Multi-objective Search for Model-based Testing

Rui Wang
Department of Computer Science, Electrical Engineering and Mathematical Sciences
Western Norway University of Applied Sciences
Bergen, Norway
rwa@hvl.no

Cyrille Artho
School of Electrical Engineering and Computer Science
KTH Royal Institute of Technology
Stockholm, Sweden
artho@kth.se

Lars Michael Kristensen
Department of Computer Science, Electrical Engineering and Mathematical Sciences
Western Norway University of Applied Sciences
Bergen, Norway
lmkr@hvl.no

Volker Stolz
Department of Computer Science, Electrical Engineering and Mathematical Sciences
Western Norway University of Applied Sciences
Bergen, Norway
vsto@hvl.no

***Abstract*—This paper presents a search-based approach relying on multi-objective reinforcement learning and optimization for test case generation in model-based software testing. Our approach considers test case generation as an exploration versus exploitation dilemma, and we address this dilemma by implementing a particular strategy of multi-objective multi-armed bandits with multiple rewards. After optimizing our strategy using the jMetal multi-objective optimization framework, the resulting parameter setting is then used by an extended version of the Modbat tool for model-based testing. We experimentally evaluate our search-based approach on a collection of examples, such as the ZooKeeper distributed service and PostgreSQL database system, by comparing it to the use of random search for test case generation. Our results show that test cases generated using our search-based approach can obtain more predictable and better state/transition coverage, find failures earlier, and provide improved path coverage.**

***Index Terms*—model-based testing, test case generation, bandit-based methods, multi-objective optimization, genetic algorithm, search-based software testing**

I. Introduction

The complexity of software systems today has amplified the importance of software testing as a scalable and efficient technique to discover defects. However, producing test cases by hand is tedious, expensive, and error-prone. *Model-based testing (MBT)* [1] addresses this problem by automatically generating test cases from abstract (formal) models of the *system under test (SUT)*. These abstract models encode the intended behaviors of the SUT and are often easier to develop and maintain compared to low-level test scripts [2]. However, for complex software systems, it is infeasible to explore and generate all the possible test cases for the software system under test. This means that a challenging decision needs to be made on how many tests cases to generate.

MBT is conducted via the automatic generation and execution of a *test suite*, that is, a set of *test cases*. Before starting MBT process, it is necessary to choose *test adequacy criteria* in order to evaluate the extent to which a test suite contains sufficient test cases. These criteria may consider the discovery of defects and obtaining a good code coverage. In this paper, our test adequacy criteria include state- and transition coverage, linearly independent path coverage, and the number of test cases needed to find the first failure.

It is a challenge for MBT to obtain test adequacy by generating a small test suite having few redundant test cases. Uncontrolled random approaches might result in test suites having redundant test cases which only cover few execution paths of the model and the SUT. Also, the decisions required to select a possible test case to generate faces the exploration versus exploitation dilemma when searching/exploring the state space. This dilemma can be described as finding a balance between: a) the exploration of diverse states/transitions which have not been selected to construct a test case; or have been selected fewer times, but might result in better addressing the test adequacy criteria; and b) the continuous exploitation of the states/transitions which have empirically resulted in better outcomes, with regard to the test adequacy criteria, when constructing a test case.

In this paper, we focus on the test case generation with a search-based approach relying on multi-objective reinforcement learning and optimization. Our aim is to find and generate a subset of test cases that optimizes the results when considering the chosen test adequacy criteria. We consider that 1) the process of test case generation is a problem that faces the exploration versus exploitation dilemma when searching/exploring possible test cases; 2) obtaining good and balanced results of the chosen test adequacy criteria with fewer generated test cases is a multi-objective optimization problem. In particular, we want to implement an efficient search that (1) does not require user-defined weights, which rely on domain knowledge; and (2) adjusts the choices dynamically based on the coverage of previous tests.

The main contribution of this paper is to present a search-based test case generation approach combining: 1) generation

978-1-7281-8913-0/20/$31.00 ©2020 IEEE
DOI 10.1109/QRS51102.2020.00029

of test cases based on a particular strategy of multi-objective multi-armed bandits with multiple rewards; and 2) optimizing the chosen adequacy criteria with a Pareto-efficient multi-objective genetic algorithm, in the form of the non-dominating sorting genetic algorithm (NSGA-II) [3]. We evaluate our approach on several models developed for the Modbat model-based API tester [4] by comparing our search-based testing with the random testing. Our experiments show that our search-based approach can obtain more predictable and better results of the chosen adequacy criteria compared to random test case generation, when considering the trade-offs of the criteria.

A second contribution is an implementation of our bandit-based search strategy in the Modbat model-based API tester, which is a new feature for the Modbat 3.4 release [5]. We define test adequacy criteria as multi-objectives so that Modbat implements our search-based test case generation in addition to its standard random search. The test adequacy criteria are optimized using the jMetal [6], [7] multi-objective optimization framework which applies the NSGA-II algorithm on the Modbat models that we use as training set to find a Pareto optimal solution set having reward parameter settings. The parameter settings in the Pareto optimal solution set is then used as inputs to our bandit-based search strategy to generate test cases for other advanced Modbat models and targeting the chosen test adequacy criteria.

The rest of this paper is organized as follows. Section II provides background on the Modbat model-based API tester, the definition of execution paths as test cases, and the test adequacy criteria. In Section III, we present our approach of multi-objective test case generation and optimization. Section IV presents our experimental evaluation and analyzes the results obtained from the experiments. Section V discusses related work, and in Section VI, we conclude and discuss future work.

II. Model-based Software Testing

Our work assumes that a mechanism for automated test execution of a system under test (SUT) is provided in the form of a test harness and properties (such as assertions) about the behavior of the SUT. In addition to executing test cases automatically, MBT can also generate inputs (or calls) to the SUT automatically, and verify that its output matches the expected output [1]. We introduce MBT of state-based systems in the context of the Modbat tester [4].

A. Modbat Model-based API Tester

Modbat is a model-based testing tool that performs online testing of state-based systems that runs on a Java Virtual Machine [4]. Modbat uses extended finite state machines (EFSMs) [8] as its theoretical foundation and implements extensions in a domain-specific language based on Scala [9]. The EFSMs used by Modbat is formally defined as:

Definition 1 (Extended Finite State Machine [10])**.** *An* extended finite state machine *is a tuple* $M = (S, s_0, V, A, T)$ *such that:*

- S *is a finite set of states, including an initial state* s_0.
- $V = V_1 \times \ldots \times V_n$ *is an* n*-dimensional vector space representing the set of values for variables.*
- A *is a finite set of actions* $A : V \rightarrow (V, R)$*, where* $res \in R$ *denotes the* result *of an action, which is either* successful*,* failed*,* backtracked*, or* exceptional*. A successful action allows a test case to continue; a failed action constitutes a test failure and terminates the current test; a backtracked action corresponds to the case where the enabling function of a transition is* false *[8]; exceptional results are defined as such by user-defined predicates that are evaluated at run-time, and cover the non-deterministic behavior of the SUT. We denote by* $Exc \subset R$ *the set of all possible exceptional outcomes.*
- T *is a transition relation* $T : S \times A \times S \times E$*; for a transition* $t \in T$*, we denote the left-side (origin) state by* $s_{origin}(t)$ *and the right-side (destination) state by* $s_{dest}(t)$*, and use the shorthand* $s_{origin} \rightarrow s_{dest}$ *if the action is uniquely defined. A transition includes a possible empty mapping* $E : Exc \rightarrow S$*, which maps exceptional results to a new destination state.*

Listing 1 illustrates a Modbat model of a garage door control system that we will use as a running example to introduce the basic concepts of Modbat. A valid execution path in a Modbat model starts from the initial state (automatically derived from the first declared state) and consists of a sequence of transitions. Transitions are declared with a concise syntax: "$origin$" $\rightarrow$ "$dest$" := $\{action\}$. The GarageDoorTester model in Listing 1 consists of five states: "$DoorUp$", "$DoorClosing$", "$DoorDown$", "$DoorOpening$", and "End". The initial state is "$DoorDown$" in Line 4.

