm that uses a genetic algorithm (IBEA) in conjunction with a mixed-integer linear programming (MILP) solver (IBM ILOG CPLEX) to repair infeasible solutions. We show in our evaluation that the MILP reparation is not only faster but also modifies fewer features in the original infeasible solutions than the SAT reparation. In other words, MILPIBEA's reparation is both more efficient and more effective at ensuring that the repaired solutions are closer to the ones generated by IBEA's operators. Furthermore, we have shown that when combining MILP with the IBEA algorithm, MILPIBEA outperforms SATIBEA on large SPLs, while achieving this performance in only a fraction of the time taken by SATIBEA.

In our paper, we also investigate a problem that is related to the multi-objective feature selection (i.e., multi-objective feature selection with evolving FMs). This problem is not studied to its full extent in the literature. Software products/libraries are subject to continuous evolution due to a perpetual shift in customer preferences in terms of software requirements. Software evolution materialises as variations between different FM versions. For instance, we have previously analysed a large FM representing the Linux kernel and showed that it evolves unceasingly [8]. Particularly, we have found differences which can go up to 7% between successive releases of the Linux kernel (every few months).

When performing optimisation of feature selection in evolving FMs, we propose to exploit the evolution context to improve the search process. We believe that it is unreasonable to create entirely random bootstrapping populations for the search process when there exists a set of well-performing solutions for a relatively close problem. Furthermore, we believe that it might be valuable to take advantage of the fact that configurations generated before the FM has evolved are similar enough and could be repaired.

In this paper, we also show that our approach, MILPIBEA, which was initially designed to address the problem of multi-objective feature selection in SPLs, is also better than SATIBEA when the FMs evolve.

This paper makes the following contributions:

- We describe our novel MILP based solution reparation which outperforms SAT [6] in terms of both execution time and 'fidelity' of the reparation to the original infeasible solutions.
- We thoroughly evaluate SATIBEA and MILPIBEA on non-evolving SPL problems and show that MILPIBEA is better than SATIBEA on average on various multi-objective performance metrics, especially for the largest feature models.
- We also evaluate SATIBEA and MILPIBEA on evolving SPL problems and show that not only MILPIBEA achieves better performance results, but it also

continues to improve the quality of solutions while SATIBEA stagnates after a certain duration.

The remainder of this paper is organised as follows: Section 2 describes the background that helps with the understanding of our study and its related work. Section 3 details our MILP reparation approach. Section 4 describes the experimental setup for validating our approach. Section 5 compares the performance of our MILPIBEA approach and SATIBEA on various performance metrics when dealing with large benchmark feature models, whereas Section 6 compares MILP-IBEA and SATIBEA when dealing with the evolutions of the largest existing feature model. Finally, Section 7 concludes the paper.

2 Background and Related Work

In this section, we present the material which forms the background and the related work of our research in four parts:

– Software Product Line Engineering: representation of software variations in terms of feature model configurations.
– Multi-Objective Optimisation (MOO): different feature selections lead to different software products which can be viewed from different (often antagonistic) quality perspectives. MOO provides us with a framework to deal with these types of problems.
– Solution Reparation: a review of existing reparation approaches in real-world optimisation problems in general and in multi-objective feature selection in SPL in particular.
– SATIBEA: a state-of-the-art algorithm to optimise the multi-objective feature selection with both non-evolving [6] and evolving [8] feature models.

2.1 Software Product Line Engineering

It is common that software engineers modify software artefacts to match the requirements of a specific client. Software Product Line Engineering is the domain that attempts to deal with those adaptations in a more systematic way. For example, it is possible to interpret all software artefacts (and their adaptations) as a set of features. These features can then be selected and put together to form a particular software product.

Feature Models can be represented as a set of features and connecting relationships (constraints). Figure 1 shows a motivating example of an FM with ten features which are connected using multiple relationships. For example, each $\boxed{\text{Screen}}$ must be of one and only one type, i.e., $\boxed{\text{Basic}}$, $\boxed{\text{Colour}}$ or $\boxed{\text{High Resolution}}$. When configuring a software product from the software product line, we need to select a subset of features $\mathcal{S} \subseteq \mathcal{F}$ which satisfies constraints of the FM $\mathcal{F}$ (i.e., technical requirements and choices of the stakeholders/customers).

The feature configuration task can be modelled as a satisfiability problem (SAT) in a conjunctive normal form (CNF). For instance, in Figure 1 the FM would have the following clauses, among others: (Basic $\vee$ Colour $\vee$ High Resolution) $\wedge$

(¬Basic ∨ ¬Colour)∧(¬Basic ∨ ¬High Resolution)∧(¬Colour ∨ ¬High Resolution), which describe the choice between the three screen alternatives.

Then, the problem becomes an instantiation of variables (in our case, features with the values True or False) in a way that ensures the satisfaction of all the constraints. Let f_i be a binary variable for every feature $F_i \in \mathcal{F}$, and f_i is set to True if F_i is selected to be part of $\mathcal{S}$ and False otherwise.

When deriving a software product from a feature model, software engineers and designers attempt to discover products optimising different objectives instead of limiting themselves to discovering a unique product that satisfies the FM.

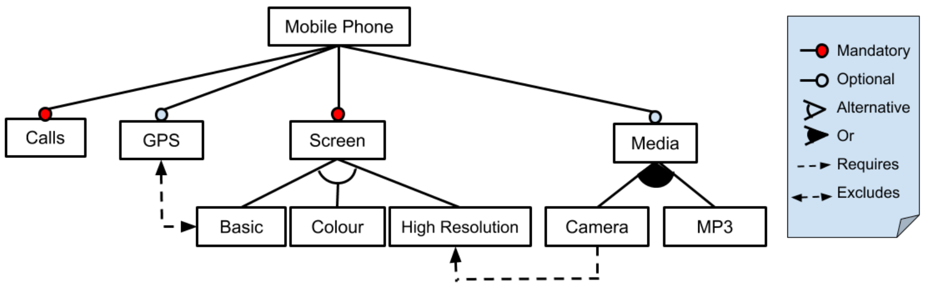


Fig. 1 Example of a Feature Model

Product lines correspond to investments in the long term [9]. Therefore, it is crucial to efficiently deal with the evolution of SPLs (and their representative FMs). For instance, we showed in a previous study of a large-scale FM (i.e., the Linux Kernel) that every few months a new FM is released with up to 7% modifications among the features (features added or removed) [8].

2.2 Multi-Objective Optimisation

Multi-Objective Optimisation (MOO) involves the simultaneous optimisation of more than one (often antagonistic) objective function. Since the quality of different software artefacts can be seen from different perspectives (e.g., reliability, usefulness, cost), feature selection in SPL is better modelled as a multi-objective optimisation problem [6].

The solutions obtained from a MOO problem correspond to a set of non-dominated solutions, which could be defined as follows: Let S be the set of all feasible solutions for a given FM. For all $x \in S$, $O = [O_1(x), ..., O_k(x)]$ is the vector containing the k objective values for the solution x. It is said that a solution x_1 dominates another solution x_2 (also written as $x_1 \succ x_2$), if and only if $\forall i \in \{1, ..., k\}$, $O_i(x_1) \leq O_i(x_2)$ and $\exists i \in \{1, ..., k\}$ such that $O_i(x_1) < O_i(x_2)$. We also say that x_i is a non-dominated solution if there is no other solution x_j that dominates x_i.

The set of all non-dominated solutions form what is called a Pareto front: in this set, it is impossible to find any solution better in all objectives than any of the other solutions in the set. For instance, the Pareto front given in Figure 2 contains the solutions x_1, x_2, x_4, x_6, and x_7 because they are not dominated by any other solution. On the other hand, the solutions x_3 and x_5 are dominated by the solution x_6, whereas the solutions x_8, x_9, and x_{10} are dominated by the solution x_2. Therefore, they are left out of the Pareto front.

