A Deep Reinforcement Learning Approach to Configuration Sampling Problem

Abolfazli, Amir ; Spiegelberg, Jakob ; Anand, Avishek ; Palmer, Gregory

DOI
10.1109/ICDM58522.2023.00009

Publication date
2023

Document Version
Final published version

Published in
Proceedings of the 2023 IEEE International Conference on Data Mining (ICDM)

A Deep Reinforcement Learning Approach to Configuration Sampling Problem

Amir Abolfazli
L3S Research Center
Hannover, Germany
abolfazli@l3s.de

Jakob Spiegelberg
Volkswagen AG
Wolfsburg, Germany
jakob.spiegelberg@volkswagen.de

Gregory Palmer
L3S Research Center
Hannover, Germany
gpalmer@l3s.de

Avishek Anand
Delft University of Technology
Delft, Netherlands
avishek.anand@tudelft.nl

2023 IEEE International Conference on Data Mining (ICDM) | 979-8-3503-0788-7/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICDM58522.2023.00009

***Abstract*—Configurable software systems have become increasingly popular as they enable customized software variants. The main challenge in dealing with configuration problems is that the number of possible configurations grows exponentially as the number of features increases. Therefore, algorithms for testing customized software have to deal with the challenge of tractably finding potentially faulty configurations given exponentially large configurations. To overcome this problem, prior works focused on sampling strategies to significantly reduce the number of generated configurations, guaranteeing a high t-wise coverage. In this work, we address the configuration sampling problem by proposing a deep reinforcement learning (DRL) based sampler that efficiently finds the trade-off between exploration and exploitation, allowing for the efficient identification of a minimal subset of configurations that covers all t-wise feature interactions while minimizing redundancy. We also present the CS-Gym, an environment for the configuration sampling. We benchmark our results against heuristic-based sampling methods on eight different feature models of software product lines and show that our method outperforms all sampling methods in terms of sample size. Our findings indicate that the achieved improvement has major implications for cost reduction, as the reduction in sample size results in fewer configurations that need to be tested.**

***Index Terms*—reinforcement learning, configuration sampling, software testing**

I. Introduction

Configuration sampling (CS) is an important problem in production and engineering companies that use software product lines (SPLs) to enable customized software variants to the requirements of users – like a customized vehicle, consumer electronics, and software products, to name a few. Each product of an SPL is defined by a unique set of features called a *configuration*. The set of all features and the constraints among features defines a *feature model* [1]. Specifically, solving a configuration sampling problem can help identify defects in software product lines (SPLs) by selecting a small set of configurations to test specific features and their interactions.

Efficiently identifying faulty configurations is not only an essential step in the software development lifecycle for ensuring the reliability and performance of software systems but also a strategic initiative that can lead to substantial cost savings for organizations by reducing debugging time and maintenance costs [2].

By testing a representative sample of configurations, the aim is to reveal any potential defects or errors that may arise from the interaction between different features. However, a large configuration space could have over a quadrillion $\left(> 10^{15}\right)$ configurations, and exhaustively enumerating all configurations for testing is simply infeasible.

To reduce such a large search space, prior works have focused on *sampling strategies* to significantly reduce the number of generated configurations. A family of such methods is called *t-wise coverage* samplers taking into account t-wise combinations of features that account for interaction failures occurring when two or more features interacting cause the program to reach an incorrect result [3]. Their objective is to achieve a high t-wise coverage, whereby all combinations of t features are covered by at least one configuration in the set. However, achieving 100% coverage may not be feasible for large configuration spaces [4]. In practice, it is well-known that most failures are triggered by only one $(t = 1)$ or two features $(t = 2)$ [3].

Despite the success of recent sampling algorithms in identifying numerous faulty test cases, they often generate an excessively large sample size, which poses challenges for expert software testers [5]. This inefficiency in dealing with a vast number of test cases highlights the need for a method that can effectively discover all faulty configurations covering all t-wise feature interaction pairs while generating a smaller number of test cases for examination [6].

Existing approaches use different sampling techniques – like greedy techniques [4], [7]–[10], local search techniques [11]–[13], population-based techniques [14]–[16], manual selection techniques [17], or feature interaction and coverage based techniques [18]–[21]. However, most of these approaches are top-down approaches and are limited by their reliance on hand-crafted heuristics, which may not be able to capture the full complexity of the configuration space. This can lead to suboptimal performance or a large sample size that is not minimal [19].

Our main idea in this paper is to attempt to solve the configuration sampling problem by using deep reinforcement learning (DRL) to learn more sophisticated representations of the configuration space. We claim that DRL has the potential to drastically reduce the search space required for configuration sampling, resulting in smaller sample sizes. In the setting of the configuration sampling problem, our *environment* is a highly configurable system, and the feedback is the *performance* of the system under a particular configuration.

979-8-3503-0788-7/23/$31.00 ©2023 IEEE
DOI 10.1109/ICDM58522.2023.00009

Our proposed approach involves using a t-wise sampling algorithm to generate an initial solution for the problem and subsequently training an auto-encoder model to derive state embedding for the given set of configurations. Due to the variable nature of the configurations at different time steps and the potential for an exceedingly large number of configurations, it is necessary to employ a condensed vector representation of the state at each time-step. We use the Branching Dueling Q-Network (BDQ) [22] to handle large multi-dimensional discrete action spaces. To optimize the sampling process, we limit the number of actions taken to only those with a net positive effect on configuration coverage. Specifically, if the addition of a configuration covers at least one feature interaction pair not previously covered, or if the removal of a configuration does not decrease the number of covered feature interaction pairs from the previous time-step, the action is deemed 'applicable' and performed accordingly.

In contrast to existing combinatorial approaches, our RL-based approach adds or removes configurations based on their coverage at different time steps. The RL-agent learns to make the best decision at each time-step using a state representation that includes the current configurations. This differs from heuristic-based methods that consider all possible feature interaction pairs upfront and may not be as effective at generating optimal configurations. Our methodology seeks to balance the trade-off between efficiency and coverage to improve the quality of software products developed using SPLs.

We conduct extensive experimental studies on eight real-world feature models and three baselines. We see that our proposed methodology outperforms existing heuristic-based sampling algorithms on various feature models of software product lines (SPLs). First, We show that our approach can significantly reduce the sample size from the initial solution generated by the t-wise combination sampler to a greater extent than other methods. Furthermore, our approach can rapidly converge to small sample sizes. However, the episode at which the optimal sample size is reached varies depending on the specific feature model of the SPL being tested. These findings indicate the effectiveness of our approach in improving the efficiency and effectiveness of configuration sampling in SPLs, which can ultimately lead to higher-quality software products.

The contributions of this work are summarized as follows:

- We propose a DRL-based method for the configuration sampling problem that takes input from the initial set of configurations generated by the t-wise sampler and finds a minimal set of configurations whose size is smaller than that of the state-of-the-art heuristic-based sampling algorithms.
- We present a Gym-based configuration sampling environment that takes input as a feature model of software product lines in the format of a CNF formula or a DIMACS file and provides all the functionalities that DRL methods dealing with multi-dimensional action spaces can be directly applied.

