Automated Generation of Computationally Hard Feature Models using Evolutionary Algorithms

Sergio Segura[a,*], José A. Parejo[a,**], Robert M. Hierons[b], David Benavides[a], Antonio Ruiz-Cortés[a]

[a]*Department of Computer Languages and Systems, University of Seville*
Av Reina Mercedes S/N, 41012 Seville, Spain
[b]*School of Information Systems, Computing and Mathematics, Brunel University*
Uxbridge, Middlesex, UB7 7NU United Kingdom

Abstract

A feature model is a compact representation of the products of a software product line. The automated extraction of information from feature models is a thriving topic involving numerous analysis operations, techniques and tools. Performance evaluations in this domain mainly rely on the use of random feature models. However, these only provide a rough idea of the behaviour of the tools with average problems and are not sufficient to reveal their real strengths and weaknesses. In this article, we propose to model the problem of finding computationally hard feature models as an optimization problem and we solve it using a novel evolutionary algorithm for optimized feature models (ETHOM). Given a tool and an analysis operation, ETHOM generates input models of a predefined size maximizing aspects such as the execution time or the memory consumption of the tool when performing the operation over the model. This allows users and developers to know the performance of tools in pessimistic cases providing a better idea of their real power and revealing performance bugs. Experiments using ETHOM on a number of analyses and tools have successfully identified models producing much longer executions times and higher memory consumption than those obtained with random models of identical or even larger size.

Keywords: Search-based testing, software product lines, evolutionary algorithms, feature models, performance testing, automated analysis.

1. Introduction

Software Product Line (SPL) engineering is a systematic reuse strategy for developing families of related software systems [16]. The emphasis is on deriving products from a common set of reusable assets and, in doing so, reducing production costs and time–to–market. The products of an SPL are defined in terms of features where a *feature* is any increment in product functionality [6]. An SPL captures the commonalities (i.e. common features) and variabilities (i.e. variant features) of the systems that belong to the product line. This is commonly done by using a so-called feature model. A *feature model* [32] represents the products of an SPL in terms of features and relationships amongst them (see the example in Fig. 1).

The automated extraction of information from feature models (a.k.a automated analysis of feature models) is a thriving topic that has received much attention in the last two decades [10]. Typical analysis operations allow us to know whether a feature model is consistent (i.e. it represents at least one product), the number of products represented by a feature model, or whether a model contains any errors. Catalogues with up to 30 analysis operations on feature models have been reported [10]. Techniques that perform these operations are typically based on propositional logic [6, 45], constraint programming [9, 76], or description logic [70]. Also, these analysis capabilities can be found in several commercial and open source tools including *AHEAD Tool Suite* [3], *Big Lever Software Gears* [15], *FaMa Framework* [19], *Feature Model Plug-in* [20], *pure::variants* [53] and SPLOT [43].

The development of tools and benchmarks to evaluate the performance and scalability of feature model analysis tools has been recognised as a challenge [7,

*Principal corresponding author
**Corresponding author
Email addresses: sergiosegura@us.es (Sergio Segura), japarejo@us.es (José A. Parejo)

10, 51, 62]. Also, recent publications reflect an increasing interest in evaluating and comparing the performance of techniques and tools for the analysis of feature models [4, 25, 26, 31, 45, 39, 50, 51, 52, 55, 64, 71]. One of the main challenges when performing experiments is finding tough problems that show the strengths and weaknesses of the tools under evaluation in extreme situations, e.g. those producing longest execution times. Feature models from real domains are by far the most appealing input problems. Unfortunately, although there are references to real feature models with hundreds or even thousands of features [7, 37, 66], only portions of them are usually available. This lack of hard realistic feature models has led authors to evaluate their tools with large randomly generated feature models of 5,000 [46, 76], 10,000 [23, 45, 67, 74] and up to 20,000 [47] features. In fact, the size of the feature models used in experiments has been increasing, suggesting that authors are looking for complex problems on which to evaluate their tools [10]. More recently, some authors have suggested looking for hard and realistic feature models in the open source community [13, 21, 49, 61, 62]. For instance, She et al. [62] extracted a feature model containing more than 5,000 features from the Linux kernel.

The problem of generating test data to evaluate the performance of software systems has been largely studied in the field of software testing. In this context, researchers realised long ago that random values are not effective in revealing the vulnerabilities of a system under test. As pointed out by McMinn [42]: *"random methods are unreliable and unlikely to exercise 'deeper' features of software that are not exercised by mere chance"*. In this context, metaheuristic search techniques have proved to be a promising solution for the automated generation of test data for both functional [42] and non–functional properties [2]. *Metaheuristic search techniques* are frameworks which use heuristics to find solutions to hard problems at an affordable computational cost. Examples of metaheuristic techniques include evolutionary algorithms, hill climbing, and simulated annealing [69]. For the generation of test data, these strategies translate the test criterion into an objective function (also called a fitness function) that is used to evaluate and compare the candidate solutions with respect to the overall search goal. Using this information, the search is guided toward promising areas of the search space. Wegener et al. [72, 73] were one of the first to propose the use of evolutionary algorithms to verify the time constraints of software back in 1996. In their work, the authors used genetic algorithms to find input combinations that violate the time constraints of real–time systems, that is, those inputs producing an output too early or too late. Their experimental results showed that evolutionary algorithms are much more effective than random search in finding input combinations maximising or minimising execution times. Since then, a number of authors have followed their steps using metaheuristics and especially evolutionary algorithms for testing non–functional properties such as execution time, quality of service, security, usability or safety [2, 42].

Problem description. Current performance evaluations on the analysis of feature models are mainly carried out using randomly generated feature models. However, these only provide a rough idea of the average performance of tools and do not reveal their specific weak points. Thus, the SPL community lacks mechanisms that take analysis tools to their limits and reveal their real potential in terms of performance. This problem has negative implications for both tool users and developers. On the one hand, tool developers have no means of performing exhaustive evaluations of the strengths and weaknesses of their tools making it hard to find faults affecting their performance. On the other hand, users are not provided with full information about the performance of tools in pessimistic cases and this makes it difficult for them to choose the tool that best meets their needs. Hence, for instance, a user could choose a tool based on its average performance and later realise that it performs very badly in particular cases that appear frequently in their application domain.

In this article, we address the problem of generating computationally hard feature models as a means to reveal the performance strengths and weaknesses of feature model analysis tools. The problem of generating hard feature models has traditionally been addressed by the SPL community by simply randomly generating huge feature models with thousands of features and constraints. That is, it is generally observed and assumed that the larger the model the harder its analysis. However, we remark that these models are still randomly generated and therefore, as warned by software testing experts, they are not sufficient to exercise the specific features of a tool under evaluation. Another negative consequence of using huge feature models to evaluate the performance of tools is that they frequently fall out of the scope of their users. Hence, both developers and users would probably be more interested in knowing whether a tool may crash with a hard model of small or medium size.

Finally, we may mention that using realistic or standard collections of problems (i.e. benchmarks) is equally insufficient for an exhaustive performance evaluation since they do not consider the specific aspects of a tool or technique under test. Thus, feature models that one tool finds hard to analyse could be trivially processed by another and vice versa.

Solution overview and contributions. In this article, we propose to model the problem of finding computationally hard feature models as an optimisation problem and we solve it using a novel *Evolutionary algoriTHm for Optimised feature Models (ETHOM)*. Given a tool and an analysis operation, ETHOM generates input models of a predefined size maximising aspects such as the execution time or the memory consumed by the tool when performing the operation over the model. For the evaluation of our approach, we performed several experiments using different analysis operations, tools and optimisation criteria. In particular, we used FaMa and SPLOT, two tools for the automated analysis of feature models developed and maintained by independent laboratories. In total, we performed over 50 million executions of analysis operations for the configuration and evaluation of our algorithm, during more than six months of work. The results showed how ETHOM successfully identified input models causing much longer executions times and higher memory consumption than randomly generated models of identical or even larger size. As an example, we compared the effectiveness of random and evolutionary search in generating feature models with up to 1,000 features maximising the time required by a constraint programming solver (a.k.a. CSP solver) to check their consistency. The results revealed that the hardest randomly generated model found required 0.2 seconds to analyse while ETHOM was able to find several models taking between 1 and 27.5 minutes to process. Besides this, we found that the hardest feature models generated by ETHOM in the range 500-1,000 features were remarkably harder to process than randomly generated models with 10,000 features. More importantly, we found that the hard feature models generated by ETHOM had similar properties to realistic models found in the literature. This suggests that the long execution times and high memory consumption detected by ETHOM might be reproduced when using real models with the consequent negative effect on the user.

Our work enhances and complements the current state of the art on performance evaluation of feature model analysis tools as follows:

- To the best of our knowledge, this is the first approach that uses a search–based strategy to exploit the internal weaknesses of the analysis tools and techniques under evaluation rather than trying to detect them by chance using randomly generated models.
- Our work allows developers to focus on the search for computationally hard models of realistic size that could reveal performance problems in their tools rather than using huge feature models out of their scope. If a tool performs poorly with the generated models, developers could use the information as input to investigate possible improvements.
- Our approach provides users with helpful information about the behaviour of tools in pessimistic cases helping them to choose the tool that best meets their needs.
- Our algorithm is highly generic and can be applied to any automated operation on feature models in which the quality (i.e. fitness) of models with respect to an optimisation criterion can be quantified.
- Our experimental results show that the hardness of feature models depends on different factors in contrast to related work in which the complexity of the models is mainly associated with their size.
- Our algorithm is ready-to-use and publicly available as a part of the open-source BeTTy Framework [14, 58].

Scope of the contribution. The target audience of this article is practitioners and researchers wanting to evaluate and test the performance of their tools that analyse feature models. Several aspects regarding the scope of our contribution may be clarified, namely:

- Our work follows a black-box approach. That is, our algorithm does not make any assumptions about an analysis tool and operation under test. ETHOM can therefore be applied to any tool or analysis operation regardless of how it is implemented.
- Our approach focuses on testing, not debugging. That is, our work contributes to the detection of performance failures (unexpected behaviour in the software) but not faults (causes of the unexpected behaviour). Once a failure is detected using the test data generated by ETHOM, a tool's developers and designers should use debugging to identify the fault causing it, e.g. bad variable ordering, bad problem encoding, parsing problems, etc.
- It is noteworthy that many different factors could contribute to a technique finding it hard to analyse

a given feature model, some of them not directly related to the analysis algorithm used. Examples including: bad variable ordering, bad problem encoding, parsing problems, bad heuristic selection, etc. However, as previously mentioned, the problem of identifying the factors that make a feature model hard to analyse when using a specific tool is out of the scope of this article.

The rest of the article is structured as follows. Section 2 introduces feature models and evolutionary algorithms. In Section 3, we present ETHOM, an evolutionary algorithm for the generation of optimised feature models. Then, in Section 4, we propose a specific configuration of ETHOM to automate the generation of computationally hard feature models. The empirical evaluation of our approach is presented in Section 5. Section 6 presents the threats to validity of our work. Related work is described in Section 7. Finally, we summarise our conclusions and describe our future work in Section 8.

2. Preliminaries

2.1. Feature models and their analyses

Feature models define the valid combinations of features in a domain and are commonly used as a compact representations of all the products of an SPL. A feature model is visually represented as a tree-like structure in which nodes represent features and connections illustrate the relationships between them. These relationships constrain the way in which features can be combined. Fig. 1 depicts a simplified sample feature model. The model illustrates how features are used to specify and build software for *Global Position System (GPS)* devices. The software loaded in the GPS is determined by the features that it supports. The root feature (i.e. *'GPS'*) identifies the SPL.