The GarageDoorTester model tests the garage door system shown in Listing 2 (only fields, public methods and the ***stop*** private method are shown due to page limitations). The garage door system controls the opening and closing of a 2-meter garage door using ***open*** and ***close*** methods to set a door motor with a speed $+0.125m/s$ or $-0.125m/s$, respectively (Line 14 and Line 21 in Listing 2). The system uses a private method ***waitLimitHit*** (Line 18 and Line 25) to check the status of the door every second and it calls the ***stop*** private method (Line 28) when the door is fully open or closed. The system takes 16 seconds to open or close the garage door, as implemented by ***waitLimitHit*** method. When the garage door is fully open or closed, the speed of the door motor is set to zero and the motor is stopped by the ***stop*** private method.

Modbat has built-in *require* and *assert* methods. The GarageDoorTester model in Listing 1 uses the *require* method (Line 5, Line 6, Line 14 and Line 15) in transitions to check if preconditions are fulfilled. Preconditions must be fulfilled in order for a transition to be enabled. For example, if the *require* methods in Line 5 and Line 6 expressing preconditions are satisfied, then the transition "$DoorDown$" $\rightarrow$ "$DoorOpening$" is enabled. The ***open*** method is then called to open the garage door. The preconditions are similar for the transition "$DoorUp$" $\rightarrow$ "$DoorClosing$" that calls the ***close*** method. The attribute ***stay*** of a transition is used to delay (in this case

16 seconds) while waiting for the door to be fully open or closed. After the ***open*** or ***close*** method is called and the corresponding transition is executed, the *assert* methods in Line 10 and Line 11 in transition "*DoorOpening*" → "*DoorUp*", or in Line 19 and Line 20 in transition "*DoorClosing*" → "*DoorDown*", are used as assertions to check that the status of the door and door motor is correct when the door is fully open or closed.

```
1  class GarageDoorTester extends Model {
2    val garage = new GarageDoor()
3    // transitions
4    "DoorDown" -> "DoorOpening" := {
5      require(gargage.doorFullyClosed)
6      require(garage.motorStopped)
7      garage.open()
8      } stay 16000
9    "DoorOpening" -> "DoorUp" := {
10     assert (garage.doorFullyOpen)
11     assert (garage.motorStopped)
12   }
13   "DoorUp" -> "DoorClosing" := {
14     require(gargage.doorFullyOpen)
15     require(garage.motorStopped)
16     garage.close()
17   } stay 16000
18   "DoorClosing" -> "DoorDown" := {
19     assert (garage.doorFullyClosed)
20     assert (garage.motorStopped)
21   }
22   "DoorDown" -> "End" := skip
23   "DoorUp" -> "End" := skip
24 }
```

Listing 1: Modbat model GarageDoorTester.

Modbat actions (which execute code related to transitions) have four possible outcomes: successful, backtracked, failed, or exceptional. Given the different possible outcomes of Modbat actions, different rewards of our bandit-based search strategy are defined in Section III. A *successful* action allows a test case to continue with another transition, if available. An action is *backtracked* and resets the transition to its original state if any of its preconditions are violated. An action *fails* if an assertion is violated, if an unexpected exception occurs, or if an expected exception does not occur. In our GarageDoorTester example, the action of transition "*DoorUp*" → "*DoorClosing*" is backtracked if any *require* methods in the action evaluate to false, and the action fails if any *assert* methods evaluate to false in, e. g., "*DoorClosing*" → "*DoorDown*". If no preconditions or assertions are violated, and no exceptional result occurs, the action is *successful*.

B. Execution Paths and Test Cases

For Modbat models, a finite *execution path* consists of a sequence of transitions starting from the initial state and leading to a *terminal state* (a state without outgoing transitions, or a state after a test failed). Each finite execution path represents a test case generated from a Modbat model. That is, a test case is an execution path consisting of a sequence of transitions. Execution paths of Modbat models are formally defined as:

```
1  class GarageDoor {
2    val garageTopHeight = 2d       // two meters
3    val garageBottomHeight = 0d
4    val motorSpeeds = Map[String, Double]("Zero"
       -> 0.0, "PlusSpeed" -> 0.125, "MinusSpeed"
       -> -0.125)
5    // initial door close
6    var currentDoorHeight = garageBottomHeight
7    // initial motor speed 0.0
8    var motorSpeed = motorSpeeds("Zero")
9    var motorStopped = true
10   var motorUp = false
11   var motorDown = false
12   var doorFullyOpen = false
13   var doorFullyClosed = true
14   def open() {
15     motorUp = setMotorSpeed("PlusSpeed")
16     if(motorUp) {
17         doorFullyClosed = false
18         waitLimitHit()
19     }
20   }
21   def close() {
22     motorDown = setMotorSpeed("MinusSpeed")
23     if(motorDown) {
24         doorFullyOpen = false
25         waitLimitHit()
26     }
27   }
28   private def stop() {
29     motorStopped = setMotorSpeed("Zero")
30     if (motorStopped){
31         currentDoorHeight match {
32             case garageTopHeight
33                 => doorFullyOpen = true
34             case garageBottomHeight
35                 => doorFullyClosed = true
36         }
37     }
38   }
39   ...
40 }
```

Listing 2: Garage door system.

Definition 2 (Execution Path [10])**.** *Let* $M = (S, s_0, V, A, T)$ *be an EFSM. A* finite execution path p *of* M *is a sequence of transitions, which constitute a path* $p = t_0 t_1 \dots t_n, \quad t_n \in T$*, such that* $s_{origin}(t_0) = s_0$*, the origin and destination states are linked:* $\forall i, 0 < i \leq n, s_{origin}(t_i) = s_{dest}(t_{i-1})$*, and* $s_{dest}(t_n) \in S_{terminal}$*;* $S_{terminal}$ *is the set of terminal states.*

C. Test Adequacy Criteria as Multi Objectives

For MBT, test adequacy criteria are often chosen to guide the automatic test case generation so that it produces a good test suite [1]. Modbat supports test adequacy criteria including state- and transition coverage [4], and linearly independent path coverage [10]. The state- and transition coverage indicates the number of states and transitions, respectively, that have been explored by a test suite. A linearly independent path (LIP) is any path through a program that contains at least one new path edge (transition) which is not included in any other linearly independent path [10]. Therefore, the linearly independent path coverage indicates the execution paths covered by a test suite.

These test coverage metrics can be measured as the outcome of the executed test suite and visualized (along with the test model) using Graphviz [4], [10] by Modbat. In addition to coverage, Modbat can also provide the measurement of failures found after a test suite is executed [4]. Thus, for our test case generation approach, we choose four different test adequate criteria: 1) state coverage (Cov_s); 2) transition coverage (Cov_t); 3) linearly independent path (LIP) coverage (Cov_{lip}); and 4) the number of test cases used to find the first failure ($NTest_{fail_l}$). We use these test adequacy criteria as objectives for multi-objective optimization.

Prior to this work, Modbat already supported static weights for transition choices, which affect the likelihood of choosing a given transition. However, a good setting of these weights requires insight into the semantics of the model, and the weights remain fixed during the entire test generation process.

III. Multi-objective Optimization

A test suite consists of a set of test cases derived from the test model, and each test case represents one execution path which in turn consists of a sequence of transitions. The generation of a test case therefore relies on the decisions made in each step to select the transitions that are to be part of the constructed execution paths. As introduced earlier, the decision made to select a transition faces the exploration versus exploitation dilemma in terms of finding a balance between: a) the exploration of different transitions which have not yet been selected; or have selected fewer times; and b) the continuous exploitation of already selected transitions which have empirically resulted in better outcomes (e. g., a high coverage). Reinforcement learning [11] is the subfield of machine learning devoted to studying problems and designing algorithms that analyze this dilemma. The multi-armed bandit problem, extensively studied by Berry and Fristedt [12], is a well-established class of sequential decision problems in the context of reinforcement learning.

