IFRIT: Focused Testing through Deep Reinforcement Learning

1st Andrea Romdhana
DIBRIS, University of Genoa
Genoa, Italy
Security & Trust Unit, FBK-ICT
Trento, Italy

2nd Mariano Ceccato
University of Verona
Verona, Italy

3rd Alessio Merlo
DIBRIS
University of Genoa
Genoa, Italy

4th Paolo Tonella
Software Institute
Università della Svizzera italiana
Lugano, Switzerland

***Abstract*—Software is constantly changing as developers add new features or make changes. This directly impacts the effectiveness of the test suite associated with that software, especially when the new modifications are in an area where no test case exists. This article addresses the issue of developing a high-quality test suite to repeatedly cover a given point in a program, with the ultimate goal of exposing faults affecting the given program point. Our approach, IFRIT, uses Deep Reinforcement Learning to generate diverse inputs while keeping a high level of reachability of the desired program point. IFRIT achieves better results than state-of-the-art and baseline tools, improving reachability, diversity and fault detection.**

***Index Terms*—reinforcement learning, software testing, focused testing**

I. Introduction

Many software projects are under constant development and new versions are released continuously. These modifications may introduce new code or may alter the execution flow inside the existing code. Therefore, existing test suites may not adequately cover the new/modified code. As a result, it is important to automate the creation of test suites focusing on specific program sections, to ensure that such new/modified code sections do not introduce bugs into the system. Although monitoring the level of code coverage is a highly recommended practice, coverage strategies alone have limited capabilities in detecting real faults. Covering a faulty statement once does not guarantee that the fault will be activated, i.e., the program state will be infected and will propagate to an observable output [1]–[3]. In addition to coverage, another key goal of test suite creation should be diversity, because diversity can increase the chances of fault exposure [4].

Many automated test generation techniques at the state-of-the-art rely on coverage to detect bugs in programs [2], [5], [6]. However, these strategies do not take diversity into account and do not focus on specific code sections. To address this problem, Menéndez et al. [7] proposed to focus the testing process on the code sections added/modified by the developers and to generate a diversified set of uniformly distributed test cases that exercise such sections. This methodology, known as *focused testing*, ensures that multiple, diverse tests (i.e., a *focused test suite*) exercise a few specific program elements. Focused testing is implemented in the DFT tool [7], which aims at satisfying two objectives at the same time: (1) maximum diversity and (2) satisfiability of the conditions required to reach the target. However, DFT uses a Satisfiability Modulo Theory (SMT) solver to generate valid inputs for the program under test. The drawback of such an approach is that it loses efficiency when dealing with an increasing number of program constraints and it becomes inapplicable when constraints are too difficult to solve or even to formulate.

We propose a different approach to focused testing, which we call IFRIT, based on Deep Reinforcement Learning. Deep Reinforcement Learning (Deep RL) is a machine learning technique that does not require a labeled training set as input, since the learning process is guided by the positive or negative reward experienced during the tentative execution of a task. Hence, it can be used to dynamically learn how to build a focused test suite, based on the feedback obtained during the past successful or unsuccessful attempts. More specifically, IFRIT manipulates a test input by applying a sequence of modifiers (actions) to it. Each action receives positive feedback if the target statements are reached upon execution of the test input and such input was never generated before (to promote diversity); neutral (zero) feedback if the target statements get executed, but the input is not new; negative feedback if the input does not reach the target.

The key algorithmic advantage of IFRIT over DFT is that it requires only runtime coverage information from the subject under test. On the contrary, DFT requires that the subject under test can undergo static symbolic execution and that the generated path constraints, once relaxed by means of parametrisation, can be solved by an SMT tool. The requirements of DFT limit its practical applicability to simple numeric functions only. On the contrary, IFRIT requires minimal runtime information on the coverage of the target statements, used to provide feedback to the RL agent during training. Hence, it is generally applicable to any, arbitrarily complex, software system.

In terms of reachability and diversity, our empirical results show that IFRIT is equally effective as or more effective than DFT when executed on subjects to which DFT is applicable (i.e., the benchmark used in the original paper [7] that proposed DFT). Results show also that IFRIT is applicable to programs that cannot be handled by DFT. On them, IFRIT

outperforms the only available baseline, which is random test input generation.

Our paper gives the following major contributions to the state of the art:

- The first Deep RL approach to focused testing. By relying just on runtime coverage feedback, this approach is applicable to a wide range of programs.
- IFRIT, an open source tool, whose code is available at the url: https://github.com/H2SO4T/IFRIT .
- An empirical study showing the effectiveness of our approach in comparison with existing and baseline techniques.

II. Related Work

Most of the related work deals with the automated generation of focused test suites [7], [8]. Other relevant work concerns the general and wide class of automated test input generators [5].

A. Focused Testing

Focused testing aims at testing specific, individual components of a program. In the work by Alipour et al. [8], the authors define focused testing as a black-box method whose aim is to reach a specific target. A specific API can be an example of a target. Alipour et al. use a general test generator to reach that target by activating or deactivating different options of the generator.

Menéndez et al. [7] adopt a white-box approach and increase the granularity. The components addressed by focused testing are said to be *program points (pp)* (i.e., specific nodes in the control flow graph). Instead of using general-purpose generators, Menéndez et al. use a generator based on SMT solvers. Their tool, i.e., DFT, does not generate inputs for the real program, but for its symbolic path abstraction. This introduces some assumptions on the possibility to derive and solve precise path conditions for the components targeted by focused testing. Moreover, the speed of test suite generation might be affected negatively when the path constraints grow in size and complexity.

Instead of using random generators or SMT solvers, IFRIT is based on Deep RL, a technique that requires limited guidance during the training process – in our case, coverage and diversity information. Hence, its applicability is wider than DFT, and its effectiveness/efficiency do not depend on the accuracy/performance of any solver.

B. Automated Test Input Generation

The two main families of test input generators make use respectively of static/dynamic symbolic execution [5], [9], [10] and search based algorithms [11]–[13]. Similarly to DFT [7], techniques based on symbolic execution encode the path constraints that must be satisfied to traverse a given path in a formula that can be passed to an SMT solver. Search based algorithms rely instead on a fitness function that measures heuristically the distance between the path traversed when executing a candidate input and the coverage target. Inputs that get closer to the target are selected and iteratively improved until the target is covered. Both families of approaches succeed when the target is covered, but none of them attempts to generate multiple, uniformly distributed inputs that reach the target.