The rest of this paper is organized as follows. Section II gives an overview of related work. Section III gives the background on configuration sampling and RL. In Section IV, we introduce our RL-based approach to configuration sampling problem. Section V describes our experiment setup. We discuss our results in Section VI, and conclude with suggestions for future work in Section VIII.

II. Related Work

Configuration sampling. Existing approaches use different sampling techniques – like greedy techniques [4], [7]–[10], local search techniques [11]–[13], population-based techniques [14]–[16], manual selection techniques [17], or feature interaction and coverage based techniques [18]–[21]. However, most of these approaches are top-down approaches and are limited by their reliance on hand-crafted heuristics, which may not be able to capture the full complexity of the configuration space. This can lead to suboptimal performance or a sample size that is not minimal [19].

The recent work [23] suggests that the greedy and meta-heuristic techniques are more often used and compared to other techniques.

Deep reinforcement learning for combinatorial optimization. Reinforcement learning has been used to solve many combinatorial optimization problems, including the Traveling Salesman Problem [24], [25], Maximum Cut Problem [26]–[29], Bin Packing Problem [30]–[32], Boolean Satisfiability Problem [33], Minimum Vertex Cover Problem [29], [34], and Maximum Independent Set [26], [35]. However, to the best of our knowledge, RL has never been applied to the configuration sampling problem.

Deep reinforcement learning for large multi-dimensional action spaces. To address large multi-dimensional discrete action spaces in reinforcement learning, [36] proposes the Wolpertinger policy architecture, which combines DDPG with an approximate nearest-neighbor method, enabling logarithmic-time lookup complexity relative to the action space cardinality. For scalability with increasing action dimensions, [37] combines DQN [38] with independent Q-learning, where each agent independently learns its own state-action value function. Xiong et al. in [39] propose the parameterized deep Q-network (P-DQN) for discrete-continuous hybrid action spaces, extending DQN with deterministic policy for continuous actions. However, P-DQN is not applicable to environments with discrete-discrete hybrid action spaces [40]. [22] introduces the Branching Dueling Q-Network (BDQ), an approach for handling high-dimensional discrete or continuous action spaces using Q-learning, featuring a shared decision module followed by multiple network branches for each action dimension, allowing linear growth in the number of network outputs with degrees of freedom.

In this work, we utilize BDQ for its efficient handling of multi-dimensional action spaces and integration of dueling networks' benefits, resulting in enhanced state-action value estimation and policy learning, making it apt for the configuration sampling problem.

III. Background

A. Reinforcement Learning

The reinforcement learning (RL) problem is typically modeled by a Markov decision process (MDP), formulated as a tuple $(\mathcal{S}, \mathcal{A}, p, r, \gamma)$, with a state space $\mathcal{S}$, an action space $\mathcal{A}$, and transition dynamics $p\left(s' \mid s, a\right)$ [41]. At each discrete time step, the agent performs an action $a \in \mathcal{A}$ in a state $s \in \mathcal{S}$, and transitions to a new state $s' \in \mathcal{S}$ based on the transition dynamics $p\left(s' \mid s, a\right)$, and receives a reward $r\left(s, a, s'\right)$. The action a is *applicable* within the state s if $p(s'|s, a) > 0$ and inapplicable if $p(s'|s, a) = 0$. The goal of the agent is to maximize the expectation of the sum of discounted rewards, also known as the *return* $R_t = \sum_{i=t+1}^{\infty} \gamma^i r\left(s_i, a_i, s_{i+1}\right)$, which weighs future rewards with respect to the *discount factor* $\gamma \in [0, 1)$, determining the effective horizon. The agent makes decisions via a policy $\pi : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$, which maps a given state s to a probability distribution over the action space $\mathcal{A}$. For a given policy π, the value function is defined as the expected return of an agent, starting from state s, performing action a, and following the policy $Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[R_t \mid s, a\right]$. The state-action value function can be computed through the Bellman equation of the Q function:

$$Q^{\pi}(s, a) = \mathbb{E}_{s' \sim p}\left[r\left(s, a, s'\right) + \gamma \mathbb{E}_{a' \sim \pi} Q^{\pi}(s', a')\right]. \quad (1)$$

Given Q^{π}, the optimal policy $\pi^* = \max_a Q^*(s, a)$, can be obtained by greedy selection over the optimal value function $Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$. For environments confronting agents with the curse of dimensionality, the value can be estimated with a differentiable function approximator $Q_{\theta}(s, a)$, with parameters θ.

DQN [42] uses deep neural networks to approximate the Q-function. To improve convergence and performance, DQN incorporates experience replay, which reduces correlations between different training samples, and uses the target Q-network, stabilizing the target Q-value.

Let Q_{θ} denote the Q-function parameterized by θ, DQN computes the target Q-value y_t^{DQN} as follows:

$$y_t^{DQN} = r + \gamma \max_{a'} Q_{\theta}\left(s', a'\right). \quad (2)$$

Double DQN (DDQN) [43] improves the performance of DQN by addressing the problem of overestimation of Q-values. DDQN decouples action selection and Q-value estimation when calculating the target Q-value y_t^{DDQN}. In BDQ, the action is selected using the current Q-network Q_{θ} while the target Q-value is estimated using the DQN's target Q-network $Q_{\theta'}$, as follows:

$$y_t^{DDQN} = r + \gamma Q_{\theta'}\left(s', \arg\max_{a'} Q_{\theta}\left(s', a'\right)\right). \quad (3)$$

Dueling DQN improves DDQN by using a dueling architecture [44]. The dueling network comprises two streams: a scalar state-value estimation stream and an advantage function estimation stream, separating the task of learning the Q-function into learning the value and advantage functions. This split facilitates more efficient identification of the correct action during the policy evaluation, as the network can learn the goodness of the states without the need to learn the value of each action for each state. Dueling DQN computes the target Q-value $y_t^{DuelingDQN}$ as follows:

$$y_t^{DuelingDQN} = V(s) + \left(A(s, a) - \frac{1}{|\mathcal{A}|}\sum_{a'} A\left(s, a'\right)\right). \quad (4)$$

The Branching Dueling Q-Network (BDQ) [22] integrates the dueling network architecture into the action branching framework, resulting in improved performance. Combining the dueling architecture with action branching is particularly beneficial for learning in environments with large action spaces, as the dueling architecture can rapidly identify redundant actions and achieve better generalization by learning a common value for a wide range of similar actions.

In order to adapt the dueling architecture into action branching network, BDQ distributes the representation of the (state-dependent) action advantages on the several action branches, meanwhile, adding a single additional branch for estimating the state-value function. Similar to the dueling architecture, the advantages and the state value are combined, via a special aggregation layer, to produce estimates of the distributed action values. For the aggregation method, the BDQ locally subtracts each branch's mean advantage from its sub-action advantages, prior to their summation with the state value.