A. Bandit Search-based Test Case Generation

Bandit problems consider a player (agent) that needs to choose among K arms (actions) in I rounds on a multi-armed bandit slot gambling machine. The objective is to maximize the cumulative reward (money) as much as possible in a casino by consistently taking the optimal arm (action) over rounds [13]. At each round $i = 1, \ldots, I$, the player selects an arm (action) $j \in \{1, \ldots, K\}$ and receives the reward $r_{(j,i)}$ (money). The player (agent) has a goal: on one hand, finding out (exploit) which arm could be currently optimal to have the highest expected reward; on the other hand, exploring other arms (actions) that currently are not optimal, but may turn out to be optimal in the long run [13], [14], [15].

Several algorithms, such as ϵ-greedy [11], Boltzmann Exploration (Softmax) [16], and Reinforcement Comparison [11] have been proposed to solve bandit problems. In our approach, we rely on the Upper Confidence Bounds (UCB) family [15] of algorithms. For reinforcement learning, the regret is one popular measure of a policy's success in addressing the exploration versus exploitation dilemma. The regret is the expected loss due to the fact that the policy does not always play the best (optimal) action [15]. Compared to other algorithms, the UCB family has been theoretically analyzed and has an expected optimal logarithmic growth of regret uniformly over time [15], [13]. An extension of UCB-style algorithms has proven very successful in computer Go [17]. Lai and Robbins [18] showed that the regret for the multi-armed bandit problem has to grow at least logarithmically in the number of rounds. We use the UCB1 bandit algorithm from the UCB family [15] as a basis for implementing our multi-objective search strategy for test case generation.

The UCB1 algorithm operates as follows:

a) each bandit arm is played once at the initialization of the algorithm.
b) afterwards, the algorithm iteratively plays bandit arm j that maximizes

$$\bar{x}_j + \sqrt{\frac{2 \ln n}{n_j}} \tag{1}$$

where $\bar{x}_j$ is the average reward (in $[0, 1]$) from arm j, n_j is the number of times arm j was played, and n is the overall number of plays so far.

The UCB1 algorithm indicates that the reward term $\bar{x}_j$ encourages the exploitation of higher reward arms, while the term $\sqrt{\frac{2 \ln n}{n_j}}$ encourages the exploration of less-visited arms [15].

Based on the UCB1 algorithm, we consider each transition $t_j \in T$ to select for constructing an execution path (a test case) as a bandit arm to play. We denote the reward function as $r : T \to \mathbb{R}$. After executing a transition $t_j \in T$, its corresponding immediate reward $r_{t_j} \in \mathbb{R}$ is received accumulatively, and computed as $r_{t_j} = \bar{r}_{t_j} + \hat{r}_{t_j}$, where $\bar{r}_{t_j}$ is a transition outcome average reward iteratively accumulated, and $\hat{r}_{t_j}$ is a transition action expected reward iteratively accumulated. All rewards are in the interval $[0, 1]$, and we show how to compute them shortly.

The above implies that our bandit heuristic search (BHS) strategy for test case generation becomes the following:

a) each transition $t \in T$ is selected once at the initialization of the strategy.
b) afterwards, the strategy iteratively select a transition $t_j \in T$ that maximizes

$$\bar{r}_{t_j} + \hat{r}_{t_j} + \sqrt{\frac{2 \ln n_{s_{origin}(t_j)}}{n_{t_j}}} \tag{2}$$

where $\bar{r}_{t_j}$ is the transition outcome average reward (in $[0, 1]$) for transition t_j, $\hat{r}_{t_j}$ is the transition action expected reward for transition t_j, n_{t_j} is the number of times transition t_j was selected, and $n_{s_{origin}(t_j)}$ is the number of times that the origin state s_{origin} of the transition t_j is visited and used to select transitions.

This strategy indicates that the reward terms $\bar{r}_{t_j}$ and $\hat{r}_{t_j}$ jointly encourage the exploitation of higher rewarded transitions, while the term $\sqrt{\frac{2 \ln n_{s_{origin}(t_j)}}{n_{t_j}}}$ encourages the exploration of less-selected transitions.

To iteratively compute a transition outcome average reward $\bar{r}_{t_j}$, we consider four types of transition outcome rewards as a set of rewards $\mathbb{R}_{to}$ including the r_{to_self}, $r_{to\_success}$, r_{to_back}, and r_{to_fail}. Among these four types, the r_{to_self} is a self-transition reward for a successful transition that has $s_{origin} = s_{dest}$, while the $r_{to\_success}$ is a reward given to a successful transition that has $s_{origin} \neq s_{dest}$. The r_{to_back} is the reward for a backtracked transition, and the r_{to_fail} is the reward for a failed transition. We denote a transition outcome reward received at the i'th iteration for a transition $t_j \in T$ by $r_{to_{(t_j,i)}} \in \mathbb{R}_{to}$, and n_{t_j} denotes the number of times transition t_j was selected. Therefore, we compute the accumulated transition outcome average reward $\bar{r}_{t_j}$ for a transition t_j using Equation 3:

$$\bar{r}_{t_j} = \frac{1}{n_{t_j}} \sum_{i=1}^{n_{t_j}} r_{to_{(t_j,i)}} \tag{3}$$

The excepted transition action reward $\hat{r}_{t_j}$ is the sum of the given rewards for pass/fail, weighted by how many times the two verdicts actually occurred. To compute an expected reward iteratively, we take into account four different rewards for two types of transition actions (precondition and assertion) as a set of rewards $\mathbb{R}_{ta}$ including the passed precondition reward $r_{precond\_pass}$, failed precondition reward $r_{precond\_fail}$, passed assertion reward r_{assert_pass}, and failed assertion reward r_{assert_fail}. Then, the excepted transition action reward $\hat{r}_{t_j}$ for a transition t_j can be computed using Equations 4, 5 and 6.

$$\hat{r}_{t_j} = \hat{r}_{t_j\_precond} + \hat{r}_{t_j\_assert} \tag{4}$$

$$\begin{aligned} \hat{r}_{t_j\_precond} = &\frac{C_{precond\_pass}}{C_{precond\_total}} \times r_{precond\_pass} \\ &+ \frac{C_{precond\_fail}}{C_{precond\_total}} \times r_{precond\_fail} \end{aligned} \tag{5}$$

$$\begin{aligned} \hat{r}_{t_j\_assert} = &\frac{C_{assert\_pass}}{C_{assert\_total}} \times r_{assert\_pass} \\ &+ \frac{C_{assert\_fail}}{C_{assert\_total}} \times r_{assert\_fail} \end{aligned} \tag{6}$$

In Equation 4, $\hat{r}_{t_j\_precond}$ represents the expected precondition (action) reward for the transition t_j; $\hat{r}_{t_j\_assert}$ represents the expected assertion (action) reward for the transition t_j. Likewise, the *counts* for passed and failed preconditions and assertions used in Equations 5 and 6, as well as their total number, are updated during each iteration. The overall steps for test case generation with our bandit heuristic search strategy are summarized in pseudocode in Algorithm 1. This is the heuristic search strategy that we have implemented in Modbat. The core of Algorithm 1 is implemented with Equation 2 which is based on UCB1 algorithm. We have mentioned that the UCB1 algorithm has an expected optimal logarithmic growth of regret uniformly over time, so Algorithm 1 based on Equation 2 also has the expected optimal logarithmic growth of regret uniformly over time as the efficiency. We explain the steps related to our search strategy, without showing the pseudocode for how transitions and test cases are executed. Modbat initializes a list of transitions *transitions* and s from an initial state s_0. We need to initialize the number of test cases n, all counter variables (with 0 values), and reward variables. In Line 4, the function EXECUTETRANSITIONS generates and executes a test case consisting of a sequence of selected transitions from an initial state s_0 to a terminal state $s_{terminal}$. In Line 6 in function EXECUTETRANSITIONS, the function BANDITHEURISTICSEARCH is invoked to select a transition *trans* using our bandit heuristic search strategy. Then, this selected transition *trans* is executed by the function EXECUTETRANSITION shown in Line 7, with a transition result of type *result* as the function return value. Meanwhile, function EXECUTETRANSITION calls the function UPDATEEXPECTEDREWARD in Line 16 to update the transition action expected reward $\hat{r}_{t_j}$ for the selected transition *trans* based on Equations 4, 5 and 6. After receiving the return value *result* in Line 7, the function UPDATEAVERAGEREWARD in Line 30 updates the transition outcome average reward $\bar{r}_{t_j}$ for *trans* with Equation 3, based on the result type of *trans*.