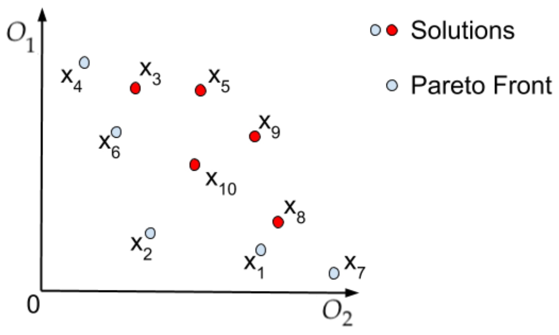


Fig. 2 Example of a Pareto front with two minimisation objectives.

2.3 Solution Reparation

Real-life engineering/optimisation problems such as feature selection in SPL usually represent large sets of constraints, which makes the task of finding a feasible solution a challenge on its own for evolutionary algorithms. To cope with that, several works have proposed different solution reparation techniques to handle constraints in evolutionary algorithms [10].

Real-life engineering/optimisation problems often have their best solutions lying close to the constraint boundaries and are therefore hard to reach/find when only relying on feasible solutions during the search process [11]. Some works investigated the use of infeasible solutions in evolutionary algorithms as a mean to improve the performance of their algorithms [12], while at the same time putting adaptive schemes to penalise them [13].

In the context of feature selection in software product lines, Sayyad et al. [14] were the first to propose defining the number of violated constraints as a minimisation objective and showed it to be an effective way to improving the search process. The definition of Sayyad et al. [14] has since been embraced by the SPL community with several works using it as their benchmark model (e.g., [6, 7, 15]). This is also the definition considered in this work.

2.4 Solution Reparation for Feature Selection in SPL

Most recent approaches to feature selection in software product lines combine exact approaches as a means to find/generate feasible solutions. Xue and Li [16] model the problem as a multi-objective linear program with binary variables and propose multi-objective integer programming approaches to solve the problem. However, their approach was only capable of handling small-scale feature models.

SATIBEA [6] built upon the work of Sayyad et al. on the multi-objective feature selection in SPL and introduced a SAT solver as a mean to repair solutions. Their approach brought an important improvement to the search process. Some works also replaced the SAT solver with some of its variations (e.g., [15]).

SATIBEA [6] is an extension of the Indicator-Based Evolutionary Algorithm (IBEA) which guides the search by a quality indicator given by the user. Prior to SATIBEA, several techniques have been tried to solve the multi-objective feature selection in SPL. As most of the random techniques and genetic algorithms tend to generate invalid solutions (given the large and constrained search space), any random, mutation or crossover operation is tricky. Therefore, setting the number of violated constraints as a minimisation objective has been proposed by Sayyad et al. [17] and has since been widely used in the literature [6, 8, 15].

SATIBEA has been introduced to help IBEA find valid products using a SAT solver. SATIBEA changes the mutation process of IBEA: when a solution is mutated, three different mutations can be applied:

1. The standard bit-flip mutation proposed by IBEA.
2. Replacing the solution by another one generated by the SAT solver that does not violate any constraints.
3. Transforming the solution into a valid one using the SAT solver (i.e., solution reparation).

Using this novel mutation approach, SATIBEA finds better solutions than IBEA: it finds valid optimised products but also gives better values in quality metrics.

In a previous publication [7], we have proposed modelling the solution reparation in the context of evolving SPLs as a MILP problem and used a MILP solver to optimise it. In this work, we are extending this work by applying the reparation process to the general case of multi-objective feature selection in SPLs (with and without evolution). In this paper, we show that our novel technique addresses some of SATIBEA's limitations (i.e., slow and stagnating performance improvements) through the reparation of solutions with the least amount of changes.

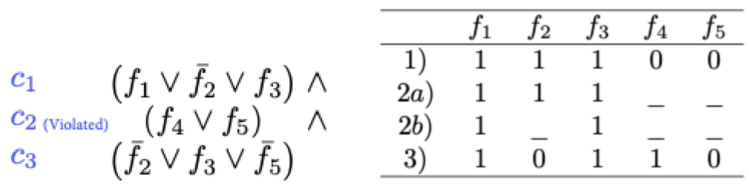
3 Proposed Approach: MILPIBEA

In this section, we present two ways of repairing non-feasible solutions: SAT and MILP. Non-feasible solutions appear very often during the execution of the genetic algorithm on our problem. Indeed, due to the size of the search space and the number of constraints that can be violated, both mutation and crossover (the basic operators in IBEA) generate quite a large ratio of infeasible solutions.
The first approach we present is the SAT reparation proposed in the definition of SATIBEA [6]. The second approach we present is our own improved reparation using the MILP model and resolution.

3.1 Solution Reparation in SATIBEA

SATIBEA's reparation method occurs in the mutation phase of the genetic algorithm. IBEA takes a solution that violates one or several constraints out of the population and corrects it using a SAT solver. This leads the solution to become valid (no longer violating constraints). Figure 3 shows an example of SATIBEA's reparation technique on a toy FM which contains 5 features (f_1 through f_5) and 3 constraints (c_1 through c_3). The constraints are shown on the left-hand side of Figure 3, with c_2 marked as violated.

1) First, a solution with assignment {*1 1 1 0 0*} is selected for repair due to the violation of the constraint c_2 (which causes the solution to be invalid). This is shown in row 1 in the table on the right-hand side of Figure 3.
2a) Second, SATIBEA unsets (this is represented by '_' in the example) all the bits that belong to a violated constraint. Here, constraint c_2 is violated, so f_4 and f_5 are unset. This is shown in row $2a$ of the table.
2b) Third, SATIBEA unsets all the bits that are evaluated as False in every constraint. Each of these can either be a feature without a negation sign in the constraint (i.e., f) that is set to False or a feature with a negation (i.e., $\overline{f}$) that is set to True. All of these are unset. In our example, f_2 is assigned to True and is evaluated at False in the constraint c_1 ($\overline{f}_2$). Therefore, SATIBEA unsets f_2. This is shown in row $2b$ of the table.
3) Eventually, the resulting partial assignment is given to the SAT solver to complete the unset values while satisfying the constraints of the FM. SATIBEA's reparation always obtains a valid solution if it exists. In our case, SATIBEA results in a new solution (i.e., {*1 0 1 1 0*}). This is shown in row 3 of the table in Figure 3. Note that this procedure cannot guarantee to always return a valid solution as the problem may be unsatisfiable.



c_1		$(f_1 \vee \bar{f}_2 \vee f_3) \wedge$
c_2	(Violated)	$(f_4 \vee f_5) \wedge$
c_3		$(\bar{f}_2 \vee f_3 \vee \bar{f}_5)$

	f_1	f_2	f_3	f_4	f_5
1)	1	1	1	0	0
2a)	1	1	1	_	_
2b)	1	_	1	_	_
3)	1	0	1	1	0

Fig. 3 Reparation of a solution in SATIBEA. The original solution, violating constraint c_2, is shown on row 1 of the right-hand side table and the different steps of SATIBEA's reparation are shown on rows $2a$, $2b$ and 3 of the same table.

Although this reparation technique is fast and improves the classical IBEA algorithm, the number of flipped bits is large. This often creates new solutions that are far from the original ones (before the reparation). The issue is that modifying solutions obtained through mutation in IBEA too much is against the idea behind genetic algorithms (i.e., inheriting and preserving good characters between parents and their offsprings). For instance, from the solution {*1 1 1 0 0*} (row 1 of Figure 3) that violates the constraints, it would be better to obtain solution {*1 1 1 1 0*} that does not violate the constraints (instead of {*1 0 1 1 0*}). The next subsection describes our MILP-based reparation technique that overcomes this problem.

