Learning How to Search: Generating Exception-Triggering Tests Through Adaptive Fitness Function Selection

Anonymous Authors
Department of Anonymous
University of Anonymous
Somewhere, Some Country
anony@mous

***Abstract*—Search-based test generation is guided by feedback from one or more fitness functions—scoring functions that judge solution optimality. Choosing informative fitness functions is crucial to meeting the goals of a tester. Unfortunately, many goals—such as forcing the class-under-test to throw exceptions—*do not* have a known fitness function formulation. We propose that meeting such goals requires presenting fitness function identification as a secondary optimization. An *adaptive* algorithm that can vary the selection of fitness functions could adjust its selection throughout the generation process to maximize goal attainment. To test this hypothesis, we have implemented two reinforcement learning algorithms in the EvoSuite framework.**

We have evaluated our framework, EvoSuiteFIT, on a set of 428 real faults. EvoSuiteFIT discovers and retain more exception-triggering input and produces suites that detect a variety of faults missed by the other techniques. The ability to adjust fitness functions during the generation process allows EvoSuiteFIT to make strategic choices that produce more effective test suites.

***Index Terms*—Automated Test Generation, Search-Based Software Testing, Reinforcement Learning**

I. Introduction

Test creation—the selection of sequences of program stimuli and judgment of the resulting execution [5]—is an expensive, effort-intensive task. If test creation could be even partially automated, the benefit to developers in terms of effort and cost would be immense. Naturally, a large body of research has been amassed around automated test input generation [3].

One area that has shown great promise is *search-based test generation* [3], [25]. Input generation—the selection of program features and parameter values—can naturally be seen as a search problem [17]. Testers approach input selection with a **goal** in mind—perhaps they would like to *cause the program to crash*, *maximize code coverage*, *detect a set of known faults*, or any number of other potential goals. Of the near-infinite number of possible input that could be provided to a program, the tester seeks input that achieves the chosen goal.

This search can be automated. Given a goal, an optimization **algorithm** can systematically sample the space of possible test input in search of a solution to that goal, guided by feedback from one or more **fitness functions**—numeric scoring functions that judge the optimality of the chosen input [13]. In other words: $algorithm + fitness\ functions \implies goal$. Maximizing attainment of your goal with search-based generation relies on the selection of the right feedback mechanism—the right fitness functions. The best fitness functions offer the information needed to rapidly increase attainment of the goal.

For example, a common goal in test generation is maximum Branch Coverage. Branch Coverage is a measurement of *how much* of the code has been executed. For each program statement that can cause the execution path to diverge—such as `if` and `case` statements—test input should ensure that at all potential outcomes are covered at least once [27]. The most effective fitness functions for attaining Branch Coverage take each subgoal we wish to cover and judge *how close* the chosen test input was to achieving covering those goals. This concept, the *branch distance* [4], offers the algorithm the feedback needed to inch closer and closer to covering each outcome.

From this example, we can see that the selection of fitness functions is crucially important to maximizing attainment of our goal. For the goal of attaining Branch Coverage, we have known, effective fitness functions that lead to rapid improvement in coverage. Unfortunately, many goals *do not* have a known, effective fitness function formulation. In fact, many goals do not inherently lend themselves to such a formulation. Consider a common goal—"cause the program to crash", often measured by counting the number of *exceptions*—program-interrupting error messages—thrown during test execution [29]. Thrown exceptions indicate the presence of faults and abnormal operating conditions in programs. Thus, tests that cause exceptions to be thrown are valuable.

However, as we cannot know ahead of time how many or what exceptions are possible to throw, "throw more exceptions" is not a goal that translates into an informative fitness representation. Prior work has proposed the use of a count of thrown exceptions as a fitness function [30]. Unfortunately, this count yields poor results in terms of both goal attainment and fault detection, as it offers the algorithm no guidance for improving its guesses [13], [14].

This does *not* mean that there is no way to effectively

This work is supported by XXXXXXXXXX

achieve this goal—throwing more exceptions—with search-based test generation. Rather, we simply do not yet know what fitness functions will be effective. There are many fitness functions available for use in search-based test generation. Careful selection of one or more of those functions could yield high goal attainment. In fact, we may even attain higher goal attainment by adjusting our chosen fitness functions at regular intervals in the generation process, adapting to the suites evolved up to that point. We hypothesize that an *adaptive* algorithm—one that can vary the selection of fitness functions—could make strategic selections that maximize attainment of a our goal.

To evaluate this hypothesis, we propose and implement a *hyperheuristic search* that optimizes the test generation process [19]. Through the use of reinforcement learning [28], this approach is able to learn the most appropriate set of fitness functions for the class-under-test (CUT) and testing goal, and adjust that set as needed during generation. Throughout the test generation process, this algorithm retains test cases that cause exceptions to be thrown, yielding effective test suites. We have implemented two reinforcement learning algorithms—Upper Confidence Bound (UCB) and Differential Semi-Gradient Sarsa (DSG-Sarsa) [28]—in the EvoSuite generation framework [31].

We call the modified framework EvoSuiteFIT. We have evaluated EvoSuiteFIT on 428 Java case examples in terms of the ability of generated test suites to discover and retain exceptions and to detect faults. We compare to three baselines: the existing exception count fitness function, Branch Coverage, and a combination of Branch Coverage and exception count. We observe that:

- EvoSuiteFIT discovers and retains more exception-triggering input than the baseline techniques. DSG-Sarsa outperforms the baselines by up to 105% in terms of exceptions discovered, and up to 340% in terms of exceptions thrown by the final suite.
- EvoSuiteFIT produces suites that detect 11-236% more faults than the baseline techniques. EvoSuiteFIT also yields more failing test cases.
- The ability to adjust the set of fitness functions at regular intervals in the generation process allows EvoSuiteFIT to make strategic choices that refine the test suite. This is not possible when a static set of fitness functions is used throughout the entire generation process.

The use of reinforcement learning algorithms allows EvoSuiteFIT to identify combinations of fitness functions effective at triggering exceptions in a CUT, and vary that set of functions throughout the ongoing generation process. We hypothesize that other goals without known effective fitness function representations could also be maximized in a similar manner. We make EvoSuiteFIT available to others for use in test generation research or practice.

II. Background

A. Search-Based Test Generation

Test case creation can naturally be seen as a search problem [17]. Of the thousands of test cases that could be generated for any CUT, we want to select—systematically and at a reasonable cost—those that meet our goals [25], [1]. Given a well-defined testing goal, and scoring functions denoting *closeness to the attainment of that goal*—called a *fitness functions*—optimization algorithms can sample from a large and complex set of options as guided by a chosen strategy (the *metaheuristic*) [6].