B. Bandit Search-Based Test Suite Optimization with JMetal

The aim of our bandit heuristic search strategy is to guide the test case generation with the objective of addressing the test adequacy criteria with smaller test suites containing less redundant test cases. The strategy relies on the configuration of eight different rewards to initialize the test case generation (as shown by the ***Require*** in Algorithm 1). Therefore, to achieve our aim, we need to find optimal solutions to configure these rewards and obtain optimized test adequacy criteria (objectives) defined as in Section II-C, while considering the trade-offs of these criteria. Thus, we consider our bandit heuristic search strategy as a multi-objective optimization problem: tune our bandit heuristic search strategy shown in Algorithm 1 to find the optimal solutions. With these optimal solutions found, we can use them for the test case generation of Modbat models in general with our strategy.

Formally, we assume for our multi-objective bandit search optimization problem that a solution can be described in terms of an 8-dimensional reward decision vector $\vec{r}$ in the reward decision space $\mathcal{R}^8$. Such a solution can be used to initialize the generation of a test suite $ts \in TS$ (initialization of Algorithm 1), where TS is a set of test suites. Then, the vector-valued objective function $\vec{f} : \mathcal{R}^8 \rightarrow \mathcal{O}$ evaluates the quality of a specific solution by assigning it an objective vector $\vec{o} = \vec{f}(\vec{r})$ in the objective space $\mathcal{O}$. We define the reward decision vector as

$$\begin{aligned} \vec{r} = (&r_{to\_self}, r_{to\_success}, r_{to\_back}, r_{to\_fail}, r_{precond\_pass}, \\ &r_{precond\_fail}, r_{assert\_pass}, r_{assert\_fail}), \end{aligned} \tag{7}$$

and the objective vector with our four objectives (test adequacy criteria) as

$$\begin{aligned} \vec{o} &= (f_1(\vec{r}), f_2(\vec{r}), f_3(\vec{r}), f_4(\vec{r})) \\ &= (Cov_s, Cov_t, Cov_{lip}, NTest_{fail1}), \end{aligned} \tag{8}$$

Algorithm 1 Bandit Heuristic Search for Test Case Generation

Require: Initialize s, $transitions$, n, r_{to_self}, $r_{to\_success}$, r_{to_back}, r_{to_fail}, $r_{precond\_pass}$, $r_{precond\_fail}$, r_{assert_pass}, r_{assert_fail}, $C_{precond\_pass}$, $C_{precond\_fail}$, C_{assert_pass}, C_{assert_fail}.

1: **function** EXECUTETESTS
2: **for** i = 1 to n **do** ▷ n:number of test cases
3: EXECUTETRANSITIONS

4: **function** EXECUTETRANSITIONS
5: **while** s is not a $s_{terminal}$ **do** ▷ s:current state, starting from s_0
6: $trans \leftarrow$ BANDITHEURISTICSEARCH($transitions$)
7: $result \leftarrow$ EXECUTETRANSITION($trans$)
8: UPDATEAVERAGEREWARD($result$, $trans$)

9: **function** BANDITHEURISTICSEARCH($transitions$)
10: **if** $transitions$ has any never selected transitions **then**
11: **return** $t_{1st\_unselected}$ in $transitions$
12: **else**
13: **return** $t_{j'}$ in $transitions$ having argmax$\{\bar{r}_{t_j} + \hat{r}_{t_j} + \sqrt{\frac{2 \ln n_{s_{origin}(t_j)}}{n_{t_j}}}\}$

14: **function** EXECUTETRANSITION($trans$)
15: UPDATEEXPECTEDREWARD($trans$)

16: **function** UPDATEEXPECTEDREWARD($trans$)
17: **if** precondition of $trans$ **then**
18: **if** pass **then**
19: update $C_{precond\_pass}$ += 1
20: **else**
21: update $C_{precond\_fail}$ += 1
22: update $C_{precond\_total} = C_{precond\_pass} + C_{precond\_fail}$
23: **if** assertion of $trans$ **then**
24: **if** pass **then**
25: update C_{assert_pass} += 1
26: **else**
27: update C_{assert_fail} += 1
28: update $C_{assert\_total} = C_{assert\_pass} + C_{assert\_fail}$
29: update $\hat{r}_{trans} = \hat{r}_{t_{trans\_precond}} + \hat{r}_{t_{trans\_assert}}$ for $trans$ with Equations 5,6

30: **function** UPDATEAVERAGEREWARD($result$, $trans$)
31: **switch** $result$ **do**
32: **case** *success*
33: $r_{to_{(trans,i)}} = r_{to\_success}$
34: **case** *self*
35: $r_{to_{(trans,i)}} = r_{to\_self}$
36: **case** *backtracked*
37: $r_{to_{(trans,i)}} = r_{to\_back}$
38: **case** *failed*
39: $r_{to_{(trans,i)}} = r_{to\_fail}$
40: update $\bar{r}_{trans} = \frac{1}{n_{trans}} \sum_{i=1}^{n_{trans}} r_{to_{(trans,i)}}$ for $trans$

according to our test adequacy criteria defined in Section II-C. Without loss of generality, it is assumed that all objectives are all equally important and our goal is to optimize them. Therefore, to solve our multi-objective bandit search optimization problem, we need to find those reward decision vectors as solutions that optimize the vector-valued objective function $\vec{f} : \mathcal{R}^8 \rightarrow \mathcal{O}$. These solutions balance the trade-offs between the objectives, and we measure the optimality of the solutions through the concepts of *Pareto optimality* and *Pareto dominance* [19], [20], [21].

Following the concept of *Pareto dominance*, given two solutions $\vec{r} \in \mathcal{R}^8$ and $\vec{r}' \in \mathcal{R}^8$ as reward decision vectors which can be used to initialize two test suites ts and ts', $\vec{r}$ is said to dominate, or *Pareto-dominate*, $\vec{r}'$ (written as $\vec{r} \succ \vec{r}'$) if and only if their objective vectors $\vec{o} = \vec{f}(\vec{r})$ and $\vec{o}' = \vec{f}(\vec{r}')$ satisfy: $\forall i \in \{1, 2, \ldots, k\}, \vec{f}(\vec{r}) \geqslant \vec{f}(\vec{r}') \wedge \exists i \in \{1, 2, \ldots, k\} : \vec{f}(\vec{r}) > \vec{f}(\vec{r}')$. Here, we use $k = 4$ since we have four test adequacy criteria used as objectives. All reward decision vectors that are not dominated by any other reward decision vectors are said to form the *Pareto optimal set* $\mathcal{R}^{8*} \subseteq \mathcal{R}^8$, while the corresponding objective vectors are said to form the *Pareto frontier* $\mathcal{O}^* = \vec{f}(\mathcal{R}^{8*}) \subseteq \mathcal{O}$. That is, the *Pareto optimal set* $\mathcal{R}^{8*}$ contains only non-dominating reward decision vectors as optimal solutions to our multi-objective bandit search optimization problem. Each non-dominating reward decision vector can then be used to initialize the generation of a test suite. This means that we find an optimal subset of test suites $TS^* \subseteq TS$ which balance the trade-offs of our four different test adequacy criteria: no other subset of TS can improve one objective without making another objective worse.

