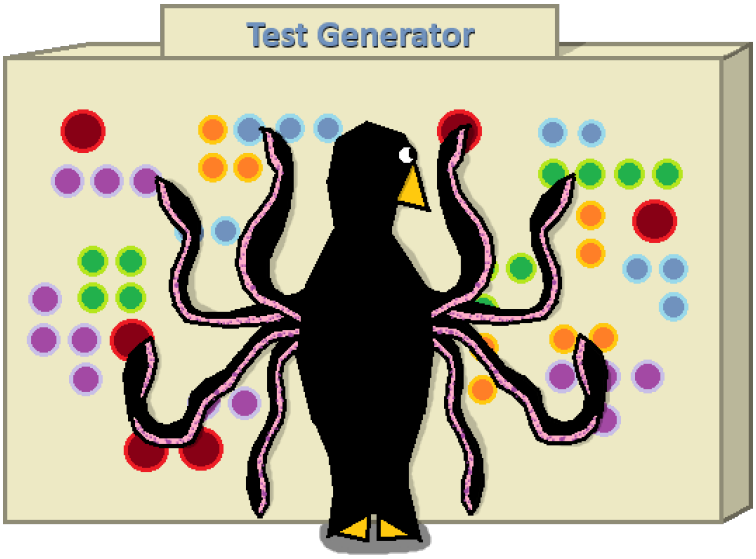
CHALMERS
UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG



Adaptive Parameter Control for Search-Based Unit Test Generation

An exploratory study of parameter control for the unit test generation framework Pynguin

Master's thesis in Computer science and engineering

ERIK BLOMBERG
HENRIK JOHANSSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Adaptive Parameter Control for Search-Based Unit Test Generation

An exploratory study of parameter control for the unit test generation framework Pynguin

ERIK BLOMBERG
HENRIK JOHANSSON

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Adaptive Parameter Control for Search-Based Unit Test Generation
An exploratory study of parameter control for the unit test generation framework Pynguin
ERIK BLOMBERG
HENRIK JOHANSSON

Supervisor: Afonso Fontes, Computer Science and Engineering
Supervisor: Gregory Gay, Computer Science and Engineering
Examiner: Richard Torkar, Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Penguin standing in front of a Test Generator machine controlling parameters by pressing buttons with its eight octopus arms.

Typeset in LaTeX
Gothenburg, Sweden 2024

Adaptive Parameter Control for Search-Based Unit Test Generation
An exploratory study of parameter control for the unit test generation framework Pynguin

ERIK BLOMBERG
HENRIK JOHANSSON
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Testing is a crucial task in software engineering, but one that is time-consuming and expensive. Test generation frameworks, such as Pynguin, are designed to automate the process of creating tests, thus alleviating some of the cost. However, the test suites generated by Pynguin do not always manage to reach 100% code coverage, thus indicating that there is room for improvement. This thesis aimed to explore whether adding reinforcement learning-based parameter control to update parameter values during the test generation process could yield improved code coverage.
We created PynguinAPC, a version of Pynguin with the addition of reinforcement learning-based parameter control. To evaluate the impact of parameter control, we performed experimental simulation in two cycles, one where we applied control to individual parameters and one where we applied control to pairs of two parameters for a set of 24 Python modules. The results were analyzed using Bayesian statistical models, concluding that there was no overall gain for the final branch coverage achieved by the generated test suites and the branch coverage growth rate from the application of parameter control. The addition of a parameter control system resulted in an overall increase in the performance overhead. However, some modules were more receptive to parameter control for specific parameters, warranting an investigation of what traits these modules inhabit that caused this increased receptiveness. Paradoxically, by introducing a system to reduce the amount of necessary manual configuration, the system inadvertently introduced additional layers of configuration. These configurable layers could merit further exploration into their impact, as the choices made for this study may be a limiting factor in the results observed.

Keywords: Software Testing, Search-Based Unit Test Generation, Search-Based Software Engineering, Parameter Control, Reinforcement Learning, Pynguin, Genetic Algorithms, Bayesian Inference, Bayesian Statistics

Acknowledgments

We want to thank our supervisors Afonso Fontes and Gregory Gay for their never-ending support and guidance during our Master's thesis project.

Computing resources were provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS), partially funded by Vetenskapsrådet grant agreement 2022-06725.

Erik Blomberg & Henrik Johansson, Gothenburg, July 2024

Contents

List of Figures

List of Tables

1

Introduction

In software development, tests play a crucial role in ensuring that software is reliable and maintainable [1], [2]. Software testing generally boils down to ensuring that a software artifact behaves as expected by attempting to identify any incorrect behavior [1], [3]–[5]. A consequence of writing tests is the cost associated with it [1]–[3]. Dedicating time to design and write tests consumes resources that could have been spent on more direct ways of generating value, such as implementing features into an application [6]. Even a slight decrease in the time, effort, or resource usage when creating tests, while retaining the quality of the resulting tests, could result in major benefits to the software industry [7].

A possible way of decreasing the time and effort required to create tests is by automating the process, which is an active area of research [2], [8]. Within this research area, there exist many different test generation frameworks, such as Pynguin [9], EvoSuite [10], and Randoop [11]. Test generation frameworks generally target code coverage to assess whether the generated tests are adequate. Most test generation frameworks have a large number of available parameters, and manually changing these to find the optimal configuration for a specific code base quickly becomes infeasible [12]. An area that has not been explored in great detail is the idea that there exists an opportunity to improve these test generation frameworks by automating the process of parameter value selection [4].

In this study, we extended a test generation framework with parameter control [13], by using reinforcement learning to adaptively change parameter values during the test generation process. Additionally, we evaluated our extension to explore the impacts of parameter control on the output and behavior of the test generation framework. The evaluation included whether the extended framework achieved higher coverage numbers with the generated test suites, whether there was a difference in the rate of coverage increase during the test generation process, and whether the added overhead of parameter control is sufficiently compensated for with improved results.

1.1 Problem Description

The idea of automating the process of test creation is not new and has been actively studied for decades [2], [14]. Within automated test generation, there are two main areas, *test input generation* and *test oracle generation* [4]. Test input generation refers to generating test cases and their respective input data, which can be in the form of numerical values, text, or actions to perform on a user interface, etc. [4], [6]. While, test oracle generation refers to generating the assertions that are used by tests to establish whether they pass or fail [4], [15].

This study specifically considers Pynguin, an open-source test suite generation framework for code modules written in the Python programming language [9]. Pynguin performs both test input generation and test oracle generation. For the test input generation, Pynguin utilizes test case generation algorithms that are openly available, while its test oracle generation is based on mutation testing [9]. In the paper where the Pynguin framework is presented, it is also evaluated. The evaluation consisted of running Pynguin using a few different test generation algorithms, i.e., *DynaMOSA*, *MIO*, *MOSA*, *Random*, *WholeSuite*, and *WholeSuite with archiving*. The results of the evaluation showed that, on average, all algorithms performed quite similarly, quickly surpassing 60% coverage before plateauing. These results indicate that there is room for improvement by, e.g., increasing the final branch coverage achieved or raising the rate at which branch coverage is gained during the test generation process.

An inherent challenge when using test generation frameworks, such as Pynguin, is the large number of parameters available to configure. Pynguin itself has over 100 parameters, out of which, over 20 directly impact the test generation algorithms, and thus, the test generation. While Python modules are written in the same language, they will vary in their structure and characteristics, and thus, it would be contradictory to assume that the same parameter configuration would lead to the optimal results for all possible modules [1]. Therefore, the goal should be to adapt the parameter configuration to the specific module under test. However, attempting to manually identify which parameter values are optimal for each specific module under test is generally not feasible. The reason for this infeasibility is that there are no established ways of identifying whether a parameter configuration is sufficient, except for running multiple different configurations and comparing the results, which is very time- and resource-consuming. Thus, the additional evaluation required to ensure improvements would likely negate any gains made by the adapted parameter values, which is further exacerbated by the likely scenario of a module being modified which would require further adaptation.

One way to attempt to overcome this challenge is to automate the process of parameter value selection. For parameter value selection there are two main strategies, *parameter tuning* and *parameter control* [13]. Parameter tuning refers to the practice of picking parameter values before execution which are then kept static. Parameter control, on the other hand, refers to the practice of automatically adapting the

parameter values during execution. Test generation in Pynguin is an evolutionary process where tests are generated iteratively, meaning that each new generation of tests is an evolution of the last. Thus, the optimal parameter configuration may evolve and change as the process progresses. Therefore, this study focuses on parameter control.

1.2 Purpose of the Study

In this study, we extended the test generation framework Pynguin with the addition of a parameter control layer, which resulted in the creation of PynguinAPC, Pynguin with Adaptive Parameter Control. The parameter control layer uses Reinforcement Learning (RL) to automate the process of updating parameter values during the test generation process. RL learns by interacting with an environment [16], which allows it to interact with and learn from Pynguin during the test generation process. Additionally, RL requires no prior training, alleviating the need for training data. If a technique that require prior training was used instead, it would limit the set of possible modules under test to those that were used for training, or possibly modules of similar characteristics as those trained on, limiting the generalizability of the tool.

The study aims to explore what potential gains can be achieved by the addition of parameter control to a test generation framework. More specifically, PynguinAPC was evaluated to investigate whether the inclusion of automated parameter control...

- ... affects the final coverage achieved by the generated test suite.
- ... affects the rate of coverage gained during the test generation process.
- ... is justifiable in relation to the added overhead.

1.3 Significance of the Study

By utilizing adaptive parameter control, the study aims to improve the process of automated test generation, e.g., by modifying the test generation process to be adaptable to the modules under test rather than the current rigid implementation. These improvements can be in the form of increasing the code coverage achieved by the test generation framework or decreasing the time it takes to reach a certain code coverage.

From a practitioner's perspective, these improvements can be beneficial since they allow for less time to be spent on manually creating tests or adapting a test generation framework to achieve adequate code coverage for different code bases. As a consequence, developers can focus more of their time on work that directly contributes to creating value for customers or other stakeholders. Similarly, from the perspective of researchers, the results from this study could aid in identifying close-to-optimal test generation framework configurations. Parameter control is a quite

well-explored research area, especially for evolutionary algorithms, both general ones and ones adapted to specific problems, e.g., the traveling salesperson problem [17], [18]. However, the application of parameter control to test generation frameworks specifically, is still a largely unexplored topic. Thus, this study can bring insights into whether the application of parameter control to test generation frameworks is an area that warrants further research.

1.4 Thesis Outline

Chapter 2, Background Provides prerequisite information and terms for the rest of the thesis.

Chapter 3, Related Work Presents relevant studies and positions this study.

Chapter 4, Implementation Presents the implementation and the choices made for the version of Pynguin with adaptive parameter control.

Chapter 5, Method Presents the research questions explored in this study and the methods used.

Chapter 6, Results Present the results of the study.

Chapter 7, Discussion Presents a discussion of the study's results, some exploration of possible reasons why, and identified threats to the validity of the study.

Chapter 8, Conclusion Presents conclusions drawn from the results of the study.

2

Background

This chapter will provide the background information required to follow all concepts, terms, and technologies utilized for this study.

2.1 Software Testing

Software testing is the practice of assessing whether a piece of software is behaving as expected [3]. There are many different types of tests, some verify that a component, or the interplay between components, works as intended, while others focus on the user interface or the product as a whole. This study focuses purely on *unit testing*.

Unit testing is the smallest form of testing granularity where individual components (e.g., classes) or pieces of code (e.g., methods) are tested to determine if they work as expected for a system under test (SUT) [19]. Unit tests are usually written as *test cases*, which consist of a sequence of interactions with the SUT that confirms whether the observed outcome produced is as intended [1], [4], [19]. A *test suite* is a collection of test cases [17]. An example of a test case is illustrated in Figure 2.1, where we have a simple `Calculator` class, with an `add()` method. We define a test case called `test_addition()` which asserts that the resulting answer from `add()` is 4 when the input is 2 and 2.

```
1  class Calculator:
2      def add(self, a, b):
3          return a + b
4
5  # Test Case
6  def test_addition():
7      calc = Calculator()
8      answer = calc.add(2, 2)
9      assert answer == 4
```

Figure 2.1: An example of a Python class and an associated test case.

2.2 Code Coverage

Many code coverage criteria exist that focus on different aspects of the source code. However, all kinds of code coverage follow the same basic idea; to provide a metric

that measures the fraction of a specific aspect of the source code covered by a test case, or a test suite [17], [20]. Code coverage provides direct feedback about what parts are covered, as well as providing information on when to stop testing. Furthermore, code coverage could also increase the probability that faults in the source code are exposed, however, a majority of faults do still go undetected by purely relying on coverage alone [20].

Code coverage is a widely supported and used metric within software testing. It is also particularly well-suited for test generation as it can be measured automatically without human involvement. For this study, we will only consider one of the most common alternatives of code coverage: *branch* coverage.

```
1  def discount_percent(member:bool, age:int, student:bool):
2      discount = 0
3
4      if member:                          # all members have a 20% discount
5          discount = 0.2
6      else:                               # non-members also get some discounts
7          if age > 65:                    # retirees have a 10% discount
8              discount = 0.1
9          elif age < 20 or student:       # students have a 5% discount
10             discount = 0.05
11
12     return discount
13
14 # Test Case
15 def test_discount_percent():
16     factor = discount_percent(member=False, age=63, student=True)
17     assert factor == 0.05
```

Figure 2.2: Code to illustrate the concept of branch coverage with a single method and a corresponding test case.

Branch coverage considers whether all possible outcomes from control-altering decisions have been explored [20]. Control-altering decisions take place at e.g., `if`-statements, `for`-loops, and `switch`-cases, where the program can end up taking different paths, i.e., branches, through the code depending on what the statement evaluates to, e.g., `True` or `False`.

Consider the example method, `discount_percent`, and the corresponding test case in Figure 2.2. The method has three control-altering decisions (lines 4, 7, and 9), resulting in the existence of six different branches. In the test case, on line 15, the `discount_percent` method is called with the arguments `False`, `63`, and `True` to test the method for a 63-year-old student who is not a member, which should result in a discount percentage of `0.05` as shown on line 17. The first control-altering decision within the method takes place on line 4, where it checks whether the person is a member, which in this case evaluates to `False`, causing us to move to the next control-altering decision on line 7. Line 7, in turn, evaluates whether the person is a retiree, which again, evaluates to `False`. Since the control-altering decision on line 7 evaluates to `False`, the test moves on to line 9 which evaluates to `True`,

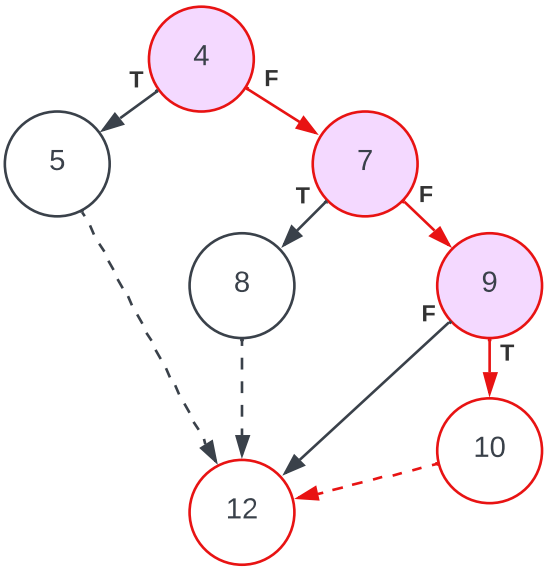


Figure 2.3: Control flow graph (CFG) representing the control flow of the method presented in Figure 2.2.

since the person is a student. Following line 9, there are no more control-altering decisions, and as such, the execution moves to line 12 where it returns the resulting discount percentage. The test case explored three out of the six branches in the code, resulting in a branch coverage of 50%.

This example can also be visualized as a control flow graph (CFG), illustrated in Figure 2.3 [6], [21]. All nodes in the CFG represent the line of code with the same number from the code snippet. Pink nodes represent the specific lines of code that contain control-altering decisions, and the solid outgoing arrows portray the branches that can be chosen depending on whether the control-altering statement evaluates to `True` (T) or `False` (F). Nodes that are not control-altering decisions are followed by dashed arrows to visualize that there is only one possible step to take. From the example test in Figure 2.2, the path through the CFG would be *4, 7, 9, 10, 12*, as indicated by the nodes and arrows colored in red.

2.3 Pynguin

Unlike the examples in Figures 2.1 and 2.2, unit tests are usually not so simple and require more thought and careful design. The effort required to produce effective tests is a major reason why testing can become expensive [1], [2]. This process can be simplified by the use of frameworks to automate the process of generating tests. One such framework is Pynguin [9]—PYthoN General UnIt test geNerator—the basis for this study.

Figure 2.4 presents an overview of the core steps Pynguin takes when generating tests. The image has been adapted from the original paper by Lukasczyk *et al.* [9]

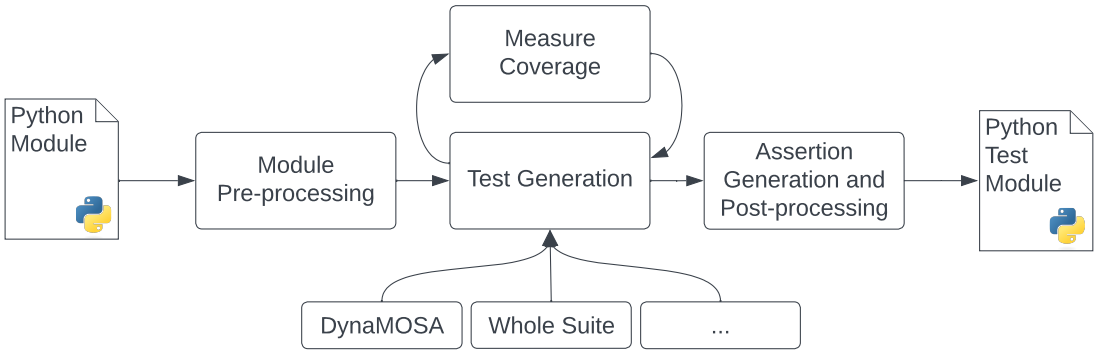


Figure 2.4: A flowchart representing the core steps and components of the Pynguin workflow.

and simplified to remove any superfluous content. The process starts with any arbitrary Python module, usually a `.py` file, that undergoes extraction and analysis of methods, classes, types, functions, and any modules/libraries it is dependent on. Using the extracted information, Pynguin deploys a test generation algorithm selected by the user (e.g., DynaMOSA, Random, or MIO, etc), to build test cases. The generated test cases undergo coverage analysis to understand what parts of the code are currently being tested and what parts are missing. The test generation algorithm will iteratively generate test cases until it reaches the desired coverage goal (usually 100%) or when the resource budget is depleted (e.g., time, iterations, etc). Finally, Pynguin generates assertions for the final set of test cases and exports the result to a human-readable Python test module.

2.4 Search-Based Test Generation

Test generation algorithms are located at the heart of Pynguin. Different test generation algorithms deploy varying strategies to create tests, e.g., some focus on the code structure, some attempt to estimate a model of the system under test to generate tests against, while others try a more random approach [2]. Pynguin offers 5 test generation algorithms, namely: *DynaMOSA* [21], *MIO* [22], *MOSA* [23], *Random* [9], and *WholeSuite* [5]. All of these, except *Random*, stems from a track of test generation algorithms called *Search-Based Software Testing* (SBST) [1], [2], [17], [24]. SBST frames the question of test generation as a search problem where we have one or more *fitness function* that guides a *meta-heuristic* algorithm to efficiently sample the search space and build test cases [1], [17].

2.4.1 Fitness Function

The core idea behind a fitness function is to be able to score our generated solution based on how close it is to some defined goal(s) [1], [17], [25]. In Pynguin, the only goal that is supported by all test generation algorithms is optimizing towards branch coverage [25], i.e., to find a test suite that covers the most branches (see Section 2.2

for more details on branches). One of the test generation algorithms, WholeSuite, generates multiple test suites and scores them using the following fitness function [5], [21]:

$$\min f_B(T) = |M| - |M_T| + \sum_{b \in B} d(b, T) \tag{2.1}$$

Dissecting this equation, $f_B(T)$ is the fitness function for a set of branches $B = \{b_1, b_2, \dots\}$ present in a system under test (SUT). $|M| - |M_T|$, is the difference between the total number of methods in the SUT and the total number of executed methods by the test suite T. The function $d(b, T)$ calculates how close a branch b is from being executed by a test suite T, known as *branch distance*. Take for example the statement `if x > 23`. If we already have explored the branch when the statement evaluates to `False`, then we also need to access the branch when the statement evaluates to `True`, this is where branch distance can be of help. Branch distance will quantify how close our input x is from evaluating the desired predicate (in our case `True`). For this example, an input of `x = 22` will result in a lower branch distance than `x = 10`. Pynguin has its own mechanism for evaluating branch distance for any arbitrary Python object or type [25]. Returning to the fitness equation, $\sum_{b \in B} d(b, T)$ sums together all the branch distances d to all the branches in B. Putting all of this together, the fitness function attempts to identify a test suite that minimizes the number of unexecuted methods and the distance to any unexplored branches.

Unlike WholeSuite, which is a single objective optimization algorithm, meaning that all goals are combined into one, all other test generation algorithms, i.e., DynaMOSA, MOSA, and MIO are multi-objective optimization algorithms, where all goals are kept separate [21], [25]. These test generation algorithms consider each branch a distinct objective, thus, each test case has one fitness score per branch, resulting in every test case having a fitness vector rather than a single fitness score [21].

$$\begin{bmatrix} \min f_1(t) = \delta(b_1, t), \\ \min f_2(t) = \delta(b_2, t), \\ \vdots \\ \min f_k(t) = \delta(b_k, t) \end{bmatrix} \tag{2.2}$$

Equation 2.2, defines a fitness vector for a test case t. For each branch $\{b_1, b_2, \dots, b_k\}$ in the SUT, we estimate how close t is to exploring that particular branch. In this case, $\delta(b, t)$ is a heuristic function that combines both branch distance and *approach level*, see Equation 2.3.

$$f_b(t) = \delta(b, t) = al(b, t) + d_y(b, t) \tag{2.3}$$

Approach level $al(b, t)$ measures how many control-altering statements are currently separating branch b from being executed by test case t. This is especially relevant when an `if`-statement is nested within other control-altering statements, thus dependent on their evaluations. The first control-altering statement that is accessible but evaluates to an undesired predicate (in relation to a specific branch), is known as the *node of diversion*, denoted as y [24]. The branch distance for a branch b is always evaluated at y. Putting these heuristics together, the result is a successful guiding mechanism for branch coverage. Approach level tells the optimization algorithm how far away a test case t is from branch b in terms of control-altering statements. Branch distance tells the algorithm how far away the statement at the node of diversion y is from obtaining the desired predicate to get closer to branch b.

An important distinction is that the aforementioned fitness vectors are not used "as-is" to compare test cases. In the case of MOSA and DynaMOSA, the fitness vectors are used as a computational foundation to assign a Rank to each individual (test case) in the population [21], [23]. The closest individuals to one or more objective(s) receive the best Rank and are, therefore, more likely to be further evolved and explored as potential solutions.

2.4.2 Meta-Heuristic (Genetic Algorithm)

After defining one or multiple fitness functions that capture how close a generated solution is to some obtainable goal, the meta-heuristic uses that information to guide the test generation process [1], [17]. This report will focus exclusively on meta-heuristics known as genetic algorithms (GA) [26] since these are the basis for all previously mentioned search-based algorithms implemented in Pynguin. Genetic algorithms try to mimic the phenomena of neutral selection by allowing the "fittest" individuals to breed, mix chromosomes, and mutate, forming new generations of possible solutions [17], [26]. While the implementation of the specific GA is varied across the search-based algorithms, their core steps remain similar. These steps will be the focus of this section and can be seen in Figure 2.5.

It all starts with a *random population* consisting of individuals. An individual in test generation is either a test suite (WholeSuite) or a test case (DynaMOSA, MOSA, and MIO). The initial *random population* becomes the first *current generation*, after which, the branch coverage criterion is used to determine whether the search should continue. Unless it is a very simple system under test (SUT), additional generations of tests need to be created to cover the entire SUT.

To form the next generation, individuals from the current generation are paired up to create offspring. Selecting these individuals is the job of a *selection function*, commonly Tournament Selection [17], [21], [24]. Tournament selection picks out n random individuals from the current generation, the "fittest" individual out of these n is declared the winner, and is paired up with the winner of another tournament [24]. Determining the fittest individual is the idea behind fitness functions (see Section 2.4.1). Algorithms like DynaMOSA and MOSA, consider something

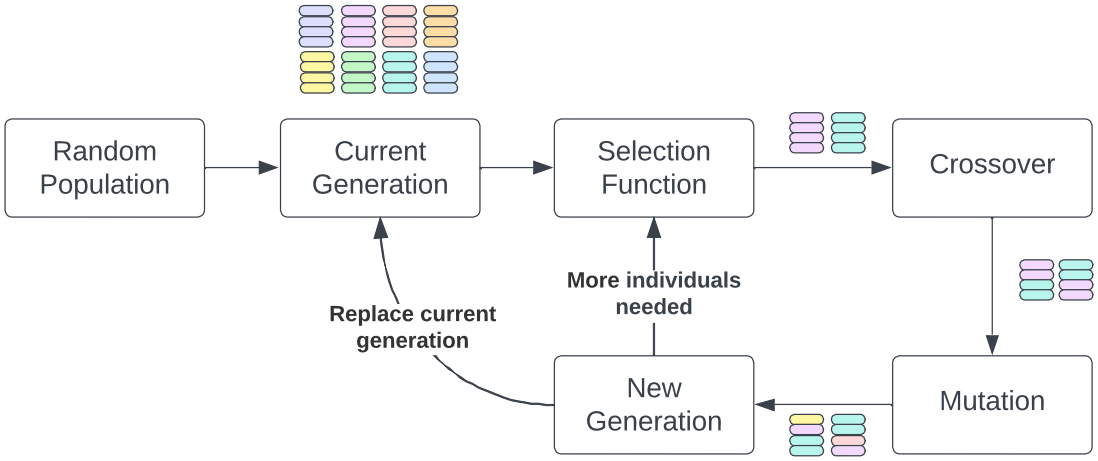


Figure 2.5: Core steps of a genetic algorithm.

called Rank rather than pure fitness scores like in the WholeSuite algorithm [21], [23]. With that said Rank cannot be determined without fitness scores, so both are closely related.

The two winners from the respective tournaments will become the parents P of two offspring O. To form an offspring, a *crossover* operation is performed. A random number $\alpha \in [0, 1]$ is selected and acts as a relative location in each parent P_1 and P_2. In WholeSuite, if $\alpha = 0.5$, then the first half of test cases in P_1 will end up in the first offspring O_1, while the latter half of P_1 will end up in the second offspring O_2. For P_2, the latter half relative to the crossover point, will end up in O_1 and the first half in O_2 [5]. The other algorithms, DynaMOSA, MOSA, and MIO, perform crossover on test cases instead, meaning that crossover is performed on code statements to build new test cases, rather than test suites [21]–[23].

Each offspring undergoes something called *mutation*, where pieces of their building blocks are added, changed, or deleted [26]. For example, mutation of a test case can involve adding a new statement and changing an old one. Which of the three operations to perform are chosen at random and can occur multiple times per test case. The process of picking parents, creating offspring, and mutating them, is repeated until the *new generation* is filled up, at which point it replaces the *current generation*. The algorithm keeps forming new generations until it has found a sufficient solution or the budget runs out.

2.5 Reinforcement Learning

Reinforcement learning can be summarized as the core idea of “learning by doing”, where actions are influenced by interactions with an environment as well as previous experiences for similar events. Humans are no strangers to this type of learning, as infants, being birthed into an unknown environment and not being taught what

do to, we still learn to talk and walk through positive/negative enforcement from our parents, the environment, by playing, or through rewards such as toys, or visual/auditory stimuli [27]. During adolescence and adulthood, as we encounter social situations like finding a place to sit in a cafeteria [28], or striking up a conversation with a new coworker, we are acutely aware of how our actions are reflected by the environment. Both the present and past, as well as positive and negative experiences shape how we navigate future known or unknown environments.

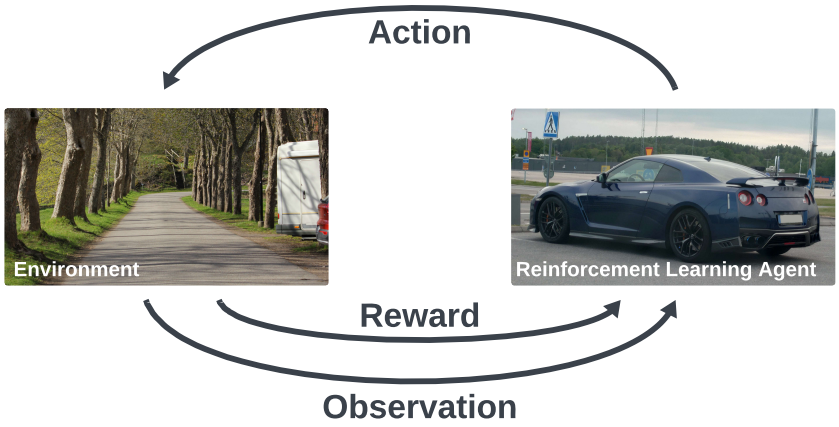


Figure 2.6: A typical reinforcement learning scenario, showing the interplay between a reinforcement learning agent and an environment exchanging actions, observations, and rewards.

A typical RL loop is illustrated in Figure 2.6, where an *agent* (e.g., a car, a robot), performs an *action* (e.g., steer, walk) in an *environment* (e.g., a road, a room) [16], [29]. After performing the action, a new updated *observation* of the environment's current state, as well as a *reward*, is transmitted back to the agent, and the process repeats [30]. The single motivator for an agent to update its *policy* (i.e., behavior, strategy) is through the idea of maximizing the received rewards [16]. The agent needs no prior knowledge about the environment, yet it can learn its objective by having a measure of success and receiving feedback in terms of environment observations and rewards [16], [29].