Differently from symbolic execution and search based test generation, random test input generation [14], [15] ensures uniformity of the generated inputs by construction. However, when the target to be covered requires that very specific path conditions are satisfied, this approach has a very low probability of generating inputs whose execution can actually traverse the target. So, despite the intrinsic high uniformity of the generated inputs, this approach is often ineffective because of the low reachability of targets that are difficult to cover randomly.

Only a few works [16]–[18] include diversity among the goals of the test generator, but none of them in the context of focused testing. Alshahwan et al. [16] maximize the diversity of the distribution of the outputs produced upon the execution of the automatically generated test inputs. So, they consider output, instead of input diversity. Feldt et al. [17] propose a new diversity metric, called test suite diameter, to quantify the degree of diversity in a test suite, but they do not use it directly for (focused) test generation. Biagiola et al. [18] use diversity as a criterion to select the most promising candidates, because in-browser web test execution is computationally expensive and diversity ensures wider exploration of behaviors. However their goal is not the generation of uniformly distributed inputs that reach a target of interest.

III. Problem Definition

Given the space of the program inputs X, we denote by X_{pp} the sub-space of inputs whose execution traverses pp. A focused test input generator aims at producing inputs x that belong to X_{pp}. In addition to this, the generator should produce a *diverse* set of inputs that belongs to X_{pp}.

Following the work by Chakraborty et al. [19], we define diversity using entropy. A diverse set is a set with high entropy. We define a generator G as an algorithm that creates inputs for a program P. We define a focused generator G_{pp} as a generator that generates inputs that traverse a specific program point pp. Considering the generator as a random variable whose values are inputs traversing pp, our goal is to make them as much diverse as possible, i.e., we aim at creating a focused generator G_{pp} whose entropy is maximized.

Since entropy of a random variable is maximum when its probability distribution is uniform [20], G_{pp} should be a uniform random variable and the generator should be a *uniform focused generator*, i.e., a generator that gives the same generation probability to every input $x \in X_{pp}$.

We need to quantify how close IFRIT is to generating samples from a uniform distribution to measure the uniformity. The work in [21] reports different statistical tests to measure it. These tests are divided into several categories: order statistics, spacing, order spacing, and collisions. Some of them do not apply to discrete distributions [21], while others do not manage

gaps in the domain [22]. Tests based on the collisions can deal with gaps and discrete distributions [23]. Consistently with the recommendation made by the authors of DFT [7], as a practical way to measure the degree of uniformity of a sample of values generated for a given random variable G_{pp} we use the $L2$ test [24]. The idea behind this test is that collisions (i.e., identical variable values) observed in the sample should be less than those that an ϵ-far uniform distribution defined on the same domain would generate, where ϵ is a user defined tolerance. When this happens, the $L2$ test is passed; otherwise it fails. The $L2$ test relies on the following formulas:

$$c = \sum_{x \neq y, (x,y) \in S \times S} \delta_{x,y}/2 \quad (1)$$

$$\theta = \binom{|S|}{2} \frac{1 + 3\epsilon^2/4}{n} \quad (2)$$

where S is a sample generated from G_{pp}; c counts the number of collisions in S using Kronecker delta $\delta_{x,y}$; ϵ defines the acceptable tolerance, i.e., the maximum allowed distance from the uniform probability distribution; n is the domain size. When $c < \theta$, the test is passed; otherwise it fails.

IV. Focused Testing based on Deep RL

In this section, we introduce the key concepts behind Reinforcement Learning. Moreover, we present IFRIT, a focused testing approach based on Deep Reinforcement Learning.

A. Reinforcement Learning in a nutshell

The objective of Reinforcement Learning [25] is to train an *agent* that interacts with an environment to achieve a given *goal*. The agent is assumed to be capable of sensing the *current state* of the *environment*, and to receive a feedback signal, named *reward*, each time the agent takes an *action*.

At each time step *t*, the agent receives an observation x_t and takes an action a_t that causes the transition of the environment from state s_t to state s_{t+1}. The agent also receives a scalar reward $R(x_t, a_t, x_{t+1})$, that quantifies the goodness of the last transition.

For simplicity, in the following we assume $x_t = s_t$ (in the general case, x_t might be a subset of s_t). The behavior of an agent is represented by a *policy* π, i.e., a rule for making the decision on what action to take, based on the perceived state s_t. A policy can be: 1) Deterministic $a_t = \pi(s_t)$, i.e. a direct mapping between states and actions; 2) Stochastic $\pi(a_t|s_t)$, a probability distribution over actions, given their state. DDPG [26] is an example of Deep RL algorithm that learns a deterministic policy, while PPO [27] is a Deep RL algorithm that learns a stochastic policy.

The standard mathematical formalism used to describe the agent environment is a *Markov Decision Process (MDP)*. An MDP is a 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$, where :

- S is the set of all valid states,
- A is the set of all valid actions,
- $R : S \times A \rightarrow \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1})$,
- $P : S \times A \rightarrow P(s)$ is the transition probability function, with $P(s_{t+1}|s_t, a_t)$ being the probability of transitioning into state s_{t+1} starting from state s_t and taking action a_t,
- $\rho_0(s)$ is the starting state distribution.

Markov Decision Processes obey the *Markov property*: a transition only depends on the most recent state and action (and not on states/actions that precede the most recent ones).

The goal in RL is to learn a policy π which maximizes the so-called *expected return*, which can be computed as:

$$R(\tau) = \sum_{t=0}^{|\tau|/2} \gamma^t r_t$$

A discount factor $\gamma \in (0, 1)$ is needed for convergence. It determines how much the agent cares about rewards in the distant future relative to those in the immediate future. τ is a sequence of states and actions in the environment $\tau = (a_0, s_0, a_1, s_1 ...)$, named *trajectory* or *episode*.

To maximize the overall expected return in practice, it is convenient to express it using an *action-value function*, which estimates the contribution that a single pair action-state gives toward maximizing the total expected reward.

B. Tabular RL & Deep RL

Tabular techniques refer to RL algorithms where the state and action spaces are approximated using an action-value function stored in a table. *Q-Learning* [28] is one of the most adopted algorithms of this family.

In large or unbounded discrete spaces, where representing all states and actions in a table is impractical, tabular methods become highly unstable and incapable to learn a successful policy [29]. The rise of deep learning, relying on the powerful function approximation properties of deep neural networks, has provided new tools to overcome these limitations. One of the first deep reinforcement learning algorithms is DQN (Deep Q-Networks) [29].