Feature models were first introduced in 1990 as a part of the FODA (Feature–Oriented Domain Analysis) method [32]. Since then, feature modelling has been widely adopted by the software product line community and a number of extensions have been proposed in attempts to improve properties such as succinctness and naturalness [56]. Nevertheless, there seems to be a consensus that at a minimum feature models should be able to represent the following relationships among features:

- **Mandatory.** If a child feature is mandatory, it is included in all products in which its parent feature appears. In Fig. 1, all GPS devices must provide support for *Routing*.
- **Optional.** If a child feature is defined as optional, it can be optionally included in products in which its parent feature appears. For instance, the sample model defines *Multimedia* to be an optional feature.
- **Alternative.** Child features are defined as alternative if only one feature can be selected when the parent feature is part of the product. In our SPL, software for GPS devices must provide support for either an *LCD* or *Touch* screen but only one of them.
- **Or-Relation.** Child features are said to have an or-relation with their parent when one or more of them can be included in the products in which the parent feature appears. In our example, GPS devices can provide support for an *MP3 player*, a *Photo viewer* or both of them.

Notice that a child feature can only appear in a product if its parent feature does. The root feature is a part of all the products within the SPL. In addition to the parental relationships between features, a feature model can also contain *cross-tree constraints* between features. These are typically of the form:

- **Requires.** If a feature A requires a feature B, the inclusion of A in a product implies the inclusion of B in the product. GPS devices with *Traffic avoiding* require *Auto-rerouting*.
- **Excludes.** If a feature A excludes a feature B, both features cannot be part of the same product. In our sample SPL, a GPS with *Touch* screen cannot include a *Keyboard* and vice-versa.

The automated analysis of feature models deals with the computer-aided extraction of information from feature models. It has been noted that in the order of 30 different analysis operations on feature models have been reported during the last two decades [10]. The analysis of feature models is usually performed in two steps. First, the analysis problem is translated into an intermediate problem such as a boolean satisfiability problem (SAT) or a Constraint Satisfaction Problem (CSP). SAT problems are often modelled using Binary Decision Diagrams (BDD). Then, an off-the-shelf solver is used to analyse the problem. Most analysis problems related to feature models are NP-hard [7, 51]. However, solvers provide heuristics that work well in practice. Experiments have shown that each technique has its strengths and weaknesses. For instance, SAT solvers are efficient when checking the consistency of a feature model but

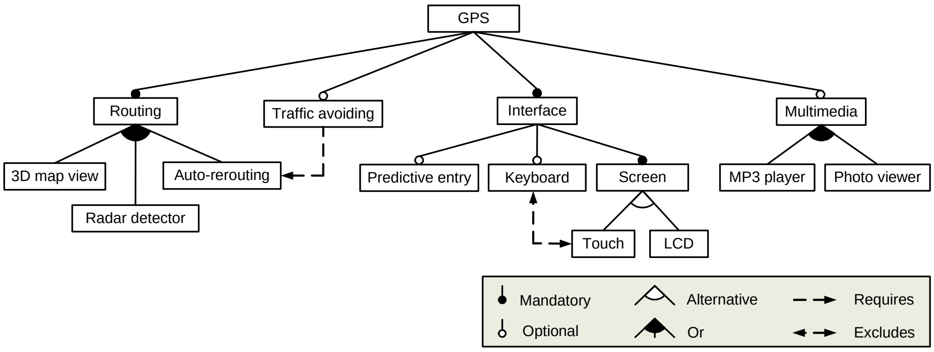


Figure 1: A sample feature model

incapable of calculating the number of products in a reasonable amount of time [11, 45, 51]. BDD solvers are the most efficient solution known for calculating the number of products but at the price of high memory consumption [11, 46, 51]. Finally, CSP solvers are especially suitable for dealing with numeric constraints associated with feature models with attributes (so-called extended feature models) [9].

2.2. Evolutionary algorithms

The principles of biological evolution have inspired the development of a whole branch of optimisation techniques called *Evolutionary Algorithms (EAs)*. These algorithms manage a set of candidate solutions to an optimisation problem that are combined and modified iteratively to obtain better solutions. Each candidate solution is referred to as an *individual* or *chromosome* in analogy to the evolution of species in biological genetics where the DNA of individuals is combined and modified along generations enhancing the species through natural selection. Two of the main properties of EAs are that they are heuristic and stochastic. The former means that an EA is not guaranteed to obtain the global optimum for the optimisation problem. The latter means that different executions of the algorithm with the same input parameters can produce different output, i.e. they are not deterministic. Despite this, EAs are among the most widely used optimisation techniques and have been applied successfully in nearly all scientific and engineering areas by thousands of practitioners. This success is due to the ability of EAs to obtain near optimal solutions to extremely hard optimisation problems with affordable time and resources.

As an example, let us consider the design of a car as an optimisation problem. A similar example was used to illustrate the working of EAs in [73]. Let us suppose that our goal is to find a car design that maximises speed. This problem is hard since a car is a highly complex system in which speed depends on a number of parameters such as engine type and the shape of the car. Moreover, there are likely to be extra constraints like keeping the cost of the car under a certain value, making some designs infeasible. All EA variants are based on a common working scheme shown in Fig. 2. Next, we describe its main steps and relate them to our example.

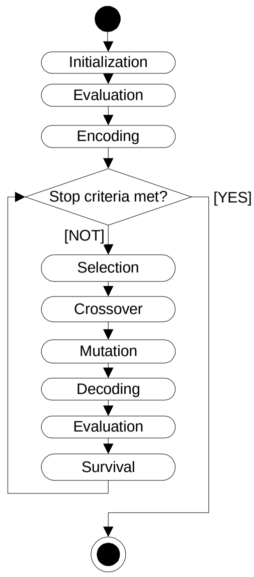


Figure 2: General working scheme of evolutionary algorithms

Initialisation. The initial population (i.e. set of candidate solutions to the problem) is usually generated randomly. In our example, this could be done by randomly choosing a set of values for the design parameters of the car. Of course, it is unlikely that this initial population with contain an optimal or near optimal car design. However, promising values found at this step will be used to produce variants along the optimisation process leading to better designs.

Evaluation. Next, individuals are evaluated using a fitness function. A *fitness function* is a function that receives an individual as input and returns a numerical value indicating the quality of the individual. This enables the objective comparison of candidate solutions with respect to an optimisation problem. The fitness function should be deterministic to avoid interferences in the algorithm, i.e. different calls to the function with the same set of parameters should produce the same output. In our car example, a simulator could be used to provide the maximum speed prediction as fitness.

Stopping criterion. Iterations of the remaining steps of the algorithm are performed until a termination criterion is met. Typical stopping criteria are: reaching a maximum or average fitness value, maximum execution times of the fitness function, number of iterations of the loop (so-called generations) or number of iterations without improvements on the best individual found.

Encoding. In order to create offspring, an individual needs to be *encoded* (represented) in a form that facilitates its manipulation during the rest of the algorithm. In biological genetics, DNA encodes an individual's characteristics on chromosomes that are used in reproduction and whose modifications produce mutants. Classical encoding mechanisms for EAs include the use of binary vectors that encode numerical values in genetic algorithms (so-called binary encoding) and tree structures that encode the abstract syntax of programs in genetic programming (so-called tree encoding) [1, 54]. In our car example, this step would require design patterns of cars to be expressed using a data structure, e.g. binary vectors for each design parameter.

Selection. In the main loop of the algorithm (see Fig. 2), individuals are selected from the current population in order to create new offspring. In this process, better individuals usually have a greater probability of being selected, with this resembling natural evolution where stronger individuals are more likely to reproduce. For instance, two classic selection mechanisms are roulette wheel and tournament selection [1]. When using the former, the probability of choosing an individual is proportional to its fitness and this can be seen as determining the width of the slice of a hypothetical spinning roulette wheel. This mechanism is often modified by assigning probabilities based on the position of the individuals in a fitness–ordered ranking (so-called rank-based roulette wheel). When using tournament selection, a group of n individuals is randomly chosen from the population and a winning individual is selected according to its fitness.

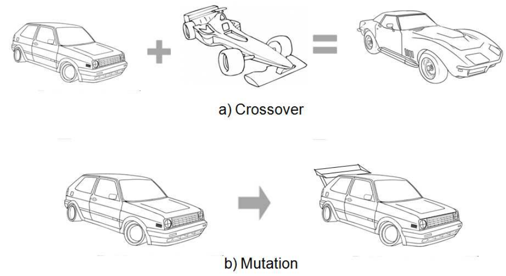


Figure 3: Sample crossover and mutation in the search of an optimal car design.

Crossover. These are the techniques used to combine individuals and produce new individuals in an analogous way to biological reproduction. The crossover mechanism used depends on the encoding scheme but there are a number of widely-used mechanisms [1]. For instance, two classical crossover mechanisms for binary encoding are one-point crossover and uniform crossover. When using the former, a location in the vector is randomly chosen as the break point and portions of vectors after the break point are exchanged to produce offspring (see Fig. 5 for a graphical example of this crossover mechanism). When using uniform crossover, the value of each vector element is taken from one parent or other with a certain probability, usually 50%. Fig. 3(a) shows an illustrative application of crossover in our example of car design. An F1 car and a small family car are combined by crossover producing a sports car. The new vehicle has some design parameters inherited directly from each parent such as number of seats or engine type and others mixed such as shape and intermediate size.

Mutation. At this step, random changes are applied to the individuals. Changes are performed with a certain probability where small modifications are more likely than larger ones. Mutation plays the important role of preventing the algorithm from getting stuck prematurely at a locally optimal solution. An example of mutation in our car optimisation problem is presented in Fig. 3(b). The shape of a family car is changed by adding a back spoiler while the rest of its design parameters remain intact.

Decoding. In order to evaluate the fitness of new and modified individuals *decoding* is performed. For instance, in our car design example, data stored on data structures is transformed into a suitable car design that our fitness function can evaluate. It often happens that the changes performed in the crossover and mutation steps create individuals that are not valid designs or break a constraint, this is usually referred to as an *infeasible individual,* e.g. a car with three wheels. Once an infeasible individual is detected, this can be either replaced by an extra correct one or it can be repaired, i.e. slightly changed to make it feasible.

Survival. Finally, individuals are evaluated and the next population is formed in which individuals with better fitness values are more likely to remain in the population. This process simulates the natural selection of the better adapted individuals that survive and generate offspring, thus improving a species.

3. ETHOM: an Evolutionary algoriTHm for Optimized feature Models

In this section, we present ETHOM, a novel evolutionary algorithm for the generation of optimised feature models. The algorithm takes several constraints and a fitness function as input and returns a feature model of the given size maximising the optimisation criterion defined by the function. A key benefit of our algorithm is that it is very generic and so is applicable to any automated operation on feature models in which the quality (i.e. fitness) of the models can be measured quantitatively. In the following, we describe the basic steps of ETHOM as shown in Fig. 2.

Initial population. The initial population is generated randomly according to the size constraints received as input. The current version of ETHOM allows the user to specify the number of features, percentage of cross-tree constraints and maximum branching factor of the feature model to be generated. Several algorithms for the random generation of feature models have been proposed in the literature [57, 67, 78]. There are also tools such as BeTTy [14, 58] and SPLOT [43, 65] that support the random generation of feature models.

Evaluation. Feature models are evaluated according to the fitness function received as input obtaining a numeric value that represents the quality of a candidate solution, i.e. its fitness.