One final aspect to consider is the idea of immediate rewards and future rewards. Only making decisions aimed toward maximizing rewards in the short term has the potential to forgo actions that would yield more rewards in the long term. Thus, an RL agent must consider the *value* (i.e., the long-term reward) of performing an action, not just the immediate reward. To reach states with higher value, the RL agent must strike a balance between exploiting already known state transitions and exploring previously undiscovered states. The topics of exploring-exploiting and calculating state-values are both difficult problems that have been heavily researched over the decades [16], [29], [30].

3

Related Work

This chapter presents relevant research and previous studies within the area of machine learning applied to test generation techniques and the area of parameter control, in addition to positioning this study in relation to the related works.

3.1 Using Machine Learning to Improve Test Generation Techniques

In this section, we present previous work that applied machine learning (ML) with the aim of improving some aspect(s) of the test generation process. Some apply reinforcement learning (RL) to adapt the search strategy, while others use neural networks to model the system under test (SUT) as a guide for the test generation process.

In a paper by Fontes *et al.*, they present the results from a systematic mapping study of publications on the use of ML applied to automated test generation [4]. They conclude that the topic of automated test generation is well-studied, but that the addition of ML is more recent. The use of ML takes on many different forms, from generating input for different kinds of testing to enhancing the performance of already existing generation frameworks and algorithms [4]. Finally, they present that there exist challenges for current studies on the topic in regards to parameter tuning, evaluation complexity, choice of ML model, and replicability among other things [4].

Zhao *et al.* used a neural network to model the system under test before applying a genetic algorithm (GA) to generate test input, effectively enabling black-box test input generation [31]. The goal of their artifact was to have the genetic algorithm search the input space until it finds the input corresponding to a desired output. Once it finds an input value producing an output sufficiently close to the desired value, the search stops. They evaluated their GA against a general GA and found that their GA identified a satisfactory input in substantially fewer iterations than the general one [31].

Another study, by Esnaashari *et al.*, explores how a genetic algorithm in combination

with reinforcement learning can be used to generate test data [6]. They present a memetic algorithm called MAAT, which is a genetic algorithm that makes use of reinforcement learning to accept and attempt to improve the best test cases generated at each step of the algorithm [6]. They evaluate MAAT by performing four experiments and comparing the outcome with 10 different algorithms (e.g., other meta-heuristics and evolutionary algorithms). For all four experiments, MAAT performed as well as, or better than all other algorithms in regards to coverage achieved. Additionally, MAAT achieved 100% coverage in substantially fewer fitness evaluations across all experiments [6].

Yazdani Banafshe Daragh *et al.* present an extension of the pseudo-random GUI testing tool *UI/Application Exerciser Monkey*, which generates user inputs, e.g., tapping or swiping, to test an Android application [32]. Their version deploys ML to make more educated guesses about where and how to interact with the GUI. This is accomplished by having the ML use screenshots or the structure of the UI to check whether an action (input) was valid, i.e., whether the application reacted to the action. Their extension is evaluated against the original Monkey tool and shows improvements for applications of higher *crawling complexity* while it generally performs as well for less complex applications [32].

In their study, Almulla *et al.* present the framework EvoSuiteFIT, based on the open-source unit test generation framework EvoSuite [1], [10]. EvoSuiteFIT employs reinforcement learning to dynamically adapt the fitness functions used during the test generation process to the system under test. Almulla *et al.* implemented EvoSuiteFIT for two different reinforcement learning algorithms, which they evaluated based on three goals in comparison to three baselines without any reinforcement learning. The results of the evaluation show that EvoSuiteFIT performs better than the baselines for all three goals, *Exception Discovery*, *Test Suite Diversity*, and *Strong Mutation Coverage*. Additionally, they state that even though EvoSuiteFIT introduces increased overhead to the standard EvoSuite, it is often faster by avoiding the calculation cost of fitness functions that provide no value to the search [1].

In a similar study by Buzdalov *et al.*, they explore how helper-objectives—additional fitness functions that add secondary goals with the hope of improving the results for the main, or target, objective—can improve test generation when using a GA. This is accomplished by adding an RL agent that chooses between a set of objectives, i.e., target- or helper-objectives, at each GA generation. The RL agent receives a reward based primarily on the increase in target fitness between two subsequent generations, but also the increase for each helper-objective. Their results show that their method for adaptive helper-objective selection leads to improvements over manual objective selection [33].

Sharma *et al.* present a property-driven testing approach, which allows *users* to define properties of the System Under Test (SUT) without writing the necessary data generation functions [34]. Since the approach works on a property-based level, it can generate test data for black-box SUTs (i.e., where the internal representation

is unknown). ML is utilized to create an approximate model of the SUT, which serves as a foundation to generate test data and thus identify any violations against any of the specified properties. They evaluated the approach on ten aggregation functions for nine properties, and their approach managed to identify violations for all properties not proven to belong to the respective aggregation function [34].

3.2 Parameter Control in Evolutionary Algorithms

Parameter Control is the idea of adjusting parameter values during execution, in comparison to *Parameter Tuning* which is the idea of adjusting parameter values before execution [13], [35]. There are many possible ways of performing parameter control, some utilize an additional genetic algorithm (GA) layer to control the parameters of an underlying GA, while some use reinforcement learning [36]. In this study, we are specifically interested in the use of RL to perform parameter control, for two main reasons. Firstly, RL does not require any procured data but instead learns from the environment during execution [16]. Learning from the environment allows RL to learn and adapt to the changing state and needs of the test generation framework as it progresses, which enables the parameter control system to be flexible as to what modules it can work with. Secondly, it is a good fit since both RLs and GAs follow a similar evolutionary structure, where they iteratively search for improvements in the generated solutions [16], [36]. The similar evolutionary structures aid in keeping the interaction between the RL and the GA simple as both have logical steps that integrate well, i.e., the fitness of the GA can work as a reward for the RL and the action of the RL can help guide the evolution of the GA.

Within parameter control for evolutionary algorithms (EA), there are three different types, *deterministic*, *adaptive*, and *self-adaptive* [13], [35]. These types relate to how the control is performed. *Deterministic* parameter control implies that the parameter values are controlled by a predefined scheme, while *adaptive* parameter control updates the parameter values based on feedback from the search process. *Self-adaptive* parameter control adds the parameter control task into the search task of the EA, thus, having the EA update its own parameter values.

An early example of parameter control in EAs is Pettinger *et al.*'s work in optimizing the Traveling Salesperson Problem (TSP) [18]. In their study, the RL decides which crossover or mutation operations their EA, in this case, a GA, should perform and on which classification of individuals (i.e., the top 10% or the bottom 90% according to fitness). The reward function used in the study is the difference in fitness between parents and their offspring. Pettinger *et al.* conclude that their RL-tuned GA outperforms a non-tuned GA by producing a better solution and learning faster for TSP [18]. Sakurai *et al.* propose a continuation of this paper, where they add computational time to the reward function [36].

Another early example is a study by Müller *et al.*, where they investigate tuning the step size of an Evolutionary Strategy using RL [37]. They compare four different reward methods on six different optimization problems, and their results show that

the choice of reward function has a measurable impact on both the average number of iterations it takes to converge and the convergence rate. The two best-performing reward functions involved the difference in fitness score between generations [37].

Eiben *et al.* applied reinforcement learning to tune the parameters of a GA during execution [13]. In their study, the RL algorithm was allowed to control four parameters, Population Size, Tournament Proportion, Crossover Probability, and Mutation Probability, by setting the values directly within a given range. As a foundation for the RL algorithm's decisions, it is provided with an observation vector containing 11 values, including the best, the mean, and the standard deviation of the fitness as well as the previous action, to name a few. In addition to the observation vector, the RL algorithm is also provided with a reward function which is specified as the increase of the best fitness. The results that Eiben *et al.* present indicate that their adaptive GA performs equally or better than a non-adaptive GA in all of their tests [13]. For more tests with higher complexity, the adaptive GA outperforms the non-adaptive GA more substantially.

Karafotias *et al.* created a generic EA parameter controller using RL that can control an arbitrary number of parameters [12]. For each controlled parameter, they defined a discrete action space where each action corresponds to a range of parameter values. After a range has been selected for each parameter, a random value from within these ranges is picked and set as the updated parameter value. Similarly to other studies, they use fitness and diversity as part of the observation of the current state. However, they also include a value representing whether the EA has stagnated to encourage more exploration. The reward is defined as the ratio between the current and previous generations' fitness averaged over the number of evaluations made since the previous generation, indicating the effort. The results show that the RL control generally performed better than the static version, however, it did not provide improved results for all their tests and performed similarly to the results of a random control for many of the tests [12].

Quevedo *et al.* controls two parameters of a GA, Mutation probability and Crossover probability, using an RL agent [38]. For each iteration, their RL chooses one out of ten possible values for each of the two parameters, corresponding to 100 possible actions in total. The observations used to provide the RL with a view of the current state consist of the difference in fitness between generations and a value representing the diversity of the population. The reward function is calculated by taking both these factors, fitness difference, and diversity, into account. From the evaluation performed in the paper, the RL-controlled GA showed potential by performing as well as, or better than its non-RL-controlled counterparts [38].

A similar example that also involved control of the Crossover and Mutation rate parameters is the study by Chen *et al.* [39]. In their paper, they defined 10 discrete actions, where a single action enabled a different set of possible ranges for both parameters. Depending on the discrete action selected, a random value within the ranges is selected for both parameters. The observation consisted of a single

value which took the average fitness of the population, population diversity, and best individual fitness and aggregated them into a weighted sum. The reward was calculated by considering both the average population fitness increase and as well as the best individual fitness in subsequent generations. The result shows significant improvement for their specific evaluation problem [39].

A more recent example is a study by Lacerda *et al.*, where they present an out-of-the-box parameter control method for any arbitrary meta-heuristic [35]. Out-of-the-box means that the method is not problem-specific, but instead can be applied to a wide variety of different problems. The parameter control is managed by an RL agent, which performs actions by setting parameter values directly for the given meta-heuristic. As a basis for these actions, it has access to an observation of the current state which in turn consists of 254 values, including a measure of population fitness, remaining budget, and fitness distances between selected individuals, among others. In addition to the observation they also use a reward which is defined as the logarithmized ratio between the best fitness scores of the current and previous generations. They also use an additional tuning layer to tune the hyper-parameters of the RL agent, known as a Population-based Training (PBT) algorithm, which needs to be manually tuned by the user [35].

3.3 Summary

As we can see from the studies presented in this chapter, machine learning applied to test generation is a quite common technique when improvements are desired. When looking at the specific case of improving methods based on some kind of Evolutionary Algorithm (EA), e.g., Genetic Algorithms (GA), Parameter Control has successfully been deployed in numerous cases. However, none of the aforementioned studies apply parameter control to test generation frameworks, echoing the results found by Fontes *et al.* [4]. While there are generic alternatives that could be applied, e.g., Karafotias *et al.* [12], Lacerda *et al.* [35], these were designed to work for single-objective optimization algorithms. Test generation algorithms in Pynguin, except Whole Suite, are multi-objective optimization algorithms, thus the proposed approaches are not applicable.

Our approach is to use an RL agent to control one to twelve genetic algorithm parameters, e.g., Population Size, Test Insertion Probability, or Crossover Rate, during the test generation process. As mentioned by Karafotias *et al.*, there are two main options for the action space; applying changes or setting the value directly [12]. The option we are using is letting the RL agent select continuous delta values, i.e., slight changes that either increase or decrease the parameter values. Our RL agent gets information about the current state of the test generation process via an observation vector containing the branch coverage of the current best test suite, in addition to the current parameter values in compliance with the recommendations provided by Karafotias *et al.* when using the delta action option [12]. We normalize both observations and actions to eliminate any precedence or bias, for or against any particular parameters, since they are operating on different scales [35], [40]. The

reward function, like many other works [13], [18], [37]–[39], is based on the difference between fitness scores, i.e., the difference in the highest attained branch coverage between generations.

4

Implementation

This chapter presents the implementation of the artifact produced for this study, PynguinAPC, the PYthoN General UnIt test geNerator with Adaptive Parameter Control (available on GitHub[1]), as well as the parameters controllable through the use of reinforcement learning.

PynguinAPC is an extension of the test generation tool Pynguin, introduced in Section 2.3. The changes made to the core of Pynguin were kept to a minimum, to reduce the risk of introducing incorrect behavior or negatively impacting the performance of the base system. The changes and additions made to Pynguin can be grouped into three main parts: the test generation algorithm, the parameter control layer, and the Pynguin execution configuration.

4.1 Test Generation Algorithm

Pynguin uses a test generation algorithm to create test cases and test suites. Within the base Pynguin there are five of these algorithms to choose from: *DynaMOSA* [21], *MIO* [22], *MOSA* [23], *Random* [9], and *WholeSuite* [5]. All these test generation algorithms, excluding the Random algorithm, are Genetic Algorithms (GA) (Section 2.4.2 explains GAs in more detail). As to not remove or modify any of the existing features of Pynguin, a new version of the DynaMOSA algorithm was implemented. This new implementation, called DynaMOSA_RL is from a test generation perspective functionally equal to the original DynaMOSA implemented in Pynguin. However, DynaMOSA_RL adds the option to perform parameter control during the test generation process. Similar versions could be implemented for the remaining test generation algorithms, but for this study, only one such test generation algorithm was implemented. The DynaMOSA algorithm was deemed to be the optimal choice, as it, according to an empirical study performed by the creators of Pynguin, outperformed the other algorithms when using branch coverage as the goal [25].

Figure 4.1 shows how DynaMOSA_RL fits into the core steps of Pynguin and provides a simplified overview of how it interacts with the Parameter Control Layer. For example, if PynguinAPC is configured with DynaMOSA_RL as the chosen test

[1] `https://github.com/henkejson/pynguin-adaptive-parameter-control`

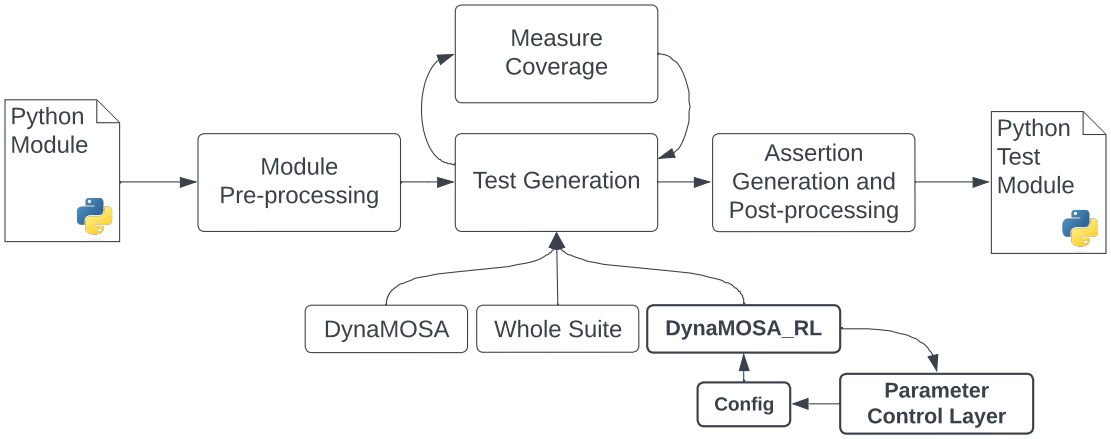


Figure 4.1: How the addition of DynaMOSA_RL and the Parameter Control Layer fits into the Pynguin workflow. New components are highlighted with bold text and borders.

generation algorithm, it will launch DynaMOSA_RL when it reaches the Test Generation step. When launched, the chosen test generation algorithm is run as per the definition of the algorithm. All test generation algorithms work iteratively to generate the resulting test suites, each iteration corresponding to one generation of the evolution during the search process. However, unlike the other test generation algorithms, DynaMOSA_RL performs one additional step in each iteration. This additional step consists of DynaMOSA_RL invoking an update from the Parameter Control Layer, to let it know that new parameter values are desired. Upon this invocation, the Parameter Control Layer checks if it is time to update the controlled parameters' values, at which point it modifies them (described in Section 4.2). After, when the values have been updated, DynaMOSA_RL reads them as normal from an object called Config. The Config object is a singleton responsible for keeping the complete configuration of Pynguin (e.g., the chosen test generation algorithm, the time budget, all parameter values, etc.). An object being a singleton means that there only exists one singular instance of that object in the system, which helps avoid the risk of having multiple different values saved for the same parameters in different parts of the system. The test generation process continues until a termination condition is met, e.g., if the generated test suite has reached 100% branch coverage, or if the time budget is exhausted, at which point Pynguin moves onto the next step to finalize the generated test suite.

4.2 Parameter Control Layer

The parameter control layer consists of three main parts: the Reinforcement Handler, the Environment, and the Reinforcement Learning (RL) Agent. Figure 4.2 provides a more detailed view of how these parts interact and work with DynaMOSA_RL and the Config object.

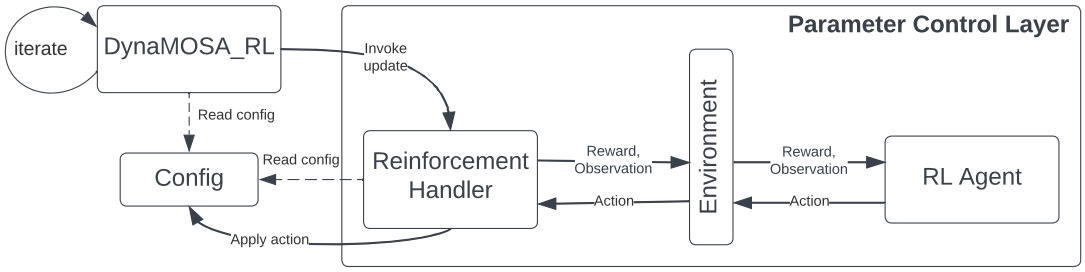


Figure 4.2: Overview of the interactions between the test generation algorithm DynaMOSA_RL and the Parameter Control Layer. Solid lines represent information being sent or forwarded, while dashed lines represent information retrieval.

4.2.1 Environment and Reinforcement Learning Agent

The core of the Parameter Control Layer follows the reinforcement learning (RL) Agent/Environment structure defined in Section 2.5. This structure is further encouraged by the tools utilized for the implementation, i.e., Gymnasium [41] and Stable-Baselines3 [42]. Gymnasium is an API standard that brings predefined functionality for an RL agent/Environment implementation, greatly decreasing the workload of getting such a system up and running [41]. Stable-Baselines3 is a collection of reinforcement learning algorithm implementations that work in a "plug-and-play" fashion with Gymnasium [42]. This synergy between Gymnasium and Stable Baselines3 makes it convenient and simple to swap reinforcement learning algorithms to any of the algorithms included in Stable Baselines3 [43].

In PynguinAPC, an action contains continuous values for all parameters currently controlled. These values represent deltas, which are applied to the respective parameters whenever an action has been taken. Thus, an action can result in a parameter value increasing, decreasing, or staying the same, depending on the delta value selected by the RL agent. This action strategy means that all new parameter values are dependent on the previous values. To help facilitate the actions taken by the RL agent, it receives a snapshot of the parameter values. Another possible way of defining how actions work is to let the RL agent select new parameter values directly, replacing the old values instead of modifying them according to a delta. However, allowing the RL agent to perform such actions could pose a problem of continuity for the parameters that are continuous [12].

Since continuous values are used for the actions, some of the reinforcement learning algorithms from Stable-Baselines3 are no longer valid options, as they only support discrete actions. Among the algorithms still applicable, the one used in PynguinAPC is Soft Actor-Critic (SAC). SAC fulfills the criteria of supporting continuous actions in addition to outperforming many of the other possible options (e.g., TD3, PPO, and DDPG) in a study comparing different reinforcement learning algorithms [44]. Because all actions are continuous, and as per a recommendation by Stable Baselines3 [40], both the action and observation spaces are normalized (between $[-1, 1]$)

in addition to the naturally normalized rewards (between $[0, 1]$). Thus, the RL agent and the Environment only interact with normalized values.

The RL agent and Environment keep their own control flow in parallel to the test generation algorithm. The Environment awaits feedback from the test generation algorithm which it forwards to the RL agent. In turn, the RL agent uses the feedback, consisting of an observation and a reward, to decide on a new action to perform. Descriptions of the actions, observations, and rewards in the context of PynguinAPC are presented below.

Actions A single continuous value per parameter currently being controlled. As the action space is normalized, all parameters are represented by the same interval ($[-1, 1]$), regardless of the actual format and values of the specific parameters. The normalization and denormalization of actions are handled by the Reinforcement Handler, explained in more detail in Section 4.2.2. After an action is denormalized, it is used as a delta to update the controlled parameter(s). For example, an action of 0.10 (denormalized) would result in a parameter with a current value of 0.65 increasing to 0.75, while an action of -0.05 would result in the same parameter value decreasing to 0.60.

Observations A snapshot of the state of Pynguin after the last action was applied. Used by the RL agent, in conjunction with the reward, to guide its future actions. Consists of the branch coverage for the current best generated test suite and all parameter values currently controlled (all normalized in the interval $[-1, 1]$).

Rewards A single value representing the improvement, or lack thereof, that came as a result of the last action taken by the RL agent. Used by the RL agent, in conjunction with the observation to guide its future actions. Consists of the change in branch coverage for the generated test suite since the last action was applied. Since DynaMOSA, and therefore DynaMOSA_RL as well, generates test suites iteratively while keeping track of the best test suite created so far, the branch coverage never decreases between iterations. Thus, the reward is a value in the interval $[0, 1]$, where a reward of 0 would mean that there was no improvement in branch coverage since the last action, while a reward of 1 would mean that the generated test suite went from 0% to 100% branch coverage since the last action. As a result of this reward format, the maximum total reward received during the whole test generation process is 1.

4.2.2 Reinforcement Handler

The Reinforcement Handler acts as the singular access point to the Parameter Control Layer and has a few crucial responsibilities. It handles the communication between DynaMOSA_RL via the Config object, and the RL agent via the Environment. For every iteration of DynaMOSA_RL, it invokes an update from the Reinforcement Handler. However, the parameter control does not start instantly

with the launch of the test generation process. Instead, there is a plateau check, where the Reinforcement Handler inspects the branch coverage trend of the generated test suite at each iteration. Only when there has not been any growth in the branch coverage for a set number of iterations will it initialize the communication with the RL agent. This plateau check is in place to stop the RL agent from learning potentially erroneous behavior from the initial test generation iterations, where the branch coverage usually grows rapidly regardless of any changes to the parameter values.

In addition to the plateau check, the Reinforcement Handler will not request an action from the RL agent every single iteration. Rather, to give the previous action time to have an impact, new actions will only be requested according to an update interval. This update interval works in relation to DynaMOSA_RL's iterations, e.g., an update interval of five means that the Reinforcement Handler requests an action to update the parameter values once every five DynaMOSA_RL iterations.

As soon as a branch coverage plateau is reached, the Reinforcement Handler will begin requesting actions from the RL agent according to the update interval. However, as the RL agent and Environment work in normalized space the Reinforcement Handler also needs to handle the transformation to and from normalized space. To enable these transformations, each parameter has a Transformation Handler that defines how that specific parameter should be transformed. E.g., some parameters are percentages (i.e., $[0, 1]$), while some others are discrete values (e.g., $\{0, 1, 2, ..., 98, 99\}$), all of which require different transformations.

Thus, to send a request, the Reinforcement Handler must first collect the necessary information, and normalize all that information, before finally forwarding it to the Environment. A request consists of a reward and observation pair. The reward is calculated from the difference in branch coverage since the last update interval, for the first iteration of the test generation process the reward will always be 0.00. The observation consists of the branch coverage achieved by the current best generated test suite and all parameter values currently being controlled. After sending the request, the Reinforcement Handler waits until it receives a response in the form of an action (containing one value per parameter being controlled). These delta values are then denormalized before being applied to the Config object, which in turn is used by DynaMOSA_RL to access the latest parameter values. This request-response cycle continues until either DynaMOSA_RL finishes or the test generation time budget is spent.

4.3 Execution Configuration

The modifications made to base Pynguin to extend it to PynguinAPC necessitated some additions to the command line interface (CLI) execution configuration to allow for parameter control to take place. These additions were,

- The addition of the DynaMOSA_RL algorithm as an option for the test gen-

eration algorithm (e.g., `--algorithm DYNAMOSA_RL`).

- The addition of an option for which parameters to control, can be one or multiple (e.g., `--tuning_parameters ChromosomeLength,TestInsertProbability`).

- The addition of an option for the coverage plateau, represented as the number of iterations by DynaMOSA_RL before parameter control is engaged (e.g., `--plateau_length 10`).

- The addition of an option for setting the update interval at which the RL agent should be invoked for updated parameter values (e.g., `--update_frequency 5`).

4.4 Parameters

To keep the number of parameters to a manageable number, a subset of the parameters available in Pynguin was selected. This subset was based on the general search algorithm configuration parameters within Pynguin (see the Pynguin documentation[2]). Any parameters that only set initial values were excluded as changing these during runtime would have no effect. Similarly, all boolean parameters were excluded since they generally represent whether a specific feature is enabled or not. This could lead to problems or failures since these features most likely were not designed to withstand being enabled or disabled during the test generation process. A specific parameter that was excluded is Rank Bias since that parameter is only used for the rank selection function. The default selection function, and the one used in PynguinAPC, is tournament selection. Thus, Rank Bias would have no impact on the test generation. Table 4.1 presents the final subset of 12 parameters that were used for this study as well as descriptions for each.

As mentioned in Section 4.2.2, each parameter has a corresponding transformation handler that defines how that specific parameter should be transformed when being normalized or denormalized. These transformations necessitate that all parameters have defined minimum and maximum values that they can assume, as well as, minimum and maximum delta values for the actions performed on that parameter. If an action would cause the parameter to extend past its allowed range, the value is set to the nearest boundary. Table 4.2 shows all this information for all 12 parameters in addition to their default values, which are the default values used in base Pynguin.

[2] `https://pynguin.readthedocs.io/en/latest/api.html#pynguin.configuration.SearchAlgorithmConfiguration`

Parameter	Description
Change Parameter Probability	Used during the mutation step to decide whether to replace parameters of a method or a constructor statement in a test case.
Chromosome Length	Used to limit the maximum length of chromosomes (test cases) during the generation process i.e., number of lines.
Crossover Rate	Used during the creation of a new generation to decide whether crossover will occur between two offspring.
Elite	Used to decide how many of the best individuals in the population will be kept for the next generation.
Population Size	Used to decide the size of the population (number of individuals) in each generation.
Random Perturbation	Used during the mutation step to decide whether to replace a primitive with a new random value instead of adding a delta.
Statement Insertion Probability	Used during the mutation step to decide how many (if any) new statements should be inserted at random positions in the chromosome (test case).
Test Change Probability	Used during the mutation step to decide whether a random statement should be replaced.
Test Delete Probability	Used during the mutation step to decide whether a random statement should be deleted.
Test Insert Probability	Used during the mutation step to decide whether a random statement should be inserted.
Test Insertion Probability	Used during the creation of a new generation to decide how many new chromosomes (test cases) to add.
Tournament Size	Used during tournament selection to decide the number of individuals in the tournament.

Table 4.1: All 12 parameters available for parameter control in PynguinAPC with descriptions.

Parameter	Default Value	Allowed Range	Delta Values
Change Parameter Probability	0.1	[0, 1]	[-0.1, 0.1]
Chromosome Length	40	[20, 80]	[-6, 6]
Crossover Rate	0.75	[0, 1]	[-0.1, 0.1]
Elite	1	[0, 10]	[-1, 1]
Population Size	50	[25, 100]	[-8, 8]
Random Perturbation	0.2	[0, 1]	[-0.1, 0.1]
Statement Insertion Probability	0.5	[0, 1]	[-0.1, 0.1]
Test Change Probability	0.33	[0, 1]	[-0.1, 0.1]
Test Delete Probability	0.33	[0, 1]	[-0.1, 0.1]
Test Insert Probability	0.33	[0, 1]	[-0.1, 0.1]
Test Insertion Probability	0.1	[0, 1]	[-0.1, 0.1]
Tournament Size	5	[2, 10]	[-1, 1]

Table 4.2: All parameters for which parameter control is implemented, with their default values, range of allowed values, and range of allowed delta values (i.e., how much an action can modify the parameter value).

5

Method

This chapter describes the research questions this study aims to answer and the methods used to answer them. Code to rerun all experiments, resulting data from the experiments, statistical model traces, and analysis scripts (Jupyter notebooks) are available as part of a replication package[1].

5.1 Research Questions

RQ1: How does parameter control affect the final branch coverage achieved by the generated test suites...

RQ1.1: ... when only controlling a single parameter at a time?

RQ1.2: ... when controlling a subset of parameters?

RQ2: How does parameter control affect the branch coverage growth rate during the generation process...

RQ2.1: ... when only controlling a single parameter at a time?

RQ2.2: ... when controlling a subset of parameters?

RQ3: How does the overhead of parameter control affect the number of test generation iterations completed during the search budget...

RQ3.1: ... when only controlling a single parameter at a time?

RQ3.2: ... when controlling a subset of parameters?

The goal of **RQ1** is to identify the impact of parameter control on a generated test suite and its subsequent effect on the final branch coverage achieved by the test suite. **RQ1.1** and **RQ1.2** attempt to identify if there is a measurable difference between the results when controlling one parameter at a time or when controlling pairs of

[1] `https://doi.org/10.5281/zenodo.11353851`

parameters. Additionally, **RQ1.1** can help determine which parameters have the largest (positive or negative) individual impact on the final branch coverage achieved for specific modules under test.

RQ2 aims to answer whether the addition of parameter control affects the rate at which branch coverage grows during the test generation process. This research question also explores, regardless of any changes in final branch coverage, if a change in branch coverage growth rate can be measured, providing another potential avenue for improvements by implementing parameter control. The separation into **RQ2.1** and **RQ2.2** function to investigate if the impact of parameter control on branch coverage growth rate differs depending on whether individuals or pairs of parameters are controlled.

