Monte Carlo Tree Search for Feature Model Analyses: a General Framework for Decision-Making

Jose-Miguel Horcas
University of Seville
Seville, Spain
jhorcas@us.es

José A. Galindo
University of Seville
Seville, Spain
jagalindo@us.es

Ruben Heradio
National Distance Education University (UNED)
Madrid, Spain
rheradio@issi.uned.es

David Fernandez-Amoros
UNED
Madrid, Spain
david@issi.uned.es

David Benavides
University of Seville
Seville, Spain
benavides@us.es

ABSTRACT

The colossal solution spaces of most configurable systems make intractable their exhaustive exploration. Accordingly, relevant anal-yses remain open research problems. There exist analyses alterna-tives such as SAT solving or constraint programming. However, none of them have explored simulation-based methods. Monte Carlo-based decision making is a simulation-based method for deal-ing with colossal solution spaces using randomness. This paper proposes a conceptual framework that tackles various of those anal-yses using Monte Carlo methods, which have proven to succeed in vast search spaces (*e.g.,* game theory). Our general framework is described formally, and its flexibility to cope with a diversity of analysis problems is discussed (*e.g.,* finding defective configurations, feature model reverse engineering or getting optimal performance configurations). Additionally, we present a Python implementation of the framework that shows the feasibility of our proposal. With this contribution, we envision that different problems can be addressed using Monte Carlo simulations and that our framework can be used to advance the state of the art a step forward.

KEYWORDS

configurable systems,
variability modeling,
feature models,
Monte Carlo tree search,
software product lines

1 INTRODUCTION

The Automated Analysis of Feature Models (AAFM) [9, 58] is one of the most active Software Product Line (SPL) research areas in the last decade [8]. In highly configurable systems, the AAFM is a challenging task because it requires to cope with numerous interrelated features and colossal configuration spaces. For example, we find multiple operations such as counting the number of products [23, 35, 70] and optimizing configurations [30, 37, 49] that are performed over the whole configuration space. Also, there are operations that are performed over the feature models (*e.g.,* evolution [44] and reverse engineering [18, 43]). Finally, there are other analyses over the products (*e.g.,* testing [26, 57]) or over the constraints of the feature models [69].

Many of these analyses have to to deal with uncertainty, as undertaking an exhaustive exploration of the whole configuration space is usually intractable. In addition, other relevant analyses have not been explored in-depth yet. In some cases, complex computations are required to take simple actions. For instance, deciding when to include or exclude a feature in a configuration impacts the convenience and analysis of further selections [54, 55]. Moreover, those decisions are not commonly intuitive. For instance, when reverse engineering feature models, practitioners have to decide the structure of the resulting feature model in terms of the parent-child relationships [4, 43]. Other situations require tackling uncertainty because of the aforementioned combinatorial nature of the configuration space. For instance, solving analyses of probability in configuration selections according to performance is challenging [48, 53].

When coping with large search spaces, existing techniques present several drawbacks [58]. AAFMs have relied on propositional logic or SAT solving [7, 67], constraint programming [9], Binary Decision Diagrams (BDD) [22, 35], statistical analysis [33], genetic algorithms [42, 43], or metaheuristics [75], among others [25]. However, SAT solvers could face scalability problems for large-scale models [67], statistical analysis [32, 33] require the construction of BDDs, which can be intractable [70] and other approaches like genetic algorithms [42, 43] and metaheuristics [75] require to incorporate specific domain knowledge, and analyzing and inferring results from the final solutions which is not straightforward.

In this paper, we present a conceptual framework based on so-called Monte Carlo methods [39], which use randomness for deterministic problems that are difficult or impossible to solve using traditional approaches. They can be used with little or no domain knowledge, and have succeeded on difficult problems where other techniques have failed. In particular, we adopt the Monte Carlo Tree Search (MCTS) [12, 14] method. MCTS has been successfully

applied to several domains [12], standing out in game theory [65] where has been shown to scale to large search spaces such as those that typically characterize SPLs. Thus, we conjecture that MCTS will show great promise in the AAFMs. In this paper, we make the following contributions:

- We formally present our MCTS framework (Section 2).
- We identify a set of analysis problems in SPL that can be worthy of examining with the MCTS method, and map them to the conceptual framework (Section 3).
- We formally describe different problems over configurations (Section 4) and features models (Section 5) using our framework, and explain the knowledge we can infer with MCTS.
- We provide a Python implementation of our proposal (Section 6)[1].

MCTS has already had a profound impact on artificial intelligence for domains that can be represented as trees of sequential decisions, particularly games and planning problems. With this contribution, we envision that different problems can be addressed using Monte Carlo simulations and that this new approach can be of big value to advance the state of the art of the AAFMs a step forward.

2 MONTE CARLO TREE SEARCH

This section presents our conceptual framework (Section 2.1) to model problems that can be solved with Monte Carlo methods, and especially with the Monte Carlo Tree Search method (Section 2.2).

2.1 MCTS conceptual framework

Monte Carlo Tree Search (MCTS) [14] is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. The resulting search tree is then used for possible future decisions. MCTS is based on decision and game theory [62], and on Monte Carlo [39] and bandit-based methods [47], where sequential decision problems are modeled as a *Markov Decision Process* [62] with the following elements $(S, s_0, t, A, \theta, \mu)$:

- S: the set of states.
- $s_0 \in S$: the initial state which specifies how the problem is set up at the start.
- $t : S \to \mathbb{B}$: a terminal condition, which is true when the problem is over (*i.e.,* there are no more decisions to be taken) and false otherwise. States where no more decisions can be taken are called *terminal states*. The set of terminal states is called $S_T \subseteq S$.
- A: the set of valid actions. An action is valid if it can be applied to a state under a certain condition of applicability.
- $\theta : S \times A \to S$: the state transition function, which defines the result of applying an action, leading to a new successor state.
- $\mu : S \to \mathbb{R}$: *reward function* (also known as *utility*, *objective* or *payoff* function). It defines the final numeric value for a problem that ends in a terminal state $s_t \in S_T$.

As illustrated in Figure 1 (a), the initial state s_0, the set of actions A, and the transition function θ define the *search space* for the problem — *i.e.,* a tree where each node represents a state of the domain, and directed links to child nodes represent actions leading to successor states. Thus, overall decisions are modeled as sequences of $(state, action)$ pairs. Monte Carlo methods are used to decide the optimal decision (*i.e.,* choosing an action) from a given state s by running *simulations*. A *simulation* is a random or statistically biased

[1] https://github.com/diverso-lab/fm_montecarlo

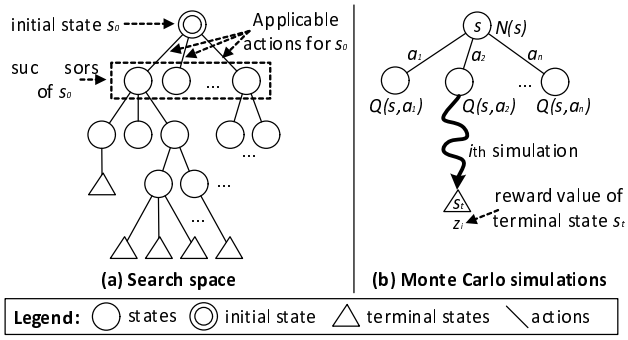


Figure 1: Problem modeled as a sequence of $(state, action)$ decisions (a), and the Monte Carlo simulations approach (b).

sequence of actions applied to the given state until a terminal terminal state s_t is found. The terminal state s_t is then evaluated using the reward function μ and obtaining a reward value z_i associated with that simulation, as shown in Figure 1 (b). Running a number of simulations N from the given state s, Monte Carlo approximates the expected reward of each action that can be performed from that state s, *i.e.,* $Q(s, a)$. The expected reward of an action is called the Q-value (also called *Monte-Carlo value* or *MC value*) [29] of that action and is defined as the mean of all rewards obtained from the simulations performed from state s choosing the action a:

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{N(s)} \mathbb{I}_i(s,a) z_i \tag{1}$$

where $N(s, a)$ is the number of times action a has been selected from state s; $N(s)$ is the number of times a simulation has been run from state s; z_i is the reward result of the ith simulation; and $\mathbb{I}_i(s, a)$ is 1 if action a was selected from state s on the ith simulation run from state s or 0 otherwise.

A great benefit of Monte Carlo methods is that the values of intermediate states do not have to be evaluated. Only the value of the terminal state at the end of each simulation is required.

2.2 The Monte Carlo Tree Search method

The MCTS method [14] extends the Monte Carlo principles by using the expected reward (Q-values) obtained from simulations to build an incremental and asymmetric search tree which is then used for subsequent decisions. As shown in Figure 2, the basic algorithm of MCTS (summarized in pseudocode in Algorithm 1) involves iteratively building and using a search tree until some predefined computational budget (*e.g.,* time, memory, number of iterations) is reached. Four steps are applied per search iteration [14]:

(1) **Selection.** Starting with the initial state s_0, the search tree is traversed by recursively applying a selection function from the root node until a leaf node s_l is reached. Several strategies for selecting the nodes in the tree can be found in [12]. The most popular one is the *upper confidence bound for trees* (UCT) [38] that, using the Q-values, attempts to balance exploitation (*i.e.,* look in areas which appear to be promising) and exploration (*i.e.,* look in areas that have not been well explored yet).
(2) **Expansion.** From the selected leaf state s_l, one or more child nodes are added to expand the tree according to the actions A.
(3) **Simulation.** A simulation is run from the new node(s) by making random actions until a terminal state is reached. The terminal state is evaluated, producing an outcome z (*i.e.,* the reward value).

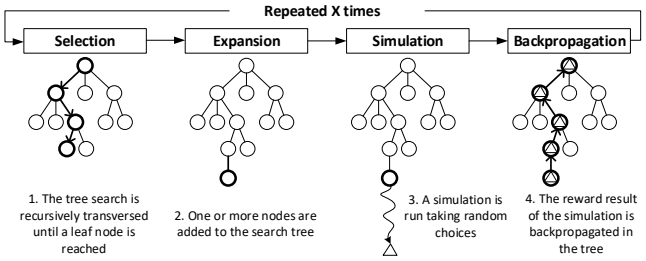


Figure 2: The MCTS approach (adapted from [14]).