For an action dimension $d \in \{1, \ldots, N\}$ with $|\mathcal{A}_d| = n$ discrete sub-actions, the individual branch's Q-value at state $s \in \mathcal{S}$ and sub-action $a_d \in \mathcal{A}_d$ is expressed in terms of the common state value $V(s)$ and the corresponding (state-dependent) sub-action advantage $A_d\left(s, a_d\right)$ by:

$$Q_{\theta}^{d}\left(s, a_d\right) = V(s) + \left(A_d\left(s, a_d\right) - \frac{1}{n}\sum_{a'_d \in \mathcal{A}_d} A_d\left(s, a'_d\right)\right). \quad (5)$$

For generating the temporal-difference (TD) targets for the DQN updates, BDQ uses the mean operator:

$$y_t^{BDQ} = r + \gamma \frac{1}{N}\sum_{d} Q_{\theta'}^{d}\left(s', \arg\max_{a'_d \in \mathcal{A}_d} Q_{\theta}^{d}\left(s', a'_d\right)\right), \quad (6)$$

where $Q_{\theta'}^{d}$ denotes the branch d of the target network $Q_{\theta'}$.

In BDQ, the loss function is defined as the expected value of the mean squared TD error across the branches:

$$L = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}}\left[\frac{1}{N}\sum_{d}\left(y_d - Q_d\left(s, a_d\right)\right)^2\right], \quad (7)$$

where $\mathcal{D}$ denotes a prioritized experience replay buffer and a denotes the joint-action tuple $(a_1, a_2, \ldots, a_N)$.

B. Configuration Sampling

Configuration sampling (CS) is an NP-hard problem that is a well-known special case of the *set cover problem* [45], that is a combinatorial testing technique used to efficiently select a subset of configurations to test, while ensuring that all possible t-wise interactions among the features are covered. The main

goal is to identify and test the most significant interactions, reducing the testing effort and resources needed, while maintaining high defect detection rates. CS can be viewed as a bipartite graph, with the configurations represented by vertices on the left side, the feature interaction pairs represented by vertices on the right side, and edges representing the coverage of pairs by configurations. The task is then to find a minimal subset of configurations (left-vertices) that covers all of the feature interaction pairs (right-vertices). The illustration of the configuration sampling problem using a bipartite graph is shown in Figure 1.

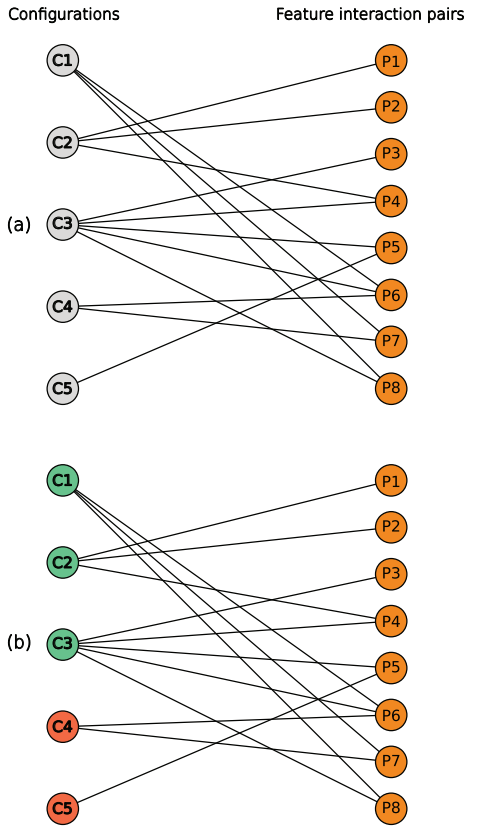

Figure 1. **(a)** Illustration of the configuration sampling problem as a bipartite graph. The goal is to find a minimal subset of configurations (vertices on the left side) that covers all the feature interaction pairs (vertices on the right side). **(b)** Example: configurations C1, C2, and C3 together cover all the feature interaction pairs P1 to P8. By removing any of these configurations, maximum coverage cannot be achieved. Adding configurations C4 and C5 just increase the sample size as the pairs that they cover (P5, P6, and P7) have already been covered by C1 and C3.

Configuration sampling methods typically take as input a feature model and generate a list of configurations (i.e., a sample). Therefore, in the following, we provide the basic notion of feature models and configurations as well as configuration sampling problem.

Feature models. A *feature model* delineates all the elements of a SPL along with their interdependencies. Formally, it is defined as a tuple $(\mathcal{F}, \mathcal{D})$, consisting of a set of features $\mathcal{F}$ and a set of dependencies $\mathcal{D}$. In the model, every feature is represented by a distinct integer value, ranging from 1 to n (i.e., $\mathcal{F} = \{1, \ldots, n\}$), where n signifies the total number of features in the model. Dependencies within a feature model are expressed as clauses of a propositional formula in CNF. Each dependency in $\mathcal{D}$ corresponds to one such clause (i.e., $\mathcal{D} = D_1, \ldots, D_m$), with m denoting the total number of clauses. A clause is defined as a set of literals, where a literal can be either a number from $\mathcal{F}$ (i.e., a positive literal) or a negated number from $\mathcal{F}$ (i.e., a negative literal), representing the selection and deselection of a particular feature in a configuration, respectively. We define the function $\mathcal{L}$ that provides the set of literals for a feature set of a feature model, $\mathcal{L}(\mathcal{F}) = \{-n, \ldots, -1, 1, \ldots, n\}$.

Configurations. A *configuration*, C, is a selection of features from a feature model, and is formally defined as a set of literals, such that $C \subseteq \mathcal{L}(\mathcal{F})$ with $\forall l \in C : -l \notin C$. When a literal appears in a configuration, it determines the status of the corresponding feature as either selected (in the case of a positive literal) or deselected (in the case of a negative literal). A configuration is referred to as *complete* when it includes all features ($|C| = |\mathcal{F}|$); otherwise, it is considered *partial*. A configuration satisfies a clause in $\mathcal{D}$ if it includes at least one literal from that clause. On the other hand, if a configuration contains all the complementary literals of a clause, it contradicts the clause and, subsequently, the entire feature model. In cases where a configuration contradicts one or more clauses, it is considered *invalid*. A configuration is considered *valid* if it allows for the satisfaction of all clauses within a feature model: $\exists; C' \supseteq C : \forall D \in \mathcal{D} : C' \cap D \neq \emptyset$. A valid configuration can be partial and does not necessarily have to satisfy every clause. It is considered valid as long as the addition of more literals to the configuration enables the satisfaction of all clauses.

Simple t-wise Combination Coverage. For a given set of configurations with n features, simple t-wise combination coverage represents the fraction of t-wise combinations covered by these configurations. The result of a t-wise interaction sampling is a configuration sample $\mathcal{S}_\mathcal{C}$, which is a set of valid configurations (i.e., $\mathcal{S}_\mathcal{C} = \{C_1, C_2, \ldots\}$). In a complete t-wise interaction sample (i.e., 100% t-wise interaction coverage) every valid interaction (i.e., not contradicting the feature model) is a subset of at least one configuration. An interaction I is represented by a set of exactly t literals and is considered valid if the partial configuration containing only this set of literals is also valid. We define the set of all valid feature interaction pairs as $\mathcal{I} = \{I | I \subseteq \mathcal{F}, |I| = t, \forall f_i, f_j \in I, -f_i \notin I;\ i \neq j\}$.

