ETHOM: An Evolutionary Algorithm for Optimized Feature Models Generation (v. 1.3)

TECHNICAL REPORT ISA-2013-TR-01

Sergio Segura and J. A. Parejo
Software Engineering School
University of Seville
{japarejo,sergiosegura}@us.es

Robert M. Hierons
School of Information Systems, Computing and Mathematics
Brunel University
Rob.Hierons@brunel.ac.uk

David Benavides, and Antonio Ruiz-Cortés
Software Engineering School
University of Seville
{benavides,aruiz}@us.es

Applied Software Engineering Research Group
http://www.isa.us.es

University of Sevilla
http://www.us.es

This work is partially supported by:

Ministerio de Ciencia e Innovación, Gobierno de España

Consejería Innovación, Ciencia y Empresa de la Junta de Andalucía

July 11, 2013

Contents

A *feature model* defines the valid combinations of features in a domain. The automated extraction of information from feature models is a thriving topic involving numerous analysis operations, techniques and tools. The progress of this discipline is leading to an increasing concern to test and compare the performance of analysis solutions using tough input models that show the behaviour of the tools in extreme situations (e.g. those producing longest execution times or highest memory consumption). Currently, these feature models are generated randomly ignoring the internal aspects of the tools under tests. As a result, 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 technical report, we model the problem of finding computationally–hard feature models as an optimization problem and we solve it using a novel evolutionary algorithm. Given a tool and an analysis operation, our algorithm generates input models of a predefined size maximizing aspects 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 behaviour of tools in pessimistic cases providing a better idea of their real power. Experiments using our evolutionary algorithm on a number of analysis operations and tools have successfully identified input models causing much longer executions times and higher memory consumption than random models of identical or even larger size. Our solution is generic and applicable to a variety of optimization problems on feature models, not only those involving analysis operations. In view of the positive results, we expect this work to be the seed for a new wave of research contributions exploiting the benefit of evolutionary programming in the field of feature modelling.

1. Introduction

A *Software Product Line* (SPL) is a family of related software systems that share a set of features [18]. The development of SPLs is based on two key ideas. First, most software systems are not new. In fact, software products usually share a number of features. Consider, for instance, the common features (a.k.a. commonalities) that can be found in current on-line purchase systems (e.g. catalogue management, credit card validation, data access, etc.). Second, many industries are adopting a mass-customization paradigm in which different variants of the same products are launched to the market by combining different features in response to different customers' preferences. As an example, consider the software loaded in mobile phones in which hundreds of different models are built by combining a common set of reusable features: calls, messaging, Bluetooth, MP3, 3G, games, etc. Based on these ideas, SPL engineering focuses on the systematic development of families of software products rather than producing each product one by one from scratch. To this aim, product variants are built using reusable assets which usually include frameworks and components.

The products of an SPL are defined in terms of features where a *feature* is any incre-

ment in product functionality [7]. A key aspect in SPLs is to capture the commonalities and variabilities of the systems that belong to the product line. This is commonly done by using a so-called feature model. A *feature model* [36] is a compact representation of all the products of an SPL in terms of features and relationships among them (see 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 caught the attention of researchers for the last twenty years [11]. Typical operations of analysis allow us to know whether a feature model is consistent (i.e. it represents at least one product), what is 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 identified in the literature have been reported [11]. Common techniques to perform these operations are those based on propositional logic [7, 49], constraint programming [10, 80] or description logic [75]. Also, these analysis capabilities can be found in several commercial and open source tools including *AHEAD Tool Suite* [3], *Big Lever Software Gears* [16], *FaMa Framework* [20], *Feature Model Plug-in* [21], *pure::variants* [57] and SPLOT [47].

The development of tools and benchmarks to evaluate the performance and scalability of feature model analysis tools has been recognized as a challenge [8, 11, 56, 66]. Also, recent publications reflect an increasing interest in evaluating and comparing the performance of techniques and tools for the analysis of feature models [4, 28, 29, 35, 37, 49, 55, 56, 68, 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 [8, 42, 70], only small examples from research publications or case studies are usually available. This lack of hard realistic feature models, has led authors to evaluate their tools with large-scale randomly generated feature models of 5,000 [50, 80], 10,000 [26, 49, 71, 78] and up to 20,000 [52] features. More recently, some authors have suggested looking for hard and realistic feature models in the open source community [14, 22, 54, 65, 66]. For instance, She et al. [66] extracted a feature model from the Linux kernel containing more than 5,000 features.

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 realized long ago that random values are not effective in revealing the vulnerabilities of the systems under tests. As pointed out by McMinn [46]: *"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 [46] and non–functional properties [2]. *Metaheuristic search techniques* are frameworks which use heuristics to find solutions to hard problems at an affordable computational cost. Typical metaheuristic techniques are evolutionary algorithms, hill climbing or simulated annealing [74]. For the generation of test data, these strategies translate the test criterion into an objective function (also called 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. [76, 77] were one of the first proposing 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 maximizing or minimizing execution times. Since then, a number of authors have followed their steps using metaheuristics and especially evolutionary algorithms for the testing of non–functional properties such as execution time, quality of service, security, usability or safety [2, 46].

Problem description. Current performance evaluations on the analysis of feature models are mainly carried out using random 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 specific mechanisms that take analysis tools to their limits and reveal their real potential in terms of performance. This problem has negative implications for both tools 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 hindering them from choosing the tool that better meets their needs. Hence, for instance, a user could choose a tool based on its average performance and later realize that it performs very badly in particular cases that appear frequently in its 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 been traditionally addressed by the SPL community by simply generating huge random feature models with thousand of features and constraints. That is, it is generally assumed that the larger the model the harder its analysis. However, we remark that these models are still random and therefore, as warned by software testing experts, they are not sufficient to exercise the specific features of the tools 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 their tool may crash with a hard model of small or medium size rather than knowing the execution times of huge random models out of their scope.

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

Solution overview and contributions. In this article, we propose the use of evolutionary algorithms for the automated generation of hard feature models. In particular, 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 gen-

erates input models of a predefined size maximizing 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 optimization 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. The results showed how ETHOM successfully identified input models causing much longer executions times and higher memory consumption than random 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 maximizing the time required by a constraint programming solver (a.k.a. CSP solver) to check their consistency. The results revealed that the hardest random model found required 0.2 seconds to be analyzed meanwhile ETHOM was able to find several models taking between 1 and 27.5 minutes to be processed. Not only that, we found the hardest feature models generated by ETHOM in the ranges 500-1,000 features were remarkably harder to process than random models with 10,000 features. More importantly, we found that the hard feature models generated by ETHOM had similar properties to the realistic models found in the literature. This suggests that the long execution times and high memory consumption detected by ETHOM might be therefore reproduced when using real models with the consequent negative effect on the user.

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

- To the best of our knowledge, this is the first approach using 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 random models.
- Our work allows developers to focus on the search for computationally–hard models of realistic size that could reveal deficiencies in their tools rather than using huge feature models out of their scope.
- Our approach provides users with helpful information about the behaviour of tools in pessimistic cases helping them to choose the tool that better meets their needs. This is especially helpful in the so-called *Dynamic Software Product Lines* [27] in which the analysis of feature models is performed in real time domains such as mobile devices [81] or autonomic computing [17, 23] where short runtimes are required.
- Our algorithm is highly generic being applicable to any automated operation on feature models in which the quality (i.e. fitness) of the models with respect to an optimization criteria can be measured quantitatively.
- Our experimental results show that the hardness of feature models depends on different factors in contrast to related works in which the complexity of the models is mainly associated to their size. Although this is generally true, our work demystifies the belief that large models have to be necessarily harder to process than small ones.

- Our algorithm is ready-to-use and publicly available as a part of the open-source BeTTy Framework [15, 61].

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

- Our work follows a black-box approach. This is, our algorithm does not make any assumption about the analysis tools and operations under test. This makes ETHOM able to generate hard feature models for any tool or analysis operation regardless of how it is implemented.
- Our approach focuses on testing, not debugging. That is, our goal is to support 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, tools' developers and designers should identify the fault causing it using debugging. Typical debugging techniques are execution tracing, test case simplification (so-called delta-debugging) or assertion checking.
- It is noteworthy that there are many possible causes of the hardness of a feature model, some of them not directly related to the analysis algorithms: bad variable ordering, bad problem encoding, parsing problems, bad heuristic selection, etc. However, as previously mentioned, identifying the reasons that make a feature model hard to analyse by 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 3.1, we present ETHOM, an evolutionary algorithm for the generation of optimized feature models. Then, in Section 3.2, 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 4. Section 5 presents the threats to validity of our work. The related works are presented and discussed in Section 6. Finally, we summarize our conclusions and describe our future work in Section 7.

2. Preliminaries

2.1. Feature models

A *feature model* defines the valid combination of features in a domain. These are commonly used as a compact representation of all the products of an SPL in terms of features. 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

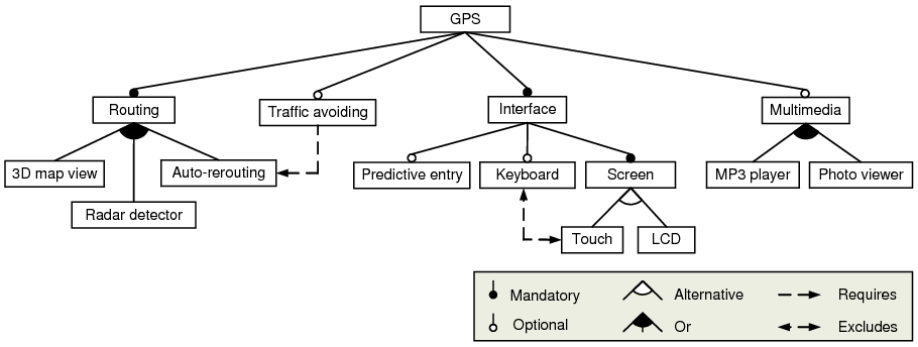


Figure 1: A sample feature model

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 [36]. 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 [59]. 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 *Auto-rerouting* as an optional feature.
- **Alternative.** A set of child features are defined as alternative if only one feature can be selected when its parent feature is part of the product. In our SPL, software for GPS devices must provide support for either a *LCD* or *Touch* screen but only one of them in the same product.
- **Or-Relation.** A set of 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 its parent feature appears. In our example, GPS devices can provide support for a *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 such product. GPS devices with *Traffic avoiding* requires *Auto-rerouting*.