Algorithm 1 General Monte Carlo Tree Search method.

```
1: procedure MCTS(s_0)
2:     while within computational budget do
3:         s_l ← SELECTION(s_0)          ▷ Apply the child selection tree policy.
4:         EXPANSION(s_l)                ▷ Add one or more nodes to the search tree.
5:         z ← SIMULATION(s_l)           ▷ Apply the default policy.
6:         BACKPROPAGATION(z, s_l)       ▷ Backup the reward.
7:     end while
8:     return BEST_ACTION(s_0)
9: end procedure
```

(4) **Backpropagation.** We update the statistics of each state in the tree that was traversed during the selection step. That is, the visit counts $N(s, a)$ are increased, and the expected reward $Q(s, a)$ is modified according to the outcome z from the simulation.

As soon as the search terminates, the best action of the initial state s_0 is selected (`BEST_ACTION`(s_0)). Several criteria are described in [15], such as choosing the action with the highest reward. In addition to the best decision, MCTS also provides useful knowledge in the form of statistics stored in the tree search that can be used to make analyses, as we will show throughout the paper.

3 AUTOMATED ANALYSIS OF FEATURE MODELS WITH MCTS

In this section we present our running example, and identify a set of problems to be formalized and analyzed with the MCTS framework.

3.1 Preliminaries and running example

Definition 3.1 (Feature, Feature model). A *feature* f is a characteristic or end-user-visible behavior of a software system [2]. A *feature model* m is a set of features F and their relationships. Formally, a feature model m is defined as a 4-tuple $(F, r, \mathcal{R}, \Pi)$:

- F is a finite set of features.
- $r \in F$ is the root feature.
- $\mathcal{R} \subseteq F \times F^n \times \lambda$ is a decomposition relation representing the relationship of a parent feature and its n sub-features with a multiplicity λ, where the lower and the upper bounds are the minimal and the maximal number of features that can be selected in a configuration. We use the notation $(f, [g_1, g_2, \ldots, g_n], \langle a..b \rangle)$ to represent the elements of R meaning that f is the parent of sub-features $g_i, 1 \leq i \leq n$ with a multiplicity of $\langle a..b \rangle$. We use $\langle 0..1 \rangle$ for optional features and $\langle 1..1 \rangle$ for mandatory features when $n = 1$; and $\langle 1..1 \rangle$ for alternative-groups and $\langle 1..n \rangle$ for or-groups when $n > 1$. For convenience, we will also use the notation $f \prec g$ to represent that feature f is the parent of feature g regardless multiplicity; and we use the notation $f \prec\prec g$ to represent that feature f is an ancestor of feature g.
- Π is a set of cross-tree constraints defined as arbitrary propositional formulas over the set of features F, *i.e.,* $\Pi \subseteq \mathbb{B}(F)$.

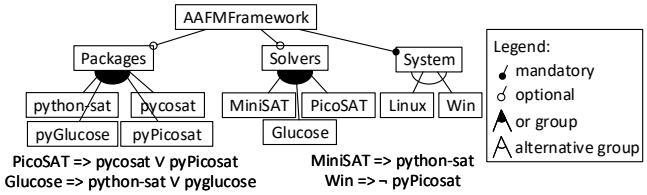


Figure 3: SPL for an AAFMs framework in Python.

Figure 3 shows a feature model representing a product line for a Python framework to support AAFMs [24]. `AAFMFramework` is the root of the feature model. The mandatory relation between the root and the `System` feature can be described by the relation (`AAFMFramework`, [`System`], $\langle 1..1 \rangle$), while the relations between the `AAFMFramework` feature and its optional children are described by the relations (`AAFMFramework`, [`Packages`], $\langle 0..1 \rangle$) and (`AAFMFramework`, [`Solvers`], $\langle 0..1 \rangle$), respectively. To reason about the models and implement the analysis operations, the framework can use a selection of `Solvers` represented by the or-group relation (`Solvers`, [`MiniSAT`, `PicoSAT`, `Glucose`], $\langle 1..3 \rangle$). Each solver will require one or more Python package which offer the implementation for that solver. For instance, the `MiniSAT` solver is provided by the `python-sat` package, while the `PicoSAT` solver is offered by the `pycosat` or by the `pyPicosat` packages. These kind of relations are represented as textual cross-tree constraints in the feature model, as for example `PicoSAT` $\Rightarrow$ `pycosat` $\vee$ `pyPicosat`. Finally, a specific version of the framework can be deployed in `Linux` or `Windows` systems, relation represented by the alternative-group (`System`, [`Linux`, `Win`], $\langle 1..1 \rangle$).

Definition 3.2 (Configuration, Partial Configuration, Valid Configuration). A *configuration* c of a feature model m is a subset of its features, *i.e.,* $c \in \mathcal{P}(F)$, meaning that features in c have been selected as part of the configuration c. Other features in F are considered as not selected as part of the configuration c. A configuration c is *partial* if there are features in F that need to be still decided in order to be selected or not selected as part of the configuration c [7]. A configuration is *valid* if and only if it fulfills all the feature dependencies of m. The feature dependencies of m are given by the set of decomposition relations $\mathcal{R}$ (*i.e.,* the tree hierarchy) and the set of cross-tree constraints Π. A partial configuration is valid if the selected features do not neglect the dependencies of m.

An example of a valid configuration of the feature model depicted in Figure 3 is {`AAFMFramework`, `System`, `Linux`, `Solvers`, `MiniSAT`, `Packages`, `python-sat`}. An example of a valid partial configuration is {`AAFMFramework`, `System`, `Solver`}.

3.2 Types of analyses with MCTS

In SPL, MCTS can be applied to find optimal decisions in problems where decisions can be difficult to handle and take because of the high number of potential configurations, products, and variants. Some of the analyses that can be performed with MCTS include:

Analyses of complex systems from simple actions. There are problems where we can easily measure the complete set of actions within the system but we are unsure of the aggregate result. For example, selecting a feature to be incorporated in a product is a very simple action to model, but analyzing how that feature selection contributes to the complete product is challenging due to

the existing relations in the feature model and the cross-tree constraints [63]. Here, MCTS can examine how each feature selection contributes to the complete product by modeling the feature selection optimization problem [20, 30, 74, 75] as a sequence of decision steps. We illustrate this type of analysis in Section 4.

Analyzing unintuitive results. There are problems where the current solution is not the only possible solution. For instance, multiple feature models can be extracted from a given set of configurations (this problem is known as *reverse engineering* of feature models [18, 43]). Without taking into account domain knowledge, features appearing in all products (*i.e.,* the core features [9]) may be considered interchangeable in the resulting feature model. Here, MCTS can help us exploring the alternatives that we may pass unnoticed if we simply observe the extracted feature model. We show this type of analysis in Section 5.

Analyses of the uncertainty. There are problems that require handling uncertainty due to the impossibility of dealing with the complete search space. An example is the optimization of configurations based on non-functional properties in large-scale feature models [30, 37]. Best configurations may be spread across the configuration space, leading to a search-based software engineering technique to deal with many local optima [42]. In this case, MCTS is useful to incorporate probability into the analysis. MCTS helps us understanding the probability distribution of the best configurations and analyzing how such distribution will impact the search-based optimization, so that we can penalize the uncertainty or incorporate it into the search-based technique.

In addition to those analyses, in general, MCTS may be used for analyses that have a probabilistic interpretation [33] or where simulation rather than optimization is the most effective decision support tool [12]. As stated by Schmid [1], Monte Carlo techniques can be promising for sensitivity analyses, but they require a sound understanding of the uncertainty in the problem to be analyzed for achieving correct and useful results.

3.3 Mapping to MCTS conceptual framework

In order to apply the MCTS method to the AAFMs, we need to formulate the problems as a sequence of $(state, action)$ decisions using the conceptual framework $(S, s_0, t, A, \theta, \mu)$ introduced in Section 2.1.

Figure 4 shows a list of SPL problems that can be described as a sequence of decisions and mapped to the MCTS conceptual framework. The most important definition is the concept of state, and thus, we classify the problems according to what a state represents. In SPLs, a state may represent a configuration of a feature model, a partial configuration, a final product, a feature model, an extended feature model, a sample of configurations, a performance model of a sample of configurations, a variation point and the set of its variants to be decided, etc. The definition of the state will depend on the problem's nature. For example, in product configuration problems, the states will represent configurations (valid or invalid) of a feature model, partial configurations, or both partial and complete configurations; while in problems dealing with the evolution of feature models, a state will represent a feature model itself. Each definition of state will lead to a different set of actions. States representing configurations will define actions that allow moving from one configuration to another (*e.g.,* actions for selecting a feature and

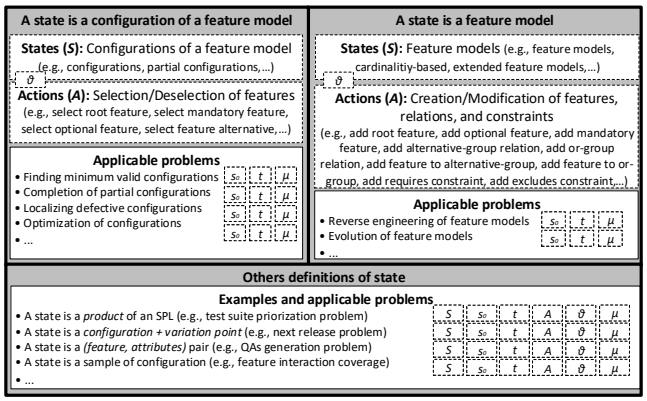


Figure 4: Mapping of SPL problems to the MCTS framework.

adding it to the configuration). States representing feature models will define actions to modify the feature models (*e.g.,* adding a new mandatory or optional feature to the model). Different definitions of states and actions will lead to a different search space.

Some of the definitions in the framework $(S, s_0, t, A, \theta, \mu)$ can be shared across several problems, while others will be specific of a particular situation or problem instance. On the one hand, the set of actions and the transition function are normally reused across different problems that share the same definition of state. For example, the actions for selecting a feature in problems where a state represents a configuration. On the other hand, the initial state s_0, the terminal condition t, and the reward function μ are problem specific or even different for a specific instance of a particular problem. For example, the initial state is different in each problem instance of the completion of partial configurations being the initial state a different input partial configuration. The terminal state can also be instance-specific, such as in the problem of the feature interaction coverage, where a state represents a set of configurations and a terminal condition can be a sampling of configurations satisfying the t-wise coverage for a specific feature [57].