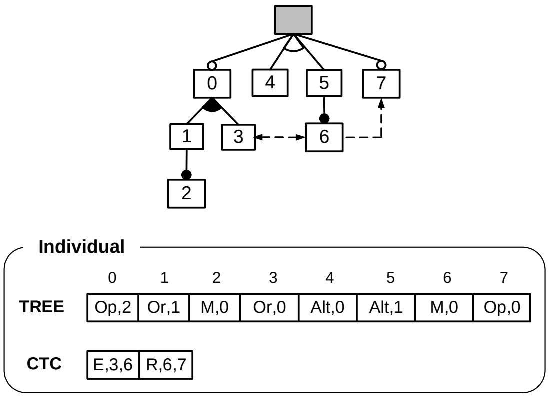


Figure 4: Encoding of a feature model in ETHOM

Encoding. For the representation of feature models as individuals (a.k.a. chromosomes) we propose using a custom encoding. Generic encodings for evolutionary algorithms were ruled out since these either were not suitable for tree structures (i.e. binary encoding) or were not able to produce solutions of a fixed size (e.g. tree encoding), a key requirement in our approach. Fig. 4 depicts an example of our encoding. As illustrated, each model is represented by means of two arrays, one storing information about the tree and another one containing information about *Cross-Tree Constraints (CTC)*. The order of each feature in the array corresponds to the *Depth–First Traversal (DFT)* order of the tree. Hence, a feature labelled with *'0'* in the tree is stored in the first position of the array, the feature labelled with *'1'* is stored the second position and so on. Each feature in the tree array is defined by a pair $< PR, C >$ where PR is the type of relationship with its parent feature (M: Mandatory, Op: Optional, Or: Or-relationship, Alt: Alternative) and C is the number of children of the given feature. As an example, the first position in the tree array, $< Op, 2 >$, indicates that the feature labelled with *'0'* in the tree has an optional relationship with its parent feature and has two child features (those labelled with *'1'* and *'3'*). Analogously, each position in the CTC array stores information about one constraint in the form $< TC, O, D >$ where TC is the type of constraint (R: Requires, E: Excludes) and O and D are the indexes of the origin and destination features in the tree array respectively.

Selection. Selection strategies are generic and can be applied regardless of how the individuals are represented. In our algorithm, we implemented both rank-based roulette-wheel and binary tournament selection strategies. The selection of one or the other

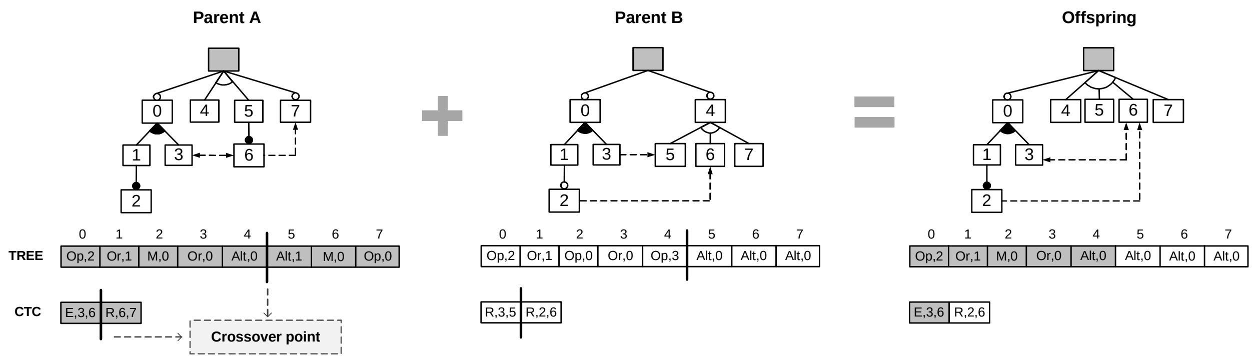


Figure 5: Example of one-point crossover in ETHOM

mainly depends on the application domain.

Crossover. We provided our algorithm with two different crossover techniques, one-point and uniform crossover. Fig. 5 depicts an example of the application of one-point crossover in ETHOM. The process starts by selecting two parent chromosomes to be combined. For each array in the chromosomes, the tree and CTC arrays, a random point is chosen (the so-called crossover point). Finally, the offspring is created by copying the contents of the arrays from the beginning to the crossover point from one parent and the rest from the other one. Notice that the characteristics of our encoding guarantee a fixed size for the individuals in terms of features and CTCs.

Mutation. Mutation operators must be specifically designed for the type of encoding used. ETHOM uses four different types of custom mutation operators, namely:

- *Operator 1.* This randomly changes the type of a relationship in the tree array, e.g. from mandatory,$< \mathbf{M}, 3 >$, to optional,$< \mathbf{Op}, 3 >$.
- *Operator 2.* This randomly changes the number of children of a feature in the tree, e.g. from $< M, \mathbf{3} >$ to $< M, \mathbf{5} >$. The new number of children is in the range $[0, BF]$ where BF is the maximum branching factor indicated as input.
- *Operator 3.* This changes the type of a cross-tree constraint in the CTC array, e.g. from excludes $< \mathbf{E}, 3, 6 >$ to requires $< \mathbf{R}, 3, 6 >$.
- *Operator 4.* This randomly changes (with equal probability) the origin or destination feature of a constraint in the CTC array, e.g. from $< E, \mathbf{3}, 6 >$ to $< E, \mathbf{1}, 6 >$. The implementation of this ensures that the origin and destination features are different.

These operators are applied randomly with the same probability.

Decoding. At this stage, the array-based chromosomes are translated back into feature models so that they can be evaluated. In ETHOM, we identified three types of patterns making a chromosome infeasible or semantically redundant, namely: *i*) those encoding set relationships (or- and alternative) with a single child feature (e.g. Fig. 6(a)), *ii*) those containing cross-tree constraints between features with parental relationship (e.g. Fig. 6(b)), and *iii*) those containing features linked by contradictory or redundant cross-tree constraints (e.g. Fig. 6(c)). The specific approach used to address infeasible individuals, replacing or repairing (see Section 2.2 for details), mainly depends on the problem and it is ultimately up to the user. In our work, we used a repairing strategy described in the next section.

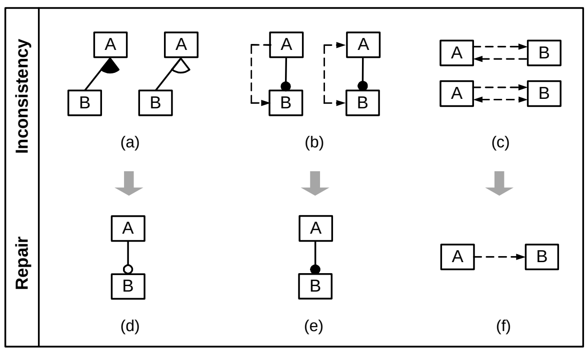


Figure 6: Examples of infeasible individuals and repairs

Survival. Finally, the next population is created by including all the new offspring plus those individuals from the previous generation that were selected for crossover but did not generate descendants.

For a pseudo-code listing of the algorithm we refer the reader to [59].

4. Automated generation of hard feature models

In this section we propose a method that models the problem of finding computationally hard feature models as an optimisation problem and explain how this is solved using ETHOM. In order to find a suitable configuration of ETHOM, we performed numerous executions of a sample optimisation problem evaluating different combination of values for the key parameters of the algorithm, presented in Table 1. The optimisation problem was to find a feature model maximising the execution time taken by the analysis tool when checking model consistency, i.e. whether it represents at least one product. We chose this analysis operation because it is currently the most frequently quoted in the literature [10]. In particular, we searched for feature models of different size maximising execution time in the CSP solver JaCoP [29] integrated into the framework for the analysis of feature models FaMa [19]. Next, we clarify the main aspects of the configuration of ETHOM:

- **Initial population.** We used a Java program implementing the algorithm for the random generation of feature models described by Thüm et al. [67]. For a detailed description of the generation approach, we refer the reader to [59].

- **Fitness function.** Our first attempt was to measure the time (in milliseconds) taken by FaMa to perform the operation. However, we found that the result of the function was significantly affected by the system load and was not deterministic. To solve this problem, we decided to measure the fitness of a feature model as the number of backtracks produced by the analysis tool during its analysis. A *backtrack* represents a partial candidate solution to a problem that is discarded because it cannot be extended to a full valid solution [68]. In contrast to the execution time, most CSP backtracking heuristics are deterministic, i.e. different executions of the tool with the same input produce the same number of backtracks. Together with execution time, the number of backtracks is commonly used to measure the complexity of constraint satisfaction problems [68]. Thus, we can assume that the higher the number of backtracks the longer the computation time.

- **Infeasible individuals.** We evaluated the effectiveness of both replacement and repair techniques. More specifically, we evaluated the following repair algorithm applied to infeasible individuals: *i*) isolated set relationships are converted into optional relationships (e.g. the model in Fig. 6(a) is changed as in Fig. 6(d)), *ii*) cross-tree constraints between features with parental relationships are removed (e.g. the model in Fig. 6(b) is changed as in Fig. 6(e)), and *iii*) two features cannot be linked by more than one cross-tree constraint (e.g. the model in Fig. 6(c) is changed as in Fig. 6(f)).

- **Stopping criterion.** There is no means of deciding when an optimum input has been found and ETHOM should be stopped [73]. For the configuration of ETHOM, we decided to allow the algorithm to continue for a given number of executions of the fitness function (i.e. maximum number of generations) taking the largest number of backtracks obtained as the optimum, i.e. the solution to the problem.

Table 1 depicts the values evaluated for each configuration parameter of ETHOM. These values were based on related work using evolutionary algorithms [23], the literature on parameter setting [18], and our previous experience in this domain [48]. Each combination of parameters used was executed 10 times to avoid heterogeneous results and to allow us to perform statistical analysis on the data. The values underlined are those that provided better results and were therefore selected for the final configuration of ETHOM. In total, we performed over 40 million executions of the objective function to find a good setup for our algorithm.

Parameter	**Values evaluated and selected**
Selection strategy	Roulette-wheel, 2-Tournament
Crossover strategy	One-point, Uniform
Crossover probability	0.7, 0.8, 0.9
Mutation probability	0.005, 0.0075, 0.02
Size initial population	50, 100, 200
#Executions fitness function	2000, 5000
Infeasible individuals	Replacing, Repairing

Table 1: ETHOM configuration

5. Evaluation

In order to evaluate our approach, we developed a prototype implementation of ETHOM. The prototype was implemented in Java to facilitate its integration into the BeTTy Framework [14, 58], an open-source Java tool for functional and performance testing of tools that analyse feature models[1].

We evaluated the efficacy of our approach by comparing it to random search since this is the usual approach for performance testing in the analysis of feature models. In particular, the evaluation of our evolutionary program was performed through a number of experiments. In each experiment, we compared the effectiveness of a random generator and ETHOM when searching for feature models maximising properties such as the execution time or memory consumption required for their analysis. Additionally, we performed some extra experiments studying the characteristics of the hard feature models generated and the behaviour of ETHOM when allowed to run for a large number of generations. The setup and results of our experiments as well as the statistical analysis of the data are summarised in this section and fully reported in an external technical report due to space limitations [59]. The experimental work and the statistical analysis of the results took more than six months and involved several people.

All the experiments were performed on a cluster of four virtual machines equipped with an Intel Core 2 CPU 6400@2.13GHz running Centos OS 5.5 and Java 1.6.0_20 on 1400 MB of dedicated memory. These virtual machines ran on a cloud of servers equipped with Intel Core 2 CPU 6400@2.13Ghz and 4GB of RAM memory managed using Opennebula 2.0.1.

5.1. Experiment #1: Maximizing execution time in a CSP solver

This experiment evaluated the ability of ETHOM to search for input feature models maximising the analysis time of a solver. In particular, we measured the execution time required by a CSP solver to determine whether the input model was consistent (i.e. it represents at least one product). This was the problem used to tune the configuration of our algorithm. Again, we chose the consistency operation because currently it is the most frequently mentioned in the literature. Next, we present the setup and results of our experiment.