To obtain a *Pareto optimal set* $\mathcal{R}^{8*}$ for our multi-objective bandit search problem, we apply the jMetal [6], [7] Java-based framework for multi-objective optimization using meta-heuristics. jMetal is specifically oriented towards multi–objective optimization, and implements a number of state-of-the-art multi–objective optimization algorithms, such as the NSGA-II [3]. NSGA-II is one of the most well-known and widely used multi-objective evolutionary algorithm to obtain the *Pareto optimal set*.

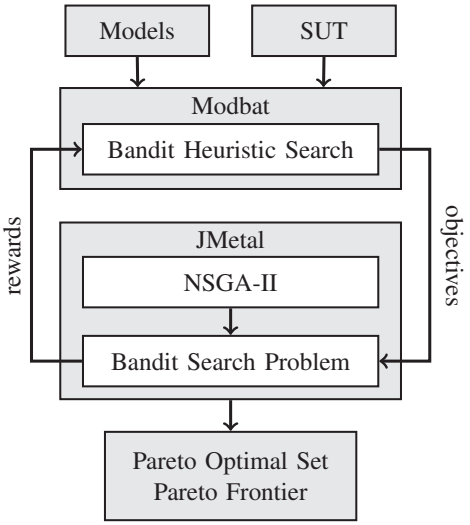

Fig. 1: Multi–objective bandit-search optimization.

Fig. 1 gives an overview of our implementation used to solve our multi–objective bandit search optimization problem for Modbat models with the aid of jMetal v5 [7] and NSGA-II. The working principle of jMetal is based on algorithms (such as NSGA-II) chosen by users and user-defined problems

to solve. Users need to first define their multi–objective optimization problems with objective functions, and then solve them with the chosen algorithm. We have implemented our multi–objective bandit search optimization problem in jMetal with our defined vector-valued objective function $\vec{f} : \mathcal{R}^8 \rightarrow \mathcal{O}$. Then we use the NSGA-II genetic algorithm provided by jMetal to solve this problem. The process goes through evaluation rounds of the NSGA-II algorithm with the number of rounds and population provided. For each evaluation round, we run different Modbat models in parallel using 8 different values of rewards (generated by the NSGA-II from jMetal) as the input parameters for our bandit heuristic search strategy. After the Modbat models are executed, the results of our defined four test adequacy criteria for all models are sent to jMetal as the values of the objectives which can then be used by the NSGA-II algorithm to generate reward values for the next evaluation round. When all evaluation rounds of the NSGA-II algorithm are finished, jMetal provides files containing the *Pareto optimal set* and the *Pareto frontier* found by the NSGA-II. The detailed configuration of our experiment will be discussed in Section IV.

IV. Experimental Evaluation

We have evaluated our bandit heuristic search strategy on a pre-existent collection of Modbat models which have earlier been used to generate test cases with the standard random approach provided by Modbat.

A. Experimental Setup

The pre-existent collection of models that we consider includes four simple models as a training set and two large and complex models as the test set. The simple models in the training set encompass the *ChooseTest* model [10], the *Java Server Socket* model [22], the *Java Array List* model [23], and the *Java Linked List* model [23]. The large complex models in the test set are the *ZooKeeper* [24] and *PostgreSQL* [25] models. The Java Array and Linked List models, PostgreSQL model, as well as the ZooKeeper model, consist of several parallel EFSMs, which are executed in an interleaved way [4]. Table I summarizes the total numbers of states, transitions, numbers of different EFSMs, and non-commenting lines of code (NCLOC) for each model. Note that states refer to labeled states, which in EFSMs are augmented with variables that may be from a potentially infinite-sized domain; therefore, the full number of *extended* model states is usually in thousands or millions per test. Moreover, in Table I, we summarize the *declared* states of all types of models involved, but we do not count states of multiple model instances in a given test multiple times. Likewise, transitions may include internal choices over different functions, or invoke functions that are arbitrarily complex. This means that the numbers in Table I only give a picture of the syntactic complexity of a model.

Specifically, we first apply our strategy on the four simple Modbat models in the training set using jMetal to optimize our strategy. Then, the weights of the eight rewards in the resulting Pareto optimal set are used in Modbat's configuration as the optimal parameters for our strategy on the two large complex Modbat models in the test set.

Table I: Number of declared states, transitions, EFSMs, and code size for each model for the evaluation

Model	States	Transitions	EFSM(s)	NCLOC
ChooseTest	3	3	1	10
JavaServerSocket	7	17	1	105
ArrayList	5	51	3	392
LinkedList	5	59	3	428
PostgreSQL	13	15	2	414
ZooKeeper	17	58	2	2225

The experiments have been performed using an Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-88-generic x86_64) on an Intel(R) Xeon(R) Gold 6136 CPU 3.00GHz (48 CPUs). For the experimental setup of Modbat, we configure that each Modbat model runs 40 test suites. We preconfigure the seed for the random number generator and fixed 40 seeds to make the test generation deterministic. Each test suite consists of 50 test cases.

To configure the NSGA-II algorithm provided in jMetal, we use its default settings except that the population size is set to 50, and the maximum number of generations in the evolutionary search is set to 100. The reason for using these two relatively small values is to keep the time used for the experimental evaluation manageable. Furthermore, for the optimization process, we configure Modbat and jMetal to run the four simple models in parallel. That is, for each evaluation, jMetal provide the values of 8 rewards as a parameter setting for Modbat to run four models in parallel (random values for 8 rewards as the initialization). The resulting values of the four test adequacy criteria of each model are then collected and used as objectives for jMetal to execute NSGA-II (16 objectives together in total due to 4 models). All collected resulting values of four test adequacy criteria are within 0 to 100, which are defined or computed as follows:

- State coverage: $Cov_s \in \{0, \ldots, 100\}$
- Transition coverage: $Cov_t \in \{0, \ldots, 100\}$
- Score of Cov_{lip}: $Cov_{lip} * 2$, $Cov_{lip} \in \{1, \ldots, 50\}$
- Score of $NTest_{fail1}$: $102 - NTest_{fail1} * 2$, $NTest_{fail1} \in \{1, \ldots, 51\}$

The two last scores are calculated based on the fact that with 50 tests, at most 50 linearly independent paths are possible, and that the best possible outcome is if the first test finds a failure; if no test finds a failure, we count the score as if the 51st test (which is never tried) would have found it.

As each parameter setting for 8 rewards was tested with 40 seeds and 50 test cases per seed on four models, we ran 8,000 tests per parameter setting to determine fitness. With a population size of 50 and 100 generations, we ran a total number of 40 million tests in the training phase, which took four days using a 48 CPUs cluster. For the ZooKeeper model, we just apply 50 optimal solution candidates in the Pareto optimal set directly and collect the results for the test adequacy criteria; while for the PostgreSQL, we perform mutation testing [26] using 86 mutants to inject 86 different

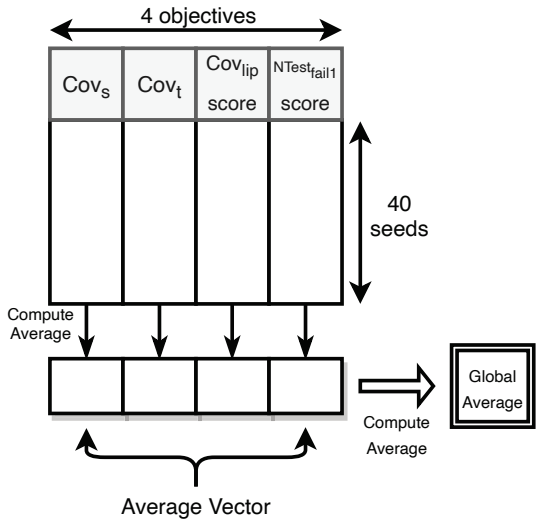

Fig. 2: Result computations for the random approach.

errors to the PostgreSQL. Then, we apply 50 optimal solution candidates to 86 mutated PostgreSQL, respectively.