Metaheuristics are often inspired by natural phenomena, such as swarm behavior [10] or evolution [18]. While the particular details vary between algorithms, the general process employed by a metaheuristic is as follows: (1) One or more solutions are generated, (2), The solutions are scored according to the fitness functions, and (3), the feedback from the fitness functions is used to reformulate the solutions for the next round of evolution. This process continues over multiple generations, ultimately returning the best-seen solutions. By determining how solutions are evolved and selected over time, the choice of metaheuristic and fitness functions impact the effectiveness and efficiency of the search process [11].

B. Reinforcement Learning

The n-armed bandit problem [23] describes a situation where you are repeatedly faced with a choice of n different options. After each choice, you receive a reward chosen from a probability distribution dependent on the action selected. Reinforcement learning algorithms are designed to learn the optimal choice of action to maximize the reward earned [28].

Each action has an expected reward when it is selected. Over time, the reinforcement learning algorithm will try different actions and refine its estimations of their value. During each round, the reinforcement learning algorithm will choose an action based on the expected reward of applying it in the current problem state. After applying the action, the algorithm will receive a reward value. The algorithm will update the expected reward for the chosen set using the new reward.

At any time, there will be a portfolio with the greatest estimated value. If the algorithm selects that portfolio, it *exploits* its current knowledge to gain immediate reward. If, instead, it selects a portfolio with an unknown or potentially lower reward, it is *exploring* the option space to improve its estimate of a portfolio's value. Reinforcement learning algorithms are designed to effectively balance exploration and exploitation for different problem spaces [28], [20], [19].

III. Approach

Search-based test generation requires the selection of one or more fitness functions to guide the search process. Careful selection is crucial, as the fitness functions act as strategies to shape the resulting test suite. Fitness functions should be selected to maximize attainment of the tester's overall goal. In practice, however, many goals do not translate cleanly to

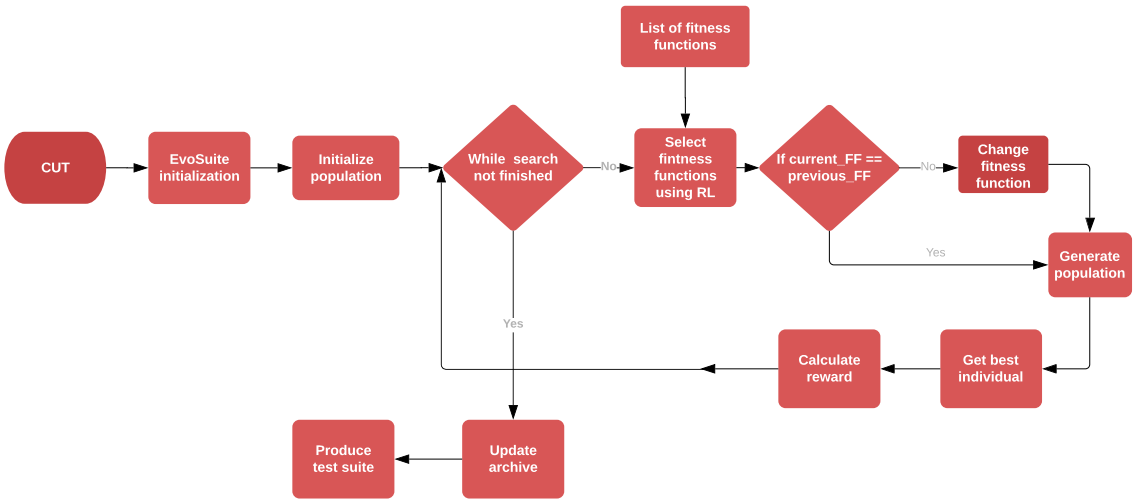


Fig. 1: An overview of how reinforcement learning fits into the test generation process.

an effective fitness function representation—one that offers feedback to the search process to attain rapid attainment.

In this work, we focus on the goal of causing the class-under-test (CUT) to throw exceptions. Exceptions—program-interrupting error messages—are thrown when the software is placed outside of normal operating conditions, and indicate situations the program is not able to handle gracefully [29]. Therefore, it is desirable to force the program into such states in order to identify areas of improvement or faults to fix. It is impossible to know all possible exceptions that can be thrown by a CUT, so there exists no feedback-based fitness function to encourage the discovery of more exceptions. A count of the number of exceptions thrown has been used in previous work [30], [13], [14]. However, this has been shown to be a poor choice for both finding faults [13] or triggering exceptions [14], with general functions such as Branch Coverage able to yield better results in both areas.

We hypothesize, however, that careful selection—at different points in the generation process—of one or more fitness functions could result in test suites that trigger more exceptions and that effectively detect faults. If this is true, identifying this set of fitness functions becomes a secondary search problem—one that could be tackled with AI, machine learning, or optimization techniques as an additional step within the normal test generation process.

We propose the use of reinforcement learning techniques to adapt the set of fitness functions for a chosen CUT and the goal of throwing exceptions. Adjusting the set of fitness functions could be considered as an instance of the n-armed bandit problem [23]. Given a measurable goal, each action—each choice of one or more fitness functions—has an expected reward when it is selected. *If we use this function combination, we will cause additional exceptions to be thrown*. Specifically, we measure reward as the sum of the exceptions discovered during generation and the exceptions thrown by the current solution. This reward function encourages both the discovery and retention of exceptions. Because test generation is a stateful process—the population of test suites at round N depends on the population from round $N-1$—reinforcement learning affords not just an opportunity to identify effective fitness functions, but to strategically adjust the functions based on the changing population of test suites.

We have implemented two reinforcement learning algorithms—Upper Confidence Bound (UCB) and Differential Semi-Gradient Sarsa (DSG-Sarsa)—as part of the EvoSuite test generation framework [31]. The EvoSuite framework uses a genetic algorithm to evolve test suites over a series of generations, forming a new population by retaining, mutating, and combining the strongest solutions. It is actively maintained and has been successfully applied to a variety of projects [32]. In this study, we implemented the two reinforcement learning approaches in EvoSuite version 1.0.7.

This *hyperheuristic search*—a search with an additional process optimization step [19], [20]—adds fitness function selection into the algorithm. The modified process is illustrated in Figure 1. At a user-defined interval, the reinforcement learning algorithm will alter the current set of fitness functions and refine its estimation of their ability to increase the count of exceptions thrown. Throughout generation, we will also retain an *archive* of tests that cause exceptions to be thrown to ensure that the final test suite is effective.

Our modified version of EvoSuite, dubbed EvoSuiteFIT, is available from **[URL removed for double-blind review]**.