DQN uses convolutional neural networks to approximate the computation of the action-value function. Training of such neural networks is achieved by means of *memory replay*: the last N experience tuples are sampled uniformly and replayed when updating the weights of the networks.

While DQN can indeed solve problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. Among the recent advancements over DQN, PPO (Proximal Policy Optimization) [27] overcomes the latter limitation and can deal with high-dimensional action spaces.

RL algorithms can update a policy in two ways:

- Off-policy optimization: each update of the policy can use data collected at any point during training, regardless of how the agent has chosen to explore the environment when the data was obtained.
- On-policy optimization: each policy update only uses data collected while acting according to the most recent version of the policy.

DQN updates the policy in an off-policy way, while PPO is an on-policy algorithm.

C. Overview

IFRIT (reInFoRcement learnIng for focused Testing) is an approach to focused testing based on Deep RL. Figure 1 shows an overview of IFRIT. The RL *environment* is represented by a program P under test, which is subject to several interaction steps. The objective is to generate inputs that reach a program point $pp \in P$ (see Section II). At each time step, IFRIT observes the code coverage measured on the program, computes the state s_t, the reward r_t, and chooses an action a_t used to generate a new input for the program. Then, it iterates, receiving the next code coverage s_{t+1} and reward r_{t+1} (not shown in Figure 1).

Intuitively, if the new state s_{t+1} reaches the target point in the program with a new input, the reward is positive; it is neutral if such input is not new. Otherwise, if the target is not reached, the reward is negative.

The reward is used to update the neural network, which learns how to guide the Deep RL algorithm to generate useful inputs for the program. The actual update strategy depends on the selected Deep RL algorithm.

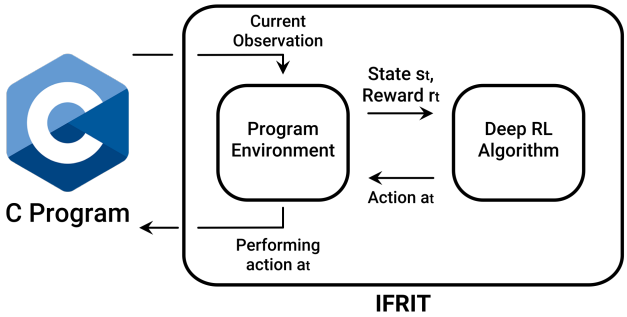

Fig. 1. The IFRIT testing workflow. Code coverage is extracted (e.g., by gcovr, a utility for generating summarized code coverage results), from which IFRIT generates the state s_t and then determines the reward r_t for the chosen action a_t. By choosing an action a_t, IFRIT generates a new input, that stimulates the program under test.

D. Instantiating RL for Focused Testing

To apply RL, we have to map the problem of focused testing to the standard mathematical formalization of RL: an MDP, defined by the 5-tuple, $\langle S, A, R, P, \rho_0 \rangle$. Moreover, we have to map the testing problem onto an RL task divided into several finite-length episodes.

State Representation: The state $s_t \in S$ is defined as a combined state $(b_0, ...b_n, i_0, ...i_m)$. The first part of the state $b_0, ...b_n$ represents the frequency of branch coverage during the last program execution, i.e., b_i is equal to $k > 0$ if the i-th branch of the program has been taken k times; it is equal to 0 if it was not traversed at all. The second part of the state vector, $i_0, ...i_m$ is equal to the last vector of input values generated by IFRIT. This means that the last execution of program P was performed by calling $P(i_0, ..., i_m)$, where $\langle i_0, ...i_m \rangle$ is called the *input vector*.

Action Representation: Each time IFRIT takes an action, it manipulates a previously generated input (e.g., a number or a string) by using modifiers. Modifiers mutate an input value based on the type of such input. Hence, an action $a = \langle a_0, a_1, a_2 \rangle$ is 3-dimensional: the first component a_0 encodes the index of the input vector to be manipulated. In fact, in the general case a program accepts a vector of input values as input and an action a will manipulate only one of them. The second component a_1 specifies the data to use to manipulate the input, depending on the context. The third component a_2 specifies how to use the second component on the input, i.e., what operation to apply using the second component as a parameter for such operation.

Numeric Input Manipulation. When managing numerical inputs, the starting input vector contains only zeros. The mutation of numeric input is done by using scale factors and operands. The first component (`i`) of the action indicates which portion of the input vector to modify (`input[i]`). The second component (`scale_factor`) indicates the scale factor, and the third specifies the operation (`<op>`) associated with that scale factor: `input[i] <op> scale_factor`, where `<op>` can be any arithmetic operator among +, -, *, /.

String Input Manipulation. When managing string inputs, the starting input is a vector of empty strings. IFRIT mutates the string by iteratively adding or removing characters to/from the input. The first component of the action indicates which portion of the input vector to modify. The second component indicates which char to use, and the third specifies the operation to perform (i.e., add char at the beginning, append char at the end, and remove char at the beginning/end). In the case of remove the second component is not used.

Transition Probability Function: The transition function determines which state the program can go to after IFRIT has taken an action. In our case, this is decided solely by the execution of the program: IFRIT observes the process passively, collecting the new state after the execution of the program has taken place.

Reward Function: The RL algorithm used by IFRIT receives a reward $r_t \in R$ every time it executes an action a_t. We define the following reward function:

$$r_t = \begin{cases} \Gamma_1 & \text{if } input(s_{t+1}) \notin inputs(E_j) \ \wedge \\ & x \in X_{pp} \\ \Gamma_0 & \text{if } input(s_{t+1}) \in inputs(E_j) \ \wedge \\ & x \in X_{pp} \\ -\Gamma_1 & x \notin X_{pp}. \end{cases} \quad (3)$$

with E_j the current episode and $\Gamma_1 > \Gamma_0 \geq 0$ (in our implementation $\Gamma_0 = 0$, $\Gamma_1 = 1$). This structure of the reward function and the chosen reward values are widely used in the literature in several different contexts [29] [26].

The exploration of IFRIT is divided into *episodes*. At time t, the reward r_t is positive (Γ_1) if IFRIT was able to reach the selected program point pp with an input never generated during the current episode E_j (i.e., the current input does not

belong to the set of inputs generated so far in E_j): if a new episode E_{j+1} is started at $t+1$, its set of inputs is reset: $inputs(E_{j+1}) = \emptyset$.