With **RQ3** we investigate the performance overhead that is added by running the parameter control system in addition to the test generation process. In combination with the results from **RQ1** and **RQ2**, **RQ3** is meant to help answer whether parameter control provides enough of an improvement in either final branch coverage or branch coverage growth rate to compensate for the added overhead.

5.2 Experimentation Overview

This study follows the experimental simulation strategy [45]. This class of research design entails the creation of a controlled setting to enable a high degree of measurement accuracy for specific behaviors of interest. In the case of this study, the setting consists of the configuration of which parameters and modules were used for the experiments and the specific configuration (e.g., search budget, choice of test generation algorithm, or update interval) of the test generation framework, PynguinAPC.

To answer the research questions (see Section 5.1), the study design was divided into two main cycles to explore whether the study would yield different results depending on the configuration of the controlled parameters. The first cycle consists of a single-parameter experiment, and the second cycle consists of a multi-parameter experiment. Figure 5.1 presents an overview of the steps of the study, followed by a more detailed explanation of what each step entailed.

Implement Parameter Control We extended the test generation framework Pynguin to add the functionality of parameter control. The resulting implementation, PynguinAPC, is detailed in Chapter 4.

Identify and Filter Parameters We examined the available parameters in Pynguin and filtered them based on what parameters would be viable to perform parameter control on. This process and the resulting set of parameters are detailed in Section 4.4.

Identify and Filter Modules We identified a set of modules to run PynguinAPC

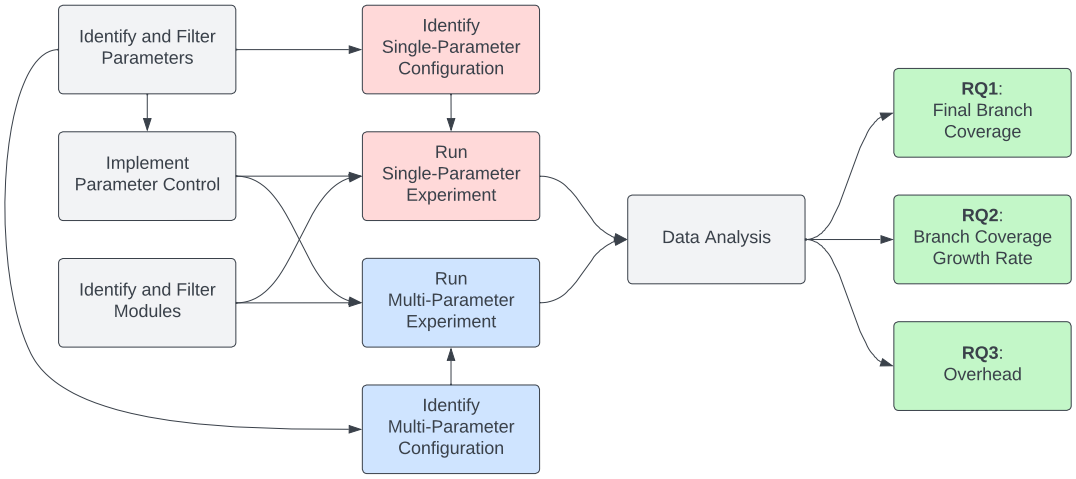


Figure 5.1: Overview of the steps taken through the experimental simulation study. Steps unique to the single-parameter experiment are highlighted in red, while the steps for the multi-parameter experiment are highlighted in blue. Steps shared between experiments are shown in gray. All steps contribute towards answering the research questions shown in green.

on for the experiments. The identification and filtering process in addition to the resulting set of modules are detailed in Section 5.3. The same modules are used in both experiments.

Identify Single-Parameter Configuration We first examined the impact of controlling a single parameter at a time. We grouped all parameters discussed in Section 4.4 pairwise with the modules discussed in Section 5.3. The configuration settings for this experiment are detailed in Section 5.4.1.

Run Single-Parameter Experiment We executed the experiment and collected the data described in Section 5.4.

Identify Multi-Parameter Configuration For this experiment we were interested in applying parameter control to multiple parameters at a time. As we wanted to run all available parameters while keeping the length of the experiment manageable within the boundaries of this study, we decided to run all pairs of two parameters. All pairs of parameters were then grouped with all modules listed in Section 5.3. The configuration settings for this experiment are detailed in Section 5.5.1.

Run Multi-Parameter Experiment We executed the experiment and collected the data described in Section 5.5.

Data Analysis To enable the analysis of the experiment results and, in turn, the answering of the research questions, we designed one Bayesian model per re-

search question. We use the same models for the single- and multi-parameter experiments, e.g., RQ1.1 and RQ1.2, but only use data from their respective experiment. These models are detailed in Section 5.6. After executing both experiments and running our Bayesian models, we analyzed and visualized the results. The results of our analysis are presented in the Results chapter, see Chapter 6.

5.3 Module Identification and Filtering

The modules used for the experiments are a subset of the ones used in an empirical evaluation of Pynguin [25]. While they list 163 modules, they only provide detailed names for 134 modules. To identify a subset of modules for use in our experiments, a few test runs were conducted. These test runs consisted of running an unmodified version of Pynguin for all modules with a search budget of 30 seconds, 1 minute, and 5 minutes while removing any modules that caused crashes or threw errors between each run. Lastly, we ran an additional test run, this time with parameter control enabled for a subset of three parameters, to ensure that all remaining modules and PynguinAPC functioned as expected, i.e., that no crashes occurred and no errors were encountered. From the test runs, the reasons why modules were filtered out were:

File could not be located These were modules for which the files specified in the experiment could not be located from their respective repositories (12 modules).

Runtime issues These were modules that encountered some kind of issue during runtime. For example, some files caused errors that were deemed too time-consuming to fix, while other files would cause the test generation process to fail without any clear reason as to why (19 modules).

100% coverage These were modules that either reached 100% coverage within a small number of iterations or modules that consistently reached 100% coverage within the search budget without parameter control (37 modules).

Lack of content These were modules that either were too small or contained no branching code or methods, e.g., a few modules only contained constants (4 modules).

Few iterations These were modules that would iterate very slowly (under 500 iterations total), which would not provide the parameter control with enough iterations to work with (4 modules).

In total, of the 134 modules, 76 modules were filtered out. The final set of modules totaled 58 (see Table A.1 in Appendix A for the full list). To control experiment costs, out of these 58, we randomly selected 24 modules used for our evaluation. These modules are detailed in Table 5.1.

Full Module Name
codetiming._timer
flutils.decorators
flutils.namedtupleutils
flutils.packages
flutils.setuputils.cmd
httpie.output.formatters.headers
httpie.plugins.base
mimesis.builtins.da
py_backwards.transformers.base
py_backwards.transformers.dict_unpacking
py_backwards.transformers.return_from_generator
py_backwards.transformers.yield_from
py_backwards.utils.helpers
pymonet.immutable_list
pymonet.maybe
pymonet.validation
pypara.accounting.journaling
pytutils.lazy.lazy_import
pytutils.python
sanic.config
sanic.helpers
sanic.mixins.signals
thonny.plugins.pgzero_frontend
typesystem.tokenize.positional_validation

Table 5.1: The 24 modules used for the experiments (see Appendix A for the complete set of 58 modules and Appendix B for the full module names).

5.4 Single-Parameter Experiment

The first experiment was conducted to explore the impact of performing parameter control on a single parameter at a time. This section presents the procedure for the experiment as well as what data was collected. The method used for the analysis of the experiment is detailed in Section 5.6.

5.4.1 Procedure

This section provides an overview of the procedure and all choices made for the single-parameter experiment.

PynguinAPC configuration: The PynguinAPC configuration used for the single-parameter experiment was the following:

- **Test generation algorithm** DynaMOSA_RL. Presented in Section 4.1.

- **Repetitions** 30. Each configuration was repeated 30 times to limit the effect of variance.
- **Search budget** 300 seconds. Each run got a search budget of 300 seconds (5 minutes) to give the parameter control ample time to work.
- **Plateau length** 10 iterations. As explained in Section 4.2.2.
- **Update interval** 5 iterations. As explained in Section 4.2.2.

Parameters: Table 4.1 presents the 12 parameters used for the experiment. These parameters were selected as they are used by the test generation algorithm during the test generation process (see Section 4.4).

Modules: The 24 modules listed in Table 5.1 were used. Section 5.3 provides further insight into the identification and filtering process of these modules.

Baseline: In addition to the parameter control, each module was also run without parameter control with the same configuration. These runs were used as a baseline to compare the runs utilizing parameter control.

Execution: The execution of the experiment consisted of running a set of configurations[2]:

$$1\ algorithm \times 24\ modules \times 13\ parameters = 312\ configurations$$

Each configuration was repeated 30 times, resulting in a total number of runs:

$$312\ unique\ configurations \times 30\ repetitions = 9,360\ runs$$

The configurations were split across four identical virtual machines (VM), depending on the module under test, as shown in Table 5.2. Each VM was equipped with 4 vCPUs, 8GB of RAM, and 20GB of storage, running a server version of Ubuntu 18.04.4 LTS. Splitting the runs over these identical VMs allowed us to compare results between runs without worrying about external factors affecting the results.

Each run of PynguinAPC was executed in a Docker container [46]. The purpose of using Docker was to enable isolation of the run time environment in case anything goes wrong, and to facilitate external package management for the modules under test, i.e., making it easier to handle duplicate package dependencies of differing versions. The isolation also simplifies the process of rerunning or replicating the experiment.

[2]To simplify the calculation, the baseline was counted as a parameter, or rather, as the parameter control of no parameter, which is why there are 13 parameters in the calculations rather than the 12 listed.

VM 1	VM 2
codetiming._timer	httpie.plugins.base
flutils.decorators	mimesis.builtins.da
flutils.namedtupleutils	py_backwards.transformers.base
flutils.packages	py_backwards.transformers.dict_unpacking
flutils.setuputils.cmd	py_backwards.transformers.return_from_generator
httpie.output.formatters.headers	py_backwards.transformers.yield_from
VM 3	**VM 4**
py_backwards.utils.helpers	pytutils.python
pymonet.immutable_list	sanic.config
pymonet.maybe	sanic.helpers
pymonet.validation	sanic.mixins.signals
pypara.accounting.journaling	thonny.plugins.pgzero_frontend
pytutils.lazy.lazy_import	typesystem.tokenize.positional_validation

Table 5.2: Module distribution across the four VMs for the single-parameter experiment.

5.4.2 Data Collection

The kinds of data we collected for this experiment can be split into two different categories, one for data from the overall experiment as a whole and one for data collected from each executed configuration.

For the experiment (one of each per VM):

`statistics.csv` The main data file, contains information about the configuration for each run: run ID, module name, algorithm, number of iterations, search time, parameters controlled, final code coverage achieved, and a coverage timeline. The coverage timeline contains the best branch coverage achieved at every one-second interval over the whole 300-second time budget.

`full_logs.txt` A file containing the command line interface (CLI) logs from running the experiment. The log files show the progress of the experiment and which configuration was run, and also note if any errors were encountered during the execution of an individual configuration.

`completed_runs.json` A file listing all successfully executed configurations.

`failed_runs.json` A file containing any failed runs. If there were none, the file is not created.

`parameters_timeline.json` A file containing a timeline over the controlled parameter values for all executed configurations. However, after running the experiment this file became too large, therefore, it was split into smaller files, one per executed configuration.

For each executed configuration:

`logs.txt` A file containing the complete logs for the specific configuration. Shows the branch coverage for each iteration of the generation process, in addition to the observed parameter values and actions for all parameters controlled.

`[MODULE_NAME]#[REPETITION].py` A Python file containing the final generated test suite.

`cov_report.html and cov_report.xml` Two files (HTML and XML) containing a comprehensive overview of every branch in the module and their coverage status, as well as the total coverage.

`pynguin-config.txt` A file containing the complete Pynguin configuration.

5.5 Multi-Parameter Experiment

The second experiment was conducted to explore the impact of performing parameter control on multiple different pairs of parameters. This section presents the procedure for the experiment as well as what data was collected. The method used for the analysis of the experiment is detailed in Section 5.6.

5.5.1 Procedure

The procedure for the multi-parameter experiment is largely the same as the single-parameter experiment, with only a few differences between the two experiments.

PynguinAPC configuration: The configuration for PynguinAPC used for the multi-parameter experiment was the following:

- **Test generation algorithm** DynaMOSA_RL. Presented in Section 4.1.
- **Repetitions** 10. Each configuration was repeated 10 times to limit the effect of variance on the results. We used a reduced number of repetitions in this experiment, in comparison to the single-parameter experiment, to control cost.
- **Search budget** 300 seconds. Each run got a search budget of 300 seconds (5 minutes) to give the parameter control ample time to work.
- **Plateau length** 10 iterations. As explained in Section 4.2.2.
- **Update interval** 5 iterations. As explained in Section 4.2.2.

Parameters: Table 4.1 presents the 12 parameters used for the experiment. These parameters were selected as they are used by the test generation algorithm during the test generation process (see Section 4.4). Rather than controlling a single parameter at a time, as in the single-parameter experiment, for the multi-parameter

experiment, we combined the parameters into all possible pairs of two.

$$\binom{12}{2} = 66 \; parameter \; combinations$$

Modules: The 24 modules listed in Table 5.1 were used. Section 5.3 provides further insight into the identification and filtering process of these modules.

Baseline: In addition to the parameter control, each module was also run without parameter control with the same configuration. These runs are used as a baseline to compare the runs utilizing parameter control.

Execution: The execution of the experiment consisted of running a set of configurations[3]:

$$1 \; algorithm \times 24 \; modules \times 67 \; parameter \; combinations = 1,608 \; configurations$$

Each configuration was repeated 10 times, resulting in a total number of runs:

$$1,608 \; unique \; configurations \times 10 \; repetitions = 16,080 \; runs$$

The modules were split across five identical virtual machines (VM) as shown in Table 5.3. Each VM had the same specifications as the VMs used for the single-parameter experiment.

VM 1	VM 2
codetiming._timer	httpie.output.formatters.headers
flutils.decorators	httpie.plugins.base
flutils.namedtupleutils	mimesis.builtins.da
flutils.packages	py_backwards.transformers.base
flutils.setuputils.cmd	py_backwards.transformers.dict_unpacking
VM 3	**VM 4**
py_backwards.transformers.return_from_generator	pymonet.validation
py_backwards.transformers.yield_from	pypara.accounting.journaling
py_backwards.utils.helpers	pytutils.lazy.lazy_import
pymonet.immutable_list	pytutils.python
pymonet.maybe	sanic.config
VM 5	
sanic.helpers	
sanic.mixins.signals	
thonny.plugins.pgzero_frontend	
typesystem.tokenize.positional_validation	

Table 5.3: Module distribution across the five VMs for the multi-parameter experiment.

Each configuration of Pynguin was executed in a Docker container [46]. The purpose of using Docker was to enable isolation of the run time environment in case anything

[3]Similarly to the single-parameter experiment, the baseline case of performing no parameter control is counted towards the number of parameter combinations in the calculations.

goes wrong, and to facilitate external package management for the modules under test, i.e., making it easier to handle duplicate package dependencies of differing versions. The isolation also simplifies the process of rerunning or replicating the experiment.

5.5.2 Data Collection

The data collected for the multi-parameter experiment is the same as for the single-parameter experiment with the only exception being the `parameters_timeline.json` file. Rather than creating one such file per VM, we instead created one such file per configuration directly. This change was done to avoid getting files that were inconvenient to handle due to their size.

Section 5.4.2 presents the rest of the data collected for the experiments.

5.6 Data Analysis

To answer our research questions, we deployed Bayesian statistical models. These models are governed by the following relationship between a posterior distribution, a likelihood function, and a prior distribution [47], [48]:

$$P(\theta|D) \propto P(D|\theta) \times P(\theta)$$

The term $P(D|\theta)$ is the *likelihood* function that given a set of parameters θ, estimates how well θ describes the data D. We multiply this term with a *prior* distribution $P(\theta)$ that describes how plausible we believe the chosen parameters θ to be. The product between the likelihood function and prior distribution is proportional $\propto$ to the *posterior* distribution $P(\theta|D)$ that describes how likely the parameter values θ are given our data D. In Bayesian statistics, parameters are modeled after distributions rather than assuming specific values. We do not believe that a "true" parameter value exists, but some values are more probable than others, therefore we use probability distributions to reflect this belief.

5.6.1 Example Bayesian Model

$$\begin{aligned} U_{ijk} &\sim \text{Normal}\,(\mu, \sigma) \\ \mu &= \alpha_i + \beta_j \\ \alpha_i &\sim \text{Normal}(0, 0.5) \quad , \text{ for } i = 1..5 \\ \beta_j &\sim \text{Normal}(0, \sigma_\beta) \quad , \text{ for } j = 1..7 \\ \sigma_\beta &\sim \text{Exponential}(3) \\ \sigma &\sim \text{Gamma}(2, 0.5) \end{aligned} \tag{5.1}$$

The Bayesian models we will present are defined using mathematical notation. To

understand this type of notation we will use a **made up** example and experiment to build a simplified model as seen in Equation 5.1. Consider an outcome variable $U_{ijk} \in \mathbb{R}$ that has been measured each time we have conducted one of our experiments.

$$U_{ijk} \sim \text{Normal}\,(\mu, \sigma)$$

The index terms ijk represent the i-th experiment setting, j-th technique used, and k-th trial in our example experiment. For this example we only know that U_{ijk} is a real number and that it has a finite variance, i.e., most values lie between $[-10, 10]$. Given these known characteristics of U_{ijk} we can use the maximum entropy principle (MEP), which states that we want to choose a likelihood that covers all known characteristics but makes the least amount of additional assumptions about the data [47]. In this particular case, MEP guides us towards the Normal distribution as our likelihood, $\mathcal{N}(\mu, \sigma)$. The likelihood function tries to quantify how well the parameters μ (the mean) and σ (the standard deviation), describe our observed data U_{ijk}.

All our parameters have a prior distribution that encodes our beliefs about what values these parameters can assume. The standard deviation σ, in this case, has a Gamma-distributed prior:

$$\sigma \sim \text{Gamma}(2, 0.5)$$

It can be hard to understand the shape of a distribution by purely seeing it written down. Instead, it is common to conduct something known as a *prior check*, where we randomly sample a large number of values from our prior distribution and then plot a histogram to show the frequency of values. In Appendix D, we randomly sample 10,000 values from $\text{Gamma}(2, 0.5)$, and plot their frequency as seen in Figure D.1a. What we can see is a distribution that has a peak around 3 and extends out to 15. However, the majority of values sampled lie in the range 0 to 7.5. One way to interpret this is that we believe that the standard deviation for our Normal-distributed likelihood is focused around 3, but has a long thin right tail to indicate larger values are possible but with less probability. While a prior reflects our initial beliefs about where the value of σ lies, it is just an initial guess, and with enough evidence, the model can find a more suitable value outside this guess.

Turning our attention to μ, it is defined as the sum of two other parameters:

$$\mu = \alpha_i + \beta_j$$

α_i has a Normal-distributed prior with a mean of 0 and a standard deviation of 0.5.

A prior check of this distribution can be seen in Figure D.1b, where most values land between $[-1, 1]$. In a normal distribution, approximately 95% of values land two standard deviations away from the mean [47]. An important detail to clarify is that we have an α value per setting i. Since our hypothetical experiment has five settings, we also have five α values: α_1, α_2, α_3, α_4, α_5. Thus, we can measure the effect each setting has on the outcome variable U. Depending on which setting i our measured outcome variable U_{ijk} belongs to, the corresponding α_i will be used to compute μ. For example, U_{2jk} will use α_2 to compute μ, thus we will end up with as many μ as data points in U_{ijk}.

One issue with allowing each setting to estimate its own effect is the negligence of any similarities between them. This can become problematic in cases where groups (e.g., setting) have few samples, which makes the effect of the setting more sensitive to outliers. To solve this issue β_j deploys a different tactic. Similarly to α_i, β_j has multiple groups (seven techniques j) with Normally distributed priors with mean 0. However, their standard deviation depends on another parameter σ_β to incorporate their inherent similarities. The idea is that σ_β is learned on the whole population, so if any group is significantly better than the rest, without sufficient evidence, that group will be pulled more towards the mean. This effect is known as *shrinkage* and is one of the strengths in *hierarchical* or *multi-level* models [47], i.e., where priors have their own priors. In our models, we will make extensive use of this idea.

σ_β has its own prior, which in this example is an Exponential distribution with rate $\lambda = 3$. A prior check for this kind of distribution is displayed in Figure D.1c, where 95% of the values lie in the interval $[0,\ 1]$[4]. Any sampled σ_β will act as the as the standard deviation when sampling β_j. Conducting a prior check for β_j as shown in Figure D.1d, most values from this prior check are centered around the interval $[-0.5, 0.5]$ with diminishing potential to extend further.

Once we have defined our priors for α_i, β_j, and σ_β, we can sample them to calculate μ. The prior check for μ is showcased in Figure D.2 and here we observe that the majority of sampled values for μ is located in the range $[-1, 1]$, with lessened probability going outside this range.

To validate a model, it is common to conduct a *prior predictive simulation* [47]. To perform such a simulation for this example, we randomly sample our parameters μ and σ thousands of times in accordance with their prior distributions. We pair values of μ and σ together, use them in our likelihood function (i.e., a Normal distribution), and randomly sample that resulting distribution once. This process will be repeated for each pair of μ and σ values, and the final product is a simulated data set telling us what kind of data our model expects. An illustration of such a simulation for 10,000 samples can be seen in Figure 5.2. As we can see, the model expects data to be somewhere in the range of $[-10, 10]$.

[4]To calculate the 95% interval we used $\frac{-\ln(1-0.95)}{3} \approx 1$.

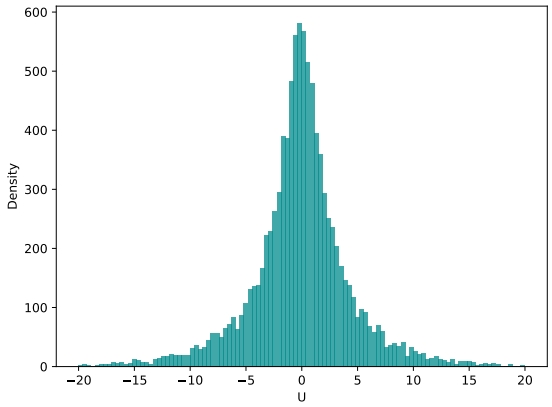


Figure 5.2: A prior predictive simulation for the example Bayesian model presented in Equation 5.1.

5.6.2 Bayesian Model for RQ1

Research Question 1 explores if there is a notable difference in the final branch coverage achieved after deploying our parameter control system. We have derived a hierarchical Bayesian model as seen in Equation 5.2 that addresses three aspects of the overall research question:

- What is the expected final branch coverage for a module?
- What is the overall effect on the final branch coverage of the parameter(s) being controlled?
- How do the individual modules respond to the specific parameter(s) being controlled?

$$
\begin{aligned}
C_{mpk} &\sim \text{Beta}\,(\bar{p}, \theta) \\
\text{logit}(\bar{p}) &= \alpha_m + \beta_p + \gamma_{mp} \\
\alpha_m &\sim \text{Normal}(\bar{\alpha}, \sigma_\alpha) \quad \text{, for } m = 1..24 \\
\beta_p &\sim \text{Normal}(0, \sigma_\beta) \quad \text{, for } p = 1..12 \text{ or } 1..66 \\
\gamma_{mp} &\sim \text{Normal}(0, \sigma_\gamma) \quad \text{, for } mp = 1..288 \text{ or } 1..1584 \\
\theta &\sim \text{Gamma}(6.0, 0.1) \\
\\
\bar{\alpha} &\sim \text{Normal}(0, 1.5) \\
\sigma_\alpha &\sim \text{Exponential}(2.0) \\
\sigma_\beta &\sim \text{Exponential}(5.0) \\
\sigma_\gamma &\sim \text{Exponential}(5.0)
\end{aligned}
\tag{5.2}
$$

Our observed variable is the final branch coverage C_{mpk}, given a module m, con-

trolled parameter(s) p, and repetition k. Coverage C is measured as a percentage, lying in the interval interval $[0, 1]$ making a Beta distribution a good alternative for a likelihood function. The Beta distribution is parameterized with the mean $\bar{p}$ and a dispersion parameter θ. The prior for θ is modeled after a Gamma distribution with $\alpha = 6$ and $\beta = 0.1$ for three reasons:

- A Gamma distribution results in θ always being positive, which is required by the Beta distribution.
- Our initial tests showed that individual modules consistently reached a similar coverage between runs, indicating a low variance. Thus, we expect θ to be a larger value (e.g., around 50), since larger θ values result in a lower variance for our Beta likelihood around the mean.
- A Gamma distribution allows for a shorter left tail in comparison to its right tail, to indicate more skepticism towards values closer to zero.

The result after conducting a prior check for θ can be seen in Figure E.1a (in Appendix E), where we expect a peak around 50 with fast decreasing plausibility towards zero and a more generous expectation towards higher values.

$\bar{p}$ is the sum of a module intercept α_m, parameter(s) effect β_p, and an interaction effect γ_{mp}. The *logit* that surrounds $\bar{p}$ is known as a link function, responsible for translating the linear combination $(\alpha_m + \beta_p + \sigma_{mp}) \in (-\infty, \infty)$ to a valid probability $[0, 1]$ [47]. Without a link function, the sum can exceed a valid probability. Thus, a common alternative is to model probabilities as log-odds, Logarithmic Odds, where the odds are the probability of an event happening divided by the probability of it not happening $odds = \frac{p}{1-p}$. The inverse of the logit transformation is the sigmoid function $\sigma(x) = \frac{1}{1+e^x}$, which is illustrated in Figure 5.3. $\sigma(x)$ is bounded between $[0, 1]$, and most probabilities, as seen on the Y-axis, can be represented with log-odds between $[-4, 4]$ where $\sigma(-4) \approx 1.8\%$ and $\sigma(4) \approx 98.2\%$. 0.5 probability is represented as 0 in log-odds, and given the non-linear nature of the sigmoid function $\sigma(x)$, any subsequent increase or decrease from this point onward has a diminishing effect on the change in probability.

α_m has a Normal-distributed prior with mean $\bar{\alpha}$ and standard deviation σ_α. The idea behind the multi-level approach is to let the whole dataset determine the mean final branch coverage $\bar{\alpha}$ and standard deviation σ_α, to recognize that there are similarities between modules. Without sufficient evidence, module intercepts will shrink towards the mean. $\bar{\alpha}$ has a Normal-distributed prior with a mean of 0 and a standard deviation of 1.5. We can see the resulting prior check for $\bar{\alpha}$ in Figure E.1b, where 95% of values land between $[-3, 3]$. Module intercepts are represented in the log-odds scale, thus we believe that the mean coverage is likely between $\sigma(-3) \approx 5\%$ and $\sigma(3) \approx 95\%$ coverage. This is a relatively uninformative prior, however, we do know from our test runs that the final coverage varies greatly between modules, covering a majority of the $[0\%, 100\%]$ interval.

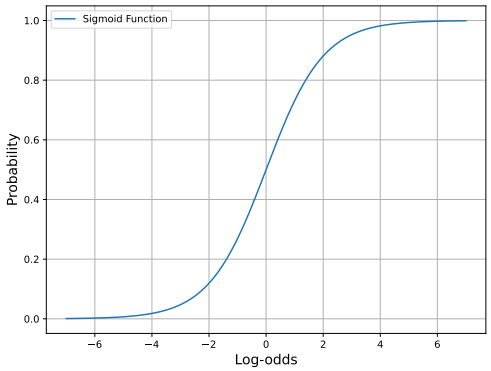


Figure 5.3: Plot showcasing a sigmoid function (inverse-logit).

σ_α has an Exponential-distributed prior with rate $\lambda = 2$. The resulting prior check, as seen in Figure E.1c, hints towards a likely standard deviation around $\bar{\alpha}$ somewhere between $[0, 1.5]$. Now that both $\bar{\alpha}$ and σ_α have been defined, we can conduct a prior check for α_m as displayed in Figure E.1d. We can observe from the graph that most values land comfortably between $[-3, 3]$ with more skepticism towards $[-4, 4]$ and beyond. This prior check supports our understanding that the final branch coverage can vary significantly depending on the module.

The parameter(s) effect β_p and the interaction effect γ_{mp} have identical priors consisting of a Normal distribution centered around 0 with a standard deviation σ_β or σ_γ, respectively. The priors for σ_β and σ_γ follow an Exponential distribution with a rate $\lambda = 5$. These are aggressive priors and are visualized in Figure E.2a. Anything beyond the interval $[0, 0.5]$ is met with harsh skepticism. Our reason for these priors is that we hypothesize that the overall effect of parameter control will be quite small on the final branch coverage. By conducting a prior check (see Figure E.2b) for β_p and γ_{mp}, the bulk of values land in the range $[-0.3, 0.3]$ log-odds. It is important to remember that log-odds is not a linear translation and effects are relative. For example, an increase by 0.25 in the close vicinity of zero, results in a difference of 6 percent units $\sigma(0 + 0.25) - \sigma(0) \approx 6\%$. Compare that with an increase further away at three which only changes around 1 percent units, $\sigma(3 + 0.25) - \sigma(3) \approx 1\%$.

Now that we have defined all components of $\bar{p} = \alpha_m + \beta_p + \sigma_{mp}$, we can run a prior check. It can be beneficial to understand the prior check for $\bar{p}$ in log-odds scale first, which is illustrated in Figure E.2c. As evident in the histogram, before applying the link function to $\bar{p}$ most values land between $[-3, 3]$ with a small probability of extending to $[-5, 5]$. Applying a link function to $\bar{p}$ as seen in Figure E.2d, this translates to a flat-looking distribution, indicating that most $\bar{p}$ values are indeed possible with a slight decrease at the edges. Finally, it is a good idea to conduct a prior predictive simulation to see what data our model expects. The generated dataset is illustrated in Figure E.3, reflecting our belief that the modules are varied in their final coverage.