A. Upper Confidence Bound (UCB) Algorithm

The Upper Confidence Bound (UCB) algorithm is well-suited to addressing n-armed bandit problems [28]. Each time a choice is made, UCB selects an action that has a higher expected reward than the other possible actions. Each action returns a numerical value that is considered as the reward of taking that action.

For a selected action A at time step t (represented as A_t), the reward R_t represents the corresponding reward of taking action A_t. Using this notation, the expected reward of action a is $q_* \doteq E[R_t|A_t = a]$. We apply UCB to select the action, as defined by Sutton [28]:

$$A_t \doteq max[Q_t(a) + c\sqrt{\frac{lnt}{N_t(a)}}] \quad (1)$$

where A_t represents the index of the combination that give the highest expected reward. The c term represents the confidence level, determining the balance between exploration—refinement of reward expectation at a potential cost to reward—and exploitation—taking advantage of the action currently thought to be best—in the algorithm. The value of c needs to be larger than 0, otherwise the algorithm will behave in a purely greedy manner. $Q_t(a)$ denotes the estimates value of choosing a combination of fitness functions (a), which can be calculated as:

$$Q_t(a) = \frac{1}{N_t}\sum_{i=1}^{t-1} R_i(a) \quad (2)$$

This equation represents the total reward of a combination a divided by the number of times that combination had been selected until the time t. In this project, t denotes the number of generations that the search has progressed through.

B. Differential Semi-Gradient Sarsa (DSG-Sarsa) Algorithm

A special class of reinforcement learning algorithms are known as *approximate solution methods* [28]. Many reinforcement learning approaches associate rewards with particular states, and work best in a constrained state space. Approximate methods are appropriate for problems with a large or unconstrained state space—i.e., test generation—where finding exact solutions is not feasible with limited time [7]. Approximate methods generalize from previously encountered states. As test case generation has a very large state space, we have explored the use of an approximate solution method, Differential Semi-Gradient Sarsa (DSG-Sarsa) [28].

DSG-Sarsa is semi-gradient, enabling continual and online learning. Relevant to our application domain, the algorithm is well-suited to problems in which there is no termination state. This is an "on-policy" method, which means that it tries to improve the policy that the agent has in place to make decisions. The agent leverages past experience to decide when to vary between exploitation and exploration [28]. On-policy methods may be better suited to our application domain than off-policy methods, because on-policy adjustment will allow more exploration than exploitation when necessary—this may be beneficial, given the large number of potential sets of fitness functions that could be chosen.

Each round, the action—choice of fitness functions—is applied, and the test suite evolves to a new state S', with observed reward R. We then use this information to choose a new action A', using the formula:

$$\hat{q}(S, A, W) \doteq W^\top \cdot X(S, A) = \sum w_i x_i(S, A) \quad (3)$$

This action-value function is calculated by the inner product of weights and feature vectors. X(S,A) is the feature vector: $X(s, A) = (x_1(S, A), x_2(S, A), \ldots x_d(S, A))$. The feature vector describes the current state of a test suite using a set of attributes. In this case, the state is represented using the current set of fitness functions, the current fitness value for that set of functions, the suite size, the number of exceptions thrown, and the number of exceptions covered. These are the features considered relevant when describing a test suite.

W represents a weight vector, used to bias action selection [28]. A weight is provided for each feature, and represents the importance of each feature in respect to its contribution to the action value. The weight for an action is updated each round using the semi-gradient with delta, controlled by the learning rate:

$$W_{t+1} \doteq W_t + \alpha\delta\nabla\hat{q}(S_t, A_t, W_t) \quad (4)$$

Where δ is an error function representing the difference between the immediate reward R and the average reward $\bar{R}_t$ and the difference between the value of a target $\hat{q}(S_{t+1}, A_{t+1}, W_t)$ and the value of the old estimate $\hat{q}(S_t, A_t, W_t)$.

$$\delta_t = R_{t+1} - \bar{R}_t + \hat{q}(S_{t+1}, A_{t+1}, W_t) - \hat{q}(S_t, A_t, W_t) \quad (5)$$

$\bar{R}_t$ is the is estimated average reward at time t. $\bar{R}_t$ is calculated using the following equation:

$$\bar{R}_{t+1} = R_t + \beta\delta \quad (6)$$

β is an algorithm parameter that represent the step size of updating the average reward.

By using the average reward, we consider the immediate reward as important as a delayed one. This means that we treat all fitness function combinations impartially without bias toward the combinations that were selected first. This mean there is no priority for the chosen combinations.

C. Implementation in EvoSuite

We have implemented both reinforcement learning algorithms in EvoSuite, and integrate their use into the standard Genetic Algorithm (GA)—adding an additional fitness function selection stage. At a user-defined interval, the RL algorithm will choose a new set of one or more fitness functions. The modified process is illustrated in Figure 1.

We use eight fitness functions alone and in combination:

- **Exception Count:** A simple count of the number of exceptions thrown by a test suite.
- **Branch Coverage:** See Section I.
- **Direct Branch Coverage:** Branch coverage may be attained by calling a method *directly*, or *indirectly*—calling a method within another method. Direct branch coverage requires each branch to be covered through a direct method call.
- **Line Coverage:** A test suite satisfies line coverage if it executes each non-comment source code line at least once. To cover each line of source code, EvoSuite tries to ensure that each basic code block is reached. The branch

distance is computed for each branch that is a control dependency of any of the statements in the CUT. For each conditional statement that is a control dependency for some other line in the code, the branch of the statement leading to the dependent code must be executed.

- **Method Coverage:** Method Coverage simply requires that all methods in the CUT are executed at least once, through direct or indirect calls. The fitness function for method coverage is discrete, as a method is either called or not called.
- **Method Coverage (Top-Level, No Exception):** Test suites sometimes achieve high levels of method coverage by calling methods in an invalid state or with invalid parameters. MNEC requires that all methods be called directly and terminate without throwing an exception.
- **Output Coverage:** Output coverage rewards diversity in the method output by mapping return types to a list of abstract values [2]. A test suite satisfies output coverage if, for each public method in the CUT, at least one test yields a concrete return value characterized by each abstract value. For numeric data types, distance functions offer feedback using the difference between the chosen value and target abstract values.
- **Weak Mutation Coverage:** Test effectiveness is often judged using synthetic faults placed in the code, claled mutants [21]. A test suite satisfies weak mutation coverage if, for each mutated statement, at least one test detects the mutation. The search is guided by the *infection distance*, a variant of branch distance tuned towards reaching and discovering mutated statements [12].

Rojas et al. provide a primer on each of these fitness functions [30]. The RL algorithm chooses a combination of one to four of these fitness functions each time it makes a selection. Initial experimentation revealed that the most effective combinations all include the exception count. Therefore, we filtered the set of choices down to all combinations of one to four fitness functions that include the exception count as one of the choices. This means that the RL algorithm can choose from 64 actions (different sets of fitness functions).