Experimental setup. This experiment was performed through a number of iterative steps. In each step, we randomly generated 5,000 feature models and checked their consistency, saving the maximum fitness obtained. Then, we executed ETHOM and allowed it to run for the same number of executions of the fitness function (5,000) and compared the results. Recall that the size of the population in our algorithm was set to 200 individuals which meant that the maximum number of generations was 25, i.e. 5,000/200. This process was repeated with different model sizes to evaluate the scalability of our algorithm. In particular, we generated models with different combinations of features, {200, 400, 600, 800, 1,000} and percentage of constraints (with respect to the number of features), {10%, 20%, 30%, 40%}. The maximum branching factor was set to 10 in all the experiments. For each model size, we repeated the process 25 times to get averages and performed statistical analysis on the data. In total, we performed about 5 million executions[2] of the fitness function for this experiment. The fitness was set to be the number of backtracks used by the analysis tool when checking the model consistency. For the analysis, we used the solver JaCoP integrated into FaMa v1.0 with the default heuristics *MostConstrainedDynamic* for the selection of variables and *IndomainMin* for the selection of values from the domains. To prevent the experiment from getting stuck, a maximum timeout of 30 minutes was used for the execution of the fitness function in both the random and evolutionary search. If this timeout was exceeded during random generation, the execution was cancelled and a new iteration was started. If the timeout was exceeded during evolutionary search, the best solution found until that moment was returned, i.e. the instance exceeding the timeout was discarded. After all the executions, we measured the execution time of the hardest feature models found for a full comparison, i.e. those producing a larger number of backtracks. More specifically, we executed each returned solution 10 times to get average execution times.

Analysis of results. Fig. 7 depicts the effectiveness of ETHOM for each size range of the feature models generated. We define the *effectiveness* of our evolutionary program as the percentage of times (out of 25) in which ETHOM found a better optimum than random search, i.e. a higher number of backtracks. As illustrated, the effectiveness of ETHOM was over 80% in most of the size ranges, reaching 96% or higher in nine of them. Overall, our evolutionary program found harder feature models than those generated randomly in 85.8% of the executions. We may remark that our algorithm revealed the lowest effectiveness with those models containing 10% of cross-tree constraints. We found that this was

[1]BeTTY was used because it was developed by the authors

[2]5 features ranges x 4 constraints ranges x 25 iterations x 10,000 (5,000 random search + 5,000 evolutionary search)

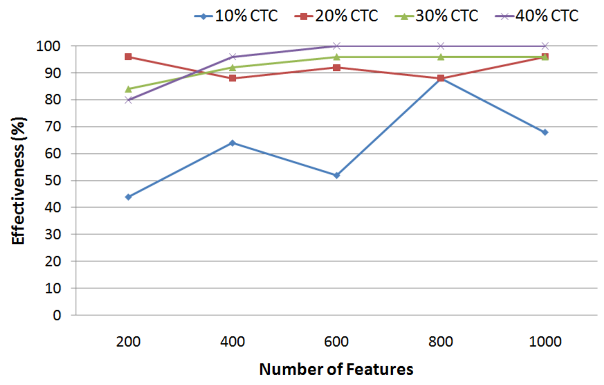


Figure 7: Effectiveness of ETHOM in Experiment #1.

due to the simplicity of the analysis in this size range. The number of backtracks produced by these models was very low, zero in most cases, and thus ETHOM had problems finding promising individuals that could evolve towards optimal solutions.

Table 2 depicts the evaluation results for the range of feature models with 20% of cross-tree constraints. For each number of features and search technique, random and evolutionary, the table shows the average and maximum fitness obtained (i.e. number of backtracks) as well as the average and maximum execution times of the hardest feature models found (in seconds). The effectiveness of the evolutionary program is also presented in the last column. As illustrated, ETHOM found feature models producing a number of backtracks larger by several orders of magnitude than those produced using randomly generated models. The fitness of the hardest models generated using our evolutionary approach was on average over 3,500 times higher than that of randomly generated models (200,668 backtracks against 45.3) and 40,500 times higher in the maximum value (23.5 million backtracks against 1,279). As expected, these results were also reflected in the execution times. On average, the CSP solver took 0.06 seconds to analyse the randomly generated models and 9 seconds to analyse those generated using ETHOM. The superiority of evolutionary search was remarkable in the maximum times ranging from the 0.2 seconds for randomly generated models to the 1,032.2 seconds (17.2 minutes) taken by the CSP solver to analyse the hardest feature model generated by ETHOM. Overall, our evolutionary approach produced a harder feature model than random techniques in 92% of the executions in the range of 20% of constraints. For details regarding the data corresponding to 10%, 30% and 40% of constraints we refer the reader to [59].

Table 3 presents a summary of the results. The table depicts the maximum execution time taken by the CSP solver to analyse the hardest models found using random and evolutionary search. The data shows that ETHOM found models that led to higher execution times than those randomly generated and this was the case for all size ranges. The hardest randomly generated model required 0.2 seconds to be processed. In contrast, ETHOM found four models whose analysis required between 1 and 27.3 minutes (1,644 seconds). We may remark that ETHOM reached the maximum timeout of 30 minutes once during the experiment but random search never produced times over 0.2 seconds. Interestingly, ETHOM was able to find smaller but significantly harder feature models (e.g. 600-10%, 60 seconds) than the hardest randomly generated model found which had 800 features, 20% of CTCs and an analysis time of 0.2 seconds. Finally, the results show that ETHOM found it more difficult to find hard feature models as the percentage of cross-tree constraints increased. We remark, however, that this trend was also observed in the random search with an average fitness of 45.3 backtracks in the range of 20% CTC, 16.6 backtracks in the range of 30% CTC and 9.1 backtracks in the range of 40% CTC. We conclude, therefore, that these results are caused by the CSP solver and the heuristic used which provide a better performance when the models have a high percentage of constraints.

Fig. 8 compares random and evolutionary techniques for the search for a feature model maximising the number of backtracks in two sample executions. Horizontally, the graphs show the number of generations where each generation represents 200 executions of the fitness function. Fig. 8(a) shows that random search reaches its maximum number of backtracks after only 5 generations (about 1000 executions). That is, the random generation of 4,000 other models does not produce any higher number of backtracks and therefore is useless. In contrast to this, ETHOM shows a continuous improvement. After 13 generations (about 2600 executions), the fitness found by evolutionary search is above that of the maximum for the randomly generated models. Fig. 8(b) depicts another example in which random search is ‘lucky’ and finds an instance with a high number of backtracks in the 14th generation. Evolutionary optimisation, however, once again manages to improve the execution times continuously overcoming the best fitness produced using random search after 22 generations. We might note that a significant leap of about 200 backtracks can also be observed in generation 23. In both examples, the curve suggests that ETHOM would find even better solutions if the number of generations was

#Features	Random Search				ETHOM				Effect. (%)
	Avg Fitness	Max Fitness	Avg Time	Max Time	Avg Fitness	Max Fitness	Avg Time	Max Time	
200	8.08	61	0.02	0.03	63.4	215	0.04	0.06	96
400	30.1	389	0.04	0.07	7,128.4	106,655	0.24	2.93	88
600	40.3	477	0.05	0.09	9,188.2	116,479	0.70	7.98	92
800	91.1	1,279	0.08	0.20	22,427.6	483,971	1.28	24.6	88
1000	57.2	582	0.10	0.13	964,532.6	23,598,675	42.5	1,032.2	96
Total	**45.3**	**1,279**	**0.06**	**0.20**	**200,668**	**23,598,675**	**8.96**	**1,032.2**	**92**

Table 2: Evaluation results on the generation of feature models maximising execution time in a CSP solver. Fitness measured in number of backtracks. Time in seconds. CTC=20%

#Features	10% CTC		20% CTC		30% CTC		40% CTC	
	Random	ETHOM	Random	ETHOM	Random	ETHOM	Random	ETHOM
200	0.04	0.06	0.03	0.06	0.04	0.17	0.04	0.08
400	0.05	0.33	0.07	2.93	0.04	0.61	0.08	0.13
600	0.10	59.9	0.09	7.98	0.06	6.62	0.07	4.09
800	0.09	280.4	0.20	24.6	0.10	13.9	0.09	0.52
1,000	0.12	1,643.9	0.13	1,032.2	0.12	1.62	0.10	0.27
Max	**0.12**	**1,643.9**	**0.20**	**1,032.2**	**0.12**	**13.9**	**0.10**	**4.09**

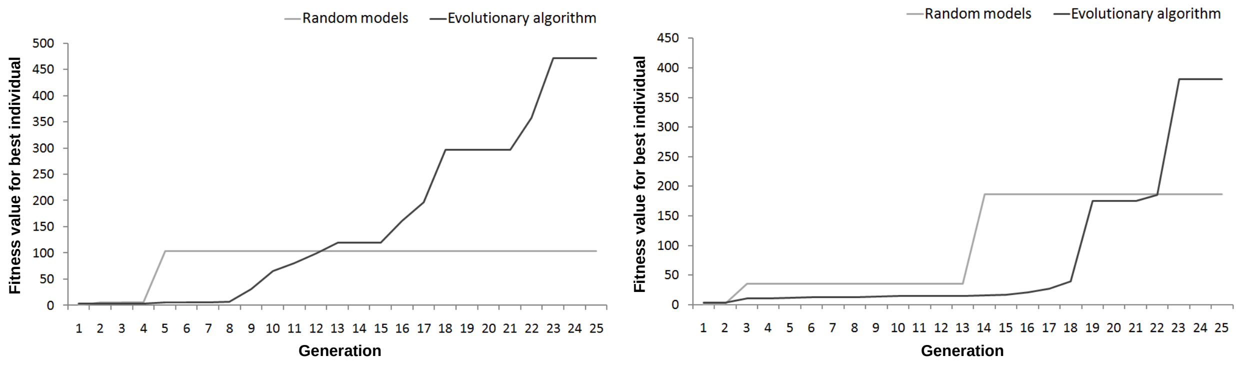
Table 3: Maximum execution times produced by random and evolutionary search. Time in seconds.

increased. This was confirmed in a later experiment in which the program was allowed to run for up to 125 generations (25,000 executions of the fitness function) finding feature models producing more than 77.6 million backtracks (see Section 5.3 for details).

5.2. Experiment #2: Maximizing memory consumption in a BDD solver

This experiment evaluated the ability of ETHOM to generate input feature models maximising the memory consumption of a solver. In particular, we measured the memory consumed by a BDD solver when determining the number of products represented by the model. We chose this analysis because it is one of the hardest operations in terms of complexity and it is the second most frequently quoted operation in the literature [10]. We decided to use a BDD-based reasoner for this experiment since it has proved to be the most efficient option to perform this operation in terms of time [10, 51]. A *Binary Decision Diagram* (BDD) solver is a software package that takes a propositional formula as input and translates it into a graph representation (the BDD itself) that provides efficient algorithms for counting the number of possible solutions. The number of nodes of the BDD is a key aspect since it determines the consumption of memory and can be exponential in the worst case [46]. Next, we present the setup and results of our experiment.

Experimental setup. The experiment consisted of a number of iterative steps. At each step, we randomly generated 5,000 models and compiled each of them into a BDD for use in counting the number of solutions of the input feature model. We then executed ETHOM and allowed it to run for 5,000 executions of the fitness function (i.e. 25 generations) searching for feature models maximising the size of the BDD. Again, this process was repeated with different combinations of features, {50, 100, 150, 200, 250} and percentages of constraints, {10%, 20%, 30%} to evaluate the scalability of our approach. For each model size, we repeated the process 25 times to get statistics from the data. In total, we performed about 3.5 million executions of the fitness function for this experiment. We may remark that we generated smaller feature models than those presented in the previous experiment in order to reduce BDD building time and make the experiment affordable. Measuring memory usage in Java is difficult and computationally expensive since memory profilers usually add a significant overload to the system. To simplify the fitness function, we decided to measure the fitness of a model as the number of nodes of the BDD representing it. This is a natural option used in the research community to compare the space complexity of BDD tools and heuristics [46]. For the analysis, we used the solver JavaBDD [30] integrated into the feature model analysis tool SPLOT [43]. We chose SPLOT for this experiment because it integrates highly



a) Feature models with 400 features and 30% of cross-tree constraints

b) Feature models with 1,000 features and 10% of cross-tree constraints

Figure 8: Comparison of randomly generated models and ETHOM for the search of the highest number of backtracks

efficient ordering heuristics specifically designed for the analysis of feature models using BDDs. In particular, we used the heuristic *'Pre-CL-MinSpan'* presented by Mendonca et al. in [46]. For a detailed description of the configuration of the solver we refer the reader to [59]. As in our previous experiment, we set a maximum timeout of 30 minutes for the fitness function to prevent the experiment from getting stuck. We measured the compilation and execution time of the hardest feature models found to allow a more detailed comparison. Each optimal solution was compiled and executed 10 times to get average times.

