A Monte Carlo Tree Search Conceptual Framework for Feature Model Analyses

Jose-Miguel Horcas[a], José A. Galindo[a], Ruben Heradio[b], David Fernandez-Amoros[b], David Benavides[a]

[a] *University of Seville, Seville, Spain*
[b] *National Distance Education University (UNED), Madrid, Spain*

Abstract

Challenging domains of the future such as Smart Cities, Cloud Computing, or Industry 4.0 expose highly variable systems with colossal configuration spaces. The automated analysis of those systems' variability has often relied on SAT solving and constraint programming. However, many of the analyses have to deal with the uncertainty introduced by the fact that undertaking an exhaustive exploration of the whole configuration space is usually intractable. In addition, not all analyses need to deal with the configuration space of the feature models, but with different search spaces where analyses are performed over the structure of the feature models, the constraints, or the implementation artifacts, instead of configurations. This paper proposes a conceptual framework that tackles various of those analyses using Monte Carlo tree search methods, which have proven to succeed in vast search spaces (*e.g.,* game theory, scheduling tasks, security, program synthesis, etc.). Our general framework is formally described, and its flexibility to cope with a diversity of analysis problems is discussed. We provide a Python implementation of the framework that shows the feasibility of our proposal, identifying up to 11 lessons learned, and open challenges about the usage of the Monte Carlo methods in the software product line context. 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 one step forward.

Keywords: automated analysis, configurable systems, feature models, Monte Carlo tree search, software product lines, variability

1. Introduction

The Automated Analysis of Feature Models (AAFM) [1, 2] is one of the most active Software Product Line (SPL) research areas in the last decade [3, 4, 5, 6]. In highly configurable systems, AAFM is a challenging task because it requires coping with a variety of problems which involve interrelated features and colossal search spaces. For example, we find multiple operations, such as counting the number of products [7, 8, 9] and optimizing configurations [10, 11, 12], which are performed on the entire configuration space. Other operations are performed on feature models (*e.g.,* evolution [13] and reverse engineering [14, 15]). Finally, there are other analyses where the main subjects to reason about are the products (*e.g.,* testing [16, 17]) or the constraints of the feature models [18].

Many of these analyses have to deal with the uncertainty introduced by the fact that undertaking an exhaustive exploration of the whole search space is usually intractable. 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 [19, 20]. 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 [14, 21]. Other situations require tackling uncertainty because of the aforementioned combinatorial nature of the search space, which includes different types of selections such as alternatives (xor), or cardinality-based selections [22]. For instance, solving analyses of probability in configuration selections according to performance goals is challenging because the whole configuration space is unknown for large-scale feature models [23, 24].

AAFM has relied on propositional logic or SAT solving [25, 26, 27], constraint programming [1], Binary Decision Diagrams (BDD) [8, 28], statistical analysis [29], genetic algorithms [14, 30], or metaheuristics [31], among others [4]. SAT solvers could face scalability problems for large-scale models [26, 27]. Some statistical analyses require the construction of BDDs (e.g., determining the distribution of the number of features among all valid configurations or testing the uniformity of a random sampler on complex models with thousands of features) [29, 32], which can be intractable [9]. Other approaches like genetic algorithms [14, 30] and metaheuristics [31] require to incorporate specific domain knowledge, and analyzing and inferring results from the final solutions which is not straightforward. In addition, these tech-

niques offer little information about the intermediate steps of the analysis process that allows considering other valuable alternatives before obtaining the final solution.

In this paper, we present a conceptual framework based on Monte Carlo methods [33], which use randomness for deterministic problems that can be represented as sequences of step-wise decisions. Monte Carlo methods can be used with little or no domain knowledge, and have succeeded on difficult problems where an exhaustive exploration of the search space cannot be performed. In particular, we adopt the Monte Carlo Tree Search (MCTS) [34, 35] method. MCTS has been successfully applied to several domains [34], standing out in game theory [36] where it has been shown to scale to large search spaces such as those that typically characterize SPLs. Thus, we conjecture that MCTS may have a great impact in the AAFM. In this paper, we make the following contributions:

- We formally present our MCTS framework, 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 provide a complete implementation of the MCTS framework[1], including three different Monte Carlo methods (Section 4).
- We formally model and solve different problems concerning configurations (Section 5) and features models (Section 6) using our framework, and explain the knowledge we can infer with MCTS.
- We identify up to 11 lessons learned and open challenges of applying Monte Carlo methods in the context of SPLs (Section 5 and 6).

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 (Section 7). In the area of SPLs and AAFM, MCTS can provide an agent with some decision-making capacity with very little domain-specific knowledge, and its selective sampling approach may provide insights into how other analysis techniques, such as search-based algorithms, could be hybridized and potentially improved [10]. We envision that different problems can be addressed using Monte Carlo simulations with this contribution. Accordingly, this new approach can be of considerable value to advance the state of the art of the AAFM one step forward.

This paper extends the work published as a conference paper in SPLC'21 [37]

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

by adding the following contributions: (1) a set of 11 lessons learned and open challenges for the application of Monte Carlo methods in the context of SPLs and AAFM; (2) a complete implementation of the MCTS conceptual framework is provided, in contrast to the prototype presented in the previous work (Section 4); (3) support for extended (attributed) feature models (Section 2) and formalization of a new analysis problem for optimizing configurations (Section 5.4); and (4) all SPL problems previously proposed have been fully formalized, extending the applicability of two of them with up to 8 feature models comparing the different Monte Carlo methods (Section 5.2 and 5.3).

2. Background

This section introduces the feature model formalization and the running example used throughout the paper.

Definition 1 (Feature, Feature model). A *feature* f is a characteristic or end-user-visible behavior of a software system [38]. 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.
- $\mathcal{R} \subseteq F \times F^n \times \mathbb{N}^2$ is the finite set of decompositional relationships between features. Each relationship is denoted as $\big(f, [g_1, g_2, \ldots, g_n], \langle a..b \rangle\big)$ meaning that f is the parent feature of sub-features $g_i, 1 \leq i \leq n$, with a multiplicity $\langle a..b \rangle$ [22]. Whenever f is included in a configuration, at least a and at most b of the g_i's must be included as well. 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 the feature f is the parent of a feature g regardless multiplicity; and we use the notation $f \prec\prec g$ to represent that the feature f is an ancestor of the feature g. Note that a feature f can appear in more than one relation $r \in \mathcal{R}$ as a parent feature.
- Π is a set of cross-tree constraints defined as arbitrary propositional formulas over the set of features F, *i.e.,* $\Pi \subseteq \mathbb{B}(F)$.

■

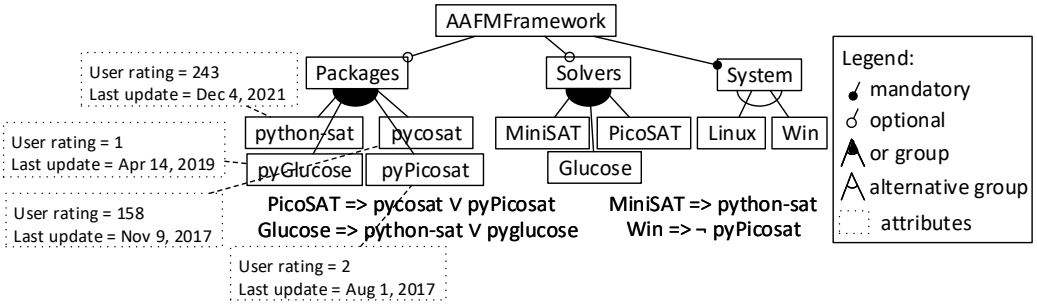


Figure 1: Extended feature model of the SPL for an AAFM framework in Python.

Feature models can be extended to incorporate additional information about the features in terms of *attributes*.

Definition 2 (Attribute, Extended feature model). An *attribute* is a non-functional property associated to a feature (*e.g.,* cost, energy consumption, etc.). Attributes are defined with a type (*e.g.,* numeric) and a domain (*e.g.,* positive integers), and optionally also with a range. An *extended feature model* (also called *attributed feature model*) is a feature model where additional information about the features is provided in terms of attributes. ■

Figure 1 shows an extended feature model representing an SPL for a Python framework to support AAFM [39]. `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 models and implement 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 packages 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 kinds of relations are represented as textual cross-tree constraints, as for example `PicoSAT` $\Rightarrow$ `pycosat` $\vee$ `pyPicosat`. Features representing the Python packages have two attributes associated: the user rating values and the last release update date, both values have been

obtained from the Python Package Index (PyPI) repository.[2] Finally, a specific version of the framework can be deployed in `Linux` or `Win`dows systems, choices represented with the alternative-group (`System`, [`Linux`, `Win`], ⟨1..1⟩).

Definition 3 (Configuration, Partial Configuration, Valid Configuration). A *configuration* c of a feature model m is a subset of its features, *i.e.,* $c \in \mathcal{P}(F)$,[3] meaning that the features in c are selected to be part of the configuration, and the remaining features in F are not included in 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 [25]. 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 the feature model m. ■

An example of a valid configuration of the feature model depicted in Figure 1 is {`AAFMFramework`, `System`, `Linux`, `Solvers`, `MiniSAT`, `Packages`, `python-sat`}. An example of a valid partial configuration is {`AAFMFramework`, `System`, `Solver`}.

3. MCTS Conceptual Framework for feature model analyses

In this section, we first present our conceptual framework (Section 3.1) to model problems that can be solved with Monte Carlo methods, especially with MCTS, and discuss the type of analyses that can be performed with MCTS. Then, we define a mapping between problems in SPLs and the concepts of the MCTS framework (Section 3.2).

3.1. Monte Carlo methods and MCTS

MCTS is a method for finding optimal decisions in a given domain by taking random samples in the search space [35]. MCTS is based on decision and game theory [40], and on Monte Carlo [33] and bandit-based methods [41], where sequential decision problems are modeled as a kind of search problems. Inspired by *Markov Decision Processes* [40] and following the concepts used in game theory [40] to formally define a game as a kind of search problem, we provide the following definition that allows representing SPL problems as a sequence of decisions to be solved by Monte Carlo methods:

[2] `https://pypi.org/`

[3] $\mathcal{P}(F)$ is the powerset of F (i.e., the set of all subsets of F).

Definition 4 (Monte Carlo decision process). A *Monte Carlo decision process* is a 6-tuple with the following elements $(\boldsymbol{S, s_0, t, A, \theta, \mu})$:

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

■

Together, the initial state s_0, the set of actions A, and the transition function θ implicitly define the *search space* (or state space) of the problem. Formally, we define the search space as follows:

Definition 5 (Search space). Given a problem $\boldsymbol{p = (S, s_0, t, A, \theta, \mu)}$, the *search space* is the set of all *nodes* reachable from the initial state s_0 by any sequence of actions in A. A *node* in the search space consists of a pair $(state, list[actions]) \in S \times A^*$. That is, each node represents a state of the domain together with a list of actions that trace the node back to the initial node (s_0, ϵ), and the links to child nodes represent actions that lead to successor nodes. ■

Please observe that since θ is a function, and as such, it is deterministic, the nodes in the search space as defined form a tree rooted in (s_0, ϵ) in which there is a transition function between nodes $\theta' : S \times A^* \times A \to S \times A^*$ such that $\theta'(s, \omega, \alpha) = (\theta(s), \omega\alpha)$. The concept of search space is illustrated in Figure 2 (a) where the nodes are depicted as states for clarity of exposition. Note that there are no replicated nodes, but there may be duplicated states

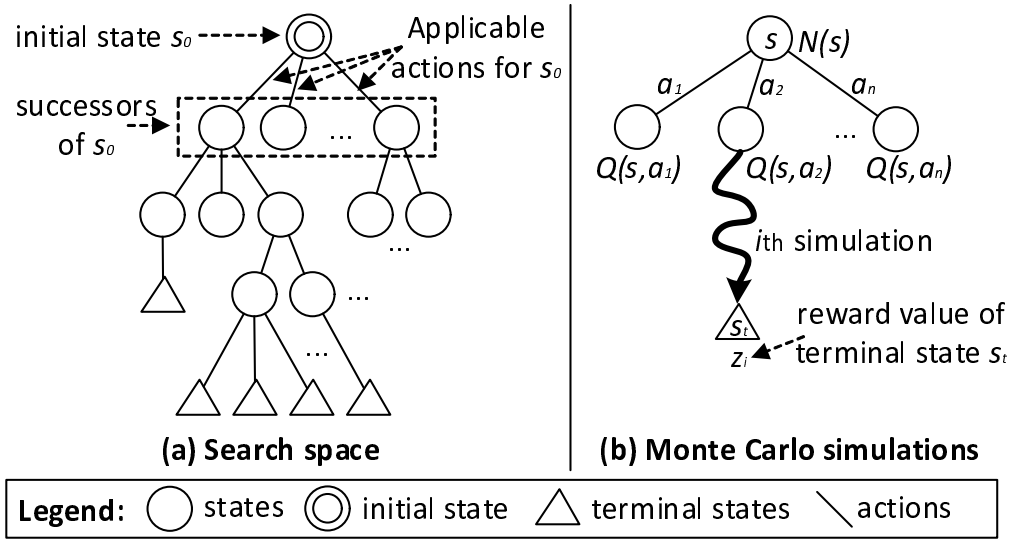


Figure 2: Search space for a generic problem modeled as a sequence of $(state, action)$ decisions (a), and the Monte Carlo simulations approach (b).

in different nodes whenever the same state can be attained through different action sequences. Thus, overall decisions are modeled as sequences of $(state, action)$ pairs.

Basic Monte Carlo approach. 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 sequence of actions applied to the given state until a terminal state s_t is found. The terminal state s_t is then evaluated using the reward function μ obtaining a reward value z_i associated with that simulation, as shown in Figure 2 (b). Running a number of simulations N from the given state s, Monte Carlo approximates the expected reward each action can achieve 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*) [42] of that action, and it is defined as the mean of all rewards obtained from the simulations performed from state s choosing action a:

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

Algorithm 1 Basic Monte Carlo method.

```
1: function MonteCarlo(s)
2:     Q, N ← 0.0, 0          ▷ Initialize dictionaries of (action, float) and (action, int), respectively.
3:     while not stopping criteria do
4:         a ← random.choice(A)                          ▷ Select a random valid action.
5:         z ← SIMULATION(s, a)                          ▷ Run a simulation.
6:         N[s, a] ← N[s, a] + 1                         ▷ Update visits count.
7:         Q[s, a] ← Q[s, a] + (z − Q[s, a]) / N[s, a]   ▷ Update the Q value.
8:     end while
       ▷ Choose the best-performing action according to some criteria (e.g., action with maximum Q).
9:     return BEST_ACTION(s, Q, N)
10: end function
```

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 or 0 otherwise. Algorithm 1 summarizes in pseudocode the basic (also called "flat") Monte Carlo method.