3.2 MILP Model for Solution Reparation

Our new method repairs solutions and avoids the problem described in the previous section (i.e., a large number of flipped bits between the original infeasible solutions and the repaired ones). This method repairs the faulty solutions and minimises the number of flipped bits.

Applied to the example in Figure 3, only features f_4 and f_5 are unset. CPLEX, a well-known mixed-integer linear programming solver, solves the problem of finding a valid solution by assigning values to f_4 and f_5 while at the same time, minimising the total bit flips on the rest of the features (i.e., f_1, f_2 and f_3). One possible output is {*1 1 1 1 0*} which does not modify any fixed bit, unlike SATIBEA's one (i.e., {*1 0 1 1 0*} which has one modification on the feature f_2).

Using our method, CPLEX is guaranteed to find a valid solution. Moreover, it returns a solution that is as close as possible to the original one. In our method, we use the model defined by Equations 1a, 1b, and 1c.

$$\text{Minimise} \qquad \sum_{x \in T} (1 - x) + \sum_{x \in F} x \tag{1a}$$

$$\text{Subject to} \qquad \sum_{x \in P_i} x + \sum_{x \in N_i} (1 - x) \geq 1, \quad \forall i \in \{1, .. n\} \tag{1b}$$

$$x \in \{0, 1\}, \quad \forall x \in X \tag{1c}$$

With n number of clauses, X set of features, $T \subset X$ set a features fixed at True, $F \subset X$ set of features fixed at False, $P_i \subset X$ set of features without negation in clause i, and $N_i \subset X$ set of features with negation in clause i.

In the MILP model above, we aim to minimise the number of flipped features in the original solution: if the feature was originally at True (i.e., '1'), then we count it as a modification if and only if it changes to False (i.e., '0'). Similarly, when the feature was originally at False and is changed to True, we also count it as a modification. Similarly to the SAT reparation, each clause is represented by a linear constraint. Every feature without negation is considered as '1' when selected, and every feature that is negated is considered as '1' when unselected. The sum of every feature within a clause has to be larger or equal to 1 to validate it.

4 Experimental Set-up

This section presents the different elements that we have used in our implementation: the dataset, the objectives we use for our multi-objective optimisation problem, the metrics we use to evaluate our approaches, the parameters we use for the genetic algorithm (i.e., IBEA) and the hardware configuration.

4.1 Dataset

For our experiments, we use the largest open-source FM we could find [15] (i.e., the Linux Kernel). The FM of the Linux Kernel is publicly available in the Linux Variability Analysis Tools (LVAT) repository and it contains 6,888 features and 343,944 constraints in its version 2.6.28.6.

FMs in the LVAT repository are not in the form of a SAT problem and they are not directly usable in our approach. These FMs are in Kconfig model extracts

(.exconfig) and need to be converted into SAT models [18] using VM2BOOL[1]. This conversion enables SATIBEA to process the FM and search for the non-dominated products in the many-objective search space.

Note that in addition to variables representing the features, VM2BOOL introduces new variables into the problem. Doing so enables VM2BOOL to (i) convert the most complex constraints in the FM and (ii) avoid the explosion in the size of the propositions.

Besides the Linux Kernel, we use an additional four major FMs that are widely used in the literature [15] and that cover large real-world FMs:

– Fiasco: a micro-kernel that can be used to construct flexible systems (e.g., Unix operating systems). Fiasco is suitable to both big/complex systems and for small/embedded applications.
– FreeBSD: a free and open-source Linux-like operating system used to power present-day servers, desktops, and embedded applications for its advanced networking, security, and storage features.
– μClinux (pronounced you-see-Linux): a variation of the Linux kernel targeted towards micro-controllers without memory management units.
– eCos (Embedded Configurable Operating System): a free and open-source real-time customisable operating system designed for embedded systems and applications that require a unique multi-thread process.

Table 1 shows the version and the size of each of the FMs that we consider in our experiments in non-evolving feature models (i.e., in Section 5. The table also reports the number of features and the size of the SAT problem (in terms of number of variables and number of clauses). Similarly to the SATIBEA paper [6], we set the execution time on the Linux Kernel to 1,200s. For the other datasets, we use smaller execution times based on the convergence time of SATIBEA [8].

Table 1 Versions and characteristics of the feature models used in our experiments in non-evolving Feature models (i.e., in section 5. The characteristics include the number of features and size of the SAT problem in terms of number of variables and number of clauses. This table also reports the execution time budget that is permitted on every feature model.

Feature Model	Version	#Features	#Variables	#Clauses	Time (s)
Linux kernel	2.6.28.6	5,701	6,888	343,944	1,200
Fiasco	2011081207	300	1,638	5,228	200
FreeBSD	8.0.0	1,396	1,396	62,183	200
μClinux	3.0	616	1,850	2,468	100
eCos	20100825	1,244	1,244	3,146	100

4.2 Optimisation Objectives

In our work, we consider five optimisation objectives that are widely used in the literature [6, 7, 15]:

1. *Correctness*: we minimise the number of violated constraints, as proposed by Sayyad et al. [17].

[1] https://bitbucket.org/tberger/vm2bool

2. *Richness of features*: we maximise the number of selected features to have products with more functionality.
3. *Features used before*: we minimise the number of selected features that were not used before as they might have unknown issues or use technologies that are uncommon to most software engineers.
4. *Known defects*: we minimise the number of known defects in selected features (randomly generated integer value for each feature between 0 and 10).
5. *Cost*: we minimise the cost of the selected features (randomly generated real value for each feature between 5.0 and 15.0).

While we are using the five aforementioned objectives, these objectives could easily be substituted or augmented with other ones, e.g., consumption of resources or various costs. All the approaches used in this work (evolutionary algorithm, SAT reparation, and MILP reparation) are agnostic of the type and number of objectives.

4.3 Many-Objective Performance Metrics

To assess the performance of our algorithms we use four many-objective performance metrics: two quality metrics (Hypervolume and Inverted Generation Distance) and two diversity metrics (Spread and Pareto Front Size).

4.3.1 Hypervolume (HV)

The idea behind the hypervolume is that it computes the volume (measured in the k dimensions of the problem's search space) that is dominated by the Pareto front. The hypervolume is the area between the non-dominated solutions and the reference point. The reference point represents the worst possible value for each objective. The obtained measure represents the region covered by our approximate Pareto front: the higher the better.

More formally, hypervolume is defined as follows: Let A be the set of solutions/points in the Pareto front, and $r[r_1, ..., r_k]$ is a reference point far from it. Then, hypervolume of A is defined by:

$$HV(A, r) = \lambda(\bigcup_{s \in A} [O_1(s), r_1] \times ... \times [O_k(s), r_k]) \tag{2}$$

where: λ is the Lebesgue measure, k is the number of objectives and $[O_1(s), r_1] \times ... \times [O_k(s), r_k]$ is the k-dimensional hyper-cuboid comprised of all solutions/points that are weakly dominated by s but that are not weakly dominated by the reference point.

4.3.2 Inverted Generation Distance (IGD)

This metric evaluates the average of distances $d(s, A)$ between every solution s in the reference front R and its closest solution in the Pareto front A. This metric is the reverse of the Generational Distance. The lower the IGD the better the Pareto front. Inverted Generation Distance of A and R is defined as:

$$IGD(A, R) = \frac{\sum_{s \in R} d(s, A)}{|R|} \tag{3}$$

4.3.3 Spread (S)

This metric computes the solutions' distribution to evaluate their extent spread in the Pareto front. The higher the spread the better the Pareto front (i.e., the more diverse the Pareto front). For a set of solutions A, consider d_a as the distance between every solution $s_a \in A$ and its closest neighbour $s_b \in A$, and $s_i \in A$ and $s_j \in A$ are the two furthest solutions in A. Spread of A is defined as:

$$S(A) = \frac{d_i + d_j + \sum_{a \in \{1..|A|-1\}} (d_a - d_{avg})}{d_i + d_j + d_{avg} \cdot (|A| - 1)} \tag{4}$$

with $d_{avg} = \frac{\sum_{a \in \{1..|A|\}} d_i}{|A|}$

4.3.4 Pareto Front Size (PFS)

This metric is the simplest to evaluate among the others as it counts the number of non-dominated solutions in the population at every generation. The higher the PFS the better.