Definition of configuration sampling problem. Let $\mathcal{F}$ denote a set consisting of n features, $\mathcal{S}_\mathcal{C}$ denote the set of all possible configurations involving these features, and $\mathcal{I}$ denote the set of all valid feature interaction pairs. The primary objective of configuration sampling is to identify a minimal subset of configurations, denoted by $\mathcal{S}^* \subseteq \mathcal{S}_\mathcal{C}$, which ensures coverage of all t-wise feature interactions $\mathcal{I}$ while minimizing the size of $\mathcal{S}^*$. More precisely, the objective is to guarantee the presence of every feature interaction pair $I \in \mathcal{I}$ in at least one configuration within S^*, while minimizing the size of S^*.

IV. Proposed Method

In this section, we present our method for the configuration sampling problem by first providing the Markov decision process (MDP) formulation and then describing our reinforcement learning approach for handling configuration sampling.

A. MDP formulation for configuration sampling

State space $\mathcal{S}$. A state represents the current *sample* which is defined as the current subset of valid configurations. Thus, state space $\mathcal{S}$ is defined as the power set of valid configurations of the feature model that are generated by t-wise sampling algorithm and given as input to the our method: $\mathcal{S} = P(\mathcal{S}_\mathcal{C}) = \{C \mid C \subseteq \mathcal{S}_\mathcal{C}\}$.
Initial state s_0. $s_0 \in \mathcal{S}$ can be defined as the empty set, or a given set of valid configurations. For the first episode, we consider a set of valid configurations generated by t-wise sampling algorithm [46]. For the next episodes, we first find the feature interaction pairs that have been covered the least in the previous episode and their corresponding covering configurations will be considered as the initial state (initial subset of configurations) for the next episode.
Action space $\mathcal{A}$. An action is defined as a two-element tuple where the first action dimension determines whether a configuration is added to, or removed from the current sample, and second action dimension corresponds to the number of the configuration which is added or removed:

$$\begin{aligned}\mathcal{A} &= \{(a_1, a_2) \mid a_1 \in \{1,2\}, a_2 \in \{1,2,3,\ldots,|\mathcal{S}_\mathcal{C}|\}\} \\ &= \{1,2\} \times \mathcal{S}_\mathcal{C}\end{aligned} \tag{8}$$

An action $a = (a_1, a_2)$ is only performed if it is *applicable* as shown in Algorithm 1. This implies that adding a new configuration is contingent upon covering a new feature interaction that has not already been covered by the existing configurations in the current sample (i.e., current state). Similarly, removing an existing configuration is subject to ensuring that the removal does not result in any new uncovered feature interaction pairs.
Transition function p. The state transition function defines the result of applying the action a within the state s (set of configurations), leading to a new successor state s' that can be either a new set of configurations where a configuration has been added/removed or the same state s if the action tuple a is not applicable in state s: $p : S \times A \to S$.
Reward function r. At each time-step t, if the action a is applicable within the state s, then the agent receives a positive reward +1, otherwise, a negative reward -1. The reward function is defined as follows:

$$r(a,s) = \begin{cases} 1, & \text{if } is\_applicable\_action(a,s) \\ -1, & \text{otherwise} \end{cases} \tag{9}$$

where the function $is\_applicable\_action$ is presented in Algorithm 1.

Terminal state. A state is a terminal state if the configurations within that state cover all the feature interaction pairs. In other words, we reach a terminal state when we achieve 100% coverage.

B. RL-based Approach to Configuration Sampling

State embedding. The first step is to embed the state information using an embedding function Φ. This process involves mapping the raw state information to a fixed low-dimensional vector representation with the size ζ that can capture the essential features of the state. In our case, we use an auto-encoder with hyperbolic tangent activation function as the embedding function. Therefore, given a set of configurations $\mathcal{S}_\mathcal{C}$, our embedding function is defined as $\Phi : \mathcal{S}_C \to [-1,1]^\zeta$.
Training branching dueling Q-network (BDQ). The training begins by feeding state (embedding of a subset of configurations) to the network. The shared representation module then extracts features from the input state. These extracted features are then decomposed into the state value and state-dependent action advantages for each independent branch. Each branch corresponds to a specific dimension of the action. The Q-values for each action dimension are computed by combining the state value and action advantages using a dedicated aggregation layer. To form joint-action tuples, the argmax function is applied to concatenate the sub-action branches.

We consider a two-dimensional action space $\mathcal{A}_d$ where $d \in \{1,2\}$ with $|\mathcal{A}_1| = 2$ discrete sub-actions for the first dimension and $|\mathcal{A}_2| = |\mathcal{S}_\mathcal{C}|$ discrete sub-actions for the second dimension. Action dimensions 1 and 2 correspond to the action type (add/remove) and the configuration number, respectively. The Q-value at state $s \in \mathcal{S}$ and sub-action $a_1 \in \mathcal{A}_d$ is calculated using Equation 10 as follows:

$$Q^1_\theta(s,a_1) = V(s) + \left(A_1(s,a_1) - \tfrac{1}{|\mathcal{A}_1|}\textstyle\sum_{a'_1 \in \mathcal{A}_1} A_1(s,a'_1)\right). \tag{10}$$

Similarly, the Q-value at state $s \in \mathcal{S}$ and sub-action $a_2 \in \mathcal{A}_d$ is calculated using Equation 11 as follows:

$$Q^2_\theta(s,a_2) = V(s) + \left(A_2(s,a_2) - \tfrac{1}{|\mathcal{A}_2|}\textstyle\sum_{a'_2 \in \mathcal{A}_2} A_2(s,a'_2)\right). \tag{11}$$

The temporal-difference (TD) target for BDQ agent is calculated as follows:

$$\begin{aligned} y_t^{BDQ} = r + \gamma\Bigg[&\frac{1}{2}Q^1_{\theta'}\left(s', \arg\max_{a'_1 \in \mathcal{A}_1} Q^1_\theta(s',a'_1)\right) \\ &+ \frac{1}{2}Q^2_{\theta'}\left(s', \arg\max_{a'_2 \in \mathcal{A}_2} Q^2_\theta(s',a'_2)\right)\Bigg] \end{aligned} \tag{12}$$

Finally, the loss function is defined as the expected value of the mean squared TD error across the branches:

$$L = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\tfrac{1}{2}\left(y_t^{BDQ} - Q^1_\theta(s,a_1)\right)^2 + \tfrac{1}{2}\left(y_t^{BDQ} - Q^2_\theta(s,a_2)\right)^2\right]. \tag{13}$$

The architecture of our RL-based method is shown in Figure 2.