In the beginning, EvoSuiteFIT will make sure that all the actions have been tried once before it starts using the standard UCB or DSG-Sarsa selection mechanisms. This allows seeding of reward estimations. Before the initial selection occurs, the list of actions is randomized to avoid an ordering bias. This is important, as the population of test suites is shaped by the action used each generation. After this stage, every time the RL algorithm makes a selection, the set of chosen fitness functions will change unless the currently-selected combination is exploited.

After changing the fitness functions, EvoSuiteFIT will proceed through the normal population evolution mechanisms, judging solutions using the new set of fitness functions. We use the reformulated population to calculate the reward. Then, we used this reward to update the expectations of the RL algorithm. For UCB, we store the accumulated reward of each combination alongside the number of times each is selected N_t, so we can calculate the average reward. Over time, the combination that gains the highest reward will be more likely to be selected again until reaching convergence. For DSG-Sarsa, after getting the reward, the new combination is selected using the policy. Based on the new and current combination, the new and current state, and the reward, the average reward and the weight of the state is updated. Then the current fitness function combination will change to the new one.

After experimentation, we found that changing the set of fitness functions every three generations allows enough time to adequately adjust reward expectations. Fewer generations does not allow sufficient time for the chosen fitness function combination to reshape the test suite. This means that the GA will have three generations to reshape the population before the reward is evaluated.

In EvoSuiteFIT, during the search and optimization process, test cases that cover a set of chosen goals can be retained in a test archive to prevent loss in coverage as the test suites are reshaped. In traditional EvoSuite, this archive is based on the chosen fitness functions. However, as we use RL to change the fitness function, we have altered how the test archive is used. We store test cases known to trigger discovered exceptions. After the search process completes, the archive is used to produce the final test suite. This prevents the loss of test cases that trigger exceptions due to changes in the fitness functions.

IV. Study

To better understand the real-world effectiveness, use, and applicability of our hyperheuristic approach, we have assessed EvoSuiteFIT against 428 case examples from the Defects4J dataset [22]. In doing so, we wish to address the following research questions:

1) Is either EvoSuiteFIT approach more effective, in terms of the number of discovered exceptions and the number of exceptions thrown by the final test suite, than test generation using static fitness function choices?
2) Is either EvoSuiteFIT approach more effective, in terms of fault detection effectiveness than test generation using static fitness function choices?
3) Are there trends that can be discerned in the behavior of EvoSuiteFIT based on class or project features?

In order to investigate these questions, we have performed the following experiment:

1) **Collected Case Examples:** We have used 428 case examples, from six Java projects, as test generation targets (Section IV-A).
2) **Generated Test Suites:** For each class, we generate 10 suites per approach. Approaches include the two reinforcement learning algorithms—UCB and DSG-Sarsa—and three baselines—generation guided by exception count, Branch Coverage, and a combination of the two. All approaches are allocated the same number of generations, based on approximately 10 minutes of generation time for DSG-Sarsa (Section IV-B).

3) **Removed Non-Compiling and Flaky Tests:** Any tests that do not compile, or that return inconsistent results, are removed (Section IV-B).
4) **Assessed Exception Discovery Effectiveness:** We measure the number of exceptions thrown and detected by each test suite (Section IV-C).
5) **Assessed Fault-finding Effectiveness:** We measure the number of faults detected and the number of failing test cases in each suite (Section IV-C).

A. Case Examples

Defects4J is an extensible database of real faults extracted from Java projects [22][1]. The current dataset (Defects4J 2.0) consists of 438 faults from six projects: Chart (26 faults), Closure (176 faults), Lang (65 faults), Math (106 faults), Mockito (38 faults), and Time (27 faults). For each fault, Defects4J provides access to the faulty and fixed versions of the code, developer-written test cases that expose the fault, and a list of classes and lines of code modified by the patch that fixes the fault.

Each fault is required to meet three properties. First, a pair of code versions must exist that differ only by the minimum changes required to address the fault. The "fixed" version must be explicitly labeled as a fix to an issue, and changes imposed by the fix must be to source code, not to other project artifacts such as the build system. Second, the fault must be reproducible—at least one test must pass on the fixed version and fail on the faulty version. Third, the fix must be isolated from unrelated code changes such as refactorings.

Ten of the case examples produce no exceptions—Closure faults 38, 44, 47, 51, and 144, Math faults 13, 31, and 59, Mockito fault 6, and Time fault 21. These ten cases are excluded from the analysis.

B. Test Suite Generation

For each case example from Defects4J, we have generated test suites using each reinforcement learning approach—UCB and DSGSarsa. In addition, we generate tests for three baseline approaches representing current practice using an unmodified version of EvoSuite:

- **Exception Count**: A common fitness function representation of the goal of throwing exceptions is to simply count the number of exceptions thrown by a test suite. This would be the "default" fitness option for a tester interested in inducing exceptions.
- **Branch Coverage**: Branch Coverage has been found in past studies to be the most effective single fitness function in EvoSuite at fault detection [13].
- **Exception Count and Branch Coverage**: A recent study found a combination of Branch Coverage and exception count to be more effective at fault detection than the use of either alone [14]. This is included as an additional baseline as an "educated guess"—what we might choose in the absence of reinforcement learning.

[1]Available from http://defects4j.org

Test suites are generated that target the classes reported as relevant to the fault by Defects4J. Tests are generated using the fixed version of the CUT and applied to the faulty version in order to eliminate the oracle problem. In practice, this translates to a regression testing scenario, where tests are generated using a version of the system understood to be "correct" in order to guard against future issues [32]. Tests that fail on the faulty version, then, detect behavioral differences between the two versions.

To perform a fair comparison between approaches, each is allocated the same search budget for test generation—measured in terms of the generations allocated to the evolution process. As the time to complete a single generation varies between CUT, we calculate the search budget as the number of generations DSG-Sarsa can complete in a ten-minute period. In past work, 10 minutes was used as the maximum generation time and represented a point of "diminishing returns" for detection of the faults in Defects4J [13].

To control experiment cost, we deactivated assertion filtering—all possible regression assertions are included. All other settings were kept at their default values. As results may vary, we performed 10 trials for each fault and search budget. This resulted in the generation of 21,400 test suites (ten trials, five approaches, 428 faults).

Generation tools may generate flaky (unstable) tests [32]. For example, a test case that makes assertions about the system time will only pass during generation. We automatically remove flaky tests. First, all non-compiling test suites are removed. Then, each remaining test suite is executed on the fixed version five times. If the test results are inconsistent, the test case is removed. This process is repeated until all tests pass five times in a row. On average, less than one percent of tests tends to be removed from each suite.