4.4 System and Algorithms Set-up

We use the source code provided by SATIBEA's authors and make MILPIBEA publicly available at https://github.com/takfarinassaber/MILPIBEA. The tests are performed on a machine with 64GB of RAM and 12 core Intel(R) Xeon(R) 2.20GHz CPU. We use the following parameters for our genetic algorithm:

– Population size: 300 solutions.
– Offspring population size: 300 solutions.
– Crossover rate: 0.8. Represents the probability of two solutions in the population to perform a crossover (an exchange of their selected features).
– Mutation rate: 0.001. Represents the probability for each bit (True if a feature is selected, 0 otherwise) of a solution to be flipped.
– Solver mutation rate: 0.02. Represents the probability of using the solver to repair a solution during the mutation process.

Note that while some of the above parameters might not be optimal (particularly the number of mutations which can grow too large with the number of features), we keep the values of these parameters the same as those defined by [6].

We also use one heuristic in our algorithm: we do not do any bit flip for mandatory or dead features as this always leads to invalid products. We use the engine of the MILP solver *IBM ILOG CPLEX*. We use the hypervolume metric proposed by Fonseca et al. [19] as the indicator in the indicator-based evolutionary algorithm. We ran all our algorithm and determined the average over 10 runs (for each randomly generated instance).

5 Performance of MILPIBEA vs. SATIBEA

In this section, we report on how MILPIBEA and SATIBEA perform on the multi-objective features selection problem. We perform our comparison in three steps:

(i) we compare the efficiency and effectiveness of both MILP and SAT solvers to repair infeasible solutions, (ii) we compare the evolution of SATIBEA and MILP-IBEA performances with respect to hypervolume, and (iii) compare the overall performance achieved by SATIBEA and MILPIBEA with respect to the considered multi-objective performance metrics.

5.1 Comparison of the Reparation Process

To validate our approach, we would like first to evaluate the way the MILP reparation approach compares against the SAT one.

Consequently, we take 300 solutions that are randomly generated by IBEA as its initial population for each FM and repair them with either SAT (in SATIBEA) or MILP (in MILPIBEA). Then, we measure the average execution time to repair each solution (in milliseconds), the average number of unset variables in the model used by the reparation technique (#unset variables), and the average number of modified features from the original solution to the repaired one (#mod). Results are reported in Table 2 (averaged over the number of solutions).

Table 2 Comparison of SATIBEA and MILP reparations. Lower values of both time and number of modifications (#mod) the better. Best values for each feature model are in bold.

Feature Model	SAT Reparation			MILP Reparation		
	Time (ms)	#unset variables	#mod	Time (ms)	#unset variables	#mod
Linux Kernel	5,934	5,725	3,384	**4,486**	**2,872**	**2,139**
Fiasco	76	959	334	**59**	**511**	**283**
FreeBSD	1,248	866	655	**520**	**386**	**316**
μClinux	35	621	303	**10**	**200**	**100**
eCos	47	1079	598	**19**	**455**	**282**

We can see that the MILP reparation is faster on average than the SAT reparation in all FMs. This is likely due to the small size of problems solved by the MILP reparation method (#unset variables are lower in MILP reparation than in SAT reparation). However, it is worth noting that since we are using external solvers (i.e., CPLEX and SAT4J), their usage might pose a threat to the validity of time measurements–especially since MILP problems are often harder to solve than SAT problems.

The most important and robust result in Table 2 is the number of feature modifications. We notice that the number of modified features per solution using the MILP reparation is lower than when using SATIBEA's reparation. As our MILP reparation method performs fewer modifications on the infeasible solutions to turn them into valid ones, it could be more interesting to use it in a genetic algorithm instead of SAT: indeed, fewer modifications imply a better conservation of the accumulated knowledge during the generations.

5.2 Evolution of Hypervolume Performance

After we have shown that MILP reparation is faster and generates feasible solutions that are closer to the original infeasible ones, we would like to evaluate the impact of using MILP instead of SAT within IBEA.

Figure 4 shows the average evolution of hypervolume achieved by MILPIBEA and SATIBEA over time on all the considered feature models.

We see from Figure 4 that MILPIBEA achieves the best hypervolume than SATIBEA in 3 out of 5 FMs. MILPIBEA and SATIBEA achieve similar results on the two other FMs. The three FMs on which MILPIBEA achieves the best results are notably the largest among them (i.e., Linux Kernel, Fiasco, and FreeBSD).

We also see from Figure 4 that MILPIBEA achieves most of its hypervolume improvement during the first quarter of the execution time. After making this large improvement, the hypervolume seems to reach a plateau. SATIBEA, on the other hand, improves its hypervolume at a slower pace on large FMs and only has a fast hypervolume improvement (at a similar rate as MILPIBEA) on the smaller FMs (i.e., μClinux and eCos). Eventually, the hypervolume obtained by SATIBEA also plateaus at either a lower than or equal value as MILPIBEA. Since both approaches plateau after some generations, it would be worth investigating the introduction of a local search as a mean to re-energise the search process (e.g., [20, 21]).

5.3 Performance over Different Metrics

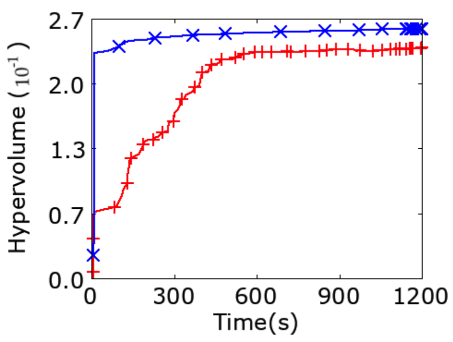
After analysing the evolution of the hypervolume obtained by MILPIBEA and SATIBEA, we would like to evaluate the overall performance achieved by each of these algorithms with respect to the different multi-objective performance metrics.

Table 3 shows the average performances with respect to hypervolume, IGD, PFS, and spread achieved by both MILPIBEA and SATIBEA at the end of their execution time on the 5 different FMs.

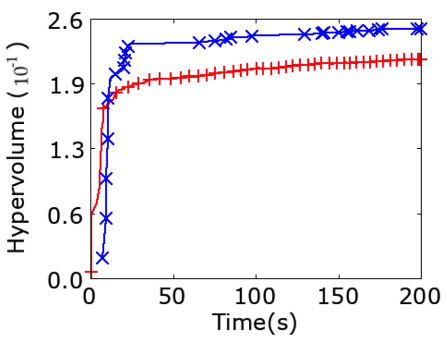
Table 3 Average performance achieved by SATIBEA and MILPIBEA within the allowed execution time on each dataset.