When an input that reaches the target has already been generated before in the same episode, the reward is zero (Γ_0), as it is no more useful during the current episode, but it remains a good input for the given task.

Resetting the set of generated inputs at the beginning of each new episode is a technique that encourages IFRIT to generate a high number of different inputs in each episode, which in turn makes the reward positive several times in the episode. In contrast, if we provide the algorithm a significant, positive reward only a few times (i.e., "sparsely"), e.g., because we have seen all new inputs already in previous episodes, the information to learn the optimal state-action combinations might be insufficient. The algorithm might fail to reproduce the sequence of actions leading to a high reward in the future and the performance of the algorithm results to be poor. On the contrary, another pattern to avoid is rewarding every successful input, regardless of its novelty, as this would encourage cycling behaviors [30]. Our definition of the reward function tries to balance the frequency of positive rewards and the avoidance of cycling behaviors, by rewarding positively inputs that are novel just in the current episode, not across all past episodes. Reward is negative ($-\Gamma_1$) when the input does not reach target (i.e., the selected program point pp).

V. Implementation

IFRIT features a custom environment based on the OpenAI Gym [31] interface, which is a de-facto standard in the RL field. OpenAI Gym is a toolkit for designing and comparing RL algorithms and includes several built-in environments. It also contains guidelines for the definition of custom environments. Our custom environment interacts with a C program.

A. Tool Overview

As soon as it is launched, IFRIT automatically generates a configuration file that contains information about the C program under test. The configuration file includes several data useful for compilation and the execution of the program (e.g., how many parameters the program takes as input, the type of each input, the target file, the target program point pp). Afterward, IFRIT instantiates a custom environment compatible with the C program and starts the testing phase. At each time step, IFRIT takes an action (i.e., modifies the input vector) according to the behavior of the exploration algorithm. Once the action has been fully processed, which includes the execution of the C program, IFRIT elaborates the code coverage information, from which IFRIT computes the observation and the reward for the algorithm.

IFRIT organizes the whole testing session into finite-length episodes. The goal of IFRIT is to maximize the total reward received during each episode. Every episode lasts 24000 time steps. To select the ideal episode boundaries, we conducted a preliminary experiment on a subset of programs coming from CodeFlaws (CF) [32]. On this subset, we trained the same Deep RL algorithm by varying the episode length. Training characterized by short episodes results in poor performance due to the impossibility of exploring the input space. Similarly, long episodes took too much training time. Once an episode comes to an end, IFRIT resets the input vector to the default value and then it uses the acquired knowledge to reach the target of the C program in the next episode. Figure 2 shows an example of numeric input manipulation. IFRIT generates an action that contains the rules to modify the previous input vector (a). We select the parameter of (a) indicated by the action and add to it the third scale factor (i.e., three). The new input vector (b) is now used to stimulate the program under test (not shown in the figure).

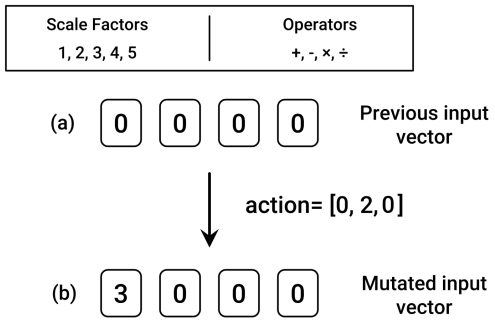

Fig. 2. The IFRIT action vector $[0, 2, 1]$ indicates that the input element at index 0 should be modified by applying the operator at index 1 (i.e., '+') with the scale factor at index 2 (i.e., '3'). The new input element at index 0 becomes thus equal to 3 (i.e., 0+3).

B. Program Environment

The application environment is responsible for handling the actions to interact with the program. Since the environment follows the guidelines of the Gym interface, it is structured as a class with two key functions. The first function `init(configuration_file)` is the initialization of the class. The additional parameter `configuration_file` consists of a dictionary containing the program tester setup and the program to be tested. The second function is the `step(a)` function, that takes an action `a` as input and returns a list of objects, including `observation` (code coverage state) and `reward`.

C. Algorithm Implementation

IFRIT exploits *Stable Baselines* [33], a modular library that adopts a plugin architecture to integrate the RL algorithm to use. In the current implementation, IFRIT provides two exploration strategies: (1) random, (2) PPO. The random algorithm interacts with the program by randomly selecting a mutation to perform on the input vector. Currently, one Deep RL algorithm is available in IFRIT: PPO. Its implementation comes from the Python library Stable Baselines. IFRIT is publicly available as open source software at the url: https://github.com/H2SO4T/IFRIT.

VI. Evaluation

We seek to address the following research questions:

- **RQ0** *Which scale factors provide the best configuration of IFRIT?* Selecting proper scale factors is essential to maximize the performance in terms of reachability and uniformity of IFRIT. To determine the optimal scale factors to use, we conducted a preliminary experiment.
- **RQ1** *What proportion of inputs generated by IFRIT reaches the target point of the program under test and how uniform are such inputs?* To evaluate the usefulness of IFRIT, we analyzed two key aspects: uniformity and reachability.
- **RQ2** *How do the test suites generated by IFRIT compare to those generated by the state-of-the-art tool DFT and random generators in terms of uniformity and reachability? What is the time required to produce them?* We aim to evaluate reachability and diversity of IFRIT in comparison with three baselines, namely, a pseudo-random generator (Random), Random Mutations, and DFT. To the best of our knowledge, DFT is the only test input generator that features focused testing, producing a diversified set of test inputs capable of reaching the target. Random performs random uniform sampling of the input domain. It has the highest possible (100%) uniformity level by construction, but it might produce only a few or no inputs that reach the target. Random Mutations performs uniform selection of the input mutation to apply to a previously generated input. This means that it uses the same mutation operators as IFRIT, but it applies them randomly instead of learning how to apply by means of RL.
- **RQ3** *What is the quality of the test suites generated by IFRIT with respect to the state-of-the-art, in terms of mutation killing capability and fault detection?* Fault detection is the ultimate goal of focused testing. We evaluate IFRIT's fault detection by considering both mutants and real faults. Mutation testing introduces small syntactic alterations to the program (i.e., mutants). It measures the ability of a test suite to detect the errors caused by these alterations, i.e., the ability to kill the mutants. We aim to evaluate the ability of IFRIT to detect these artificial faults, in comparison with three baselines, Random, Random Mutations, and DFT. We also evaluate fault detection on real faults.