- **Excludes.** If a feature A excludes a feature B, both features cannot be part of the same product. In our sample SPL, GPS with *Touch* screen cannot include a *Keyboard* and vice-versa.

2.2. Evolutionary algorithms

Principles of biological evolution have inspired the development of a whole branch of optimization techniques called *Evolutionary Algorithms (EAs)*. These algorithms manage a set of candidate solutions to an optimization problem that are combined and modified iteratively to obtain better solutions. Each candidate solution is referred to as *individual* or *chromosome* in analogy to the evolution of species in biological genetics where DNA of individuals is combined and modified along generations enhancing species through natural selection. Two of the main properties of EAs are that they are heuristic and stochastic. The former means that there is no guarantee of obtaining the global optimum for the optimization 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 optimization techniques being applied successfully in nearly all scientific and engineering areas by thousands of practitioners [6, Section D]. This success is due to the ability of EAs to obtain near optimal solutions to extremely hard optimization problems with affordable time and resources.

As an example, let us consider the design of a car as an optimization problem. A similar example was used to illustrate the working of EAs in [77]. Let us consider that our goal is to find a car design that maximize 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, components as well as shape and body elements. Moreover, this problem is likely to have 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 detail its main steps relating them to our example.

Initialization. The initial population (i.e. set of candidate solutions to the problem) is usually generated randomly. In our example, this could be done by choosing a set of random values for the design parameters of the car. Of course, the chances of finding optimal or near optimal car designs in this initial population are very small. However, promising values found at this step will be used to produce variants along the optimization 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 its optimality for the problem. This enables the objective comparison of candidate solutions with respect to an optimization 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 inputs parameters should produce the same output. In our car example, a simulator could be used to provide the maximum speed prediction as fitness.

Stop criteria. Iterations on the remainder of the algorithm are performed until a ter-

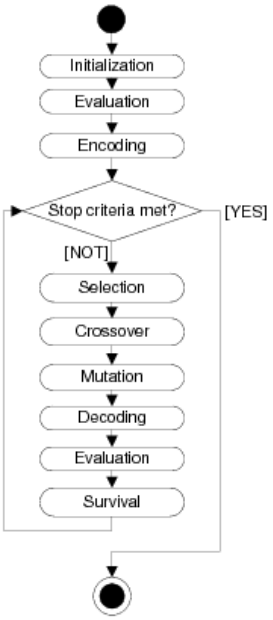


Figure 2: UML activity diagram of evolutionary algorithms

mination criterion is met. Typical stop 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, individuals need to be *encoded* expressing its characteristics in a form that facilitates its manipulation during the rest of the algorithm. In biological genetics, DNA encodes individual's characteristics on chromosomes that are used on reproduction and whose modifications produce mutants. For instance, classical encoding mechanisms on EAs are binary vectors encoding numerical values in genetic algorithms (so-called binary enconding) [6, Sec. C1.2] and tree structures encoding abstract syntax of programs in genetic programming (so-called tree encoding) [40]. In our car example, this step would imply to express design parameters of cars using some kind of 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 current population in order to create new offspring. In this process, better individuals usually have more probability of being selected resembling the natural evolution where stronger individuals have more chances of reproduction. For instance, two classic selection mechanisms are roulette wheel and tournament selection [25]. When using the former, the probability of choosing an individual is proportional to its fitness determining the width of the slice of a hypothetic spinning roulette wheel. This mechanism is often modified assigning probability 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.

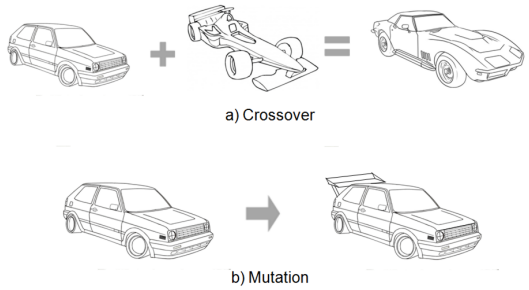


Figure 3: Sample crossover and mutation in the search for an optimal car design.

Crossover. These are the techniques used to combine individuals and produce new individuals in an analogous way to biological reproduction. Crossover mechanisms depend on the encoding scheme used but standard mechanisms are present in literature for widely used encodings [6, Sec. C3.3]. For instance, two classical crossover mechanisms for binary encoding are one-point crossover [30] and uniform crossover [1]. When using the former, a random location in the vector is chosen as 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 a high–level application of crossover in our example of car design. An F1 car and an small family car are combined by crossover producing a sports car. The new vehicle has some design parameters inherited directly of 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 certain probability where small modifications are more likely than larger ones. This step is crucial to prevent the algorithm from getting stuck prematurely at a locally optimal solution. An example of mutation in our car optimization 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* [6], 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 conformed 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 improving species.

In order to better clarify the operation of EAs, in algorithm 1 we provide a pseudocode that complements the common working scheme shown in Fig. 2.

Algorithm 1 Evolutionary Algorithm pseudocode

```
bestEval ← −∞
Population ← initialPopulation() {Initialization of Population}
for all individual ∈ Population do
  nextEval ← f(decode(individual))
  if nextEval < bestEval then
    bestSolution ← individual
    bestEval ← nextEval
  end if
end for
repeat
    {Main loop}
    Parents ← crossoverSelection(Population)
    {Select Individuals for Crossover}
    Offspring ← crossover(Parents) {Crossover}
    Population ← mutation(Population) {Mutation}
    for all individual ∈ (Population ∪ Offspring) do
    {Evaluation of new population and Offspring}
    nextEval ← f(decode(individual))
    if nextEval < bestEval then
      bestSolution ← decode(individual)
      bestEval ← nextEval
    end if
  end for
  {Selection of survival individuals (Next population)}
  Population ← survivalSelection(Population ∪ Offspring)
until Termination Criteria is satisfied
return bestSolution
```

The first nine lines of the algorithm correspond to the initialization, where the individuals of the population are generated randomly based on the function $initialPopulation$. The loop in this section searches for the best individual in this initial population and stores it in the variable $bestSolution$. Next, the main loop of the algorithm executes the main elements of the evolutionary algorithm: crossover, mutation and selection for survival. Along the iterative execution of this loop, the best individual found is maintained and stored in the variable $bestSolution$, that is returned as a result or the algorithm.

3. Automated generation of hard feature models

In this section, we present the core of our contribution. First, we introduce a novel evolutionary algorithm to deal with optimization problems on feature models. Then, we present a specific instantiation of the algorithm to search for computationally-hard feature models.

3.1. An evolutionary algorithm for feature models

In this section, we present ETHOM, a novel evolutionary algorithm for the generation of optimized feature models. The algorithm takes several size constraints and a fitness function as input and returns a feature model of the given size maximizing the optimization criteria defined by the function. A key benefit of our algorithm is that it is very generic being 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 [60, 71, 83]. There are also tools supporting the random generation of feature models such as BeTTy [15, 61] or SPLOT [47, 69].

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

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 were either not adequate to represent 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 with 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, feature labelled with *'0'* in the tree is stored in the first position of the array, feature labelled with *'1'* is stored the second position and so on. Each feature in the tree array is defined as a two-tuple $< PR, C >$ 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, first position in the tree array, $< Op, 2 >$, indicates that 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 mainly depends on the application domain.

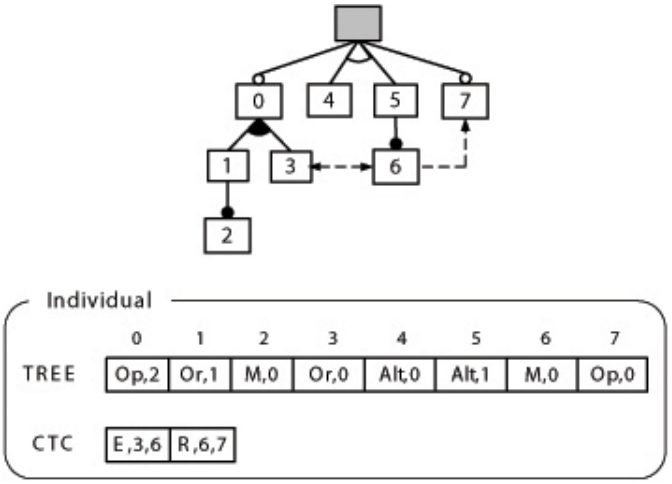


Figure 4: Encoding of a feature model in ETHOM

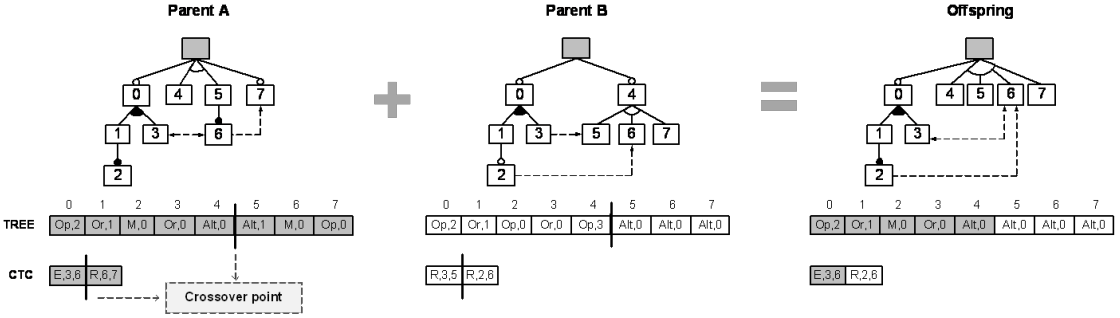


Figure 5: Example of one-point crossover in ETHOM

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 (so-called crossover point). Finally, the offspring is created by copying the content 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.

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.* It changes randomly the type of a relationship in the tree array, e.g. from mandatory,$< \mathbf{M}, 3 >$, to optional,$< \mathbf{Op}, 3 >$.
- *Operator 2.* It changes randomly 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.* It 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.* It changes randomly (with equal probability) the origin or destination feature of a constraint in the CTC array, e.g. from $< E, \mathbf{3}, 6 >$ to $< E, \mathbf{1}, 6 >$. Origin and destination features are ensured to be different.