Feature Model	Algorithm	HV (10^{-1})	IGD (10^{-4})	PFS	Spread (10^{-1})
Linux Kernel	SATIBEA	2.37	5.38	291	**12.4**
	MILPIBEA	**2.57**	**2.44**	**300**	6.71
Fiasco	SATIBEA	2.17	31.2	99.5	**9.67**
	MILPIBEA	**2.51**	**3.72**	**300**	6.19
FreeBSD	SATIBEA	2.43	5.54	220	**13.1**
	MILPIBEA	**2.65**	**2.64**	**300**	7.26
μClinux	SATIBEA	**2.40**	29.5	**398**	7.55
	MILPIBEA	2.39	**3.80**	291	**8.37**
eCos	SATIBEA	2.42	18.8	126	**12.7**
	MILPIBEA	**2.50**	**3.93**	**300**	7.24

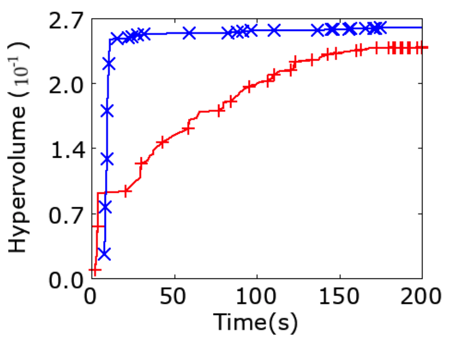
Table 3 shows that MILPIBEA achieves the best results in terms of hypervolume in 4 out of 5 FMs on average. It also shows that MILPIBEA achieves the best



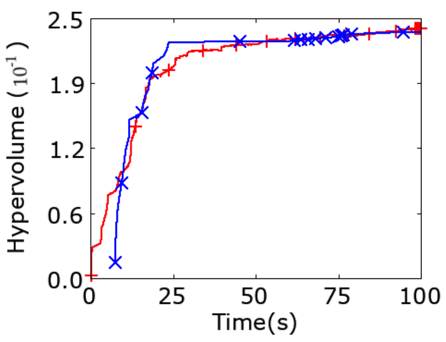
Linux Kernel



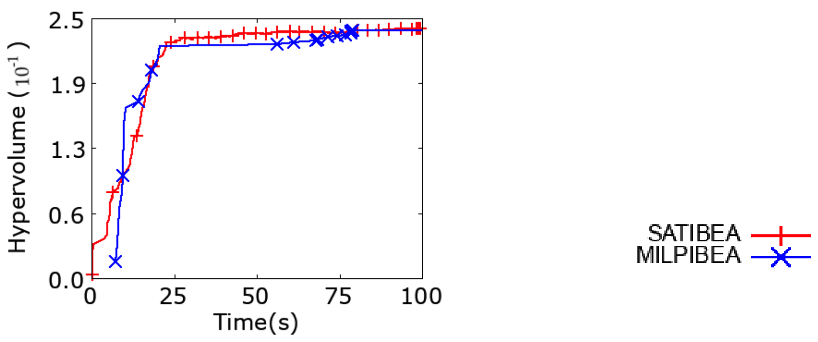
Fiasco



FreeBSD



μClinux



eCos

Fig. 4 Average hypervolume evolution over time with MILPIBEA and SATIBEA on various FM models. The higher the better for the hypervolume.

results in IGD (in 5 out of 5 FMs on average) and in PFS (in 4 out of 5 FMs on average). This is an indication that MILPIBEA not only finds good solutions, but it also finds many of them. Similarly to what has been seen in Figure 4, MILPIBEA achieves better results on the larger FMs (i.e., Linux Kernel, Fisco and FreeBSD) and similar results as SATIBEA on the two smaller FMs (i.e., μClinux and eCos).

Spread is the only performance metric on which MILPIBEA does not perform as well. SATIBEA achieves the best spread on 4 out 5 FMs on average. However, while spread is an important metric, it should not be considered alone, as solutions can be well spread over the obtained Pareto front and also be of poor quality.

In an attempt to better understand the reasons why MILPIBEA is not performing on spread as well as on other performance metrics, we show in Figure 5 the average evolution of spread achieved by MILPIBEA and SATIBEA over time on all the considered feature models.

We see from Figure 5 that the spread in both SATIBEA and MILPIBEA increases drastically at the initial generations to reach its highest levels which indicates that both algorithms perform a large exploration at the beginning of their evolutionary process. Looking at the evolution of HV (Figure 4), it is also clear that this increase in spread comes with large improvements in HV. However, we notice that the smaller spread reached by MILPIBEA in its first generations enables it to achieve higher a hypervolume than SATIBEA (despite a higher spread).

After peaking, spread often stabilises in the case of MILPIBEA around the same levels and only drops slightly (when it does). This drop in spread comes with small HV improvements which indicates that MILPIBEA finds more non-dominated solutions close to those it already found and that its distant solutions as still part of its non-dominated solution set. However, in the case of SATIBEA, spread behaves in different ways: it stabilises (i.e., Fiasco, FreeBSD, and eCos), drops (i.e., μClinux), or drops then stabilises (i.e., Linux kernel). When considering the evolution of both spread and HV achieved by SATIBEA at the same time, there is no clear correlation. spread can drop while only achieving a small improvement in HV (e.g., in μClinux). Spread can also stagnate while achieving significant HV improvements (e.g., in Linux Kernel).

Overall, both evolutionary approaches (SATIBEA and MILPIBEA) tend to increase spread to a certain level, then maintain it (except on μClinux). However, MILPIBEA tends to focus more on achieving a large hypervolume rather than a large spread during its initial steps. While this seems to be a good behaviour in our usecase, it might cause a diversity problem in other cases.

6 Performance on Evolved Feature Models

We now compare MILPIBEA and SATIBEA in the case of the multi-objective feature selection problem in evolving FMs. As described in Section 2, the notion of evolution is represented by features/constraints modifications in the FM. In our case, we considered the Linux kernel version 2.6.28.6 as the original FM, and we generated 10 modified versions. Because of evolution, the original FM has been optimised, and its solutions are given as initial populations to SATIBEA and MILPIBEA: the purpose is to improve the quality of results on modified FMs as fast as possible.