A. Evaluation Design

To evaluate the proposed approach, we used the software subjects coming from two open-source software repositories: the Software-artifact Infrastructure Repository (SIR) [34], and CodeFlaws (CF) [32]. SIR contains C programs that accept numeric or string inputs. From the SIR repository, we used `tcas` as a numeric program. It is used to avoid collisions in aircraft systems. The `tcas` program has 135 LoC and 40 branch statements. As programs that accept string input, we used `printokens`, `printokens2`, `flex`, `gzip`, `grep`. The programs have from 570 to 10459 LoC and have 45 to 1065 branch statements. The CF repository includes 7,436 C programs in total. Each of them has on average 50 LoC. Among these, around 3900 programs have both a buggy and a fixed version. We selected 100 pairs of buggy/fixed programs for our experiment at random. These programs have an average of 10 branch statements. To identify the program points pp on which we should focus the test generation process, we applied a tree edit distance algorithm to buggy and fixed versions of each program. Finally, we evaluated the performance of IFRIT, by computing reachability and uniformity.

1) Tree Alignment: The target point in our experiments is the one that contains the fault or the mutation. When this error is fixed by adding new lines or deleting existing lines of code, the identification of the target program is rather complex. Using Zhang and Shasha's algorithm [35], the alignment process converts the programs into their abstract syntax trees and calculates the tree edit distance between them. This algorithm provides the shortest sequence of edit commands at the node granularity that converts one tree into the other. Each node is labeled with the edit operation corresponding to it, which can be *transform, insert, delete,* or *keep*. Using this information, we set the program point after the modified node in the fixed version. In the presence of multiple modified lines, we treat them all as target program points, with the beginning of each area serving as the program point of interest. This ensures that IFRIT targets all modified code.

```
1  int function(int x) {
2      x--; //keep
3      x+=2; // del
4      //pp
5      if(x>5) //keep
6          x = x+1; //keep
7      }
```

```
1  int function(int x) {
2      x--; //keep
3
4      //pp
5      if(x>5) //keep
6          x = x+1; //keep
7      }
```

Fig. 3. Example of alignment

Figure 3 shows an example of target point identification. In this example, the fix of the fault requires deleting a line. We set the program point at line 5, immediately after the deleted node.

B. Evaluation Procedure

To answer RQ0, we conducted a preliminary experiment on numeric input manipulation. The objective is to determine the optimal number of scale factors to use. We selected a subset of 10 programs coming from CodeFlaws (CF) [32]. We tested on each program 5 different groups of scale factors. We selected scale factors that can ensure a good exploration of the input space once combined, considering the input domain and the episode length during the testing phase. Configurations A-B-C uses small scale factors that allow to easily generate inputs close among them. Configurations D-E feature both small both large scale factors, that allow to quickly traverse the input space.

To answer RQ1, we measure the performance of IFRIT in terms of reachability and uniformity. In RQ2, we compare the

same metrics with respect to other baseline techniques. Every experiment was repeated 20 times per program to account for the non determinism of the algorithms involved. Moreover, we checked whether there is a statistically significant difference between these distributions by using the Wilcoxon rank sum test. When the p-value is smaller than 0.05, we consider that there is a significant difference between IFRIT and the best performing generator between the opponents.

The last part (RQ3) compares the ability of IFRIT to detect mutants and real faults with respect to the baseline tools. We dedicate one hour of testing time per technique for each program point.

VII. Experimental Results

A. Reachability and Uniformity

Configuration	Scale Factors	Reachability
A	1,2,3	90%
B	1,2,3,4,5	100%
C	3,4,5,6,7	92%
D	1,2,5,10	87%
E	1,2,5,10,100	83%

TABLE I

Table I shows the reachability obtained with different configurations of scale factors of IFRIT. Configuration *B* achieves the highest score in terms of reachability. When only a few scale factors are used (i.e., *Configuration A*), IFRIT does not explore enough the input space of the programs and does not generate enough different inputs within the same episode. The problem with the other configurations (i.e., *Configurations C-D-E*) is that they tend to generate sparse inputs, which do not explore accurately the neighborhoods of the inputs that are close to those reaching the program point. Configuration *B* has enough scale factors to explore the input space at large, but at the same time it has factors that are small enough to avoid overly big jumps in the input space.

RQ0: Configuration *B* achieves the highest score in terms of reachability. Hence it has been selected as the default configuration of IFRIT.

Table II shows the comparison between IFRIT and baseline tools when dealing with programs that accept only numeric inputs, as DFT can handle only this type of programs. We chose a program point at the beginning of each branch of each program in our corpus to measure reachability and uniformity. When there is a statistically significant difference between IFRIT and the second best performing generator according to the Wilcoxon test (p-value < 0.05), we show the performance metric in boldface, including the Vargha-Delaney effect size in brackets (N = Negligible; S = Small; M = Medium; L = Large).

To measure reachability, we read the traces produced by the instrumentation and verified that the flag of the program point is active for the given test case. We calculated the percentage of tests that reached their target program points for each test suite and show their descriptive statistic in Table II (mean). This table shows that the Random has the worst reachability results (around or lower than 50%), and Random Mutations behaves similarly. DFT performs well on the programs coming from CodeFlaws. The metric "Time to IFRIT" (i.e., T. to IFRIT) in Table II is computed for Random as the hypothetical time in minutes needed to produce the same number of inputs that reach the target as IFRIT by continuing random generation beyond the test suite size limit (indicated in column 2). For instance, if the test suite size is 2000 and the IFRIT reachability is 90% (Random reachability is 40%), we know that in total IFRIT has been able to produce 1800 (Random: 800) inputs that reach the target. Hence, the execution time of Random should be multiplied by a factor $\times 2.25$ (i.e., 1800/800) to generate the same number of reaching inputs produced by IFRIT.

On simple programs, like the ones contained in CF, there is no advantage in using IFRIT rather than a random algorithm. In fact, the time to achieve reachability with Random is lower than the time that IFRIT takes to generate the same amount of inputs that reach the target. Instead, on *tcas* it is clear that a random generator can not match the performance of IFRIT in a reasonable amount of time. Considering DFT, its reachability on CF is close to that of IFRIT. This could be due to the simplicity of the programs that the CF repository contains. On *tcas*, IFRIT performs better than DFT, and the difference is statistically significant. This could be due to the higher complexity of the program under test, which the symbolic execution component of DFT can not handle efficiently.