In the following, we detail how to model different types of SPL problems using the MCTS conceptual framework and analyze them.

4 CONFIGURATION BASED ANALYSIS

One of the most important and widely studied types of problems in AAFMs deal with feature model configurations. Example of these problems are the optimization of configurations according to non-functional properties [30, 63, 75], the completion of partial configurations [71], the localization of defective configurations in SPL testing [10, 11, 28, 31], or the diagnosis of configurations [21, 71], among other many problems. In order to analyze this kind of problems with MTCS, we first model the concepts that are shared among these problems. That is the definition of the set of states S, the set of actions A, and the state transition function θ.

The set of states S is the set of all possible configurations of the feature model. Depending on the definitions of the actions, S may consider either valid/invalid configurations or both.

The set of actions A. There are multiple possibilities to define the set of valid actions that can be performed over a given configuration. An action is valid if it can be applied to a state under a certain *condition of applicability* (CA). The most basic action is to select a random feature of F (see the formalization of the action a_0 in Appendix A). This action is independent of the relations defined

in the feature model (Definition 3.1), and thus it can be used for any other definition of feature model as long as it is based on a set of features. However, this action leads to an intractable search space with all possible valid and invalid configurations (*i.e.,* $S = \mathcal{P}(F)$) where most of the states represent invalid configurations. Note that the search space is bigger than 2^F because we can reach the same configuration through a different sequence of actions. A more convenient definition for the actions is considering the relations of the feature model, reducing the resulting search space. Following our Definition 3.1 for feature models, we define the following set of actions $A = \{a_1, a_2, a_3, a_4, a_5\}$ (see formalization of the actions in Appendix A): select root feature (a_1), select mandatory feature (a_2), select optional feature (a_3), select feature alternative (a_4), and select feature selection (a_5).

Note that the configuration is always built from scratch step by step. In each execution of the MCTS method, a unique feature will be selected following the tree hierarchy structure of the feature model. This way, we do not need to define actions for de-selecting a feature and avoid cycles in the tree search space. The successive application of the actions A assures the validity of the (partial) configurations according to the tree hierarchy structure of the feature model, but not for cross-tree constraints. We can define a generic condition of applicability for all actions so that an action can only be applied if the resulting partial configuration does not violate any relation nor constraints in the model (*e.g.,* checking it with a SAT solver).

The state transition function $\theta : S \times A \rightarrow S$ defines the result of applying an action $a \in A$ to the given configuration c. Starting from the initial empty configuration c_0 and iteratively applying the state transition function with all possible applicable actions, we could build the whole search space. However, this is an intractable task, and the Monte Carlo methods, and in particular MCTS, will explore the search space resulting of applying the transition function only to the most promising pairs of $(state, action)$.

The initial state, the terminal condition, and the reward function are specific for each problem. In the following, we show how to model three concrete problems where a state represents a configuration by providing complete definitions of the concepts $(S, s_0, t, A, \theta, \mu)$.

4.1 Localizing defective configurations

A common problem in software testing and maintenance is to identify the configurations that lead to a given defect or some other undesired program behavior [10, 28, 31]. Continuing with our running example, let us suppose we want to identify those valid configurations in our feature model (Figure 3) that present anomalies when they are deployed. Anomalies in the Python framework for AAFMs may happen due to incompatible versions of packages, deprecated libraries, or some other errors. Despite those defective configurations can be found with a search-based software engineering technique [42], localizing the feature that causes the configuration to fail is not an easy task due to the complex relations between the features, requiring complex analysis for the complete configuration. Moreover, Python packages are often updated and may cause *breaking changes*. Kästner et al. [11] define a breaking change as any change in a package that would cause a fault in a dependent package if it were to adopt that change blindly. Thus, in order to provide a robust Python Framework for AAFMs, apart from identifying the defective configurations, we need to identify those features that

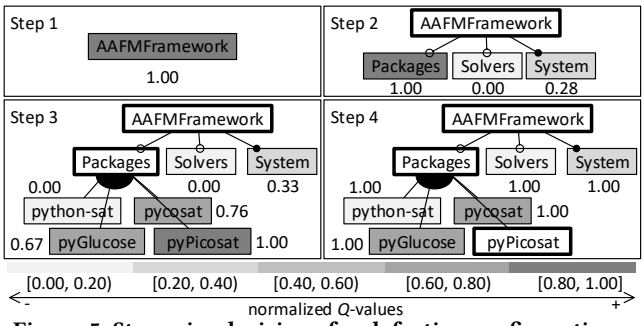


Figure 5: Step-wise decisions for defective configurations.

cause the defects. Using a step-wise decisions approach, we can infer which feature(s) is causing the configuration to fail.

Modeling the problem. We model this problem in the MCTS conceptual framework (S, s_0, t, A, f, μ) as follows:

- $S = \mathcal{P}(F)$: the set of all possible configurations of the feature model, including partial and complete configurations.
- $s_0 = \emptyset$: the initial state is the empty configuration where no feature has been already selected.
- The terminal condition t determines that a configuration is terminal if it is a valid complete configuration or no more valid actions can be applied to the configuration:

$$t(s) = \begin{cases} True, & \text{if } is\_valid(s) \vee applicable\_actions(s) = \emptyset, \\ False, & \text{otherwise} \end{cases}$$

  The `is_valid(s)` operation [9] is performed with a SAT solver.
- The set of valid actions $A = \{a_1, \ldots, a_5\}$ defined in Appendix A.
- The state transition function $\theta : S \times A \rightarrow S$ is given by the definitions of the actions and their conditions of applicability.
- The reward function μ for a terminal configuration:

$$\mu(s) = \begin{cases} errors(s), & \text{if } is\_valid(s) \wedge errors(s) > 0 \\ -1, & \text{otherwise} \end{cases}$$

  where $error(s)$ is a function that counts the number of errors that the configuration presents when it is deployed. In our running example, this value corresponds with the number of Python packages selected that raise errors when installing them. The reward value of partial and invalid configurations, as well as for those valid configurations that do not contain errors is -1.

Algorithm 2 Generic algorithm to solve a problem with MCTS.

```
1: procedure FindSolution(s0)
2:     state ← s0
3:     while not is_terminal(state) do
4:         state ← MCTS(state)                ▷ Run the MCTS (Algorithm 1).
5:     end while
6:     return state
7: end procedure
```

Solving the problem and analyzing results. We can solve this problem by consecutively applying MCTS from the initial empty configuration (Algorithm 3). Each execution of the MCTS method (line 3) will decide and add the most promising feature to the current configuration following the four steps of the MCTS method presented in Section 2.2. The most promising feature is the next feature in the tree hierarchy of the feature model (according to actions A) that moves the configuration closer to the complete valid

Algorithm 3 Finding defective configurations with MCTS.

```
1:  procedure FindDefectiveConfiguration(state)
2:      while reward(state) ≤ 0 and actions(state) ≠ ∅ do
3:          state ← MCTS(state)                          ▷ Run the MCTS (Algorithm 1).
4:      end while
5:      if reward(state) > 0 then                        ▷ Defective configuration found!
6:          return state
7:      else
8:          return False
9:      end if
10: end procedure
```

configuration with more number of errors. We modify the stopping condition of the generic Algorithm 2 so that the search will run until a configuration with errors is found or no valid actions can be applied to the current configuration (line 2 in Algorithm 3). The execution of the algorithm is illustrated step by step in Figure 5. We use a data visualization technique called *heat maps* [41, 72, 73]. A heat map encodes quantitative values as colors (like in the weather maps), so that it compacts large amounts of information (our Q-values) to bring out coherent patterns in the data (optimal feature selections over the feature model). As shown in Figure 5, we use a grey-scale heat map for each algorithm step representing the contribution of each feature selection, where darker colors mean a higher probability of finding a defective configuration if that feature is selected. Given the empty configuration as initial state, in the first step, the only action available is to add the root feature `AAFMFramework` (Step 1 in Figure 5). In Step 2, three possible features can be added to the configuration according to the set of actions A. To make a choice, MCTS performs a number of simulations (*e.g.,* 100) that consist in completing the configuration with random selections (always following the set of actions A), evaluating the number of errors for the complete configuration achieved in each simulation, and gathering the statistical outcomes of the simulations as explained in Section 2.1 — *i.e.,* the Q-values. Figure 5 shows normalized Q-values in range $[0, 1]$, being 1 the most promising feature decision. In Step 3, the `pyPicosat` package is selected as part of the configuration, while, in Step 4, any possible decision will lead to defective configuration (*i.e.,* all candidate features are Q-values 1). This suggests that the previous choice (`pyPicosat`) is the feature provoking the failure. The algorithm will continue completing the configuration with a valid selection of features (*e.g.,* selecting mandatories features), but MCTS has only needed four steps to discover the problematic feature. The algorithm finishes when it find a valid defective configuration, as for example the final state found: {`AAFMFramework`, `Solvers`, `System`, `Linux`, `Packages`, `pyPicosat`, `PicoSAT`}. As Figure 6 shows, when deploying that configuration in Python some errors raise: the package `pyPicosat` cannot be correctly installed in Linux. In fact, following this procedure, we found that deploying any configuration containing the `pyPicosat` feature in the Linux system, leads to failures, so we may decide to update the constraint (`Win` ∨ `Linux` ⇒ ¬ `pyPicosat`) converting it into a dead feature [9].

Using the knowledge gathered by MCTS in the tree search, we can infer two interesting results: (1) which features are more probable to be the cause provoking the defect in the configuration; and (2) which features are contributing more to the solution obtained, so that we may find additional defective configurations by following the sequence of feature selections done by MCTS to find the current

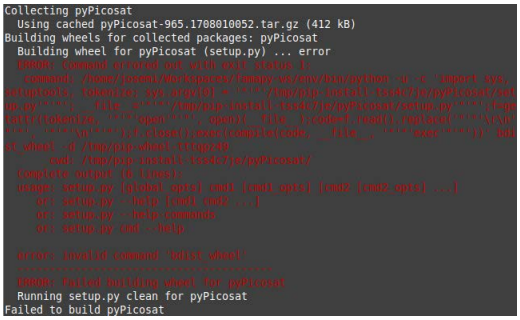


Figure 6: Deploying a defective configuration.

configuration. In the following section, we show how we can solve another problem where a state represents a configuration by only modifying the reward function and reusing the other definitions.