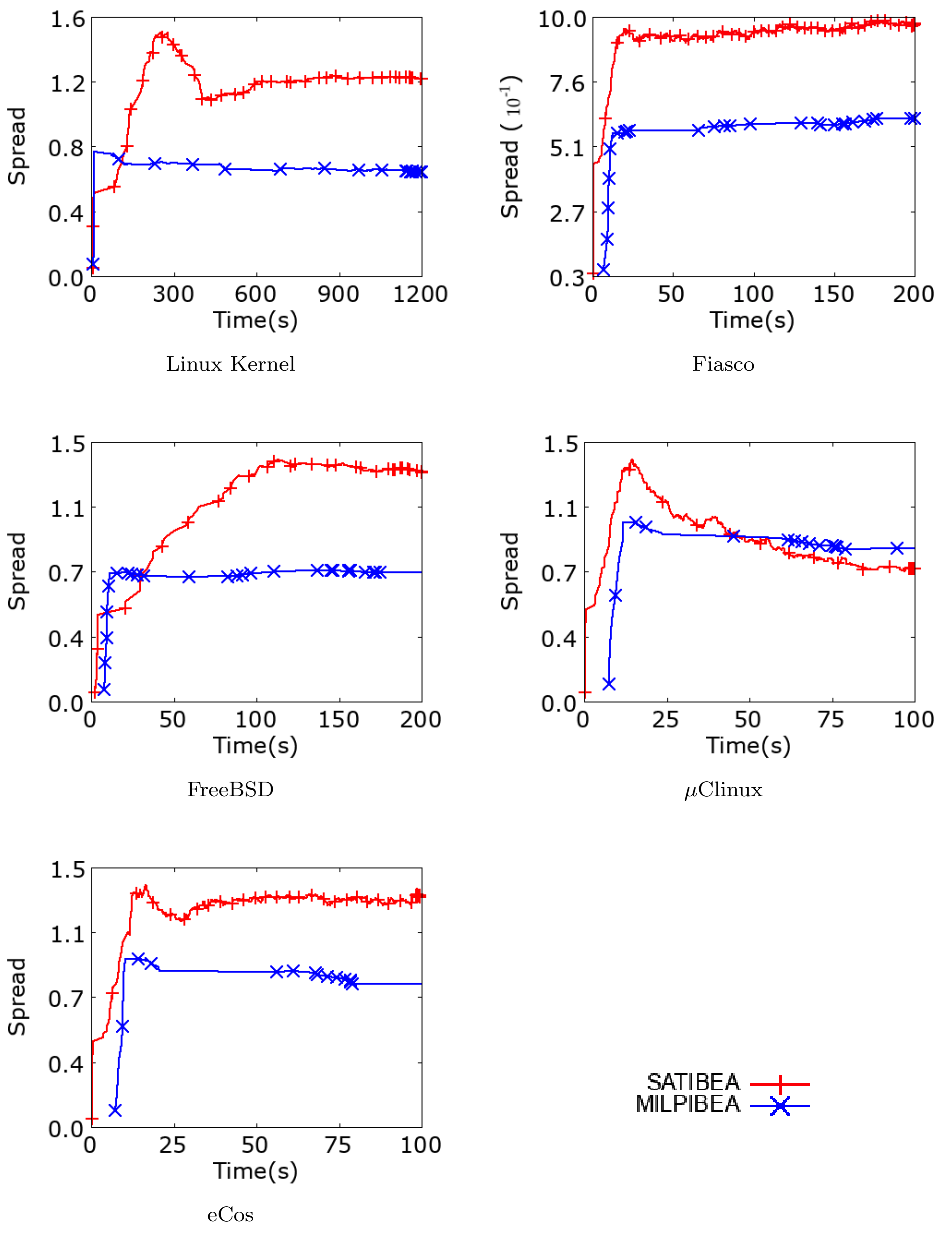


Fig. 5 Average spread evolution over time with MILPIBEA and SATIBEA on various FM models. The higher the better for the spread.

6.1 Evolved Dataset

We have previously studied the demographics (features/constraints) and evolution patterns of 21 successive versions of the Linux kernel [8,22]. We have observed that on average there was only 4.6% difference in terms of features between a version and its successor (out of those changes, 21.22% were removed features and 78.78% were added features). We have also evaluated the size of the clauses/constraints in the problem, as we need to know how the constraints we add in the problem should look and found that a large proportion of the FMs' constraints have 6 features (39%), 5 features (16%), 18 features (14%) or 19 features (14%). Besides, we have designed a generator of synthetic FM evolutions based on the real evolution of the Linux kernel – hence a realistic benchmark but with more variability than in a real one, allowing us also to get several synthetic datasets corresponding to these characteristics.

Our FM generator uses two parameters representing the percentage of feature modifications (added/removed) and the percentage of constraint modifications (added/removed) from the original FM (Linux kernel version 2.6.28.6). The higher those percentages are, the more different the new FM is from its original. The FM generator uses the proportions observed in the 20 FMs to generate new features/remove old ones, and to generate new constraints of a particular length. We use the following values to generate evolved FMs[2]: from 5% of modified features and 1% of modified constraints (FM 5_1) to 20% of modified features and 10% of modified constraints (FM 20_10).

6.2 Performance Without Seeding from Original Feature Model

We now report on how MILPIBEA and SATIBEA perform on the multi-objective features selection problem – in particular with respect to the achieved hypervolume and the required time to reach that performance. We initially discuss the case where we do not take advantage of the evolved feature models and provide the different algorithms with entirely random initial populations.

Figure 4 shows the evolution using SATIBEA or MILPIBEA in terms of hypervolume when applied on our 10 evolved feature models: each of them is a modification of the 2.6.28 version of Linux kernel represented by a pair (x_y) where x is the percentage of features modified and y is the percentage of constraints modified. Hypervolume is computed based only on the solutions of the current population. We are not seeding the initial population: the problem studied in these results is the multi-objective feature selection problem without taking advantage of the evolution of FMs. The initial population is generated randomly for both algorithms.

Our results indicate that MILPIBEA outperforms SATIBEA in terms of hypervolume on all instances on average. Figure 6 also indicates that MILPIBEA is more efficient on the most constrained problems (i.e., with constraint modifications $\geq$ 5%). MILPIBEA reaches a good hypervolume performance only after 100s, then its performance increases slowly. On the other hand, we can see that SATIBEA's hypervolume increases at a slower pace than MILPIBEA's; then its hypervolume stays stable (within a small interval).

[2] The dataset is available at https://github.com/aventresque/EvolvingFMs

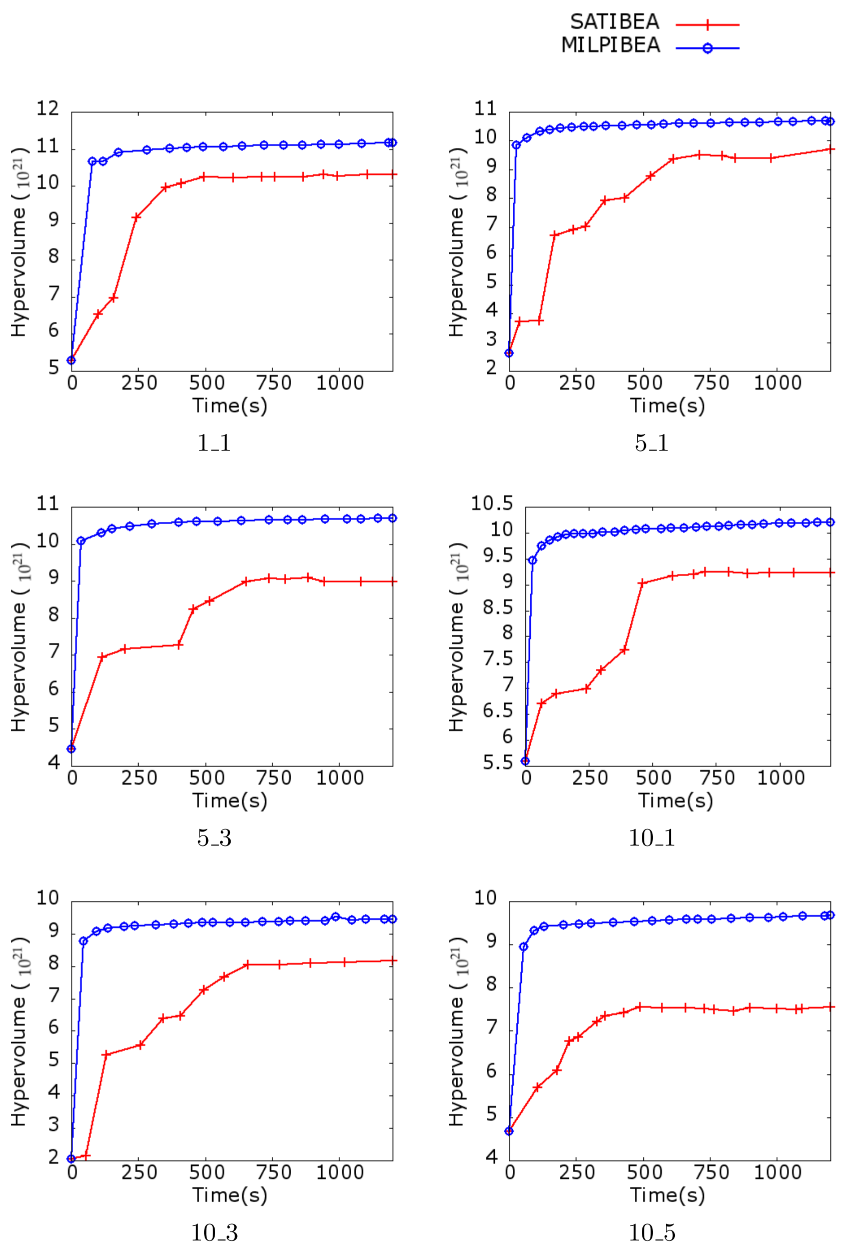
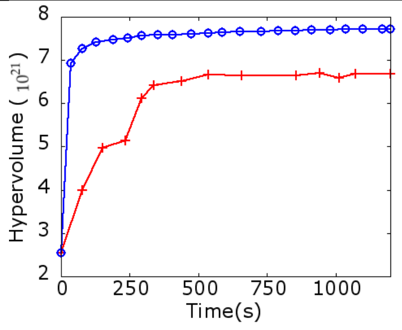
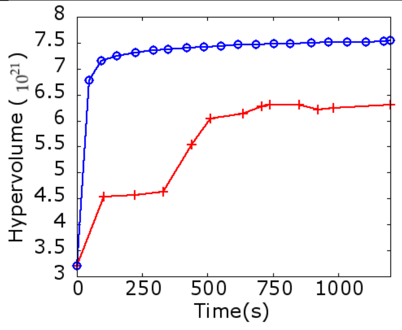