B. Data Post-processing

We have compared the performance of our bandit heuristic search strategy and its optimization to the random approach already provided by Modbat. Fig. 2 shows the basic process of collecting and computing result data of the random approach for each model. We compute the average of collected resulting values for each objective obtained using the 40 fixed seeds to get an average result vector for the four objectives (Pareto frontier). Based on the average vector, we also compute a global average to indicate an overall result of the random approach. For the PostgreSQL model, since it has 86 mutated versions, we perform this process for each mutated version, respectively, to collect result data. Then, we have 86 resulting average vectors and 86 global averages for the 86 mutated versions of the PostgreSQL model.

Fig. 3 illustrates the basic process for each model to collect and compute results from the Pareto solution set obtained by our bandit heuristic search strategy and its optimization with jMetal. For each model, the Pareto solution set has 50 solution candidates, and each candidate has resulting values obtained for four objectives (Pareto frontier) using the 40 fixed seeds. Therefore, for each candidate, we compute the average result vector and the global average by applying the same process as for the random approach. Then, for the Pareto solution set, we have 50 average result vectors and 50 global averages in total. Also, for the PostgreSQL model, we first perform this process for its 86 mutated versions. For each mutated version, we then compute an global average result from the 50 average vectors, and an overall average result from the 50 global averages as the final result for this mutated version.

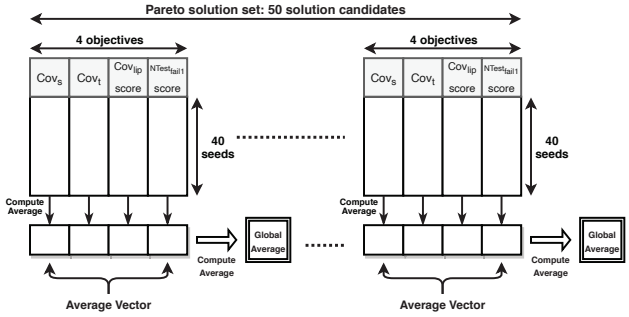

Fig. 3: Result computations for the bandit heuristic search approach

C. Analysis of Results

We visualize the result data collected using the processes discussed in Fig. 2 and Fig. 3 using box plots. Each box in the plot shows the range between the first and third quartiles (Q1 and Q3) as a rectangle, with a solid red line indicating the median value. The distance between Q1 and Q3 is the inter-quartile range (IQR); 25 % of the data lies below Q1, and 75 % of the data lies below Q3. The blue dashed lines indicate the smallest (largest) observed point from the dataset that falls within a distance of 1.5 times the IQR below Q1 (and above Q3, respectively). Circles indicate outliers that lie outside 1.5 times the IQR.

Fig. 4 and Fig. 5 shows the box plots for the Java Server Socket and Array List models of the training set. The result data are collected directly when jMetal finish all generations of the NSGA-II algorithm. The size of the resulting dataset to generate each box plot for the random approach is 40 (obtained from 40 seeds shown in Fig. 2), while for the heuristic approach the size of the dataset is 50 (obtained from 50 average vectors shown in Fig. 3).

From Fig. 4 and Fig. 5, we can observe that for the box plots of the Java Server Socket model, our bandit heuristic approach gives better results on the transition coverage Cov_t

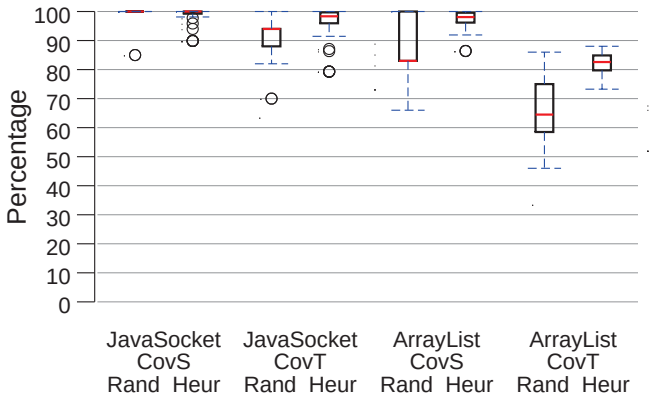

Fig. 4: Comparison of state and transition coverages for Java server socket and array list models

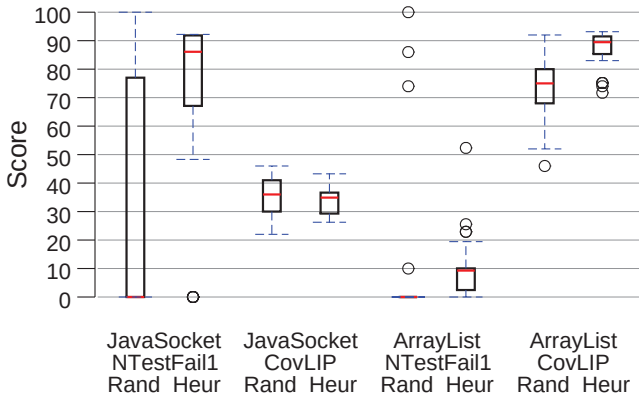

Fig. 5: Comparison of scores to number of test cases used to find the first failure and LIP coverage for Java server socket and array list models

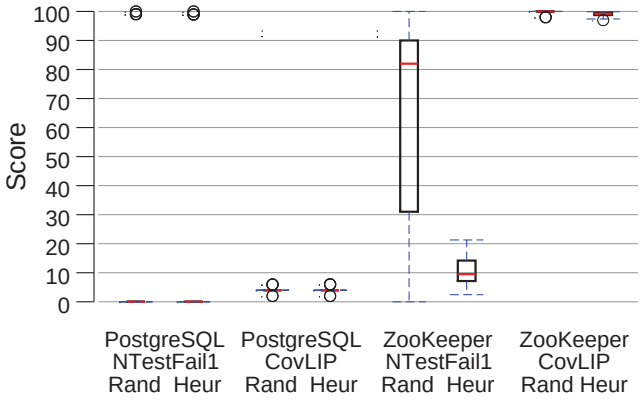

Fig. 7: Comparison of scores to number of test cases used to find the first failure and LIP coverage for PostgreSQL and ZooKeeper models

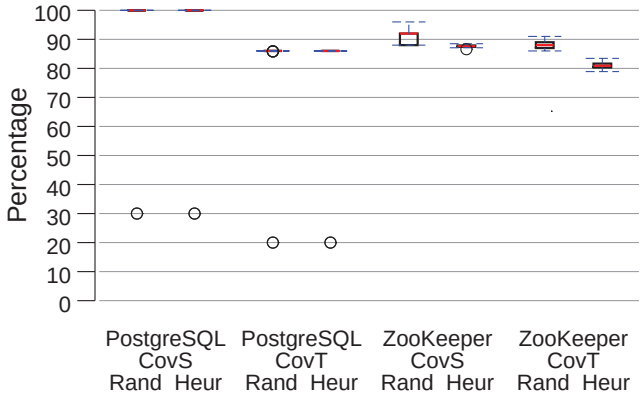

Fig. 6: Comparison of state and transition coverages for PostgreSQL and ZooKeeper models

and the score of $NTest_{fail1}$ compared to the random approach. Concerning state coverage Cov_s and the score of Cov_{lip}, the random approach is slightly better. From the box plots of the Array List model in Fig. 4 and Fig. 5, it can be observed that our bandit heuristic approach has better performance on all objectives in comparison to the random approach. The box plots for both the Linked List and ChooseTest models show that the heuristic approach has better results on all objectives, which is similar to the results of Array List model. Hence, we do not specifically discuss their box plots here due to the space limitations.