We measured the uniformity of the approaches by running the L2-test, which requires the definition of a metric of distance and of the associated ϵ-margin. We used $\epsilon = 0.05$. This distance is smaller than the traditional distance from the state-of-the-art experiments, where it is generally around 0.25 [36], hence setting a stricter criterion for the L2-test. We limited the number of samples generated for the experiments considering the following domain sizes: 2,000, 6,000, 12,000, and 24,000, respectively. The L2-test results are shown in Table II. Percentages represent the proportion of test suites generated for each program point that passed the L2-test. Because it samples directly from the uniform distribution, Random passes the L2-test in all cases by construction (its value is always 100%). Hence, it is not included in Table II. As we can see, IFRIT achieves significant improvements in almost every scenario w.r.t. DFT.

Table III shows the comparison between IFRIT and baselines tools when dealing with programs that accept string inputs. In this scenario, we were not able to test DFT since it only generates numeric inputs. The Random string generator, which builds up a random string length at first, and then it generates random characters to fill the string, shows the worst reachability results. We do not report its uniformity results because it generates a uniform distribution by construction. Random Mutations, which randomly mutates previously computed strings, performs better than Random. Still, the reachability of IFRIT is higher than that of Random Mutations,

Repo		IFRIT			DFT			Random			Random Mutations			
	Size	Reach.	Unif.	Time	Reach.	Unif.	Time	Reach.	Time	T. To IFRIT	Reach.	Unif.	Time	T. To IFRIT
CF	2000	100%	**100% (M)**	468	100%	85%	158	48%	8	17	60%	90%	12	20
	6000	100%	**95% (S)**	468	99%	85%	483	51%	25	49	70%	90%	28	40
	12000	88%	**95% (M)**	468	82%	80%	683	52%	53	90	53%	85%	59	98
	24000	84%	**90% (L)**	468	81%	60%	1071	53%	97	153	51%	85%	108	178
tcas	2000	**95% (S)**	95%	477	85%	90%	161	9%	23	243	24%	85%	29	115
	6000	**93% (M)**	90%	477	83%	85%	476	10%	70	651	20%	90%	82	381
	12000	**90% (M)**	**95% (S)**	477	80%	85%	657	13%	140	969	19%	85%	157	744
	24000	**86% (S)**	**95% (S)**	477	78%	85%	1034	13%	280	1852	14%	85%	284	1745

TABLE II

COMPARISON ON MEAN REACHABILITY AND MEAN UNIFORMITY ACROSS THE CONSIDERED TOOLS. BOLDFACE HIGHLIGHTS THE BEST RESULTS, WHEN STATISTICAL SIGNIFICANCE IS REACHED (THE EFFECT SIZE IS SUMMARIZED IN BRACKETS: N = NEGLIGIBLE; S = SMALL; M = MEDIUM; L = LARGE).

with statistical significance and large effect size. Looking at the metric "Time To IFRIT", on simple programs (e.g., *printtokens*, and *printtokens2*), both Random and Random Mutations are more convenient to use than IFRIT. IFRIT becomes more convenient to use when dealing with more complex programs (e.g., *gzip* and *grep*) and a test suite of size 6000. Figure 4 shows the reachability of the different generators on `tcas` (with test suite size: 12,000) over time. Random is the fastest to terminate, but its reachability is low compared to the other generators. DFT reaches a plateau after 500 minutes and remains quite stable until the end. This plot confirms that IFRIT is faster than DFT, and achieves better results than Random and DFT.

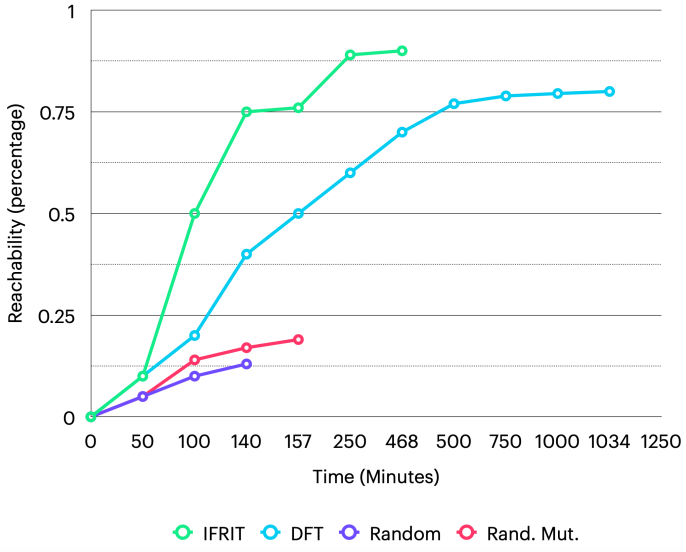

Fig. 4. Reachability over time for IFRIT, DFT, and Random on `tcas` (with test suite size: 12,000).

RQ1: IFRIT produces test suites with a close to uniform distribution and it reaches the target program point 85% of the times on average.

RQ2: IFRIT improves reachability and uniformity with respect to the baselines and state-of-art tools.

```
int alt_sep_test(){
    bool enabled, tcas_equipped, intent_not_known;
    bool need_upward_RA, need_downward_RA;
    int alt_sep;

    enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)
              && (Cur_Vertical_Sep > MAXALTDIFF);
    tcas_equipped = Other_Capability == TCAS_TA;
    intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;

    alt_sep = UNRESOLVED;

    if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)){
        need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
        need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
        ...
    }
    ...
}
```

Fig. 5. Example of conditional branch from `tcas`, where the DFT parameterized constraints are unsatisfiable or divergent, while IFRIT input mutations are guided toward the target.

Qualitative Analysis: Figure 5 shows an excerpt from `tcas`, which includes a target program point controlled by the condition `enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped)`. The boolean variables that appear in this condition are defined in previous statements as boolean expressions that involve input variables (identifiers made of words starting with a capital letter and continuing with lower case letter, such as `Other_Capability`) and constants (upper case identifiers, such as `TCAS_TA`).

The symbolic execution step of DFT will replace all program variables (e.g., `tcas_equipped`) with the expressions defining them along the path of interest and will repeat the process recursively until only input variables and constants are left. In the example in Figure 5, such symbolic substitution produces a rather complex boolean expression for the condition controlling the target. Indeed, the resulting symbolic condition contains 6 distinct input variables and 4 distinct constants, and involves 8 boolean or relational operators. Then, DFT relaxes the constraints by introducing parameters to be optimized by a search algorithm. However, due to the complexity of the parameterized expression, the SMT solver executed to find a solution either finds no solution (8% of the cases) or finds a diverging solution, i.e., a solution that respects the parameterized constraints but do not lead to the target.