Analysis of results. Fig. 9 depicts the effectiveness of ETHOM for each size range of the feature models generated, i.e. percentage of times (out of 25) in which evolutionary search found feature models producing higher memory consumption than randomly generated models. As illustrated, the effectiveness of ETHOM was over 96% in most cases, reaching 100% in 10 out of the 15 size ranges. The lowest percentages were registered in the range of 250 features. When analysing the results, we found that the timeout of 30 minutes was reached frequently in the range of 250 features hindering ETHOM from evolving toward promising solutions. In other words, the feature models generated were so hard that they often took more than 30 minutes to analyse and were discarded. In fact, the maximum timeout was reached 18 times during random generation and 62 times during evolutionary search, 25 of them in the range of 250 features and 30% of constraints. In this size range, ETHOM exceeded the timeout after only 7 generations on average (25 being the maximum). Overall, ETHOM found feature models producing higher memory consumption than random search in 94.4% of the executions. The results suggest, however, that in-

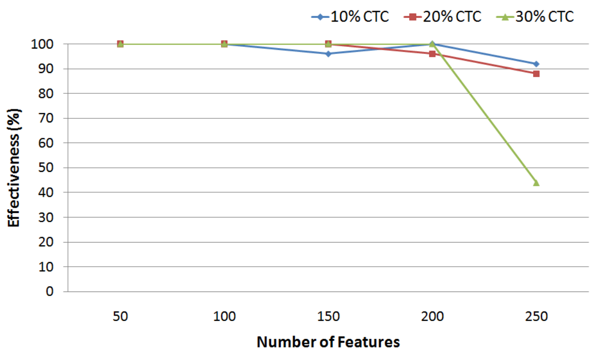


Figure 9: Effectiveness of ETHOM in Experiment #2.

creasing the maximum timeout would significantly improve the effectiveness.

Table 4 depicts the number of BDD nodes of the hardest feature models found using random and evolutionary search. For each size range, the table also shows the computation time (BDD building time + execution time) taken by SPLOT to analyse the model. As illustrated, ETHOM found higher maximum values than random techniques in all size ranges. On average, the BDD size found by our evolutionary approach was between 1.03 and 10.3 times higher than those obtained with random search. The largest BDD generated in random search had 14.8 million nodes while the largest BDD obtained using ETHOM had 20.6 million nodes. Again, the results revealed that ETHOM was able to find smaller but harder models (e.g. 150-30%, 17.7 million nodes) than the hardest randomly generated model found, 250-30% 14.8 million nodes. We may recall that the maximum timeout was reached 62 times during the execution of ETHOM. This result suggests that the maximum found by evolutionary search would have been

	10% CTC				20% CTC				30% CTC			
	Random		ETHOM		Random		ETHOM		Random		ETHOM	
#Features	**BDD size**	**Time**	**BDD Size**	**Time**	**BDD Size**	**Time**	**BDD Size**	**Time**	**BDD Size**	**Time**	**BDD Size**	**Time**
50	687	0.02	1,579	0.01	2,067	0.00	6,892	0.01	4,233	0.01	20,481	0.02
100	7,947	0.04	22,608	0.03	44,560	0.03	240,941	0.24	128,970	0.14	989,046	2.19
150	52,641	0.04	176,466	0.15	477,174	1.52	4,872,868	3.50	808,881	7.07	17,719,021	67.7
200	294,534	0.20	1,126,682	1.18	2,829,486	3.26	17,447,587	68.8	10,098,279	170.9	17,634,083	452.7
250	2,327,128	1.10	8,806,065	41.1	10,812,118	116.2	20,680,364	898.3	14,878,606	929.7	17,680,923	960.8
Max	**2,327,128**	**1.10**	**8,806,065**	**41.1**	**10,812,118**	**116.2**	**20,680,364**	**898.3**	**14,878,606**	**929.7**	**17,719,021**	**960.8**

Table 4: BDD size and computation time of the hardest feature models found using random and evolutionary search. Time in seconds.

much higher if we had not limited the time to make the experiment affordable. As expected, the superiority of ETHOM was also observed in the computation times required by each model. This suggests that our approach can also deal with optimisation criteria involving compilation and execution time in BDD solvers.

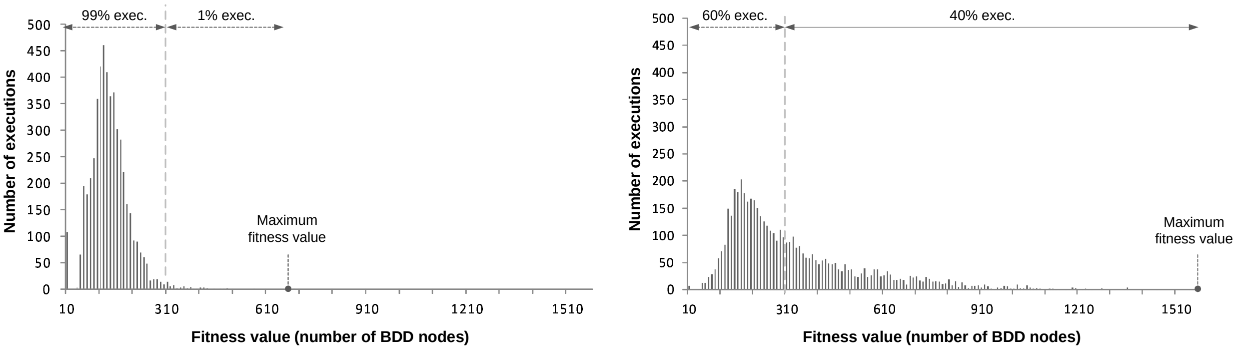
Fig. 10 shows the frequency with which each fitness value was found during the search. The data presented corresponds to the hardest feature models generated in the range of 50 features and 10% of cross-tree constraints. We chose this size range because it produced the smallest BDD sizes and facilitated the representation of the results using a common scale. For randomly generated models (Fig. 10(a)), a narrow curve is obtained with more than 99% of the executions producing fitness values under 310 BDD nodes. During evolutionary execution (Fig. 10(b)), however, a wider curve is obtained with 40% of the executions producing values over 310 nodes. Both histograms clearly show that ETHOM performed a more exhaustive search in a larger portion of the solution space than random search. This trend was also observed in the other size ranges.

5.3. Additional results and discussion

We performed some extra experiments reported in an external technical report due to space limitations [59]. Among other results, we studied the ability of ETHOM to generate input models maximising execution time in a propositional logic-based solver (a.k.a. SAT solver). The setup and results of this experiment were similar to those presented in Sections 5.1 and 5.2. The fitness of each model was measured as the number of decisions (i.e. steps) taken by the SAT solver when checking model consistency. In the experiment, our evolutionary approach succeeded in finding harder feature models than those generated randomly in 87.8% of the executions. We may remark, however, that the differences in the execution times obtained using random and evolutionary techniques were relatively small. This finding supports the results of Mendoca et al. [45] that show that checking the consistency of feature models with simple cross-tree constraints (i.e. those involving three features or less) using SAT solvers is highly efficient. We emphasise, however, that SAT solvers are not the optimum solution for all the analyses that can be performed on a feature model [10, 11, 51]. Previous studies show that CSP and BDD solvers are often better alternatives for certain operations and therefore experiments with these and others solvers are still necessary.

All the experiments performed suggested that ETHOM would find even better solutions if allowed to run longer. To check this, we reproduced Experiments #1 and #2, increasing the number of generations from 25 to 125. As expected, we found that the results provided by evolutionary search improved as the number of generations increased and did not reach a clear peak. In contrast, the results of random search showed little or no improvement at all. In the execution with the CSP solver, ETHOM produced a new maximum fitness of more than 77 million backtracks (computed in 27.5 minutes) while random search found a maximum value of only 1,603 backtracks (computed in 0.2 seconds). Similarly, the maximum fitness produced in our experiment with BDD and random search was 89,779 nodes, far from the best fitness obtained by our evolutionary program, 22.7 million nodes.

As part of our evaluation, we also studied the characteristics of the hardest feature models generated by ETHOM for each size range in the experiments with CSP, SAT and BDD solvers; the results are presented in Table 5. The data reveals that the models generated have a fair proportion of all relationships and constraints. This is interesting since ETHOM was free to remove any type of relationship or constraint from the model if this helped to make it harder, but this did not happen in our experiments. Recall that the only constraints imposed by our algorithm are those regarding the number of features, number of constraints and maximum branching factor. Another piece of evidence is that differences between the minimum and maximum percent-



a) Distribution of fitness values for random models

b) Distribution of fitness values for our evolutionary approach

Figure 10: Histograms with the distribution of fitness values for random and evolutionary techniques when searching for a feature model maximizing the size of the BDD.

ages of each modelling element are small. More importantly, the average percentages found are very similar to those of feature models found in the literature. In [61], She et al. studied the characteristics of 32 published feature models and reported that they contain, on average, 25% of mandatory features (between 17.1% and 27.9% in our models), 44% of set subfeatures[3] (between 37% and 46.3% in our models), 16% of set relationships[4] (between 13.8% and 16.1% in our models), 6% of or-relationships (between 7% and 8.9% in our models) and 9% of alternative relationships (between 6.7% and 7.2% in our study). As a result, we conclude that the models generated by our algorithm are by no means unrealistic. On the contrary, in the context of our study, they are a fair reflection of the realistic models found in the literature. This suggests that the long execution times and high memory consumption found by ETHOM might be reproduced when using real models with the consequent negative effect on the user.

Regarding the consistency of the models, the results are heterogeneous. On the one hand, we analysed all the models generated using ETHOM in our experiment with CSP and found that most of them are inconsistent (92.8%). That is, only 7.2% of the generated models represent at least one valid product. On the other hand, we found that 100% of the models generated using ETHOM in our experiments with SAT and BDD are consistent. This suggests that the consistency of the input models affects strongly but quite differently the performance of each solver. Also, it shows the ability of our algorithm to guide the search for hard feature models regardless of their consistency.

Our experimental results revealed that ETHOM is able to find smaller but much harder feature models than those found using random search. We also compared the results obtained in our experiments with the execution times and memory consumption produced by large randomly generated models. More specifically, we randomly generated 100 feature models with 10,000 features and 20% of CTCs and recorded the execution times taken by the CSP solver JaCoP to check their consistency. The results revealed an average execution time of 7.5 seconds and a maximum time of 8.1 seconds[5], far from the 27 minutes required by the hardest feature models found by ETHOM for 500-1000 features. Similarly, we generated 100 randomly generated feature models with 500 features and 10% of CTCs and recorded the size of the BDD generated when counting the number of products using the JavaBDD solver. The results revealed an average BDD size of 913,640 nodes and a maximum size of 17.2 million nodes, far from the 22 millions of BDD nodes reached by ETHOM in the range of 100 features [59]. These results clearly show the potential of ETHOM to find hard feature models of realistic size that are likely to reveal deficiencies in analysis tools rather than using large randomly generated models.

In another experiment, we checked whether the hard feature models generated by ETHOM were also hard for other tools and heuristics. In particular, we first checked whether the hardest feature models found in Experiment #1 using a CSP solver were also hard when using a SAT solver. The results showed, as expected, that all models

[3] Subfeatures in alternative an or-relationships

[4] Alternative and or-relationships

[5] Most of the time was taken by the translation from the feature model to a constraint satisfaction problem while the analysis itself was trivial. In fact, the maximum number of backtracks generated was 7.