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

Monte Carlo Tree Search. The MCTS method [35] extends the Monte Carlo principle by using the expected reward (Q-values) obtained from simulations to build an incremental and asymmetric *tree search* which is then used for subsequent decisions. We use the term *tree search* for a tree, generated by MCTS, that is superimposed on the full search space, and examines enough nodes to allow the MCTS method to determine what decision to make. As shown in Figure 3, the basic MCTS algorithm (summarized in pseudocode in Algorithm 2) involves iteratively building and using a tree search until some predefined stopping criteria (*e.g.,* time, memory, number of iterations) is reached. Four steps are applied per search iteration [35]:

1. **Selection.** Starting with the initial state s_0, the tree search is traversed by recursively applying a selection function from the root node until a *frontier* node s_l is reached in the tree search. A *frontier* node is a leaf node in the tree search that will be expanded in the following step. Several strategies for selecting the nodes in the tree can be found in [34]. The most popular one is the *Upper Confidence bound for Trees* (UCT) [43] that, using the Q-values, attempts to balance exploitation (*i.e.,* search on areas that appear to be promising) and exploration

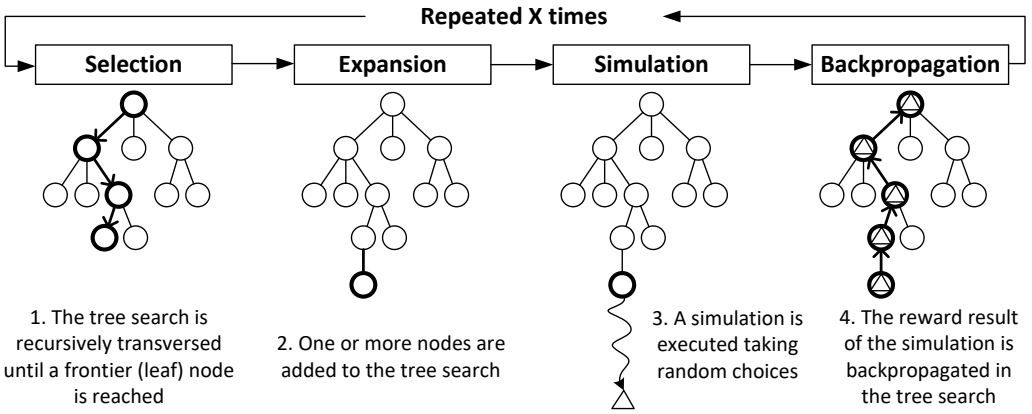


Figure 3: The MCTS approach (adapted from [35]).

(*i.e.,* search on areas that have not been well explored yet).

2. **Expansion.** From the selected frontier node s_l in the tree search, one or more child nodes are added to expand the tree according to the actions A. Note that at this point, the frontier node is not a leaf node anymore in the tree search.

3. **Simulation.** A simulation is run from the new node(s) by doing random actions until a terminal state is reached. The terminal state is evaluated, producing an outcome z (*i.e.,* the reward value).

4. **Backpropagation.** The statistics of each node in the tree that was traversed during the selection step are updated. 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 [44], 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.

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:

Algorithm 2 General Monte Carlo Tree Search method.

```
1: tree ← ∅                                  ▷ Initialize tree search: dictionary of (node, list[node]).
2: Q, N ← 0.0, 0          ▷ Initialize dictionaries of (action, float) and (action, int), respectively.
3: function MCTS(s_0)
4:     while not stopping criteria do
5:         s_l ← SELECTION(s_0)                          ▷ Apply the child selection tree policy.
6:         EXPANSION(s_l)                  ▷ Add one or more nodes to the tree search.
7:         z ← SIMULATION(s_l)                           ▷ Apply the default policy.
8:         BACKPROPAGATION(z, s_l)          ▷ Backup the reward, updating Q and N.
9:     end while
10:    return BEST_ACTION(s_0)
11: end function
    ▷ SELECTION, EXPANSION, SIMULATION, and BACKPROPAGATION are illustrated in
    Figure 3. The implementation details are available online.
```

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 [45]. Here, MCTS can examine how each feature selection contributes to the complete product by modeling the feature selection optimization problem [10, 31, 46, 47] as a sequence of decision steps. We illustrate this type of analysis in Section 5.

Analyzing unintuitive results. Some problems admit multiple solutions. For instance, in feature models' *reverse engineering* [14, 15], an input set of configurations may correspond to many potential output feature models. Without taking into account domain knowledge, the features appearing in all products (*i.e.,* the *core features* [1]) may be considered interchangeable in the resulting feature model. For that reason, MCTS can help a domain engineer explore the alternatives to select the best model. We show this type of analysis in Section 6.

Analyses of uncertainty. Some problems 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 [10, 12]. The best configurations may be spread across the configuration space, leading to a search-based software engineering technique to deal with many local optima [30]. In this case, MCTS is useful to incorporate probability into the analysis. MCTS helps understand

the probability distribution of the best configurations and analyze how such distribution impacts the search-based optimization, so that we could 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 [29] or where simulation rather than optimization is the most effective decision support tool [34]. As stated by Schmid [48], 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.

3.2. Mapping SPL problems to the MCTS conceptual framework

To apply MCTS to SPL problems, we need to formulate the problem as a sequence of $(state, action)$ decisions using the conceptual framework $\boldsymbol{(S, s_0, t, A, \theta, \mu)}$ introduced in the previous section.

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 configuration sample, a performance model of a configuration sample, 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 adding it to the configuration). States representing feature models will define actions to modify the feature model (*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 $\boldsymbol{(S, s_0, t, A, \theta, \mu)}$ 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

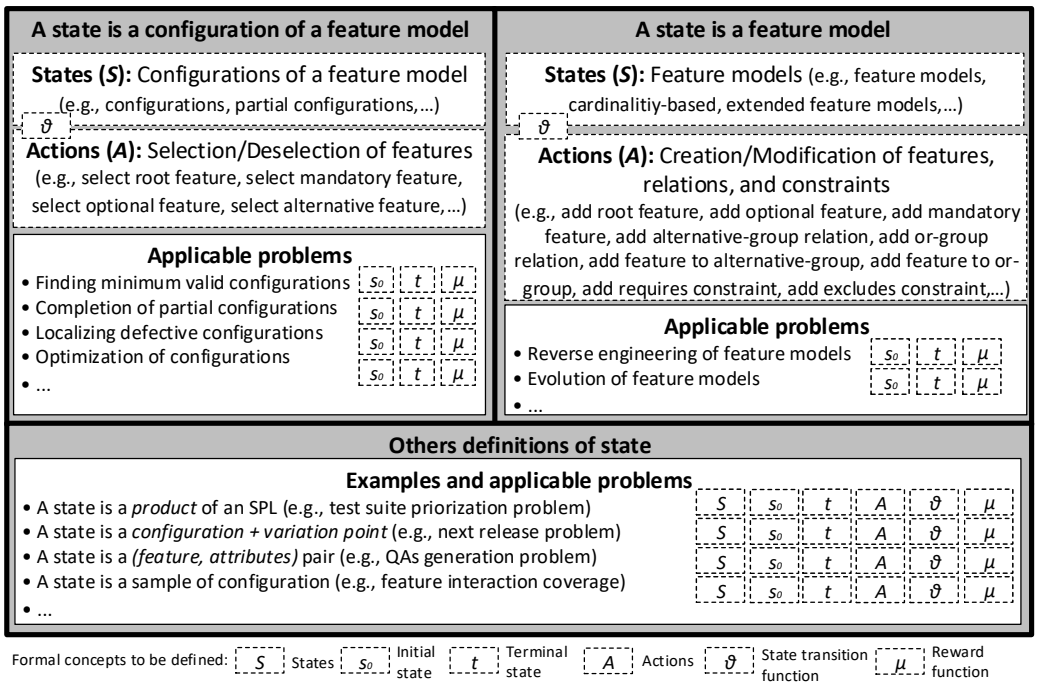


Figure 4: Mapping of SPL problems to the MCTS framework $(\boldsymbol{S, s_0, t, A, \theta, \mu})$.

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 [17].

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

4. Implementation of the MCTS framework

To demonstrate the applicability of our proposal, we provide an implementation of our MCTS conceptual framework, as well as an implementation of the analysis problems described in the following sections. This section

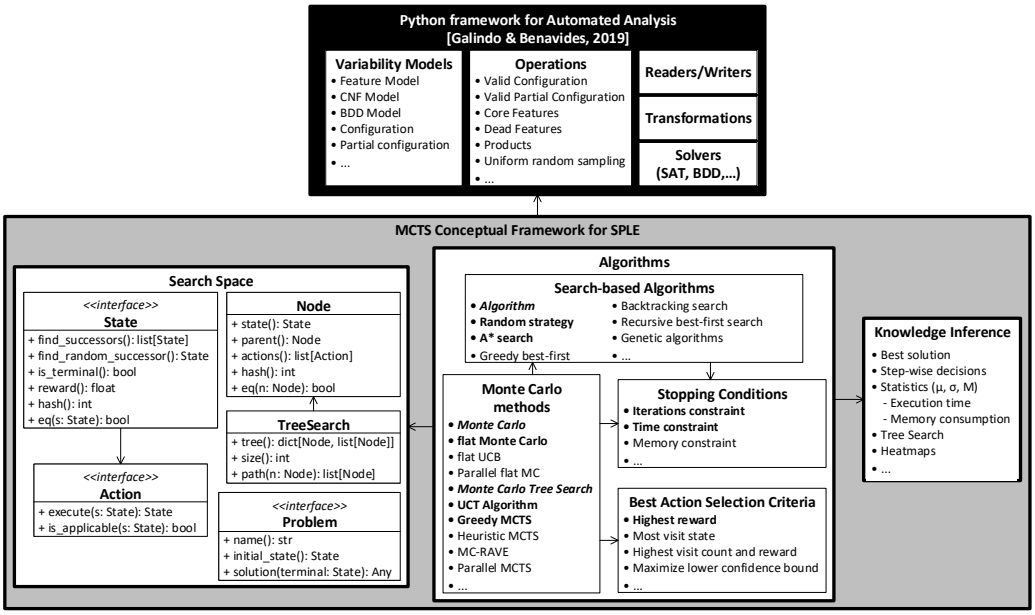


Figure 5: MCTS conceptual framework architecture.

presents the architecture of the framework, including the Monte Carlo methods available.

The framework is available online[4], and has been developed on top of the Python framework for AAFM proposed in [39]. Note that the AAFM tool has been tested in front of well-known testing techniques for AAFM (*e.g.,* metamorphic testing) [49]. Figure 5 overviews the core architecture of our implementation. It consists of three main modules: (1) the search space, including the interfaces to model SPL problems; (2) the search-based algorithm module, which includes the Monte Carlo methods; and (3) the knowledge information inference with the output information from the search.

4.1. Search space and interfaces for $(S, s_0, t, A, \theta, \mu)$ *modeling concepts*

We provide three main interfaces (`State`, `Action`, and `Problem`) to be implemented for modeling SPL problems as sequences of (state, actions) pairs. The `State` interface specifies the methods necessary to build and explore the search space, so that from a given initial state s_0, we can reach all

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

states in S. The `State` interface has to be implemented only once defining the state transition function θ (`successors()` and `random_successor()`), the `is_terminal()` condition t, and the `reward()` function μ. Additional implementation methods are required to guarantee the correct functioning of the algorithms: for example, states must be comparable (`eq()` method) and hashable (`hash()` method). The `Action` interface is defined for each applicable action. The `Problem` interface specifies how a problem is set up at the start by providing an initial state, and how a solution is decoded from a terminal state to represent the solution in a human-readable form. Finally, we provide an implementation of `TreeSearch` that the MCTS method will use during the search. The tree search is based on `Node`s representing the states and the list of actions that trace the path to the initial state.

4.2. Search-based algorithms and Monte Carlo methods

Although our framework focuses on Monte Carlo methods, it can support any search-based algorithm that is built using the previous interfaces, such as a classical A-star search (A*) [40] or genetic algorithms [30]. In the `Algorithms` module of Figure 5, the algorithms and methods highlighted in bold are fully implemented and ready to be used in our framework; the algorithms in italics represent interfaces that can be further specialized; and other algorithms and methods in regular font are possible extensions worthy of being incorporated into our framework in the future. The `Algorithm`, `Monte Carlo` and `Monte Carlo Tree Search` interfaces provide a base abstract implementations of any search-based algorithm, Monte Carlo, and MCTS method, respectively, which can be specialized with different algorithms and variants of Monte Carlo methods [42]. The following generic search-based algorithms are currently available:

- **Random strategy.** It chooses a random action from the current state without running any simulation. This is not a Monte Carlo method itself, but it is often used in game theory to simulate a random player and it is widely used as a baseline to compare Monte Carlo methods [34].

- **A* search.** It is an implementation of the most widely known form of best-first search [40]. It evaluates nodes by combining the cost to reach the current node and a heuristic of the cost to get from the current node to a terminal node. This method is available only for testing the correctness of the implementation of the states and the actions because

this method performs an exhaustive search, which makes it infeasible in practice for medium and large search spaces.

An implementation of several Monte Carlo methods, including MCTS, is available to solve any problem that implements the aforementioned interfaces (state, actions, problem). The Monte Carlo and MCTS methods available in our framework are the following:

- **UCT algorithm.** An implementation of MCTS (Algorithm 2) that builds a search tree and uses the UCT selection strategy [43], which favors actions with a higher Q-value but allows at the same time to explore those actions that have not yet been sufficiently explored. The exploration *vs.* exploitation balance is controlled with an exploration constant ($0 \leq EC \leq 1$) parameter with a default value of 0.5.
- **Greedy MCTS.** A best-first strategy that favors exploitation against exploration. This method is equivalent to the UCT Algorithm with the exploration constant $EC = 0$, and always chooses the action with higher Q-value.
- **Flat Monte Carlo.** An implementation of the basic Monte Carlo method (Algorithm 1) with random action selection and no tree growth. It only considers the simulations from the current state without using the information from previous simulations.

The framework is open for extension, and thus, further variants of Monte Carlo and MCTS methods can be added to the framework such as parallel versions of both methods [50, 51] to improve the efficiency, Heuristic MCTS [42], MC-RAVE [42], and further specialization that may include the use of *minimal cut sets* [52], *rare event simulations* [53], or *importance splitting* [54], among others.

Configuration of the algorithms and Monte Carlo methods. Each search-based algorithm, including the Monte Carlo methods, can be configured with a stopping condition that specifies a computational budget (e.g., time, memory, or a number of steps) to find a solution, after which the algorithm will return the solution (if found), as shown in Algorithm 3. If no stopping condition is provided, the algorithm will run until a solution is found. In addition, each Monte Carlo method can be configured with a second different stopping condition that allows specifying a stopping criterion

for making a decision during each algorithm step (*e.g.,* number of simulations or iterations, time to run a simulation or iteration), so that when the condition is reached, the Monte Carlo method will return the best decision found at that particular step, as shown in Algorithms 1 and 2. Currently, two different stopping conditions are implemented: (1) an `Iterations constraint` which allows specifying a maximum number of steps for the algorithm to find a solution or a maximum number of simulations or iterations for Monte Carlo methods to make a decision; and (2) a `Time constraint` that allows specifying a maximum execution time in seconds for the whole search or for making a decision in Monte Carlo methods.

Moreover, each Monte Carlo method can also be configured with a selection criterion for the best action decision. For example, 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 that maximizes a lower confidence bound [44]. Currently, the selection criterium that returns the child with the highest reward is available.

Finally, additional configuration parameters can be provided to Monte Carlo methods. Concretely, due to Monte Carlo methods rely on randomness, we provide a seed parameter to initialize the random generator and enable experimental reproducibility. Also, a number of runs can be specified (default is 1) to repeat an experiment several times (*e.g.,* 30) and obtain the median, means and standard deviation of different statistics, as explained below.