After we obtained the Pareto optimal set from the training phase of the four simple models in the training set, we apply the resulting values of eight different rewards in the Pareto optimal set to the 86 mutated PostgreSQL and ZooKeeper models, respectively, from the test set. Fig. 6 and Fig. 7 show the collected data for the PostgreSQL and ZooKeeper models as box plots. The size of the dataset is 86 for the PostgreSQL model with the heuristic approach, since we collected results from 86 mutated versions of the PostgreSQL model. For each version, we collect an average result for each objective from 50 average vectors as was shown in Fig. 3. For the random approach of the PostgreSQL model, the size of the dataset is also 86. This dataset has 86 average vectors, and each vector is computed from the results of 40 seeds as was shown in Fig. 2. For the ZooKeeper model, the sizes of datasets for the heuristic and random approaches are the same as the datasets for the four simple models in the training set.

From the box plots for the PostgreSQL model, we see that the heuristic approach is slightly better than random approach, since the box plot of the transition coverage Cov_t of the heuristic approach does not have the extra outlier (around 85) shown in the plot of the random approach in Fig. 6.

For the resulting box plots of the ZooKeeper model in Fig. 6 and Fig. 7, although the box plots of the random approach seems to have better results on the four objectives than the heuristic approach, the box plots also show that the distribution of the resulting data for heuristic approach is more concentrated than the random approach. For instances, the resulting box plot of the $NTest_{fail1}$ score for the random approach has some extremely bad results (0) and extremely good results (100), compared to the box plot of the heuristic approach. The box plots of the heuristic approach from other models also reflect this characteristic, i. e., they have a more concentrated distribution of their resulting data than random approach.

D. Discussion

Our box plots compare the performance between our bandit heuristic approach and a random approach for each objective separately. To compare all values at a glance, Table II shows the *global average* over all four metrics across all generated tests, as obtained by the process shown in Fig. 2 and Fig. 3. For PostgreSQL, we additonally average the fault-detection rate over 86 mutants [25] on the code.

For the training set, we can see a large improvement on the results both in the best and in the average case, which shows that our heuristic adapts well to four different types of models

Table II: Comparison of global averages obtained by heuristic and random approaches for each model.

Model	Heuristic		Random
	Max GA	Aver GA	GA
ChooseTest	76.81	61.11	50.80
JavaServerSocket	82.26	75.38	64.35
ArrayList	82.62	68.95	59.15
LinkedList	71.19	69.77	58.06
PostgreSQL	72.74	48.45	48.45
ZooKeeper	72.91	69.55	84.79

and produces consistent results. For the test set (PostgreSQL and ZooKeeper), the difference is less clear. The heuristic approach for PostgreSQL has better transition coverage, but the difference is not significant, and the average scores even match up to two digits after the decimal point. For ZooKeeper, the random search does better than the heuristic approach.

Therefore, the results of the bandit heuristic search for the test set are not as good as the results from the training set. The reason for this are that our training set is too small, resulting in overfitting. Even so, the results of the PostgreSQL model from the test set also show a potential for the heuristic approach to perform much better than the random approach.

V. Related Work

A. Test Generation with Multi-objective Optimization

Test generation related to multi-objective optimization using Pareto-effective approaches have been developed in the existing literature. Oster and Saglietti [27] proposed a technique for test data generation using multi-objective optimization and evolutionary algorithms to handle two objectives including data flow coverage and the number of test cases required. They used two variants of genetic algorithms, including the Multi-Objective Aggregation (MOA) and classical NSGA to compare with a random approach and a simulated annealing algorithm, in order to test object-oriented programs implemented in Java. The results showed that simulated annealing outperformed other algorithms, but NSGA offered a higher flexibility since it can provide a complete solution set instead of a single optimal result.

A multi-objective approach to test data generation was presented by Harman et al. [28], which focused on applying multi-objective optimization to branch coverage and generating branch adequate test sets for branch adequate testing. Two objectives were considered by the authors, including branch coverage and dynamic memory consumption. The authors compared the effectiveness of three search approaches: a random, a weighted genetic algorithm search, and the NSGA-II. The case studies were carried out on testing C code from both real-world and synthetic examples.

In this paper, we define four test adequacy criteria as objectives. Instead of branch coverage, we consider path coverage as one of our test adequacy criteria, since path coverage is a stronger test adequacy criterion than branch coverage and it concerns a sequence of branch decisions instead of only one branch at a time. Also, our approach is based on models of the system under test. Path coverage and other criteria are optimized at the model level rather than coverage of the SUT code. We notice that the process of test case generation faces the exploration versus exploitation dilemma, so we propose the heuristic search strategy to handle this dilemma and guide the test case generation. We then use the jMetal framework with the genetic algorithm NSGA-II to tune the strategy in order to optimize the four objectives we defined, with respect to their trade-offs. For the experimental evaluation, we also compare our approach with a random approach.

B. Search-based Test Generation

Our bandit-based heuristic approach also relates to some extent to work on search-based testing, where random testing is augmented with heuristics to find fault-revealing test cases more efficiently [29]. In random testing, the problem of choosing suitable input with the right values and types exists; these problems are taken care of in model-based testing because the user provides a model that generates these inputs. Similarities exist in three of the six heuristics used in Guided Random Testing [29]: 1) Impurity: We use a different weight for self-loop transitions, which contain at least some impure methods as not to be completely redundant; 2) Bloodhound: We also choose transitions based on coverage, but we use coverage at the model level rather than coverage of the SUT code; 3) Orienteering: At this point, we do not consider the time it takes to execute a transition, because the execution times of transition actions did not differ in major ways in our examples.

In addition to using Pareto-efficient approaches for search-based test generation, Salahirad et al. [30] discussed different fitness functions for white-box testing. Their findings confirm that high (source code) coverage is a prerequisite for successful fault detection, and that branch coverage stands out as the most effective single criterion. They also used *treatment learning* to discover which metrics best predicts fault detection. We have not investigated how different subsets of our criteria (especially when used within a limited resource budget) compare to each other, as we only have four, and hence much less than they had to consider. Rojas et al. [31] found that multi-objective optimization algorithms based on Pareto dominance are less suitable than a linear combination of the different non-conflicting objectives. It is not obvious to us how we would prioritize weights among the four different objectives, a question which also [31] left for their future work.

C. Test Section with Multi-objective Optimization

Related work also exists in multi-objective optimization for test selection. Yoo and Harman [32] presented a multi-objective formulation of the regression test case selection problem to show how multiple objectives can be optimized using a Pareto efficient approach. Their goal was to find a subset of a test suite, which is a Pareto optimal set with respect to the chosen test criteria. They gave three algorithms to compare their effectiveness, including a single-objective greedy algorithm, NSGA-II, and vNSGA-II. The evaluation was carried out on five programs in the Siemens suite and a

program from the European Space Agency. For each program, the authors randomly selected test suites from existing available test suites as the input to the multi-objective optimization process. The results showed that the NSGA-based approaches can out-perform the greedy approach.

Mondal et al. [33] studied multi-objective test case selection to analyze both coverage-based and diversity-based test case selection approaches. They proposed two approaches for bi- and three-objectives optimization, respectively, by considering code coverage, test case diversity, and test execution time. They used the Additional-Greedy and NSGA-II algorithms for bi-objective optimization and NSGA-II only for three-objective optimization on 16 versions of five real-world programs, such as JBoss and Apache Ant. The results showed an improvement of the fault detection rate by the three-objective optimization approach.