Modelling element	CSP Solver Min	CSP Solver Avg	CSP Solver Max	SAT Solver Min	SAT Solver Avg	SAT Solver Max	BDD Solver Min	BDD Solver Avg	BDD Solver Max
% relative to no. of features									
Mandatory	25.3	27.9	31.0	20.0	25.1	28.0	10.0	17.1	24.8
Optional	27.5	34.9	45.0	30.5	36.9	44.0	18.0	35.7	46.5
Set subfeatures	29.0	37.0	41.5	31.0	37.8	45.5	34.5	46.3	62.0
Set relationships	11.0	14.1	16.0	12.0	13.8	15.3	13.3	16.1	20.0
- Or	5.5	7.0	9.0	5.5	7.1	8.3	6.0	8.9	12.0
- Alternative	5.5	7.1	8.5	4.0	6.7	8.8	3.3	7.2	10.0
% relative to no. of constraints									
Requires	31.3	47.5	56.6	41.1	51.9	68.4	31.0	48.5	64.3
Excludes	43.4	52.5	68.7	31.6	48.1	58.9	35.7	51.5	69.0

Table 5: Properties of the hardest feature models found in our experiments.

were trivially analysed in a few seconds. Then, we repeated the analysis of the hardest feature models found in Experiment #1 using the other seven heuristics available in the CSP solver JaCoP. The results revealed that the hardest feature models found in our experiment, using the heuristic *MostConstrainedDynamic*, were trivially solved by some of the others heuristics. For example, the hardest model in the range of 800 features and 10% CTC produced 5.3 million backtracks when using the heuristic *MostContrainedDynamic* and only 43 backtracks when using the heuristic *SmallestMin*. This finding clearly shows that feature models that are hard to analyse by one tool or technique could be trivially processed by others and vice-versa. Hence, we conclude that using a standard set of problems, randomly generated or not, is not sufficient for a full evaluation of the performance of different tools. Instead, as in our approach, the techniques and tools under evaluation should be exercised to identify their strengths and weaknesses providing helpful information for both users and developers.

The average effectiveness of our approach ranged from 85.8% to 94.4% in all the experiments. As expected from an evolutionary algorithm, we found that these variations in the effectiveness were caused by the characteristics of the search spaces of each problem. In particular, ETHOM behaves better when the search space is heterogeneous and there are many different fitness values, i.e. it is easy to compare the quality of the individuals. However, results get worse in homogeneous search spaces in which most fitness values are equal (e.g. Experiment #1, range of 10% of CTCs). A common strategy to alleviate this problem is to use a larger population, increasing the chances of the algorithm finding promising individuals during initialisation. We also found that the maximum timeout of 30 minutes was insufficient in some size ranges (e.g. Experiment #2, 250 features and 30% CTCs), adversely affecting the results. Increasing this timeout would have certainly increased the effectiveness of ETHOM at the price of making our experiments more time-consuming.

Finally, as a safety check, we tested ETHOM with different optimisation problems. In particular, we used problems with a known global maximum where the efficacy of ETHOM was easier to observe. For instance, we used ETHOM to search for feature models with n features and m% of CTCs that represent as many products as possible, 2^n being the maximum. Interestingly, the algorithm progressively removed the relationships constraining the set of products (i.e. mandatory and alternative), generating models with optional and or-relationships only. This demonstrates the ability of ETHOM to change the model if that helps to make it better for the given problem. This and other examples are available as a part of the BeTTy testing framework [14].

5.4. Statistical analysis

Statistical analysis is usually performed by formulating two contrary hypotheses. The first hypothesis is referred to as the *null hypothesis* (H_0^i) and says that the algorithm has no impact at all on the goodness of the results obtained, i.e. there is no difference between the results obtained by ETHOM and random search. Opposite to the null hypothesis, an *alternative hypothesis* (H_1^i) is formulated, stating that ETHOM has a significant effect in the quality of the results obtained. Statistical tests provide a probability (named *p-value*) ranging in [0,1]. A low p-value indicates that the null hypothesis is probably false and the alternative hypothesis is probably true, i.e. ETHOM works. Alternatively, high p-values suggest that ETHOM does not work. Researchers have established by convention that p-values under 0.05 or

0.01 are so-called *statistically significant* and are sufficient to reject the null hypothesis, i.e. demonstrate that ETHOM provides better results that random search. The statistical analysis described in this section was performed using the SPSS 17 statistical package [28].

The techniques used to perform the statistical analysis and obtain the p-values depend on whether the data follows a normal frequency distribution or not. After some preliminary tests (Kolmogorov-Smirnov [35, 63] and Shapiro-Wilk [60] tests) we concluded that our data did not follow a normal distribution and thus our tests required the use of so-called non–parametric techniques. In particular, we applied the Mann-Withney U non–parametric test [41] to the experimental results obtained with ETHOM and random search. Tables A.6 and A.7 show the results of these tests in SPSS for Experiments #1 and #2 respectively. For each number of features and percentage of cross-tree constraints, the values of the test are provided. As illustrated, the tests rejected the null hypotheses with extremely low p-values (zero in most cases) for nearly all experimental configurations of both experiments. This, coupled with the results shown in Section 5, clearly shows the superiority of our algorithm when compared to random search. As expected, statistical tests accepted some null hypotheses in the range of 10% of CTCs in Experiment #1. As explained in Section 6, this is due to the small complexity of the analysis on those models which made the fitness landscape extremely flat. Similarly, the tests accepted some null hypotheses in the range of 250 features and 30% of CTCs in Experiment #2. This was due to the maximum timeout of 30 minutes used for our experiments that made our algorithm stop prematurely, stopping it from evolving toward promising solutions.

For a more detailed explanation of our statistical analysis of the data we refer the reader to [59].

6. Threats to validity

In order to clearly delineate the limitations of the experimental study, next we discuss internal and external validity threats.

Internal validity. This refers to whether there is sufficient evidence to support the conclusions and the sources of bias that could compromise those conclusions. In order to minimise the impact of external factors in our results, ETHOM was executed 25 times for each problem to get averages. Moreover, statistical tests were performed to ensure significance of the differences identified. Regarding the random generation of feature models, we avoided the risk of creating syntactically incorrect models as follows. First, we used a publicly available (and previously used) algorithm for the random generation of feature models. Second, we performed several checks using the parser of BeTTy, FaMa and SPLOT to make sure that the generated models were syntactically correct and had the desired properties, e.g. a maximum branching factor. A related risk is the possibility of our random and evolutionary algorithms having different expressiveness, e.g. tree patterns that can be generated with ETHOM but not with our random algorithm. To minimise this risk, we imposed the same generation constraints on both our random and evolutionary generators. More specifically, both generators received exactly the same input constraints: number of features, percentage of CTC and maximum branching factor of the model to be generated. Also, both generators prohibit the generation of CTCs between features with parental relation and features linked by more than one CTC. A related limitation of the current ETHOM encoding is that it does not allow there to be more than one set relationship of the same type (e.g. alternative group) under a parent feature. Hence, for instance, if two alternative groups are located under the same feature, these are merged into one during decoding. We may remark, however, that this only affects the expressiveness of ETHOM putting it at a disadvantage against random search. Also, the results do not reveal any correlation between the number of set relationships and the hardness of the models which means that this restriction did not benefit our algorithm. Besides this, the results show that ETHOM is equally capable of generating consistent or inconsistent models if that make them harder for the target solver. Therefore, it seems unlikely that our algorithm has a tendency to generate only consistent or inconsistent models.

External validity. This is concerned with how the experiments capture the objectives of the research and the extent to which the conclusions drawn can be generalised. This can be mainly divided into limitations of the approach and generalizability of the conclusions.

Regarding the limitations, the experiments showed no significant improvements when using ETHOM with problems of low complexity, i.e. feature models with 10% of constraints in Experiment #1. As stated in Section 5.1, this limitation is due to the fitness landscape being relatively flat for simple problems; most fitness values are zero or close to zero. Another limitation of the experimental approach is that experiments for extremely hard feature models become too time consuming, e.g. feature models with 250 features in Experiment #2. This threat is caused by the nature of the hard feature models we intend to find, with the analysis of promising feature models becoming increasingly time consuming and memory intensive. We may remark, however, that this limitation is intrinsic to the problem of looking for hard feature models and thus it equally affects random search. Finally, we emphasise that in the worst case ETHOM behaves randomly equalling the strategies for the generation of hard feature models used in the current state of the art.

Regarding the generalisation of the conclusions, we used two different analysis operations and the results might not generalise further. We remark, however, that these operations are currently the most frequently quoted in the literature, have different complexity and, more importantly, are the basis for the implementation of many other analysis operations on feature models [10]. Thus, feature models that are hard to analyse for these operations would certainly be hard to analyse for those operations that use them as an auxiliary function making our results extensible to other analyses. Similarly, we only used two analysis tools for the experiments, FaMa and SPLOT. However, these tools are developed and maintained by independent laboratories providing a sufficient degree of heterogeneity for our study. Also, the results revealed that a number of metrics for the generated models (e.g. percentage of CTCs) were in the ranges observed in realistic models found in the literature, which supports the realism of the hard feature models being generated. We may remark, however, that these models could still contain structures that are unlikely in real-world models and therefore this issue requires further research. Finally, our random and evolutionary generators do not allow two features to be linked by more than one CTC for simplicity (see Section 4). This implicitly prohibits the generation of cycles of requires constraints, i.e. $A- > B$ and $B- > A$. However, these cycles express equivalence relationships and seem to appear in real models (e.g. Linux kernel feature model [49]) which could slightly affect the generalisation of our results. These cycles will be allowed in future versions of our algorithm.

7. Related work

In this section we discuss related work in the areas of software product lines and search-based testing.

7.1. Software product lines

A number of authors have used realistic feature models to evaluate their tools [4, 9, 24, 26, 31, 33, 46, 45, 50, 51, 55, 64, 67, 70]. By *realistic* models we mean those modelling real–world domains or a simplified version of them. Some of the realistic feature models most quoted in the literature are e-Shop [36] with 287 features, graph product line [38] with up to 64 features and BerkeleyDB [34] with 55 features. Although there are reports from industry of feature models with hundreds or even thousands of features [7, 37, 66], only a portion of them is typically published. This has led authors to generate feature models automatically to show the scalability of their approaches with large problems. These models are generated either randomly [12, 11, 22, 26, 44, 47, 57, 74, 75, 76, 78, 79] or using a process that tries to produce models with the properties of those found in the literature [23, 45, 64, 67]. More recently, some authors have suggested looking for tough and realistic feature models in the open source community [13, 21, 49, 61, 62]. As an example, She et al. [62] extracted a feature model from the Linux kernel containing more than 5,000 features and compared it with publicly available realistic feature models.

Regarding the size of the models used for experiments, there is a clear tendency for model size to increase: this ranges from the model with 15 features used in 2004 [8] to models with up to 10,000 and 20,000 features used in recent years [23, 45, 47, 67, 74]. These findings reflect an increasing interest in using complex feature models in performance evaluation. This also suggests that the only mechanism used to increase the complexity of the models is by increasing size. When compared to previous work, our approach is the first to use a search–based strategy to reveal the performance weaknesses of the tools and techniques under evaluation rather than simply using large randomly generated models. This allows developers to focus on the search for tough models of realistic size that could reveal deficiencies in their tools rather than using huge feature models out of their scope. Similarly, users could have more information about the expected behaviour of the tools in pessimistic cases helping them to choose the tool or technique that best meets their needs.

The application of optimisation algorithms in the context of software product lines has been explored by several authors. Guo et al. [23] proposed a genetic algorithm called GAFES for optimised feature selection in feature models, e.g. selecting the set of features that minimises the total cost of the product. Sayyad et al. [55] compared the effectiveness of five multi-objective optimization algorithms for the selection of optimised products. Other authors [25, 39, 71] have proposed algorithms for the selection of test suites (i.e. set of products) maximising or minimising certain preferences, e.g. feature coverage. Compared to their work, our approach differs in several aspects. First, our work addresses a different problem domain, hard feature model generation. Second, and more importantly, ETHOM searches for optimum feature models while related algorithms search for optimum product configurations. This means that ETHOM and related algorithms bear no resemblance and face completely different challenges. For instance, related algorithms use a standard binary encoding to represent product configurations while ETHOM uses a custom array encoding to represent feature models of fixed size.