Algorithm 3 Generic algorithm to solve a problem with MCTS.

```
1: function FindSolution(s_0)
2:     state ← s_0
3:     while within computational budget and not is_terminal(state) do
4:         state ← MCTS(state)                          ▷ Run MCTS (Algorithm 2).
5:     end while
6:     return state
7: end function
```

4.3. Usage of the Monte Carlo methods and knowledge inference

Two main usages of the Monte Carlo methods are available. First, Monte Carlo methods can be used as a search-based algorithm so that from a given initial state, the algorithm will look for the best solution(s) — *i.e.,* terminal state(s). Algorithm 3 illustrates how to use the MCTS method as a search-based algorithm to find a solution to a generic problem. Given an initial state (s_0), the algorithm will run until some predefined computational

budget is reached or until a terminal state is reached (line 3). In each step, the algorithm calls the MCTS method in charge of choosing the best action (line 4). In addition to the best solutions found by the algorithm, the MCTS framework provides further knowledge information gathered during the search. The partially optimal decisions made step by step are available, so that we can observe, for example, which feature has been selected in each step during the configuration process for configuration-based analyses, or which relation has been added to the feature model in each step for evolution-based problems or reverse engineering problems.

Second, Monte Carlo methods can also be used to analyze a particular state and its possible alternatives. Given a state, the method will analyze the possible alternatives that can be reached from this state and will return information about the best choices, so that the practitioner can make better decisions. In this respect, the MCTS framework also reports the information stored in the tree search about the Q-values and visit accounts of each (state, action) pair gathered by the MCTS method. To illustrate the knowledge stored in the tree search, we use a data visualization technique called *heatmap* [55, 56, 57], which encodes quantitative values as colors (like in weather maps), so that it compacts large amounts of information (our Q-values) to bring out coherent patterns in the data (*e.g.,* optimal feature selection over the feature model). In the following sections, we use heatmaps as one of the tools to represent the contribution of each decision to the global solution achieved by MCTS.

Finally, the MCTS framework also reports statistical information about the execution time and memory consumption of the algorithms, such as the median, mean, and standard deviation for both the global solution and each step-wise decision, that allows a deep study of the different Monte Carlo methods and analysis problems. The experiments shown in the following sections 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.

5. Configuration based analysis

One of the most important and widely studied types of problems in AAFM deals with feature model configurations. Examples of these problems are the optimization of configurations according to non-functional properties [10, 31, 45], the completion of partial configurations [58], the localization

of defective configurations in SPL testing [59, 60, 61, 62], or the diagnosis of configurations [58, 63], among many other problems. To analyze this kind of problems with MCTS, 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 state set S encompasses all possible combinations of features of the feature model. Depending on the definition of the set of actions A, S may consider either valid or invalid configurations or both, but also either partial or complete configurations or both.

The action set A. There are multiple possibilities to define the set of valid actions that can be performed over a given configuration. For instance, actions for configurations can include (de)selecting a unique feature or (de)selecting a set of features. In this paper, we opt to follow an incremental approach in which a configuration is built from scratch (or from a given partial configuration) by selecting features. Formally, given a state $s \in S$ representing a (partial) configuration c, the application of an action $a \in A$ with argument $f \in F$ will lead to a new state $s' \in S$ representing a new (partial) configuration $c' = \{f\} \cup c$. An action is valid if it can be applied to a state under a certain *Condition of Applicability* (CA). The condition of applicability is defined for each action and its result depends on the current state. An action may receive any kind of parameters in order to be executed. The most basic action is selecting a random feature from F:

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, $f \notin c$.

This action is independent of the relations defined in the feature model (Definition 1), and thus it can be used for any other definition of the 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. A more convenient definition for the actions is considering the relations of the feature model, reducing the resulting search space, but losing the independency from the feature model stated before. Following our feature model Definition 1, we specify the following set of actions $A = \{a_1, a_2, a_3, a_4, a_5\}$:

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: **SelectFeatureOr**. 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.

Note that the configuration is always built incrementally 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 deselecting a feature and avoid cycles in the 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 cross-tree constraints in the feature model (*e.g.,* checking it with a SAT solver). Note also that we may take into account *atomic sets* [64][5] of the feature model to add several features in each step, reducing the search space. However, this will limit the analysis of the decisions made when features are added step by step as illustrated in the following analysis problems. It is also worth mentioning that actions $a_1, ..., a_5$ fully characterize feature diagrams

[5]An *atomic set* is a group of features that can be treated as a unit when performing certain analyses, and therefore, those features appear together in configurations.

as defined in FODA [65] (*i.e.,* optional and mandatory features, and xor and or-groups), but not all the possible feature models as they are defined in Definition 1 because Definition 1 supports more generic feature models that may include relations supporting mutex groups (*i.e.,* $\langle 0..n \rangle$) or arbitrary cardinality groups (*i.e.,* $\langle a..b \rangle$) [22]. To support these relations, one may define additional actions to incorporate the required features in the configuration to satisfy these relations. Moreover, in the literature [22, 66] it has been demonstrated that these relations can be refactored using the FODA concepts and arbitrary cross-tree constraints (*i.e.,* the feature model can be transformed to an equivalent feature model with the same semantics). For simplicity illustrating the problems in this paper, we assume feature models only contain optional and mandatory features, and xor and or-groups relations in the feature diagram.

The state transition function $\boldsymbol{\theta : S \times A \rightarrow S}$ 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 to 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 from 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 four concrete problems where a state represents a configuration by providing complete definitions of the concepts $\boldsymbol{(S, s_0, t, A, \theta, \mu)}$.

5.1. Localizing defective configurations

A common problem in software testing and maintenance is identifying the configurations that lead to a given defect or some other undesired program behavior [59, 60, 61]. Continuing with our running example, let us suppose that we want to identify those valid configurations in our feature model (Figure 1) that present anomalies when they are deployed. Anomalies in the Python framework for AAFM 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 [30], 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. [62] 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, to provide a robust AAFM Framework, apart from identifying the defective configurations, we need to identify those features that cause the defects. Using a step-wise decision approach, we can infer which features are causing the configuration to fail.

Modeling the problem. We model this problem in the MCTS conceptual framework $\boldsymbol{(S, s_0, t, A, \theta, \mu)}$ as follows:

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

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

  The `is_valid(s)` operation is performed with a SAT solver [1].
- $\boldsymbol{A}$**:** The set of valid actions $\boldsymbol{A} = \{a_1, \ldots, a_5\}$ as previously defined.
- $\boldsymbol{\theta}$**:** The state transition function is given by the definitions of the actions and their conditions of applicability: $\boldsymbol{\theta} = S \times A \to S$.
- $\boldsymbol{\mu}$**:** The reward function for a terminal configuration s:

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

  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 to 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. Note that the reward function for Monte Carlo methods is usually defined in game theory with fixed values (*e.g.,* +1: win, -1: loss, 0: draw) with no intermediate values, and we are defining here a continuous function based on the number of errors found.

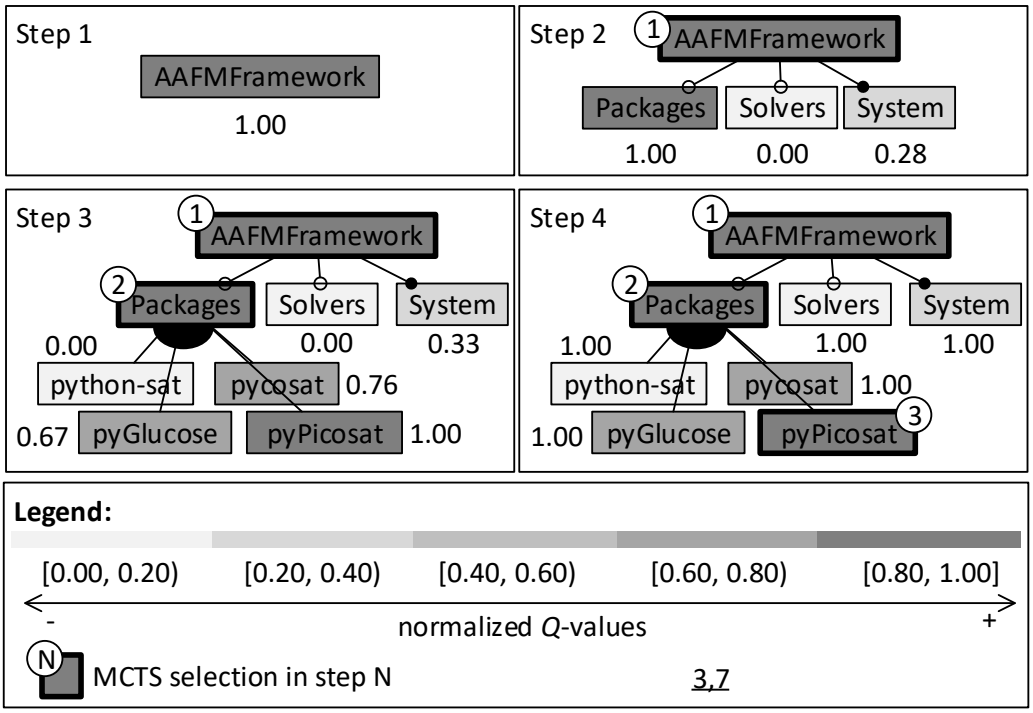


Figure 6: Heatmap representing step-wise decisions for defective configurations.

Solving the problem and analyzing the results. We can solve the problem of finding defective configurations by consecutively applying MCTS to the initial empty configuration (Algorithm 4). We modify the stopping condition of the generic Algorithm 3 so that the search will run until a configuration with errors is found or no valid action can be applied to the current configuration (line 2 in Algorithm 4). 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 3.1. The most promising feature is the next one in the feature model hierarchical tree (according to the set of actions A) that moves the configuration closer to a complete valid configuration with the highest number of errors.

The execution of the algorithm is illustrated step by step in Figure 6. We use a grey-scale heatmap for each algorithm step to represent each feature's contribution. Darker colors mean a higher probability of finding a defective configuration. if that feature is selected. Given the empty configuration as the initial state, in the first step, the only action available is to add the root

Algorithm 4 Finding defective configurations with MCTS.

```
1: function FindDefectiveConfiguration(state)
2:     while reward(state) ≤ 0 and actions(state) ≠ ∅ do
3:         state ← MCTS(state)                          ▷ Run MCTS (Algorithm 2).
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 function
```

feature `AAFMFramework` (Step 1 in Figure 6). In Step 2, three possible features can be added to the configuration according to the action set 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 action set 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 3.1 — *i.e.,* the Q-values. Figure 6 shows normalized Q-values in the 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 mandatory features), despite the problematic feature that has already been discovered. From step 3 where `pyPicosat` is selected, the algorithm will find a defective configuration regardless of the selections of the following steps, because all the complete configurations will contain the `pyPicosat` feature. The algorithm finishes when it finds a valid defective configuration, as, for example, the final state found: {`AAFMFramework`, `Solvers`, `System`, `Linux`, `Packages`, `pyPicosat`, `PicoSAT`}. As Figure 7 shows, when deploying that configuration in Python some errors raise: the package `pyPicosat` cannot be correctly installed in Linux. Thanks to the heatmap shown step by step, we are able to identify that the feature `pyPicosat` is one of the problematic features that provokes the configuration to be defective. Following this procedure, we can find that deploying any configuration with `pyPicosat` in the Linux system leads to failures, so we opt to update the original constraint `Win` $\Rightarrow \neg$ `pyPicosat` to `Win` $\vee$ `Linux` $\Rightarrow \neg$ `pyPicosat`, converting `pyPicosat` into a dead feature [1].

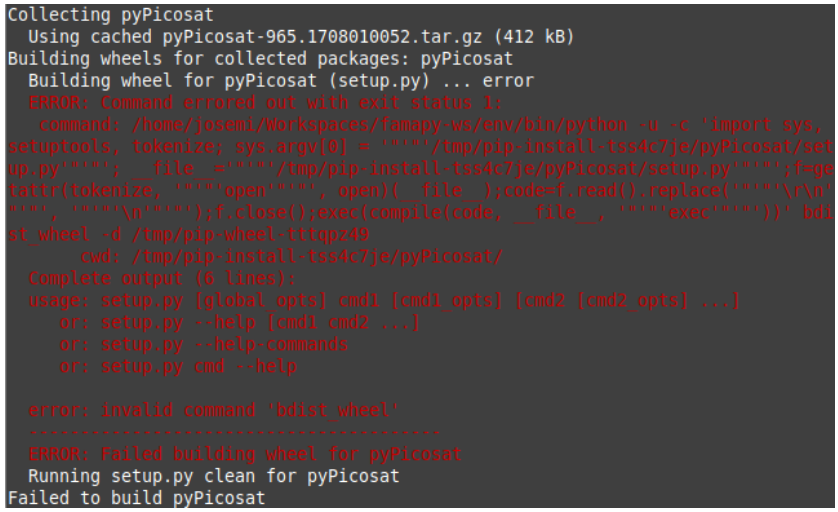


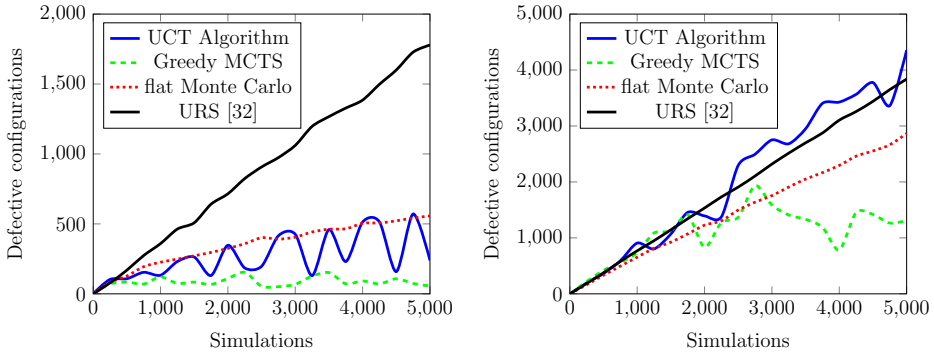
Figure 7: Deploying a defective configuration.

Lessons learned and open challenges. 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 found, so that we may find additional defective configurations by following the sequence of feature selections done by MCTS to find the current configuration. However, two limitations arose at this point on the applicability of MCTS. First, the problem of finding defective configurations requires specific domain knowledge represented by the information about what configurations present errors when they are deployed. This limits the applicability of this analysis to those feature models that contain such domain knowledge. Second, MCTS can function as a search-based algorithm, but it is more appropriate to find just one solution, not all of them. To find all defective configurations, we need to track the solutions found and use that information as part of the reward function so that it penalizes (return -1) those simulations that reach the defective configurations already found.