These operators are applied randomly with the same probability.

Decoding. At this stage, the array-based chromosomes are translated back into feature models in order to 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 sharing 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 depend on the problem and it is ultimately up to the user. In our work, we used a repairing strategy described in the next section.

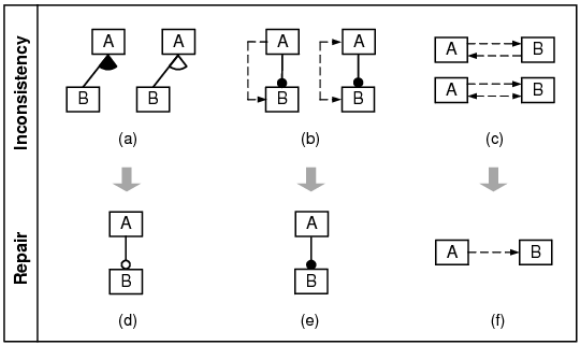


Figure 6: Examples of infeasible individuals and repairs

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 due to probability.

3.2. Instantiation of the algorithm

In this section, we propose to model the problem of finding computationally–hard feature models as an optimization problem and to solve it using an instantiation of our evolutionary algorithm. We chose evolutionary computation because it has proved to be a robust search technique suited for the complex search spaces and noisy objective functions used when dealing with non–functional properties [2]. A key benefit of our approach is that it takes into account the characteristics of the tools under test trying to exploit its vulnerabilities. Also, our approach is very generic being applicable to any automated operation on feature models, not only analyses, in which the quality (i.e. fitness) of the models can be measured quantitatively.

In order to find a suitable configuration of our algorithm, we performed numerous executions of a sample optimization problem evaluating different combination of values

for the key parameters of the algorithm, presented in Table 1. The optimization problem was to find a feature model maximizing the execution time invested by the analysis tool when checking the model consistency (i.e. whether it represents at least one product). We chose this analysis operation because it is currently the most quoted in the literature [11]. In particular, we looked for feature models of different size maximizing execution time in the CSP solver JaCoP integrated into the FaMa framework v1.0. FaMa is a widely-used tool of significant size, highly tested [62] and integrated into tools like MOSKitt [51]. These reasons, coupled to our familiarity with the tool as leaders of the project, made us choose FaMa as a good tool to be use in our work. Next, we clarify the main aspects of the configuration of our algorithm:

- **Initial population.** We used a Java program implementing the algorithm for the random generation of feature models described by Thüm et al. [71]. In particular, our algorithm for the generation of random feature models with n features work as follows. The generation process starts by creating a single root node and running several iterations. In each iteration, an existing node without children is randomly selected, and a random amount (between one and the maximum branching factor) of child nodes is added. A child node is connected to its parent feature using a mandatory, optional, or-relationship or alternative relationship with equal probability. After each iteration, relationships of type or and alternative with a single child feature are removed to make the model syntactically correct. This iteration is continued until the feature model has n features. Once the tree has been generated, CTCs are added in several iterations. In each iteration, two random features are selected and a 'requires' or 'excludes' constraints is set between both features with equal probability. Features with parental relationship are not allowed to share CTC since that would generate either redundant or inconsistent information. Also, any two features cannot share more than one CTC.

- **Fitness function.** Our first attempt was to measure the execution time in milliseconds invested by FaMa to perform the operation. However, we found that this was very inaccurate since the result of the function was deeply affected by the system load, i.e. it 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 [72]. In contrast to the execution time, most CSP backtracking heuristics are deterministic, i.e. different executions of the tool with the same input produces the same number of backtracks. Together with execution time, the number of backtracks is commonly used to measure the complexity of constraint satisfaction problems [72]. 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 repairing techniques. More specifically, we evaluated the following repairing algorithm with 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 share more than one constraint (e.g. the model in Fig. 6(c) is changed as in Fig. 6(f)).

- **Stop criterion.** There is no means of deciding when an optimum input has been found and ETHOM should be stopped [77]. In our work, 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. solution to the problem.

Table 1 depicts the values evaluated for each parameter. These values were based on: related works using evolutionary algorithms [26], the literature on parameter setting [6, Section E], and our previous experience in this domain [53]. Each combination of parameters was executed 10 times to avoid heterogeneous results and to allow us to perform statistical analysis on the data. Underlined values were those providing better results and therefore those 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

4. 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 [15, 61], an open-source Java tool for functional and performance testing on the analysis of feature models developed by the authors.

We evaluated the efficacy of our approach by comparing it to random search since this is the most extended strategy for performance testing in the analysis of feature models. In particular, the evaluation of our evolutionary program was performed through a number of experiments. On each experiment, we compared the effectiveness of random generators and ETHOM on the search for feature models maximizing 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.

In general, it is not possible to verify that the solution obtained by ETHOM represents a global optimum, i.e. there is not a known objective value. Nevertheless, although the problem of finding hard inputs does not have a known optimum, this problem can still be regarded as an optimization problem by defining user-defined optimum values to be reached. For instance, Wegener et al. [76, 77] used genetic algorithms to find inputs that maximize the execution time of real time systems. In their work, the objective value was a time constraint introduced to the algorithm as input. Similarly, ETHOM can also receive an optional user-defined optimum as input. In fact, to prevent the experiments from getting stuck, we used a timeout of 30 minutes as a stopping criterion which can be regarded as a defined optimum for the problem of generating inputs maximizing runtime. For the experiments with memory, however, we decided not to set any optimum to show how good the results of ETHOM could be when allowed to run freely for a maximum number of generations (or a maximum timeout).

For the analysis of feature models, we used two independent frameworks to avoid biased results, FaMa [20] and SPLOT [47]. Both tools are widely used by the community and are implemented in Java which facilitated its use with our prototype. Both frameworks can be configured to use different built-in solvers. In order to evaluate the effectiveness of ETHOM in heterogeneous scenarios, we selected solvers dealing with different paradigms i.e. boolean SATisfiability (SAT), Constraint Satisfaction Problems (CSP) and Binary Decision Diagrams (BDD). In particular, we used the CSP solver JaCoP [33] used by default in FaMa and the two solvers available in SPLOT, Sat4j [58] and JavaBDD [34] for SAT and BDD-based analyses respectively.

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.

4.1. Experiment #1: Maximizing execution time

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

4.1.1. Experiment #1(a): Maximizing execution time in a CSP Solver

Experimental setup. This experiment was performed through a number of iterative steps. On each step, we generated 5,000 random 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. We may recall that the size of 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 perform statistical analysis of the data. In total, we performed about 5 million executions[1] of the fitness function for this experiment. The fitness was set equal to the number of backtracks obtained 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. The models that exceeded the timeout could be considered as the result of lucky executions of ETHOM. We think that discarding these models improves the robustness of our evaluation since it makes easier to observe how the search process converges over time toward a solution in normal executions, i.e. when not being as lucky to find the hardest model quickly.

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 10 times each returned solution to get average execution times.

The dependent variable of this experiment is the number of backtracks generated in the consistency analysis of FMs. The independent variable is the technique used to generate the FMs analyzed (either ETHOM or random search). The alternative hypothesis (H_1) of this experiment states that "*There is a significant difference between the average number of backtracks executed by the CSP solver JACOP in order to analyze the consistency of the FM generated by ETHOM , and the average number of backtracks generated for analyzing those FMs generated by random search*". The null hypothesis (H_0) states the absence of such a difference. The hypotheses and design of the experiment #1(a) are summarized in table 2.

Analysis of results. Fig. 7 depicts the effectiveness of ETHOM for each size range of the feature models generated. We define the *effectiveness* (a.k.a score) of our evolutionary program as the percentage of times (out of 25) in which the program found a better optimum than random models, 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 due to the simplicity of the analysis in this size

[1] 5 features ranges x 4 constraints ranges x 25 iterations x 10,000 (5,000 random search + 5,000 evolutionary search)

Hypotheses			
Null Hypothesis (H_0)	There is no significant difference in the average number of backtracks needed to analyze the consistency of the FMs generated by ETHOM and those generated randomly		
Alt. Hypotheis (H_1)	There is a statistically significant difference in the average number of backtracks needed to analyze the consistency of the FMs generated by ETHOM and those generated randomly		
Design			
Depedent Variable	Number of backtracks generated by the analysis of the FM using the consistency operation		
Independent Variable	Technique used for FM generation	**Levels:**	ETHOM, Random
Blocking Variables	Number of Features	**Levels:**	200, 400, 600, 800, 1000
	Percentage of Cross-Tree Constraints (with respect to the number of features)	**Levels:**	10%, 20%, 30%, 40%
Constants	CSP solver	**Value:**	JaCoP
	Heuristic for variable selection in the CSP solver	**Value:**	*MostConstrainedDynamic*
	Heuristic for value selection from the domains in the CSP solver	**Value:**	*IndomainMin*

Table 2: Hypotheses and design of Experiment #1(a) (number of backtracks with CSP)

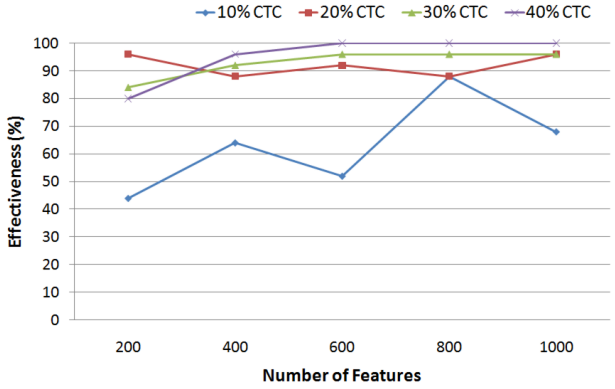


Figure 7: Effectiveness of ETHOM in Experiment #1.

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 4 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 (i.e. number of backtracks) obtained 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 by random models. The fitness of the hardest models generated using our evolutionary approach was on average over 3,500 times higher than that of random 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 invested 0.06 seconds to analyse the random 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 of random models to the 1,032.2 seconds (17.2 minutes) invested 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.