5.6.3 Bayesian Model for RQ2

The idea behind Research Question 2 is to see if the rate at which branch coverage is obtained is impacted when deploying our parameter control system. While the final branch coverage might be the same in the end, the rate at which it is achieved can be faster or slower. To answer this research question, we have derived a hierarchical Bayesian statistical model as seen in Equation 5.3, that addresses three aspects of the overall research question:

- What is the expected branch coverage growth rate for a module?
- What is the overall effect on the branch coverage growth rate of the parameter(s) being controlled?
- How do the individual modules respond to the specific parameter(s) being controlled?

For every run, we recorded a timeline of the best obtained branch coverage at one-second intervals. By integrating this interval using the trapezoid rule [49], we can compute a value representing the branch coverage growth rate over the 300-second interval. A value of 0 would indicate that the generated tests covered no branches in the code, while a value of 300 would mean that 100% coverage was achieved on the first iteration. Using integration retains information from the whole interval, not just the endpoints.

$$
\begin{aligned}
CGR_{mpk} &\sim \text{Normal}\,(\mu, \sigma) \\
\mu &= \alpha_m + \beta_p + \gamma_{mp} \\
\alpha_m &\sim \text{Normal}(\bar{\alpha}, \sigma_\alpha) \quad , \text{ for } m = 1..24 \\
\beta_p &\sim \text{Normal}(0, \sigma_\beta) \quad , \text{ for } p = 1..12 \text{ or } 1..66 \\
\gamma_{mp} &\sim \text{Normal}(0, \sigma_\gamma) \quad , \text{ for } mp = 1..288 \text{ or } 1..1584 \\
\sigma &\sim \text{Exponential}(0.1)
\end{aligned}
\tag{5.3}
$$

$$
\begin{aligned}
\bar{\alpha} &\sim \text{Normal}(150, 30) \\
\sigma_\alpha &\sim \text{Exponential}(0.03) \\
\sigma_\beta &\sim \text{Exponential}(0.1) \\
\sigma_\gamma &\sim \text{Exponential}(0.1)
\end{aligned}
$$

Our observed variable in our model is branch Coverage Growth Rate CGR_{mpk}, given a module m, controlled parameter(s) p, and repetition k. The maximum entropy principle guides us towards a Normal-distributed likelihood since it makes the least amount of assumptions beyond the fact that our data has finite variance [47]. While a Normal distribution does not possess any boundaries, unlike our data which is constrained between $[0, 300]$, we can still shape the distribution to sufficiently satisfy this boundary.

The likelihood function has two parameters that need to be estimated, μ and σ. The prior for σ is an Exponential distribution with rate $\lambda = 0.1$. Performing a prior check for σ, see Figure F.1a, yields that 95% of all values lie in the range $[0, 30]$. We assume that each module will retain a similar branch coverage growth rate across all runs, which makes the Exponential distribution with rate $\lambda = 0.1$ a fitting alternative.

The mean μ is defined as the linear combination between module intercept α_m, parameter(s) effect β_p, and interaction effect γ_{mp}. The module intercept α_m has a Normal-distributed prior made up of two other parameters, mean $\bar{\alpha}$ and standard deviation σ_α. $\bar{\alpha}$ captures the average module branch coverage growth rate and has a Normal-distributed prior with a mean of 150 and a standard deviation of 30. Conducting a prior check for $\bar{\alpha}$, as demonstrated in Figure F.1b, reveals that 95% of values land between 90 and 210. During our test runs, we observed that the final branch coverage had a large variance between modules. Thus, it is fair to assume that the branch coverage growth rate will behave similarly, explaining our wide prior.

σ_α has an Exponential-distributed prior with rate $\lambda = 0.03$. Inspecting the prior check as demonstrated in Figure F.1c, plausible values for σ_α are in the range $[0, 100]$. This choice of prior for the standard deviation σ_α is to allow for potential large variance between the different modules. Now that $\bar{\alpha}$ and σ_α have been defined, we can see how they impact the module intercept α_m's prior check as plotted in Figure F.1d. The plot indicates that a majority of values lie between $[50, 250]$ with a possibility of extending out to $[0, 300]$ and beyond. Given our expectation that the branch coverage growth rate will vary greatly between modules, this interval is reasonable.

The parameter(s) effect β_p and the interaction effect γ_{mp} have zero-centered Normal distributions with standard deviation σ_β or σ_γ respectively, as priors. Priors for σ_β and σ_γ are both Exponential-distributed, with rate $\lambda = 0.1$. Plotting the prior checks, see Figure F.2a, reveal that the most probable values are in the $[0, 30]$ interval with more concentration closer to zero. Taking σ_β and σ_γ into consideration and plotting the prior check for both β_p and γ_{mp}, the expected effect of parameter control on branch coverage growth rate is somewhere in the interval $[-20, 20]$.

Plotting the prior check for the linear combination $\mu = \alpha_m + \beta_p + \gamma_{mp}$ in Figure F.2c shows us that a majority of values for μ are in the range $[50, 250]$ with diminishing probability for further extension. Conducting a prior predictive simulation for the entire model, we see a very similar pattern emerge in Figure F.3. This pattern matches our expectation that a large range of values is possible, but branch coverage growth rates closer to zero or 300 are less likely.

5.6.4 Bayesian Model for RQ3

The final research question deals with how much overhead, or rather, the computational impact our parameter control system imposes. To measure this, we recorded the number of test generation iterations completed during the 300-second time budget for every run. The idea is to see if there is any significant reduction or increase in the number of iterations when utilizing parameter control during the test generation process. We constructed a hierarchical Bayesian statistical model as demonstrated in Equation 5.4, that addresses three aspects of the overall research question:

- What is the expected number of iterations completed for a module?
- What is the overall impact on the number of iterations completed of the parameter(s) being controlled?
- How does the number of iterations completed for an individual module respond to the specific parameter(s) being controlled?

$$
\begin{aligned}
I_{mpk} &\sim \text{NegativeBinomial}\,(\mu, \theta) \\
\log(\mu) &= \alpha_m + \beta_p + \gamma_{mp} \\
\alpha_m &\sim \text{Normal}(\bar{\alpha}, \sigma_\alpha) \quad , \text{ for } m = 1..24 \\
\beta_p &\sim \text{Normal}(0, \sigma_\beta) \quad , \text{ for } p = 1..12 \text{ or } 1..66 \\
\gamma_{mp} &\sim \text{Normal}(0, \sigma_\gamma) \quad , \text{ for } mp = 1..288 \text{ or } 1..1584 \\
\theta &\sim \text{Gamma}(5.0, 0.1) \\
\\
\bar{\alpha} &\sim \text{Normal}(7, 0.5) \\
\sigma_\alpha &\sim \text{Exponential}(4) \\
\sigma_\beta &\sim \text{Exponential}(4) \\
\sigma_\gamma &\sim \text{Exponential}(4)
\end{aligned}
\tag{5.4}
$$

Our observed variable is the number of Iterations I_{mpk}, a discrete number with index terms, module m, controlled parameter(s) p, repetition k. For discrete data with an unknown upper bound, the Maximum Entropy Principle advises us towards a Poisson distribution as the likelihood function [47]. However, this choice of likelihood assumes that the mean is roughly equal to the variance, which we cannot comfortably state. A common substitution for Poisson is the closely related Negative Binomial distribution, which allows us to estimate a mean μ as well as a shape parameter θ [48].

Starting with the shape parameter θ. When θ is a value close to or equal to zero, the shape of the distribution resembles that of an Exponential distribution. While, if we increase θ, the majority of values will move towards the mean μ and the shape will more closely resemble a Gaussian distribution. In our model, θ has a Gamma-distributed prior with $\alpha = 5.0$ and $\beta = 0.1$, and the resulting prior check can be seen

in Figure G.1a. The mode of the distribution is around 40, with quickly diminishing plausibility towards values closer to zero and higher expectations towards larger values up to 100 and onward. This agrees with our belief that a module's iterations are more likely to be focused around the mean μ.

The mean of the Negative Binomial likelihood μ is the linear combination between module intercept α_m, parameter(s) effect β_p, and interaction effect γ_{mp}. By applying a log link function to the mean μ we accomplish two things: it is always strictly positive and we can keep the same linear combination relationship between intercept and effects as in the other models. The module intercept α_m has a Normal-distributed prior with two parameters, mean $\bar{\alpha}$ and standard deviation σ_α. $\bar{\alpha}$ captures the iteration mean of the whole population, which has a Normal-distributed prior with a mean of 7 and a standard deviation of 0.5. During our experimentation, we noticed that most modules reached numbers somewhere between 500 to 2000 iterations with the potential to reach even higher numbers. For a module to qualify to be a part of our experiment, it had to consistently reach a minimum of 500 iterations in our test runs. Conducting a prior check for $\bar{\alpha}$ as visualized in Figure G.1b, we notice that most values land in the interval $[6, 8]$. $\bar{\alpha}$ is represented in the natural logarithm scale, translating the observed interval to the number of iterations we get the following results: $e^6 \approx 400$, $e^7 \approx 1100$ and $e^8 \approx 3000$, making this interval reasonable based on our prior knowledge.

Parameter(s) effect β_p and interaction effect γ_{mp} both have Normal-distributed priors centered around 0 with standard deviation σ_β and σ_γ respectively. σ_α, σ_β, and σ_γ use identical priors consisting of an Exponential distribution with rate $\lambda = 4$, as demonstrated in Figure G.1c. This prior check shows that 95% of values lie in the range $[0, 0.75]$ with an increased expectation for values closer to zero. We can see what this implies for the module intercept α_m by inspecting its prior check, see Figure G.1d, where the most probable values lie between $[6, 8]$, which translates to $[400, 3000]$ iterations. Inspecting the prior check for β_p and γ_{mp} as illustrated in Figure G.2a, we are skeptical that parameter control will exceed the boundary $[-0.4, 0.4]$ which translates to a decrease of 33% or an increase of 49% in the total number of iterations.

Conducting a prior check for the linear combination $\mu = \alpha_m + \beta_p + \gamma_{mp}$ before applying the link function can be seen in Figure G.2b, where most values lie in the range $[5.5, 8.5]$. After applying the link function, the resulting prior check can be seen in Figure G.2c, where most probable values for μ lie in the range of $[250, 4900]$ iterations, with the majority lying in the $[250, 3000]$ range. Performing a prior predictive simulation to validate our whole model, see Figure G.3, we can observe a peak close to 1000, with a quickly diminishing probability towards zero, while a longer right tail indicates the inclusion of modules reaching higher iterations.

5.6.5 Reparameterization

One striking resemblance between all three models presented in Equations 5.2, 5.3, and 5.4 is their structure. They all virtually include the same parameters and the relationships between them. This is no coincidence, what we have been utilizing is something known as Generalized Linear Models [47], [48]. The only real difference is the choice of likelihood function as well as a link function that is responsible for keeping the linear combination of module intercept α_m, parameter(s) effect β_p, and interaction effect γ_{mp} within acceptable bounds. This consistency facilitates easier interpretation of the results.

In addition to being generalized linear models, all three models are also multi-level models, where the prior for a parameter is dependent on another parameter's prior (e.g., the prior for α_m is dependent on both $\bar{\alpha}$ and σ_α). A regularly occurring issue when sampling multi-level models is divergent transitions, that indicate a difficulty in exploring the posterior distribution space [47], [48]. To address this issue, we can change how the models are expressed so they become easier to explore, while keeping them mathematically equivalent. For example, recall our prior for the module intercept α_m from Section 5.6.2 and how that is dependent on $\bar{\alpha}$ and σ_α (see Equation 5.5). This is what is known as centered parameterization, where α_m's prior is directly dependent on both $\bar{\alpha}$ and σ_α, making it difficult to traverse the posterior distribution.

$$\begin{aligned} \alpha_m &\sim \text{Normal}(\bar{\alpha}, \sigma_\alpha) \\ \bar{\alpha} &\sim \text{Normal}(0, 1.5) \\ \sigma_\alpha &\sim \text{Exponential}(2.0) \end{aligned} \tag{5.5}$$

However, there is an equivalent way to write this where α_m's prior is not a function of two other parameters, known as non-centered parameterization (see Equation 5.6) [47]. We can define α_m as $\bar{\alpha}$ plus the product of the standard deviation σ_α and the α_{offset}. Rewriting the priors in this way retains the mathematical relationship while making it easier for the model to explore possible parameter values.

$$\begin{aligned} \alpha_m &= \bar{\alpha} + \sigma_\alpha \times \alpha_{offset} \\ \bar{\alpha} &\sim \text{Normal}(0, 1.5) \\ \sigma_\alpha &\sim \text{Exponential}(2.0) \\ \alpha_{offset} &\sim \text{Normal}(0, 1) \end{aligned} \tag{5.6}$$

We apply this same principle to the rest of our parameters β_p and γ_{mp} to create a non-centered parameterized version of all our models. When sampling the models, we are using the non-centered versions, while we present the models as centered in this thesis. This discrepancy exists solely to make the models easier to understand in the thesis while running the reparameterized version to avoid divergent transitions. Both model definitions are equivalent.

6

Results

This chapter presents the results from the single- and multi-parameter experiment analyses and relates these to the research questions.

6.1 Final Branch Coverage (RQ1)

Two models were run, one for the single-parameter experiment and one for the multi-parameter experiment, attempting to establish if any change in the final branch coverage of a module could be observed after deploying our parameter control system. The models were run using four chains and 6,000 samples each, with 1,000 used as tuning samples for warm-up. Both models' chains converged successfully, without any divergent transitions, for all parameters, which can be seen in the tables located in Appendix E, indicated by $\hat{R}$ values of 1.0. Additionally, the effective sample size overwhelmingly exceeds the recommended amount of 10% of total samples [48] and the Monte Carlo Standard Error (MCSE) is zero for the vast majority of parameters which indicates that the chains have sufficient accuracy [50].

For a complete breakdown of priors checks, trace plots, summary tables, and posterior distributions we refer you to our Jupyter notebooks hosted on GitHub[1,2].

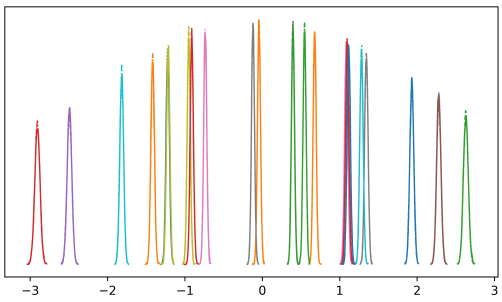
6.1.1 Single-Parameter Experiment (RQ1.1)

Analyzing the result for the single-parameter final branch coverage model (see Section 5.6.2), we can start by looking at the intercept for each module. In Figure 6.1 the posterior distributions for the 24 modules' intercepts are displayed. The Bayesian model works in the log-odd scale as seen in Figure 6.1a, however, a more intuitive way is to transform the log-odds to probabilities as shown in Figure 6.1b. Two clear patterns can quickly be established by investigating these posterior distributions. First, the variance between all module intercepts is very large, and they do not share a common center point but are evenly spread out across the entire probability

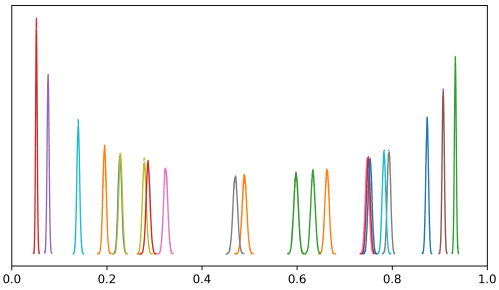
[1] `https://github.com/henkejson/analysis-adaptive-parameter-control/blob/main/Notebooks/final_coverage_single_parameter.ipynb`

[2] `https://github.com/henkejson/analysis-adaptive-parameter-control/blob/main/Notebooks/final_coverage_multi_parameter.ipynb`

scale. The second characteristic is the narrowness of all the posterior distributions, hinting towards large amounts of confidence in the predicted values for the module intercepts. In Appendix E, Table E.1 shows the mean, standard deviation, and 89% credible interval for each respective distribution, sorted in ascending order by the mean. By sorting it in ascending order, the left-most distribution in Figure 6.1a will be first in the table, while the last entry will be the furthest to the right.



(a) Logit (log-odds) scale.



(b) Probability scale.

Figure 6.1: Module intercept (α_m) posterior distributions for the single-parameter final branch coverage model.

The module intercepts show the expected amount of branch coverage a module can achieve without deploying parameter control. To understand what the overall effect of controlling a single parameter has on the final branch coverage, we turn to Figure 6.2. The figure shows the posterior distributions of the *overall* parameter effect β_p for each of the 12 parameters (in log-odds scale). The common thread for all of these posterior distributions is that they are centered around zero, indicating that a statistically significant effect cannot be determined for any of the parameters controlled. Table E.2 in Appendix E, presents the mean, standard deviation, and 89% credible interval for the the parameter effect posterior distributions (sorted in ascending order according to the mean). In the table we can confirm that all 89% credible intervals contain zero, meaning no statistically significant relationship can be determined.

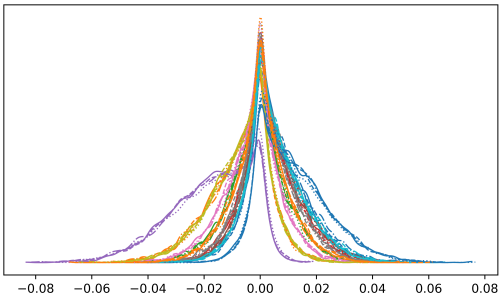


Figure 6.2: Parameter effect (β_p) posterior distributions for the single-parameter final branch coverage model. Logit (log-odds) scale.

While we did not witness an *overall* parameter effect on final branch coverage, if we instead consider the interaction effect, i.e., how specific modules reacted to

certain controlled parameters, we can see that some modules are more receptive to parameter control. In Figure 6.3, we see that a majority of the 288 interaction effects are centered around zero, indicating no statistically significant effect. However, there are a few exceptions.

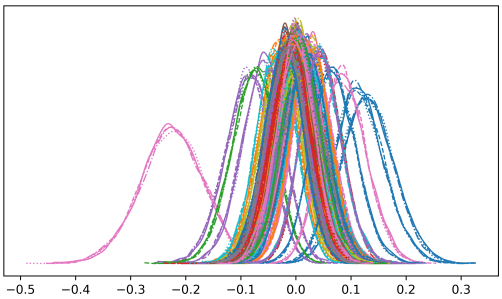


Figure 6.3: Interaction effect (γ_{mp}) posterior distributions for the single-parameter final branch coverage model. Logit (log-odds) scale.

By investigating Table E.3 that presents the mean, standard deviation, and 89% credible interval for the posterior distributions, we can see that eight of the posterior distributions have credible intervals that do not contain zero. These eight interaction effects are showcased in Table 6.1, with module intercepts (expected coverage) as well as the relative effects of controlling a specific parameter. To calculate the relative effects for each interaction effect, we took the corresponding module intercept and added the ends of the credible interval. For example, in the first row `g_mp[return_from_generator x Pop]`, the module `a_m[return_from_generator]` has an intercept of -1.222 (see Table E.1). We added the lower limit of the 89% credible interval from the interaction effect to the intercept before applying the sigmoid function to convert it to a probability. The same procedure is repeated for the upper limit of the 89% credible interval. From these values, we can derive how many percentage points the final branch coverage shifts when performing parameter control for a specific module and parameter combination.

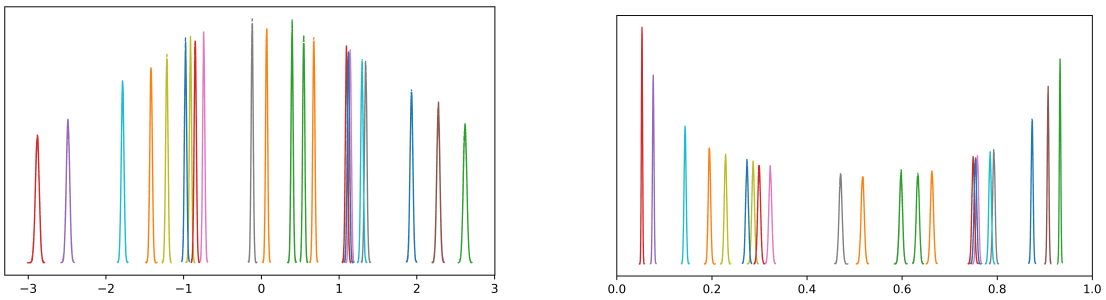
Module and parameter pair	**Module intercept**	**Relative effect** (5.5%)	**Relative effect** (94.5%)	Percentage point difference
g_mp[return_from_generator x Pop]	22.76%	17.69%	20.52%	[-5.07, -2.24]
g_mp[return_from_generator x ChromLen]	22.76%	20.18%	22.36%	[-2.58, -0.4]
g_mp[return_from_generator x TestDeleteProb]	22.76%	20.38%	22.51%	[-2.38, -0.25]
g_mp[return_from_generator x StatemInsertProb]	22.76%	20.65%	22.76%	[-2.11, 0]
g_mp[return_from_generator x TestInsertProb]	22.76%	23.13%	25.5%	[0.37, 2.74]
g_mp[return_from_generator x TestInsertionProb]	22.76%	22.86%	25.18%	[0.1, 2.42]
g_mp[return_from_generator x RandPert]	22.76%	23.49%	26.13%	[0.73, 3.37]
g_mp[return_from_generator x ChangeParamProb]	22.76%	23.79%	26.54%	[1.03, 3.78]

Table 6.1: Significant interactions between modules and parameters for the single-parameter final branch coverage model (see Appendix B for full names of modules and parameters).

Based on the results presented in Table 6.1, controlling Population Size, Chromosome Length, Test Delete Probability or Statement Insert Probability has a statistically significant *negative* impact on the final branch coverage for the module `return_from_generator` final branch coverage. Meanwhile, Test Insert Probability, Test Insertion Probability, Random Perturbation, and Change Parameter Probability have a *positive* impact on the final branch coverage for the same module.

6.1.2 Multi-Parameter Experiment (RQ1.2)

For the multi-parameter experiment, the module intercepts presented in Figure 6.4 closely resemble the result from the single-parameter experiment (see Figure 6.1). The posterior distributions have a sharp and narrow shape indicating a high confidence in the estimated module intercepts. Furthermore, the intercepts are evenly spaced across the entire scale, highlighting the inherent differences in final branch coverage between the modules. In Table E.4 (see Appendix E) we present the mean, standard deviation, and 89% credible interval for each of these posterior distributions sorted in ascending order according to the mean.



(a) Logit (log-odds) scale. **(b)** Probability scale.

Figure 6.4: Module intercept (α_m) posterior distributions for the multi-parameter final branch coverage model.

Investigating the *overall* parameter effect for the multi-parameter experiment once again paints a similar picture to the single-parameter experiment. In Figure 6.5 we can see the posterior distributions for all the 66 parameter combinations' effect on the final branch coverage. The posterior distributions are centered around 0, indicating no statistically significant overall effects on the final branch coverage when controlling two parameters at the same time. Table E.5 showcases the mean, standard deviation and 89% credible interval for these distributions, confirming that all distributions include zero.

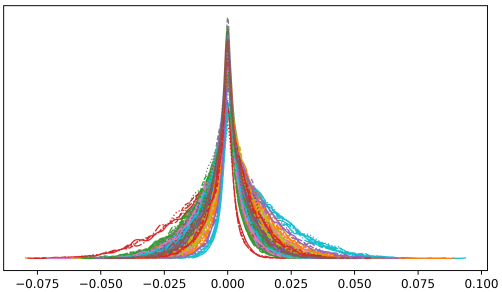


Figure 6.5: Parameter effect (β_p) posterior distributions for the multi-parameter final branch coverage model. Logit (log-odds) scale.

Similarly to the single-parameter experiment, there are some significant interaction effects between modules and pairs of parameters. Figure 6.6 presents the posterior

distributions for the interaction effects, where we can see that some distributions are not centered around zero. To identify these interactions, we can consider Table E.6, which showcases the mean, standard deviation, and 89% credible intervals for the posterior distributions. From these interactions, we have collected the 16 statistically significant ones in Table 6.2 with the module intercepts and the relative effect for these interactions. These are calculated in the same way as the significant interaction effects in Section 6.1.1.

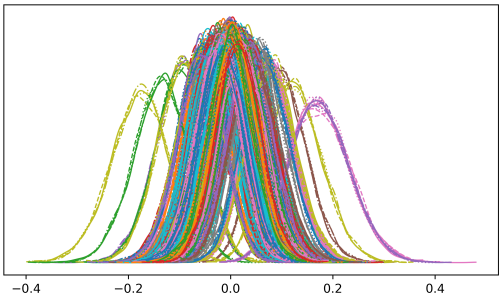


Figure 6.6: Interaction effect (γ_{mp}) posterior distributions for the multi-parameter final branch coverage model. Logit (log-odds) scale.

Of the 16 statistically significant interaction effects, 15 are for the same module, `return_from_generator`. This is no different than for the single-parameter experiment where the same module was the only one reporting a significant result. Depending on the parameter pair, this module has both positive and negative impacts on the final branch coverage. The final significant interaction effect is for a different module, `positional_validation`, which for the specific parameter combination of Chromosome Length and Population Size results in an improvement of the final branch coverage achieved.

Module and parameters pair	**Module intercept**	**Relative effect** (5.5%)	**Relative effect** (94.5%)	Percentage point difference
g_mp[return_from_generator x ChromLen|TestDeleteProb]	27.39%	22.46%	25.73%	[-4.93, -1.66]
g_mp[return_from_generator x Elite|TestChangeProb]	27.39%	23.27%	26.37%	[-4.12, -1.02]
g_mp[return_from_generator x ChromLen|Crossover]	27.39%	23.99%	26.91%	[-3.4, -0.48]
g_mp[return_from_generator x Crossover|RandPert]	27.39%	24.03%	26.93%	[-3.36, -0.46]
g_mp[return_from_generator x Elite|TourSize]	27.39%	24.10%	26.97%	[-3.29, -0.42]
g_mp[return_from_generator x StatemInsertProb|TestInsertionProb]	27.39%	27.41%	30.43%	[0.02, 3.04]
g_mp[return_from_generator x ChromLen|RandPert]	27.39%	27.43%	30.47%	[0.04, 3.08]
g_mp[return_from_generator x Crossover|TestInsertionProb]	27.39%	27.41%	30.47%	[0.02, 3.08]
g_mp[return_from_generator x ChangeParamProb|TestInsertionProb]	27.39%	27.47%	30.58%	[0.08, 3.19]
g_mp[return_from_generator x Crossover|TourSize]	27.39%	27.43%	30.56%	[0.04, 3.17]
g_mp[return_from_generator x TestChangeProb|TestInsertProb]	27.39%	27.45%	30.62%	[0.06, 3.23]
g_mp[positional_validation x ChromLen|Pop]	29.94%	30.32%	33.69%	[0.38, 3.75]
g_mp[return_from_generator x Crossover|TestInsertProb]	27.39%	28.19%	31.71%	[0.8, 4.31]
g_mp[return_from_generator x RandPert|TestInsertionProb]	27.39%	28.97%	32.94%	[1.58, 5.55]
g_mp[return_from_generator x StatemInsertProb|TestChangeProb]	27.39%	28.95%	32.98%	[1.56, 5.59]
g_mp[return_from_generator x ChromLen|TestInsertionProb]	27.39%	29.01%	33.07%	[1.62, 5.68]

Table 6.2: Significant interactions between modules and parameter combinations for the multi-parameter final branch coverage model (see Appendix B for full names of modules and parameters).

6.2 Branch Coverage Growth Rate (RQ2)

To establish if any difference could be observed regarding branch coverage growth rate, we ran one Bayesian statistical model each for the single- and multi-parameter experiments. The models ran four chains, picking 6,000 samples each, where 1,000

were discarded as warm-up. The parameters were estimated accurately and successfully as observable by looking at the Monte Carlo Standard Error (MCSE), Effective Sample Size (ESS), and $\hat{R}$ values in the tables presented in Appendix F. All four chains successfully converged, indicated by $\hat{R} = 1.0$, with minimal MCSE and with no divergent transitions. Additionally, the sampling was very efficient as can be seen by inspecting the ESS values, overwhelmingly exceeding the recommended minimum amount of 10% of total samples [48].

For a complete breakdown of priors checks, trace plots, summary tables, and posterior distributions we refer you to our Jupyter notebooks hosted on GitHub[3,4].

6.2.1 Single-Parameter Experiment (RQ2.1)

We start by understanding how the module intercepts' posterior distributions are shaped, as seen in Figure 6.7. What is evident is the uniform spread of the posterior distributions, indicating vastly different branch coverage growth rates across modules. Additionally, the posterior shapes are extremely narrow, showing high confidence in the estimated module intercepts. Table F.1 in Appendix F presents a summary of the mean, standard deviation, and 89% credible interval for each of these distributions sorted by the mean in ascending order.

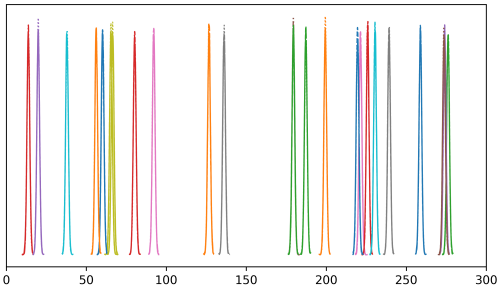


Figure 6.7: Module intercept (α_m) posterior distributions for the single-parameter branch coverage growth rate model.

The *overall* effect of controlling a parameter on the branch coverage growth rate is illustrated in Figure 6.8. What is noticeable is the zero-centered nature of all the distributions, hinting towards no statistically significant overall effect on the branch coverage growth rate. Table F.2, which presents the mean, standard deviation, and 89% credible interval for all parameter effects, echoes the same results.