Figure 6 – *Continued on next page*

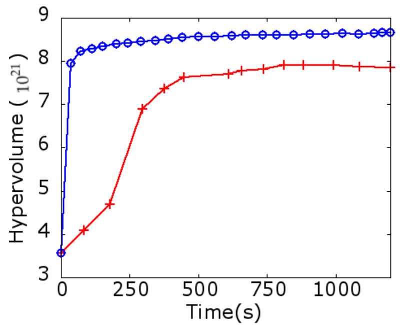
Figure 6 – *Continued from previous page*



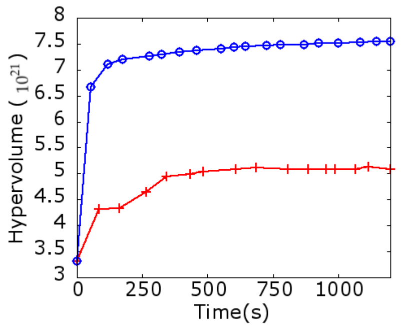
20_1



20_3



20_5



20_10

Figure 6: Comparison of MILPIBEA and SATIBEA on various evolved Linux Kernel FMs without seeding. The higher the better for the hypervolume.

6.3 Performance With Seeding from Original Feature Model

Figure 7 shows the hypervolume of the populations at every new generation for both SATIBEA and MILPIBEA when given (seeded with) the solutions of the original FM as initial populations. We see that both algorithms start from a relatively good hypervolume, which shows the quality of the initial population and the interest for taking advantage of the evolution.

We also see that MILPIBEA successfully improves the hypervolume, whereas SATIBEA struggles when seeded. This is mainly because MILPIBEA has a reparation method that allows it to take advantage of the initial population's good characteristics by not changing a lot of features in solutions that are obtained from the crossover. However, SATIBEA requires to modify several features, making the solutions obtained from the reparation procedure almost random.

Moreover, we can observe that unlike in SATIBEA, MILPIBEA's hypervolume continues improving slowly even after the limit (i.e., 1,200s). A larger allowed time would lead to better solutions. MILPIBEA stagnates only after 2,400s beyond which we might consider adding some local search phase [21, 23, 24].

When comparing MILPIBEA without seeds and MILPIBEA with seeds: after the first generation, MILPIBEA with seeds is 10.5% better in hypervolume than without seeds. It also reaches 97.28% of MILPIBEA's final hypervolume (computed in 1,200s) after only one generation (42.29s on average). This shows us that a good initial population improves the time needed to reach good solutions.

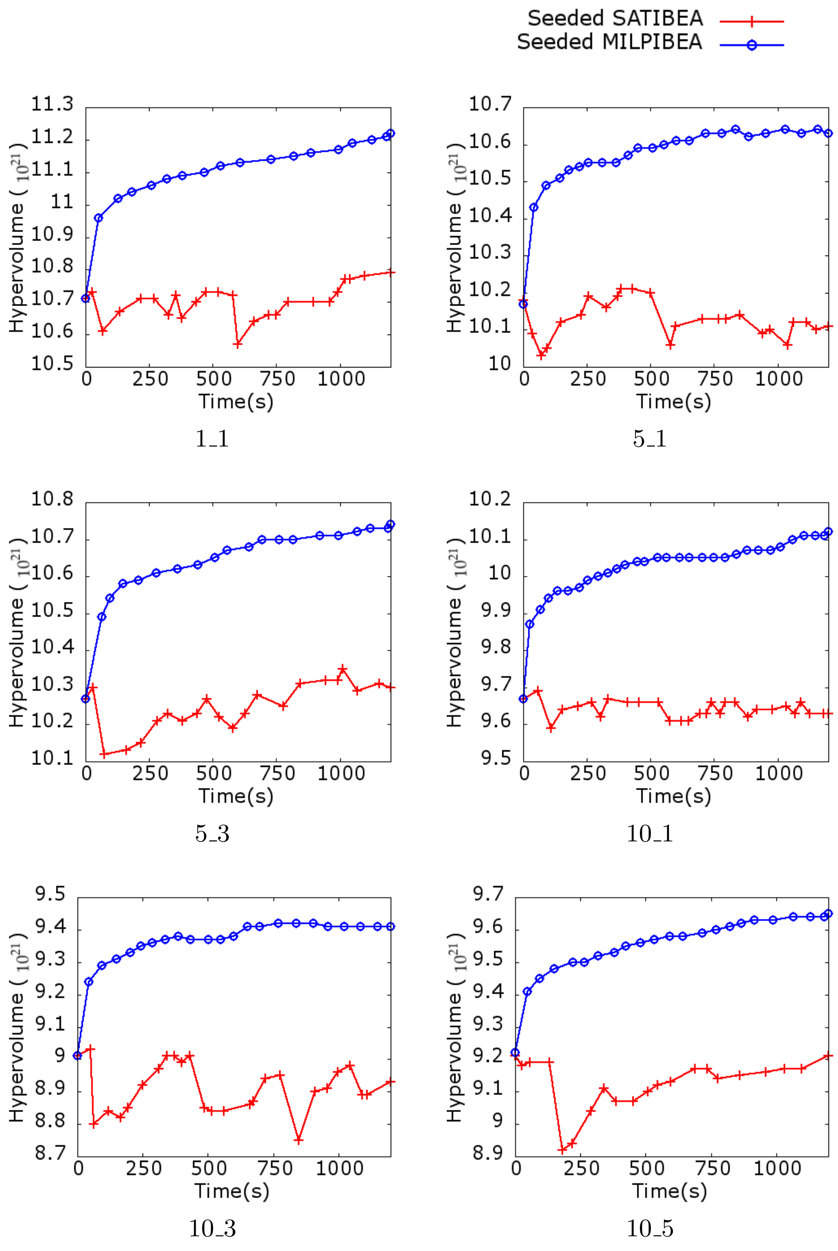
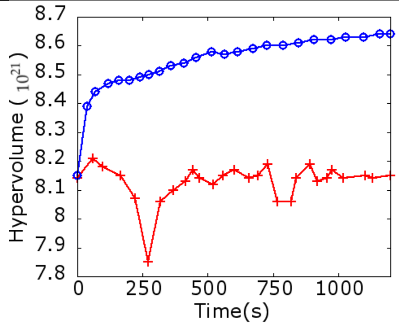
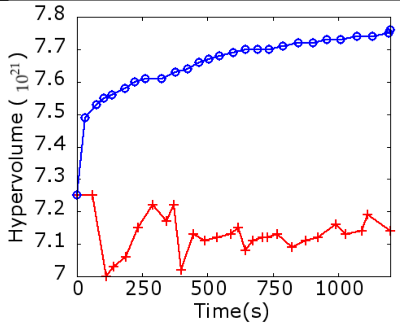


Figure 7 – *Continued on next page*

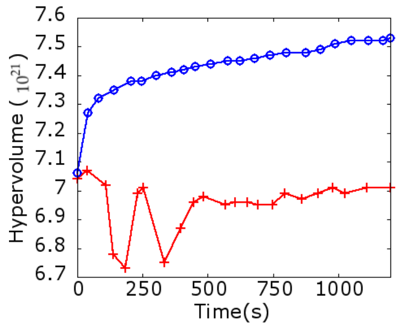
Figure 7 – *Continued from previous page*



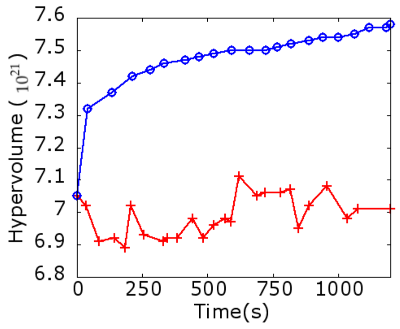
20_1



20_3



20_5



20_10

Figure 7: Comparison of the hypervolume achieved by MILPIBEA and SATIBEA on various evolved Linux Kernel FMs by starting with solutions from the original FM as initial populations. The higher the hypervolume the better.