Tables 3, 5, and 4 depict the evaluation results for feature models with 10%, 30%, and 40% of cross–tree constraints repectively. Like in Table 4, Tables 3, 5, and 6 show the average and maximum fitness obtained, as well as the average and maximum execution times of the hardest feature models found, and the effectivennes of ETHOM in the last column. It is important to note that the effectiveness of ETHOM is higher than 80% on all the cases of tables 5, and 4, supporting the consclussions stated above. However, table 3 shows a significantly lower effectiveness, drawing a limitation in the applicability of the proposal.

	Random Testing				ETHOM				
#Features	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Score (%)**
200	5.24	17	20.97	38.90	24.64	361.00	27.77	60.50	44
400	11.72	96	35.06	51.80	417.60	7871.00	57.97	330.40	64
600	28.56	287	50824.00	96.30	72508.72	1780227.00	2525.58	59862.30	52
800	15.16	87	61.25	89.00	336715.28	5316995.00	18007.71	280405.60	88
1000	40.6	186	94.09	120.70	1184620.60	29491237.00	66244.50	1643863.60	68
Total	20.256	287	10207.07	120.70	318857.37	29491237.00	17372.71	1643863.60	63.2

Table 3: Evaluation results on the generation of feature models maximizing execution time in a CSP solver. CTC=10%

	Random Testing				ETHOM				
#Features	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Score (%)**
200	8.08	61	0.02	0.03	63.36	215	0.04	0.06	96
400	30.08	389	0.04	0.07	7,128.44	106,655	0.24	2.93	88
600	40.28	477	0.05	0.09	9,188.20	116,479	0.70	7.98	92
800	91.08	1,279	0.08	0.20	22,427.60	483,971	1.28	24.56	88
1000	57.24	582	0.10	0.13	964,532.64	23,598,675	42.54	1,032.19	96
Total	**45.35**	**1,279**	**0.06**	**0.20**	**200,668.05**	**23,598,675**	**8.96**	**1,032.19**	**92**

Table 4: Evaluation results on the generation of feature models maximizing execution time in a CSP solver. CTC=20%

	Random Testing				ETHOM				
#Features	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Score (%)**
200	10.4	32	22.94	37.20	526.00	8069.00	52.16	171.30	84
400	19.28	103	35.57	41.90	1212.56	21696.00	93.46	611.00	92
600	11.72	26	48.99	64.10	8826.72	214536.00	345.91	6621.50	96
800	18.32	80	70.31	97.50	11487.36	174361.00	787.81	13940.30	96
1000	23.08	170	87.78	117.20	1510.76	15372.00	287.13	1617.10	96
Total	16.56	170	53.12	117.20	4712.68	214536.00	313.29	13940.30	92.8

Table 5: Evaluation results on the generation of feature models maximizing execution time in a CSP solver. CTC=30%

	Random Testing				ETHOM				
#Features	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Score (%)**
200	9,6	39	25,78	35,50	132,24	959,00	38,26	83,10	80
400	11,28	56	39,00	75,90	131,76	987,00	56,45	130,80	96
600	8,76	31	50,09	68,80	6224,72	77181,00	352,03	4087,10	100
800	7,64	14	64,50	85,90	386,84	4299,00	111,38	519,00	100
1000	8,24	19	82,09	101,40	133,72	949,00	120,36	274,80	100
Total	9,104	56	52,29	101,40	1401,86	77181,00	135,70	4087,10	95,2

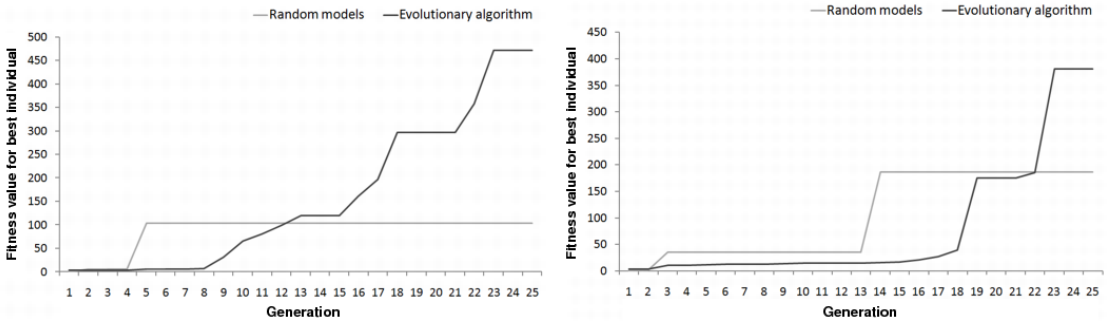
Table 6: Evaluation results on the generation of feature models maximizing execution time in a CSP solver. CTC=40%

A global summary of the results is presented in Table 7. The table depicts the maximum execution times invested by the CSP solver to analyse the hardest models found using random and evolutionary search. The data show that our approach was more effective than random models in all size ranges. The hardest random model required 0.2 seconds to be processed. In contrast, our evolutionary approach found four models requiring between 1 and 27.3 minutes to be analysed. Interestingly, our algorithm was able to find smaller and significantly harder feature models (e.g. 600 features and 10% of CTC) than the hardest random model found which had 800 features and 20% of constraints. This emphasizes the ability of our approach to generate motivating input models of realistic size that reveal the vulnerabilities of tools and heuristics instead of just stressing them using large random models.

	10% CTC		20% CTC		30% CTC		40% CTC	
#Features	**Random Time**	**ETHOMTime**	**Random Time**	**ETHOMTime**	**Random Time**	**ETHOMTime**	**Random Time**	**ETHOMTime**
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	59.86	0.09	7.98	0.06	6.62	0.07	4.09
800	0.09	280.41	0.20	24.56	0.10	13.94	0.09	0.52
1,000	0.12	1,643.86	0.13	1,032.19	0.12	1.62	0.10	0.27

Table 7: Maximum execution times produced by random models and our evolutionary program.

Fig. 8 compares random and evolutionary techniques for the search for a feature model maximizing the number of backtracks in two sample executions. This occurred because the results obtained by our evolutionary program were so much higher than those of random models that it was unfeasible to represent them using a similar scale. Horizontally, the graphs show the number of generations where each generation represent 200 executions of the fitness function. Fig. 8(a) shows that random models reaches its maximum number of backtracks after only 5 generations (about 1000 executions). That is, the generation of 4,000 other random models do not produce any higher number of backtracks and therefore are useless. In contrast to this, our evolutionary approach shows a continuous improvement. After 13 generations (about 2600 executions), the fitness found by evolutionary search are above of those of random models. Fig. 8(b) depicts another example in which random models are lucky to find a high number of backtracks in the 14th generation. Evolutionary optimization, however, once again manages to improve the execution times continuously overcoming the best random fitness after 22 generations. In generation number 23, even a significant leap of about 200 backtracks can be observed. In both examples, the curve trace suggests that the evolutionary algorithm would find even better solutions if the number of generations were increased. This was confirmed in a later experiment in which the program



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 random models and our evolutionary algorithm for the search for the highest number of backtracks

was allowed to run for up to 125 generations (25,000 executions of the fitness function) finding feature models producing more than 70 million backtracks (see Section 4.5 for details).

4.1.2. Experiment #1(b): Maximizing execution time in a SAT Solver

Experimental setup. The experimental setup used for this experiment was similar to that used with the CSP-based one. The fitness of each model was measured as the number of decisions (i.e. steps) taken by the SAT solver when checking its consistency. For the analysis, we used the SAT solver integrated into FaMa v1.0. Just as in the experiment above described, in order 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 time was exceeded, a new iteration was started. 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 decisions. More specifically, we executed 10 times each optimal solution to get average execution times.

The dependent variable of this experiment is the execution time spent in the consistency analysis of FMs. The independent variable is the technique used to generate the FMs analyzed (either ETHOM or random search). The alternative hypothesis (H_1) of this experiment states that "*There is a significant difference between the average execution time spent by the SAT solver for analyzing the consistency of the FM generated by ETHOM and the average time spent for analyzing those FMs generated by random search*". The null hypothesis (H_0) states the absence of such a difference. The hypotheses and design of the experiment #1(b) are summarized in table 8.

Analysis of results. Fig. 9 depicts the effectiveness of our algorithm for each size range of the feature models generated. As illustrated, the effectiveness of evolutionary program was over 80% in most of the cases reaching 92% or higher in nine of them. Overall, our evolutionary program found harder feature models than those generated randomly in 87.8% of the executions.

Tables 9, 10, 11, and 12 depict the evaluation results obtained using the SAT-based analysis for feature models with 10%, 20% 30%, and 40% of cross–tree constraints repectively. Specifically, those tables show the average and maximum fitness obtained,

Hypotheses			
Null Hypothesis (H_0)	There is no significant difference in the average number of decisions needed to analyze the consistency of the FMs generated by ETHOM and those generated randomly		
Alt. Hypotheis (H_1)	There is a statistically significant difference in the average number of decisions needed to analyze the consistency of the FMs generated by ETHOM and those generated randomly		
Design			
Depedent Variable	Execution time needed for the analysis of the consistency of the FMs		
Independent Variable	Technique used for FM generation	**Levels:**	ETHOM, Random
Blocking Variables	Number of Features	**Levels:**	200, 400, 600, 800, 1000
	Percentage of Cross-Tree Constraints (with respect to the number of features)	**Levels:**	10%, 20%, 30%, 40%
Constants	SAT solver	**Value:**	SAT solver integrated in FaMa v1.0

Table 8: Hypotheses and design of Experiment #1(b) (execution time with SAT)

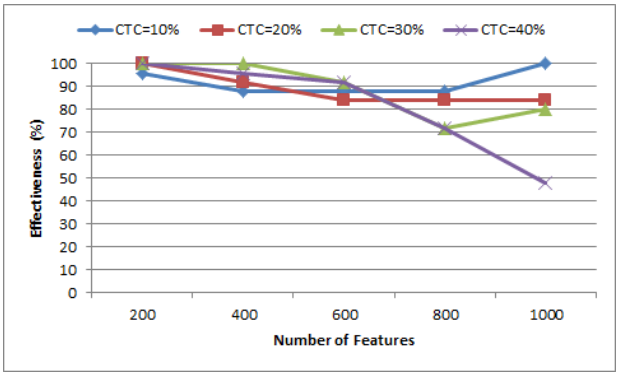


Figure 9: Effectiveness of our evolutionary algorithm in Experiment #1.

as well as the average and maximum execution times of the hardest feature models found, and the effectivennes of ETHOM in the last column.