4.2 Finding minimum valid configurations

In our running example, a requirement for the development of the AAFMs framework in Python is to depend on the smallest number of third-party packages as possible. Thus, we are interested in finding a valid configuration with the minimum number of features.

Modeling the problem. We reuse the definitions of (S, s_0, t, A, θ) of the previous problem and modifying the reward function μ.

The reward function μ counts the difference between the number of features in the feature model ($|F|$) and the number of features in the configuration represented by the state s.

$$\mu(s) = \begin{cases} |F| - |s|, & \text{if } is\_valid(s) \\ -1, & \text{otherwise} \end{cases}$$

Solving the problem and analyzing results. We use generic Algorithm 2 where the terminal condition checks if the current state is a complete valid configuration. Figure 7 shows the number of decisions (features) taken by MCTS to find minimum valid configurations starting from the empty configuration. We run 30 executions for each number of iterations (simulations). Using the complete version of the AAFMs Python framework (see Section 6), we can observe as the number of decisions decreases as long as we increase the number of simulations, improving the solutions found.

In this problem, the tree search built by MCTS contains statistical information regarding decisions to select the minimum set of features to form a valid configuration. Figure 8 shows the resulting heat map with the accumulated Q-values when we use a partial configuration as initial states (as we will show in the following section). Q-values capture the expected reward of a decision if we decide to make such choice. When using the empty configuration as initial state, higher Q-values indicate which feature may be selected to obtain a minimum valid configuration, and features in darker colors will approximate to the core features. In addition, the heat map shows the feature `pyPicosat` in blank, indicating that it has never been considered in a decision. Effectively, the constraint we updated (`Win` ∨ `Linux` ⇒ ¬ `pyPicosat`) prevents that feature from being part of any configuration, indicating that it is a dead feature.

4.3 Completion of partial configurations

The previous problem of finding minimum valid configurations can be seen as a specialization of the problem of completing partial configurations [71]. The completion of partial configuration problem

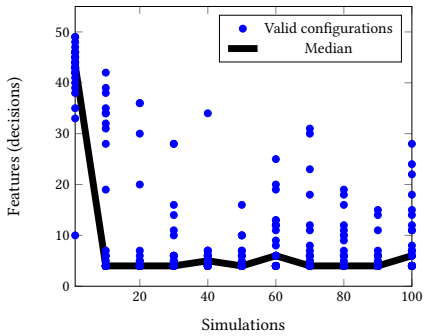


Figure 7: Finding minimum valid configurations.

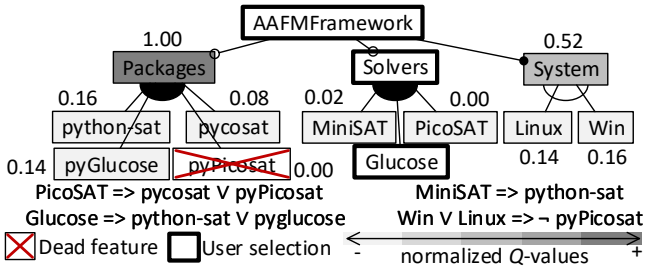


Figure 8: Heat map for completion of partial configurations.

consists of finding the set of non-selected features necessary for getting a complete valid configuration. While in a complete configuration each feature is decided to be either present or absent in the resulting configuration, in partial configurations, some features are undecided (see Definition 3.2). In our case study, let us suppose we have decided to use the `Glucose` solver in our AAFMs framework. We are interested in finding the minimum valid complete configuration with such user's requirement.

Modeling the problem. We modify the initial state s_0, while leaving the other definitions (S, t, A, θ, μ) as in the previous problem. In order to provide a valid initial state s_0 to the MCTS method so that s_0 does not violate the tree hierarchy of the feature model and allows applying our actions $A = \{a_1, \ldots, a_5\}$, we pre-process the initial configuration provided by the user by recursively selecting all parents for the features already selected. If the resulting partial configuration does not violate the tree hierarchy nor cross-tree constraints, we can use it as initial state for MCTS. In other case, the partial selection made by the user is not valid.

Solving the problem and analyzing results. We use the features {`AAFMFramework`, `Solver`, `Glucose`} as initial configuration and execute Algorithm 2 where the terminal condition checks if the current state is a valid complete configuration, as in the previous problem. Figure 8 shows the resulting heat map for completing the partial configuration given as initial state by the user with the minimum valid selections. Features in darker colors indicate selections to be first made in order to get closer to a complete configuration, as for example the `Packages` and the `System` features. The `Packages` features appears with a higher normalized Q-value, indicating that MCTS has first explored that feature (in contrast to the mandatory feature `System`). That is because a complete configuration needs to include both features, satisfying the cross-tree constraints (*i.e.,* the `Glucose` solver is implemented by the `python-sat` or the `pyglucose` package), so that the `Packages` feature must be selected. To satisfy the constraint, the `python-sat` or the `pyglucose` package must be selected. They appear with a higher normalized Q-value than the other alternative packages. Note that how other features like `pycosat` (0.08) or the solver `MiniSAT` (0.02) are not strictly necessary to complete the configuration, but has been marked as possible candidates. Remember that MCTS is based on simulations and probabilities and those feature selections have also been explored resulting in valid configurations.

5 FURTHER ANALYSES

Analyses with MCTS can also be performed over other concepts beyond the configuration space of an SPL, such as feature models, extended feature models, variation points and variants, or products. This section shows how to model and analyze a problem where the concept of state can represent a feature model. Examples of these problems are the reverse engineering of feature models [4, 43], the extraction of feature models from propositional formulas [19], or the evolution of feature models [44]. Here, we illustrate the reverse engineering of feature models problem [4, 43].

5.1 Reverse engineering of feature models

A well-known problem in SPLs is to synthesize a feature model from a set of configurations automatically. Given a set of feature combinations present in an SPL (*i.e.,* a set of configurations), the goal is to extract a feature model that represents all the configurations. Formally, let be C_i the set of configurations given as input. F_i is the set of features present in C_i. The problem is to build a feature model m with features in F_i so that $C_i \subseteq C_m$ where C_m is the set of valid configurations of the feature model m.

Modeling the problem. We formulate the problem with the following definitions of $(S, s_0, t, A, \theta, \mu)$:

The set of states S is the set of all feature models that can be built with any combination of the input features F_i following the Definition 3.1 of feature model. Thus, $S = \{m | m = (F, r, \mathcal{R}, \Pi)\}$ where $F \in \mathcal{P}(F_i)$ and $\Pi \subset \{f \Rightarrow g, f \Rightarrow \neg g | f, g \in F_i\}$[2], r is the root of the feature model, and the set of relations $\mathcal{R}$ is the same as in Definition 3.1 (*i.e.,* optional, mandatory, alternative, and or).

The initial state s_0 is the empty feature model, with no features.

The terminal condition t determines that a feature model is terminal if it contains all features given as input (*i.e.,* $F = F_i$).

The set of actions A includes 9 actions ($A = \{b_1, \ldots, b_9\}$) to be performed over a feature model (see formalization of the actions in Appendix A): add the root feature (b_1), add an optional feature (b_2), add a mandatory feature (b_3), add an or-group relation (b_4), add an alternative-group relation (b_5), add a feature to an existing or-group (b_6), add a feature to an existing alternative-group (b_7), add a "requires" constraint (b_8), and add a "excludes" constraint (b_9). Each action is also applicable under a certain condition of applicability (CA). An invariant condition of applicability that holds for all actions is that the features to be added are not already in the feature model (*i.e.,* $\exists f \in F_i, f \notin F$).

The state transition function θ defines the result of applying an action $a \in A$ to the given feature model m.

[2]To simplify the problem we consider here only "requires" and "excludes" constraints.

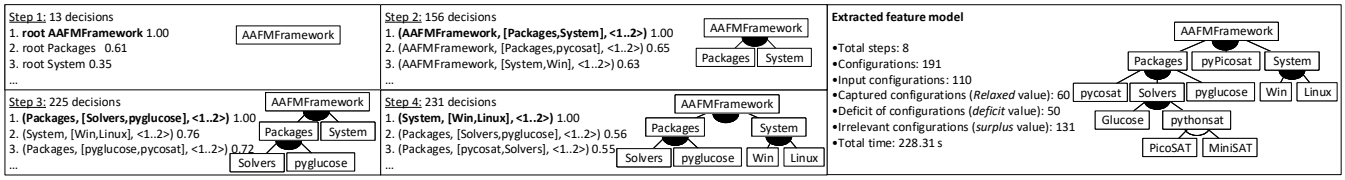


Figure 9: Step-wise decisions for reverse engineering of feature models.

The reward function μ : for a terminal feature model is a combination of two objective functions extracted from [43]:

$$R(s) = Relaxed(s) - MinDiff(s)$$

where $Relaxed(s)$ expresses the concern of capturing primarily the configurations provided as input. Its value is the number of configurations in C_i that are valid according to the feature model m represented by this state. We want to maximize the $Relaxed(s)$ value. $MinDiff(s)$ is a minimal difference function expressing the concern of obtaining a closer fit to the configurations provided C_i while other configurations are not relevant. Its value is the sum of the number of configurations in C_i that are not contained in the configurations C_m of the feature model (also called *deficit* value), and the number of configurations in C_m that are not contained in the required input configurations C_i (also called *surplus* value). So $MinDiff(s) = deficit(s) + surplus(s)$, value to be minimized.