Instead, considering how specific modules reacted to parameter control, we see that a majority of the posterior distributions are centered around zero, as seen in Figure 6.9, however, some exceptions can be discerned. By studying Table F.3 that offers the mean, standard deviation, and 89% credible interval for the presented interaction posterior distributions, we can identify which interactions are significant.

[3] `https://github.com/henkejson/analysis-adaptive-parameter-control/blob/main/Notebooks/coverage_rate_single_parameter.ipynb`

[4] `https://github.com/henkejson/analysis-adaptive-parameter-control/blob/main/Notebooks/coverage_rate_multi_parameter.ipynb`

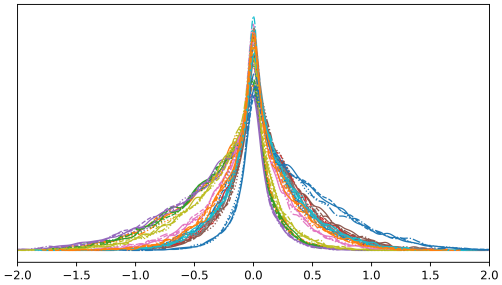


Figure 6.8: Parameter effect (β_p) posterior distributions for the single-parameter branch coverage growth rate model.

The entries in the table are sorted by the mean in ascending order, which entails that the leftmost distribution in Figure 6.9 corresponds with the first element in the table. The identified exceptions are the 10 interactions presented in Table 6.3. The table shows the expected branch coverage growth rate intercepts for the respective modules and the relative effects the specific parameters had on the individual modules' branch coverage growth rate. To calculate the relative effects, we took the particular module's intercept and added the lower limit of the interaction effect's credible interval. The same procedure was repeated for the upper limit of the credible interval as well.

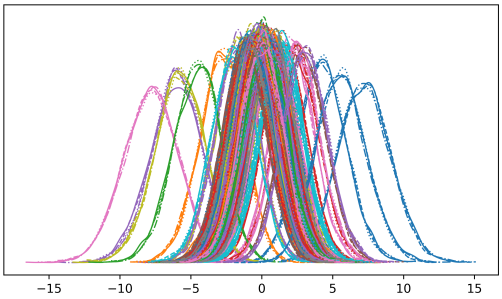


Figure 6.9: Interaction effect (γ_{mp}) posterior distributions for the single-parameter branch coverage growth rate model.

Module and parameter pair	**Module intercept**	**Relative effect** (5.5%)	**Relative effect** (94.5%)
g_mp[return_from_generator x Pop]	60.2	49.4	55.5
g_mp[return_from_generator x ChromLen]	60.2	51.2	57.0
g_mp[return_from_generator x Crossover]	60.2	51.7	57.3
g_mp[return_from_generator x TestDeleteProb]	60.2	53.1	58.4
g_mp[yield_from x ChangeParamProb]	126.7	121.3	126.4
g_mp[packages x TestDeleteProb]	13.8	14.2	19.2
g_mp[return_from_generator x TourSize]	60.2	60.8	65.8
g_mp[return_from_generator x ChangeParamProb]	60.2	61.9	67.2
g_mp[return_from_generator x TestInsertionProb]	60.2	63.0	68.6
g_mp[return_from_generator x RandPert]	60.2	64.5	70.3

Table 6.3: Significant interactions between modules and parameters for the single-parameter branch coverage growth rate model (see Appendix B for full names of modules and parameters).

Based on the results presented in Table 6.3, performing parameter control on Population Size, Chromosome Length, Crossover Rate, or Test Delete Probability has a

negative impact on the branch coverage growth rate achieved when generating tests for the module `return_from_generator`. Additionally, the module `yield_from`, which is from the same project as `return_from_generator`, had a negative response to control of the specific parameter Change Parameter Probability. However, controlling Tournament Size, Change Parameter Probability, Test Insertion Probability, or Random Perturbation results in an overall *positive* impact on the branch coverage growth rate for the module `return_from_generator`. Furthermore, the module `packages` responded positively to control of the parameter Test Delete Probability.

6.2.2 Multi-Parameter Experiment (RQ2.2)

Regarding the multi-parameter experiment, the module intercepts remain largely unchanged in comparison to the results from the single-parameter experiment, as evident in Figure 6.10. Similarly to the single-parameter experiment, the posterior distributions are evenly spread and shaped sharply and narrowly. In Appendix F, Table F.4 showcases the mean, standard deviation, and 89% credible interval for every posterior distribution presented in Figure 6.10. The table is sorted in ascending order of means, resulting in the module with the smallest mean intercept being located on the first row.

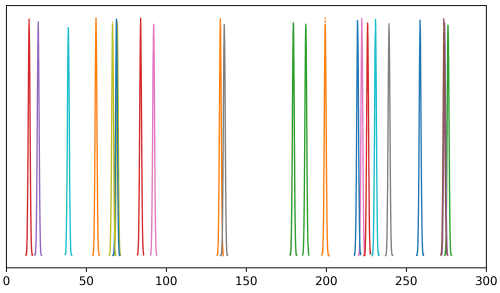


Figure 6.10: Module intercept (α_m) posterior distributions for the multi-parameter branch coverage growth rate model.

Concerning the *overall* parameter effect on the branch coverage growth rate when controlling multiple parameters, there is no statistically significant impact, as visible in Figure 6.11. All posterior distributions are centered around zero and have a narrow shape. Inspecting Table F.5 that shows the mean, standard deviation and 89% credible intervals for each parameter pair, we can see that all effects are non-significant.

Figure 6.12 presents the interaction effects on the branch coverage growth rate, i.e., the impact of performing parameter control on pairs of parameters for each specific module. Similarly to the parameter effect, there are no statistically significant effects, as indicated by all distributions visibly containing zero. Table F.6 shows the mean, standard deviation and 89% credible intervals for all interactions, and reaffirms the same results.

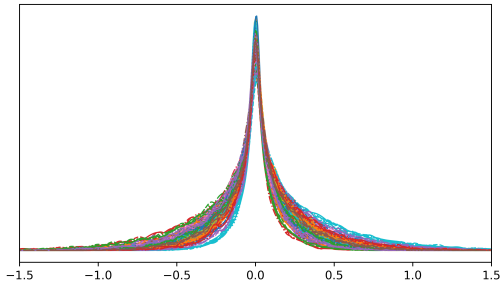


Figure 6.11: Parameter effect (β_p) posterior distributions for the multi-parameter branch coverage growth rate model.

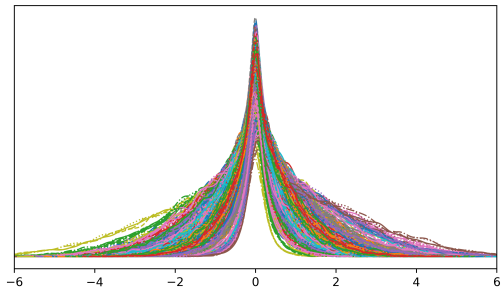


Figure 6.12: Interaction effect (γ_{mp}) posterior distributions for the multi-parameter branch coverage growth rate model.

6.3 Overhead (RQ3)

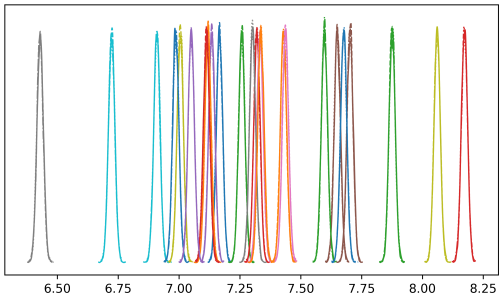
The performance impact of our parameter control system on the test generation process can be measured by recording the number of iterations performed during the 300-second search budget. We deployed two separate Bayesian statistical models, one per experiment. The Bayesian Models were run using 4 chains, taking 6,000 samples each, of which 1,000 were discarded as warm-up. As evident in our Tables located in Appendix G, the models sampled very efficiently with Monte Carlo Standard Error (MCSE) close to, or equal to zero. All $\hat{R}$ values are equal to 1.00, indicating that all chains converged. The multi-parameter experiment, however, did have a lower effective sample size (ESS) compared to the single-parameter experiment. Still, it should prove sufficient, since it still exceeds the recommended minimum amount of 10% of total samples [47]. For a complete breakdown of priors checks, trace plots, summary tables, and posterior distributions, refer to our Jupyter notebooks hosted on GitHub[5,6].

[5]`https://github.com/henkejson/analysis-adaptive-parameter-control/blob/main/Notebooks/overhead_model_single_parameter.ipynb`

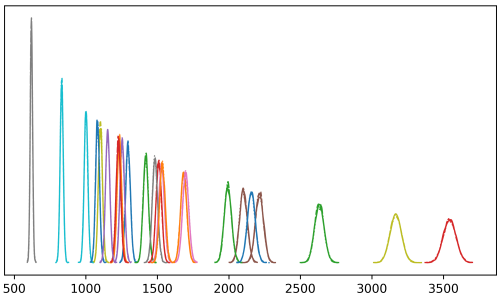
[6]`https://github.com/henkejson/analysis-adaptive-parameter-control/blob/main/Notebooks/overhead_model_multi_parameter.ipynb`

6.3.1 Single-Parameter Experiment (RQ3.1)

Figure 6.13 presents the posterior distributions for the module intercepts, or rather, the expected amount of iterations completed for each of the 24 modules. The Bayesian model represents module intercepts in the natural logarithm scale, as seen in Figure 6.13a, where most modules have an intercept in the range $[6, 8]$. To understand what this translates to in an actual number of iterations, we exponentiate the posterior distributions as visualized in Figure 6.13b. A large majority of the modules complete, on average, 1000 to 2500 iterations, with some extending into the three-thousands. To identify what modules are associated with each distribution, we refer to Table G.1 presented in Appendix G, where the mean, standard deviation, and 89% credible interval for the posterior distributions are documented (in natural logarithm scale). Note that the table is sorted by the mean in ascending order, meaning that the first element in the table corresponds to the left-most distribution in Figure 6.13a.



(a) Natural logarithm scale.



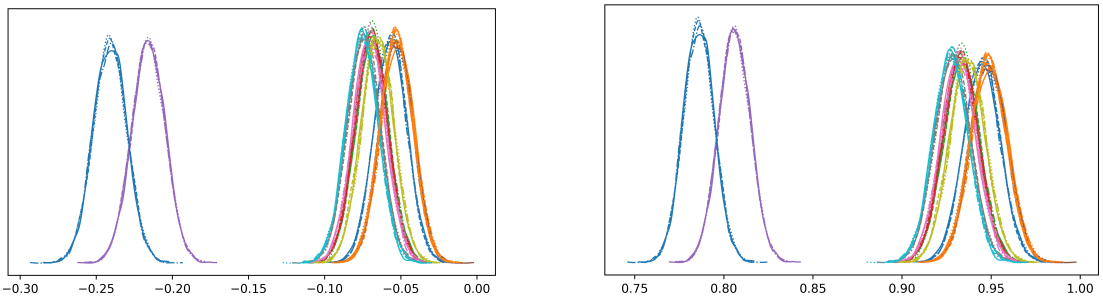
(b) Natural exponential scale (number of iterations).

Figure 6.13: Module intercept (α_m) posterior distributions for the single-parameter overhead model.

To understand what the overall implications are of deploying our parameter control system for individual parameters, we turn to Figure 6.14. These two graphs show the posterior distributions for the parameter effect on the total number of iterations. Figure 6.14a shows the distributions in the natural logarithm scale, while Figure 6.14b shows them in the natural exponential scale.

The exponential scale translates directly to the decrease in the number of iterations (measured as the percentage of the decrease). For example, the two left-most distributions belong to controlling Test Insertion Probability and Population Size respectively, and reduce the number of iterations by, on average, 21.4% ($e^{-0.241} \approx 0.786$) and 19.4% ($e^{-0.216} \approx 0.806$). Meanwhile, the rest of the parameters result in a decrease in the number of iterations, somewhere in the range of four to ten percent. Table G.2 showcases the mean, standard deviation, and 89% credible interval for these posterior distributions sorted in ascending order in terms of mean (in natural logarithm scale). The results show that controlling any parameter has a negative impact on the number of iterations completed during the 300-second search bud-

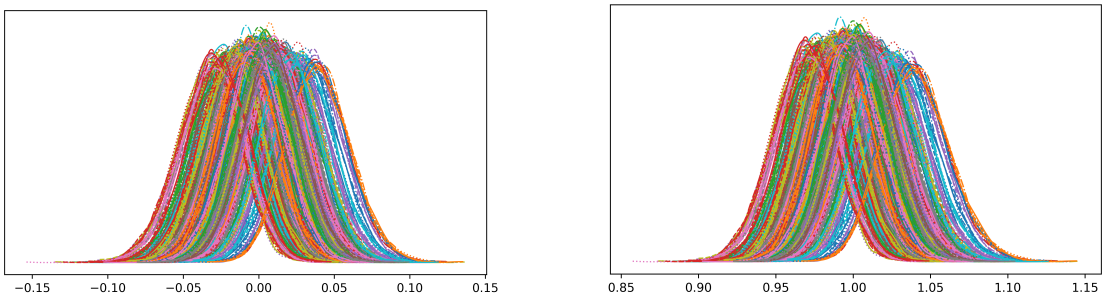
get, with the greatest reduction caused by controlling the parameters Test Insertion Probability and Population Size.



(a) Natural logarithm scale. **(b)** Natural exponential scale.

Figure 6.14: Parameter effect (β_p) posterior distributions for the single-parameter overhead model.

Figure 6.15 shows the individual effects of parameter control for specific modules. There are 288 posterior distributions, each belonging to a specific module and parameter combination represented both in natural logarithm scale (Figure 6.15a) as well as natural exponential scale (Figure 6.15b). What can quickly be deduced is the zero-centered nature of an overwhelming majority of the posterior distributions. However, by investigating Table G.3, which presents the mean, standard deviation, and 89% credible interval, seven out of the 288 effects have a statistically significant impact on the number of iterations. In Table 6.4, we present these seven significant interactions and their effects according to their 89% credible intervals. To calculate these effects, we exponentiate (e^x) the lower and upper bound of the credible intervals.



(a) Natural logarithm scale. **(b)** Natural exponential scale.

Figure 6.15: Interaction effect (γ_{mp}) posterior distributions for the single-parameter overhead model.

For the module `positional_validation`, Test Insertion Probability has an additional negative impact (beyond the overall effect) on the number of test generation iterations. The same is true for the module `namedtupleutils`, except that it responds negatively towards controlling the Elite parameter. There are also some interactions that lead to an increase in the number of iterations completed within

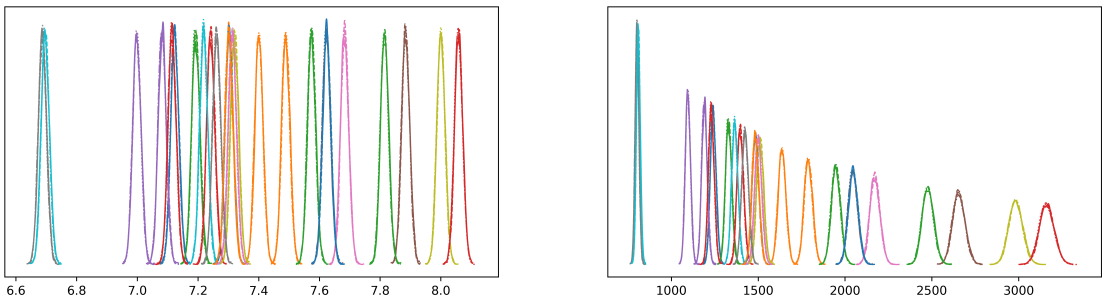
Module and parameter pair	**Effect** (5.5%)	**Effect** (94.5%)
g__mp[positional_validation x TestInsertionProb]	0.9352	0.9990
g__mp[namedtupleutils x Elite]	0.9343	0.9980
g__mp[py_helpers x Crossover]	1.0010	1.0672
g__mp[py_helpers x ChromLen]	1.0040	1.0714
g__mp[py_helpers x TestInsertionProb]	1.0060	1.0747
g__mp[validation x Crossover]	1.0060	1.0736
g__mp[da x TourSize]	1.0050	1.0736

Table 6.4: Significant interactions between modules and parameters for the single-parameter overhead model (see Appendix B for full names of modules and parameters).

the search budget. Controlling the parameters Crossover Rate, Chromosome Length, and Test Insertion Probability has a statistically significant positive effect on the number of iterations completed for the module `py_helpers`. Additionally, Crossover Rate has a positive impact on the module `validation`. Finally, the module `da` is positively receptive to controlling the parameter Tournament Size.

6.3.2 Multi-Parameter Experiment (RQ3.2)

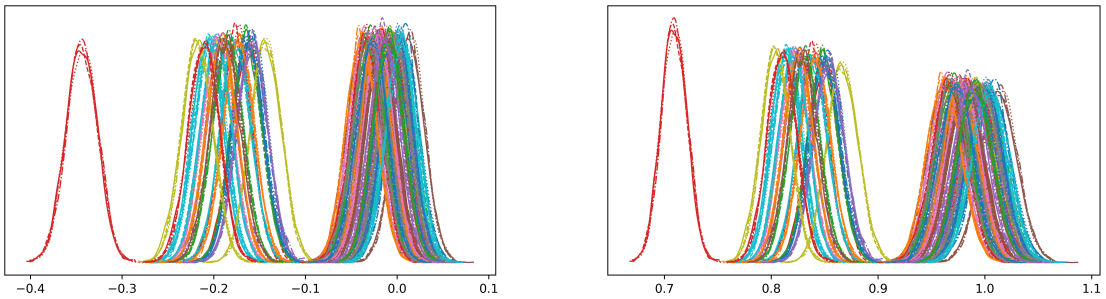
If we consider the module intercepts for the multi-parameter experiment, see Figure 6.16b, we can observe that a majority of the posterior distributions lie between 1000 and 2500 iterations, while some extend further beyond that interval. Comparing these graphs to the module intercepts estimated in the single-parameter experiment (see Figure 6.13), some distributions have shifted their mean. For example, the module `headers` (displayed in brown color) has significantly increased the expected amount of iterations, now located in the top three. However, modules like `positional_validation` (colored in red) have decreased slightly to around 3500 iterations. For the mean, standard deviation, and 89% credible interval for each of these distributions, see Table G.4 in Appendix G. The table uses the natural logarithm scale like Figure 6.16a and is sorted in ascending order based on their mean value.



(a) Natural logarithm scale. **(b)** Natural exponential scale.

Figure 6.16: Module intercept (α_m) posterior distributions for the multi-parameter overhead model.

Looking at the *overall* parameter effect on the total number of iterations we can identify three clusters, see Figure 6.17. The first cluster (see either Figure 6.17a or Figure 6.17b) is located furthest to the left and only includes a single posterior distribution belonging to the parameter combination Population Size and Test Insertion Probability. In the single-parameter experiment, these two parameters had the most negative effect on the total number of iterations (see Figure 6.14). When controlled together, they create an even more negative impact on the number of iterations, with an average of 29% decrease in the amount of iterations. The second cluster, located towards the middle, contains all combinations of either Population Size or Test Insertion Probability, meaning that these two parameter will always have a negative effect on the number of iterations no matter their pairing.



(a) Natural logarithm scale. **(b)** Natural exponential scale.

Figure 6.17: Parameter effect (β_p) posterior distributions for the multi-parameter overhead model.

The last cluster consists of the remaining combinations of parameters, i.e., those that do not include either Population Size or Test Insertion Probability. Some of these have a statistically significant negative impact according to their 89% credible interval (collected in Table 6.5), while most distributions' credible intervals overlap zero, resulting in no significant impact being observable. The effects were calculated by exponentiating the lower and upper 89% credible intervals for these distributions. For the mean, standard deviation, and 89% credible interval for all the posterior distributions see Table G.5.

To understand how the interactions affect the number of iterations, i.e., how specific modules respond to different pairing of parameters, see Figure 6.18. All posterior distributions assume a similar shape, all centered around zero, meaning that no statistically significant effect can be determined. To verify this, see Table G.6 which contains the mean, standard deviation and 89% credible intervals for the presented posterior distributions.

Parameter combinations	**Effect** (5.5%)	**Effect** (94.5%)
b_p[Crossover|TestChangeProb]	0.938	0.989
b_p[StatemInsertProb|TestChangeProb]	0.938	0.988
b_p[Crossover|TestDeleteProb]	0.941	0.991
b_p[StatemInsertProb|TestInsertProb]	0.943	0.994
b_p[TestChangeProb|TestInsertProb]	0.944	0.994
b_p[RandPert|TestDeleteProb]	0.944	0.995
b_p[StatemInsertProb|TestDeleteProb]	0.945	0.996
b_p[Elite|StatemInsertProb]	0.947	0.998
b_p[TestDeleteProb|TestInsertProb]	0.948	0.998

Table 6.5: Parameter pairs that result in statistically significant results within the third cluster (i.e., those not including either `Population Size` or `Test Insertion Probability`) for the multi-parameter overhead model (see Appendix B for full names of modules and parameters).

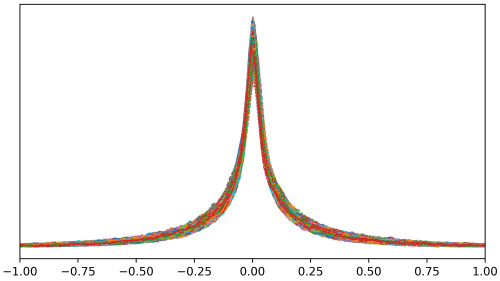


Figure 6.18: Interaction effect (γ_{mp}) posterior distributions for the multi-parameter overhead model.

7

Discussion

This chapter provides discussions and summaries for the research questions based on the results from the experiments and highlights some threats to the validity of the study and how they were addressed. Additionally, ideas for future work based on this study are presented.

7.1 Final Branch Coverage (RQ1)

Research question 1 (RQ1) tried to answer if deploying our parameter control system yielded any significant change in the final branch coverage achieved by the generated test suite. In general, no significant changes were observed for the single- or multi-parameter experiment. However, two modules showed some receptiveness to parameter control, albeit to differing extents. The first of these modules is `return_from_generator`, which is responsible for all statistically significant results in the single-parameter experiment and all but one significant results in the multi-parameter experiment. From the single-parameter experiment, control of eight out of the twelve parameters available resulted in a significant result for this module. Half of the parameters lead to a decrease in final branch coverage, while the other half lead to an increase. Meanwhile, in the multi-parameter experiment, five parameter combinations showed a negative impact on final branch coverage, while ten provided a positive effect. The second module showcasing significant results is `positional_validation`, which resulted in a slight increase in final branch coverage when the parameter pair of Chromosome Length and Population Size was controlled.

To investigate why these modules showed increased receptiveness to parameter control in comparison to all other modules, we looked at some static code analysis metrics, see Table C.1 (in Appendix C). What can be noted from these metrics is that neither module exhibits any striking differences from the modules that were unreceptive to parameter control. In addition to the lack of unique metric values, the module `return_from_generator` is from a project that is represented by multiple different modules, none of which result in any significant results when considering the final branch coverage. Therefore, it is unclear why these modules responded differently. Future experiments should increase the range of modules considered, to

clarify what factors affect the performance of parameter control.

Final Branch Coverage (RQ1): Controlling individual or pairs of parameters does not have an overall statistically significant effect on the final branch coverage. However, certain modules are more receptive to parameter control for specific parameters.

7.2 Branch Coverage Growth Rate (RQ2)

Research question 2 (RQ2) attempted to measure the degree to which the application of parameter control changes the branch coverage growth rate during the test generation process. The overall trend indicates that there is no significant change in the branch coverage growth rate when performing parameter control on either individual or pairs of parameters. However, the single-parameter experiment had some notable exceptions to this trend.

From the single-parameter experiment, we observed that the modules `return_from_generator`, `yield_from`, and `packages` were more responsive to parameter control of certain individual parameters. The modules `yield_from` and `packages` showed one significant result each, one leading to an increase in branch coverage growth rate and the other to a decrease, respectively. The `return_from_generator` module, in a similar manner to the results from RQ1, showed more receptiveness. For four of the available parameters, the module showed a negative association towards the branch coverage growth rate, while simultaneously showing a positive association towards four other parameters.

The responsiveness of the `return_from_generator` module is not surprising as it did show increased receptiveness to parameter control in terms of final branch coverage (RQ1). However, interestingly, the parameters for which it showed responsiveness are not entirely the same across both experiments. For example, Crossover Rate had a negative impact on the branch coverage growth rate but no significant impact on final branch coverage. While Test Insert Probability did have a positive effect on final branch coverage but did not have any significant impact on the branch coverage growth rate. Still, most parameters remained the same.

No such overlap exists between the experiments for the remaining receptive modules, `yield_from` and `packages`. These only showed some significant impact on the branch coverage growth rate for control of one parameter each, Change Parameter Probability, and Test Delete Probability, respectively. Considering the code metrics for these three modules, see Table C.1, there is only one notable value, namely, the average cyclomatic complexity (CC) per function for the `packages` module. This metric is considerably higher than all other modules at 7.8, indicating that on average this module's functions are more complex. However, neither `return_from_generator` nor `yield_from` have as extreme average CC values as `packages`, but they are still towards the higher end of the spectrum. This might

indicate that there exists some kind of relationship between a module's average CC and its receptiveness to parameter control, in regards to branch coverage growth rate. Still, many modules with similar average CC values showed no significant effect of parameter control on the branch coverage growth rate. A further confusion factor is the fact that the same module can respond in both a positive and negative manner, like `return_from_generator` did. Thus, we cannot draw any clear conclusion about the existence of any correlation between any of these code metrics and the receptiveness to parameter control for branch coverage growth rate.

Branch Coverage Growth Rate (RQ2): Parameter control for individual or pairs of parameters did not show any overall significant effect on branch coverage growth rate. However, a few modules were more receptive to control of some specific individual parameters, which resulted in slight changes to the results when compared to the baseline of no parameter control.

7.3 Overhead (RQ3)

Research question 3 (RQ3) investigated the performance overhead of performing parameter control by considering the number of iterations (corresponding to the number of generations) completed within the search budget. The addition of parameter control increases the workload, thus it is no surprise that the overall trend points towards a decrease in the number of iterations regardless if we control individual or pairs of parameters. Two parameters resulted in a particularly negative effect, Population Size and Test Insertion Probability. These parameters resulted in negative effects both when controlled individually and in combination with any other parameters. Their effects might be explained by the impact that they have on the test generation process. Both directly affect the number of test cases handled (created, mutated, evaluated, and ranked). Population Size states how many test cases the test generation algorithm works with each iteration, while Test Insertion Probability decides how many new test cases should be added to the population (according to $Population\ Size \times Test\ Insertion\ Probability$). In every iteration, the test cases are mutated, ranked, and sorted to identify which should be kept for the following generations. Thus, these parameters have a direct impact on the amount of work required for each iteration, which understandably affects how many iterations can be completed within the search budget. A potential way to limit the impact of these could be to further limit the allowed parameter values. However, making the range too restrictive, could run the risk of making it impossible to find the theoretical optimal parameter value.

Interestingly, when controlling individual parameters, there were a few module and parameter combinations that resulted in an increase in the number of iterations completed within the search budget. A possible theory as to why, is that the parameter values selected by the RL Agent could result in a reduction of the amount of work that the test generation algorithm needs to perform. For example, lowering the value for Tournament Size should result in fewer comparisons needed to per-

form the selection of individuals to become parents for future generations. Another reason might be that certain modules are not equally impacted by the overall parameter effect, thus some modules might compensate for this discrepancy using the interaction term.

Overhead (RQ3): The performance impact of the parameter control system is generally negative regardless of whether individual or pairs of parameters are controlled. The parameters Population Size and Test Insertion Probability had the largest negative impact on performance.

7.4 Parameter Assignment Exploration

One potential reason why the impact of parameter control was limited, might be that the space of possible parameter values was not properly explored by the reinforcement learning (RL) agent, or if the RL agent was focused on a limited set of ineffective parameter values. To investigate this possibility, we utilize the timeline of parameter values selected by the RL agent during the experiments (as described in Section 5.4.2). We grouped the data according to the parameter controlled, extracted the chosen parameter values, and plotted them as histograms as seen in Appendix H. Note that these are the aggregated results only for the single-parameter experiment, to observe the isolated behavior of the RL agent per parameter. The Jupyter notebook used to generate these plots are available on GitHub[1].

By examining the different plots, we observe some similarities between them. One recurring noteworthy detail is the high frequency of values close to or equal to the bounds of each parameter interval. We theorize that this is the RL agent finding and attempting to cross the boundary of allowed parameter values. Most likely, the RL agent gets stuck there for some iterations, until it makes a large step in the other direction, as can be seen by the slight bump approximately one delta size away from either bound (according to Table 4.2). These aforementioned features are present in every histogram except for Elite (Figure H.1d) and Tournament Size (Figure H.3b), as these only have a narrow discrete set of possible values to choose from.

Another striking feature of these histograms is the relative uniformity, with a slight decline toward values further away from the default values. For example, observe the histogram for Change Parameter Probability (see Figure H.1a) that has a default value closer to the lower bound, resulting in a decline in frequency of values closer to the upper bound. In contrast, consider the histogram for Statement Insertion Probability (see Figure H.2c) where the default value is exactly in the middle of the interval, and the resulting symmetry around this point.

From these results, we can conclude that the parameter spaces are explored in a reasonably even manner, given the uniform trend of all parameter values with some

[1] `https://github.com/henkejson/analysis-adaptive-parameter-control/blob/parameter_assignment/Notebooks/parameter_assignment_analysis.ipynb`

slight added frequency around the default values. While the RL agent seems to spend a considerable amount of time at the bounds, as it learns the nature of the parameter spaces, it is not stuck there indefinitely. The time spent at these edges may have prevented the RL agent from adequately estimating the impact of—and subsequently exploiting—certain specific values within each range. However, we can see that all possible values have been attempted many times across the experiments.