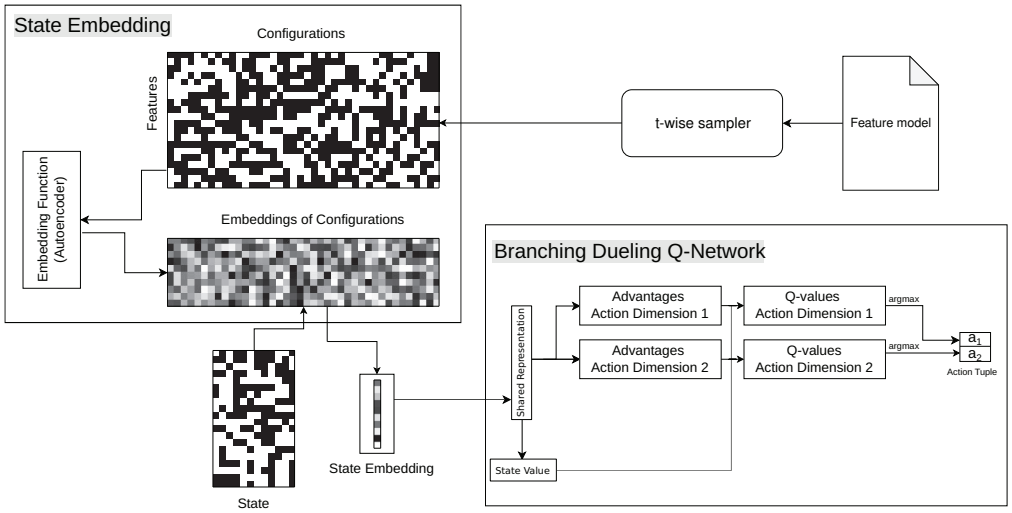

Figure 2. Architecture of our proposed method. For the given feature model, t-wise sampler is first used to generate a set of valid configurations. The generated configurations are then passed to the embedding function to learn embeddings of the configurations. The shared representation module then extracts features from the input state. These extracted features are then decomposed into the state value and state-dependent action advantages for each independent branch. Each branch corresponds to a specific dimension of the action. The Q-values for each action dimension are computed by combining the state value and action advantages using a dedicated aggregation layer. To form joint-action tuples, the argmax function is applied to concatenate the sub-action branches.

Algorithm 1 $is\_applicable\_action(a, s)$

Input: Action tuple $a = (a_1, a_2)$; state s.
Output: Action applicability at state s.
Initialize: $is\_applicable = False$; $action\_type = a_1$; configuration $c = a_2$
$up = get\_uncovered\_pairs(s)$
$cp = get\_pairs\_covered\_by\_configuration(c)$
if $action\_type$ == 1 **then**
    // adding the configuration c
    if $cp \cap up \neq \emptyset$ **then**
        $S.add(c)$
        $is\_applicable = True$
else if $action\_type$ == 2 **then**
    // removing the configuration c
    if $cp \cap up = \emptyset$ **then**
        $S.remove(c)$
        $is\_applicable = True$
return $is\_applicable$

RL-based configuration sampling. Algorithm 2 shows the pseudocode for our proposed RL-based method for configuration sampling. The algorithm takes as input a feature model $\mathcal{F}$, a t-wise sampler, the parameter t, a branching dueling Q-Network (BDQ) agent, the number of episodes, maximum time-steps, an embedding function Φ, an embedding size ζ, and a configuration sampling environment. It returns a minimal subset of configurations $\mathcal{S}^*$.

The algorithm starts by initializing an empty set, denoted as $\mathcal{S}^*$, to serve as the initial state. Subsequently, it generates an initial solution employing the t-wise coverage sampler, setting the $best\_sample\_size$ to the size of this initial solution. During the iteration of the algorithm through the specified number of episodes, the BDQ agent selects an action tuple at each time-step within an episode, and the environment performs a step using the action tuple if applicable, transitioning to a new state and receiving a positive reward. If the action tuple is not applicable, the agent remains in the same state and receives a negative reward. This information is recorded as a transition item, after which the agent is trained using the stored transitions. Subsequently, the next state is embedded using the embedding function, and the current state is updated to the next state. Upon the completion of an episode or reaching the maximum time-steps, the algorithm assesses whether the current state's size is smaller than the best sample size. If this condition is met, the best sample size and the subset of configurations, $\mathcal{S}_{\mathcal{F}}$, are updated. Ultimately, the algorithm returns the subset of configurations exhibiting 100% coverage, denoted as $\mathcal{S}_{\mathcal{F}}$.

C. Gym-based configuration sampling environment

Our Gym-based Configuration Sampling environment (CS-Gym) takes as input a feature model as a DIMCAS file or a CNF formula, and the parameter t and employs a t-wise sampler. CS-Gym is designed to be compatible with any deep reinforcement learning algorithm that supports multi-dimensional action spaces. It consists of four functions:

- **__init__:** This function initializes the state space and action space for the given feature model, generates an initial set of configurations using t-wise sampler, and generates all the possible feature interaction pairs.
- **step:** This function takes as input an action tuple and returns a transition item based on the applicability of the given action tuple within the current step.
- **reset:** This functions resets the environment and sets the initial state of next episode to the smallest subset

Algorithm 2 RL-based Algorithm for Configuration Sampling

Input: Feature model $\mathcal{F}$; t-wise coverage sampler $tw\_sampler$, parameter t, branching dueling Q-Network (BDQ) $agent$; number of episodes $n\_episodes$; maximum time-steps max_ts; embedding function Φ, embedding size ζ; configuration sampling environment env.
Output: A minimal subset of configurations $\mathcal{S}^*$.
Initialize: $\mathcal{S}^* = \varnothing$;
$initial\_solution = tw\_sampler(\mathcal{F}, t)$
$best\_sample\_size = |initial\_solution|$
$state\_embedding = \Phi(initial\_solution, \zeta)$
for i=1, ... $n\_episodes$ **do**
    $state, done = env.reset(), False$
    $state = state\_embedding(state)$
    for j=1, ..., max_ts **do**
        $action\_tuple = agent.act(state)$
        $next\_state, reward, done = env.step(action\_tuple)$
        $agent.store(state, action\_tuple, reward, next\_state, terminal)$
        $agent.train()$
        $next\_state = state\_embedding(next\_state)$
        $state = next\_state$
        if $done$ **or** $j \mod max\_ts == 0$ **then**
            if $|state| < best\_sample\_size$ **then**
                $best\_sample\_size = |state|$
                $\mathcal{S}^* = state$
            break
return $\mathcal{S}^*$

of configurations, so that the agent can more efficiently converge to the optimal solution (the minimal subset of configurations).

- **render:** This function reports relevant information about the behavior of the environment that has been collected so far, and visualizes the sample size and the coverage of the sample over the episodes in real-time.

V. Experimental Setup

In this section, we formulate the research questions, describe the baselines, and feature models used for our experiments.

Research questions. We consider the following three research questions (RQs):

- **RQ-1:** Given an initial solution generated by t-wise sampler to the RL-based sampler, how much does the RL-based sampler reduce the sample size?
- **RQ-2:** How does the RL-based sampler compare to state-of-the-art heuristic-based sampling algorithms in terms of sample size?
- **RQ-3:** How early does the RL-based sampler converge to the minimal sample size and how does it compare the minimal sample size found by the best heuristic method?

Baselines. We consider the following heuristic-based sampling algorithms as the baselines:

- Chvatal [47] is a greedy algorithm in which the combinations of features are generated to be considered during the sampling process. The configurations are added to the sample set in a greedy manner, and each added configuration should cover at least an uncovered combination.
- ICPL [48] is an algorithm for generating covering arrays for large-scale feature models. It is built on the Chvatal algorithm [47] with additional performance improvements.
- YASA [19] is a greedy sampling algorithm that starts with an empty sample and then iterates over all t-wise interactions one at a time. For each, either a new partial configuration with the features of the interaction is added to the sample or the features are added to an existing configuration. YASA enhances the basic algorithm by applying different heuristic and caching methods.