Pohl et al. [51] presented a performance comparison of nine CSP, SAT and BDD solvers on the automated analysis of feature models. As input problems, they used 90 realistic feature models with up to 287 features taken from the SPLOT repository [65]. The longest execution time found in the consistency operation was 23.8 seconds, far from the 27.5 minutes found in our work. Memory consumption was not evaluated. As part of their work, the authors tried to find correlations between the properties of the models and the performance of the solvers. Among other results, they identified an exponential runtime increase with the number of features in CSP and SAT solvers. This is not supported by our results, at least not in general, since we found feature models producing much longer execution times than larger randomly generated models. Also, the authors mentioned that SAT and CSP solvers provided a similar performance in their experiment. This was not observed in our work in which the SAT solver was much more efficient than the CSP solver, i.e. random and evolutionary search were unable to find hard problems for SAT. Overall, we consider that using realistic feature models is helpful but not sufficient for an exhaustive evaluation of the performance of solvers. In contrast, our work provides the community with a limitless source of motivating problems to explore the strengths and weaknesses of analysis tools.

In later work, Pohl et al. [52] proposed using width measures from graph theory to characterise the structural complexity of feature models as a way to estimate the difficulty in analysing them. They performed several experiments running the consistency operation on randomly generated models of up to 1,000 features in nine state of the art CSP, SAT and BDD solvers. As a result, for some of the solvers they found a correlation between one of the metrics and the time taken by the analysis. When compared to their work, ETHOM uses a black-box strategy and thus it may be used to find hard input feature models for any analysis tool or analysis operation regardless of their implementation details. Furthermore, ETHOM explores the whole search space of feature models, not only those with different width properties, in looking for input problems that increase the execution times of analysis tools. Having said this, we think that both works are complementary since ETHOM generates hard feature models and their approach tries to determine what makes the models hard to analyse.

During the preparation of this article, we presented a novel application of ETHOM in the context of reverse engineering of feature models [40]. More specifically, we used ETHOM to search for a feature model that represents a specific set of products provided as input. The results showed that within a few generations our algorithm was able to find feature models that represent a superset of the desired products. This contribution supports our claims about the generalisability of our algorithm showing its applicability to other domains beyond the analysis of feature models.

Finally, we would like to remark that our approach does not intend to replace the use of realistic or randomly generated models which can be used to evaluate the average performance of analysis techniques. Instead, our work complements previous approaches enabling a more exhaustive evaluation of the performance of analysis tools using hard problems.

7.2. Search-based testing

Regarding related work in search-based testing, Wegener et al. [72] were the first to use genetic algorithms to search for input values that produce very long or very short execution times in the context of real time systems. In their experiments, they used C programs receiving hundreds or even thousands of integer parameters. Their results showed that genetic algorithms obtained more extreme execution times with equal or less test effort than random testing. Our approach may be considered a specific application of the ideas of Wegener and later authors to the domain of feature modelling. In this sense, our main contribution is the development and configuration of a novel evolutionary algorithm to deal with optimisation problems on feature models and its application to performance testing in this domain.

Many authors continued the work of Wegener et al. in the application of metaheuristic search techniques to test non-functional properties such as execution time, quality of service, security, usability or safety [2]. The techniques used by the search-based testing community include, among others, hill climbing, ant colony optimisation, tabu search and simulated annealing. In our approach, we used evolutionary algorithms inspired by the work of Wegener et al. and their promising results in a related optimisation problem, i.e. generation of input values maximising the execution time in real time systems. We remark, however, that the use of other metaheuristic techniques for the generation of hard feature models is a promising research topic that requires further study.

Genetic Algorithms (GAs) [1] are a subclass of evolutionary algorithms in which solutions are encoded using bit strings. However, it is difficult to encode the hierarchical structure of feature models using this approach and therefore we discarded their use. Genetic Programming (GP) is another variant of evolutionary algorithms in which solutions are encoded as trees [54]. This encoding is commonly used to represent programs whose abstract syntax can be naturally represented hierarchically. Crossover in GP is applied on an individual by switching one of its branches with another branch from another individual in the population, i.e. individuals can have different sizes. We identified several factors that make GPs unsuitable for our problem. First, the classic tree encoding does not consider cross-tree constraints as in feature models. As a result, crossover would probably generate many dangling edges which may require costly repairing heuristics. Second, and more importantly, crossover in GP does not guarantee a fixed size for the solution which was a key constraint in our work. These reasons led us to design a custom evolutionary algorithm, ETHOM, supporting the representation of feature trees of fixed size with cross-tree constraints.

7.3. Performance evaluation of CSP and SAT solvers

CSP and SAT solvers (hereinafter, CP solvers) use algorithms and techniques of Constraint Programming (CP) to solve complex problems from domains such as computer science, artificial intelligence or hardware design[6]. The underlying problems of CSP and SAT solvers are NP-complete and so CSP and SAT solvers have an exponential worst case runtime. This makes efficiency a crucial matter for these types of tools. Hence, there exist a number of available benchmarks to evaluate and compare the performance of CP solvers [27]. Also, several competitions are held every year to rank the performance of the participants' tools. As an example, 93 solvers took part in the SAT competition[7] in 2013.

CP solvers use three main types of problems for performance evaluation: problems from realistic domains (e.g. hardware design), randomly generated problems and hard problems. Both randomly generated and hard problems are automatically generated and are often forced to have at least one solution (i.e. be satisfiable). The CP research community realised long ago that there are benefits in using hard problems to test the performance of their tools. In 1997, Cook and Mitchell [17] presented a survey on the strategies to find hard SAT instances proposed so far. In their work, the authors warned about the importance of generating hard problems for understanding their complexity and for providing challenging benchmarks. Since then, many other contributions have explored the generation of hard SAT and CSP problems [5, 77].

A common strategy to generate hard CSP and SAT problems is by exploiting what is known as the *phase transition phenomenon* [77]. This phenomenon establishes that for many NP-complete problems the hardest instances occur between the region in which most problems are satisfiable and the region in which most problems are unsatisfiable. This happens because for these problems the solver has to explore the search space in depth before finding out whether the problem is satisfiable or not. CSP and SAT solvers can be parametrically guided to search in the phase transition region enabling the systematic generation of hard problems. We are not aware of any work using evolutionary algorithms for the generation of hard CP problems.

When compared to CP problems, the analysis of feature models differs in several ways. First, CSP and SAT are related problems within the constraint programming paradigm. The analysis of feature models, however, is a high-level problem usually solved using quite heterogeneous approaches such as constraint programming, description logic, semantic web technologies or ad-hoc algorithms [10]. Also, CP solvers focus on a single analysis operation (i.e. satisfiability) for which there exist a number of well known algorithms. In the analysis of feature models, however, more than 30 analysis operations have been reported. In this scenario, we believe that our approach may help the community to generate hard problems and study their complexity, leading to a better understanding of the analysis operations and the performance of analysis tools.

We identified two main advantages in our work when compared to the systematic generation of hard CP problems. First, our approach is generic and can be applied to any tool, algorithm or analysis operation for the automated treatment of feature models. Second, our algorithm is free to explore the whole search space looking for input models that reveal performance vulnerabilities. In contrast, CP related work focuses the search for inputs problem in a specific area (the transition phase region).

[6] A SAT problem can be regarded a subclass of CSP with only boolean variables.

[7] `http://www.satcompetition.org`

Overall, we conclude that related work in CP support our approach for the generation of hard feature models as a way to evaluate the performance strengths and weakness of feature model analysis tools.

8. Conclusions and future work

In this paper, we presented ETHOM, a novel evolutionary algorithm to solve optimisation problems on feature models and showed how it can be used for the automated generation of computationally hard feature models. Experiments using our evolutionary approach on different analysis operations and independent tools successfully identified input models producing much longer executions times and higher memory consumption than randomly generated models of identical or even larger size. In total, more than 50 million executions of analysis operations were performed to configure and evaluate our approach. This is the first metaheuristic-based strategy to guide the search for computationally hard feature models rather than simply using randomly generated models. This approach will allow developers to focus on the search for tough models of realistic size that could reveal deficiencies in their tools rather than using huge randomly generated feature models out of the scope of their tools. Similarly, users are provided with more information about the expected behaviour of the tools in pessimistic cases, helping them to choose the tool or technique that better meets their needs. Contrary to general belief, we found that model size has an important, but not decisive, effect on performance. Also, we found that the hard feature models generated by ETHOM had similar properties to realistic models found in the literature. This means that the long execution times and high memory consumption found by our algorithm might be reproduced in real scenarios with the consequent negative effect on the user. In view of the positive results obtained, we expect this work to be the seed for many other research contributions exploiting the benefits of ETHOM in particular, and evolutionary computation in general, on the analysis of feature models. In particular, we envision two main research directions to be explored by the community in the future, namely:

- **Algorithms development.** The combination of different encodings, selection techniques, crossover strategies, mutation operators and other parameters may lead to a whole new variety of evolutionary algorithms for feature models to be explored. Also, the use of other metaheuristic techniques (e.g. ant colony optimisation) is a promising topic that need further study. The development of more flexible algorithms would be desirable in order to deal with other feature modelling languages (e.g. cardinality-based feature models) or stricter structural constraints, e.g. enabling the generation of hard models with a given percentage of mandatory features. Also, the generation of feature models with complex cross-tree constraints (those involving more than two features) remains an open challenge that we intend to address in our future work.

- **Applications.** Further applications of our algorithm are still to be explored. Some promising applications are those dealing with the optimisation of non–functional properties in other analysis operations or even different automated treatments, e.g. refactoring feature models. The application of our algorithm to minimisation problems is also an open issue in which we have started to obtain promising results. Additionally, it would be nice to apply our approach to verify the time constraints of real time systems dealing with variability like those of mobile phones or context–aware pervasive systems. Last, but not least, we plan to study the hard feature models generated and try to understand what makes them hard to analyse. From the information obtained, more refined applications and heuristics could be developed leading to more efficient tool support for the analysis of feature models.

A Java implementation of ETHOM is ready-to-use and publicly available as a part of the open-source BeTTy Framework [14, 58].

Material

The prototype implementation of ETHOM, hard feature models generated (in XML format), statistical results (in SPSS format) and raw experiment data are available at `http://www.lsi.us.es/~segura/files/material/ESWA13/`.

Acknowledgments

We would like to thank Dr. Don Batory, Dr. Javier Dolado, Dr. Arnaud Gotlieb, Dr. Andreas Metzger, Dr. Jose C. Riquelme, Dr. David Ruiz and Dr. Javier Tuya whose comments and suggestions helped us to improve the article substantially. We would also like to thank José A. Galindo for his work integrating ETHOM into the framework BeTTy.

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT projects SETI (TIN2009-07366) and TAPAS (TIN2012-32273) and the Andalusian Government projects THEOS (TIC-5906) and COPAS (P12-TIC-1867).

Appendix A. Statistical analysis results

#Features	CTC (%)			
	10	20	30	40
200	0.53	0	0	0
400	0.28	0	0	0
600	0.36	0	0	0
800	0	0	0	0
1000	0.12	0	0	0

Table A.6: p-values obtained in Experiment #1 using the Mann-Whitney-Wilcoxon test

#Features	CTC (%)		
	10	20	30
50	0	0	0
100	0	0	0
150	0	0	0
200	0	0	0
250	0	0	0.85

Table A.7: p-values obtained in Experiment #2 using the Mann-Whitney-Wilcoxon test

References

[1] M. Affenzeller, S. Wagner, S. Winkler, and A. Beham. *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*. Numerical Insights. Taylor & Francis, 2009.