This leads us to hypothesize that the reason for the limited effectiveness of parameter control, might not lie in the exploration of parameter values. Rather, this hints towards the DynaMOSA algorithm not being particularly sensitive to any changes made to the parameter values. However, in future work, the core RL algorithm should be adjusted to spend less time at the bounds of the range.

7.5 Threats to Validity

This section presents the identified threats to the validity of this study. We will divide the threats into four categories: construct validity, conclusion validity, internal validity, and external validity [51].

7.5.1 Construct Validity

Construct validity is mainly concerned with whether what we are measuring is the correct thing to answer our research questions. For RQ1 we are explicitly asking about the specific metric we are measuring, the final branch coverage achieved by the generated test suite. Thus, there is little left to interpretation. However, for RQ2 and RQ3, we are asking about more abstract and multi-faceted metrics, branch coverage growth rate, and performance overhead. We chose to model the branch coverage growth rate (RQ2) as the integration of the coverage timeline. Another way this could have been measured is as an average coverage change[2]. However, using this strategy would cause us to lose more information since we only consider the start- and end-points. Basing the branch coverage growth rate on the integration of the coverage timeline retains information from the whole interval.

Similarly, for the measurement of the added overhead when performing parameter control, many different metrics could have been utilized, e.g., CPU usage, RAM usage, or instructions executed per second. However, we measured the number of iterations completed within the search budget. By measuring the number of iterations, we get a concrete value that directly represents the impact the addition of parameter control has on how many generations the algorithms manage to complete during the set search budget.

An additional aspect of construct validity for this study is the use of artificial intelligence (AI) tools e.g., ChatGPT. Due to the inherent risks of AI tools, such as hallucination, where the AI fabricates incorrect information, we used these tools

[2]Average coverage change: $\frac{final\ coverage - initial\ coverage}{total\ time}$.

with great caution. One way we used ChatGPT was to have it explain concepts or to bounce ideas off of it. All information received from ChatGPT was confirmed using external sources, i.e., books and papers to mitigate any risk of misinformation. Another way we used ChatGPT was to generate boilerplate code e.g., different ways to visualize our data and results, that we then adapted and modified to our needs. In addition to any changes we made to the generated code, we also thoroughly examined it to ensure that it was correct and properly fulfilled its objective. ChatGPT was not deployed to replace our work, rather it was used to free up time from smaller and monotonous tasks so that we could spend more time on larger and more important work. ChatGPT has not been used as part of writing this thesis.

7.5.2 Conclusion Validity

Parameters are a fundamental part of the test generation process and are explicitly linked to the decisions and formation of the search process. For example, having a smaller Population Size limits the test generation algorithm's potential to pursue multiple goals simultaneously. Similarly, all other parameters play a specific role in shaping the test generation procedure. Therefore, there exists a relationship between the treatment (parameter values) and the outcome (generated test cases). However, the significance of this relationship is unknown.

Upon further inspection of how the selected parameters are used in PynguinAPC, we identified that the Elite parameter is only used by the WholeSuite algorithm, even though it was listed in the general search algorithm configuration. Since this parameter is not used in DynaMOSA, and thus, not used in DynaMOSA_RL either, controlling it would not have an impact on the test generation process, other than the overhead of the RL itself. This discovery was made after both experiments and analyses were completed, and thus, too little time was left to redo these steps. However, because of the way our Bayesian models were set up, the impact of including this parameter on the results should be negligible at most. By utilizing varying effects models, where we allow each parameter to estimate its individual overall effect as well as its module-specific effect, the result of controlling Elite should have no, or minimal impact on all other estimates.

7.5.3 Internal Validity

The test generation process is inherently random, thus we can never be completely sure that our results stem solely from our parameter control system. Additionally, reinforcement learning also has a large degree of uncertainty and the choice of algorithm will affect how it behaves, e.g., how it explores or exploits the space of possible states. To combat the issue of randomness and uncertainty we ran multiple repetitions of each configuration, 30 for the single-parameter experiment and 10 for the multi-parameter experiment. If a majority of the repetitions exhibit the same or similar behavior, we can be more certain that what we are measuring are the results of our treatment rather than the inherent randomness.

To avoid additional factors of uncertainty, most factors that could be controlled have been. For example, all experiments were run on identical hardware in isolated Docker containers and used the same RL algorithm, update interval, plateau length, search budget, etc. Similarly to how repetitions were used, this helps to ensure that what we measure is the effect of PynguinAPC.

7.5.4 External Validity

In our experiments, we used 24 randomly selected modules out of a larger subset of 58. These modules come from a wide variety of projects in different domains, developed by different organizations, following different development practices. Thus, we have a sufficient set of modules that are representative of many common and distinct code styles. However, more modules are always desirable to be able to draw more generalized conclusions on the effect of parameter control.

Additionally, since this study follows an experimental simulation design with a controlled setting, we trade generalizability for measurement accuracy. Defining our setting necessitated us to make selections to control the variability, all of which limits the generalizability of our results. The selections we had to make regarding our setting included the test generation framework (Pynguin) and the test generation algorithm (DynaMOSA), which do not necessarily translate to other frameworks (e.g., EvoSuite) or algorithms (e.g., WholeSuite or MIO). All choices in regards to the reinforcement learning part of PynguinAPC also bring consequences to our study's generalizability, e.g., the chosen RL algorithm, how we chose to define our action and observation spaces, and our reward function. Had we used a different RL algorithm, we could have observed other results. However, without any possibility of knowing the optimal choice of RL algorithm *a priori* we had to simply choose the one that performed the best according to other studies, without knowing whether the same results would apply to ours. When conducting our experiments we also had to make certain limitations, such as the selection and pairing of parameters, update interval, and plateau length. All these decisions will, to some extent, have an impact on the results.

7.6 Future Work

During our work, we have identified some possible areas of future work:

Algorithm tuning Due to the limited impact of parameter control and the parameter assignments measured during the experiments, further optimizing the reinforcement learning algorithm could provide new avenues for improvements. For example, reducing the time spent at the bounds of the parameter value spaces might lead to a more effective and even exploration of the possible parameter values. Another possibility could be to replace the algorithm or create a purpose-built one.

More modules It would be beneficial to see a larger set of modules with varying

lengths and complexity to more accurately determine the effect of parameter control. Preferably, we would like to see an even wider array of domains, organizations, and development practices represented that are not part of the original Pynguin evaluation modules.

Larger parameter combinations In this study we only explored the impact of performing parameter control on individual parameters and pairs of parameters. It would be interesting to explore whether performing parameter control on larger sets of parameters (e.g., groups of 5 parameters or all 12) at a time would result in a different outcome.

Understanding module response From our experiment results we noticed that a few modules were more receptive to parameter control than the rest. In this study, we performed some initial exploration into module receptiveness, by conducting static code analysis for the modules. We were not able to identify any clear patterns that could explain our results. Thus, future work could further investigate possible reasons as to why, by e.g., looking at additional metrics, performing other types of analyses, or considering other module characteristics.

Parameter tuning Another strategy to optimize parameter values is parameter tuning, where an attempt is made to find optimal parameter values before execution. It may be of interest to explore whether parameter tuning would produce different results when applied to search-based test generation than parameter control. This could entail comparing parameter tuning to the parameter control system we developed.

Bug detection rate While we did not observe any major differences in final coverage when applying parameter control. It would be interesting to examine whether parameter control can have an effect on the bug detection rate for the generated test suite, e.g., by using the BugsInPy dataset [52].

8

Conclusion

To help developers with the unit test creation process, there exist frameworks that automate this process. One such framework for Python code is called Pynguin, which accepts a Python module, deploys one out of several test generation algorithms, and outputs a test suite. However, there is no guarantee that the generated test suites will reach anywhere near 100% branch coverage.

One possible way to improve the final branch coverage is to pick parameter values for the test generation algorithm that are more suited for the module under test. To avoid having to choose parameter values manually, this process could be automated, e.g., by using parameter control which entails updating parameter values during the test generation process. Our goal was to see if any improvements could be obtained by using parameter control, which we evaluated using the final branch coverage, branch coverage growth rate, and performance overhead.

We developed PynguinAPC, an extension of Pynguin, that includes an RL-powered parameter control system. The idea was that, at regular intervals, the test generation algorithm requests new parameter changes from an RL agent. The RL agent responds with deltas, i.e., how much each controlled parameter should change, which are then applied to the controlled parameter(s). The test generation runs for a couple of iterations, and then the next time it requests new parameter values, the RL agent is rewarded with the change in the final branch coverage from the last update. The process repeats until either the generated test suite achieves 100% branch coverage or the search budget is exhausted.

From the evaluation of our parameter control system for pairs of, or individual, parameters, we can conclude that our system does not produce any overall significant change in final branch coverage. The same is true for our evaluation of the branch coverage growth rate, where no overall change was observed. However, certain module-parameter combinations were receptive to parameter control, both positively and negatively, for final branch coverage and branch coverage growth rate. Unsurprisingly, the addition of parameter control had a general negative impact on the test generation performance.

Taking all results into account, our recommendation for parameter control in searched-

based test generation is to avoid controlling Population Size or Test Insertion Probability since these caused a substantial increase in overhead. With that being said, rather than giving up on those parameters completely, more work should be done to explore the impact of the parameter control configuration, e.g., the chosen RL algorithm, the allowed parameter value intervals, the range of allowed delta values for all parameters, action strategy (delta values vs. direct parameter overwriting). All these configuration choices might have had a substantial impact on the results we measured, and thus, there is likely room for improvement by modifying these as well. Additionally, some modules were more receptive to parameter control of certain parameters for both final branch coverage and branch coverage growth rate (positively and negatively) which merits future investigation into why these specific modules exhibited an increased responsiveness while others were largely unaffected.

Bibliography

[1] H. Almulla and G. Gay, "Learning how to search: Generating effective test cases through adaptive fitness function selection," *Empirical Software Engineering*, vol. 27, no. 2, p. 38, Jan. 2022, ISSN: 1573-7616. DOI: 10.1007/s10664-021-10048-8.

[2] S. Anand, E. K. Burke, T. Y. Chen, *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2013.02.061.

[3] V. H. S. Durelli, R. S. Durelli, S. S. Borges, *et al.*, "Machine learning applied to software testing: A systematic mapping study," *IEEE Transactions on Reliability*, vol. 68, no. 3, pp. 1189–1212, 2019. DOI: 10.1109/TR.2019.2892517.

[4] A. Fontes and G. Gay, "The integration of machine learning into automated test generation: A systematic mapping study," 2023. arXiv: 2206.10210 [cs.SE].

[5] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013. DOI: 10.1109/TSE.2012.14.

[6] M. Esnaashari and A. H. Damia, "Automation of software test data generation using genetic algorithm and reinforcement learning," *Expert Systems with Applications*, vol. 183, p. 115 446, 2021, ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2021.115446.

[7] H. Almulla and G. Gay, "Learning how to search: Generating exception-triggering tests through adaptive fitness function selection," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 63–73. DOI: 10.1109/ICST46399.2020.00017.

[8] M. Brunetto, G. Denaro, L. Mariani, and M. Pezzè, "On introducing automatic test case generation in practice: A success story and lessons learned," *arXiv*, vol. 2103.00465, 2021, Accessed: 2023-12-12. [Online]. Available: https://arxiv.org/abs/2103.00465.

[9] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 168–172. DOI: 10.1145/3510454.3516829.

[10] EvoSuite. "Evosuite | automatic test suite generation for java." (2021), [Online]. Available: https://www.evosuite.org/ (visited on 2023-12-07).

[11] Randoop. "Randoop: Automatic unit test generation for java." (2023), [Online]. Available: https://randoop.github.io/randoop/ (visited on 2023-12-12).

[12] G. Karafotias, A. E. Eiben, and M. Hoogendoorn, "Generic parameter control with reinforcement learning," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '14, Vancouver, BC, Canada: Association for Computing Machinery, 2014, pp. 1319–1326, ISBN: 9781450326629. DOI: 10.1145/2576768.2598360.

[13] A. E. Eiben, M. Horvath, W. Kowalczyk, and M. C. Schut, "Reinforcement learning for online control of evolutionary algorithms," in *Engineering Self-Organising Systems*, S. A. Brueckner, S. Hassas, M. Jelasity, and D. Yamins, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 151–160, ISBN: 978-3-540-69868-5.

[14] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990. DOI: 10.1109/32.57624.

[15] A. Fontes and G. Gay, "Using machine learning to generate test oracles: A systematic literature review," *CoRR*, vol. abs/2107.00906, 2021. [Online]. Available: https://arxiv.org/abs/2107.00906.

[16] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. MIT Press, 2018.

[17] A. Fontes, G. Gay, F. G. de Oliveira Neto, and R. Feldt, "Automated support for unit test generation," in *Optimising the Software Development Process with Artificial Intelligence*, J. R. Romero, I. Medina-Bulo, and F. Chicano, Eds. Singapore: Springer Nature Singapore, 2023, pp. 179–219, ISBN: 978-981-19-9948-2. DOI: 10.1007/978-981-19-9948-2_7.

[18] J. E. Pettinger and R. M. Everson, "Controlling genetic algorithms with reinforcement learning," in *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO'02, New York City, New York: Morgan Kaufmann Publishers Inc., 2002, p. 692, ISBN: 1558608788.

[19] G. J. Myers, T. Badgett, and C. Sandler, *The art of software testing.* Wiley, 2012, ISBN: 1118133137.

[20] H. Hemmati, "How effective are code coverage criteria?" In *2015 IEEE International Conference on Software Quality, Reliability and Security*, 2015, pp. 151–156. DOI: 10.1109/QRS.2015.30.

[21] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 122–158, 2018. DOI: 10.1109/TSE.2017.2663435.

[22] A. Arcuri, "Many independent objective (mio) algorithm for test suite generation," in *Search Based Software Engineering*, T. Menzies and J. Petke, Eds., Cham: Springer International Publishing, 2017, pp. 3–17, ISBN: 978-3-319-66299-2.

[23] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *2015 IEEE 8th International*

Conference on Software Testing, Verification and Validation (ICST), 2015, pp. 1–10. DOI: 10.1109/ICST.2015.7102604.

[24] A. Arcuri, "It really does matter how you normalize the branch distance in search-based software testing," *Software Testing, Verification and Reliability*, vol. 23, no. 2, pp. 119–147, 2013. DOI: https://doi.org/10.1002/stvr.457.

[25] S. Lukasczyk, F. Kroiß, and G. Fraser, "An empirical study of automated unit test generation for python," *Empirical Software Engineering*, vol. 28, no. 36, 2023. DOI: 10.1007/s10664-022-10248-w.

[26] M. Mitchell, *An Introduction to Genetic Algorithms*. The MIT Press, Mar. 1998, ISBN: 9780262280013. DOI: 10.7551/mitpress/3927.001.0001.

[27] K. Nussenbaum and C. A. Hartley, "Reinforcement learning across development: What insights can we draw from a decade of research?" *Developmental Cognitive Neuroscience*, vol. 40, p. 100 733, 2019, ISSN: 1878-9293. DOI: https://doi.org/10.1016/j.dcn.2019.100733.

[28] C. Insel, M. Charifson, and L. H. Somerville, "Neurodevelopmental shifts in learned value transfer on cognitive control during adolescence," *Developmental Cognitive Neuroscience*, vol. 40, p. 100 730, 2019, ISSN: 1878-9293. DOI: https://doi.org/10.1016/j.dcn.2019.100730.

[29] M. Sugiyama, *Statistical Reinforcement Learning: Modern Machine Learning Approaches*, 1st ed. New York: Chapman and Hall/CRC, 2015, p. 206, ISBN: 9780429105364.

[30] C. Szepesvári, *Algorithms for Reinforcement Learning* (Synthesis Lectures on Artificial Intelligence and Machine Learning), 1st ed. Springer Cham, 2010, ISBN: 978-3-031-00423-0. DOI: 10.1007/978-3-031-01551-9.

[31] R. Zhao and S. Lv, "Neural-network based test cases generation using genetic algorithm," in *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, 2007, pp. 97–100. DOI: 10.1109/PRDC.2007.63.

[32] F. Yazdani Banafshe Daragh and S. Malek, "Deep gui: Black-box gui input generation with deep learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 905–916. DOI: 10.1109/ASE51524.2021.9678778.

[33] M. Buzdalov and A. Buzdalova, "Adaptive selection of helper-objectives for test case generation.," *2013 IEEE Congress on Evolutionary Computation, Evolutionary Computation (CEC), 2013 IEEE Congress on*, pp. 2245–2250, 2013, ISSN: 978-1-4799-0452-5.

[34] A. Sharma, V. Melnikov, E. Hüllermeier, and H. Wehrheim, "Property-driven testing of black-box functions," in *2022 IEEE/ACM 10th International Conference on Formal Methods in Software Engineering (FormaliSE)*, 2022, pp. 113–123. DOI: 10.1145/3524482.3527657.

[35] M. G. P. de Lacerda, F. B. de Lima Neto, T. B. Ludermir, and H. Kuchen, "Out-of-the-box parameter control for evolutionary and swarm-based algorithms with distributed reinforcement learning," *Swarm Intelligence*, vol. 17, pp. 173–217, 2023, ISSN: 1935-3820. DOI: 10.1007/s11721-022-00222-z.

[36] Y. Sakurai, K. Takada, T. Kawabe, and S. Tsuruta, "A method to control parameters of evolutionary algorithms by using reinforcement learning," in

2010 Sixth International Conference on Signal-Image Technology and Internet Based Systems, 2010, pp. 74–79. DOI: 10.1109/SITIS.2010.22.

[37] S. Müller, N. Schraudolph, and P. Koumoutsakos, "Step size adaptation in evolution strategies using reinforcement learning," in *Proceedings of the 2002 Congress on Evolutionary Computation. CEC'02 (Cat. No.02TH8600)*, vol. 1, 2002, 151–156 vol.1. DOI: 10.1109/CEC.2002.1006225.

[38] J. Quevedo, M. Abdelatti, F. Imani, and M. Sodhi, "Using reinforcement learning for tuning genetic algorithms," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '21, Lille, France: Association for Computing Machinery, 2021, pp. 1503–1507, ISBN: 9781450383516. DOI: 10.1145/3449726.3463203.

[39] R. Chen, B. Yang, S. Li, and S. Wang, "A self-learning genetic algorithm based on reinforcement learning for flexible job-shop scheduling problem," *Computers & Industrial Engineering*, vol. 149, p. 106 778, 2020, ISSN: 0360-8352. DOI: 10.1016/j.cie.2020.106778.

[40] Stable Baselines3. "Tips and tricks when creating a custom environment." (2024), [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/guide/rl_tips.html#tips-and-tricks-when-creating-a-custom-environment (visited on 2024-03-24).

[41] Farama Foundation. "Gymnasium documentation." (2023), [Online]. Available: https://gymnasium.farama.org/ (visited on 2023-03-24).

[42] Stable Baselines3. "Stable-baselines3 docs - reliable reinforcement learning implementations." (2024), [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/ (visited on 2024-03-24).

[43] Stable Baselines3. "Rl algorithms." (2024), [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/guide/algos.html (visited on 2024-03-24).

[44] T. Haarnoja, A. Zhou, K. Hartikainen, *et al.*, *Soft actor-critic algorithms and applications*, 2019. arXiv: 1812.05905 [cs.LG].

[45] K.-J. Stol and B. Fitzgerald, "The abc of software engineering research," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, Sep. 2018, ISSN: 1049-331X. DOI: 10.1145/3241743.

[46] Docker Inc. "Docker: Accelerated container application development." (2024), [Online]. Available: https://www.docker.com/ (visited on 2024-04-01).

[47] R. McElreath, *Statistical Rethinking: A Bayesian Course with Examples in R and Stan*, 2nd ed. Boca Raton, FL: CRC Press LLC, 2019.

[48] C. A. Furia, R. Torkar, and R. Feldt, "Applying bayesian analysis guidelines to empirical software engineering data: The case of programming languages and code quality," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, Mar. 2022, ISSN: 1049-331X. DOI: 10.1145/3490953.

[49] P. Pacheco, *Introduction to Parallel Programming*. Elsevier, 2011, ISBN: 978-0-12-374260-5.

[50] J. Kruschke, *Doing Bayesian Data Analysis : A Tutorial with R, JAGS, and Stan*, 2nd ed. San Diego, CA: Elsevier Science & Technology, 2015, ISBN: 978-0-12-405888-0.

[51] R. Feldt and A. Magazinius, "Validity threats in empirical software engineering research - an initial survey.," Jan. 2010, pp. 374–379.

[52] R. Widyasari, S. Q. Sim, C. Lok, *et al.*, "Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1556–1560, ISBN: 9781450370431. DOI: 10.1145/3368089.3417943.

[53] SonarSource. "Code Quality, Security & Static Analysis Tool with SonarQube | Sonar." (2024), [Online]. Available: https://www.sonarsource.com/products/sonarqube/ (visited on 2024-06-17).

[54] rubik. "Radon." (2024), [Online]. Available: https://github.com/rubik/radon (visited on 2024-06-17).

A

Modules

Table A.1: All resulting 58 modules after identification and filtering, with the associated commit hash when they were accessed.

	Module Name	Commit Hash
1	codetiming._timer	e51e4c3
2	flake8.formatting.base	5c52d75
3	flake8.formatting.default	
4	flake8.main.debug	
5	flutils.decorators	df0f84e1
6	flutils.namedtupleutils	
7	flutils.packages	
8	flutils.setuputils.cmd	
9	httpie.cli.dicts	9e8e369
10	httpie.output.formatters.headers	
11	httpie.output.processing	
12	httpie.output.streams	
13	httpie.plugins.base	
14	httpie.sessions	
15	httpie.ssl_	
16	isort.exceptions	7de1829
17	isort.utils	
18	mimesis.builtins.da	c87af6d
19	mimesis.builtins.pt_br	
20	py_backwards.conf	fd2d89a
21	py_backwards.files	
22	py_backwards.transformers.base	
23	py_backwards.transformers.dict_unpacking	
24	py_backwards.transformers.metaclass	
25	py_backwards.transformers.python2_future	
26	py_backwards.transformers.return_from_generator	
27	py_backwards.transformers.starred_unpacking	
28	py_backwards.transformers.string_types	

	Module Name	Commit Hash
29	py_backwards.transformers.variables_annotations	fd2d89a
30	py_backwards.transformers.yield_from	
31	py_backwards.utils.helpers	
32	py_backwards.utils.snippet	
33	pymonet.immutable_list	94610ee
34	pymonet.lazy	
35	pymonet.maybe	
36	pymonet.monad_try	
37	pymonet.semigroups	
38	pymonet.task	
39	pymonet.validation	
40	pypara.accounting.journaling	bf39934
41	pypara.commons.errors	
42	pytutils.excs	d7a37c0
43	pytutils.lazy.lazy_import	
44	pytutils.props	
45	pytutils.python	
46	sanic.config	acb29c9
47	sanic.headers	
48	sanic.helpers	
49	sanic.mixins.listeners	
50	sanic.mixins.middleware	
51	sanic.mixins.routes	
52	sanic.mixins.signals	
53	sanic.models.protocol_types	
54	sanic.views	
55	string_utils.manipulation	78929d8
56	thonny.plugins.pgzero_frontend	5efec4a
57	thonny.roughparse	
58	typesystem.tokenize.positional_validation	7decd48

B

Module and Parameter Name Conversions

Table B.1: Shortened names for the 24 modules used in the experiments.

Original name	Shortened name
codetiming._timer	timer
flutils.decorators	decorators
flutils.namedtupleutils	namedtupleutils
flutils.packages	packages
flutils.setuputils.cmd	cmd
httpie.output.formatters.headers	headers
httpie.plugins.base	h_base
mimesis.builtins.da	da
py_backwards.transformers.base	py_base
py_backwards.transformers.dict_unpacking	dict_unpacking
py_backwards.transformers.return_from_generator	return_from_generator
py_backwards.transformers.yield_from	yield_from
py_backwards.utils.helpers	py_helpers
pymonet.immutable_list	immutable_list
pymonet.maybe	maybe
pymonet.validation	validation
pypara.accounting.journaling	journaling
pytutils.lazy.lazy_import	lazy_import
pytutils.python	python
sanic.config	config
sanic.helpers	s_helpers
sanic.mixins.signals	signals
thonny.plugins.pgzero_frontend	pgzero_frontend
typesystem.tokenize.positional_validation	positional_validation

Table B.2: Shortened names for the 12 parameters.

Original name	Shortened name
Change Parameter Probability	ChangeParamProb
Chromosome Length	ChromLen
Crossover Rate	Crossover
Elite	Elite
Population Size	Pop
Random Perturbation	RandPert
Statement Insertion Probability	StatemInsertProb
Test Change Probability	TestChangeProb
Test Delete Probability	TestDeleteProb
Test Insert Probability	TestInsertProb
Test Insertion Probability	TestInsertionProb
Tournament Size	TourSize

C

Module Metrics

Table C.1: Metrics for the 24 modules used in the experiments. Includes the number of logical lines of code (LLOC), classes, functions, branches, the total cyclomatic complexity (CC), and the average cyclomatic complexity per function for each module. SonarQube [53] and the Python package Radon [54] were used to collect these metrics. See Table B.1 for the full-length module names.

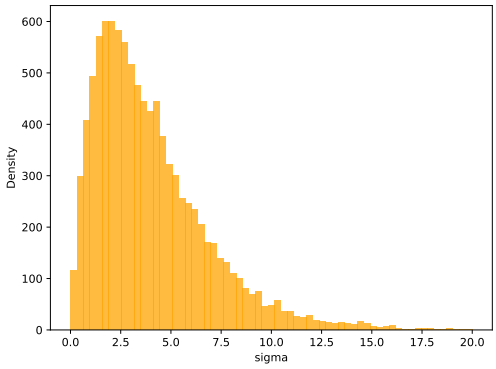
Module	LLOC	Classes	Functions	Branches	Total CC	Avg CC
cmd	106	0	7	30	20	2.9
config	355	2	16	80	43	2.7
da	81	2	5	25	8	1.6
decorators	53	1	4	9	6	1.5
dict_unpacking	62	1	7	25	13	1.9
h_base	113	5	9	15	9	1.0
headers	13	1	2	5	2	1.0
immutable_list	127	1	14	45	33	2.4
journaling	138	4	10	40	16	1.6
lazy_import	397	4	23	101	64	2.8
maybe	142	1	14	36	29	2.1
namedtupleutils	160	0	5	35	20	4.0
packages	262	2	5	79	39	7.8
pgzero_frontend	24	0	3	5	4	1.3
positional_validation	32	0	1	8	3	3.0
py_base	105	3	12	39	25	2.1
py_helpers	32	1	6	11	7	1.2
python	35	2	1	9	3	3.0
return_from_generator	58	1	4	35	15	3.8
s_helpers	154	1	8	23	13	1.6
signals	116	1	9	16	13	1.4
timer	82	4	5	27	14	2.8
validation	123	1	15	24	20	1.3
yield_from	65	1	7	23	19	2.7

D

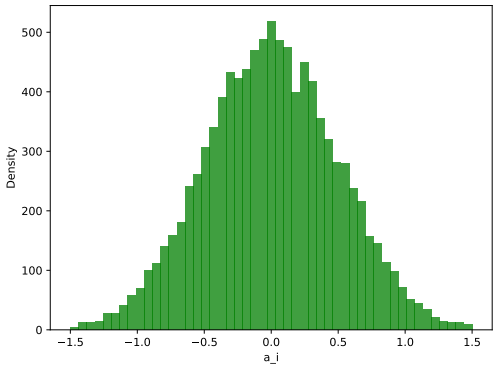
Example Bayesian Model

Figure D.1: Prior checks for the example Bayesian model (Part 1).

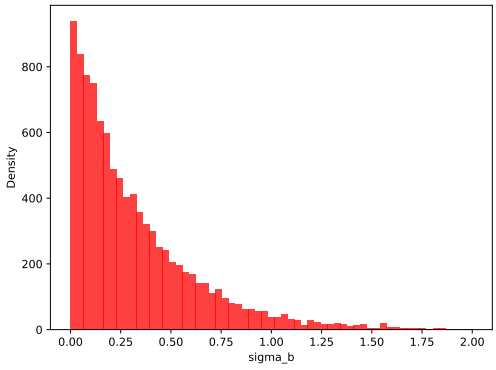
(a) Prior check for σ.



(b) Prior check for α_i.



(c) Prior check for σ_β.



(d) Prior check for β_j.

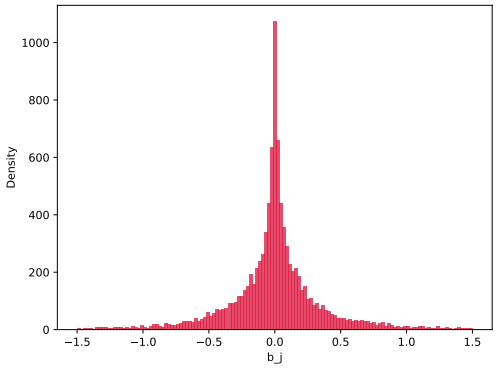
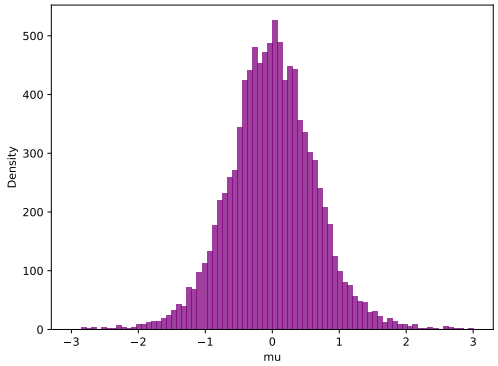


Figure D.2: Prior check for μ in the example Bayesian model (Part 2).