To illustrate these concerns and 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 [60], and the complete version of the Python framework for AAFM [39]. On the one hand, we choose the JHipster SPL because its configuration space (26,256 configurations) has been fully evaluated in [60] (having 9,376 defective configurations,

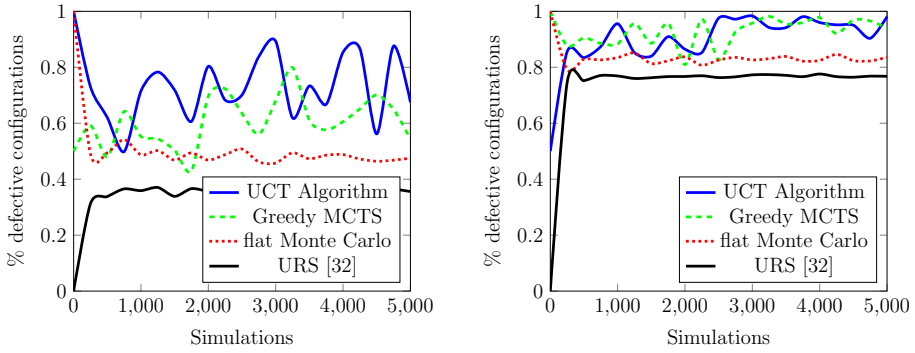
i.e., 35.70%) and can be used to evaluate the results obtained with Monte Carlo methods [67]. On the other hand, the complete `AAFMFramework` product line presented in Section 2 has 53 features, 26 relations, and 10 cross-tree constraints, leading to a total of $3.1264 \cdot 10^9$ configurations. It serves as a large-scale configuration space where we have already identified the specific features that cause errors when the configurations are deployed by manually testing each feature individually. We compare three Monte Carlo methods for different numbers of simulations *w.r.t.* uniform random sampling (URS) [32]. URS is the simplest way to solve search-related problems on configurable systems [24, 68, 69]. URS-based search consists of generating a random sample of configurations, testing (or benchmarking) them, and selecting the ones that fail (or the one that achieves the best performance). Accordingly, URS is the baseline of any more elaborated search algorithms, whose existence only makes sense if they can beat pure random. In contrast to URS, the random strategy presented in Section 4.2 is not able to find any defective configuration in most cases because the random strategy selects the actions randomly, but the resulting configurations are not uniformly selected from the full search space as URS does. Thus, in this particular problem, URS is a more reasonable base line to compare Monte Carlo methods. The results are summarized in Figures 8, 9, and 10. We present the number of configurations found with defects (a), the number of configurations (terminal states) evaluated (b), the efficiency as the percentage of defective configurations found *w.r.t.* the configurations evaluated (c), and the execution time (d). We identified the following Lessons Learned (LS) and Open Challenges (OC).

LS1 *MCTS is a selective sampling method which balances exploitation and exploration, in contrast to uniform random sampling.* A first observation is a higher fluctuation in the MCTS. Especially in the UCT Algorithm where we set the exploration constant to 0.5, leading to a balance between exploitation and exploration [42, 43]. In JHipster (Figure 8a), defective configurations are localized in regions, making the MCTS method focuses on that area until the region is sufficiently explored. On the contrary, in the AAFM Python framework (Figure 8b), defective configurations are scattered through the configuration space, and MCTS will find more defective configurations with the same number of simulations. The Greedy MCTS method favors exploitation in contrast to exploration (the exploration constant is set to 0), and therefore the



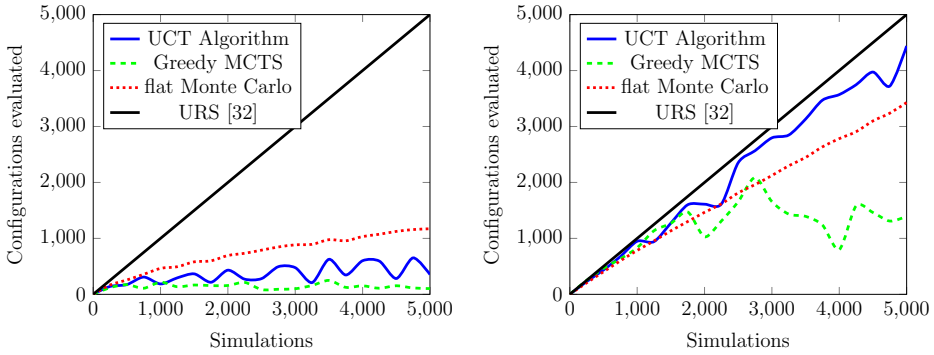
(a) JHipster. (b) AAFM Framework.

Figure 8: Finding defective configurations.



(a) JHipster. (b) AAFM Famework.

Figure 9: Efficiency of the search for defective configurations.



(a) JHipster. (b) AAFM Framework.

Figure 10: Evaluation of terminal states.

greedy version of MCTS finds the same defective configurations more than once during simulations than the other methods. This occurs because greedy MCTS chooses the action with the highest Q-value and the subsequent simulations will choose the same action with highest Q-value. Regarding the flat version of Monte Carlo method, it behaves more similar to random sampling because it does not use the information gathered in previous simulations for the subsequent decisions, but it still gets benefits from the current simulations run. **OC1:** *The challenge is identifying the most appropriate Monte Carlo method for a specific SPL problem. Monte Carlo methods are not a silver bullet for analyzing SPL problems, but these results show that MCTS can also be used as a search-based optimization technique in SPL, possibly as a complement of existing approaches (*e.g., *genetic algorithms [30]).*

LS2 *The efficiency of Monte Carlo methods depends on the distribution of the configuration space and the structure of the feature model.* A second observation is the efficiency of Monte Carlo methods (Figure 9). Monte Carlo methods are superior on average to URS when comparing the amount of defective configurations found *w.r.t.* the configurations evaluated. In JHipster, with 5000 simulations, the UCT Algorithm evaluates 5000 configurations, of which 36% are defective configurations, flat Monte Carlo finds 48%, and Greedy MCTS finds 54%, in contrast to URS, which finds 36% of defective configurations. In the AAFM Python framework feature model, with 5000 simulations, the UCT Algorithm finds 98% of defective configurations, flat Monte Carlo finds 84%, and Greedy MCTS finds 94%, in contrast to URS that finds 77% of defective configurations. This result is not surprising since MCTS is a selective sampling approach. However, the number of solutions found (Figure 9), as well as the number of solutions evaluated, that is, the number of terminal states evaluated with the reward function (Figure 10) by MCTS will depend on the distribution of the configuration space [29]. This implies that the same configurations may be found more than once, requiring MCTS more simulations and evaluations to find distinct defective configurations because several simulations may lead to the same terminal states. Furthermore, the performance of flat Monte Carlo is better in one case study (AAFM Framework) and worst in the other (JHipster) because of the structure of the feature model, and thus, the structure of the search space. In the JHipster, despite

having only 26,256 configurations, the valid configurations are larger in the number of features requiring more steps (decisions) of flat Monte Carlo to find a valid configuration. In contrast, in the AAFM Framework, despite having more than 10^9 configurations, those configurations are smaller in the number of features, requiring fewer steps (decisions) from flat Monte Carlo [29]. Remember that the configuration is built from scratch following the top-down structure of the feature model. The same reason applies to the performance of Greedy MCTS, but take into account that when Greedy MCTS finds a "good" solution it will continue exploring that part of the search space in depth because its exploration constant is always zero. For instance, in JHipster (Figures 8a, 9a, and 10a), when Greedy MCTS finds a "good" solution with a lower number of features, the subsequent valid configurations it finds are similar or even the same configuration previously found. A selective sampling algorithm may help to better understand the configuration space of large-scale feature models. **OC2:** *The challenge here is twofold: (1) to investigate how Monte Carlo methods can be employed in the understanding of the configuration spaces, since a selective sampling algorithm may help to identify the structure and form of the configuration space of large-scale feature models; and (2) use the information about the configuration space to improve the efficiency of Monte Carlo methods in search-based algorithms (e.g., enhancing Monte Carlo methods with* transpositions *and* action groups *[70]).*

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.

5.2. Finding minimum valid configurations

In our running example, a requirement for the development of the AAFM framework in Python is to depend on the smallest number of third-party packages as possible. Thus, another interesting problem is finding a valid configuration with the minimum number of features [58].

Modeling the problem. We reuse the definitions of states ($\boldsymbol{S}$), initial state ($\boldsymbol{s_0}$), terminal condition ($\boldsymbol{t}$), actions ($\boldsymbol{A}$), and state transition function ($\boldsymbol{\theta}$) of the previous problem, while the reward function $\boldsymbol{\mu}$ changes.

- $\boldsymbol{S}$**:** All possible partial and complete configurations ($\boldsymbol{S} = \mathcal{P}(F)$).

- $\boldsymbol{s_0}$: The empty configuration with no feature selected ($\boldsymbol{s_0} = \emptyset$).
- $\boldsymbol{t}$: A configuration is terminal if it is valid and complete:

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

- $\boldsymbol{A}$: The set of valid actions $\boldsymbol{A} = \{a_1, \ldots, a_5\}$.
- $\boldsymbol{\theta}$: $S \times A \to S$.
- $\boldsymbol{\mu}$: The reward function 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:

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

Solving the problem and analyzing results. We use generic Algorithm 3 where the terminal condition checks if the current state is a complete valid configuration. In this problem, the tree search built by MCTS contains statistical information regarding the decisions to select the minimum set of features to form a valid configuration. Figure 13 shows the resulting heatmap with the accumulated Q-values when we use a partial configuration as the initial state. As we will show in the following section, the problem of finding minimum valid configurations is similar to the problem of completing partial configurations, thus the heatmaps show similar information. Q-values capture the expected reward of a decision if we decide to make such choice. When using the empty configuration as the 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 heatmap shows the feature `pyPicosat` in blank, indicating that it has never been considered in a decision. Effectively, the constraint we updated ($\texttt{Win} \vee \texttt{Linux} \Rightarrow \neg\ \texttt{pyPicosat}$) prevents that feature from being part of any configuration, indicating that it is a dead feature.

Lessons learned and open challenges. As discussed in **LS2** Monte Carlo methods can help to explore the configuration space of a feature model, offering information to make better decisions, in contrast to an exact method

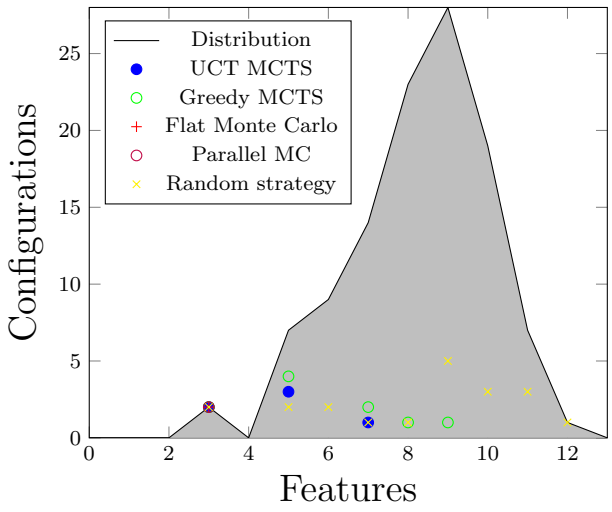


Figure 11: Minimum valid configurations found by Monte Carlo methods in the AAFM Framework feature model (excerpt) for 30 runs. Higher marks *w.r.t.* the y-axis indicate more distinct configurations found for the same number of features.

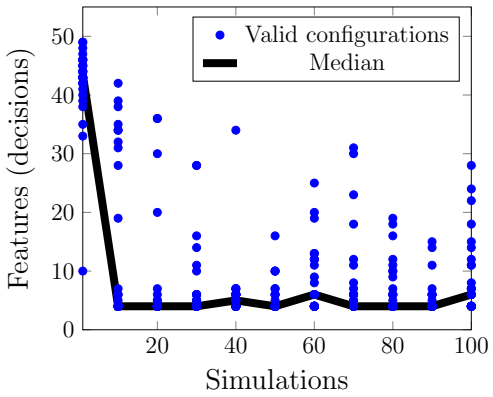


Figure 12: Finding minimum valid configurations in the complete AAFM Framework feature model with MCTS. We vary the number of simulations of MCTS from 1 to 100 and execute 30 times the search algorithm (Algorithm 3) for each number of simulations to calculate the median.

to find the best solution. To illustrate this, Figure 11 shows the minimum valid configurations found by the Monte Carlo methods (100 simulations) for 30 runs of each algorithm. Most of the configurations found are concentrated in the real minimum valid configuration with only 3 features. The MCTS methods have also found others (non-minimum) valid configurations because of the balancing behavior discussed in **LS1** which is useful in the case that the minimum valid configuration is not unique. Figure 12 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 AAFM Python framework, we can observe as the number of decisions decreases as long as we increase the number of simulations, improving the solutions found. The following lessons learned and open challenges have been extracted.

LS3 *Monte Carlo methods are very sensitive to the various inputs and parameters.* Monte Carlo methods are techniques that rely on randomness, and thus, as stated by Lopez-Herrejón [30] these techniques are very sensitive to various inputs and parameters, meaning that slightly changing a value (*e.g.,* the number of simulations) can totally change how you would infer the results. For example, the UCT MCTS and the Greedy MCTS only differ in the value used as the exploration constant, leading to a totally different result as discussed in LS1 (Figure 8).
OC3: *The challenge is to find the most appropriate set of configuration parameters of the Monte Carlo methods for a specific feature model input.*

LS4 *Monte Carlo methods are anytime algorithms which accomplish better results the longer they keep running.* As shown in Figure 12, the number of Monte Carlo simulations affects both the solution quality and the number of steps (decisions) to obtain the solution. As the number of simulations increases, the number of decisions decreases because MCTS can make better decisions, and thus the solutions found are also better. Establishing the appropriate number of simulations is a complex task, and it depends on the size of the search space. A large number of simulations are needed before significant learning could occur in MCTS.
OC4: *Despite the implementation of our MCTS framework also offers a time constraint in seconds as stopping condition for the Monte Carlo decisions, the open challenge here is determining the appropriate num-*

ber of simulations or specifying the time needed in advance to guarantee a certain quality in the solutions.

LS5 *There exist important trade-offs between reproducibility of results, randomness, and performance in Monte Carlo methods.* The results' reproducibility is compromised as Monte Carlo methods rely on randomness to solve problems. Monte Carlo methods make intensive use of random operations such as "choice", "shuffle", or "sample" of actions during both the selection and the simulation phase. Furthermore, the random module is not the only source of randomness, and the results can still present a small variation when using an initialized random seed in Monte Carlo methods and especially in MCTS. The implementation of the MCTS method is usually based on data structures (*e.g.,* the tree search) that do not maintain the order of the states (*e.g.,* sets, maps, or dictionaries). For instance, in a configuration of a feature model, the order of the features is irrelevant. Using those structures does not guarantee obtaining identical results when using random operations such as choosing a random feature in a configuration. Moreover, the states in our framework can represent features, configurations, or even feature models like in the reverse engineering problem. Maintaining a total order for these concepts is not straightforward. For example, defining that a feature model is smaller than others is not trivial. Providing reproducibility also impacts and significatively degrades the performance of the solution because it requires continuously sorting the collections or using inefficient sorted data structures, which Monte Carlo methods do not really need. **OC5:** *The challenge is to address the trade-off between performance and reproducibility due to the randomness nature of the Monte Carlo methods.* To mitigate these issues and provide reproducibility in our MCTS framework, we allow setting a random seed as an argument to initialize the random module and our implementation relies on sorted data structures (*i.e.,* lists) in contrast to sets.

The problem of finding minimum valid configurations can be seen as a specialization of the following problem of completing partial configurations [58].

5.3. Completion of partial configurations

The completion of partial configurations problem 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 partial configurations, some features are undecided (see Definition 3). In our running example, let us suppose we have decided to use the `Glucose` solver in our AAFM framework. We are interested in finding the minimum valid complete configuration with such user's requirement.