We may remark, that the differences in the execution times obtained using random and evolutionary techniques were not significant. This finding supports the results of Mendoca et al. [49] 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 emphasize, however, that SAT solvers are not the optimum solution for all the analyses that can be performed on a feature model [11, 12, 56]. Previous studies shows that CSP and BDD solvers are often a better alternative than SAT solvers and therefore experiments with these and others solvers are still necessary to study their applicability.

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

In this experiment, we evaluated the ability of ETHOM to generate input feature models maximizing the memory consumption of a solver. In particular, we measured the mem-

	Random Testing				ETHOM				
#Features	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Score (%)**
200	100.64	121.00	35.72	146.50	148.44	208.00	29.54	59.70	96
400	163.76	221.00	69.53	73.70	234.72	338.00	65.38	77.60	88
600	227.64	306.00	121.63	144.00	299.84	423.00	120.72	128.20	88
800	289.20	397.00	188.85	201.30	390.36	546.00	188.87	199.40	88
1000	316.88	463.00	270.38	316.10	504.88	703.00	287.44	313.20	100
Total	219.62	463.00	137.22	316.10	315.65	703.00	138.39	313.20	92

Table 9: Evaluation results on the generation of feature models maximizing execution time in a SAT solver. CTC=10%

	Random Testing				ETHOM				
#Features	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Score (%)**
200	97.64	116.00	31.79	36.10	151.44	244.00	28.72	33.60	100
400	164.36	211.00	75.18	83.20	230.28	406.00	691.72	76.90	92
600	219.36	285.00	12.61	133.90	303.36	408.00	126.80	137.30	84
800	278.44	398.00	198.87	210.60	317.24	399.00	197.79	224.40	84
1000	321.80	457.00	282.14	291.60	422.32	582.00	299.70	312.00	84
Total	216.32	457.00	120.12	291.60	284.93	582.00	268.95	312.00	88.8

Table 10: Evaluation results on the generation of feature models maximizing execution time in a SAT solver. CTC=20%

	Random Testing				ETHOM				
#Features	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Score (%)**
200	92.96	116	33.16	35.00	146.52	257.00	28.95	37.40	100
400	131.72	164	78.48	84.60	190.84	252.00	74.12	83.00	100
600	176.24	229	134.12	150.30	252.64	356.00	133.52	143.40	92
800	218.04	276	207.63	247.40	257.36	388.00	212.49	256.50	72
1000	227.4	280	297.06	315.00	271.48	395.00	311.38	334.30	80
Total	169.27	280.00	150.09	315.00	223.77	395.00	152.09	334.30	88.8

Table 11: Evaluation results on the generation of feature models maximizing execution time in a SAT solver. CTC=30%

	Random Testing				ETHOM				
#Features	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Avg Fitness**	**Max Fitness**	**Avg Time**	**Max Time**	**Score (%)**
200	79.96	130	35	39.1	133.52	181	30	33.1	100
400	108.88	147	82.34	87.5	174.88	244	78	85.7	96
600	137.08	190	140	149.1	184.12	298	141	157.6	92
800	145.28	206	217.64	242.8	181.96	334	223	257.6	72
1000	145.68	216	310	320.5	150.92	214	328	364	
Total	117.8	206	118.661	242.8	168.62	334	118.018	257.6	90

Table 12: Evaluation results on the generation of feature models maximizing execution time in a SAT solver. CTC=40%

ory consumed by a BDD solver when finding out the number of products represented by the model. We chose this analysis operation because it is the hardest one in terms of complexity and it is currently the second operation most quoted in the literature [11, 56]. 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 [11]. 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 [50]. 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 generated 5,000 random models and compiled each of them into a BDD for counting the number of solutions measuring its size. We then executed our evolutionary program and allowed it to run for 5,000 executions of the fitness function looking for feature models maximizing the size of the BDD and compared the results. Again, this process was repeated with different combination of features, {50, 100, 150, 200, 250} and percentage of constraints, {10%, 20%, 30%} to evaluate the scalability of our approach. For each size of the model, we repeated the process 25 times to get statistics from the data. In total, we performed 3.75 million executions of the fitness function for this experiment. We may remark that we generated smaller feature models than those presented in 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 [50]. For the analysis, we used the solver JavaBDD integrated into the feature model analysis tool SPLOT. We chose SPLOT because it integrates highly 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 [50]. As in our previous experiment, we set a maximum timeout of 30 minutes for the fitness function to prevent the experiment from getting stuck when finding too good solutions. After all the executions, we measured the compilation and execution time of the hardest feature models found for a more detailed comparison, i.e. those producing a largest BDD. Each optimal solution was compiled and executed 10 times to get average times.
The dependent variable of this experiment is the execution time spent in the consistency analysis of FMs. The independent variable is the technique used to generate the FMs analyzed (either ETHOM or random search). The alternative hypothesis (H_1) of this experiment states that "*There is a significant difference between the average number of nodes generated by the BDD-solver for analyzing the number of products of the FMs generated by ETHOM and the average number nodes generated for analyzing those FMs generated by random search*". The null hypothesis (H_0) states the absence of such a difference.

The hypotheses and design of experiment #2 are summarized in table 13.

Analysis of results. Fig. 10 depicts the effectiveness of ETHOM for each size range of

Hypotheses			
Null Hypothesis (H_0)	There is no significant difference between the number of nodes generated by the BDD-solver in order to analyze the number of products of the FMs generated by ETHOM and the number of nodes generated by the BDD-solver in order to analyze the number of products of the FMs generated randomly.		
Alt. Hypotheis (H_1)	There is a statistically significant difference between the number of nodes generated by the BDD-solver in order to analyze the number of products of the FMs generated by ETHOM and the number of nodes generated by the BDD-solver in order to analyze the number of products of the FMs generated randomly.		
Design			
Depedent Variable	Number of nodes generated by the BDD-solver in order to analyze the FM using the number of products (#products) operation		
Independent Variable	Technique used for FM generation	**Levels:**	ETHOM, Random
Blocking Variables	Number of Features	**Levels:**	200, 400, 600, 800, 1000
	Percentage of Cross-Tree Constraints (with respect to the number of features)	**Levels:**	10%, 20%, 30%, 40%
Constants	BDD-solver	**Value:**	JavaBDD as integrated in SPLOT
	Ordering Heuristic	**Value:**	*Pre-CL-MinSpan*

Table 13: Hypotheses and design of Experiment #2 (number of nodes generated by the BDD-solver)

the feature models generated, i.e. percentage of times (out of 25) in which evolutionary search found feature models producing higher memory consumption than random models. As illustrated, the effectiveness of ETHOM was over 96% in most of the 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 concluded that this was not a limitation but a proof of the quality of the solutions found. In particular, we found that 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 model generated were so hard that they often took more than 30 minutes to be analyzed 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 models in 94.4% of the executions. The results suggest, however, that increasing the maximum timeout would increase the effectiveness significantly.

Table 14 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) invested 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 models. The largest BDD generated from random models had 14.8 million nodes while the largest BDD obtained using ETHOM had 20.6 million nodes. Again, results revealed that ETHOM

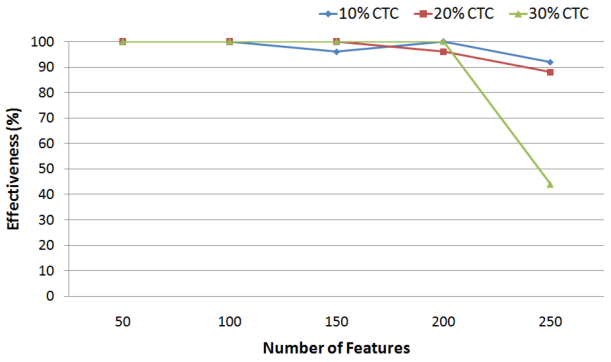
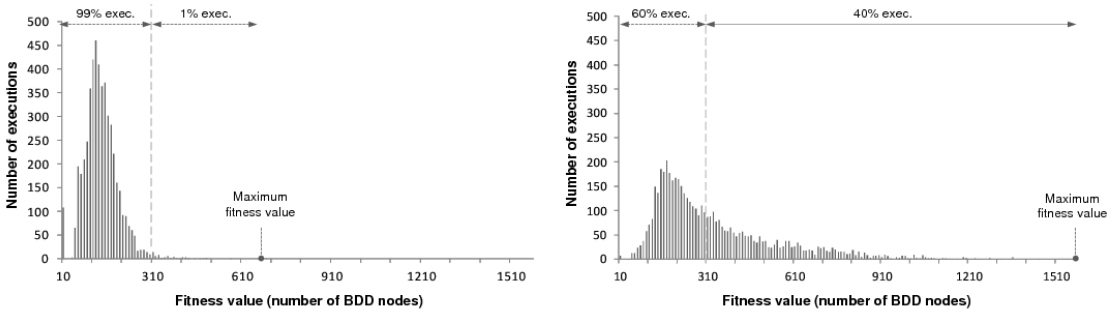


Figure 10: Effectiveness of ETHOM in Experiment #2.



a) Distribution of fitness values for random models

b) Distribution of fitness values for our evolutionary approach

Figure 11: Histograms with the distribution of fitness values for random and evolutionary techniques when searching for a feature model maximizing the size of the BDD.

was able to find smaller but harder models (e.g. 150-30%, 17.7 million nodes) than the hardest random 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 much higher if we would not have 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 to be compiled and analysed. This suggests that our approach can also deal with optimization criteria involving compilation time in BDD solvers.