On the contrary, the incremental input mutation process of

Repo		IFRIT			Random Mutations				Random		
	Size	Reach.	Unif.	Time	Reach.	Unif.	Time	T. To IFRIT	Reach.	Time	T. To IFRIT
printtokens	2000	**89% (L)**	90%	523	51%	100%	70	123	40%	58	129
	6000	**86%(L)**	85%	501	60%	95%	130	186	39%	112	246
printtokens2	2000	**88% (L)**	85%	502	56%	85%	74	116	37%	62	147
	6000	**84%(L)**	90%	490	65%	85%	134	173	39%	117	252
flex	2000	**83% (L)**	80%	496	22%	85%	69	260	13%	65	415
	6000	**80% (L)**	85%	478	32%	90%	123	307	15%	123	656
gzip	2000	**70% (L)**	85%	533	14%	85%	87	435	8%	75	656
	6000	**70% (L)**	85%	520	22%	85%	160	509	10%	137	959
grep	2000	**72% (L)**	85%	487	12%	80%	92	552	9%	81	648
	6000	**71% (L)**	85%	483	24%	85%	174	514	13%	143	781

TABLE III
REACHABILITY AND UNIFORMITY ACROSS THE CONSIDERED TOOLS. BOLDFACE HIGHLIGHTS THE BEST RESULTS, WHEN STATISTICAL SIGNIFICANCE IS REACHED (THE EFFECT SIZE IS SUMMARIZED IN BRACKETS: N = NEGLIGIBLE; S = SMALL; M = MEDIUM; L = LARGE).

Repo	Mutants				Faults			
	IFRIT	DFT	Random	Rand. Mut.	IFRIT	DFT	Random	Rand. Mut.
CF	**89%(S)**	79%	71%	72%	81%	80%	74%	72%
tcas	**85%(M)**	73%	7%	10%	**65% (S)**	55%	10%	13%
	IFRIT	**Rand. Mut.**	**Random**		**IFRIT**	**Rand. Mut.**	**Random**	
printtokens	**80% (L)**	32%	23%		**73% (L)**	40%	26%	
printtokens2	**74% (L)**	38%	27%		**75% (L)**	35%	28%	
flex	**70% (L)**	32%	25%		**81% (L)**	28%	23%	
gzip	**74% (L)**	15%	13%		**78% (L)**	12%	17%	
grep	**70% (L)**	14%	9%		**73% (L)**	9%	14%	

TABLE IV
MUTATION SCORE AND FAULTS DETECTED ON THE TWO REPOSITORIES IN A FIXED TIME BUDGET (ONE HOUR). BOLDFACE HIGHLIGHTS THE BEST RESULTS, WHEN STATISTICAL SIGNIFICANCE IS REACHED (THE EFFECT SIZE IS SUMMARIZED IN BRACKETS: N = NEGLIGIBLE; S = SMALL; M = MEDIUM; L = LARGE).

IFRIT can find a solution by exploring the neighborhood of previously attempted candidate inputs. While initially such a search process is mostly random, once any viable solution is found, the positive reward received by the RL algorithm is consolidated into its exploration policy. Hence, in the next iterations the RL algorithm will exploit such accumulated knowledge to select the actions (input mutations) that are more likely to lead to the target. At the same time, as a larger reward Γ_1 is granted only when new inputs that reach the target are generated, the learned policy will avoid the mere repetition of previous actions, which would re-generate the same inputs multiple times (this is associated with a smaller reward Γ_0), and will promote diversity in the generation process.

Overall, reachability of IFRIT for this target branch was 92%, while DFT had a reachability of 80%. At the same time, IFRIT passed the uniformity L2 test 90% of the times.

B. Mutation Score and Faults Detected

We evaluate IFRIT in terms of mutation score and number of faults detected and compare it with the baseline techniques. We created up to 100 mutants per program using Milu [37]. We did not filter the generated mutants. We used the same test suite on both the mutant and the original program to see if they produce different results. When the test results differ, we conclude that the test suite strongly kills the mutant. The mutation score represents the percentage of mutants that were killed. We used the alignment algorithm described in Section VI to determine the program point where the mutation or the fault is located. We allocate one hour of testing time per technique for each program point.

Table IV shows the mutation score and percentage of faults detected for each technique and repository. On numeric inputs, we compare Random, Random Mutation, DFT, and IFRIT. On CF, IFRIT performs statistically better than DFT in killing mutants. The effect size is small. On `tcas`, IFRIT is statistically better both in killing mutants and in fault detection. Considering the programs that accept string inputs, IFRIT reaches up to 61% improvement in mutation score and up to 59% improvement in detecting real faults.

Figure 6 shows the asymptotic behavior of the different

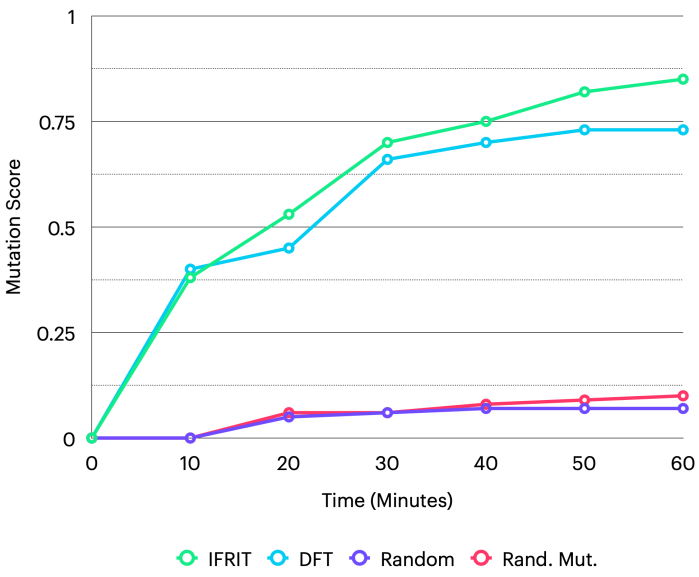

Fig. 6. Mutation score over time for IFRIT, DFT, and Random on `tcas`.

generators on `tcas`. DFT reaches a plateau and remains quite stable earlier than IFRIT, which keeps a positive derivative all over the allotted time budget, while Random and Random Mutations are not enough powerful to kill a significant proportion of mutants. This plot confirms that IFRIT outperforms the rest of the tools, achieving better results even on the SIR repository.