Modeling the problem. We modify the initial state $\boldsymbol{s_0}$, while leaving the other definitions $\boldsymbol{S}$, $\boldsymbol{t}$, $\boldsymbol{A}$, $\boldsymbol{\theta}$, and $\boldsymbol{\mu}$ as in the previous problem:

- $\boldsymbol{S}$**:** All possible partial and complete configurations ($\boldsymbol{S} = \mathcal{P}(F)$).
- $\boldsymbol{s_0}$**:** A given partial configuration. To guarantee that the initial partial configuration does not violate the tree hierarchy of the feature model and allows applying our actions $\boldsymbol{A}$, we preprocess 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 the cross-tree constraints, we can use it as the initial state $\boldsymbol{s_0}$ for MCTS. In the other case, the partial selection made by the user is not valid.
- $\boldsymbol{t}$**:** A configuration is terminal if it is valid and complete:

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

- $\boldsymbol{A}$**:** The set of valid actions $\boldsymbol{A} = \{a_1, \ldots, a_5\}$.
- $\boldsymbol{\theta}$**:** $S \times A \to S$.
- $\boldsymbol{\mu}$**:** Difference between the number of features in the feature model ($|F|$) and the number of features in the configuration ($|s|$):

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

Solving the problem and analyzing results. To form a valid initial configuration with the user requirements (*i.e.,* the `Glucose` feature selected), we automatically select the parent features of `Glucose` recursively, obtaining the set of features {`AAFMFramework`, `Solver`, `Glucose`} to be used as the initial configuration. We execute Algorithm 3, whose terminal condition

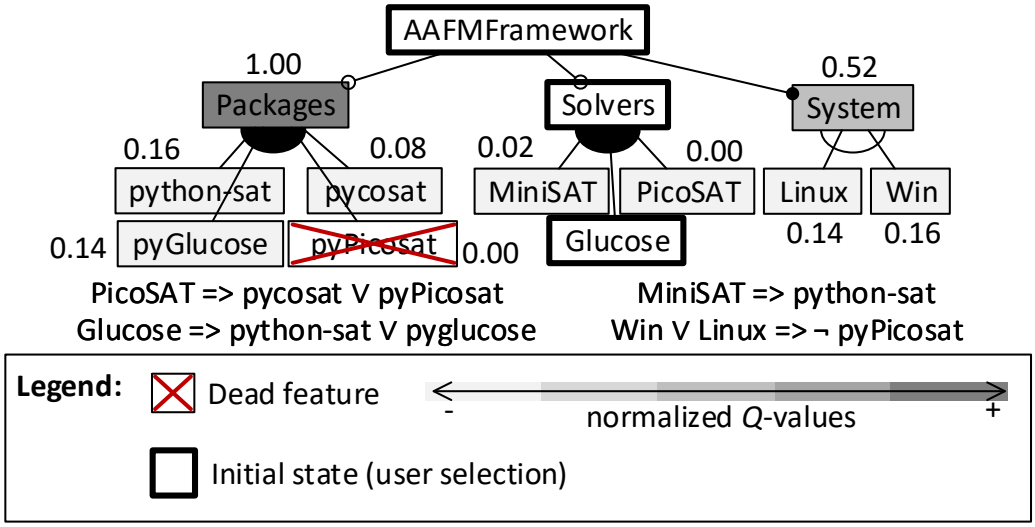


Figure 13: Global heatmap for completion of partial configurations. The initial (input) state is {`AAFMFramework`, `Solver`, `Glucose`}. The heatmap indicates the selections to be first made to get closer to a complete configuration. Features `Packages` and `System` are the two features added in the first steps.

checks if the current state is a valid complete configuration, as in the previous problem. Figure 13 shows the resulting heatmap for completing the partial configuration given as the initial state by the user with the minimum valid selections. Features in darker colors indicate selections to be first made 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 have 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.

Table 1: Feature models corpus used for evaluation, with number of features (#Features), optional features (#Opt), mandatory features (#Mnd), or-group features (#Or), alternative group features (#Xor), the average branching factor (AvgBF), number of cross-tree constraints (#CTCs), and configurations (#Configs).

Feature model	#Features	#Opt	#Mand	#Or	#Xor	AvgBF	#CTCs	#Configs
Pizzas [66]	12	8	4	1	2	2.75	1	42
GPL [38]	18	13	5	1	0	3.40	13	436
Wget [71]	17	15	2	0	1	8.00	0	8192
jHipster [60]	45	36	9	0	10	3.38	13	26256
Tank war [71]	37	30	7	2	6	3.27	0	1.74e6
Mobile media [72]	43	30	13	4	3	3.50	3	2.12e6
AAFM Framework [37]	59	52	7	6	1	4.14	14	1.32e11
WeaFQAs [73]	179	138	41	13	23	3.24	7	2.93e24

Lessons learned and open challenges. Completing valid configurations is an example of an analysis problem which uses the feature model without additional domain information. Therefore, the analysis can be extended to any feature model. To show the feasibility of Monte Carlo methods, we use a set of feature models varying in size and structure (Table 1). Results are shown in Table 2 for all Monte Carlo methods. We identify the following lessons learned and open challenges.

LS6 *Despite Monte Carlo methods often scale to large search spaces [34], SPL problems introduce additional complexity that can affect the feasibility of Monte Carlo methods.* The feasibility of Monte Carlo methods relies on the performance of the simulations. To obtain good solutions with Monte Carlo methods, they need to run a large number of simulations (*e.g.,* hundreds or thousands). Therefore, a simulation must be a lightweight operation, in contrast to a computationally expensive task. In configuration-based analysis, we have identified a bottleneck during the simulations due to the high number of calls to the SAT solver employed to check if the selection of a feature leads to a valid partial configuration. This check needs to be done for each possible feature that can be added to the configuration when applying the actions to select the next feature. A simple cross-tree constraint involving a feature at the top of the feature model can increase considerably the steps needed to complete a valid configuration. Even with only just one simulation, the number of calls to the SAT solver can be exponential in the number of features in the feature model during simulation. **OC6:**

Table 2: Completion of partial configurations. For each feature model (FM), we show the number of features in the minimum valid configuration (Min. $|F|$). We performed 30 execution runs, and show the median values for the number of features in the minimum valid configuration, time (in seconds), and memory (in MB) of each method to complete the initial empty configuration with the minimum number of features.

Feature model	Min	Random strategy			Flat Monte Carlo			UCT MCTS			Greedy MCTS		
	$\|F\|$	$\|F\|$	Time	Mem.	$\|F\|$	Time	Mem.	$\|F\|$	Time	Mem.	$\|F\|$	Time	Mem.
Pizzas	7	9	4e-4	0.20	7	0.12	0.35	7	0.55	1.14	7	0.52	0.90
GPL	7	16	9e-4	0.36	10	0.46	0.88	12	3.44	7.30	11	3.13	6.94
Wget	2	11.5	8e-4	0.25	2	0.05	0.19	2	0.16	3.11	2	0.16	3.10
jHipster	11	20	1.8e-3	0.61	14	1.38	1.79	13	3.23	9.07	13.5	2.33	5.65
Tank war	12	26	2.6e-3	0.70	14.5	2.42	2.35	12	4.27	25.31	14	5.44	23.38
Mobile media	14	36	4e-3	1.04	20	4.65	3.63	17.5	8.72	42.95	16	8.39	36.83
AAFM Framework	4	52.5	0.01	1.54	4	2.07	0.67	5.5	4.15	15.97	6.5	5.00	15.10
WeaFQAs	3	129	0.10	7.44	3	14.03	0.90	3	23.08	18.51	3	22.44	18.57

Runs: 30. Simulations for Monte Carlo methods: 100. Highlighted the best results for time and memory for those methods with the minimum number of features in the configurations found.

The challenge is to define and implement lightweight simulations for configuration-based analysis. Note that we refer in this challenge to the simulation process (*i.e.,* the successive random application of the actions), without considering the execution of the reward function that we will discuss in LS10.

LS7 *Uniform random sampling may improve the performance of Monte Carlo simulations.* A possible solution to address the previous challenge is to replace the actions (Section 5) for selecting the features individually with uniform random sampling [32, 69, 24], which returns a sample of configurations of size equal to the number of simulations needed. We can substitute the random choices during the simulation step of the MCTS method by a random sample representing the terminal states reached by the simulations. Adopting this solution implies three important changes in our MCTS framework. First, we need to separate the actions used to generate the possible successors of a given state from the actions used during simulations, since we still need to know which are the possible alternatives from a given state to analyze and choose the most promising one. Second, to implement random sampling and guarantee uniformity, we can use a Binary Decision Diagram (BDD) solver [32, 8]. Given a partial configuration, the BDD solver returns a sample of complete configurations that includes the features

of the provided partial configuration. However, the BDD solver also presents scalability issues regarding memory when dealing with large-scale feature models [9], and therefore, it limits the applicability of our MCTS framework to those feature models whose associated BDD can be built. **OC7:** *Providing uniform random sampling for large-scale feature models is actually one of the open challenges in the SPL community [17]. Moreover, building a BDD for large-scale feature models is also a well-known identified challenge [9], and the application of BDDs to our MCTS framework evidences the need of addressing this challenge.* And third, we need to determine the size of the sampling for each possible feature alternative available, in contrast to determine only the global number of simulations to be performed to make a decision. In [67], the required sample size is calculated based on the product distribution (*i.e.,* the number of configurations containing a specific feature) by specifying a shared percentage of configurations to be sampled for all alternatives, so that the same ratio of simulations are done for each possible alternative. However, this calculation is only valid for the flat Monte Carlo method, but not for MCTS which balances exploration and exploitation.

LS8 *The independent nature of each simulation in Monte Carlo methods makes them a good target for parallelization to improve the performance, but it requires a deep understanding of Monte Carlo methods, the parallelization mechanisms, and application context (*e.g., *SPL in this paper).* Parallelization has the advantage that more simulations can be performed in a given amount of time. There is a vast literature about parallelizing Monte Carlo methods [34, 50, 51] which identify different parts to be parallelized (*e.g.,* the simulation phase, selection phase, global iterations of MCTS, etc.). Parallelization raises issues such as the combination of the results (Q-values) from different simulations, synchronization of threads/processes when simulations differ in time, or when the simulations depend on the previous ones as in the MCTS method. **OC8:** *The challenge is to implement parallel versions of Monte Carlo methods in the context of the SPL problem that guarantee the soundness/correctness of the methods and reliability of the results.*

The next problem modifies the reward function while reusing the other definitions of the MCTS conceptual framework.

5.4. Optimization of configurations: optimal feature selection problem

The goal of this problem is to find optimum configurations according to some criteria, usually non-functional properties. In our running example, let us suppose that we want to use the most updated and user-rated packages in Python for our AAFM framework.

Modeling the problem. We use the attributes information about the features to define a reward function $\boldsymbol{\mu}$ that serves as a multi-objective function to guide the search. Concretely, we use the release update date and the user rating values publicly available in the Python Package Index (PyPI) repository[6] to enrich our feature model with those attributes (see Definition 2), so that now our reward function $\boldsymbol{\mu}$ can use such information (the other definitions remain the same as in the previous problems):

- $\boldsymbol{S}$**:** All possible partial and complete configurations ($\boldsymbol{S} = \mathcal{P}(F)$).
- $\boldsymbol{s_0}$**:** The empty configuration with no feature selected ($\boldsymbol{s_0} = \emptyset$).
- $\boldsymbol{t}$**:** A configuration is terminal if it is valid and complete:

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

- $\boldsymbol{A}$**:** The set of valid actions $\boldsymbol{A} = \{a_1, \ldots, a_5\}$.
- $\boldsymbol{\theta}$**:** $S \times A \to S$.
- $\boldsymbol{\mu}$**:** The reward function for a terminal configuration is an objective function that evaluates the configuration if it is valid, or returns a penalization if the configuration is not valid:

$$\boldsymbol{\mu(s)} = \begin{cases} \text{ObjectiveFunction(s)}, & \text{if } is\_valid(s) \\ \text{Penalization(s)}, & \text{otherwise} \end{cases}$$

  For instance, for this problem, we define the $ObjectiveFunction(s)$ as a multi-objective function considering the release update date and the user rating of the Python packages in the PyPI repository:

$$ObjectiveFunction(s) = -w_1 \frac{LastUpdate(s)}{N_{LU}} + w_2 \frac{UserRate(s)}{N_{UR}}$$

[6] `https://pypi.org/`

where $LastUpdate(s)$ is the median difference in days of the current date and the last update date for all Python packages in the configuration s, and $UserRate(s)$ is the median of the user ratings for all packages in the configuration. w_1 and w_2 are the weights for each objective function, and N_{LU} and N_{UR} are normalization constants. By assigning different weights to each objective, all possible optimum configurations of the Pareto optimal solutions can be generated. The $Penalization(s)$ function returns a negative value (*e.g.,* -1000 in this case) if the configuration is not valid.

Solving the problem and analyzing the results. As we have only modified the reward function *w.r.t.* the problem of finding minimum configurations, this optimization problem can be solved using the same generic Algorithm 3. The only difference is that the new reward function requires domain-specific knowledge which is provided as attribute information associated with each feature, so we have provided a guided search for MCTS which leads to those valid configurations which maximize the objective function. Figure 14 shows the heatmap that corresponds with step 4 of the search algorithm in which MCTS will select the feature package that get closer to the optimum valid configuration. The user has provided as input the initial configuration {`AAFMFramework`, `Solver`}, and MCTS has selected the features `System`, `Linux`, and `Packages` in the first three steps. In the four step, according to the values of the attributes, the feature `python-sat` is the most promising, in contrast to `pyGlucose` which is a poorly valued package by the users and `pycosat` which is a too old package. Feature `pyPicosat` is not considered because it is a dead feature. After selecting the package `python-sat`, MCTS will select the next feature to complete a valid configuration (i.e., `Glucose` or `MiniSAT`), resulting in an optimum valid configuration: the configuration with features `AAFMFramework`, `Packages`, `Solver`, `System`, `python-sat`, the solver `Glucose`, and the `Linux` system. Note that there are more configurations in the Pareto optimal solutions since the configuration could be completed with `MiniSAT` instead of `Glucose`, or choosing `Win` as system instead of `Linux` in step 2.