C. Data Collection

In order to address our research questions, we collect the following data for each test suite:

- **Number of Exceptions Discovered During Generation**
- **Number of Exceptions Thrown by the Final Test Suite**: Tests that trigger an exception can be lost during the generation process.
- **Number of Faults Detected**
- **Number of Failing Test Cases**: A single failing test case is all that is needed to detect a fault. However, multiple failing test cases can provide a tester with more information during the debugging process.
- **Decisions Made by EvoSuiteFIT**: The reinforcement learning algorithms reformulate the fitness function combination in use at regular intervals. Each time a combination is selected, we log the decision made. This can assist in understanding how the reinforcement learning algorithms function, and how they make decisions in service of goal attainment.

TABLE I: Median count of exceptions thrown and exceptions discovered for each technique, along with the median ratio of thrown to discovered. Counts are normalized between 0-1 for each fault to allow comparison across case examples.

	DSG-Sarsa		UCB		Exception Count		Branch Coverage		Branch + Exception	
System	**Thrown**	**Discovered**	**Thrown**	**Discovered**	**Thrown**	**Discovered**	**Thrown**	**Discovered**	**Thrown**	**Discovered**
Chart	0.89	0.89	0.86	0.86	0.37	0.37	0.30	0.46	0.76	0.76
Closure	0.83	0.83	0.81	0.81	0.42	0.42	0.22	0.42	0.74	0.80
Lang	0.94	0.94	0.92	0.92	0.65	0.65	0.25	0.50	0.90	0.91
Math	0.90	0.90	0.89	0.89	0.59	0.59	0.05	0.33	0.50	0.82
Mockito	0.89	0.89	0.93	0.93	0.50	0.50	0.10	0.50	0.50	0.75
Time	0.88	0.88	0.85	0.85	0.42	0.42	0.27	0.43	0.77	0.77
Overall	0.88	0.88	0.86	0.86	0.50	0.50	0.20	0.43	0.71	0.82

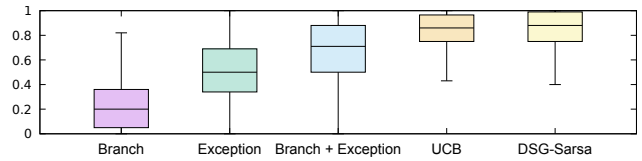


Fig. 2: Exceptions thrown by each technique. Counts are normalized between 0-1 for each fault to allow comparison across case examples.

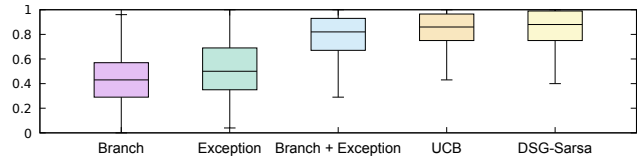


Fig. 3: Exceptions discovered by each technique. Counts are normalized between 0-1 for each fault to allow comparison across case examples.

V. Results & Discussion

We are interested in understanding the effectiveness of EvoSuiteFIT in terms of discovering and exposing exceptions and faults. The following subsections detail our observations.

A. Ability to Discover and Retain Exceptions

Our first research question asks whether reinforcement learning approaches can be used to more effectively meet our goal of throwing more exceptions than baseline approaches. We assess this along two dimensions. First, we look at the number of exceptions **discovered** by each approach during the test generation process. Second, we look at the number of exceptions **thrown** by the final test suite produced by each method. Are exception-inducing tests cases both produced *and* retained by the test generation framework? An effective approach must excel at both tasks.

We do not know a priori how many exceptions can possibly be thrown when testing a class. However, we do know that the number of possible exceptions varies from class to class. Therefore, it is not reasonable to compare raw counts of exceptions between each case example. If we discover up to thirty exceptions when testing one class, and up to five when testing another, we should not compare five to thirty. Instead, we *normalize* exception counts between 0-1, using the formula $\left(\frac{\text{Number of Exceptions Discovered/Thrown}}{\text{Maximum Number of Observed Exceptions for the Current Class}}\right)$. Scaling all counts in this manner allows relative comparisons.

The median count of exceptions thrown and discovered for each technique is listed in Table I for each project and overall.

TABLE II: P-Values for Mann-Whitney rank-sum test for exceptions discovered. Exceptions thrown are the same.

	DSG-Sarsa	UCB	Exception	Branch	B + E
DSG-Sarsa	-	$<$ **0.01**	$<$ **0.01**	$<$ **0.01**	$<$ **0.01**
UCB	0.99	-	$<$**0.01**	$<$**0.01**	$<$**0.01**
Exception	1.00	1.00	-	$<$**0.01**	1.00
Branch	1.00	1.00	1.00	-	1.00
B + E	1.00	1.00	$<$ **0.01**	$<$ **0.01**	-

TABLE III: Results of Vargha-Delaney A Measure for exceptions thrown/discovered. Large positive effect sizes are bolded.

	DSG-Sarsa	UCB	Exception	Branch	B + E
DSG-Sarsa	-	0.52, 0.52	**0.83**, **0.83**	**0.95**, **0.89**	0.69, 0.59
UCB	0.48, 0.48	-	**0.82**, **0.82**	**0.95**, **0.88**	0.68, 0.57
Exception	0.17, 0.17	0.18, 0.18	-	**0.80**, 0.60	0.33, 0.22
Branch	0.05, 0.11	0.05, 0.12	0.20, 0.40	-	0.12, 0.15
B + E	0.31. 0.41	0.32, 0.43	0.67, **0.78**	**0.88**, **0.85**	-

Boxplots of the exceptions thrown and discovered by each technique are shown in Figures 2-3.

The results show that both reinforcement learning techniques have a higher median performance in both measurements than the three baselines, particularly over the default fitness function—counting the number of exceptions thrown. DSG-Sarsa, overall, yields the best performance. In terms of exception discovery, it outperforms the exception count baseline by 76%, the Branch Coverage baseline by 105%, the Branch + exception baseline by 7%, and UCB by 2%. DSG-Sarsa performs even better on exceptions thrown by the final test suite, outperforming the exception count baseline by 76%, the Branch Coverage baseline by 340%, the Branch + exception baseline by 123%, and UCB by 2%. Figures 2-3 show an overall upward trend for each technique, with higher first and third quartiles for the two EvoSuiteFIT techniques than the three baselines.

Both reinforcement learning techniques yield notably superior ability to discover and retain exception-triggering input over traditional baselines like an exception count and Branch Coverage. Our "educated guess", using a combination of Branch and Exception Coverage, yields reasonable good results. However, both reinforcement learning techniques still discover more exceptions on average. Additionally, due to the use of a test archive to retain exception-triggering input, the two techniques do a better job of retaining exceptions.