Solving the problem and analyzing results. We use as input the set of 110 configurations of our running example (excerpt shown in Figure 3). We can use the same generic Algorithm 2 to solve this problem. Starting from the empty (void) feature model (*i.e.,* initial state), MCTS will incorporate in each decision step a feature or a cross-tree constraint to the feature model until all features contained in the given configurations are present in the feature model. Figure 9 shows the first four decision steps made by MCTS and the final extracted feature model. The resulting feature model looks similar to the expected one (Figure 3) with some significant differences. It leads to a total of 191 configurations, 60 of which correspond with the 110 configurations provided as input, presenting a deficit of 50 configurations (almost half of the configurations). Such deficit may be corrected with a couple of manual changes over the resulting feature model. In each step, MCTS has run 1000 simulations, meaning that to make a decision, it has completed up to 1000 random feature models, enumerating their configurations with a SAT solver, and calculating the reward value for each model.

The most interesting result obtained from MCTS is the information gathered in its tree search over the process. In this case, the tree search contains statistical information about how promising is to add a specific feature, relation, or constraints. As illustrated in Figure 9, for each step, we show the best three possible decisions (with normalized Q-values), highlighting the choice selected. For example, step 1 adds the root feature, where the three most promising options (from 13 candidates) are to use `AAFMFramework`, `Packages`, or `System` as root. In the following step, or-group relations are added with features `Packages` and `System` as children, but the tree search offers information about how promising other alternatives are out of a total of 156 possibilities. This can be seen as user-assistant to make better decisions, offering to the user alternatives so that she does not have to blindly rely on the result of a black box tool. MCTS can be integrated as part of a recommender system [51, 56, 61] to assist the user.

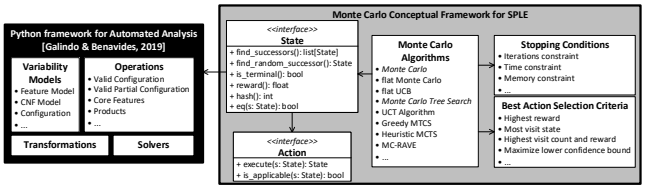


Figure 10: Monte Carlo conceptual framework architecture.

6 PROOF OF CONCEPTS

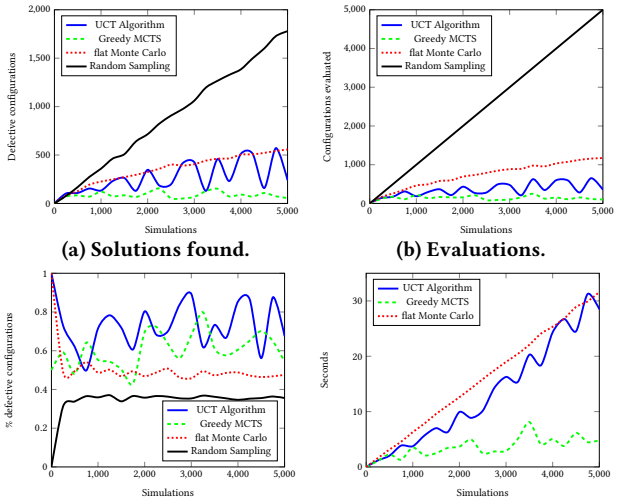
This section illustrates the applicability of our proposal by providing an implementation of the MCTS conceptual framework[3] built on top of the Python framework for AAFMs proposed in [24].

Figure 10 overviews the core architecture which defines the two main interfaces (`State` and `Action`) to be implemented in order to model and solve a problem with Monte Carlo methods — *i.e.,* the $(S, s_0, t, A, \theta, \mu)$ modeling concepts. The `State` interface specifies the necessary methods to explore the whole search space, so that from a given initial state s_0 we can reach all states in S. The `State` interface has to be implemented only once defining the state transition function θ (`find_successors()` and `find_random_successor()`), the `is_terminal()` condition t, and the `reward()` function μ. The `Action` interface is defined for each applicable actions. Different Monte Carlo algorithms can be then applied. Each Monte Carlo algorithm can be configured with a stopping condition such as a time, memory, or iteration constraint, and with a selection criteria for the best action decision. For instance, to select the child with the highest reward, the most visited child, the child with both the highest visit count and the highest reward, or the child which maximizes a lower confidence bound [15]. At the time of writing this paper, the following Monte Carlo algorithms are available:

- **UCT Algorithm.** An implementation of MCTS that builds a search tree and uses the *upper confidence bound for trees* (UCT) [38] selection strategy. This strategy favors actions with a higher Q-value but allows at the same time to explore those actions that have not yet been sufficiently explored.
- **Greedy MCTS.** A best-first strategy that favors exploitation against exploration.
- **flat Monte Carlo.** A basic Monte Carlo method with random action selection and no tree growth.

To show the applicability of our Monte Carlo methods, we apply them for finding defective configurations in two real-world SPLs: the JHipster Web development stack [31], and the complete version of the Python framework for AAFMs [24]. On the one hand, we choose the jHipster SPL because its configuration space (26,256 configurations) has been fully evaluated in [31] (having 9,376 defective configurations, *i.e.,* 35.70%) and can be used to verify the results obtained with our methods. On the other hand, the complete `AAFMFramework` product line presented in Section 3.1 has 53 features,

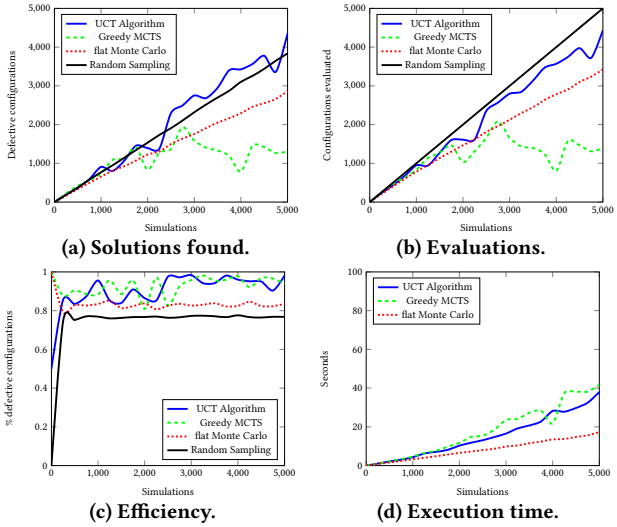
[3] https://github.com/diverso-lab/fm_montecarlo



(a) Solutions found. (b) Evaluations.

(c) Efficiency. (d) Execution time.

Figure 11: Finding defective configurations in jHipster.



(a) Solutions found. (b) Evaluations.

(c) Efficiency. (d) Execution time.

Figure 12: Finding defective configurations in the complete configuration space of the AAFM Python framework.

26 relations, and 10 cross-tree constraints, leading to a total of $3.1264 \cdot 10^9$ configurations. It serves as a large-scale configuration space where we have already identified the specific features that cause the defective configurations using the MCTS method as we did in Section 4.1. The experiments were performed on a desktop computer with Intel Core i9-9900K CPU @ 3.60GHz x 8, 32 GB of memory, Linux Mint 20.1 Cinnamon, and Python 3.9.1.

The results are summarized in Figure 11 and Figure 12 for the jHipster SPL and the AAFMs Python framework, respectively. We compare the three implemented Monte Carlo algorithms for different numbers of simulations *wrt.* a uniform random sampling strategy [32]. We present the number of configurations found with defects (a), the number of configurations evaluated (b), the efficiency as the percentage of defective configurations found *wrt.* the configurations evaluated (c), and the execution time (d). A first observation is a higher fluctuation in the MCTS (especially in the UCT Algorithm) because of the balance between exploitation and exploration [29, 38]. This helps MCTS to get out of local optima when those solutions have been sufficiently explored, but demonstrates the evidence of Lopez-Herrejón [42] about techniques that rely on randomness, which are very sensitive to the various inputs and parameters, meaning that slightly changing a value (the number of simulations in this case) can totally change how you would infer the results. The number of solutions found (a) and evaluated (b) by MCTS will depend on the distribution of the configuration space [33]. In jHipster, defective configurations are localized in regions, making MCTS focuses on that area (same configurations may be found more than once, but are evaluated once) until the region is sufficiently explored (requiring MCTS more simulations). On the contrary, in the AAFM Python framework, defective configurations are scattered through the configuration space, and MCTS will evaluate and find more with the same number of simulations.

A second observation is the efficiency of the Monte Carlo methods (c), being superior on average the UCT Algorithm (68% in jHipster and 98% in the AAFM Python framework) than the random sampling (36% and 77%, respectively), when comparing the amount of defective configurations found *wrt.* the configurations evaluated. This result is not surprising due to MCTS is a selective sampling approach, but it shows that MCTS can also be used as a search-based optimization technique in SPL as a complement of existing approaches (*e.g.,* genetic algorithms [42]). Finally, regarding the performance (d), Monte Carlo algorithms accomplish better results the longer they keep running. They are appropriate for problems that do not require achieving immediate results but require taking optimum decisions in the medium and long term.

7 RELATED WORK

Over the last decade, MCTS has been adopted as part of the solution to many problems in a variety of domains beyond AI games [65]. For instance, MCTS has achieved great success with complex real-world problems, such as combinatorial optimization to evaluate system vulnerabilities [68], constraint satisfaction problems (CSP) [6], boolean satisfiability [60], model checking [59], scheduling problems [50], and feature selection problems in the field of machine learning [17], among others [12]. However, MCTS has not been already applied in the context of the AAFMs [25]. To the best of our knowledge, Monte Carlo methods have been mainly applied in SPL from an economic point of view [13, 27, 52]. For instance, to analyze the return on investment expectations of an SPL [52] and understanding the effort required for building reusable assets [13], to compare costs and benefits of different test strategies [27], or to estimate the payoff of an SPL [34]. Also, Monte Carlo simulations have been used for validation when there is a lack of available data [1], as for example, to check the stability of solutions in SPL optimization [37, 45]. Regarding MCTS, our work is the first study that proposes applying it in SPLs.

Several work have incorporated randomness into AAFMs. Czarnecki et al. [18] introduced the concept of *probabilistic feature models* (PFM) to automate the choice propagation of features according to the constraints, and apply an entropy measure to guide the configuration process. Martinez et al [46] also estimate the feature probabilities to provide feedback to the user. Both works [18, 46] rely on historical data to extract probabilities. Heradio et al. [33] propose statistical analysis to reason on variability models. They can extract probability distributions from the whole configuration

Table 1: Comparison of the MCTS conceptual framework and Genetic Algorithms as search-based techniques for SPL.