Table I
The feature models with the number of features and clauses/dependencies in their corresponding CNF formula.

Feature Model	# Features	# Clauses
BerkeleyDBC	18	29
Dune	17	16
JavaGC	39	105
JHipster	45	104
lrzip	20	63
Polly	40	100
VP9	42	104
X264	16	11

Feature models. We evaluate our approach using eight feature models from real-world systems, commonly used in recent works, that are listed in Table I and described below.

- **BerkeleyDB:** An embedded database library providing efficient low-level data management.
- **Dune:** A build system designed for OCaml/Reason projects, focusing on simplicity and fast build times.
- **JavaGC:** Garbage Collection system of the Java virtual machine, responsible for automatic memory management.
- **JHipster:** A platform for generating, developing, and deploying Spring Boot + Angular/React/Vue web applications and microservices.
- **lrzip:** A compression utility handling large files due to its ability to handle long-distance redundancies.
- **Polly:** A domain-specific language in LLVM for expressing high-level, optimizable loop structures.
- **VP9:** An open-free video coding format.
- **X264:** A free software library for encoding video streams into the H.264/MPEG-4 AVC compression format, known for its efficiency and quality.

Evaluation metrics. We consider the metrics *sample size* and *coverage* (i.e., the number of feature interaction pairs covered by configurations); however, we report only sample size as, in our case, each episode of environment ends when we achieve 100% coverage.

Evaluation of RL-based sampler For the training phase, all configurations generated by the t-wise sampling algorithm are considered at the initial state and the RL-based sampler finds a minimal subset of those configurations by progressively reducing the sample size. However, for the evaluation phase, we consider an empty set (empty sample) at the initial state

and use the learned policy to sample the next configurations based on the current configurations in the sample. For all the experiments, we consider t-values 2 and 3.

Parameter tuning. Hyperparameters of our method are selected by grid search and listed in Table II.

Table II
HYPERPARAMETERS OF OUR RL-BASED SAMPLER.

Parameter	Description	Value
n_episodes_training	Number of episodes for training	1000
n_episodes_evaluation	Number of episodes for evaluating	100
max_ts	Maximum number of time-steps	2000
gamma	Discount factor	0.99
optimizer	Optimizer	RMSprop
eta	Learning rate for the RMSprop optimizer	0.0005
lambda	Weight decay (L2 penalty) for the RMSprop optimizer	0.0001
buffer_type	Replay buffer	Prioritized
buffer_size_max	Replay buffer size	1M
buffer_min_size	Minimum size of the replay buffer before training begins	1000
alpha	Parameter determining how much prioritization is used	0.6
beta	Parameter representing the importance-sampling weight	0.1
beta_increase_steps	Number of steps over which beta is linearly increased to 1	50000
batch_size	Number of experiences to sample from the replay buffer	64
replays	Number of batches to train on after each step	1
shared_size	Size of the layers in the shared part of the network	(512, 512)
branch_size	Size of the layers in the branched part of the network	(128, 128)
activation_func	Activation function	Leaky ReLU
epsilon_start	Starting value of epsilon for the epsilon-greedy	1.0
epsilon_decay_steps	Number of steps over which epsilon is linearly decayed	20000
epsilon_min	Minimum value that epsilon can reach after decay	0.1
new_actions_prob	Probability of choosing a new random action at each step	0.05
tau	Parameter for soft update of target network parameters	0.01

Implementation. All the experiments were conducted on DGX-Station (NVIDIA DGX-1) under Ubuntu 22.10, and implemented in Python 3.10. The experiments for $t = 2$ and $t = 3$ took around 10 hours and 3 days, respectively. The code and datasets are available on GitHub[1].

VI. RESULTS

A. Reduction of sample size from the initial solution

To address the RQ-1, we compare the the sample sizes generated by a t-wise sampler and our RL-based sampler on eight feature models (BerkeleyDBC, Dune, JavaGC, JHipster, lrzip, Polly, VP9, and X264) and two different t-values: 2 and 3. The bar charts in Figure 3 show that the RL-based sampler consistently achieves a high percentage of reduction across all feature models and both t-values 2 (left subplot) and 3 (right subplot). Table III provides the percentage reduction for each feature model and t-value. The results suggest that the percentage reduction in sample size increases as the value of t increases for all feature models. More precisely, increasing the level of interaction between features, as measured by the value of t, can result in a greater reduction in sample size while maintaining the ability to detect faulty test cases. This is particularly beneficial because larger sample sizes can lead to increased costs in terms of the time and resources required to execute and analyze the tests.

B. Performance of RL-based Sampling Algorithm

To address the RQ-2, we compare the sample sizes achieved by the RL-based sampler and three state-of-the-art heuristic-based sampling algorithms for different feature models and

[1]https://amir-abolfazli.github.io/RLSampler/

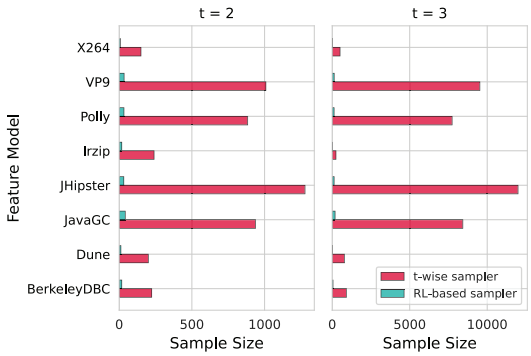

Figure 3. Reduction of sample size from initial solution generated by t-wise sampler compared to the sample size generated by RL-based sampler for the considered feature models.

Table III
PERCENTAGE REDUCTION IN SAMPLE SIZE FOR RL-BASED SAMPLER COMPARED TO t-WISE SAMPLER.

Feature Model	t = 2	t = 3
BerkeleyDBC	91.96	94.16
Dune	94.03	96.12
JavaGC	95.52	97.77
JHipster	97.49	98.95
lrzip	92.5	81.97
Polly	96.27	98.38
VP9	96.63	98.64
X264	93.33	94.68

t-values, and present the results in Table IV. We consider 1M time-steps (1K episodes, each with 10K time-steps). The percentage reduction in sample size compared to the best result among the other algorithms is also provided (in green parentheses). In all cases, the RL-based sampler achieves the smallest sample size among all the algorithms, demonstrating its effectiveness in reducing the sample size while achieving 100% t-wise coverage. Generally, the RL-based sampler achieves a higher percentage reduction in sample size when t=2 compared to t=3. In addition, the results indicate that the RL-based sampler is effective in reducing the required sample size for testing even as the number of pairs increases with higher values of t.

C. Convergence of RL-based sampler to minimal sample size

To address the RQ-3, we analyze the convergence behavior of our RL-based approach on eight feature models of software product lines and compare our results with the best heuristic-based sampler (i.e., the heuristic-based sampler with the smallest sample size reported in Table IV). We evaluate the RL-based sampler on 200K time-steps (100 episodes, each with 2K time-steps) and report the average sample size generated over 10 different runs with randomly chosen seeds.