	10% CTC				20% CTC				30% CTC			
	Random		Evolutionary		Random		Evolutionary		Random		Evolutionary	
#Features	BDD size	Time	BDD Size	Time	BDD size	Time	BDD Size	Time	BDD size	Time	BDD Size	Time
50	781	0	1,963	0	2,074	0	8,252	0.01	2,455	0.01	10,992	0.01
100	7,629	0.01	20,077	0.02	33,522	0.03	161,157	0.20	95,587	0.08	419,835	0.73
150	65,627	0.10	188,985	0.31	374,675	0.91	3,060,590	12.80	673,410	1.28	11,221,303	24.22
200	203,041	0.09	924,832	0.86	2,735,005	4.34	19,698,780	75.05	3,394,435	58.22	23,398,161	380.52
250	1,720,983	3.69	7,170,121	25.94	25,392,597	82.28	27,970,630	253.32	20,579,015	343.72	22,310,416	431.62

Table 14: BDD size and computation time of the hardest feature models found using random techniques and our evolutionary program.

Fig. 11 shows the frequency with which each fitness value was found during the search for a feature model producing the largest BDD. 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 comparison of the results of both techniques using the same scale.

For random models (Fig. 11(a)), a narrow Gaussian-like curve is obtained with more than 99% of the executions producing fitness values under 310 BDD nodes. During evolutionary execution (Fig. 11(b)), however, a wider curve is obtained with 40% of the execution producing values over 310 nodes. Both histograms clearly show how evolutionary programming performed a more exhaustive search in a larger portion of the solution space than that explored by random models. This trend was also observed in the rest of size ranges.

4.3. Experiment #3: Evaluating the impact of the number of generations

During the work with ETHOM, we detected that the maximum number of generations used as stop criterion had a great impact in the results of the algorithm. In this experiment, we evaluated that impact with a double aim. First, we tried to find out the minimum number of generations required by ETHOM to offer better results than random techniques on the search for hard feature models. Second, we wanted to find out whether ETHOM was able to find even harder models than in our previous experiments when allowed to run for a large number of generations. Next, we present the setup and results of our experiment.

Experimental setup. We repeated Experiments #1 and #2 using different number of generations from 10 to 25, 50, 75, 100 and 125. To make the experiments affordable, we used a fixed size for the models being generated. In particular, we searched for: i) feature models with 500 features and 20% CTCs maximizing the number of backtracks in the CSP solver JaCoP, and ii) feature models with 100 features and 20% CTCs maximizing the number of BDD nodes in JavaBDD. In total, we performed over 7.5 million executions of the fitness functions for this experiment. A timeout of 30 minutes was used for all the executions.

The hypotheses and design of the Experiment #3 are summarized in table 15.

Hypotheses			
Null Hypothesis (H_0)	There is no significant difference in the effectiveness of ETHOM as we increase the number of generations executed from 10 to 125		
Alt. Hypotheis (H_1)	There is a statistically significant significant difference in the effectiveness of ETHOM as we increase the number of generations executed from 10 to 125		
Design			
Depedent Variable	Effectiveness of ETHOM		
Independent Variable	Number of generations executed	**Levels:**	10, 25, 50, 75, 100, 125
Blocking Variables	FM properties and Analysis	**Levels:**	i) feature models with 500 features and 20% CTCs maximizing the number of backtracks in the CSP solver JaCoP, and ii) feature models with 100 features and 20% CTCs maximizing the number of BDD nodes in JavaBDD

Table 15: Hypotheses and design of Experiment #3 (Effectiveness of ETHOM for different number of generations)

Analysis of results. Table 16 depicts the results for this experiments. For each number of generations (i.e. stop criterion), the maximum fitness and the effectiveness of both random and evolutionary search are presented. The results revealed that the effectiveness of ETHOM was around 96% (CSP solver) and 100% (BDD solver) when the number of generations was 25 or higher. More importantly, we found that the results provided by evolutionary search were better and better as the number of generations

was increased without reaching a clear peak meanwhile 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 meanwhile random search found a maximum value of only 1,603 backtracks. Similarly, the maximum random fitness produced in our experiment with BDD was 89,779 nodes, far from the best fitness obtained by our evolutionary program, 22.7 million nodes. Finally, we may emphasize that the maximum number of BDD nodes found by ETHOM in the range of 125 generations (22.2 million nodes) was 120 times higher than the maximum obtained when using 25 generations as stop criterion (185,203 nodes). This shows the power of ETHOM when it is allowed to run for a long number of generations.

	CSP Solver (fitness=#backtracks)			**BDD Solver** (fitness=#BDD nodes)		
#Generations	**Random Fitness**	**ETHOM Fitness**	**Score (%)**	**Random Fitness**	**ETHOM Fitness**	**Score (%)**
10	1,603	258	84	36,975	46,023	92
25	588	34,185	96	33,876	185,203	100
50	234	1,123,030	96	89,779	1,531,579	100
75	573	25,603,183	96	66,950	10,831,443	100
100	917	15,085,200	92	38,267	22,714,010	100
125	438	77,635,583	96	80,101	22,237,169	100
Max	**1,603**	**77,635,583**	**96**	**89,779**	**22,714,010**	**100**

Table 16: Maximum fitness values obtained in Experiment #3

4.4. Experiment #4: Evaluating the generalizability of hardness of Feature Models

In this experiment, we checked whether the hard feature models generated by ETHOM-for a specific tool and configurarion 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 were trivially analyzed 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. This last experiment is described in detail in this section.

Experimental setup. In this expecirment we fixed the value generation heurisitic of the CSP solver JaCoP to "IndomainMin", and repeated the analysis of the hardest feature models found in Experiment #1 using the other seven heuristics available for variable selection, {$MaxRegret$, $LargestMin$, $SmallestMax$, $MostConstrainedDynamic$, $SmallestMin$, $MinDomainOverDegree$, $LargestDomain$,$SmallestDomain$}. A timeout of 30 minutes was used for all executions.

The hypotheses and design of Experiment #4 are summarized in table 17.

Hypotheses			
Null Hypothesis (H_0)	There is no significant difference between the average number of backtracks generated by the CSP-solver in order to analyze the consistency of the FMs generated by ETHOM, and the number of backtracks generated by the CSP-solver in order to analyze the consistency of the FMs generated randomly.		
Alt. Hypotheis (H_1)	There is a statistically significant difference between the average number of backtracks generated by the CSP-solver in order to analyze the consistency of the FMs generated by ETHOM, and the number of backtracks generated by the CSP-solver in order to analyze the consistency of the FMs generated randomly.		
Design			
Depedent Variable	Number of backtracks generated by the analysis of the FM using the consistency operation		
Independent Variable	Heuristic for variable selection in the CSP solver	**Levels:**	*MaxRegret*, *LargestMin*, *SmallestMax*, *MostConstrainedDynamic*, *SmallestMin*, *MinDomainOverDegree*, *LargestDomain*,*SmallestDomain*
Blocking Variables	Number of Features	**Levels:**	200, 400, 600, 800, 1000
	Percentage of CrossTree Constraints (with respect to the number of features)	**Levels:**	10%, 20%, 30%, 40%
Constants	CSP solver	**Value:**	JaCoP
	Heuristic for value selection from the domains in the CSP solver	**Value:**	*IndomainMin*

Table 17: Hypotheses and design of Experiment #4 (number of backtracks of the CSP solver with different heuristics)

Analysis of results. Table 18 shows the number of backtracks generated in the analysis of the hardest FMs of Experiment #1 with diffferent variable selection heuristics in Ja-

Features	% CTC	Variable Selection Heuristic							
		MaxRegret	LargestMin	SmallestMax	MostConstrainedDynamic	SmallestMin	MinDomainOverDegree	LargestDomain	SmallestDomain
200	10	383645.0	1,65E+14	383645.0	361.0	383645.0	361.0	Ex. time > 3600	383645.0
200	20	56.0	3043.0	56.0	215.0	56.0	215.0	65987.0	56.0
200	30	Ex. time > 3600	Ex. time > 3600	Ex. time > 3600	8069.0	Ex. time > 3600	8069.0	Ex. time > 3600	Ex. time > 3600
200	40	200.0	602.0	200.0	959.0	200.0	959.0	417858.0	200.0
400	10	159652.0	Ex. time > 3600	159652.0	7871.0	159652.0	7871.0	Ex. time > 3600	159652.0
400	20	5.0	6.0	5.0	106655.0	5.0	106655.0	Ex. time > 3600	5.0
400	30	649.0	Ex. time > 3600	649.0	21696.0	649.0	21696.0	Ex. time > 3600	649.0
400	40	Ex. time > 3600	Ex. time > 3600	Ex. time > 3600	987.0	Ex. time > 3600	987.0	Ex. time > 3600	Ex. time > 3600
600	10	1.0	11.0	1.0	1780227.0	1.0	1780227.0	Ex. time > 3600	1.0
600	20	2507.0	1407.0	2507.0	116479.0	2507.0	116479.0	Ex. time > 3600	2507.0
600	30	1.0	5.0	1.0	214536.0	1.0	214536.0	Ex. time > 3600	1.0
600	40	13.0	17542.0	13.0	77181.0	13.0	77181.0	Ex. time > 3600	13.0
800	10	43.0	3205.0	43.0	5316995.0	43.0	5316995.0	Ex. time > 3600	43.0
800	20	39.0	2897.0	39.0	483971.0	39.0	483971.0	Ex. time > 3600	39.0
800	30	Ex. time > 3600	Ex. time > 3600	Ex. time > 3600	174361.0	Ex. time > 3600	174361.0	Ex. time > 3600	Ex. time > 3600
800	40	1.0	1.0	1.0	4299.0	1.0	4299.0	Ex. time > 3600	1.0
1000	10	Ex. time > 3600	Ex. time > 3600	Ex. time > 3600	2,95E+14	Ex. time > 3600	2,95E+14	Ex. time > 3600	Ex. time > 3600
1000	20	Ex. time > 3600	Ex. time > 3600	Ex. time > 3600	2,36E+14	Ex. time > 3600	2,36E+14	Ex. time > 3600	Ex. time > 3600
1000	30	Ex. time > 3600	Ex. time > 3600	Ex. time > 3600	15372.0	Ex. time > 3600	15372.0	Ex. time > 3600	Ex. time > 3600
1000	40	1093.0	2090.0	1093.0	949.0	1093.0	949.0	Ex. time > 3600	1093.0

Table 18: Number of backtracks generated in the analysis of the hardest FMs of Exp. #1 with diffferent variable selection heuristics in JaCoP

CoP. The two first columns of this table indentify the feature model used for the analysis in this row. For instance, the feature model used in the first data row was the hardest obtained in experiment #1 with 200 feature and 10% of CTC. The results have shown that the hardest feature models found in our experiment, using the heuristic "MostConstrainedDynamic", were trivially solved by some of the others heuristics. Hence, for instance, the hardest model in the range of 800 features and 10% CTC produced 5.3 million backtracks when usingthe heuristic "*MostConstrainedDynamic*", and only 43 backtracks when using the heuristic "*SmallestMin*". This finding supports our working hypothesis: 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 standard set of problems, random or not, is therefore not sufficient for a full evaluation of the performance of different tools. Instead, as in our approach, the characteristics of the techniques and tools under evaluation must be carefully examined to identify their strengths and weaknesses providing helpful information for both users and developers.