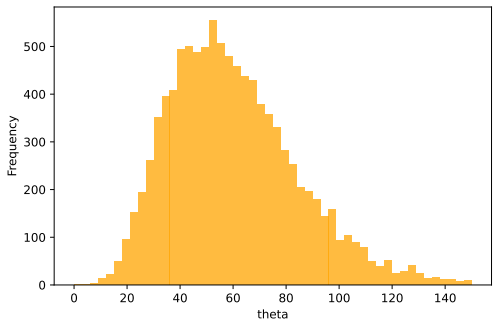


E

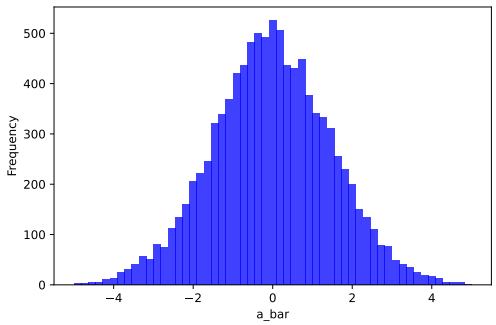
Final Branch Coverage Model

Figure E.1: Prior checks for the final branch coverage model (part 1).

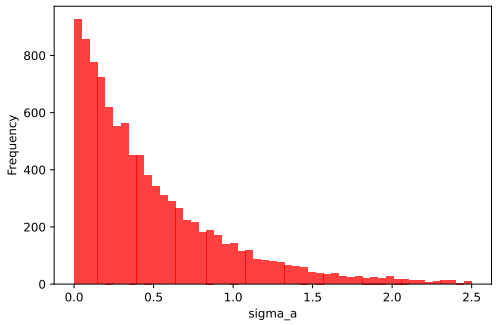
(a) Prior check for θ.



(b) Prior check for $\bar{\alpha}$.



(c) Prior check for σ_α.



(d) Prior check for α_m.

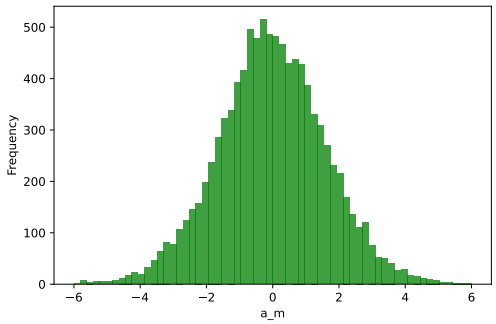
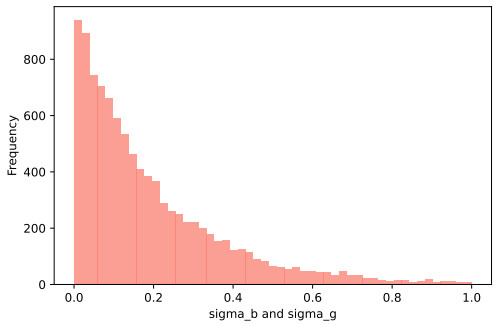
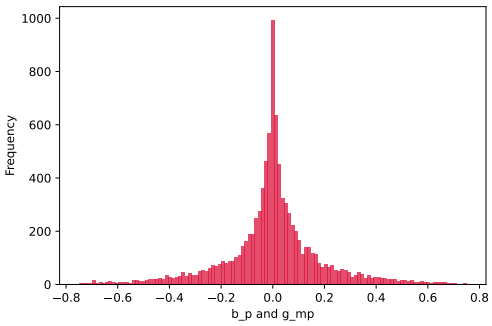


Figure E.2: Prior checks for the final branch coverage model (part 2).

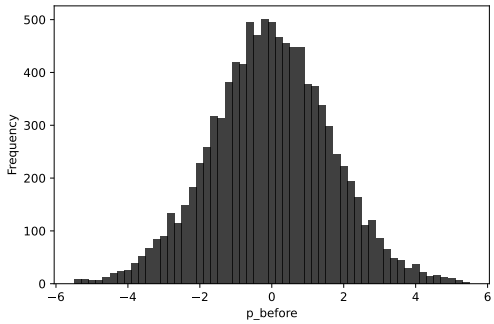
(a) Prior check for σ_β and σ_γ.



(b) Prior check for β_p and γ_{mp}.



(c) Prior check for $\bar{p}$.



(d) Prior check for $\sigma(\bar{p})$.

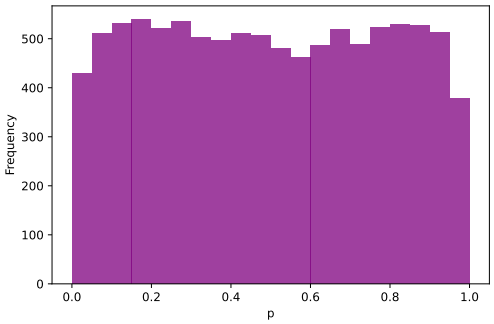


Figure E.3: A prior predictive simulation for the final branch coverage model.

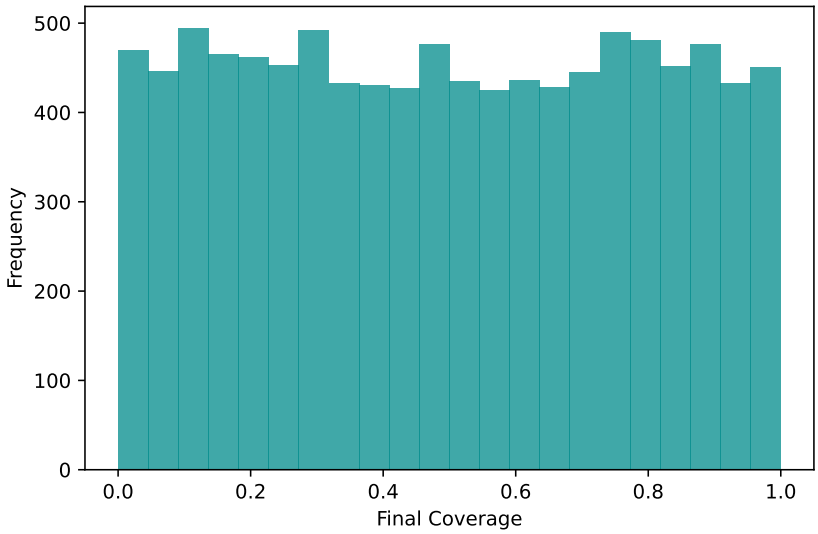


Table E.1: A summary table for the module intercept (α_m) posterior distributions for the single-parameter final branch coverage model (see Appendix B for full names of modules).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
a_m[packages]	-2.907	0.034	-2.961	-2.854	0.0	0.0	21505.0	16165.0	1.0
a_m[cmd]	-2.492	0.03	-2.541	-2.445	0.0	0.0	22147.0	16181.0	1.0
a_m[dict_unpacking]	-1.817	0.025	-1.855	-1.777	0.0	0.0	21691.0	17480.0	1.0
a_m[signals]	-1.417	0.022	-1.453	-1.381	0.0	0.0	22225.0	16838.0	1.0
a_m[return_from_generator]	-1.222	0.022	-1.258	-1.188	0.0	0.0	21213.0	15847.0	1.0
a_m[python]	-1.216	0.022	-1.251	-1.182	0.0	0.0	21153.0	15871.0	1.0
a_m[py_base]	-0.951	0.021	-0.983	-0.917	0.0	0.0	21735.0	16299.0	1.0
a_m[positional_validation]	-0.911	0.021	-0.944	-0.878	0.0	0.0	22461.0	16220.0	1.0
a_m[journaling]	-0.737	0.02	-0.77	-0.706	0.0	0.0	23203.0	15788.0	1.0
a_m[lazy_import]	-0.12	0.019	-0.152	-0.09	0.0	0.0	22297.0	16086.0	1.0
a_m[yield_from]	-0.043	0.02	-0.074	-0.011	0.0	0.0	21968.0	16930.0	1.0
a_m[pgzero_frontend]	0.396	0.02	0.365	0.427	0.0	0.0	21947.0	16339.0	1.0
a_m[headers]	0.396	0.02	0.365	0.427	0.0	0.0	21195.0	15555.0	1.0
a_m[py_helpers]	0.546	0.02	0.516	0.579	0.0	0.0	22715.0	16083.0	1.0
a_m[decorators]	0.676	0.02	0.645	0.709	0.0	0.0	21717.0	16565.0	1.0
a_m[h_base]	1.085	0.021	1.051	1.119	0.0	0.0	22331.0	15198.0	1.0
a_m[immutable_list]	1.098	0.021	1.064	1.132	0.0	0.0	21524.0	16727.0	1.0
a_m[s_helpers]	1.117	0.021	1.082	1.15	0.0	0.0	21195.0	16611.0	1.0
a_m[config]	1.282	0.022	1.246	1.316	0.0	0.0	23140.0	16523.0	1.0
a_m[da]	1.345	0.022	1.309	1.379	0.0	0.0	21652.0	16208.0	1.0
a_m[timer]	1.932	0.025	1.891	1.973	0.0	0.0	21105.0	17135.0	1.0
a_m[maybe]	2.281	0.028	2.238	2.327	0.0	0.0	22018.0	16877.0	1.0
a_m[validation]	2.281	0.028	2.236	2.324	0.0	0.0	20792.0	17273.0	1.0
a_m[namedtupleutils]	2.63	0.031	2.58	2.679	0.0	0.0	20574.0	14332.0	1.0

Table E.2: A summary table for the parameter effect (β_p) posterior distributions for the single-parameter final branch coverage model (see Appendix B for full names of parameters).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b_p[Pop]	-0.017	0.015	-0.039	0.003	0.0	0.0	7959.0	15565.0	1.0
b_p[ChromLen]	-0.007	0.011	-0.025	0.009	0.0	0.0	16894.0	16807.0	1.0
b_p[TestDeleteProb]	-0.007	0.011	-0.025	0.009	0.0	0.0	15533.0	16642.0	1.0
b_p[StatemInsertProb]	-0.003	0.01	-0.019	0.014	0.0	0.0	27862.0	15045.0	1.0
b_p[Crossover]	-0.001	0.01	-0.017	0.016	0.0	0.0	28463.0	16694.0	1.0
b_p[TourSize]	0.0	0.01	-0.016	0.017	0.0	0.0	28456.0	17397.0	1.0
b_p[TestChangeProb]	0.002	0.01	-0.014	0.019	0.0	0.0	25234.0	16116.0	1.0
b_p[Elite]	0.003	0.01	-0.012	0.021	0.0	0.0	24445.0	17063.0	1.0
b_p[RandPert]	0.004	0.011	-0.011	0.022	0.0	0.0	22779.0	17326.0	1.0
b_p[TestInsertProb]	0.005	0.011	-0.011	0.023	0.0	0.0	19108.0	17214.0	1.0
b_p[ChangeParamProb]	0.006	0.011	-0.01	0.024	0.0	0.0	20063.0	16722.0	1.0
b_p[TestInsertionProb]	0.011	0.012	-0.007	0.03	0.0	0.0	11652.0	16005.0	1.0

Table E.3: A summary table containing 40 interaction effect (γ_{mp}) posterior distributions for the single-parameter final branch coverage model. These 40 are a subset of the total 288 distributions and include the 20 with the smallest mean and the 20 with the largest mean. See Appendix B for full names of modules and parameters.

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
g_mp[return_from_generator x Pop]	-0.226	0.058	-0.315	-0.132	0.001	0.0	10506.0	12304.0	1.0
g_mp[return_from_generator x ChromLen]	-0.088	0.041	-0.153	-0.023	0.0	0.0	21392.0	14670.0	1.0
g_mp[return_from_generator x TestDeleteProb]	-0.078	0.04	-0.141	-0.014	0.0	0.0	19762.0	14801.0	1.0
g_mp[return_from_generator x StatemInsertProb]	-0.061	0.039	-0.124	-0.0	0.0	0.0	25763.0	14554.0	1.0
g_mp[positional_validation x TestInsertProb]	-0.037	0.036	-0.094	0.021	0.0	0.0	30501.0	14509.0	1.0
g_mp[yield_from x ChangeParamProb]	-0.033	0.036	-0.089	0.025	0.0	0.0	30052.0	13992.0	1.0
g_mp[py_base x RandPert]	-0.028	0.036	-0.085	0.03	0.0	0.0	32281.0	14855.0	1.0
g_mp[journaling x Crossover]	-0.024	0.036	-0.081	0.033	0.0	0.0	30601.0	14779.0	1.0
g_mp[journaling x TourSize]	-0.022	0.036	-0.079	0.036	0.0	0.0	28668.0	14024.0	1.0
g_mp[config x Pop]	-0.021	0.037	-0.08	0.038	0.0	0.0	29518.0	15056.0	1.0
g_mp[s_helpers x ChangeParamProb]	-0.021	0.036	-0.079	0.037	0.0	0.0	35806.0	13983.0	1.0
g_mp[yield_from x TestDeleteProb]	-0.02	0.035	-0.075	0.037	0.0	0.0	32862.0	15802.0	1.0
g_mp[yield_from x RandPert]	-0.02	0.035	-0.077	0.036	0.0	0.0	35990.0	14847.0	1.0
g_mp[positional_validation x TestChangeProb]	-0.018	0.036	-0.077	0.038	0.0	0.0	33644.0	14697.0	1.0
g_mp[positional_validation x Crossover]	-0.018	0.036	-0.073	0.042	0.0	0.0	35484.0	15016.0	1.0
g_mp[yield_from x TourSize]	-0.018	0.035	-0.071	0.04	0.0	0.0	36330.0	14922.0	1.0
g_mp[positional_validation x TourSize]	-0.018	0.036	-0.073	0.041	0.0	0.0	37365.0	15106.0	1.0
g_mp[py_base x StatemInsertProb]	-0.016	0.036	-0.073	0.043	0.0	0.0	32759.0	14734.0	1.0
g_mp[h_base x Elite]	-0.016	0.036	-0.075	0.041	0.0	0.0	36166.0	15488.0	1.0
g_mp[positional_validation x ChromLen]	-0.016	0.036	-0.073	0.042	0.0	0.0	28941.0	15022.0	1.0
...	...	...	...	...	...	...	...	...	...
g_mp[py_base x Elite]	0.015	0.036	-0.04	0.074	0.0	0.0	32635.0	14561.0	1.0
g_mp[packages x TourSize]	0.015	0.041	-0.048	0.082	0.0	0.0	39282.0	13401.0	1.0
g_mp[s_helpers x TestInsertProb]	0.016	0.037	-0.041	0.076	0.0	0.0	34013.0	13989.0	1.0
g_mp[config x ChromLen]	0.017	0.037	-0.042	0.076	0.0	0.0	34624.0	14833.0	1.0
g_mp[h_base x Crossover]	0.018	0.037	-0.045	0.072	0.0	0.0	35651.0	14473.0	1.0
g_mp[return_from_generator x Elite]	0.018	0.037	-0.042	0.077	0.0	0.0	34262.0	15194.0	1.0
g_mp[h_base x TourSize]	0.022	0.037	-0.04	0.077	0.0	0.0	29577.0	14609.0	1.0
g_mp[packages x TestDeleteProb]	0.022	0.041	-0.042	0.087	0.0	0.0	30754.0	14994.0	1.0
g_mp[dict_unpacking x TestInsertionProb]	0.024	0.039	-0.039	0.087	0.0	0.0	33056.0	14658.0	1.0
g_mp[py_base x Crossover]	0.026	0.037	-0.034	0.083	0.0	0.0	35454.0	14184.0	1.0
g_mp[yield_from x TestInsertionProb]	0.028	0.035	-0.028	0.084	0.0	0.0	31575.0	15278.0	1.0
g_mp[s_helpers x TestChangeProb]	0.029	0.036	-0.028	0.086	0.0	0.0	29910.0	13399.0	1.0
g_mp[h_base x TestInsertionProb]	0.031	0.036	-0.027	0.088	0.0	0.0	28672.0	15172.0	1.0
g_mp[positional_validation x Pop]	0.046	0.037	-0.012	0.105	0.0	0.0	28526.0	14864.0	1.0
g_mp[return_from_generator x TourSize]	0.046	0.037	-0.015	0.102	0.0	0.0	24807.0	13936.0	1.0
g_mp[yield_from x Elite]	0.047	0.036	-0.012	0.104	0.0	0.0	28666.0	14626.0	1.0
g_mp[return_from_generator x TestInsertionProb]	0.07	0.04	0.006	0.133	0.0	0.0	22712.0	14222.0	1.0
g_mp[return_from_generator x TestInsertProb]	0.087	0.041	0.021	0.15	0.0	0.0	18670.0	14151.0	1.0
g_mp[return_from_generator x RandPert]	0.116	0.045	0.041	0.183	0.0	0.0	16655.0	14039.0	1.0
g_mp[return_from_generator x ChangeParamProb]	0.133	0.046	0.058	0.204	0.0	0.0	14448.0	13541.0	1.0

Table E.4: A summary table for the module intercept (α_m) posterior distributions for the multi-parameter final branch coverage model (see Appendix B for full names of modules).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
a_m[packages]	-2.882	0.025	-2.922	-2.844	0.0	0.0	20935.0	15074.0	1.0
a_m[cmd]	-2.488	0.022	-2.522	-2.453	0.0	0.0	21318.0	15202.0	1.0
a_m[dict_unpacking]	-1.785	0.018	-1.814	-1.757	0.0	0.0	20845.0	15383.0	1.0
a_m[signals]	-1.418	0.016	-1.444	-1.393	0.0	0.0	20380.0	15812.0	1.0
a_m[python]	-1.216	0.015	-1.24	-1.192	0.0	0.0	21613.0	15104.0	1.0
a_m[return_from_generator]	-0.975	0.015	-0.998	-0.952	0.0	0.0	19850.0	15011.0	1.0
a_m[py_base]	-0.911	0.014	-0.934	-0.888	0.0	0.0	20902.0	14904.0	1.0
a_m[positional_validation]	-0.85	0.014	-0.873	-0.827	0.0	0.0	21731.0	16048.0	1.0
a_m[journaling]	-0.741	0.014	-0.763	-0.719	0.0	0.0	21403.0	15741.0	1.0
a_m[lazy_import]	-0.117	0.013	-0.139	-0.096	0.0	0.0	21390.0	14746.0	1.0
a_m[yield_from]	0.068	0.013	0.047	0.09	0.0	0.0	21806.0	15141.0	1.0
a_m[pgzero_frontend]	0.396	0.013	0.375	0.418	0.0	0.0	21162.0	14673.0	1.0
a_m[headers]	0.396	0.014	0.373	0.417	0.0	0.0	22126.0	15139.0	1.0
a_m[py_helpers]	0.545	0.014	0.524	0.568	0.0	0.0	20414.0	14440.0	1.0
a_m[decorators]	0.676	0.014	0.654	0.699	0.0	0.0	21916.0	14095.0	1.0
a_m[immutable_list]	1.096	0.015	1.073	1.12	0.0	0.0	21617.0	15615.0	1.0
a_m[s_helpers]	1.121	0.015	1.097	1.145	0.0	0.0	21395.0	15233.0	1.0
a_m[h_base]	1.143	0.015	1.12	1.167	0.0	0.0	22274.0	15991.0	1.0
a_m[config]	1.295	0.016	1.271	1.321	0.0	0.0	22049.0	14411.0	1.0
a_m[da]	1.343	0.016	1.317	1.367	0.0	0.0	21538.0	15049.0	1.0
a_m[timer]	1.932	0.018	1.903	1.961	0.0	0.0	20498.0	15091.0	1.0
a_m[maybe]	2.277	0.02	2.245	2.309	0.0	0.0	19617.0	15988.0	1.0
a_m[validation]	2.277	0.02	2.244	2.309	0.0	0.0	19790.0	15555.0	1.0
a_m[namedtupleutils]	2.622	0.023	2.584	2.657	0.0	0.0	20743.0	15270.0	1.0

Table E.5: A summary table for the parameter effect (β_p) posterior distributions for the multi-parameter final branch coverage model (see Appendix B for full names of parameters).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b_p[Elite|TestChangeProb]	-0.011	0.013	-0.032	0.006	0.0	0.0	7516.0	14565.0	1.0
b_p[Elite|TourSize]	-0.006	0.01	-0.023	0.009	0.0	0.0	15323.0	13734.0	1.0
b_p[ChromLen|TestDeleteProb]	-0.006	0.011	-0.023	0.01	0.0	0.0	15688.0	15012.0	1.0
b_p[StatemInsertProb|TestDeleteProb]	-0.006	0.011	-0.024	0.009	0.0	0.0	14106.0	15260.0	1.0
b_p[ChromLen|Crossover]	-0.005	0.01	-0.022	0.01	0.0	0.0	18625.0	14397.0	1.0
b_p[ChangeParamProb|ChromLen]	-0.004	0.01	-0.021	0.011	0.0	0.0	21125.0	14392.0	1.0
b_p[Elite|TestInsertProb]	-0.004	0.01	-0.02	0.011	0.0	0.0	20242.0	13187.0	1.0
b_p[TestChangeProb|TestDeleteProb]	-0.004	0.01	-0.02	0.011	0.0	0.0	23102.0	13913.0	1.0
b_p[TestChangeProb|TourSize]	-0.003	0.01	-0.019	0.012	0.0	0.0	24149.0	15072.0	1.0
b_p[ChangeParamProb|TestDeleteProb]	-0.003	0.01	-0.021	0.011	0.0	0.0	23178.0	15302.0	1.0
b_p[ChromLen|TourSize]	-0.003	0.01	-0.02	0.012	0.0	0.0	31660.0	15352.0	1.0
b_p[Elite|Pop]	-0.003	0.01	-0.02	0.012	0.0	0.0	24169.0	15364.0	1.0
b_p[Elite|TestDeleteProb]	-0.002	0.01	-0.018	0.012	0.0	0.0	29359.0	14316.0	1.0
b_p[Crossover|RandPert]	-0.002	0.01	-0.017	0.014	0.0	0.0	29169.0	13499.0	1.0
b_p[Crossover|Pop]	-0.002	0.01	-0.018	0.013	0.0	0.0	29626.0	14168.0	1.0
b_p[Pop|TestChangeProb]	-0.002	0.01	-0.018	0.013	0.0	0.0	26440.0	15352.0	1.0
b_p[Elite|RandPert]	-0.002	0.009	-0.017	0.013	0.0	0.0	28195.0	14115.0	1.0
b_p[ChromLen|Pop]	-0.002	0.009	-0.017	0.013	0.0	0.0	30269.0	15841.0	1.0
b_p[ChromLen|Elite]	-0.002	0.01	-0.018	0.012	0.0	0.0	27202.0	16202.0	1.0
b_p[StatemInsertProb|TourSize]	-0.002	0.01	-0.017	0.013	0.0	0.0	33687.0	14469.0	1.0
b_p[ChangeParamProb|TestChangeProb]	-0.002	0.01	-0.019	0.013	0.0	0.0	29550.0	15420.0	1.0
b_p[RandPert|StatemInsertProb]	-0.002	0.01	-0.018	0.013	0.0	0.0	33439.0	14223.0	1.0
b_p[Pop|TestDeleteProb]	-0.001	0.01	-0.016	0.015	0.0	0.0	34729.0	13930.0	1.0
b_p[StatemInsertProb|TestInsertProb]	-0.001	0.01	-0.017	0.014	0.0	0.0	34096.0	13935.0	1.0
b_p[ChangeParamProb|RandPert]	-0.001	0.01	-0.016	0.015	0.0	0.0	36591.0	14719.0	1.0
b_p[Crossover|TestChangeProb]	-0.001	0.009	-0.017	0.014	0.0	0.0	36876.0	15495.0	1.0
b_p[ChangeParamProb|Pop]	-0.001	0.01	-0.017	0.014	0.0	0.0	30339.0	14087.0	1.0
b_p[TestDeleteProb|TestInsertProb]	-0.001	0.009	-0.016	0.015	0.0	0.0	31668.0	14900.0	1.0
b_p[TestDeleteProb|TourSize]	-0.001	0.009	-0.017	0.013	0.0	0.0	30698.0	14124.0	1.0
b_p[TestInsertProb|TourSize]	-0.001	0.01	-0.016	0.015	0.0	0.0	30266.0	14592.0	1.0
b_p[ChromLen|StatemInsertProb]	-0.001	0.009	-0.015	0.015	0.0	0.0	32766.0	14836.0	1.0
b_p[Pop|TourSize]	0.0	0.009	-0.015	0.015	0.0	0.0	37048.0	14221.0	1.0
b_p[TestInsertionProb|TourSize]	-0.0	0.009	-0.016	0.015	0.0	0.0	35647.0	14185.0	1.0
b_p[Crossover|StatemInsertProb]	-0.0	0.01	-0.016	0.015	0.0	0.0	37237.0	15088.0	1.0
b_p[ChangeParamProb|Crossover]	-0.0	0.01	-0.015	0.015	0.0	0.0	33797.0	15189.0	1.0
b_p[ChromLen|TestInsertProb]	0.0	0.01	-0.015	0.016	0.0	0.0	32987.0	14350.0	1.0
b_p[Elite|StatemInsertProb]	0.001	0.01	-0.014	0.017	0.0	0.0	36237.0	14952.0	1.0
b_p[Crossover|TestInsertProb]	0.001	0.01	-0.014	0.017	0.0	0.0	32593.0	14411.0	1.0
b_p[Crossover|TestDeleteProb]	0.001	0.01	-0.014	0.016	0.0	0.0	33305.0	14732.0	1.0
b_p[TestChangeProb|TestInsertionProb]	0.001	0.009	-0.014	0.017	0.0	0.0	34210.0	14871.0	1.0
b_p[Pop|RandPert]	0.001	0.01	-0.015	0.016	0.0	0.0	38265.0	15111.0	1.0
b_p[Pop|StatemInsertProb]	0.001	0.01	-0.014	0.017	0.0	0.0	32801.0	14893.0	1.0
b_p[ChangeParamProb|TourSize]	0.001	0.01	-0.015	0.016	0.0	0.0	34173.0	15761.0	1.0
b_p[ChromLen|RandPert]	0.001	0.01	-0.014	0.017	0.0	0.0	33831.0	15730.0	1.0
b_p[ChangeParamProb|StatemInsertProb]	0.001	0.01	-0.013	0.017	0.0	0.0	29444.0	13770.0	1.0
b_p[RandPert|TourSize]	0.001	0.01	-0.013	0.017	0.0	0.0	34345.0	14144.0	1.0
b_p[Crossover|TourSize]	0.001	0.01	-0.015	0.016	0.0	0.0	34117.0	14725.0	1.0
b_p[RandPert|TestDeleteProb]	0.001	0.01	-0.014	0.017	0.0	0.0	34554.0	14057.0	1.0
b_p[ChromLen|TestChangeProb]	0.002	0.01	-0.013	0.018	0.0	0.0	35307.0	14924.0	1.0
b_p[ChangeParamProb|TestInsertProb]	0.002	0.01	-0.013	0.018	0.0	0.0	32745.0	14491.0	1.0
b_p[Pop|TestInsertionProb]	0.002	0.01	-0.013	0.018	0.0	0.0	27643.0	15406.0	1.0
b_p[Pop|TestInsertProb]	0.002	0.01	-0.013	0.018	0.0	0.0	31660.0	14536.0	1.0
b_p[TestInsertProb|TestInsertionProb]	0.002	0.01	-0.012	0.019	0.0	0.0	28739.0	13972.0	1.0
b_p[ChangeParamProb|Elite]	0.002	0.01	-0.012	0.019	0.0	0.0	27744.0	14655.0	1.0
b_p[TestDeleteProb|TestInsertionProb]	0.003	0.01	-0.012	0.019	0.0	0.0	26035.0	13929.0	1.0
b_p[Elite|TestInsertionProb]	0.003	0.01	-0.012	0.019	0.0	0.0	27380.0	15836.0	1.0
b_p[ChangeParamProb|TestInsertionProb]	0.003	0.01	-0.013	0.019	0.0	0.0	29171.0	14575.0	1.0
b_p[TestChangeProb|TestInsertProb]	0.003	0.01	-0.012	0.018	0.0	0.0	26990.0	15182.0	1.0
b_p[Crossover|TestInsertionProb]	0.004	0.01	-0.011	0.02	0.0	0.0	19395.0	14486.0	1.0
b_p[RandPert|TestInsertProb]	0.004	0.01	-0.011	0.02	0.0	0.0	21550.0	14535.0	1.0
b_p[RandPert|TestChangeProb]	0.004	0.01	-0.011	0.021	0.0	0.0	19657.0	14643.0	1.0
b_p[Crossover|Elite]	0.005	0.01	-0.01	0.023	0.0	0.0	16212.0	14413.0	1.0
b_p[StatemInsertProb|TestChangeProb]	0.007	0.011	-0.009	0.024	0.0	0.0	12574.0	14284.0	1.0
b_p[StatemInsertProb|TestInsertionProb]	0.008	0.012	-0.008	0.026	0.0	0.0	9297.0	13719.0	1.0
b_p[RandPert|TestInsertionProb]	0.011	0.013	-0.007	0.031	0.0	0.0	7327.0	15573.0	1.0
b_p[ChromLen|TestInsertionProb]	0.012	0.014	-0.006	0.034	0.0	0.0	6827.0	13794.0	1.0

Table E.6: A summary table containing 40 interaction effect (γ_{mp}) posterior distributions for the multi-parameter final branch coverage model. These 40 are a subset of the total 1584 distributions and include the 20 with the smallest mean and the 20 with the largest mean. See Appendix B for full names of modules and parameters.