We can perform statistical analysis to assess our observations. For each pair of techniques and baselines, we formulate hypotheses and null hypotheses:

- H_1: Test suites generated using technique A will have a different distribution of exception discovery results than

TABLE IV: Number of faults detected, along with the median number of failing test cases for each failing suite.

	DSG-Sarsa		UCB		Exception Count		Branch Coverage		Branch + Exception	
System	**# Faults**	**# Failing**	**# Faults**	**# Failing**	**# Faults**	**# Failing**	**# Faults**	**# Failing**	**# Faults**	**# Failing**
Chart	22	7	21	7	9	1	21	2	20	2
Closure	24	2	19	2	6	1	21	1	21	1
Lang	42	3	38	3	12	1	38	2	39	1
Math	73	4	73	4	17	1	59	1	57	1
Mockito	5	3	5	4	3	1	4	1	4	1
Time	15	4	17	3	6	1	17	1	16	1
Overall	178	4	176	4	53	1	160	1	157	1

suites generated using technique *B*.

- H_2: Test suites generated using technique *A* will have a different distribution of exception retention results than suites generated using technique *B*.
- $H0_1$: Observations of exception discovery for both techniques are drawn from the same distribution.
- $H0_2$: Observations of exception retention for both techniques are drawn from the same distribution.

Our observations are drawn from an unknown distribution; To evaluate the null hypotheses without any assumptions on distribution, we use a one-sided (strictly greater) Mann-Whitney-Wilcoxon rank-sum test [34], a non-parametric test for determining if one set of observations is drawn from a different distribution that another set. We apply the test for each pairing of techniques and baselines with $\alpha = 0.05$.

The resulting p-values are listed in Table II. P-values are the same for both discovered and thrown. The results confirm our informal observations. For DSG-Sarsa, we can reject both null hypotheses for UCB and all three baselines. For UCB, we can reject the null hypotheses for all three baselines. For the Branch Coverage + exception count baseline, we can reject the null hypotheses for the other two baselines, but not for either EvoSuiteFIT technique.

We have also used the Vargha-Delaney A measure to assess effect size [33]. The results for both exception discovery and retention are listed in Table III, with large effect sizes in bold. This test further confirms our observations. In terms of exceptions retained, both EvoSuiteFIT techniques outperform the Branch and exception count baselines with a large effect size, and they outperform the Branch + exception baseline with a medium effect size. In terms of exception discovery, both EvoSuiteFIT techniques outperform the first two baselines with a large effect size, and outperform the Branch + exception baseline with a small effect size. DSG-Sarsa outperforms UCB, but with a negligible effect size.

> Both EvoSuiteFIT techniques discover and retain more exception-triggering input than the three baseline techniques, with DSG-Sarsa yielding the best results.

B. Fault Detection Effectiveness

In theory, forcing the class-under-test to throw exceptions will help developers identify faults in the system. Therefore, our second research question revolves around the ability of the generated test suites to trigger and detect failures. Both DGS-Sarsa and UCB trigger more exceptions than the baseline fitness functions. Does this translate into greater fault detection?

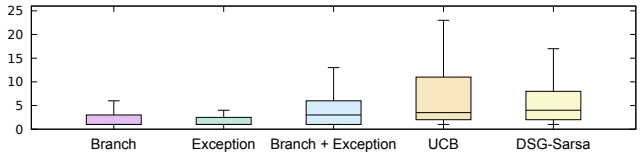


Fig. 4: Number of failing test cases.

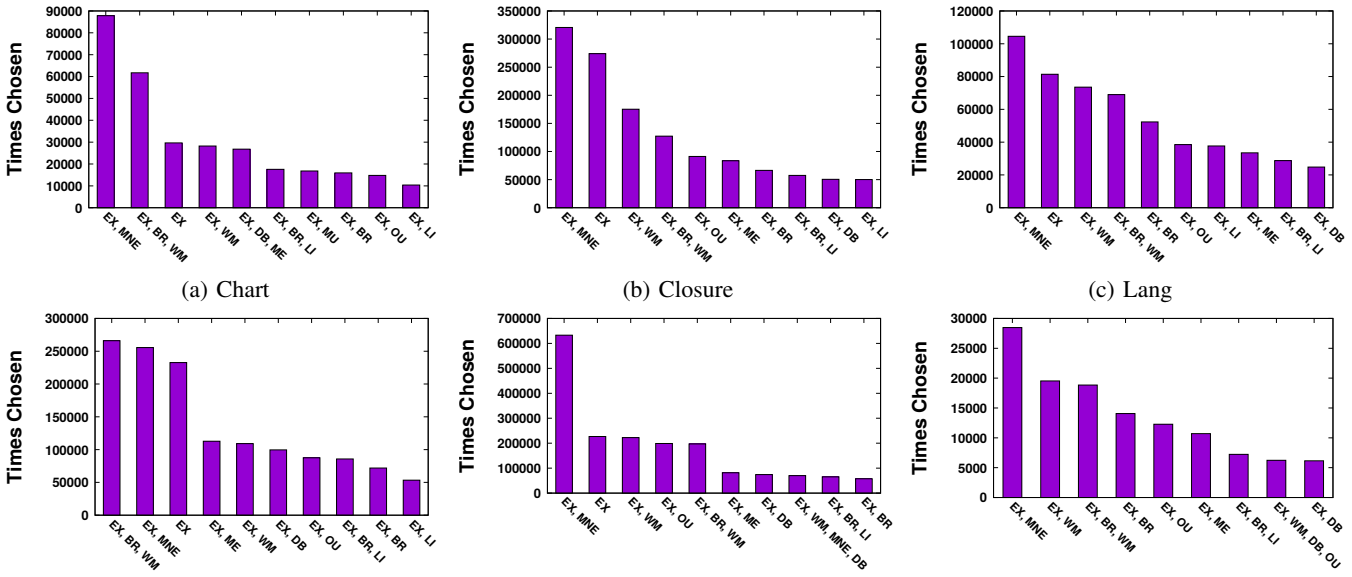
Table IV lists the number of faults detected by each technique. We can immediately see that both EvoSuiteFIT techniques generate suites that are able to detect faults that are missed by suites generated using the baselines. DGS-Sarsa, in particular, detects the most faults—identifying two more faults than UCB, 18 more than Branch Coverage (11% more), 21 more than Branch + exception (13%), and 125 more than the exception count (236%).

> Both EvoSuiteFIT techniques produce suites that detect faults missed by the other techniques. DGS-Sarsa detects 11-236% more faults than the baseline techniques.