4.5. Discussion

As a part our evaluation, we also studied the characteristics of the hardest feature models generated for each size range using ETHOM in the experiments with CSP, SAT and BDD solvers, presented in Table 19. The data reveals that the models generated have a fair proportion of all different relationships and constraints. This is interesting since ETHOM was free to remove any type of relationship or constraints from the model if this helped to make it harder, but this did not happen in our experiments. We 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 percentages of each modelling element are considerably small. More importantly, the average percentages found are very similar to those of feature models found in the literature. In [65], She et al. studied the characteristic 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[2] (between 37% and 46.3% in our models), 16% of set relationships[3] (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 detected by ETHOM might be therefore 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 analyzed 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 a 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 significantly harder feature models than those found using random search. For a further exploration of this finding, we compared the results obtained in our experiments with the execution times and memory consumption produced by large-scale random models. More specifically, we generated 100 random feature models with 10,000 features and 20% of CTCs and recorded the execution times invested 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[4], far from the 27 minutes required by the hardest feature models found by ETHOM in the ranges 500-1000 features. Similarly, we generated 100 random feature models with 500 features and 10% of CTCs and recorded the size of the

[2] Subfeatures in alternative an or-relationships

[3] Alternative and or-relationships

[4] These times were mainly invested in 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.

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 found by ETHOM in the range of 100 features. These results clearly show the potential of ETHOM to find hard feature model of realistic size that are likely to reveal deficiencies in analysis tools rather using average large-scale random models.

	CSP Solver			SAT Solver			BDD Solver		
Modelling element	**Min**	**Avg**	**Max**	**Min**	**Avg**	**Max**	**Min**	**Avg**	**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	68.8	31.6	48.1	58.9	35.7	51.5	69.0

Table 19: Statistics of the hardest feature models found in our experiments.

As previously mentioned, ETHOM was able to find several models harder to analyse than those found in larger size ranges. This was expected since generating large feature models implies searching in large search spaces where finding an optimum is harder. Results suggest, however, that it would still be possible to find larger and harder feature models than those found in smaller ranges by performing more exhaustive searches, i.e. allowing the algorithm to run for a longer number of generations.

Finally, we may remark that 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 to find promising individuals during initialization. We also found that the maximum timeout of 30 minutes was insufficient in some size ranges (e.g. 250 features, 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.

4.6. Statistical Analysis

The goal of statistical analysis is to provide formal and quantitative evidences showing that the algorithm works and that the results were not obtained by mere chance. In fact, this type of analysis is considered mandatory in fields such as data mining and bio-

informatics with a long experience in the analysis of experimental data. The statistical analysis of the data was performed using the SPSS 17 statistical package [32].

Statistical analysis is usually performed by formulating two contrary hypothesis. The first hypothesis is referred to as *null hypothesis* (H_0^i) and assume that the algorithm has no impact at all on the goodness of the results obtained, i.e. there is no difference between our algorithm and random search. Opposite to the null hypothesis, an *alternative hypothesis* (H_1^i) is formulated, stating that the algorithm has a significant effect in the quality of the results obtained. Statistical tests provide a probability (named *p-value*) ranging in [0,1]). The lower the p-value of a test, the more likely that the null hypothesis is false and the alternative hypothesis is true, i.e. the algorithm works. Alternatively, high p-values indicates more chances of the null hypothesis being true i.e. the algorithm 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. prove that the algorithm is actually working.

The techniques used to perform the statistical analysis and obtain the p-values depend on whether the data follow a normal frequency distribution or not. The former assumes that data has come from a type of probability distribution and makes inferences about the parameters. The latter makes no assumptions at all. After some preliminary tests (Kolmogorov-Smirnov [39, 67] and Shapiro-Wilk [64] 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 [45] to the experimental results obtained with our evolutionary algorithm and random search. Tables 22 and 23 show the results of these tests in SPSS for the 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, tests rejected 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 previous sections, clearly shows the great superiority of our algorithm when compared to random search. Only when the percentage of cross tree constraints (CTC) was 10% in Experiment #1, statistical test accepted some null hypotheses. As explained in Section 5, this problem is due to the small complexity of the analysis on those models. This problem makes our fitness landscape extremely flat, with scarce and disperse points of high fitness, where a random algorithm can find solutions nearly as good as those found by our evolutionary algorithm. In experiment #2 all null hypotheses where rejected except for the last one. In this last hypothesis, the number of features, and percentage of cross tree constraints becomes bigger, and consequently it is easier for the random algorithm find hard feature models. Moreover, as described in section 4.2, in this case the maximum timeout of 30 minutes was reached frequently hindering the evolutionary program from evolving toward promising solutions. This effect gives the random algorithm more chance to find solutions similar to those obtained with ETHOM.

For more details about statistical tests and their meaning we refer the reader to [73].

5. 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 minimize 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 between the results obtained using random and evolutionary search. Regarding the generation of random feature models, we avoided the risk of creating syntactically incorrect models as follows. First, we used a publicly available algorithm for the random generation of feature models previously used by the community. 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 minimize this risk, we imposed the same generation constraints to both our random and evolutionary generators. More specifically, both generators receive 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 sharing more than one CTC. A related limitation of current ETHOM encoding is that it disables the possibility of having 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. Finally, the experiments were executed in a cluster of virtual machines running in a powerful cloud of servers for computing-intensive tasks which provided us with a stable and efficient experimental platform.

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 generalized. This can be mainly divided into limitations of the approach and generalizability of the conclusions.

Regarding the limitations, 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 4.1, this limitation is due to the extremely flat shape of fitness landscape found in simple problems in which most fitness values are equal or close to zero. Another limitation of the experimental approach is that ex-

periments 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 hard feature models we intend to find. As the analysis of those promising feature models becomes more time consuming and memory intensive, evaluating fitness function becomes a difficult task leading to a collapse in the experiment or to a nearly nil advance of the experiment execution along time. 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 emphasize 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 generalization of the conclusions, in our experiments, we used two different analysis operations which could seem not to be sufficient to generalize the conclusions of our study. We remark, however, that these operations are currently the most quoted in the literature, have different complexity and, more importantly, are the basis for the implementation of many other analysis operations on feature models [11]. Thus, feature models that are hard to analyze for these operations would certainly be hard to analyze for those operations that use them as an auxiliary function making our results extensible to other analyses. Similarly, we just used two different 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 CTC) 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 share more than one CTC for simplicity (see Section 3.2). 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 [54]) which could slightly affect the generalization of our results. These cycles will be allowed in future version of our algorithm.

6. Related work

In the next sections, we discuss the related works in the areas of software product lines, search-based testing and performance evaluation of CSP and SAT solvers.

6.1. Software product lines

A number of authors have used realistic feature models to evaluate and show the performance of their tools [4, 10, 28, 29, 35, 37, 50, 49, 55, 56, 68, 71, 75]. 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 [41] with 287 features, graph product line [43] with up to 64 features and BerkeleyDB [38] with 55 features. Although there are reports from the industry of feature models with hundreds

or even thousands of features [8, 42, 70], 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 [13, 12, 24, 29, 48, 52, 60, 78, 79, 80, 83, 84] or trying to imitate the properties of the realistic models found in the literature [26, 49, 68, 71]. More recently, some authors have suggested looking for tough and realistic feature models in the open source community [14, 22, 54, 65, 66]. As an example, She et al. [66] 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 experimentation, there is a clear ascendant tendency ranging from the model with 15 features used in 2004 [9] to models with up to 10,000 and 20,000 features used in the recent years [26, 49, 52, 71, 78]. These findings reflect an increasing concern to evaluate and compare the performance of different solutions using complex feature models. This also suggests that the only mechanism used to increase the complexity of the models is by increasing its size. When compared to previous works, our approach is the first one using a search–based strategy to exploit the internal weaknesses of the tools and techniques under evaluation rather than simply using large random 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 better adapts to their needs.

The application of genetic algorithms to the context of software product lines was explored by Guo et al. [26]. In their work, the authors proposed a genetic algorithm called GAFES for optimized feature selection in feature models, e.g. selecting the set of features that minimizes the total cost of the product. Compared to their work, our approach differs in several aspects. First, our work addresses a different problem domain, hard feature model generation. Second, ETHOM produces optimum feature models while GAFES produces optimum product configurations. This means that both algorithms bear no resemblance and face different challenges. For instance, GAFES uses a standard binary encoding to represent product configurations meanwhile ETHOM uses a custom array encoding to represent feature models of fixed size.

Pohl et al. [56] 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 [69]. 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 a 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 random models of larger size. 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 model

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 exploit the strengths and weaknesses of analysis tools.

During the preparation of this article, we presented a novel application of ETHOM in the context of reverse engineering of feature models [44]. More specifically, we used ETHOM to search for a feature model that represents a specific set of products provided as an input. The results showed that our algorithm was able to find feature models that represent a superset of the desired products with a small number of generations. This contribution supports our claims about the generalizability 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 random models which have proved to be adequate 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.