In Figure 4, blue and orange lines correspond to the sample sizes generated by RL-based sampler with the t-values 2 and 3, respectively. Similarly, the black and brown dashed lines correspond the smallest sample sizes generated by the heuristic-based samplers, respectively. Figure 4 shows that our approach can rapidly converge to small sample sizes. However,

Table IV
SAMPLE SIZES OF OUR RL-BASED SAMPLER COMPARED TO HEURISTIC-BASED SAMPLING ALGORITHMS ON EIGHT FEATURE MODELS FOR TWO t-VALUES (2 AND 3). THE SMALLEST SIZE IS MARKED IN BOLD AND SECOND SMALLEST SAMPLE SIZE IS DENOTED BY (+). THE PERCENTAGE OF REDUCTION IN SAMPLE SIZE FOR OUR RL-BASED SAMPLER, COMPARED TO THE BEST HEURISTIC-BASED METHOD, IS SHOWN IS GREEN.

Feature Model	t	# Pairs	Sample Size			
			Chvatal	ICPL	YASA	RL-based Sampler
BerkeleyDBC	2	529	21	21	20 (+)	**18** (↓ 10%)
	3	5020	59 (+)	60	59 (+)	**54** (↓ 8%)
Dune	2	472	14	14	13 (+)	**12** (↓ 7%)
	3	4264	39	38	34 (+)	**31** (↓ 8%)
JavaGC	2	2399	52	53	48 (+)	**42** (↓ 14%)
	3	51457	216 (+)	217	217	**188** (↓ 13%)
JHipster	2	3151	38	40	37 (+)	**32** (↓ 13%)
	3	74032	127 (+)	129	127 (+)	**126** (↓ 0.78%)
lrzip	2	619	22	23	19 (+)	**18** (↓ 5%)
	3	5832	55	55	49 (+)	**44** (↓ 10%)
Polly	2	2402	39	42	38 (+)	**33** (↓ 13%)
	3	51618	184 (+)	187	189	**125** (↓ 32%)
VP9	2	2695	41	40	38 (+)	**34** (↓ 10%)
	3	61850	182	185	177 (+)	**130** (↓ 26%)
X264	2	387	15	15	12 (+)	**10** (↓ 20%)
	3	3155	34	30 (+)	31	**27** (↓ 10%)

the episode at which the minimal sample size is reached varies depending on the specific feature model of the SPL being tested. For the feature model BerkeleyDBC, RL-based sampler outperforms the heuristic-based methods with $t = 2$. Our method also significantly outperforms heuristic-based methods on the feature modes JavaGC, VP9, and Polly, with $t = 3$, and has competitive performance compared to heuristic-based methods on the feature models JavaGC, lrzip, VP9, JHipster, Polly, and X264, with $t = 2$, and similarly on feature models BerkeleyDBC, lrzip, Dune, JHipster, and X264, with $t = 3$.

VII. DISCUSSION AND FUTURE WORK

Applying deep reinforcement learning to the configuration sampling problem has shown promising results, having significant implications for cost reduction, as the reduced sample size leads to fewer configurations to test, and subsequently resulting in lower testing efforts and resource utilization. The findings open up opportunities for future research in configuration sampling. The heuristic methods typically identify the minimal sample size as the final output that is considered as the most efficient solution, where the smallest set of configurations fully covers all t-wise feature interactions. However, in an RL setting, in early iterations, the model might generate a smaller sample size, which could be an optimal solution. Nonetheless, due to the inherent exploratory nature of reinforcement learning, the model does not stop at this point. Instead, it continues to explore other configurations in the search space. Although this exploration phase is crucial for the robustness of the learning process, it often leads to larger sample sizes. It is important to note that there are some limitations that need to be addressed in future research. One limitation of our work is that the policy learned for a specific feature model may not be directly applied to a feature model with a different cardinality of the action space. This limitation implies that

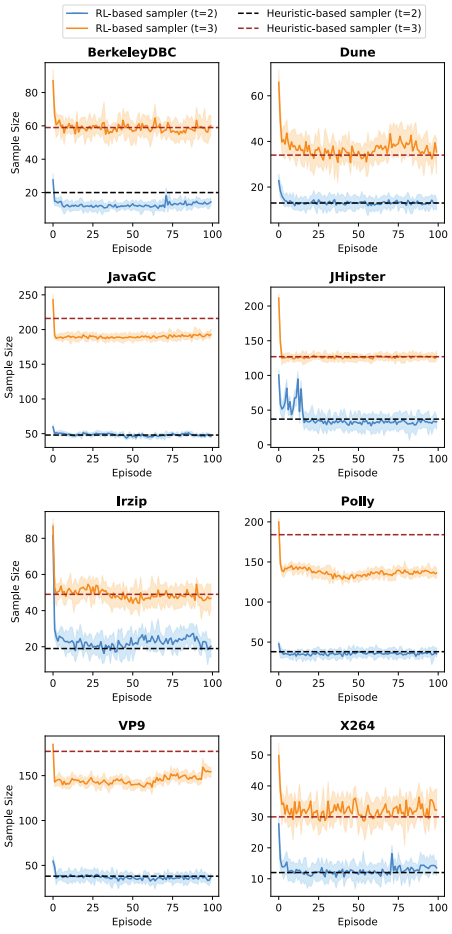

Figure 4. The sample size of our RL-based method over 100 episodes with 10 randomly chosen seeds for the considered feature models compared to the minimal sample size of the best heuristic-based sampler which is marked in black and brown dashed lines t=2 and t=3, respectively.

further investigations are necessary to extend the applicability of the DRL-based sampler to feature models with varying action space sizes. For future work, we aim to explore the use of multi-agent approaches for further optimization of the configuration sampling process.

VIII. CONCLUSION

In this work, we proposed a deep reinforcement learning based sampler that finds a minimal subset of configurations guaranteeing 100% coverage given an initial set of configurations generated by t-wise sampling algorithm. We also presented the CS-Gym, an environment for the configuration sampling problem. Our experimental results showed that the proposed method significantly outperforms the heuristic-based sampling methods on eight feature models of software product lines in terms of sample size. This improvement has substantial implications for cost reduction, as the reduced sample size leads to fewer configurations to test, resulting in lower testing efforts, resource utilization, and overall testing time.

ACKNOWLEDGMENT

The authors gratefully acknowledge that the proposed research is a result of the research project "QuBRA" granted by the BMBF via funding code 13N16052.