Lessons learned and open challenges. While providing domain-specific knowledge to the reward function can help MCTS with a more direct search, it can also limit the efficiency of the balanced approach between exploration and exploitation because MCTS will easily stack on local optima. For instance, multiple configurations with the same updated well-rated packages

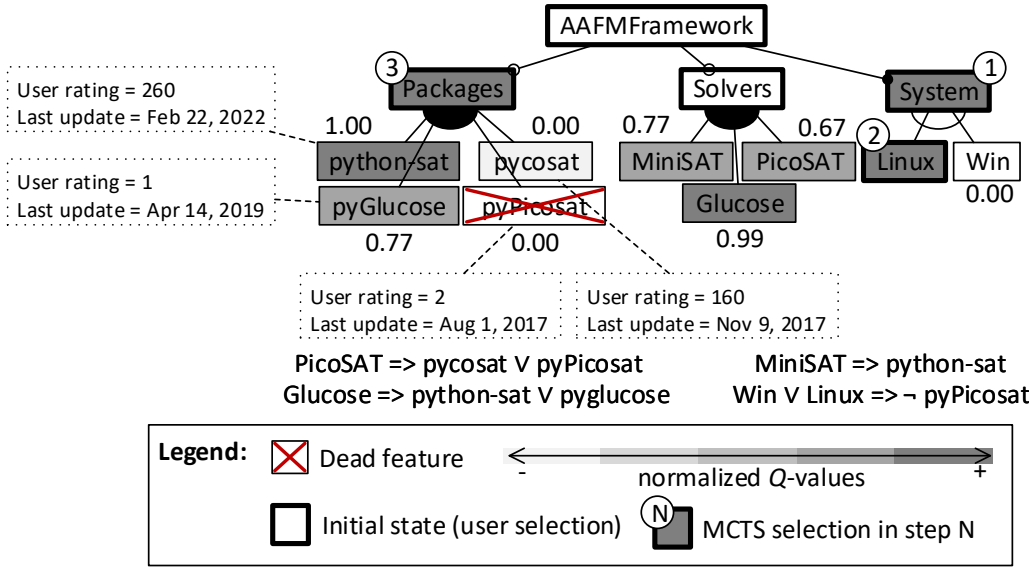


Figure 14: Heatmap of the 4th step of MCTS in the optimal feature selection problem according to the values of the feature attributes.

exist, or configurations where several packages are selected since the feature model allows selecting more than one package. We may also add little domain knowledge to improve the reward function so that, for example, we can penalize those configurations that contain more than one package for implementing a solver (*e.g.*, assigning a negative value to those configurations).

LS9 *Introducing domain-specific knowledge drastically reduces the number of simulations needed, but may also reduce the variance of simulation results.* Apart from the domain knowledge introduced in the reward function which primarily guides the search, other parts of Monte Carlo methods can benefit from feature model knowledge to improve the search. For instance, in our implementation of the MCTS framework, we provide an optional parameter as an argument to allow using as the initial state the partial configuration with the *core features*[7] [1] selected, instead of using an empty configuration. This reduces the number of simulations required as well as the steps done by Monte Carlo methods because there are fewer features to be decided until

[7]The core features are those features that are selected in all configurations.

finding an optimum valid configuration. However, this may also affect the final output because Monte Carlo methods, and especially MCTS, are step-wise techniques in which the selection order of the decisions may affect the subsequent simulations, and MCTS may ignore some regions of the search space, as discussed in **LS1**. **OC9:** *The challenge is to improve the efficiency of Monte Carlo methods by providing as little domain knowledge as possible while maintaining the feasibility of the methods.*

6. Analysis with Feature Models as States

Analyses with MCTS can also be performed over other concepts beyond the configuration space of a 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 represents a feature model. Examples of these problems are the reverse engineering of feature models [14, 21], the extraction of feature models from propositional formulas [74], or the evolution of feature models [13].

Here, we illustrate the reverse engineering of feature models problem [14, 21] defined as follows. Given a set of feature combinations present in a 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 $(\boldsymbol{S, s_0, t, A, \theta, \mu})$:

The set of states $\boldsymbol{S}$ encompasses all feature models that can be built with any combination of the input features F_i following the Definition 1 of feature model. Thus, $S = \{m | m = (F, r, \mathcal{R}, \Pi)\}$ where $F \in \mathcal{P}(F_i)$ and $\Pi \subset \{f \Rightarrow g, f \Rightarrow \neg g | f, g \in F_i\}$[8], r is the root of the feature model, and the set of relations $\mathcal{R}$ is the same as in Definition 1 (*i.e.,* optional, mandatory, alternative, and or).

The initial state $\boldsymbol{s_0}$ is the empty feature model, with no features.

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

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. 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 set of actions is:

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.

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

The reward function $\boldsymbol{\mu}$: for a terminal feature model is a combination of two objective functions extracted from [14]:

$$\boldsymbol{\mu(s)} = Relaxed(s) - MinDiff(s)$$

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 the *deficit* value), and the number of configurations in C_m that are not contained in the required input configurations C_i (also called the *surplus* value). So $MinDiff(s) = deficit(s) + surplus(s)$, value to be minimized.

Solving the problem and analyzing the results. We use as input the set of 110 configurations of our running example (an excerpt is shown in Figure 1). We can use the same generic Algorithm 3 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 15 shows the first four decision steps made by MCTS and the final extracted feature model. The resulting feature model

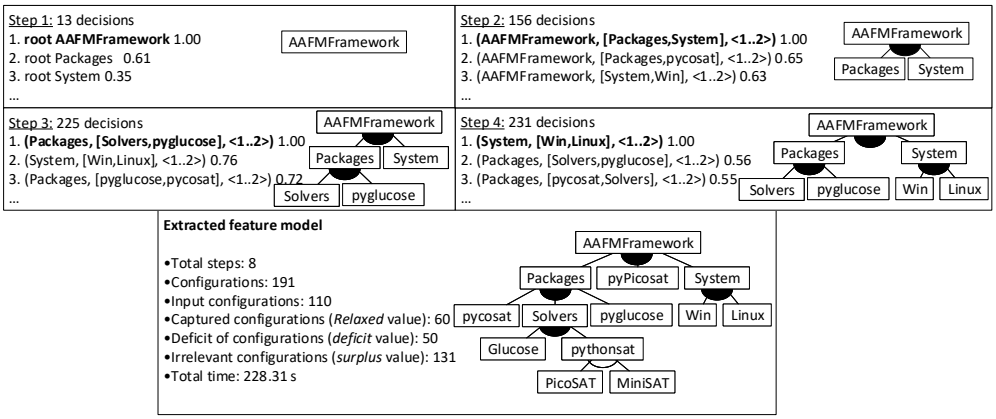


Figure 15: Step-wise decisions for reverse engineering of feature models.

looks similar to the expected one (Figure 1) with some significant differences. It leads to a total of 191 configurations, 60 of which correspond to 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.

An 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 it is to add a specific feature, relation, or constraint. As illustrated in Figure 15, 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.

Lessons learned and open challenges. MCTS can be employed as a user assistant to make better decisions, providing alternatives so that she does not have to blindly rely on the result of a black-box tool, as occurs in

genetic algorithms or neural networks [10, 31, 45]. Therefore, MCTS can be integrated as part of a recommender system [75, 76, 77] to assist the user. However, some considerations should be taken into account when engineering a SPL solution based on Monte Carlo methods, as exposed in the following lessons learned and open challenges.

LS10 *The reward function must be a lightweight function.* As observed in Figure 15, the total time MCTS consumes is considerably high for a small feature model with only 13 features [14]. This is because the reward function in the reverse engineering problem requires generating all configurations of a feature model every time a simulation reaches a final state (a new feature model). Generating all valid configurations from a feature model is one of the most expensive computational tasks in SPLs. **OC10:** *Since the efficiency of the Monte Carlo methods is based on performing as many simulations as possible, a challenge to enable the applicability of Monte Carlo methods is to define lightweight reward functions in the context of the SPL that can evaluate a terminal state as faster as possible, ideally in constant time.*

LS11 *Monte Carlo methods are appropriate for problems that do not require achieving immediate results but taking optimum decisions in the medium and long term.* Providing a high-performance Monte Carlo method is a complex task [34] due to the restrictive requirements of the simulations and reward function regarding performance. There are other techniques such as genetic algorithms and meta-heuristics [10, 31, 45] that have achieved great success in the AAFM area for several problems where both configurations and feature models are the main concepts, such as the feature selection optimization problem [10, 31, 46], or the reverse engineering of feature models problem [14, 21, 78, 79, 80]. While genetic algorithms and meta-heuristics provide better results for search-based optimization problems, Monte Carlo methods are more appropriate for analyzing step-wise decisions and provide knowledge about the possible alternatives as shown through this paper. Despite MCTS and genetic algorithms share some similarities when applied for search-based optimization, they have important differences as Table 3 details. **OC11:** *The challenge is twofold: (1) to find additional SPL problems to those presented in this paper where the application of the Monte Carlo methods makes sense; and (2) to quantitatively compare*

Table 3: 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. Its values are 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).

Monte Carlo methods with other techniques such as genetic algorithms that can handle the same problems.

7. Related Work

This section presents related work about the applications of Monte Carlo methods, especially the MCTS method, and concretely in the context of SPL and AAFM. We also compare MCTS with other techniques such as sampling techniques, genetic algorithms, and traditional approaches (SAT solvers, BDD, constraints programming,. . .) that have been used in the context of the AAFM.

Applications of MCTS. Over the last decade, MCTS has been adopted as part of the solution to many problems in a variety of domains beyond AI games [36], where it has achieved transcendental results (*i.e.,* playing Go and Chess) [36]. For instance, MCTS has achieved great success on complex real-world problems, such as combinatorial optimization to evaluate system

vulnerabilities [81], constraint satisfaction problems (CSP) [82], boolean satisfiability [83], model checking [84], scheduling problems [85], and feature selection problems in the field of machine learning [86], among others. In particular, MCTS has shown great promise in applications where simulation rather than optimization is the most effective decision support tool [34].

Monte Carlo methods in Software Product Lines. To the best of our knowledge, Monte Carlo methods have been mainly applied in SPL from an economic point of view [87, 88, 89]. For example, analyze the return on investment expectations of an SPL [89] and to understand the effort required for building reusable assets [87], to compare the costs and benefits of different test strategies [88], or to estimate the payoff of an SPL [90]. Monte Carlo simulations have also been used for validation when there is a lack of available data [48], as for example, to check the stability of solutions in SPL optimization [12, 91]. Marseguerra et al. [91] combine Genetic Algorithms and Monte Carlo simulation, introducing the concept of *Gradual Monte Carlo optimization*, to evaluate the stability of the solutions in the context of system design (*e.g.,* choice of redundancy configuration and component types). Regarding MCTS, our work is the first study that proposes its application to SPLs.

Randomness in the AAFM. Despite MCTS has not been already applied in the context of AAFM [4]. Several works have incorporated randomness into AAFM. Czarnecki et al. [15] 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. [92] also estimate the feature probabilities to provide feedback to the user. Both works [15, 92] rely on historical data to extract probabilities. Heradio et al. [29] propose statistical analysis to reason on variability models. They extract probability distributions from the whole configuration space to make different analyses, including a uniform random sampling technique [32, 93], but their analyses require building a BDD of the feature model, and this task is intractable for very large-scale models like the Linux kernel [94], existing even a specific challenge for this purpose [9]. MCTS can work directly with the feature model or some other *knowledge compilation technique* [9] (*e.g.,* BDD) as long as it can be modeled using the concepts $(S, s_0, t, A, \theta, \mu)$. One of the most widespread applications of incorporating probability into AAFM has been to assist the user by means of recommendation systems and interactive configuration processes [20, 75, 76, 77, 92]. For instance, Pereira et al. [20, 77] propose different

algorithms [77] for recommender systems in SPL configuration, as well as visualization mechanisms [20] to aid the user. Nöhrer et al. [76] investigate the ordering of the decisions in the decision-making process. Rodas-Silva et al. [75] propose a recommender system to select the 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, it can use 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.

Sampling techniques for AAFM. Configuration sampling [17] is a technique used to avoid exhaustive analysis, providing a subset of all valid configurations. Several sampling strategies have been proposed in the SPL literature [95]: uniform random sampling [23, 24, 32] to select configurations uniformly, coverage-based sampling [15, 96, 97, 98] to select configurations that cover all combinations of t selected features (*e.g.,* pair-wise sampling for $t = 2$), or distance-based sampling [99] to select configurations according to a given probability distribution and a distance metric, among other techniques reviewed in [95]. These techniques have shown great results in SPL testing [16] and learning configuration spaces [100], and despite recent studies [101, 102] have been able to face the scalability challenge [17], they present some limitations when compared with Monte Carlo techniques for the AAFM. Sampling techniques produce samples which are too large to be analyzed [17]. In addition, analyzing and making decisions from a sample of configurations that considers the whole configuration space can be difficult for the user that configures a product. Finally, from the analysis of a particular complete configuration, it is challenging to comprehend a priori the influence of each feature variant in such configuration and in the rest of configurations of the SPL [97]. MCTS can be seen as a selective sampling that combines randomness and evaluation (the reward function) to obtain samples built from step-wise decisions.

Search-based techniques for AAFM. Although sampling techniques, especially uniform random sampling, can be used as a simple way to solve search-related problems on highly-configurable systems [24, 68, 69], there exist other search-based software engineering techniques [30] that have been applied in AAFM. For instance, genetic algorithms and meta-heuristics [31,

10, 45] 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 [31, 10, 46], or the reverse engineering of feature models [21, 14, 78, 79, 80]. A quantitative comparison of MCTS and genetic algorithms is out of the scope of this paper and has been identified as an open challenge in Section 6. To help address this challenge, Table 3 maps the concepts of our MCTS framework to the concepts used in genetic algorithms for search-based problems and exposes the differences between both techniques.

Traditional approaches for AAFM. AAFMs have been traditionally addressed using SAT solving [25, 27, 26], constraint programming [103], description logic [104, 105], BDD solvers [8, 28], or ad-hoc algorithms [106, 107, 108]. An extensive review of about 30 analysis operations that can be performed with these techniques was reported in [1]; and Mendonca et al [27] and Liang et al. [26] report that analyzing feature models with SAT is typically easy. These analysis operations are at a different level of abstraction that the analysis problems presented in this paper. In fact, MCTS often relies on SAT solvers to perform some operations such as checking whether a partial configuration is valid. While there are different approaches to address those analysis problems such as *FastDiag* [58] for completing partial configurations, or genetic algorithms for configuration optimization [109, 10] and reverse engineering of feature models [14], they report the final result (*e.g.,* the complete valid configuration, the optimum configuration, the feature models generated) but no information about the process regarding the decisions that were considered or made, as our MCTS framework offers. Such additional knowledge inferenced during the analysis of the problem allows users to be aware of which decisions were made in each step and to consider alternative decisions that can lead to different desired solutions.

8. Conclusions and Future Work

We have presented a conceptual framework that enables the use of Monte Carlo methods on AAFM, 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 [33] 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 [10]. With this contribution, we envision that different problems and analyses can be addressed using Monte Carlo methods, becoming part of the SPL engineer's toolkit when analyzing feature models and their configurations. This new approach can be of big value to advance the AAFM state-of- the-art.

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 [30] (*e.g.,* genetic algorithms) is also on our agenda. Finally, we plan to extend our MCTS conceptual framework with other variants of the MCTS method [34]. For instance, the independent nature of each simulation in MCTS means that the algorithm is a good target for parallelization [50, 51], so that we can improve its performance. Also, other techniques and extensions of Monte Carlo methods, such as the use of *minimal cut sets* [52], *rare event simulations* [53], or *importance splitting* [54] can be applied to specific problems (*e.g.,* the finding defective configuration problem) to guide the search to effectively handle rare properties and improve the results.

Material

Following open science's good practices, our software artifacts are available publicly.

- MCTS Conceptual Framework: `https://github.com/diverso-lab/fm_montecarlo`

Acknowledgements

This work has been partially funded by the EU FEDER program, the MINECO project OPHELIA (RTI2018-101204-B-C22), the Junta de Andalucia COPERNICA (P20_01224) and *METAMORFOSIS* (FEDER_US-1381375) projects, the Universidad Nacional de Educacion a Distancia under grant 096-034091 2021V/PUNED/008 (OPTIVAC), the Community of Madrid, under the research network CAM ROBOCITY2030-DIH-CM S2018/NMT-4331, and the Spanish Government under Juan de la Cierva—Formación 2019

grant. We would like to thank José A. Troyano for having inspired us in the usage of Monte Carlo methods in software product line analyses, and to A. Germán Márquez, David Romero, and Pablo Pazo for technical support.