6.2. Search-based testing

Regarding related works in search-based testing, Wegener et al. [76] were the first ones using 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 input parameters. Their results showed that genetic algorithms obtained more extreme execution times with equal or less testing 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 optimization 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, tabu search or simulated annealing. In our approach, we used evolutionary algorithms inspired by the work of Wegener et al. and their promising results in a related optimization problem, i.e. generation of input values maximizing 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) [30] are a subclass of evolutionary algorithms in which solutions are encoded using bit strings. This type of encoding presents difficulties to represent the hierarchical structure of feature models and therefore we discarded their use. Genetic Programming (GP) [40] is another variant of evolutionary algorithms in which solutions are encoded as trees. This encoding is commonly used to represent software 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 size. We identified several problems for the application of GP in our work. First, 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.

6.3. 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[5]. The underlying problems of CSP and SAT solvers are NP-complete that means that every CSP and SAT solver has an exponential worst case runtime. This makes performance 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 [31]. Also, several competitions are held every year to rank the performance of the participants' tools. As an example, 26 solvers took part in the SAT competition[6] in 2011.

CP solvers use three main types of problems for performance evaluation: problems from realistic domains (e.g. hardware design), random problems and hard problems. Both, random and hard problems are automatically generated and are often forced to have at least one solution (i.e. be satisfiable). The CP research community realized long ago of the benefits of using hard problems to test the performance of their tools. In 1997, Cook and Mitchell [19] 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 SAT and CSP hard problems [82, 5].

A common strategy to generate hard CSP and SAT problems is by exploiting the known as the *phase transition phenomenon* [63, 82]. This phenomenon establishes that for many NP-complete problems the hardest instances occur between the regions 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 [11]. Also, CP solvers focus on a single analysis operation (i.e. satisfiability) for which there exists a number of well

[5] A SAT problem can be regarded a subclass of CSP with only boolean variables.
[6] `http://www.satcompetition.org`

known algorithms. In the analysis of feature models, however, more than 30 analysis operations has been reported with no standard algorithms nor in-depth complexity analyses for them. In this scenario, we believe that our approach may be helpful to generate hard problems and study their complexity leading to a better understanding of the analysis operations and the performance of analysis tools.

We identify two main advantages in our work when compared to the systematic generation of hard CP problems. First, our approach is generic being applicable 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 to CP related works that focus on the search of input problem only in specific areas (transition phase region).

Overall, we conclude that related works 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.

7. Conclusions and future work

In this paper, we presented ETHOM, a novel evolutionary algorithm to solve optimization 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 random 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. When compared to previous works, our approach is the first one using a search-based strategy to exploit the internal weaknesses of the tools and techniques under evaluation rather than simply using large-scale random 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 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 the size of the models has an important impact, but not decisive, in the performance of analysis tools. Also, we found that the hard feature models generated by ETHOM had similar properties to the realistic models found in the literature. This means that the long execution times and high memory consumption detected 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 first of 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) 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 as 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 optimization of non–functional properties in other analysis operations or even different automated treatments (e.g. feature models refactorings). The application of our algorithm to minimization problems is also an open issue in which we have started to obtained 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 [15, 61].

8. Acnowledgements

We would like to thank Dr. Don Batory, Dr. Javier Dolado, Dr. Arnaud Gotlieb, Dr. Andreas Metzger, Dr. Jose C. Riquelme and Dr. Javier Tuya whose comments and suggestions helped us to improve the article substantially. We would also like to thank Dr. M. Mendonca for kindly sending us a standalone version of SPLOT to be used in our evaluation.

This work has been partially supported by the European Commission (FEDER) and Spanish Government under CICYT project SETI (TIN2009-07366) and the Andalusian Government projects ISABEL (TIC-2533) and THEOS (TIC-5906).

http://www.juntadeandalucia.es/organismos/economiainnovacionyciencia.html

References

[1] D.H. Ackley. *A connectionist machine for genetic hillclimbing*. Kluwer Academic Publishers, Norwell, MA, USA, 1987.

[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 January 2012.

[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] T. Back, D.B Fogel, and Z. Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, UK, 1997.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] BeTTy Framework. `http://www.isa.us.es/betty`, accessed January 2012.

[16] BigLever. Biglever software gears. `http://www.biglever.com/`, accessed January 2012.

[17] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Autonomic computing through reuse of variability models at runtime: The case of smart homes. *Computer*, 42(10):37–43, October 2009.

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

[19] 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.

[20] FaMa Tool Suite. `http://www.isa.us.es/fama/`, accessed January 2012.

[21] Feature Modeling Plug-in. `http://gp.uwaterloo.ca/fmp/`, accessed January 2012.

[22] 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.

[23] N. Gamez and L. Fuentes. Architectural evolution of famiware using cardinality-based feature models. *Information and Software Technology*, 2012. In press.

[24] 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.

[25] D.E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. 1991.

[26] 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.

[27] S. Hallsteinsen, M. Hinchey, P. Sooyong, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93 –95, april 2008.

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

[29] 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.

[30] J.H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975.

[31] 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.

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

[33] JaCoP. `http://jacop.osolpro.com/`, accessed January 2012.

[34] JavaBDD. `http://javabdd.sourceforge.net/`, accessed January 2012.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] J.R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[41] 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.

[42] 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.

[43] 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.

[44] 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 *4th Symposium on Search Based Software Engineering*, 2012. To appear.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] Moskitt Feature Modeler. `http://www.pros.upv.es/mfm`, accessed January 2012.

[52] 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.

[53] J.A. Parejo, P. Fernandez, and A. Ruiz-Cortés. QoS-aware services composition using tabu search and hybrid genetic algorithms. In *Proceeding of the workshops of the JISB2008*, volume 2, pages 55–66, 2008. ISSN: 1988-3455.

[54] 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.

[55] 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.

[56] 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.

[57] pure::variants. `http://www.pure-systems.com/`, accessed January 2012.

[58] Sat4j. `http://www.sat4j.org/`, accessed May 2010.

[59] 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.

[60] 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.

[61] 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.

[62] S. Segura, R.M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *International Conference on Software Testing, Verification and Validation*, pages 35–44, Paris, France, 2010. IEEE press.

[63] B. Selman, D.G. Mitchell, and H.J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1-2):17–29, 1996.

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

[65] 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.

[66] 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.

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

[68] 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.

[69] S.P.L.O.T.: Software Product Lines Online Tools. `http://www.splot-research.org/`, accessed January 2012.

[70] 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.

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

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

[73] I. Valiela. *Doing science: design, analysis, and communication of scientific research*. Oxford University Press New York, 2001.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] 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.

[81] J. White, D.C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating product-line variant selection for mobile devices. In *Proceedings of the 11th International Software Product Line Conference*, pages 129–140, Washington, DC, USA, 2007. IEEE Computer Society.

[82] 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.

[83] 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.

[84] 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.

A. Statistical Analysis Data

Tests of Normality				
Features	CTC	Algorithm	Kolmogorov-Smirnov (a)	Shapiro-Wilk
			p-value	p-value
200	10	ETHOM	.000	.000
		RANDOM	.001	.000
	20	ETHOM	.003	.001
		RANDOM	.000	.000
	30	ETHOM	.000	.000
		RANDOM	.000	.000
	40	ETHOM	.000	.000
		RANDOM	.004	.000
400	10	ETHOM	.000	.000
		RANDOM	.000	.000
	20	ETHOM	.000	.000
		RANDOM	.000	.000
	30	ETHOM	.000	.000
		RANDOM	.000	.000
	40	ETHOM	.000	.000
		RANDOM	.000	.000
600	10	ETHOM	.000	.000
		RANDOM	.000	.000
	20	ETHOM	.000	.000
		RANDOM	.000	.000
	30	ETHOM	.000	.000
		RANDOM	.011	.001
	40	ETHOM	.000	.000
		RANDOM	.000	.000
800	10	ETHOM	.000	.000
		RANDOM	.000	.000
	20	ETHOM	.000	.000
		RANDOM	.000	.000
	30	ETHOM	.000	.000
		RANDOM	.000	.000
	40	ETHOM	.000	.000
		RANDOM	.045	.015
1000	10	ETHOM	.000	.000
		RANDOM	.000	.000
	20	ETHOM	.000	.000
		RANDOM	.000	.000
	30	ETHOM	.000	.000
		RANDOM	.000	.000
	40	ETHOM	.000	.000
		RANDOM	.004	.000
a. Lilliefors Significance Correction				

Table 20: Experiment #1 Normality test results

Tests of Normality				
Features	CTC	Algorithm	Kolmogorov-Smirnov (a)	Shapiro-Wilk
			p-value	p-value
50	10	RANDOM	.200*	.639
		ETHOM	.200*	.732
	20	RANDOM	.200*	.078
		ETHOM	.089	.027
	30	RANDOM	.034	.004
		ETHOM	.003	.000
100	10	RANDOM	.083	.007
		ETHOM	.005	.001
	20	RANDOM	.001	.000
		ETHOM	.000	.000
	30	RANDOM	.000	.000
		ETHOM	.000	.000
150	10	RANDOM	.194	.009
		ETHOM	.062	.009
	20	RANDOM	.008	.001
		ETHOM	.000	.000
	30	RANDOM	.005	.002
		ETHOM	.000	.000
200	10	RANDOM	.000	.000
		ETHOM	.014	.001
	20	RANDOM	.200*	.002
		ETHOM	.032	.001
	30	RANDOM	.000	.000
		ETHOM	.200*	.499
250	10	RANDOM	.000	.000
		ETHOM	.002	.000
	20	RANDOM	.083	.030
		ETHOM	.200*	.155
	30	RANDOM	.200*	.475
		ETHOM	.200*	.088
a. Lilliefors Significance Correction				
*. This is a lower bound of the true significance.				

Table 21: Experiment #2 Normality Tests results

Test Statisticsa		
Features	CTC	p-value
200	10	.537
	20	.000
	30	.000
	40	.000
400	10	.289
	20	.000
	30	.000
	40	.000
600	10	.360
	20	.000
	30	.000
	40	.000
800	10	.000
	20	.000
	30	.000
	40	.000
1000	10	.123
	20	.000
	30	.000
	40	.000

Table 22: Experiment #1 Test Statistics

Test Statistics		
Features	CTC	p-value
50	10	.000
	20	.000
	30	.000
100	10	.000
	20	.000
	30	.000
150	10	.000
	20	.000
	30	.000
200	10	.000
	20	.000
	30	.000
250	10	.000
	20	.000
	30	.854

Table 23: Experiment #2 Test Statistics