REFERENCES

[1] D. Batory, "Feature models, grammars, and propositional formulas," in *SPLC*. Springer, 2005, pp. 7–20.
[2] M. Fewster and D. Graham, *Software test automation*. Addison-Wesley Reading, 1999.
[3] D. R. Kuhn, R. N. Kacker, and Y. Lei, *Introduction to combinatorial testing*. CRC press, 2013.
[4] M. Al-Hajjaji, S. Krieter, T. Thüm, M. Lochau, and G. Saake, "Incling: efficient product-line testing using incremental pairwise sampling," *ACM SIGPLAN Notices*, vol. 52, no. 3, pp. 144–155, 2016.
[5] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, pp. 1–29, 2011.
[6] M. B. Cohen, M. B. Dwyer, and J. Shi, "Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach," *IEEE Trans. Softw.*, vol. 34, no. 5, pp. 633–650, 2008.
[7] I. Abal, J. Melo, Ş. Stănciulescu, C. Brabrand, M. Ribeiro, and A. Wąsowski, "Variability bugs in highly configurable systems: A qualitative analysis," *TOSEM*, vol. 26, no. 3, pp. 1–34, 2018.
[8] M. Al-Hajjaji, J. Meinicke, S. Krieter, R. Schröter, T. Thüm, T. Leich, and G. Saake, "Tool demo: testing configurable systems with featureide," in *GPCE*, 2016, pp. 173–177.
[9] P. Arcaini, A. Gargantini, and P. Vavassori, "Generating tests for detecting faults in feature models," in *ICST*. IEEE, 2015, pp. 1–10.
[10] K. Kitsawad and N. Tuntisripreecha, "Sensory characterization of instant tom yum soup," *Applied Science and Engineering Progress*, vol. 9, no. 2, pp. 145–152, 2016.
[11] H. Eichelberger and K. Schmid, "A systematic analysis of textual variability modeling languages," in *Proceedings of the SPLC*, 2013, pp. 12–21.
[12] C. Henard, M. Papadakis, and Y. Le Traon, "Mutation-based generation of software product line test configurations," in *SBSE*. Springer, 2014, pp. 92–106.
[13] C. Henard, M. Papadakis, G. Perrouin, J. Klein, P. Heymans, and Y. Le Traon, "Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines," *IEEE Trans. Softw.*, vol. 40, no. 7, pp. 650–670, 2014.
[14] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans, "Covering spl behaviour with sampled configurations: An initial assessment," in *VaMoS*, 2015, pp. 59–66.
[15] C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon, "Multi-objective test generation for software product lines," in *SPLC*, 2013, pp. 62–71.
[16] R. A. Matnei Filho and S. R. Vergilio, "A multi-objective test data generation approach for mutation testing of feature models," *JSERD*, vol. 4, pp. 1–29, 2016.
[17] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann, "Static analysis of variability in system software," in *USENIX ATC*, 2014, pp. 421–432.
[18] E. Baranov, A. Legay, and K. S. Meel, "Baital: an adaptive weighted sampling approach for improved t-wise coverage," in *ESEC/FSE*, 2020, pp. 1114–1126.
[19] S. Krieter, T. Thüm, S. Schulze, G. Saake, and T. Leich, "Yasa: yet another sampling algorithm," in *VaMoS*, 2020, pp. 1–10.
[20] D. Marijan, A. Gotlieb, S. Sen, and A. Hervieu, "Practical pairwise testing for software product lines," in *SPLC*, 2013, pp. 227–235.
[21] J. Oh, P. Gazzillo, and D. Batory, "t-wise coverage by uniform sampling," in *SPLC*, 2019, pp. 84–87.
[22] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," in *AAAI*, vol. 32, 2018.
[23] M. Varshosaz, M. Al-Hajjaji, T. Thüm, T. Runge, M. R. Mousavi, and I. Schaefer, "A classification of product sampling for software product lines," in *SPLC*, 2018, pp. 1–13.
[24] Q. Cappart, T. Moisan, L.-M. Rousseau, I. Prémont-Schwarz, and A. A. Cire, "Combining reinforcement learning and constraint programming for combinatorial optimization," in *AAAI*, vol. 35, no. 5, 2021, pp. 3677–3687.
[25] H. Lu, X. Zhang, and S. Yang, "A learning-based iterative method for solving vehicle routing problems," in *ICLR*, 2020.
[26] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, "Solving np-hard problems on graphs with extended alphago zero," *arXiv preprint arXiv:1905.11623*, 2019.
[27] T. Barrett, W. Clements, J. Foerster, and A. Lvovsky, "Exploratory combinatorial optimization with reinforcement learning," in *AAAI*, vol. 34, no. 04, 2020, pp. 3243–3250.
[28] S. Gu and Y. Yang, "A deep learning algorithm for the max-cut problem based on pointer network structure with supervised learning and reinforcement learning strategies," *Mathematics*, vol. 8, no. 2, p. 298, 2020.
[29] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," *NeurIPS*, vol. 30, 2017.
[30] Q. Cai, W. Hang, A. Mirhoseini, G. Tucker, J. Wang, and W. Wei, "Reinforcement learning driven heuristic optimization," *arXiv preprint arXiv:1906.06639*, 2019.
[31] L. Duan, H. Hu, Y. Qian, Y. Gong, X. Zhang, Y. Xu, and J. Wei, "A multi-task selected learning approach for solving 3d flexible bin packing problem," *arXiv preprint arXiv:1804.06896*, 2018.
[32] D. Li, C. Ren, Z. Gu, Y. Wang, and F. Lau, "Solving packing problems by conditional query learning," 2019.
[33] V. Kurin, S. Godil, S. Whiteson, and B. Catanzaro, "Can q-learning with graph networks learn a generalizable branching heuristic for a sat solver?" *Advances in NeurIPS*, vol. 33, pp. 9608–9621, 2020.
[34] S. Manchanda, A. Mittal, A. Dhawan, S. Medya, S. Ranu, and A. Singh, "Learning heuristics over large graphs via deep reinforcement learning," *arXiv preprint arXiv:1903.03332*, 2019.
[35] Q. Cappart, E. Goutierre, D. Bergman, and L.-M. Rousseau, "Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning," in *AAAI*, vol. 33, no. 01, 2019, pp. 1443–1451.
[36] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt, T. Mann, T. Weber, T. Degris, and B. Coppin, "Deep reinforcement learning in large discrete action spaces," *arXiv preprint arXiv:1512.07679*, 2015.
[37] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *PloS one*, vol. 12, no. 4, p. e0172395, 2017.
[38] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.
[39] J. Xiong, Q. Wang, Z. Yang, P. Sun, L. Han, Y. Zheng, H. Fu, T. Zhang, J. Liu, and H. Liu, "Parametrized deep q-networks learning: Reinforcement learning with discrete-continuous hybrid action space," *arXiv preprint arXiv:1810.06394*, 2018.
[40] J. Zhang, J. Li, Y. Zhang, Q. Wu, X. Wu, F. Shu, S. Jin, and W. Chen, "Collaborative intelligent reflecting surface networks with multi-agent reinforcement learning," *IEEE J-STSP*, vol. 16, no. 3, pp. 532–545, 2022.
[41] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.
[42] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.
[43] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.
[44] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *ICML*. PMLR, 2016, pp. 1995–2003.
[45] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 2010.
[46] Y. Lei, R. Kacker, D. R. Kuhn, V. Okun, and J. Lawrence, "Ipog: A general strategy for t-way software testing," in *ECBS*. IEEE, 2007, pp. 549–556.
[47] M. F. Johansen, Ø. Haugen, and F. Fleurey, "Properties of realistic feature models make combinatorial testing of product lines feasible," in *MODELS*. Springer, 2011, pp. 638–652.
[48] M. F. Johansen, O. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *SPLC*, 2012, pp. 46–55.