[2] W. Afzal, R. Torkar, and R. Feldt. A systematic review of search-based testing for non-functional system properties. *Information and Software Technology*, 51(6):957–976, 2009.

[3] AHEAD Tool Suite. `http://www.cs.utexas.edu/users/schwartz/ATS.html`, accessed July 2013.

[4] N. Andersen, K. Czarnecki, S. She, and A. Wasowski. Efficient synthesis of feature models. In *16th International Software Product Line Conference*, pages 106–115, 2012.

[5] C. Ansotegui, R. Bejar, C. Fernandez, and C. Mateu. Edge matching puzzles as hard SAT/CSP benchmarks. In P. Stuckey, editor, *Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, pages 560–565. Springer Berlin / Heidelberg, 2008.

[6] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Sciences*, pages 7–20. Springer–Verlag, 2005.

[7] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, December:45–47, 2006.

[8] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Coping with automatic reasoning on software product lines. In *Proceedings of the 2nd Groningen Workshop on Software Variability Management*, November 2004.

[9] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. In *17th International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3520 of *Lecture Notes in Computer Sciences*, pages 491–503. Springer–Verlag, 2005.

[10] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615 – 636, 2010.

[11] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, 2006.

[12] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using Java CSP solvers in the automated analyses of feature models. *LNCS*, 4143:389–398, 2006.

[13] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 73–82. ACM, 2010.

[14] BeTTy Framework. `http://www.isa.us.es/betty`, accessed July 2013.

[15] BigLever. Biglever software gears. `http://www.biglever.com/`, accessed July 2013.

[16] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison–Wesley, August 2001.

[17] S.A. Cook and D.G. Mitchell. Finding hard instances of the satisfiability problem: A survey. In *Satisfiability Problem: Theory and Applications*, volume 35 of *Dimacs Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–17. American Mathematical Society, 1997.

[18] A.E. Eiben and S.K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 1(1):19 – 31, 2011.

[19] FaMa Tool Suite. `http://www.isa.us.es/fama/`, accessed July 2013.

[20] Feature Modeling Plug-in. `http://gp.uwaterloo.ca/fmp/`, accessed July 2013.

[21] J.A. Galindo, D. Benavides, and S. Segura. Debian packages repositories as software product line models. Towards automated analysis. In *Proceedings of the 1st International Workshop on Automated Configuration and Tailoring of Applications (ACoTA)*, Antwerp, Belgium, 2010.

[22] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in Alloy. In *Proceedings of the ACM SIGSOFY First Alloy Workshop*, pages 71–80, Portland, United States, nov 2006.

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

[24] A. Hemakumar. Finding contradictions in feature models. In *First International Workshop on Analyses of Software Product Lines (ASPL)*, pages 183–190, 2008.

[25] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y.L. Traon. Multi-objective test generation for software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 62–71, New York, NY, USA, 2013. ACM.

[26] R. Heradio-Gil, D. Fernandez-Amoros, J.A. Cerrada, and C. Cerrada. Supporting commonality-based analysis of software

product lines. *Software, IET*, 5(6):496 –509, dec. 2011.

[27] H. Hoos and T. Stutzle. SATLIB: An online resource for research on SAT. In I.P. van Maaren, H. Gent, and T. Walsh, editors, *Sat2000: Highlights of Satisfiability Research in the Year 2000*, pages 283–292. IOS Press, 2000.

[28] IBM. SPSS 17 Statistical Package. `http://www.spss.com/`, accessed November 2010.

[29] JaCoP. `http://jacop.osolpro.com/`, accessed July 2013.

[30] JavaBDD. `http://javabdd.sourceforge.net/`, accessed July 2013.

[31] M.F. Johansen, Ø. Haugen, and F. Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *16th International Software Product Line Conference*, pages 46–55, 2012.

[32] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature–Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[33] A. Karatas, H. Oguztüzün, and A. Dogru. Global constraints on feature models. In D. Cohen, editor, *Principles and Practice of Constraint Programming*, volume 6308 of *Lecture Notes in Computer Science*, pages 537–551, 2010.

[34] C. Kastner, S. Apel, and D. Batory. A case study implementing features using AspectJ. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society.

[35] A. Kolmogorov. Sulla determinazione empirica di una legge di distribuzione. *G. Inst. Ital. Attuari*, 4:83, 1933.

[36] S.Q. Lau. Domain analysis of e-commerce systems using feature–based model templates. master's thesis. Dept. of ECE, University of Waterloo, Canada, 2006.

[37] F. Loesch and E. Ploedereder. Optimization of variability in software product lines. In *Proceedings of the 11th International Software Product Line Conference (SPLC)*, pages 151–162, Washington, DC, USA, 2007. IEEE Computer Society.

[38] R.E Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *GCSE '01: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, pages 10–24, London, UK, 2001. Springer-Verlag.

[39] R.E. Lopez-Herrejon, F. Chicano, J. Ferrer, A. Egyed, and E. Alba. Multi-objective optimal test suite computation for software product line pairwise testing. In *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013.

[40] R.E. Lopez-Herrejon, J.A. Galindo, D. Benavides, S. Segura, and A. Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *Search Based Software Engineering*, volume 7515 of *Lecture Notes in Computer Science*, pages 168–182. Springer Berlin Heidelberg, 2012.

[41] H.B. Mann and D.R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Stat.*, 18:50–60, 1947.

[42] P. McMinn. Search-based software test data generation: a survey. *Software Testing Verification and Reliability.*, 14(2):105–156, 2004.

[43] M. Mendonca, M. Branco, and D. Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Companion to the 24th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 761–762, Orlando, Florida, USA, October 2009. ACM.

[44] M. Mendonca, D.D. Cowan, W. Malyk, and T. Oliveira. Collaborative product configuration: Formalization and efficient algorithms for dependency analysis. *Journal of Software*, 3(2):69–82, 2008.

[45] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT–based analysis of feature models is easy. In *Proceedings of the International Sofware Product Line Conference (SPLC)*, 2009.

[46] M. Mendonca, A. Wasowski, K. Czarnecki, and D.D. Cowan. Efficient compilation techniques for large scale feature models. In *7th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 13–22, 2008.

[47] A. Osman, S. Phon-Amnuaisuk, and C.K. Ho. Using first order logic to validate feature model. In *Third International Workshop on Variability Modelling in Software-intensive Systems (VaMoS)*, pages 169–172, 2009.

[48] J.A.. Parejo, A. Ruiz-Cortés, S. Lozano, and P. Fernandez. Metaheuristic optimization frameworks: a survey and benchmarking. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 16:527–561, 2012.

[49] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wasowski. A study of non-boolean constraints in variability models of an embedded operating system. In *Third International Workshop on Feature-Oriented Software Development (FOSD)*, SPLC '11, pages 2:1–2:8. ACM, 2011.

[50] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry, and Y. Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20:605–643, 2012.

[51] R. Pohl, K. Lauenroth, and K. Pohl. A performance comparison of contemporary algorithmic approaches for automated analysis operations on feature models. In *26th International Conference on Automated Software Engineering*, pages 313–322. IEEE, 2011.

[52] R. Pohl, V. Stricker, and K. Pohl. Measuring the structural complexity of feature models. In *28th International Conference on Automated Software Engineering*, pages 454–464. IEEE, 2013.

[53] pure::variants. `http://www.pure-systems.com/`, accessed July 2013.

[54] F. Rothlauf. *Representations for Genetic and Evolutionary Algorithms*. Springer, 2nd edition, 2012.

[55] A.S. Sayyad, T. Menzies, and H. Ammar. On the value of user preferences in search-based software engineering: A case study in software product lines. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 492–501, Piscataway, NJ, USA, 2013. IEEE Press.

[56] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Minneapolis, Minnesota, USA, September 2006.

[57] S. Segura. Automated analysis of feature models using atomic sets. In *First Workshop on Analyses of Software Product Lines (ASPL)*, pages 201–207, Limerick, Ireland, September 2008.

[58] S. Segura, J.A. Galindo, D. Benavides, J.A. Parejo, and A. Ruiz-Cortés. BeTTy: Benchmarking and Testing on the Automated Analysis of Feature Models. In U.W. Eisenecker, S. Apel, and S. Gnesi, editors, *Sixth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'12)*, pages 63–71, Leipzig, Germany, 2012. ACM.

[59] S. Segura, J.A. Parejo, R.M. Hierons, D. Benavides, and A. Ruiz-Cortés. ETHOM: An evolutionary algorithm for optimized feature models generation (v1.3). Technical Report ISA-2013-TR-01, Applied Software Engineering Research Group, Seville, Spain, 2013. `http://www.isa.us.es/sites/default/files/HardFMUsingEA_1.pdf`.

[60] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):pp. 591–611, 1965.

[61] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. The variability model of the linux kernel. In *Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Linz, Austria, January 2010.

[62] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Reverse engineering feature models. In *Proceeding of the 33rd International Conference on Software Engineering*, pages 461–470. ACM, 2011.

[63] N. V. Smirnov. Tables for estimating the goodness of fit of empirical distributions. *Annals of Mathematical Statistic*, 19:279, 1948.

[64] S. Soltani, M. Asadi, D. Gasevic, M. Hatala, and E. Bagheri. Automated planning for feature model configuration based on functional and non-functional requirements. In *16th International Software Product Line Conference*, pages 56–65, 2012.

[65] S.P.L.O.T.: Software Product Lines Online Tools. `http://www.splot-research.org/`, accessed July 2013.

[66] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber. Introducing PLA at Bosch gasoline systems: Experiences and practices. In *International Sofware Product Line Conference (SPLC)*, pages 34–50, 2004.

[67] T. Thüm, D. Batory, and C. Kästner. Reasoning about edits to feature models. In *International Conference on Software Engineering*, pages 254–264, 2009.

[68] Edward Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1995.

[69] S. Voß. Meta-heuristics: The state of the art. In *ECAI '00: Proceedings of the Workshop on Local Search for Planning and Scheduling-Revised Papers*, pages 1–23. Springer-Verlag, London, UK, 2001.

[70] H.H. Wang, Y.F. Li, J. Sun, H. Zhang, and J. Pan. Verifying feature models using OWL. *Journal of Web Semantics*, 5:117–129, June 2007.

[71] S. Wang, S. Ali, and A. Gotlieb. Minimizing test suites in software product lines using weight-based genetic algorithms. In *Proceeding of the Fifteenth Annual Conference on Genetic and Evolutionary Computation Conference*, GECCO '13, pages 1493–1500, New York, NY, USA, 2013. ACM.

[72] J. Wegener, K. Grimm, M. Grochtmann, and H. Sthamer. Systematic testing of real-time systems. In *Proceedings of the Fourth International Conference on Software Testing and Review (EuroSTAR)*, 1996.

[73] J. Wegener, H. Sthamer, B.F. Jones, and D.E. Eyres. Testing real-time systems using genetic algorithms. *Software Quality Control*, 6(2):127–135, 1997.

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

[75] J. White, B. Doughtery, D. Schmidt, and D. Benavides. Automated reasoning for multi-step software product-line configuration problems. In *Proceedings of the Sofware Product Line Conference*, pages 11–20, 2009.

[76] J. White, D. Schmidt, D. Benavides P. Trinidad, and Ruiz-Cortés. Automated diagnosis of product-line configuration errors in feature models. In *Proceedings of the 12th Sofware Product Line Conference (SPLC)*, Limerick, Ireland, September 2008.

[77] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. Random constraint satisfaction: Easy generation of hard (satisfiable) instances. *Artificial Intelligence*, 171(8-9):514–534, 2007.

[78] H. Yan, W. Zhang, H. Zhao, and H. Mei. An optimization strategy to feature models' verification by eliminating verification-irrelevant features and constraints. In *ICSR*, pages 65–75, 2009.

[79] W. Zhang, H. Yan, H. Zhao, and Z. Jin. A BDD–based approach to verifying clone-enabled feature models' constraints and customization. In *10th International Conference on Software Reuse (ICSR)*, LNCS, pages 186–199. Springer, 2008.