Monte Carlo Tree Search	Genetic Algorithms
States. They represent the possible status of the problem (*e.g.,* valid/invalid and partial/complete configurations, or feature models). They do not require a special encoding.	**Population (chromosomes).** Set of candidate solutions. They represent complete configurations or feature models which need to be encoded (*e.g.,* as binary strings) and decoded to be evaluated.
Initial state. It is a unique well defined state (*e.g.,* empty or partial configuration, void feature model) that will transition to a terminal one.	**Initial population.** It is randomly initialized with a number of (normally valid) completed configurations (or feature models).
Terminal condition. It is determined by the status of the current configuration or feature model (*e.g.,* a complete configuration or feature model).	**Stopping condition.** It is always a predefined computational budget (*e.g.,* number of generations, time) or a specific fitness value achieved.
Actions. They define the set of successors for a given state (*e.g.,* a configuration with more features selected, or a feature model with an additional cross-tree constraint).	**Mutation and crossover operators.** They define modifications or combinations, to the candidate solutions (*e.g.,* selecting/deselecting a feature, making mandatory an optional feature).
State transition function. It applies the possible valid actions to the current state. Actions can be exhaustive applied (during expansion), or randomly (*e.g.,* during simulation). Only the current state is considered at a given time.	**Evolution of the population.** It requires to evaluate (using the fitness function) each individual solution in the population. Mutation and crossover operators are then applied with a given probability to the selected candidate solutions.
Reward function. It is only applied to final solutions, while intermediate states do not need to be evaluated. The utility values may be arbitrary (*e.g.,* positive values for accumulated reward, negative values for cost incurred).	**Fitness function.** It is evaluated for each candidate solution of the whole population. It values is defined in order to be maximized or minimized. Additional constraints of the problem are encoded in the fitness function by penalizing solutions.
Results. A unique optimal solution and statistics about each decision step (*i.e.,* the tree search).	**Results.** A set of optimal solutions (*e.g.,* a pareto front in case of multi-objective optimization).

space to make different analysis, including a uniform random sampling technique [32], but their analyses require building a BDD of the feature model, and this task is intractable for very large-scale models like the Linux one [64], being even a specific challenge for this purpose [70]. MCTS can work directly with the feature model or some other *knowledge compilation technique* [70] (*e.g.,* BDD) as long as it can be modeled using the concepts $(S, s_0, t, A, \theta, \mu)$. One of the applications most widespread of incorporating probability into AAFMs has been to assist the user by means of recommendation systems and interactive configuration processes [46, 51, 55, 56, 61]. For instance, Pereira et al. [55, 56] propose different algorithms [56] for recommender systems in SPL configuration, as well as, the visualization mechanisms [55] to aid the user. Nöhrer [51] investigates the ordering of the decisions in the decision-making process. Rodas-Silva et al. [61] proposes a recommender system to select implementation components of an SPL based on users' rating of such components. However, those works are based on historical data from previous users' configurations, while MCTS does not require domain knowledge, but MCTS can benefit from it to improve, for example, the reward function. Moreover, they mainly focus on the configuration space, while MCTS can also be applied to other analyses, such as in the reverse engineering of feature models problem.

Finally, search-based software engineering techniques [42], especially genetic algorithms and meta-heuristics [30, 63, 75] have achieved great success in the AAFMs area for several problems where both configurations and feature models are the main concepts, such as the feature selection optimization [20, 30, 75], or the reverse engineering of feature models [3–5, 40, 43]. Despite MCTS and genetic algorithms share some similarities when applied for search-based optimization, they have important differences as Table 1 details. A quantitative comparison of both techniques is out of the scope of this paper and is in our planning agenda.

8 CONCLUSIONS AND FUTURE WORK

We have presented a conceptual framework that enables the use of Monte Carlo methods on the AAFMs, and we have mapped different problems that can be analyzed with the MCTS method. Monte Carlo methods incorporate probability into analysis to solve problems that are difficult to handle using deterministic approaches [39] due to the large search space. Especially, MCTS can provide existing analyses with some decision-making capacity, working directly with the feature models, and modeling the problem as a sequence of decision steps with very little domain-specific knowledge. The selective sampling approach of MCTS may provide insights into how other analysis methods could be hybridized and potentially improved. With this contribution, we envision that different problems and analyses can be addressed using Monte Carlo methods, becoming part of the SPL researcher's toolkit when analyzing feature models and their configurations. This new approach can be of big value to advance the state of the art of the AAFMs a step forward.

As part of our ongoing work, we plan to model other problems subject to be analyzed with Monte Carlo methods. Moreover, a quantitative comparison with existing search-based optimization techniques [42] (*e.g.,* genetic algorithms) is also on our agenda. Finally, we also plan to improve our MCTS conceptual framework by developing other variants of the MCTS method [12]. For instance, the independent nature of each simulation in MCTS means that the algorithm is a good target for parallelization [16, 66], so that we can improve its performance; or to explore the use of importance splitting to guide the search checking rare properties [36].

ACKNOWLEDGMENTS

We would like to thank José A. Troyano for have inspired us in the usage of Monte Carlo methods in software product line analyses. This work has been partially funded by the EU FEDER program, the MINECO project OPHELIA (RTI2018-101204-B-C22), the Junta de Andalucia COPERNICA project, and the Spanish Government under Juan de la Cierva—Formación 2019 grant.

REFERENCES

[1] Muhammad Sarmad Ali, Muhammad Ali Babar, and Klaus Schmid. 2009. A Comparative Survey of Economic Models for Software Product Lines. In *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE Computer Society, 275–278. https://doi.org/10.1109/SEAA.2009.89

[2] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer. https://doi.org/10.1007/978-3-642-37521-7

[3] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants. *Empir. Softw. Eng.* 22, 4 (2017), 1763–1794. https://doi.org/10.1007/s10664-016-9462-4

[4] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering legacy applications into software product lines: a systematic mapping. *Empir. Softw. Eng.* 22, 6 (2017), 2972–3016. https://doi.org/10.1007/s10664-017-9499-z

[5] Wesley K. G. Assunção, Silvia R. Vergilio, and Roberto E. Lopez-Herrejon. 2020. Automatic extraction of product line architecture and feature models from UML class diagram variants. *Inf. Softw. Technol.* 117 (2020). https://doi.org/10.1016/j.infsof.2019.106198

[6] Satomi Baba, Yongjoon Joe, Atsushi Iwasaki, and Makoto Yokoo. 2011. Real-Time Solving of Quantified CSPs Based on Monte-Carlo Game Tree Search. In *22nd International Joint Conference on Artificial Intelligence (IJCAI)*. Barcelona, Spain, 655–661. https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-116

[7] Don S. Batory. 2005. Feature Models, Grammars, and Propositional Formulas. In *9th International Conference on Software Product Lines (SPLC)*. 7–20. https://doi.org/10.1007/11554844_3

[8] David Benavides. 2019. Variability Modelling and Analysis During 30 Years. In *From Software Engineering to Formal Methods and Tools, and Back (LNCS, Vol. 11865)*. Springer, 365–373. https://doi.org/10.1007/978-3-030-30985-5_21

[9] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. 2010. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (2010), 615–636. https://doi.org/10.1016/j.is.2010.01.001

[10] Megha Bhushan, José Ángel Galindo Duarte, Piyush Samant, Ashok Kumar, and Arun Negi. 2021. Classifying and resolving software product line redundancies using an ontological first-order logic rule based method. *Expert Systems with Applications* 168 (2021), 114167. https://doi.org/10.1016/j.eswa.2020.114167

[11] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and how to make breaking changes. *ACM Trans. Softw. Eng. Methodol* 1, 1 (2021).

[12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (2012), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810

[13] Murray Cantor. 2011. Calculating and improving ROI in software and system programs. *Commun. ACM* 54, 9 (2011), 121–130. https://doi.org/10.1145/1995376.1995404

[14] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. 2008. Monte-Carlo Tree Search: A New Framework for Game AI. In *4th Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. AAAI Press, 216–217.

[15] Guillaume Chaslot, Mark Winands, H. Herik, Jos Uiterwijk, and Bruno Bouzy. 2008. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation* 04 (2008), 343–357. https://doi.org/10.1142/S1793005708001094

[16] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. 2008. Parallel Monte-Carlo Tree Search. In *6th International Conference on Computers and Games (CG) (LNCS, Vol. 5131)*. Springer, 60–71. https://doi.org/10.1007/978-3-540-87608-3_6

[17] Muhammad Umar Chaudhry and Jee-Hyong Lee. 2018. MOTiFS: Monte Carlo Tree Search Based Feature Selection. *Entropy* 20, 5 (2018). https://doi.org/10.3390/e20050385

[18] Krzysztof Czarnecki, Steven She, and Andrzej Wasowski. 2008. Sample Spaces and Feature Models: There and Back Again. In *12th International Conference on Software Product Lines (SPLC)*. IEEE Computer Society, 22–31. https://doi.org/10.1109/SPLC.2008.49

[19] Krzysztof Czarnecki and Andrzej Wasowski. 2007. Feature Diagrams and Logics: There and Back Again. In *11th International Conference on Software Product Lines (SPLC)*. IEEE Computer Society, 23–34. https://doi.org/10.1109/SPLINE.2007.24

[20] Thiago do Nascimento Ferreira, Jackson A. Prado Lima, Andrei Strickler, Josiel Neumann Kuk, Silvia Regina Vergilio, and Aurora T. R. Pozo. 2017. Hyper-Heuristic Based Product Selection for Software Product Line Testing. *IEEE Comput. Intell. Mag.* 12, 2 (2017), 34–45. https://doi.org/10.1109/MCI.2017.2670461

[21] Alexander Felfernig, Rouven Walter, José Angel Galindo, David Benavides, Seda Polat Erdeniz, Müslüm Atas, and Stefan Reiterer. 2018. Anytime diagnosis for reconfiguration. *J. Intell. Inf. Syst.* 51, 1 (2018), 161–182. https://doi.org/10.1007/s10844-017-0492-1

[22] David Fernández-Amorós, Ruben Heradio, Carlos Cerrada, Enrique Herrera-Viedma, and Manuel J. Cobo. 2017. Towards Taming Variability Models in the Wild. In *16th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT) (Frontiers in Artificial Intelligence and Applications, Vol. 297)*. 454–465. https://doi.org/10.3233/978-1-61499-800-6-454