For our work, we do not have existing available test suites, so we focus on using the bandit-based heuristic search to generate optimal test cases directly, with the aid of the Pareto-efficient approach to optimize the four test adequacy criteria. Also, instead of considering test execution time as an objective, we use the number of test cases to find the first failure together with three different coverages as objectives.

VI. Conclusions and Future Work

Our main contribution is a heuristic search based test case generation approach for model-based testing, aiming at performing well on test adequacy criteria with considering their trade-offs. We have proposed a bandit-based heuristic search strategy to handle the exploration versus exploitation dilemma for test case generation. Then, we applied an optimization technique to tune our strategy and optimize the chosen test adequacy criteria with the aid of the jMetal multi-objective optimization framework and the NSGA-II Pareto-efficient multi-objective genetic algorithm. We have evaluated our approach on several models for the Modbat model-based testing tool by comparing the results of the bandit-based heuristic search with a random test case generation approach. Our experiments show that our bandit-based heuristic search approach has potential to obtain better and more consistent results on the chosen adequacy criteria compared to random test case generation, while considering the trade-offs of the test adequacy criteria.

The second contribution is an implementation of the bandit-based heuristic search strategy in the Modbat. This implementation is now included as a new feature in the Modbat 3.4 release. We have defined test adequacy criteria as multi-objectives so that Modbat implements our strategy for test case generation in addition to its standard random search. The test adequacy criteria are optimized using the jMetal framework which applies the NSGA-II algorithm on the Modbat models that we use as the training set to find a Pareto optimal set. The reward parameter settings obtained in the Pareto optimal set can then be used to initialize the test case generation with our bandit-based heuristic search strategy to generate test cases for advanced Modbat models in general and targeting the chosen test adequacy criteria.

The work presented in this paper opens up several directions of future work. The results of our heuristic search from the training set are promising, but the results from the test set are not as good, especially for the ZooKeeper model, due to overfitting. To leverage the potential of our approach, we need to obtain more and more diverse models for the training set, especially in case of large-scale and complex systems, so that multi-objective optimization can get well-fitted reward parameter settings in Pareto optimal set. Additionally, increasing the size of the population and the number of generations for the NSGA-II might give us better results in the Pareto optimal set. Also, we may consider using alternative algorithms provided by jMetal to solve our multi-objective optimization problem.

Another direction of future work is to investigate self-optimizing approaches with optimization on the models at runtime to achieve the potential of our bandit heuristic search strategy for test case generation. Furthermore, other test adequacy criteria such as the execution time of test cases could be considered as additional objectives for the optimization. Also, in addition to bandit heuristic search strategy, we plan to implement other heuristic strategies for test case generation. Finally, the application of our approach to platforms other than Modbat is another possibility.

Acknowledgements

This work was partially supported by the European Horizon 2020 project COEMS under grant agreement no. 732016 (https://www.coems.eu/).

References

[1] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Software testing, verification and reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.

[3] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II," in *International conference on parallel problem solving from nature*. Springer, 2000, pp. 849–858.

[4] C. Artho, A. Biere, M. Hagiya, E. Platon, M. Seidl, Y. Tanabe, and M. Yamamoto, "Modbat: A model-based API tester for event-driven systems," in *Haifa Verification Conference*, ser. Lecture Notes in Computer Science, vol. 8244. Springer, 2013, pp. 112–128.

[5] "Modbat 3.4," https://github.com/cyrille-artho/modbat/tree/3.4.

[6] J. J. Durillo and A. J. Nebro, "jMetal: A Java framework for multi-objective optimization," *Advances in Engineering Software*, vol. 42, no. 10, pp. 760 – 771, 2011.

[7] A. J. Nebro, J. J. Durillo, and M. Vergne, "Redesigning the JMetal multi-objective optimization framework," in *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO Companion '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 1093–1100.

[8] K.-T. Cheng and A. S. Krishnakumar, "Automatic functional test generation using the extended finite state machine model," in *30th ACM/IEEE Design Automation Conference*. IEEE, 1993, pp. 86–91.

[9] Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne, "The Scala Programming Language," https://www.scala-lang.org.

[10] R. Wang, C. Artho, L. M. Kristensen, and V. Stolz, "Visualization and abstractions for execution paths in model-based software testing," in *Integrated Formal Methods*, ser. Lecture Notes in Computer Science, W. Ahrendt and S. L. Tapia Tarifa, Eds., vol. 11918. Springer, 2019, pp. 474–492.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press, 2011.

[12] D. A. Berry and B. Fristedt, "Bandit problems: sequential allocation of experiments (monographs on statistics and applied probability)," *London: Chapman and Hall*, vol. 5, pp. 71–87, 1985.

[13] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. on Comput. Intelligence and AI in games*, vol. 4, no. 1, pp. 1–43, 2012.

[14] V. Kuleshov and D. Precup, "Algorithms for multi-armed bandit problems," *arXiv preprint arXiv:1402.6028*, 2014.

[15] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[16] M. Tokic and G. Palm, "Value-difference based exploration: adaptive control between epsilon-greedy and softmax," in *Annual Conference on Artificial Intelligence*. Springer, 2011, pp. 335–346.

[17] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in computer Go," *Artificial Intelligence*, vol. 175, no. 11, pp. 1856–1875, 2011.

[18] T. L. Lai and H. Robbins, "Asymptotically efficient adaptive allocation rules," *Advances in applied mathematics*, vol. 6, no. 1, pp. 4–22, 1985.

[19] Y. Collette and P. Siarry, *Multiobjective optimization: principles and case studies*. Springer Science & Business Media, 2013.

[20] M. Ehrgott, *Multicriteria optimization*. Springer Science & Business Media, 2005.

[21] C. A. C. Coello, G. B. Lamont, D. A. Van Veldhuizen *et al.*, *Evolutionary algorithms for solving multi-objective problems*. Springer, 2007.

[22] C. Artho and G. Rousset, "Model-based testing of the Java network API," *arXiv preprint arXiv:1703.07034*, 2017.

[23] C. Artho, M. Seidl, Q. Gros, E. Choi, T. Kitamura, A. Mori, R. Ramler, and Y. Yamagata, "Model-based testing of stateful APIs with Modbat," in *Proc. 30th Intl. Conf. on Automated Software Engineering (ASE 2015)*. Lincoln, USA: IEEE, Nov 2015, pp. 858–863.

[24] C. Artho, Q. Gros, G. Rousset, K. Banzai, L. Ma, T. Kitamura, M. Hagiya, Y. Tanabe, and M. Yamamoto, "Model-based API testing of Apache ZooKeeper," in *2017 IEEE Intl. Conf. on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 288–298.

[25] D. Tziatzios, "Model-based testing for SQL databases," Master's thesis, KTH, School of Electrical Engineering and Computer Science (EECS), 2019.

[26] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[27] N. Oster and F. Saglietti, "Automatic test data generation by multi-objective optimisation," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2006, pp. 426–438.

[28] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *Proc. of the 9th annual conference on Genetic and evolutionary computation*. ACM, 2007, pp. 1098–1105.

[29] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler, "GRT: program-analysis-guided random testing," in *Proc. 30th Intl. Conf. on Automated Software Engineering (ASE 2015)*. Lincoln, USA: IEEE, Nov 2015, pp. 212–223.

[30] A. Salahirad, H. Almulla, and G. Gay, "Choosing the fitness function for the job: Automated generation of test suites that detect real faults," *Softw. Test., Verif. Reliab.*, vol. 29, no. 4-5, 2019.

[31] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, and A. Arcuri, "Combining multiple coverage criteria in search-based unit test generation," in *Search-Based Software Engineering*, M. Barros and Y. Labiche, Eds. Springer, 2015, pp. 93–108.

[32] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*, 2007, pp. 140–150.

[33] D. Mondal, H. Hemmati, and S. Durocher, "Exploring test suite diversification and code coverage in multi-objective test case selection," in *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2015, pp. 1–10.