> RQ3: On every repository, IFRIT performs better than the baselines, with an improvement between 18% and 78% on mutation score, and between %7 and %61 on fault detection.

VIII. Extensions and Limitations

A. Future extensions

The current implementation of IFRIT is limited to the manipulation of numeric and string inputs only. However IFRIT can be extended to deal with types such as structures and pointers, by recursively applying the generators for strings and numbers, for structure fields that belong to these two types, and by choosing abstract memory references to typed data structures from a pool of available memory references, when pointers are used as structure fields. This extension of IFRIT is part of the ongoing tool development.

Another potential extension of IFRIT would be to add support for other programming languages. Given the black box nature of the RL algorithm being used, the main limitations are technological (e.g., how to collect coverage information) rather than conceptual.

B. Threats to Validity

Construct Threats. Our definition of diversity may pose a threat to construct validity. Due to its grounding on information theory, we used uniformity as a measure of diversity. Other authors, on the other hand, use test case similarity metrics. It is possible that different findings would have been obtained if similarity measures had been used instead of relying on diversity/distance.

External Threats. Our experiments are performed on open-source code repositories. Although our subjects have been previously used in the literature [7], [38], [39], our results might not generalize to other subjects or programming languages.

IX. Conclusion

IFRIT improves the quality of fault detection and mutation killing when a focused test suite that reach a given target is to be generated automatically. By using Deep RL, our approach enhances the diversity of the test suite, making the input distribution close to a uniform distribution. Empirical results show that the quality of the test suites generated by IFRIT significantly outperforms random generation and the state-of-the-art tool DFT. Moreover, IFRIT is faster in generating big test suites since it does not rely on symbolic execution.

REFERENCES

[1] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the 36th International Conference on Software Engineering, ICSE*, 2014, pp. 435–445. [Online]. Available: https://doi.org/10.1145/2568225.2568271

[2] I. Ahmed, R. Gopinath, C. Brindescu, A. Groce, and C. Jensen, "Can testedness be effectively measured?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 547–558. [Online]. Available: https://doi.org/10.1145/2950290.2950324

[3] J. M. Voas, "PIE: A dynamic failure-based technique," *IEEE Trans. Software Eng.*, vol. 18, no. 8, pp. 717–727, 1992. [Online]. Available: https://doi.org/10.1109/32.153381

[4] D. Shin, S. Yoo, and D. Bae, "A theoretical and empirical study of diversity-aware mutation adequacy criterion," *IEEE Trans. Software Eng.*, vol. 44, no. 10, pp. 914–931, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2732347

[5] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI*, 2008, pp. 209–224. [Online]. Available: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[6] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 8:1–8:42, 2014. [Online]. Available: https://doi.org/10.1145/2685612

[7] H. D. Menéndez, G. Jahangirova, F. Sarro, P. Tonella, and D. Clark, "Diversifying focused testing for unit testing," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 4, pp. 1–24, 2021.

[8] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 70–81.

[9] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[10] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, 2005, pp. 213–223. [Online]. Available: https://doi.org/10.1145/1065010.1065036

[11] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verification Reliab.*, vol. 14, no. 2, pp. 105–156, 2004. [Online]. Available: https://doi.org/10.1002/stvr.294

[12] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013. [Online]. Available: https://doi.org/10.1109/TSE.2012.14

[13] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Software Eng.*, vol. 44, no. 2, pp. 122–158, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2663435

[14] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "GRT: program-analysis-guided random testing (T)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, 2015, pp. 212–223. [Online]. Available: https://doi.org/10.1109/ASE.2015.49

[15] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering, ICSE*, 2007, pp. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[16] N. Alshahwan and M. Harman, "Augmenting test suites effectiveness by increasing output diversity," in *Proceedings of 34th International Conference on Software Engineering, ICSE*, 2012, pp. 1345–1348. [Online]. Available: https://doi.org/10.1109/ICSE.2012.6227083

[17] R. Feldt, S. M. Poulding, D. Clark, and S. Yoo, "Test set diameter: Quantifying the diversity of sets of test cases," in *Proceedings of IEEE International Conference on Software Testing, Verification and Validation, ICST*, 2016, pp. 223–233. [Online]. Available: https://doi.org/10.1109/ICST.2016.33

[18] M. Biagiola, A. Stocco, F. Ricca, and P. Tonella, "Diversity-based web test generation," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*, 2019, pp. 142–153. [Online]. Available: https://doi.org/10.1145/3338906.3338970

[19] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "A scalable approximate model counter," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2013, pp. 200–216.

[20] T. M. Cover, *Elements of information theory*. John Wiley & Sons, 1999.

[21] Y. Marhuenda, D. Morales, and M. Pardo, "A comparison of uniformity tests," *Statistics*, vol. 39, no. 4, pp. 315–327, 2005.

[22] R. L. Plackett, "Karl pearson and the chi-squared test," *International Statistical Review/Revue Internationale de Statistique*, pp. 59–72, 1983.

[23] O. Goldreich and D. Ron, "On testing expansion in bounded-degree graphs," in *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*. Springer, 2011, pp. 68–75.

[24] ——, "On testing expansion in bounded-degree graphs," in *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation*. Springer, 2011, pp. 68–75.

[25] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[26] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[28] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[29] 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.

[30] J. Clark and D. Amodei. (2016) Faulty reward functions in the wild. [Online]. Available: https://openai.com/blog/faulty-reward-functions/

[31] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[32] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[33] A. Hill, A. Raffin, M. Ernestus, A. Gleave, A. Kanervisto, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[34] A. K. Gregg Rothermel, Sebastian Elbaum and H. Do. (2006) Software-artifact infrastructure repository. [Online]. Available: https://sir.csc.ncsu.edu/portal/index.php

[35] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[36] I. Diakonikolas, T. Gouleakis, J. Peebles, and E. Price, "Collision-based testers are optimal for uniformity and closeness," *arXiv preprint arXiv:1611.03579*, 2016.

[37] Y. Jia and M. Harman, "Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language," in *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*, 2008, pp. 94–98.

[38] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse, "Adaptive random testing: The art of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[39] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 130–140.