References

[1] D. Benavides, S. Segura, A. R. Cortés, Automated analysis of feature models 20 years later: A literature review, Inf. Syst. 35 (6) (2010) 615–636. doi:10.1016/j.is.2010.01.001.
URL https://doi.org/10.1016/j.is.2010.01.001

[2] M. Pol'la, A. Buccella, A. Cechich, Analysis of variability models: a systematic literature review, Software and Systems Modeling 20 (4) (2020) 1–35.

[3] D. Benavides, Variability modelling and analysis during 30 years, in: From Software Engineering to Formal Methods and Tools, and Back, Vol. 11865 of LNCS, Springer, 2019, pp. 365–373. doi:10.1007/978-3-030-30985-5_21.
URL https://doi.org/10.1007/978-3-030-30985-5_21

[4] J. A. Galindo, D. Benavides, P. Trinidad, A. M. Gutiérrez-Fernández, A. Ruiz-Cortés, Automated analysis of feature models: Quo vadis?, Computing 101 (5) (2019) 387–433. doi:10.1007/s00607-018-0646-1.
URL https://doi.org/10.1007/s00607-018-0646-1

[5] M. Raatikainen, J. Tiihonen, T. Männistö, Software product lines and variability modeling: A tertiary study, J. Syst. Softw. 149 (2019) 485–510. doi:10.1016/j.jss.2018.12.027.
URL https://doi.org/10.1016/j.jss.2018.12.027

[6] J. M. Horcas, M. Pinto, L. Fuentes, Empirical analysis of the tool support for software product lines, Software and Systems Modeling (2022). doi:10.1007/s10270-022-01011-2.
URL https://doi.org/10.1007/s10270-022-01011-2

[7] D. Fernández-Amorós, R. Heradio, J. A. Cerrada, C. Cerrada, A scalable approach to exact model and commonality counting for extended feature models, IEEE Trans. Software Eng. 40 (9) (2014) 895–910.

doi:10.1109/TSE.2014.2331073.
URL https://doi.org/10.1109/TSE.2014.2331073

[8] R. Heradio, H. Perez-Morago, D. Fernández-Amorós, R. Bean, F. J. Cabrerizo, C. Cerrada, E. Herrera-Viedma, 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), Vol. 286 of Frontiers in Artificial Intelligence and Applications, 2016, pp. 139–154. doi:10.3233/978-1-61499-674-3-139.
URL https://doi.org/10.3233/978-1-61499-674-3-139

[9] T. Thüm, A BDD for linux?: the knowledge compilation challenge for variability, in: 24th ACM International Systems and Software Product Line Conference (SPLC), Vol. A, 2020, pp. 16:1–16:6. doi:10.1145/3382025.3414943.
URL https://doi.org/10.1145/3382025.3414943

[10] J. Guo, J. H. Liang, K. Shi, D. Yang, J. Zhang, K. Czarnecki, V. Ganesh, H. Yu, SMTIBEA: a hybrid multi-objective optimization algorithm for configuring large constrained software product lines, Softw. Syst. Model. 18 (2) (2019) 1447–1466. doi:10.1007/s10270-017-0610-0.
URL https://doi.org/10.1007/s10270-017-0610-0

[11] V. Nair, T. Menzies, N. Siegmund, S. Apel, Using bad learners to find good configurations, in: 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), ACM, Paderborn, Germany, 2017, pp. 257–267. doi:10.1145/3106237.3106238.
URL https://doi.org/10.1145/3106237.3106238

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

[13] M. Marques, J. Simmonds, P. O. Rossel, M. C. Bastarrica, Software product line evolution: A systematic literature review, Inf. Softw. Technol. 105 (2019) 190–208. doi:10.1016/j.infsof.2018.08.014.
URL https://doi.org/10.1016/j.infsof.2018.08.014

[14] R. E. Lopez-Herrejon, L. Linsbauer, J. A. Galindo, J. A. Parejo, D. Benavides, S. Segura, A. Egyed, An assessment of search-based techniques for reverse engineering feature models, J. Syst. Softw. 103 (2015) 353–369. doi:10.1016/j.jss.2014.10.037.
URL https://doi.org/10.1016/j.jss.2014.10.037

[15] K. Czarnecki, S. She, A. Wasowski, Sample spaces and feature models: There and back again, in: 12th International Conference on Software Product Lines (SPLC), IEEE Computer Society, 2008, pp. 22–31. doi:10.1109/SPLC.2008.49.
URL https://doi.org/10.1109/SPLC.2008.49

[16] J. A. Galindo, H. A. Turner, D. Benavides, J. White, Testing variability-intensive systems using automated analysis: an application to android, Softw. Qual. J. 24 (2) (2016) 365–405. doi:10.1007/s11219-014-9258-y.
URL https://doi.org/10.1007/s11219-014-9258-y

[17] T. Pett, T. Thüm, T. Runge, S. Krieter, M. Lochau, I. Schaefer, Product sampling for product lines: the scalability challenge, in: 23rd International Systems and Software Product Line Conference (SPLC), ACM, Paris, France, 2019, pp. 14:1–14:6. doi:10.1145/3336294.3336322.
URL https://doi.org/10.1145/3336294.3336322

[18] P. Temple, J. A. Galindo, M. Acher, J. Jézéquel, Using machine learning to infer constraints for product lines, in: 20th International Systems and Software Product Line Conference (SPLC), ACM, Beijing, China, 2016, pp. 209–218. doi:10.1145/2934466.2934472.
URL https://doi.org/10.1145/2934466.2934472

[19] J. A. Pereira, L. Maciel, T. F. Noronha, E. Figueiredo, 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, 2018, p. 247. doi:10.1145/3233027.3236395.
URL https://doi.org/10.1145/3233027.3236395

[20] J. A. Pereira, J. Martinez, H. K. Gurudu, S. Krieter, G. Saake, Visual guidance for product line configuration using recommendations and

non-functional properties, in: 33rd Annual ACM Symposium on Applied Computing (SAC), 2018, pp. 2058–2065. doi:10.1145/3167132.3167353.
URL https://doi.org/10.1145/3167132.3167353

[21] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, A. Egyed, Reengineering legacy applications into software product lines: a systematic mapping, Empir. Softw. Eng. 22 (6) (2017) 2972–3016. doi:10.1007/s10664-017-9499-z.
URL https://doi.org/10.1007/s10664-017-9499-z

[22] K. Czarnecki, S. Helsen, U. W. Eisenecker, Formalizing cardinality-based feature models and their specialization, Softw. Process. Improv. Pract. 10 (1) (2005) 7–29. doi:10.1002/spip.213.
URL https://doi.org/10.1002/spip.213

[23] D.-J. Munoz, J. Oh, M. Pinto, L. Fuentes, D. Batory, Uniform random sampling product configurations of feature models that have numerical features, in: 23rd International Systems and Software Product Line Conference (SPLC), Vol. A, 2019, pp. 289–301.

[24] J. Oh, D. S. Batory, M. Myers, N. Siegmund, Finding near-optimal configurations in product lines by random sampling, in: 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), Paderborn, Germany, 2017, pp. 61–71. doi:10.1145/3106237.3106273.
URL https://doi.org/10.1145/3106237.3106273

[25] D. S. Batory, Feature models, grammars, and propositional formulas, in: 9th International Conference on Software Product Lines (SPLC), 2005, pp. 7–20. doi:10.1007/11554844_3.
URL https://doi.org/10.1007/11554844_3

[26] J. H. Liang, V. Ganesh, K. Czarnecki, V. Raman, Sat-based analysis of large real-world feature models is easy, in: Proceedings of the 19th International Conference on Software Product Line, SPLC '15, Association for Computing Machinery, New York, NY, USA, 2015, p. 91–100. doi:10.1145/2791060.2791070.
URL https://doi.org/10.1145/2791060.2791070

[27] M. Mendonca, A. Wasowski, K. Czarnecki, Sat-based analysis of feature models is easy, in: Proceedings of the 13th International Software Product Line Conference, SPLC '09, Carnegie Mellon University, USA, 2009, p. 231–240.

[28] D. Fernández-Amorós, R. Heradio, C. Cerrada, E. Herrera-Viedma, M. J. Cobo, Towards taming variability models in the wild, in: 16th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques (SoMeT), Vol. 297 of Frontiers in Artificial Intelligence and Applications, 2017, pp. 454–465. doi:10.3233/978-1-61499-800-6-454.
URL https://doi.org/10.3233/978-1-61499-800-6-454

[29] R. Heradio, D. Fernández-Amorós, C. Mayr-Dorn, A. Egyed, Supporting the statistical analysis of variability models, in: 41st International Conference on Software Engineering (ICSE), IEEE / ACM, 2019, pp. 843–853. doi:10.1109/ICSE.2019.00091.
URL https://doi.org/10.1109/ICSE.2019.00091

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

[31] H. Yadav, A. C. Kumari, R. Chhikara, Feature selection optimisation of software product line using metaheuristic techniques, International Journal of Embedded Systems 13 (1) (2020) 50–64. doi:10.1504/IJES.2020.108284.
URL https://www.inderscienceonline.com/doi/abs/10.1504/IJES.2020.108284

[32] R. Heradio, D. Fernández-Amorós, J. A. Galindo, D. Benavides, D. Batory, Uniform and scalable sampling of highly configurable systems, Empir. Softw. Eng. 27 (44) (2022). doi:10.1007/s10664-021-10102-5.
URL https://doi.org/10.1007/s10664-021-10102-5

[33] D. P. Kroese, T. Brereton, T. Taimre, Z. I. Botev, Why the monte carlo method is so important today, WIREs Computational Statistics

6 (6) (2014) 386–392. doi:https://doi.org/10.1002/wics.1314.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/wics.1314

[34] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of monte carlo tree search methods, IEEE Transactions on Computational Intelligence and AI in Games 4 (1) (2012) 1–43. doi:10.1109/TCIAIG.2012.2186810.

[35] G. Chaslot, S. Bakkes, I. Szita, P. Spronck, Monte-carlo tree search: A new framework for game ai, in: 4th Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), AAAI Press, 2008, p. 216–217.

[36] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, D. Hassabis, Mastering the game of go without human knowledge, Nature 550 (7676) (2017) 354–359. doi:10.1038/nature24270.
URL https://doi.org/10.1038/nature24270

[37] J. M. Horcas, J. A. Galindo, R. Heradio, D. Fernández-Amorós, D. Benavides, Monte carlo tree search for feature model analyses: a general framework for decision-making, in: M. Mousavi, P. Schobbens (Eds.), SPLC '21: 25th ACM International Systems and Software Product Line Conference, Leicester, United Kingdom, September 6-11, 2021, Volume A, ACM, 2021, pp. 190–201. doi:10.1145/3461001.3471146.
URL https://doi.org/10.1145/3461001.3471146

[38] S. Apel, D. S. Batory, C. Kästner, G. Saake, Feature-Oriented Software Product Lines - Concepts and Implementation, Springer, 2013. doi:10.1007/978-3-642-37521-7.
URL https://doi.org/10.1007/978-3-642-37521-7

[39] J. A. Galindo, D. Benavides, 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, 2020, pp. 52–55.

doi:10.1145/3382026.3425773.
URL https://doi.org/10.1145/3382026.3425773

[40] S. J. Russell, P. Norvig, Artificial Intelligence - A Modern Approach, Fourth edition, Pearson Education, 2020.
URL http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html

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

[42] S. Gelly, D. Silver, Monte-carlo tree search and rapid action value estimation in computer go, Artificial Intelligence 175 (11) (2011) 1856–1875. doi:https://doi.org/10.1016/j.artint.2011.03.007.
URL https://www.sciencedirect.com/science/article/pii/S000437021100052X

[43] L. Kocsis, C. Szepesvári, Bandit based monte-carlo planning, in: Machine Learning: ECML 2006, Berlin, Heidelberg, 2006, pp. 282–293.

[44] G. Chaslot, M. Winands, H. Herik, J. Uiterwijk, B. Bouzy, Progressive strategies for monte-carlo tree search, New Mathematics and Natural Computation 04 (2008) 343–357. doi:10.1142/S1793005708001094.

[45] A. S. Sayyad, J. Ingram, T. Menzies, H. H. Ammar, 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, 2013, pp. 22–27. doi:10.1109/CMSBSE.2013.6604432.
URL https://doi.org/10.1109/CMSBSE.2013.6604432

[46] T. do Nascimento Ferreira, J. A. P. Lima, A. Strickler, J. N. Kuk, S. R. Vergilio, A. T. R. Pozo, Hyper-heuristic based product selection for software product line testing, IEEE Comput. Intell. Mag. 12 (2) (2017) 34–45. doi:10.1109/MCI.2017.2670461.
URL https://doi.org/10.1109/MCI.2017.2670461

[47] Y. Xue, J. Zhong, T. H. Tan, Y. Liu, W. Cai, M. Chen, J. Sun, IBED: combining IBEA and DE for optimal feature selection in software product line engineering, Appl. Soft Comput. 49 (2016) 1215–1231. doi:10.1016/j.asoc.2016.07.040.
URL https://doi.org/10.1016/j.asoc.2016.07.040

[48] M. S. Ali, M. A. Babar, K. Schmid, A comparative survey of economic models for software product lines, in: 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE Computer Society, 2009, pp. 275–278. doi:10.1109/SEAA.2009.89.
URL https://doi.org/10.1109/SEAA.2009.89

[49] S. Segura, J. A. Galindo, D. Benavides, J. A. Parejo, A. R. Cortés, Betty: benchmarking and testing on the automated analysis of feature models, in: U. W. Eisenecker, S. Apel, S. Gnesi (Eds.), Sixth International Workshop on Variability Modelling of Software-Intensive Systems, Leipzig, Germany, January 25-27, 2012. Proceedings, ACM, 2012, pp. 63–71. doi:10.1145/2110147.2110155.
URL https://doi.org/10.1145/2110147.2110155

[50] E. S. Steinmetz, M. Gini, More trees or larger trees: Parallelizing monte carlo tree search, IEEE Transactions on Games (2020) 1–1doi:10.1109/TG.2020.3048331.

[51] G. Chaslot, M. H. M. Winands, H. J. van den Herik, Parallel monte-carlo tree search, in: 6th International Conference on Computers and Games (CG), Vol. 5131 of LNCS, Springer, 2008, pp. 60–71. doi:10.1007/978-3-540-87608-3_6.
URL https://doi.org/10.1007/978-3-540-87608-3_6

[52] C. E. Budde, M. Stoelinga, Automated rare event simulation for fault tree analysis via minimal cut sets, in: International Conference on Measurement, Modelling and Evaluation of Computing Systems, Springer, 2020, pp. 259–277.

[53] G. Rubino, B. Tuffin, Rare event simulation using Monte Carlo methods, John Wiley & Sons, 2009.