We can also examine the number of failing test cases in the generated suites. If a large number of tests fail, we can offer developers more information to understand, localize, and fix the underlying fault. The EvoSuiteFIT technqiues expose more faults than the baselines, but do they also offer more failing test cases? Table IV notes the median number of failing test cases for suites produced by each technique that detect faults. Figure 4 shows a boxplot of the measure.

From the results, we can see that UCB and DSG-Sarsa yield more failing test cases, on average, than the three baselines. Overall, both techniques yield a median of four failing test cases, to the one offered by the baselines. The same results hold across the individual systems, with Chart yielding the largest differences (a median of seven for both EvoSuiteFIT techniques to 1-2 for the baselines). Figure 4 shows an upward trend for the techniques, with both UCB and DSG-Sarsa having higher first and third quartiles as well. UCB, interestingly, has the highest third quartile—sometimes yielding more failing tests than any other technique.

> Both EvoSuiteFIT techniques produce a median of four failing tests per suite, while the baselines yield a median of one failing test per suite.



(a) Chart (b) Closure (c) Lang

(d) Math (e) Mockito (f) Time

Fig. 5: Top ten fitness function combinations chosen by DSG-Sarsa for each system. EX = Exception Count, BR = Branch Coverage, DB = Direct Branch Coverage, LI = Line Coverage, OU = Output Coverage, ME = Method Coverage, MNE = Method (No Exception), WM = Weak Mutation Coverage

TABLE V: P-Values for Mann-Whitney rank-sum test for number of failing test cases.

	DSG-Sarsa	UCB	Exception	Branch	B + E
DSG-Sarsa	-	0.34	**<0.01**	**<0.01**	**<0.01**
UCB	0.66	-	**<0.01**	**<0.01**	**<0.01**
Exception	1.00	1.00	-	1.00	1.00
Branch	1.00	1.00	**<0.01**	-	0.70
B + E	1.00	1.00	**<0.01**	0.30	-

TABLE VI: Results of Cliff's Delta Measure for number of failing test cases. Large positive effect sizes are bolded.

	DSG-Sarsa	UCB	Exception	Branch	B + E
DSG-Sarsa	-	0.01	**0.52**	**0.34**	**0.33**
UCB	-0.01	-	**0.52**	**0.33**	**0.32**
Exception	-0.52	-0.52	-	-0.20	-0.20
Branch	-0.34	-0.33	**0.20**	-	-0.01
B + E	-0.33	-0.32	**0.20**	0.01	-

We can perform statistical analysis to assess our observations on the number of failing test cases. For each pair of techniques and baselines, we formulate hypothesis:

- H_3: Test suites generated using technique *A* will have a different distribution of the number of failing test cases than suites generated using technique *B*.
- $H0_3$: Observations of the number of failing test cases for both techniques are drawn from the same distribution.

Results for the Mann-Whitney-Wilcoxon rank-sum are shown in Table V. The test results confirm our observations. We can reject the null hypothesis for both EvoSuiteFIT techniques—they yield differing numbers of failing test cases. The Vargha-Delaney effect size is inappropriate for this data, as we do not have equal sample sizes for each technique. Instead, we apply Cliff's Delta measure [8] to assess the effect size. Results are shown in Table VI, further confirming our observations. EvoSuiteFIT produces more failing test cases than the baselines.

C. Trends in Reinforcement Learning Behavior

EvoSuiteFIT is able to freely alternate between 64 combinations of fitness functions. To understand why its techniques are effective, it is important to understand how they choose between these actions. We have examined the choices made by DSG-Sarsa. Figure 5 shows the ten combinations chosen most often by DSG-Sarsa for each system.

From this figure, we can see that the systems differ in terms of the frequency choices are made, and to a small extent, in terms of the combinations chosen. For example, DSG-Sarsa frequently used a combination of exception count, Direct Branch Coverage, Weak Mutation Coverage, and Output Coverage for the Time system. However, although the ordering differs, there are also a lot of commonalities between the choices made as well.

For the most part, the favored combinations are simple—pairing the exception count with, at most, one additional fitness function. It is reasonable that simple combinations would be used frequently. Larger combinations introduce a risk of conflicting goals, and are harder to maximize. Simple combinations offer a reasonable feedback mechanism to increase the exception count. Complex combinations may offer less feedback by adding noise to the search.

The exception count alone and a combination of Branch Coverage and exception count appear frequently in the choices made by DSG-Sarsa. Why, then, does DSG-Sarsa outperform the static use of either during test generation? Here it is important to remember that test generation is a stateful process. Each round of the generation process builds on the results of

previous rounds. The ability of DSG-Sarsa to adjust the set of fitness functions is a strength, as there may be times where the strategic choice to apply a function is more effective than using that function at all times. If a suite *already* achieves a high level of Branch Coverage, then it might make sense to switch to the pure use of the exception count to further tune the population of test suites. This adaptation is not possible when a static fitness function is chosen.

> The ability to adjust the set of fitness functions at regular intervals in the generation process allows EvoSuiteFIT to make strategic choices that refine the test suite. This is not possible when a static set of fitness functions is used throughout the generation process.

VI. Related Work

Hyperheuristic search—often based on reinforcement learning—has been employed in addressing a number of search-based software engineering problems. Jia et al. used reinforcement learning to select the metaheuristic algorithm for Combinatorial Interaction Testing, improving performance by learning the best algorithm for test generation for targeted problems [19], [20]. Similarly, Zamli et al. used hyperheuristic search to learn the selection and acceptance mechanisms used by the metaheuristic in Combinatorial Interaction Testing [35]. Guizzo et al. have used a reinforcement learning-based hyperheuristic search to tune the algorithm for optimizing the integration and test ordering problem [15], [16]. In addition, Kumari and Srinivas have used hyperheuristic search to tune software design—with the algorithm learning how to cluster classes for maximum cohesion and minimum coupling [24].

In all of these cases, the hyperheuristic is used to tune the algorithm itself, and not the fitness functions. Fitness function selection has been performed by hyperheuristic search in other domains, such as production scheduling [9], [26]. However, our approach is the first automated technique for optimizing the set of fitness functions used during test generation.

VII. Threats to Validity

External Validity: Our study has focused on six systems—a relatively small number. Nevertheless, we believe that such systems are representative of, at minimum, other small to medium-sized Java systems. We believe that Defects4J offers enough fault examples that our results are generalizable to other, sufficiently similar, projects. As Defects4J is used across multiple research fields, the use of this dataset also allows comparisons of our approach with other research, and allows others to replicate our experiments.