[23] David Fernández-Amorós, Ruben Heradio, José Antonio Cerrada, and Carlos Cerrada. 2014. A Scalable Approach to Exact Model and Commonality Counting for Extended Feature Models. *IEEE Trans. Software Eng.* 40, 9 (2014), 895–910. https://doi.org/10.1109/TSE.2014.2331073

[24] José A. Galindo and David Benavides. 2020. A Python framework for the automated analysis of feature models: A first step to integrate community efforts. In *24th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. B. ACM, Montreal, Canada, 52–55. https://doi.org/10.1145/3382026.3425773

[25] José Angel Galindo, David Benavides, Pablo Trinidad, Antonio Manuel Gutiérrez-Fernández, and Antonio Ruiz-Cortés. 2019. Automated analysis of feature models: Quo vadis? *Computing* 101, 5 (2019), 387–433. https://doi.org/10.1007/s00607-018-0646-1

[26] José Angel Galindo, Hamilton A. Turner, David Benavides, and Jules White. 2016. Testing variability-intensive systems using automated analysis: an application to Android. *Softw. Qual. J.* 24, 2 (2016), 365–405. https://doi.org/10.1007/s11219-014-9258-y

[27] Dharmalingam Ganesan, Jens Knodel, Ronny Kolb, Uwe Haury, and Gerald Meier. 2007. Comparing Costs and Benefits of Different Test Strategies for a Software Product Line: A Study from Testo AG. In *11th International Conference on Software Product Lines (SPLC)*. 74–83. https://doi.org/10.1109/SPLINE.2007.21

[28] Paul Gazzillo, Ugur Koc, ThanhVu Nguyen, and Shiyi Wei. 2018. Localizing configurations in highly-configurable systems. In *22nd International Systems and Software Product Line Conference (SPLC)*, Vol. 1. ACM, Gothenburg, Sweden, 269–273. https://doi.org/10.1145/3233027.3236404

[29] Sylvain Gelly and David Silver. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence* 175, 11 (2011), 1856–1875. https://doi.org/10.1016/j.artint.2011.03.007

[30] Jianmei Guo, Jia Hui Liang, Kai Shi, Dingyu Yang, Jingsong Zhang, Krzysztof Czarnecki, Vijay Ganesh, and Huiqun Yu. 2019. SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines. *Softw. Syst. Model.* 18, 2 (2019), 1447–1466. https://doi.org/10.1007/s10270-017-0610-0

[31] Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. 2019. Test them all, is it worth it? Assessing configuration sampling on the JHipster Web development stack. *Empir. Softw. Eng.* 24, 2 (2019), 674–717. https://doi.org/10.1007/s10664-018-9635-4

[32] Ruben Heradio, David Fernández-Amorós, José A. Galindo, and David Benavides. 2020. Uniform and scalable SAT-sampling for configurable systems. In *24th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. A. ACM, Montreal, Canada, 17:1–17:11. https://doi.org/10.1145/3382025.3414951

[33] Ruben Heradio, David Fernández-Amorós, Christoph Mayr-Dorn, and Alexander Egyed. 2019. Supporting the statistical analysis of variability models. In *41st International Conference on Software Engineering (ICSE)*. IEEE / ACM, 843–853. https://doi.org/10.1109/ICSE.2019.00091

[34] Ruben Heradio, David Fernández-Amorós, Luis Torre-Cubillo, and Alberto Pérez García-Plaza. 2012. Improving the accuracy of COPLIMO to estimate the payoff of a software product line. *Expert Syst. Appl.* 39, 9 (2012), 7919–7928. https://doi.org/10.1016/j.eswa.2012.01.109

[35] Ruben Heradio, Hector Perez-Morago, David Fernández-Amorós, Roberto Bean, Francisco Javier Cabrerizo, Carlos Cerrada, and Enrique Herrera-Viedma. 2016. Binary Decision Diagram Algorithms to Perform Hard Analysis Operations on Variability Models. In *15th International Conference on New Trends in Software Methodologies, Tools and Techniques (SoMeT) (Frontiers in Artificial Intelligence and Applications, Vol. 286)*. 139–154. https://doi.org/10.3233/978-1-61499-674-3-139

[36] Cyrille Jégourel, Axel Legay, and Sean Sedwards. 2013. Importance Splitting for Statistical Model Checking Rare Properties. In *25th International Conference on Computer Aided Verification (CAV) (LNCS, Vol. 8044)*. Springer, 576–591. https://doi.org/10.1007/978-3-642-39799-8_38

[37] Reza Karimpour and Guenther Ruhe. 2017. Evolutionary robust optimization for software product line scoping: An explorative study. *Comput. Lang. Syst. Struct.* 47 (2017), 189–210. https://doi.org/10.1016/j.cl.2016.07.007

[38] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*. Berlin, Heidelberg, 282–293.

[39] Dirk P. Kroese, Tim Brereton, Thomas Taimre, and Zdravko I. Botev. 2014. Why the Monte Carlo method is so important today. *WIREs Computational Statistics* 6, 6 (2014), 386–392. https://doi.org/10.1002/wics.1314

[40] Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2017. Variability extraction and modeling for product variants. *Softw. Syst. Model.* 16, 4 (2017), 1179–1199. https://doi.org/10.1007/s10270-015-0512-y

[41] Roberto Erick Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. 2018. A systematic mapping study of information visualization for software product line engineering. *J. Softw. Evol. Process.* 30, 2 (2018). https://doi.org/10.1002/smr.1912

[42] Roberto E. Lopez-Herrejon, Lukas Linsbauer, and Alexander Egyed. 2015. A systematic mapping study of search-based software engineering for software product lines. *Inf. Softw. Technol.* 61 (2015), 33–51. https://doi.org/10.1016/j.infsof.2015.01.008

[43] Roberto Erick Lopez-Herrejon, Lukas Linsbauer, José Angel Galindo, José Antonio Parejo, David Benavides, Sergio Segura, and Alexander Egyed. 2015. An assessment of search-based techniques for reverse engineering feature models. *J. Syst. Softw.* 103 (2015), 353–369. https://doi.org/10.1016/j.jss.2014.10.037

[44] Maira Marques, Jocelyn Simmonds, Pedro O. Rossel, and María Cecilia Bastarrica. 2019. Software product line evolution: A systematic literature review. *Inf. Softw. Technol.* 105 (2019), 190–208. https://doi.org/10.1016/j.infsof.2018.08.014

[45] Marzio Marseguerra, Enrico Zio, and Luca Podofillini. 2007. Genetic Algorithms and Monte Carlo Simulation for the Optimization of System Design and Operation. In *Computational Intelligence in Reliability Engineering: Evolutionary Techniques in Reliability Analysis and Optimization*. Studies in Computational Intelligence, Vol. 39. Springer, 101–150. https://doi.org/10.1007/978-3-540-37368-1_4

[46] Jabier Martinez, Tewfik Ziadi, Raúl Mazo, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2014. Feature Relations Graphs: A Visualisation Paradigm for Feature Constraints in Software Product Lines. In *2nd IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE Computer Society, Victoria, BC, Canada, 50–59. https://doi.org/10.1109/VISSOFT.2014.18

[47] R. Munos. 2014. *From Bandits to Monte-Carlo Tree Search: The Optimistic Principle Applied to Optimization and Planning*. https://doi.org/10.1561/2200000038

[48] Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. 2019. Uniform random sampling product configurations of feature models that have numerical features. In *23rd International Systems and Software Product Line Conference (SPLC)*, Vol. A. 289–301.

[49] Vivek Nair, Tim Menzies, Norbert Siegmund, and Sven Apel. 2017. Using bad learners to find good configurations. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, Paderborn, Germany, 257–267. https://doi.org/10.1145/3106237.3106238

[50] Hootan Nakhost and Martin Müller. 2009. Monte-Carlo Exploration for Deterministic Planning. In *21st International Jont Conference on Artifical Intelligence (IJCAI)*. Morgan Kaufmann Publishers Inc., San Francisco, USA, 1766–1771.

[51] Alexander Nöhrer and Alexander Egyed. 2011. Optimizing User Guidance during Decision-Making. In *15th International Conference on Software Product Lines (SPLC)*. IEEE Computer Society, 25–34. https://doi.org/10.1109/SPLC.2011.45

[52] Makoto Nonaka and Liming Zhu. 2007. Impact of Architecture and Quality Investment in Software Product Line Development. In *11th International Conference on Software Product Lines (SPLC)*. IEEE Computer Society, Kyoto, Japan, 63–73. https://doi.org/10.1109/SPLINE.2007.35

[53] Jeho Oh, Don S. Batory, Margaret Myers, and Norbert Siegmund. 2017. Finding near-optimal configurations in product lines by random sampling. In *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. Paderborn, Germany, 61–71. https://doi.org/10.1145/3106237.3106273

[54] Juliana Alves Pereira, Lucas Maciel, Thiago F. Noronha, and Eduardo Figueiredo. 2018. Heuristic and exact algorithms for product configuration in software product lines. In *22nd International Systems and Software Product Line Conference (SPLC)*, Vol. 1. Gothenburg, Sweden, 247. https://doi.org/10.1145/3233027.3236395

[55] Juliana Alves Pereira, Jabier Martinez, Hari Kumar Gurudu, Sebastian Krieter, and Gunter Saake. 2018. Visual guidance for product line configuration using recommendations and non-functional properties. In *33rd Annual ACM Symposium on Applied Computing (SAC)*. 2058–2065. https://doi.org/10.1145/3167132.3167353

[56] Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. 2016. A feature-based personalized recommender system for product-line configuration. In *ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*. ACM, Pau, France, 120–131. https://doi.org/10.1145/2993236.2993249

[57] Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. 2019. Product sampling for product lines: the scalability challenge. In *23rd International Systems and Software Product Line Conference (SPLC)*. ACM, Paris, France, 14:1–14:6. https://doi.org/10.1145/3336294.3336322

[58] Matias Pol'la, Agustina Buccella, and Alejandra Cechich. 2020. Analysis of variability models: a systematic literature review. *Software and Systems Modeling* (2020), 1–35.