[54] C. Jégourel, A. Legay, S. Sedwards, Importance splitting for statistical model checking rare properties, in: 25th International Conference

on Computer Aided Verification (CAV), Vol. 8044 of LNCS, Springer, 2013, pp. 576–591. doi:10.1007/978-3-642-39799-8_38.
URL https://doi.org/10.1007/978-3-642-39799-8_38

[55] R. E. Lopez-Herrejon, S. Illescas, A. Egyed, A systematic mapping study of information visualization for software product line engineering, J. Softw. Evol. Process. 30 (2) (2018). doi:10.1002/smr.1912.
URL https://doi.org/10.1002/smr.1912

[56] L. Wilkinson, M. Friendly, The history of the cluster heat map, The American Statistician 63 (2) (2009) 179–184. doi:10.1198/tas.2009.0033.
URL https://doi.org/10.1198/tas.2009.0033

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

[58] C. Vidal-Silva, J. A. Galindo, J. Giráldez-Cru, D. Benavides, Automated completion of partial configurations as a diagnosis task using fastdiag to improve performance, in: Intelligent Systems in Industrial Applications, Springer International Publishing, Cham, 2021, pp. 107–117.

[59] P. Gazzillo, U. Koc, T. Nguyen, S. Wei, Localizing configurations in highly-configurable systems, in: 22nd International Systems and Software Product Line Conference (SPLC), Vol. 1, ACM, Gothenburg, Sweden, 2018, pp. 269–273. doi:10.1145/3233027.3236404.
URL https://doi.org/10.1145/3233027.3236404

[60] A. Halin, A. Nuttinck, M. Acher, X. Devroey, G. Perrouin, B. Baudry, Test them all, is it worth it? assessing configuration sampling on the jhipster web development stack, Empir. Softw. Eng. 24 (2) (2019) 674–717. doi:10.1007/s10664-018-9635-4.
URL https://doi.org/10.1007/s10664-018-9635-4

[61] M. Bhushan, J. Ángel Galindo Duarte, P. Samant, A. Kumar, A. Negi, Classifying and resolving software product line redundancies using an ontological first-order logic rule based method, Expert Systems with Applications 168 (2021) 114167.

doi:https://doi.org/10.1016/j.eswa.2020.114167.
URL https://www.sciencedirect.com/science/article/pii/S0957417420309052

[62] C. Bogart, C. Kästner, J. Herbsleb, F. Thung, When and how to make breaking changes, ACM Trans. Softw. Eng. Methodol 1 (1) (2021).

[63] A. Felfernig, R. Walter, J. A. Galindo, D. Benavides, S. P. Erdeniz, M. Atas, S. Reiterer, Anytime diagnosis for reconfiguration, J. Intell. Inf. Syst. 51 (1) (2018) 161–182. doi:10.1007/s10844-017-0492-1.
URL https://doi.org/10.1007/s10844-017-0492-1

[64] S. Segura, Automated analysis of feature models using atomic sets, in: Software Product Lines, 12th International Conference, SPLC 2008, Limerick, Ireland, September 8-12, 2008, Proceedings. Second Volume (Workshops), 2008, pp. 201–207.

[65] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-oriented domain analysis (foda) feasibility study, Tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, technical Report CMU/SEI-90-TR-21 (1990).

[66] A. Knüppel, T. Thüm, S. Mennicke, J. Meinicke, I. Schaefer, Is there a mismatch between real-world feature models and product-line research?, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 291–302.

[67] J. M. H. Aguilera, A. G. Márquez, J. A. Galindo, D. Benavides, Monte carlo simulations for variability analyses in highly configurable systems, in: M. Aldanondo, A. A. Falkner, A. Felfernig, M. Stettinger (Eds.), Proceedings of the 23rd International Configuration Workshop (CWS/ConfWS 2021), Vienna, Austria, 16-17 September, 2021, Vol. 2945 of CEUR Workshop Proceedings, CEUR-WS.org, 2021, pp. 37–44.
URL http://ceur-ws.org/Vol-2945/32-JMHA-ConfWS21_paper_19.pdf

[68] R. Heradio, D. Fernández-Amorós, V. Ruiz, M. J. Cobo, A rule-learning approach for detecting faults in highly configurable software systems

from uniform random samples, in: 55th Hawaii International Conference on System Sciences, HICSS, ScholarSpace, Maui, Hawaii, USA, 2022, pp. 1–10.
URL http://hdl.handle.net/10125/79595

[69] D. S. Batory, J. Oh, R. Heradio, D. Benavides, Logic, Computation and Rigorous Methods - Essays Dedicated to Egon Börger on the Occasion of His 75th Birthday, Springer, 2021, Ch. Product Optimization in Stepwise Design, pp. 63–81. doi:10.1007/978-3-030-76020-5_4.

[70] B. E. Childs, J. H. Brodeur, L. Kocsis, Transpositions and move groups in monte carlo tree search, in: 2008 IEEE Symposium On Computational Intelligence and Games, 2008, pp. 389–395. doi:10.1109/CIG.2008.5035667.

[71] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, S. S. Kolesnikov, Scalable prediction of non-functional properties in software product lines: Footprint and memory consumption, Information and Software Technology 55 (3) (2013) 491–507.

[72] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, et al., Evolving software product lines with aspects, in: ACM/IEEE 30th International Conference on Software Engineering, 2008, pp. 261–270.

[73] J. Horcas, M. Pinto, L. Fuentes, Software product line engineering: a practical experience, in: 23rd International Systems and Software Product Line Conference, SPLC 2019, ACM, 2019, pp. 25:1–25:13.

[74] K. Czarnecki, A. Wasowski, Feature diagrams and logics: There and back again, in: 11th International Conference on Software Product Lines (SPLC), IEEE Computer Society, 2007, pp. 23–34. doi:10.1109/SPLINE.2007.24.
URL https://doi.org/10.1109/SPLINE.2007.24

[75] J. Rodas-Silva, J. A. Galindo, J. García-Gutiérrez, D. Benavides, Selection of software product line implementation components using recommender systems: An application to wordpress, IEEE Access 7 (2019) 69226–69245. doi:10.1109/ACCESS.2019.2918469.
URL https://doi.org/10.1109/ACCESS.2019.2918469

[76] A. Nöhrer, A. Egyed, Optimizing user guidance during decision-making, in: 15th International Conference on Software Product Lines (SPLC), IEEE Computer Society, 2011, pp. 25–34. doi:10.1109/SPLC.2011.45.
URL https://doi.org/10.1109/SPLC.2011.45

[77] J. A. Pereira, P. Matuszyk, S. Krieter, M. Spiliopoulou, G. Saake, 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, 2016, pp. 120–131. doi:10.1145/2993236.2993249.
URL https://doi.org/10.1145/2993236.2993249

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

[79] W. K. G. Assunção, R. E. Lopez-Herrejon, L. Linsbauer, S. R. Vergilio, A. Egyed, Multi-objective reverse engineering of variability-safe feature models based on code dependencies of system variants, Empir. Softw. Eng. 22 (4) (2017) 1763–1794. doi:10.1007/s10664-016-9462-4.
URL https://doi.org/10.1007/s10664-016-9462-4

[80] L. Linsbauer, R. E. Lopez-Herrejon, A. Egyed, Variability extraction and modeling for product variants, Softw. Syst. Model. 16 (4) (2017) 1179–1199. doi:10.1007/s10270-015-0512-y.
URL https://doi.org/10.1007/s10270-015-0512-y

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

[82] S. Baba, Y. Joe, A. Iwasaki, M. Yokoo, Real-time solving of quantified csps based on monte-carlo game tree search, in: 22nd International Joint Conference on Artificial Intelligence (IJCAI), Barcelona, Spain, 2011, pp. 655–661. doi:10.5591/978-1-57735-516-8/IJCAI11-116.
URL https://doi.org/10.5591/978-1-57735-516-8/IJCAI11-116

[83] A. Previti, R. Ramanujan, M. Schaerf, B. Selman, Monte-carlo style UCT search for boolean satisfiability, in: Artificial Intelligence Around Man and Beyond (AI*IA), Berlin, Heidelberg, 2011, pp. 177–188.

[84] S. Poulding, R. Feldt, Heuristic model checking using a monte-carlo tree search algorithm, in: Annual Conference on Genetic and Evolutionary Computation (GECCO), Association for Computing Machinery, 2015, p. 1359–1366. doi:10.1145/2739480.2754767.
URL https://doi.org/10.1145/2739480.2754767

[85] H. Nakhost, M. Müller, Monte-carlo exploration for deterministic planning, in: 21st International Jont Conference on Artifical Intelligence (IJCAI), Morgan Kaufmann Publishers Inc., San Francisco, USA, 2009, p. 1766–1771.

[86] M. U. Chaudhry, J.-H. Lee, MOTiFS: Monte carlo tree search based feature selection, Entropy 20 (5) (2018). doi:10.3390/e20050385.
URL https://www.mdpi.com/1099-4300/20/5/385

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

[88] D. Ganesan, J. Knodel, R. Kolb, U. Haury, G. Meier, 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), 2007, pp. 74–83. doi:10.1109/SPLINE.2007.21.
URL https://doi.org/10.1109/SPLINE.2007.21

[89] M. Nonaka, L. Zhu, 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, 2007, pp. 63–73. doi:10.1109/SPLINE.2007.35.
URL https://doi.org/10.1109/SPLINE.2007.35

[90] R. Heradio, D. Fernández-Amorós, L. Torre-Cubillo, A. P. García-Plaza, Improving the accuracy of COPLIMO to estimate the payoff of a software product line, Expert Syst. Appl. 39 (9) (2012) 7919–7928.

doi:10.1016/j.eswa.2012.01.109.
URL https://doi.org/10.1016/j.eswa.2012.01.109

[91] M. Marseguerra, E. Zio, L. Podofillini, 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, Vol. 39 of Studies in Computational Intelligence, Springer, 2007, pp. 101–150. doi:10.1007/978-3-540-37368-1_4.
URL https://doi.org/10.1007/978-3-540-37368-1_4

[92] J. Martinez, T. Ziadi, R. Mazo, T. F. Bissyandé, J. Klein, Y. L. Traon, 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, 2014, pp. 50–59. doi:10.1109/VISSOFT.2014.18.
URL https://doi.org/10.1109/VISSOFT.2014.18

[93] R. Heradio, D. Fernández-Amorós, J. A. Galindo, D. Benavides, Uniform and scalable sat-sampling for configurable systems, in: 24th ACM International Systems and Software Product Line Conference (SPLC), Vol. A, ACM, Montreal, Canada, 2020, pp. 17:1–17:11. doi:10.1145/3382025.3414951.
URL https://doi.org/10.1145/3382025.3414951

[94] S. She, R. Lotufo, T. Berger, A. Wasowski, K. Czarnecki, The variability model of the linux kernel, in: 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS), Vol. 37 of ICB-Research Report, Universität Duisburg-Essen, Linz, Austria, 2010, pp. 45–51.
URL http://www.vamos-workshop.net/proceedings/VaMoS_2010_Proceedings.pdf

[95] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, I. Schaefer, A classification of product sampling for software product lines, in: 22nd International Systems and Software Product Line Conference (SPLC), 2018, pp. 1–13. doi:10.1145/3233027.3233035.
URL https://doi.org/10.1145/3233027.3233035

[96] S. Fischer, R. E. Lopez-Herrejon, R. Ramler, A. Egyed, A preliminary empirical assessment of similarity for combinatorial interaction testing of software product lines, in: 9th Workshop on Search-Based Software Testing (SBST@ICSE), 2016, pp. 15–18. doi:10.1145/2897010.2897011.
URL https://doi.org/10.1145/2897010.2897011

[97] J. A. Pereira, M. Acher, H. Martin, J. Jézéquel, Sampling effect on performance prediction of configurable systems: A case study, in: ACM/SPEC International Conference on Performance Engineering (ICPE), Amsterdam, The Netherlands, 2020, pp. 277–288. doi:10.1145/3358960.3379137.
URL https://doi.org/10.1145/3358960.3379137

[98] M. F. Johansen, O. Haugen, F. Fleurey, An algorithm for generating t-wise covering arrays from large feature models, in: Proceedings of the 16th International Software Product Line Conference - Volume 1, SPLC '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 46–55. doi:10.1145/2362536.2362547.
URL https://doi.org/10.1145/2362536.2362547

[99] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, S. Apel, Distance-based sampling of software configuration spaces, in: 41st International Conference on Software Engineering (ICSE), IEEE/ACM, 2019, pp. 1084–1094. doi:10.1109/ICSE.2019.00112.
URL https://doi.org/10.1109/ICSE.2019.00112

[100] J. A. Pereira, H. Martin, M. Acher, J. Jézéquel, G. Botterweck, A. Ventresque, Learning software configuration spaces: A systematic literature review, CoRR abs/1906.03018 (2019). arXiv:1906.03018.
URL http://arxiv.org/abs/1906.03018

[101] M. Acher, G. Perrouin, M. Cordy, BURST: A Benchmarking Platform for Uniform Random Sampling Techniques, Association for Computing Machinery, New York, NY, USA, 2021, p. 36–40.
URL https://doi.org/10.1145/3461002.3473070

[102] J. Oh, P. Gazzillo, D. Batory, M. Heule, M. Myers, Scalable uniform sampling for real-world software product lines, Tech. rep., Technical

Report TR-20-01, Dept. of Computer Science, University of Texas at ... (2020).

[103] D. Benavides, P. T. Martín-Arroyo, A. R. Cortés, Automated reasoning on feature models, in: O. Pastor, J. F. e Cunha (Eds.), Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal, June 13-17, 2005, Proceedings, Vol. 3520 of Lecture Notes in Computer Science, Springer, 2005, pp. 491–503. doi:10.1007/11431855_34.
URL https://doi.org/10.1007/11431855_34

[104] H. H. Wang, Y. F. Li, J. Sun, H. Zhang, J. Pan, Verifying feature models using owl, Journal of Web Semantics 5 (2) (2007) 117–129, software Engineering and the Semantic Web. doi:https://doi.org/10.1016/j.websem.2006.11.006.
URL https://www.sciencedirect.com/science/article/pii/S1570826807000042

[105] S. Fan, N. Zhang, Feature model based on description logics, in: B. Gabrys, R. J. Howlett, L. C. Jain (Eds.), Knowledge-Based Intelligent Information and Engineering Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2006, pp. 1144–1151.

[106] R. C. Bachmeyer, H. S. Delugach, A conceptual graph approach to feature modeling, in: U. Priss, S. Polovina, R. Hill (Eds.), Conceptual Structures: Knowledge Architectures for Smart Applications, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 179–191.

[107] J. White, B. Dougherty, D. C. Schmidt, Selecting highly optimal architectural feature sets with filtered cartesian flattening, Journal of Systems and Software 82 (8) (2009) 1268–1284, sI: Architectural Decisions and Rationale. doi:https://doi.org/10.1016/j.jss.2009.02.011.
URL https://www.sciencedirect.com/science/article/pii/S0164121209000284

[108] R. Gheyi, T. Massoni, P. Borba, Algebraic laws for feature models, J. Univers. Comput. Sci. 14 (21) (2008) 3573–3591. doi:10.3217/jucs-014-21-3573.
URL https://doi.org/10.3217/jucs-014-21-3573

[109] J. M. Horcas, D. Struber, A. Burdusel, J. Martinez, S. Zschaler, We're not gonna break it! consistency-preserving operators for efficient product line configuration, IEEE Transactions on Software Engineering (2022) 1–1doi:10.1109/TSE.2022.3171404.