We have implemented our reinforcement learning techniques in a single test generation framework. There are many search-based methods of generating tests and these methods may yield different results. Unfortunately, no other generation framework offers the same number and variety of fitness functions. Therefore, a more thorough comparison of tool performance cannot be made at this time. By using the same framework to generate all test suites, we can compare our approach to the baselines on an equivalent basis.

To control experiment cost, we have only generated ten test suites for each combination of fault, budget, and fitness function. It is possible that larger sample sizes may yield different results. However, given a lack of outliers in our experiment results, we believe that this is a sufficient number of repetitions to draw stable conclusions.

Conclusion Validity: When using statistical analyses, we have attempted to ensure the base assumptions behind these analyses are met. We have favored non-parametric methods, as distribution characteristics are not generally known a priori, and normality cannot be assumed.

VIII. Conclusions

Search-based test generation algorithms sample the space of possible test input, guided by feedback from one or more fitness functions—numeric scoring functions that judge the optimality of chosen input. Choosing informative fitness functions is crucial to meeting the goals of a tester. Unfortunately, many testing goals—such as forcing the class-under-test to throw exceptions—*do not* have a known, effective fitness function formulation. We propose that the key to meeting such goals is to treat fitness function identification as a learning problem. An *adaptive* algorithm—one that can vary the selection of fitness functions—could adjust fitness functions throughout the generation process to maximize attainment of the chosen goal. To test this hypothesis, we have implemented two reinforcement learning algorithms in the EvoSuite framework.

We have evaluated our modified framework, EvoSuiteFIT, on a set of 428 case examples. Both techniques discover and retain more exception-triggering input than three baseline techniques. Both techniques also produce suites that detect a variety of faults missed by the other techniques, and that yield more failing test cases. The ability to adjust the set of fitness functions at regular intervals in the generation process allows EvoSuiteFIT to make strategic choices that refine the test suite. This is not possible when a static set of fitness functions is used throughout the generation process.

The use of reinforcement learning allows EvoSuiteFIT to identify combinations of fitness functions effective at triggering exceptions in a CUT, and vary that set of functions based on the ongoing generation process. We hypothesize that other goals without known effective fitness function representations could also be maximized in a similar manner. We make EvoSuiteFIT available to others for use in test generation research or practice. In future work, we will examine the ability of EvoSuiteFIT to optimize other testing goals, and will further analyze the decisions made by both algorithms.

References

[1] S. Ali, L. C. Briand, H. Hemmati, and R. K. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on*, 36(6):742–762, 2010.

[2] N. Alshahwan and M. Harman. Coverage and fault detection of the output-uniqueness test selection criteria. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 181–192, New York, NY, USA, 2014. ACM.

[3] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[4] A. Arcuri. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability*, 23(2):119–147, 2013.

[5] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[6] L. Bianchi, M. Dorigo, L. Gambardella, and W. Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing*, 8(2):239–287, 2009.

[7] L. Buoniu, D. Ernst, B. De Schutter, and R. Babuka. Approximate reinforcement learning: An overview. In *2011 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 1–8, April 2011.

[8] N. Cliff. *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.

[9] B. Crawford, R. Soto, E. Monfroy, W. Palma, C. Castro, and F. Paredes. Parameter tuning of a choice-function based hyperheuristic using particle swarm optimization. *Expert Systems with Applications*, 40(5):1690 – 1695, 2013.

[10] M. Dorigo and L. M. Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *Evolutionary Computation, IEEE Transactions on*, 1(1):53–66, 1997.

[11] R. Feldt and S. Poulding. Broadening the search in search-based software testing: It need not be evolutionary. In *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*, pages 1–7, May 2015.

[12] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.

[13] G. Gay. The fitness function for the job: Search-based generation of test suites that detect real faults. In *Proceedings of the International Conference on Software Testing*, ICST 2017. IEEE, 2017.

[14] G. Gay. Generating effective test suites by combining coverage criteria. In *Proceedings of the Symposium on Search-Based Software Engineering*, SSBSE 2017. Springer Verlag, 2017.

[15] G. Guizzo, G. M. Fritsche, S. R. Vergilio, and A. T. R. Pozo. A hyper-heuristic for the multi-objective integration and test order problem. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, GECCO '15, pages 1343–1350, New York, NY, USA, 2015. ACM.

[16] G. Guizzo, S. R. Vergilio, and A. T. R. Pozo. Evaluating a multi-objective hyper-heuristic for the integration and test order problem. In *2015 Brazilian Conference on Intelligent Systems (BRACIS)*, pages 1–6, Nov 2015.

[17] M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.

[18] J. H. Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[19] Y. Jia. Hyperheuristic search for sbst. In *Proceedings of the Eighth International Workshop on Search-Based Software Testing*, SBST '15, pages 15–16, Piscataway, NJ, USA, 2015. IEEE Press.

[20] Y. Jia, M. B. Cohen, M. Harman, and J. Petke. Learning combinatorial interaction test generation strategies using hyperheuristic search. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 540–550, Piscataway, NJ, USA, 2015. IEEE Press.

[21] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 433–436, New York, NY, USA, 2014. ACM.

[22] R. Just, D. Jalali, and M. D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[23] M. N. Katehakis and A. F. Veinott Jr. The multi-armed bandit problem: decomposition and computation. *Mathematics of Operations Research*, 12(2):262–268, 1987.

[24] A. C. Kumari and K. Srinivas. Hyper-heuristic approach for multi-objective software module clustering. *Journal of Systems and Software*, 117:384 – 401, 2016.

[25] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14:105–156, 2004.

[26] G. Ochoa, J. A. Vazquez-Rodriguez, S. Petrovic, and E. Burke. Dispatching rules for production scheduling: A hyper-heuristic landscape analysis. In *2009 IEEE Congress on Evolutionary Computation*, pages 1873–1880, May 2009.

[27] M. Pezze and M. Young. *Software Test and Analysis: Process, Principles, and Techniques*. John Wiley and Sons, October 2006.

[28] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, Second Edition An Introduction*. 2018.

[29] M. P. Robillard and G. C. Murphy. Designing robust java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 2–10, New York, NY, USA, 2000. ACM.

[30] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri. Combining multiple coverage criteria in search-based unit test generation. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering*, volume 9275 of *Lecture Notes in Computer Science*, pages 93–108. Springer International Publishing, 2015.

[31] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering*, 22(2):852–893, Apr 2017.

[32] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE 2015, New York, NY, USA, 2015. ACM.

[33] A. Vargha and H. D. Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[34] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):pp. 80–83, 1945.

[35] K. Z. Zamli, F. Din, G. Kendall, and B. S. Ahmed. An experimental study of hyper-heuristic selection and acceptance mechanism for combinatorial t-way test suite generation. *Information Sciences*, 399:121 – 153, 2017.