[59] Simon Poulding and Robert Feldt. 2015. Heuristic Model Checking Using a Monte-Carlo Tree Search Algorithm. In *Annual Conference on Genetic and Evolutionary Computation (GECCO)* (Madrid, Spain). Association for Computing Machinery, 1359–1366. https://doi.org/10.1145/2739480.2754767

[60] Alessandro Previti, Raghuram Ramanujan, Marco Schaerf, and Bart Selman. 2011. Monte-Carlo Style UCT Search for Boolean Satisfiability. In *Artificial Intelligence Around Man and Beyond (AI*IA)*. Berlin, Heidelberg, 177–188.

[61] Jorge Rodas-Silva, José Angel Galindo, Jorge García-Gutiérrez, and David Benavides. 2019. Selection of Software Product Line Implementation Components Using Recommender Systems: An Application to Wordpress. *IEEE Access* 7 (2019), 69226–69245. https://doi.org/10.1109/ACCESS.2019.2918469

[62] Stuart J. Russell and Peter Norvig. 2020. *Artificial Intelligence - A Modern Approach, Fourth edition.* Pearson Education. http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html

[63] Abdel Salam Sayyad, Joseph Ingram, Tim Menzies, and Hany H. Ammar. 2013. Optimum feature selection in software product lines: Let your model and values guide your search. In *1st International Workshop on Combining Modelling and Search-Based Software Engineering (CMSBSE@ICSE)*. San Francisco, CA, USA, 22–27. https://doi.org/10.1109/CMSBSE.2013.6604432

[64] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2010. The Variability Model of The Linux Kernel. In *4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS) (ICB-Research Report, Vol. 37)*. Universität Duisburg-Essen, Linz, Austria, 45–51. http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf

[65] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354–359. https://doi.org/10.1038/nature24270

[66] E. S. Steinmetz and M. Gini. 2020. More Trees or Larger Trees: Parallelizing Monte Carlo Tree Search. *IEEE Transactions on Games* (2020), 1–1. https://doi.org/10.1109/TG.2020.3048331

[67] Chico Sundermann, Thomas Thüm, and Ina Schaefer. 2020. Evaluating #SAT solvers on industrial feature models. In *14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*. ACM, Magdeburg, Germany, 3:1–3:9. https://doi.org/10.1145/3377024.3377025

[68] Y. Tanabe, K. Yoshizoe, and H. Imai. 2009. A study on security evaluation methodology for image-based biometrics authentication systems. In *3rd IEEE International Conference on Biometrics: Theory, Applications, and Systems*. 1–6. https://doi.org/10.1109/BTAS.2009.5339016

[69] Paul Temple, José Angel Galindo, Mathieu Acher, and Jean-Marc Jézéquel. 2016. Using machine learning to infer constraints for product lines. In *20th International Systems and Software Product Line Conference (SPLC)*. ACM, Beijing, China, 209–218. https://doi.org/10.1145/2934466.2934472

[70] Thomas Thüm. 2020. A BDD for Linux?: the knowledge compilation challenge for variability. In *24th ACM International Systems and Software Product Line Conference (SPLC)*, Vol. A. 16:1–16:6. https://doi.org/10.1145/3382025.3414943

[71] Cristian Vidal-Silva, José A. Galindo, Jesús Giráldez-Cru, and David Benavides. 2021. Automated Completion of Partial Configurations as a Diagnosis Task Using FastDiag to Improve Performance. In *Intelligent Systems in Industrial Applications*. Springer International Publishing, Cham, 107–117.

[72] Leland Wilkinson and Michael Friendly. 2009. The History of the Cluster Heat Map. *The American Statistician* 63, 2 (2009), 179–184. https://doi.org/10.1198/tas.2009.0033

[73] Bang Wong. 2010. Color coding. *Nature Methods* 7, 8 (2010), 573–573. https://doi.org/10.1038/nmeth0810-573

[74] Yinxing Xue, Jinghui Zhong, Tian Huat Tan, Yang Liu, Wentong Cai, Manman Chen, and Jun Sun. 2016. IBED: Combining IBEA and DE for optimal feature selection in software product line engineering. *Appl. Soft Comput.* 49 (2016), 1215–1231. https://doi.org/10.1016/j.asoc.2016.07.040

[75] Hitesh Yadav, A. Charan Kumari, and Rita Chhikara. 2020. Feature selection optimisation of software product line using metaheuristic techniques. *International Journal of Embedded Systems* 13, 1 (2020), 50–64. https://doi.org/10.1504/IJES.2020.108284

A FORMALIZATION OF THE ACTIONS A

A.1 Actions for feature model configurations

a_0: **SelectRandomFeature.** This action adds a random feature $f \in F$ to the configuration c.
CA: f is not already part of the configuration c, that is, $f \notin c$.

a_1: **SelectRootFeature.** It adds the root $r \in F$ of the feature model m to the configuration c.
CA: The configuration is empty: $c = \emptyset$.

a_2: **SelectMandatoryFeature**. It adds a mandatory feature $f \in F$ to the configuration c.
CA: There is a mandatory relation between a feature g already present in the configuration c and feature f. Formally, $f \notin c \wedge \exists g \in c, \exists r \in \mathcal{R} | r = (g, [f], \langle 1..1 \rangle)$.

a_3: **SelectOptionalFeature**. It adds an optional feature $f \in F$ to the configuration c.
CA: There is an optional relation between a feature g already present in the configuration c and feature f. That is, $f \notin c \wedge \exists g \in c, \exists r \in \mathcal{R} | r = (g, [f], \langle 0..1 \rangle)$.

a_4: **SelectFeatureAlternative**. It adds a feature $f_i \in F$, which belongs to an alternative-group, to the configuration c.
CA: There is an alternative relation between a feature g already present in the configuration c and feature f_i, and there is not any other child of g already selected in c. That is, $f_i \notin c \wedge \exists g \in c, \exists r \in \mathcal{R} | r = (g, [f_1, \ldots, f_i, \ldots, f_n], \langle 0..1 \rangle) \wedge f_j \notin c, \forall j \neq i$.

a_5: **SelectFeatureSelection**. It adds a feature $f_i \in F$ of an or-group to the configuration c.
CA: There is an or-group relation between a feature g already present in the configuration c and feature f_i. That is, $f_i \notin c \wedge \exists g \in c, \exists r \in \mathcal{R} | r = (g, [f_1, \ldots, f_i, \ldots, f_n], \langle 0..1 \rangle)$. This action allows selecting more than one child in an or-group.

A.2 Actions for reverse engineering

b_1: **AddRootFeature.** This action adds a feature $f \in F_i$ as the root r of the feature model m.
CA: The feature model m is empty: $F = \emptyset$.

b_2: **AddOptionalFeature.** This action adds a new feature $f \in F_i$ to the feature model m with the optional relation $(g, [f], \langle 0..1 \rangle)$ where $g \in F$ is a feature already present in m.
CA: The feature model m contains at least one feature: $F \neq \emptyset$.

b_3: **AddMandatoryFeature.** This action adds a new feature $f \in F_i$ to the feature model m with the mandatory relation $(g, [f], \langle 1..1 \rangle)$ where $g \in F$ is a feature already present in m.
CA: The feature model m contains at least one feature: $F \neq \emptyset$.

b_4: **AddOrGroupRelation.** This action adds a new or-group relation $(g, [f_1, f_2], \langle 1..2 \rangle)$ with two features $f_1, f_2 \in F_i$ as children of an existing non-group feature $g \in F$ in the model m.
CA: There is a feature g in m that is not the parent of an alternative-group nor or-group relation already created in m. That is, $\exists g \in F, \nexists r \in \mathcal{R} | r = (g, [g_1, \ldots, g_n], \langle 1..1 \rangle) \vee r = (g, [g_1, \ldots, g_n], \langle 1..n \rangle)$ where $n \geq 2$ and g_i are the children of g.

b_5: **AddAlternativeGroupRelation.** It adds a new alternative-group relation $(g, [f_1, f_2], \langle 1..1 \rangle)$ with two features $f_1, f_2 \in F_i$ as children of an existing non-group feature $g \in F$ in m.
CA: Same condition as for action b_4.

b_6: **AddFeatureToOrGroup.** This action adds a new feature $f \in F_i$ to an existing or-group relation r in the feature model m and updates the upper cardinality of r increased by 1.
CA: There is an or-group relation in the model m: $\exists r \in \mathcal{R} | r = (g, [g_1, \ldots, g_n], \langle 1..n \rangle)$, $n \geq 2$ and g_i are the children of g.

b_7: **AddFeatureToAlternativeGroup.** It adds a feature $f \in F_i$ to an existing alternative-group relation r in the feature model m.
CA: There is an alternative-group relation in m: $\exists r \in \mathcal{R} | r = (g, [g_1, \ldots, g_n], \langle 1..1 \rangle)$, $n \geq 2$ and g_i are the children of g.

b_8: **AddRequiresConstraint.** It adds a new "requires" constraint $(f \Rightarrow g)$ involving two existing features $f, g \in F$ in the model m.
CA: Three conditions apply: (1) there are at least two features in m without considering the root feature r — *i.e.,* $|F| \geq 3$; (2) both features $f, g \in F$ cannot be related between them with a parent-child relation — *i.e.,* $\exists f, g \in F | \neg(f \prec g \vee g \prec f)$; and (3) there is not an "excludes" constraint between both features (*i.e.,* $f \Rightarrow \neg g$ or $g \Rightarrow \neg f$), nor a "requires" constraint such that $f \Rightarrow g$ already created in m.

b_9: **AddExcludesConstraint.** It adds a new "excludes" constraint $(f \Rightarrow \neg g)$ involving two existing features $f, g \in F$ in m.
CA: Three conditions apply: (1) there are at least two features in m without considering the root feature r — *i.e.,* $|F| \geq 3$; (2) both features $f, g \in F$ cannot be related between them with a parent-child relation — *i.e.,* $\exists f, g \in F | \neg(f \prec g \vee g \prec f)$; and (3) there is not an "excludes" constraint between both features (*i.e.,* $f \Rightarrow \neg g$ or $g \Rightarrow \neg f$), nor a "requires" constraints such that $f \Rightarrow g$ or $g \Rightarrow f$ already created in m.