7 Conclusion and Future Work

Real-life optimisation problems such as multi-objective feature selection in SPL include several constraints, thus rendering the task of finding a feasible solution a challenge on its own.

Some works in the literature have designed exact approaches to generate feasible solutions or repair infeasible ones in the context of multi-objective feature selection in SPL. However, while they have achieved better performances, their improvements stagnate over time.

In this paper, we build upon our previous work on evolving FMs where we proposed a method based on a combination of a genetic algorithm (IBEA) with a reparation using a MILP solver (i.e., CPLEX). We use our novel MILP reparation approach on the general multi-objective feature selection in SPL and show its efficiency with respect to a SAT reparation in both regular and evolving SPL scenarios. Our evaluation showed the importance of using a MILP solver to reduce the number of modifications when repairing a solution. Moreover, our evaluation also showed that our approach achieves better and faster results than SATIBEA on the multi-objective features selection in large scale SPLs. It also showed that our approach outperforms SATIBEA in the context of evolving SPL.

Our future work will be to investigate the impact of performing solution reparation (i) at different stages of the evolutionary process, (ii) with varying frequencies, and (iii) at diverse optimality ratios. The goal would be to have a guideline for using reparation techniques in evolutionary algorithms and ultimately create automatic/adaptive parametrisation of the reparation method in evolutionary algorithms. Our future work will also evaluate the utility of introducing a local search when the genetic algorithm stagnates as a mean to re-energise the search process to find better solutions.

Acknowledgement

This work was supported with the financial support of the Science Foundation Ireland grant 13/RC/2094_P2 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Science Foundation Ireland Research Centre for Software (www.lero.ie).

Declarations

Conflicts of interest/Competing interests

The authors declare that they have no conflict of interest.

Availability of data and material

Dataset is publicly available at https://github.com/aventresque/EvolvingFMs.

Code availability

MILPIBEA publicly available at https://github.com/takfarinassaber/MILPIBEA.

Authors' contributions

- Conceptualisation and Discussion: T.S., D.B., G.B. and A.V.
- Data preparation: T.S., D.B., and A.V.
- Experiments: T.S., D.B., and A.V.
- Writing original draft: T.S., D.B., G.B., and A.V.

All authors have read and agreed to the published version of the manuscript.

References

1. Ramirez, A., Romero, J.R., Ventura, S.: A survey of many-objective optimisation in search-based software engineering. Journal of Systems and Software **149** (2019) 382–395
2. Metzger, A., Pohl, K.: Software product line engineering and variability management: achievements and challenges. In: FSE. (2014) 70–84
3. Neto, J.C., da Silva, C.H., Colanzi, T.E., Amaral, A.M.M.M.: Are MAs profitable to search-based PLA design? IET Software **13**(6) (2019) 587–599
4. Nair, V., Agrawal, A., Chen, J., Fu, W., Mathew, G., Menzies, T., Minku, L., Wagner, M., Yu, Z.: Data-driven search-based software engineering. In: MSR. (2018) 341–352
5. Harman, M., Jia, Y., Krinke, J., Langdon, W.B., Petke, J., Zhang, Y.: Search based software engineering for software product line engineering: a survey and directions for future work. In: SPLC. (2014) 5–18
6. Henard, C., Papadakis, M., Harman, M., Le Traon, Y.: Combining multi-objective search and constraint solving for configuring large software product lines. In: ICSE. (2015) 517–528
7. Saber, T., Brevet, D., Botterweck, G., Ventresque, A.: Milpibea: Algorithm for multi-objective features selection in (evolving) software product lines. In: EvoCop. (2020) 164–179
8. Saber, T., Brevet, D., Botterweck, G., Ventresque, A.: Is seeding a good strategy in multi-objective feature selection when feature models evolve? Information and Software Technology **95** (2018) 266–280
9. Pleuss, A., Botterweck, G., Dhungana, D., Polzer, A., Kowalewski, S.: Model-driven support for product line evolution on feature level. Journal of Systems and Software **85**(10) (2012) 2261–2274
10. Salcedo-Sanz, S.: A survey of repair methods used as constraint handling techniques in evolutionary algorithms. Computer science review **3**(3) (2009) 175–192
11. Ray, T., Singh, H.K., Isaacs, A., Smith, W.: Infeasibility driven evolutionary algorithm for constrained optimization. In: Constraint-handling in evolutionary optimization. Springer (2009) 145–165
12. Singh, H.K., Alam, K., Ray, T.: Use of infeasible solutions during constrained evolutionary search: A short survey. In: ACALCI. (2016) 193–205
13. Carvalho, É.C., Bernardino, H.S., Hallak, P.H., Lemonge, A.C.: An adaptive penalty scheme to solve constrained structural optimization problems by a craziness based particle swarm optimization. Optimization and Engineering **18**(3) (2017) 693–722
14. Sayyad, A.S., Goseva-Popstojanova, K., Menzies, T., Ammar, H.: On parameter tuning in search based software engineering: A replicated empirical study. In: RESER. (2013) 84–90
15. Guo, J., Liang, J.H., Shi, K., Yang, D., Zhang, J., Czarnecki, K., Ganesh, V., Yu, H.: SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines. Software & Systems Modeling **18**(2) (2019) 1447–1466
16. Xue, Y., Li, Y.F.: Multi-objective integer programming approaches for solving optimal feature selection problem: a new perspective on multi-objective optimization problems in SBSE. In: ICSE. (2018) 1231–1242
17. Sayyad, A.S., Menzies, T., Ammar, H.: On the value of user preferences in search-based software engineering: a case study in software product lines. In: ICSE. (2013) 492–501
18. Berger, T., She, S., Lotufo, R., Wasowski, A., Czarnecki, K.: A study of variability models and languages in the systems software domain. IEEE Transactions on Software Engineering (2013) 1611–1640
19. Fonseca, C.M., Paquete, L., López-Ibánez, M.: An improved dimension-sweep algorithm for the hypervolume indicator. In: CEC. (2006) 1157–1163
20. Saber, T., Thorburn, J., Murphy, L., Ventresque, A.: VM reassignment in hybrid clouds for large decentralised companies: A multi-objective challenge. Future Generation Computer Systems **79** (2018) 751–764
21. Saber, T., Gandibleux, X., O'Neill, M., Murphy, L., Ventresque, A.: A comparative study of multi-objective machine reassignment algorithms for data centres. Journal of Heuristics (2019) 1–32
22. Brevet, D., Saber, T., Botterweck, G., Ventresque, A.: Preliminary study of multi-objective features selection for evolving software product lines. Symposium on search based software engineering (2016)

23. Shi, K., Yu, H., Fan, G., Guo, J., Chen, L., Yang, X., Sun, H.: Mutation with local searching and elite inheritance mechanism in multi-objective optimization algorithm: A case study in software product line. International Journal of Software Engineering and Knowledge Engineering **29**(09) (2019) 1347–1378
24. Saber, T., Delavernhe, F., Papadakis, M., O'Neill, M., Ventresque, A.: A hybrid algorithm for multi-objective test case selection. In: CEC. (2018) 1–8