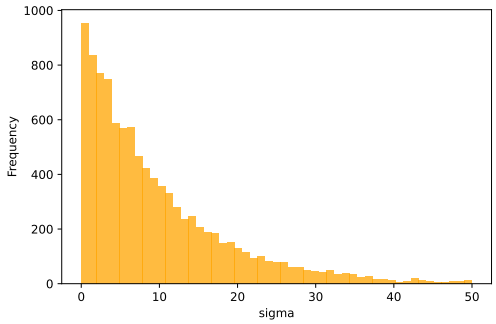
	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
g__mp[return_from_generator x ChromLen|TestDeleteProb]	-0.179	0.056	-0.264	-0.085	0.001	0.0	8933.0	10362.0	1.0
g__mp[return_from_generator x Elite|TestChangeProb]	-0.138	0.052	-0.218	-0.052	0.0	0.0	11784.0	11538.0	1.0
g__mp[return_from_generator x ChromLen|Crossover]	-0.101	0.048	-0.178	-0.024	0.0	0.0	15876.0	13756.0	1.0
g__mp[return_from_generator x Crossover|RandPert]	-0.099	0.048	-0.176	-0.023	0.0	0.0	15735.0	13175.0	1.0
g__mp[return_from_generator x Elite|TourSize]	-0.097	0.048	-0.172	-0.021	0.0	0.0	15558.0	13410.0	1.0
g__mp[return_from_generator x ChangeParamProb|StatemInsertProb]	-0.063	0.046	-0.136	0.01	0.0	0.0	27634.0	14351.0	1.0
g__mp[yield_from x Elite|TestChangeProb]	-0.063	0.045	-0.135	0.011	0.0	0.0	23461.0	14175.0	1.0
g__mp[return_from_generator x StatemInsertProb|TestDeleteProb]	-0.062	0.046	-0.134	0.011	0.0	0.0	24197.0	14036.0	1.0
g__mp[return_from_generator x Pop|TestInsertionProb]	-0.061	0.045	-0.133	0.011	0.0	0.0	23929.0	13845.0	1.0
g__mp[return_from_generator x ChromLen|Pop]	-0.061	0.045	-0.13	0.014	0.0	0.0	25307.0	14143.0	1.0
g__mp[return_from_generator x Pop|TestChangeProb]	-0.06	0.046	-0.136	0.01	0.0	0.0	24537.0	14734.0	1.0
g__mp[return_from_generator x ChromLen|TestChangeProb]	-0.059	0.045	-0.129	0.015	0.0	0.0	26076.0	14274.0	1.0
g__mp[return_from_generator x ChromLen|TourSize]	-0.059	0.045	-0.129	0.016	0.0	0.0	18990.0	14744.0	1.0
g__mp[return_from_generator x TestDeleteProb|TestInsertionProb]	-0.059	0.046	-0.134	0.014	0.0	0.0	23787.0	14437.0	1.0
g__mp[return_from_generator x ChangeParamProb|Pop]	-0.059	0.045	-0.131	0.014	0.0	0.0	23298.0	14685.0	1.0
g__mp[return_from_generator x StatemInsertProb|TourSize]	-0.059	0.046	-0.131	0.015	0.0	0.0	27260.0	13147.0	1.0
g__mp[return_from_generator x Elite|Pop]	-0.058	0.046	-0.13	0.017	0.0	0.0	25899.0	14174.0	1.0
g__mp[return_from_generator x ChangeParamProb|TestChangeProb]	-0.057	0.045	-0.131	0.012	0.0	0.0	24962.0	14042.0	1.0
g__mp[py_base x Crossover|TestInsertProb]	-0.052	0.045	-0.125	0.019	0.0	0.0	30308.0	14237.0	1.0
g__mp[yield_from x Elite|Pop]	-0.051	0.044	-0.121	0.019	0.0	0.0	26417.0	15065.0	1.0
...	...	...	...	...	...	...	...	...	...
g__mp[packages x TestInsertProb|TestInsertionProb]	0.05	0.049	-0.027	0.127	0.0	0.0	27974.0	14142.0	1.0
g__mp[positional_validation x TestDeleteProb|TestInsertionProb]	0.053	0.045	-0.018	0.124	0.0	0.0	28006.0	14890.0	1.0
g__mp[yield_from x StatemInsertProb|TestInsertionProb]	0.063	0.045	-0.009	0.134	0.0	0.0	24412.0	15063.0	1.0
g__mp[h_base x TestDeleteProb|TestInsertionProb]	0.069	0.046	-0.004	0.143	0.0	0.0	21341.0	14526.0	1.0
g__mp[positional_validation x ChangeParamProb|StatemInsertProb]	0.071	0.047	-0.005	0.143	0.0	0.0	21879.0	14644.0	1.0
g__mp[positional_validation x ChromLen|TestChangeProb]	0.071	0.047	-0.005	0.144	0.0	0.0	20714.0	14064.0	1.0
g__mp[positional_validation x Pop|StatemInsertProb]	0.071	0.047	-0.001	0.148	0.0	0.0	19957.0	14368.0	1.0
g__mp[return_from_generator x StatemInsertProb|TestInsertionProb]	0.074	0.046	0.001	0.148	0.0	0.0	19449.0	14226.0	1.0
g__mp[return_from_generator x RandPert|TestDeleteProb]	0.075	0.047	-0.001	0.149	0.0	0.0	20489.0	14041.0	1.0
g__mp[return_from_generator x ChromLen|RandPert]	0.075	0.047	0.002	0.15	0.0	0.0	20389.0	14352.0	1.0
g__mp[return_from_generator x Crossover|TestInsertionProb]	0.075	0.047	0.001	0.15	0.0	0.0	22201.0	14149.0	1.0
g__mp[return_from_generator x Pop|TourSize]	0.079	0.048	0.0	0.151	0.0	0.0	21986.0	14703.0	1.0
g__mp[return_from_generator x ChangeParamProb|TestInsertionProb]	0.079	0.047	0.004	0.155	0.0	0.0	19551.0	14257.0	1.0
g__mp[return_from_generator x Crossover|TourSize]	0.08	0.048	0.002	0.154	0.0	0.0	19053.0	13884.0	1.0
g__mp[return_from_generator x TestChangeProb|TestInsertProb]	0.081	0.048	0.003	0.157	0.0	0.0	18439.0	13866.0	1.0
g__mp[positional_validation x ChromLen|Pop]	0.099	0.049	0.018	0.173	0.0	0.0	16464.0	14128.0	1.0
g__mp[return_from_generator x Crossover|TestInsertProb]	0.126	0.053	0.04	0.208	0.0	0.0	13501.0	11849.0	1.0
g__mp[return_from_generator x RandPert|TestInsertionProb]	0.173	0.059	0.078	0.264	0.001	0.0	9368.0	9537.0	1.0
g__mp[return_from_generator x StatemInsertProb|TestChangeProb]	0.174	0.059	0.077	0.266	0.001	0.0	9247.0	11045.0	1.0
g__mp[return_from_generator x ChromLen|TestInsertionProb]	0.174	0.06	0.08	0.27	0.001	0.0	9687.0	10082.0	1.0

F

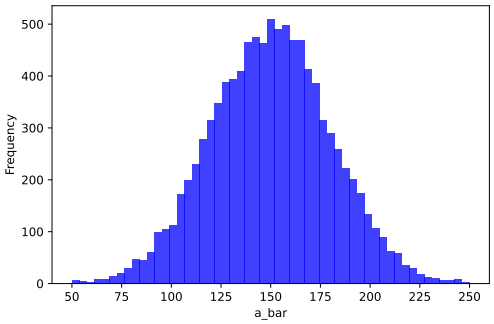
Branch Coverage Growth Rate Model

Figure F.1: Prior checks for the branch coverage growth rate model (part 1).

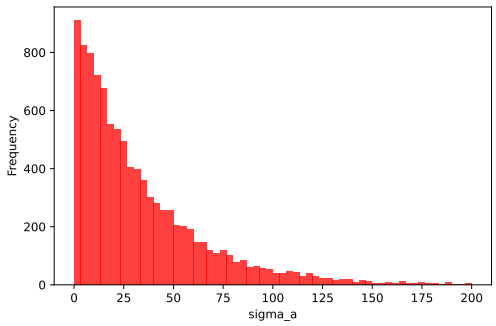
(a) Prior check for σ.



(b) Prior check for $\bar{\alpha}$.



(c) Prior check for σ_α.



(d) Prior check for α_m.

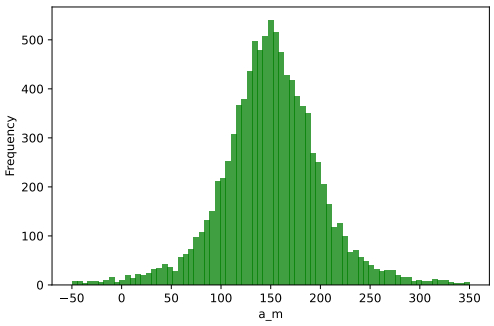
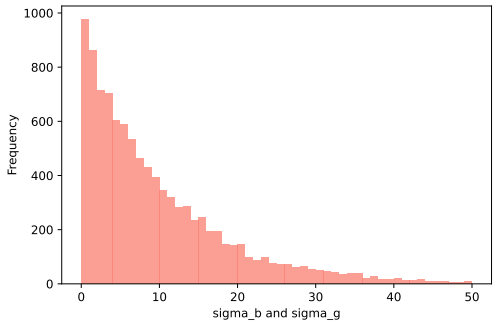
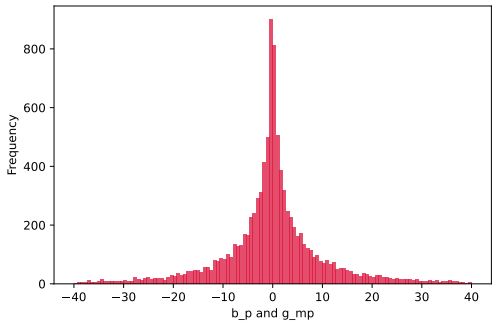


Figure F.2: Prior checks for the branch coverage growth rate model (part 2).

(a) Prior check for σ_β and σ_γ.



(b) Prior check for β_p and γ_{mp}.



(c) Prior check for μ.

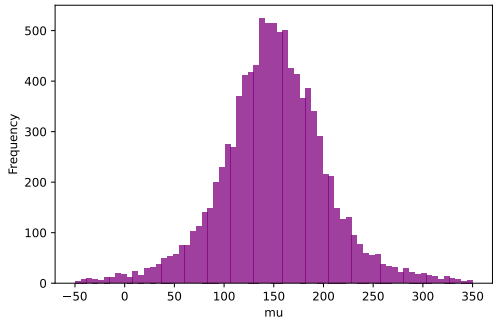


Figure F.3: A prior predictive simulation for the branch coverage growth rate model.

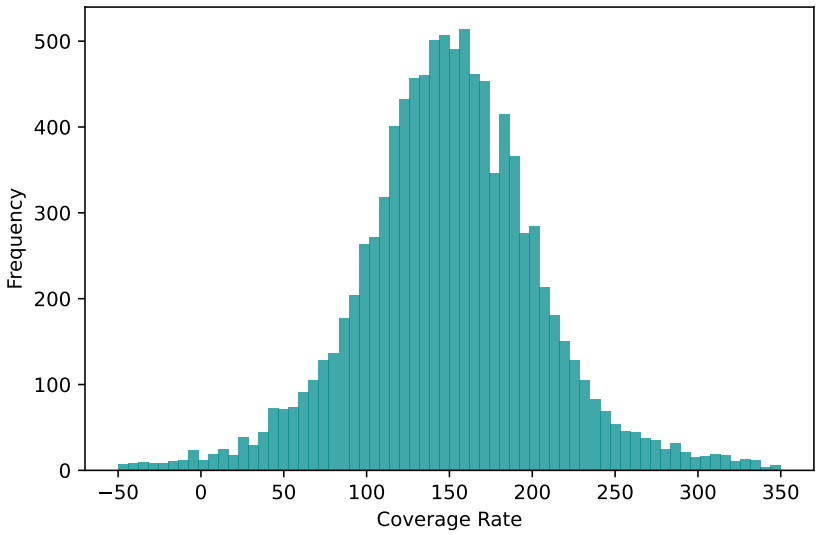


Table F.1: A summary table for the module intercept (α_m) posterior distributions for the single-parameter branch coverage growth rate model (see Appendix B for full names of modules).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
a_m[packages]	13.796	0.844	12.472	15.16	0.006	0.004	20388.0	17521.0	1.0
a_m[cmd]	19.959	0.842	18.71	21.412	0.006	0.004	20955.0	17639.0	1.0
a_m[dict_unpacking]	37.938	0.843	36.548	39.243	0.006	0.004	19778.0	18547.0	1.0
a_m[signals]	56.204	0.836	54.874	57.534	0.006	0.004	19888.0	16918.0	1.0
a_m[return_from_generator]	60.199	0.841	58.865	61.551	0.006	0.004	19236.0	17543.0	1.0
a_m[py_base]	65.421	0.839	64.116	66.794	0.006	0.004	20937.0	17723.0	1.0
a_m[python]	66.475	0.84	65.112	67.782	0.006	0.004	20650.0	17975.0	1.0
a_m[positional_validation]	80.254	0.852	78.903	81.619	0.006	0.004	20954.0	16304.0	1.0
a_m[journaling]	92.123	0.842	90.832	93.523	0.006	0.004	19877.0	17386.0	1.0
a_m[yield_from]	126.706	0.839	125.347	128.012	0.006	0.004	20065.0	17120.0	1.0
a_m[lazy_import]	136.109	0.842	134.793	137.499	0.006	0.004	20208.0	16781.0	1.0
a_m[headers]	179.414	0.829	178.131	180.773	0.006	0.004	20442.0	16550.0	1.0
a_m[pgzero_frontend]	179.418	0.833	178.128	180.772	0.006	0.004	21423.0	17883.0	1.0
a_m[py_helpers]	187.194	0.845	185.83	188.521	0.006	0.004	20029.0	17174.0	1.0
a_m[decorators]	199.349	0.839	198.014	200.699	0.006	0.004	20340.0	16014.0	1.0
a_m[s_helpers]	219.529	0.841	218.126	220.818	0.006	0.004	21158.0	16966.0	1.0
a_m[h_base]	221.368	0.847	220.07	222.774	0.006	0.004	20673.0	16279.0	1.0
a_m[immutable_list]	225.838	0.834	224.509	227.168	0.006	0.004	20986.0	17909.0	1.0
a_m[config]	230.458	0.843	229.122	231.805	0.006	0.004	20898.0	17948.0	1.0
a_m[da]	239.218	0.834	237.902	240.556	0.006	0.004	20936.0	17194.0	1.0
a_m[timer]	258.799	0.838	257.478	260.155	0.006	0.004	21055.0	17899.0	1.0
a_m[validation]	273.575	0.845	272.249	274.939	0.006	0.004	20913.0	17809.0	1.0
a_m[maybe]	273.985	0.848	272.667	275.373	0.006	0.004	19696.0	17023.0	1.0
a_m[namedtupleutils]	276.089	0.843	274.778	277.469	0.006	0.004	21195.0	17427.0	1.0

Table F.2: A summary table for the parameter effect (β_p) posterior distributions for the single-parameter branch coverage growth rate model (see Appendix B for full names of parameters).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b_p[Pop]	-0.322	0.431	-1.015	0.248	0.004	0.003	10616.0	14325.0	1.0
b_p[Crossover]	-0.271	0.401	-0.937	0.267	0.004	0.003	11538.0	15972.0	1.0
b_p[ChromLen]	-0.268	0.403	-0.935	0.264	0.004	0.003	11461.0	15050.0	1.0
b_p[TestDeleteProb]	-0.181	0.37	-0.77	0.365	0.003	0.002	17209.0	15300.0	1.0
b_p[StatemInsertProb]	-0.041	0.345	-0.598	0.513	0.002	0.002	26029.0	17555.0	1.0
b_p[TourSize]	0.019	0.347	-0.534	0.584	0.002	0.002	26888.0	16617.0	1.0
b_p[ChangeParamProb]	0.053	0.34	-0.458	0.648	0.002	0.002	26652.0	17816.0	1.0
b_p[TestInsertProb]	0.077	0.346	-0.462	0.65	0.002	0.002	25152.0	16167.0	1.0
b_p[TestChangeProb]	0.09	0.346	-0.44	0.656	0.002	0.002	20362.0	16384.0	1.0
b_p[Elite]	0.1	0.348	-0.441	0.669	0.002	0.002	22696.0	16992.0	1.0
b_p[RandPert]	0.141	0.353	-0.381	0.735	0.003	0.002	18461.0	16399.0	1.0
b_p[TestInsertionProb]	0.316	0.418	-0.247	0.983	0.004	0.003	10381.0	16178.0	1.0

Table F.3: A summary table containing 40 interaction effect (γ_{mp}) posterior distributions for the single-parameter branch coverage growth rate model. These 40 are a subset of the total 288 distributions and include the 20 with the smallest mean and the 20 with the largest mean. See Appendix B for full names of modules and parameters.

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
g_mp[return_from_generator x Pop]	-7.822	1.924	-10.82	-4.694	0.016	0.011	14749.0	13306.0	1.0
g_mp[return_from_generator x ChromLen]	-6.045	1.787	-8.97	-3.248	0.014	0.01	16888.0	14842.0	1.0
g_mp[return_from_generator x Crossover]	-5.79	1.753	-8.483	-2.885	0.013	0.009	19290.0	15460.0	1.0
g_mp[return_from_generator x TestDeleteProb]	-4.562	1.68	-7.116	-1.758	0.012	0.009	18784.0	14865.0	1.0
g_mp[yield_from x ChangeParamProb]	-2.833	1.591	-5.433	-0.345	0.009	0.008	29591.0	14180.0	1.0
g_mp[positional_validation x TestInsertProb]	-2.008	1.57	-4.401	0.591	0.009	0.008	29092.0	15198.0	1.0
g_mp[py_base x RandPert]	-1.8	1.535	-4.277	0.593	0.008	0.008	33650.0	14442.0	1.0
g_mp[return_from_generator x StatemInsertProb]	-1.779	1.558	-4.246	0.717	0.009	0.009	27680.0	14620.0	1.0
g_mp[yield_from x TourSize]	-1.246	1.527	-3.591	1.285	0.009	0.009	31155.0	14642.0	1.0
g_mp[py_base x TestDeleteProb]	-1.214	1.517	-3.697	1.137	0.009	0.009	32255.0	15475.0	1.0
g_mp[yield_from x Crossover]	-1.187	1.525	-3.617	1.258	0.009	0.009	28365.0	15594.0	1.0
g_mp[positional_validation x Crossover]	-1.178	1.538	-3.617	1.291	0.009	0.009	30827.0	14904.0	1.0
g_mp[positional_validation x StatemInsertProb]	-1.145	1.512	-3.585	1.226	0.009	0.009	30962.0	14263.0	1.0
g_mp[positional_validation x TestChangeProb]	-1.086	1.522	-3.479	1.37	0.009	0.009	31317.0	15510.0	1.0
g_mp[packages x TestInsertionProb]	-1.045	1.559	-3.56	1.433	0.009	0.01	32600.0	15444.0	1.0
g_mp[journaling x Crossover]	-1.01	1.524	-3.403	1.421	0.009	0.009	29214.0	15541.0	1.0
g_mp[packages x TestChangeProb]	-0.964	1.542	-3.413	1.47	0.009	0.01	26519.0	15074.0	1.0
g_mp[packages x Elite]	-0.963	1.519	-3.395	1.444	0.009	0.009	28762.0	16089.0	1.0
g_mp[packages x ChangeParamProb]	-0.944	1.523	-3.37	1.498	0.008	0.009	33204.0	13884.0	1.0
g_mp[positional_validation x TourSize]	-0.867	1.532	-3.287	1.59	0.009	0.01	31446.0	14215.0	1.0
...	...	...	...	...	...	...	...	...	...
g_mp[packages x RandPert]	0.722	1.538	-1.691	3.204	0.008	0.011	33634.0	14032.0	1.0
g_mp[yield_from x TestInsertionProb]	0.748	1.515	-1.722	3.082	0.009	0.01	30466.0	14918.0	1.0
g_mp[return_from_generator x TestChangeProb]	0.821	1.543	-1.555	3.353	0.01	0.01	24397.0	15645.0	1.0
g_mp[packages x TourSize]	0.881	1.51	-1.482	3.35	0.009	0.009	30990.0	15995.0	1.0
g_mp[return_from_generator x Elite]	0.984	1.522	-1.452	3.417	0.009	0.009	27233.0	15429.0	1.0
g_mp[journaling x StatemInsertProb]	1.044	1.528	-1.325	3.568	0.009	0.009	29903.0	15759.0	1.0
g_mp[py_base x TestChangeProb]	1.066	1.51	-1.284	3.509	0.008	0.009	36902.0	13570.0	1.0
g_mp[journaling x Elite]	1.109	1.52	-1.328	3.545	0.009	0.009	28718.0	14775.0	1.0
g_mp[yield_from x TestChangeProb]	1.287	1.538	-1.127	3.805	0.009	0.009	29906.0	15214.0	1.0
g_mp[positional_validation x Pop]	1.448	1.524	-0.99	3.894	0.009	0.009	30994.0	14708.0	1.0
g_mp[dict_unpacking x TestInsertionProb]	1.475	1.556	-0.98	3.969	0.01	0.009	26532.0	14125.0	1.0
g_mp[positional_validation x ChangeParamProb]	1.651	1.555	-0.757	4.208	0.008	0.009	34697.0	14659.0	1.0
g_mp[py_base x Crossover]	2.185	1.529	-0.317	4.546	0.009	0.008	31068.0	14789.0	1.0
g_mp[yield_from x Elite]	2.48	1.573	-0.013	4.977	0.01	0.008	26781.0	14941.0	1.0
g_mp[return_from_generator x TestInsertProb]	2.561	1.567	-0.074	4.959	0.009	0.008	28537.0	14908.0	1.0
g_mp[packages x TestDeleteProb]	2.982	1.572	0.424	5.416	0.01	0.008	25407.0	15031.0	1.0
g_mp[return_from_generator x TourSize]	3.103	1.573	0.559	5.562	0.01	0.007	27052.0	14770.0	1.0
g_mp[return_from_generator x ChangeParamProb]	4.339	1.658	1.719	6.983	0.011	0.008	22493.0	15044.0	1.0
g_mp[return_from_generator x TestInsertionProb]	5.619	1.755	2.766	8.356	0.012	0.009	20697.0	14260.0	1.0
g_mp[return_from_generator x RandPert]	7.265	1.851	4.26	10.126	0.014	0.01	16540.0	13967.0	1.0

Table F.4: A summary table for the module intercept (α_m) posterior distributions for the multi-parameter branch coverage growth rate model (see Appendix B for full names of modules).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
a_m[packages]	14.285	0.561	13.372	15.171	0.004	0.003	20412.0	16298.0	1.0
a_m[cmd]	19.937	0.563	19.017	20.812	0.004	0.003	20018.0	17577.0	1.0
a_m[dict_unpacking]	38.773	0.567	37.876	39.691	0.004	0.003	20541.0	16928.0	1.0
a_m[signals]	56.073	0.56	55.175	56.963	0.004	0.003	19709.0	16263.0	1.0
a_m[python]	66.444	0.572	65.523	67.354	0.004	0.003	20068.0	15257.0	1.0
a_m[return_from_generator]	68.869	0.562	67.945	69.734	0.004	0.003	19464.0	16585.0	1.0
a_m[py_base]	69.39	0.569	68.478	70.295	0.004	0.003	19939.0	15745.0	1.0
a_m[positional_validation]	83.95	0.565	83.073	84.875	0.004	0.003	20173.0	15606.0	1.0
a_m[journaling]	92.079	0.569	91.132	92.944	0.004	0.003	18814.0	14676.0	1.0
a_m[yield_from]	133.772	0.564	132.857	134.653	0.004	0.003	19987.0	15783.0	1.0
a_m[lazy_import]	136.23	0.56	135.352	137.135	0.004	0.003	20063.0	15165.0	1.0
a_m[headers]	179.401	0.565	178.529	180.331	0.004	0.003	20546.0	15128.0	1.0
a_m[pgzero_frontend]	179.404	0.562	178.528	180.322	0.004	0.003	20286.0	14197.0	1.0
a_m[py_helpers]	187.177	0.564	186.246	188.039	0.004	0.003	19611.0	14967.0	1.0
a_m[decorators]	199.326	0.562	198.379	200.177	0.004	0.003	20132.0	15036.0	1.0
a_m[s_helpers]	219.568	0.563	218.695	220.5	0.004	0.003	20758.0	15078.0	1.0
a_m[h_base]	222.213	0.561	221.307	223.105	0.004	0.003	19658.0	16257.0	1.0
a_m[immutable_list]	225.806	0.559	224.918	226.701	0.004	0.003	20428.0	17238.0	1.0
a_m[config]	230.736	0.562	229.82	231.619	0.004	0.003	19521.0	16813.0	1.0
a_m[da]	239.195	0.566	238.281	240.085	0.004	0.003	20088.0	15871.0	1.0
a_m[timer]	258.611	0.565	257.746	259.554	0.004	0.003	20296.0	16865.0	1.0
a_m[validation]	273.579	0.563	272.69	274.487	0.004	0.003	20881.0	16491.0	1.0
a_m[maybe]	273.982	0.56	273.085	274.881	0.004	0.003	19955.0	16474.0	1.0
a_m[namedtupleutils]	276.102	0.563	275.194	276.984	0.004	0.003	19711.0	15018.0	1.0

Table F.5: A summary table for the parameter effect (β_p) posterior distributions for the multi-parameter branch coverage growth rate model (see Appendix B for full names of parameters).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b_p[Elite|TestChangeProb]	-0.175	0.316	-0.662	0.235	0.003	0.002	15136.0	15575.0	1.0
b_p[StatemInsertProb|TestDeleteProb]	-0.157	0.308	-0.641	0.253	0.003	0.002	14734.0	14725.0	1.0
b_p[Elite|TestDeleteProb]	-0.113	0.282	-0.559	0.278	0.002	0.002	19236.0	15279.0	1.0
b_p[ChromLen|Pop]	-0.099	0.273	-0.533	0.295	0.002	0.002	21004.0	16483.0	1.0
b_p[Pop|TestDeleteProb]	-0.089	0.274	-0.534	0.31	0.002	0.002	20179.0	15151.0	1.0
b_p[ChromLen|Crossover]	-0.081	0.268	-0.498	0.331	0.002	0.002	18825.0	16135.0	1.0
b_p[TestInsertProb|TourSize]	-0.077	0.265	-0.501	0.315	0.002	0.002	20607.0	14795.0	1.0
b_p[ChromLen|TourSize]	-0.071	0.268	-0.496	0.333	0.002	0.002	23892.0	15543.0	1.0
b_p[ChromLen|TestDeleteProb]	-0.071	0.266	-0.489	0.34	0.002	0.002	23110.0	15100.0	1.0
b_p[RandPert|StatemInsertProb]	-0.063	0.264	-0.485	0.328	0.002	0.002	24599.0	15625.0	1.0
b_p[TestChangeProb|TestDeleteProb]	-0.062	0.264	-0.477	0.339	0.002	0.002	25130.0	15707.0	1.0
b_p[Elite|TestInsertProb]	-0.059	0.264	-0.471	0.339	0.002	0.002	23713.0	15428.0	1.0
b_p[ChangeParamProb|ChromLen]	-0.055	0.265	-0.472	0.356	0.002	0.002	26216.0	15272.0	1.0
b_p[Crossover|StatemInsertProb]	-0.047	0.26	-0.456	0.352	0.002	0.002	23070.0	15929.0	1.0
b_p[Elite|TourSize]	-0.047	0.262	-0.462	0.358	0.002	0.002	26539.0	16217.0	1.0
b_p[ChangeParamProb|StatemInsertProb]	-0.046	0.261	-0.494	0.331	0.002	0.002	25726.0	15789.0	1.0
b_p[Elite|RandPert]	-0.041	0.257	-0.434	0.372	0.002	0.002	26087.0	16480.0	1.0
b_p[Crossover|Pop]	-0.039	0.259	-0.463	0.345	0.002	0.002	28281.0	16037.0	1.0
b_p[Pop|TestChangeProb]	-0.034	0.265	-0.472	0.36	0.002	0.002	25391.0	15857.0	1.0
b_p[TestDeleteProb|TourSize]	-0.032	0.255	-0.427	0.371	0.002	0.002	23347.0	15267.0	1.0
b_p[TestDeleteProb|TestInsertProb]	-0.031	0.254	-0.445	0.354	0.002	0.002	25068.0	15889.0	1.0
b_p[Pop|TourSize]	-0.021	0.254	-0.43	0.36	0.002	0.002	25251.0	15700.0	1.0
b_p[StatemInsertProb|TestInsertProb]	-0.019	0.257	-0.406	0.4	0.001	0.002	32667.0	14751.0	1.0
b_p[Crossover|TestDeleteProb]	-0.019	0.259	-0.411	0.407	0.002	0.002	25627.0	15086.0	1.0
b_p[TestChangeProb|TourSize]	-0.019	0.26	-0.439	0.385	0.002	0.002	25059.0	16112.0	1.0
b_p[ChangeParamProb|TestInsertionProb]	-0.016	0.257	-0.422	0.388	0.002	0.002	26493.0	15996.0	1.0
b_p[Crossover|RandPert]	-0.016	0.254	-0.411	0.388	0.002	0.002	25105.0	15040.0	1.0
b_p[ChangeParamProb|TestDeleteProb]	-0.015	0.254	-0.424	0.378	0.002	0.002	22498.0	14896.0	1.0
b_p[RandPert|TestDeleteProb]	-0.014	0.261	-0.455	0.374	0.002	0.002	26160.0	15312.0	1.0
b_p[ChromLen|StatemInsertProb]	-0.011	0.251	-0.416	0.373	0.002	0.002	27467.0	16458.0	1.0
b_p[ChangeParamProb|Crossover]	-0.006	0.254	-0.415	0.381	0.002	0.002	28987.0	15969.0	1.0
b_p[Crossover|TestChangeProb]	-0.003	0.256	-0.405	0.406	0.002	0.002	28418.0	15788.0	1.0
b_p[Pop|RandPert]	0.002	0.254	-0.397	0.399	0.002	0.002	27314.0	15633.0	1.0
b_p[TestChangeProb|TestInsertionProb]	0.007	0.259	-0.424	0.4	0.002	0.002	27122.0	15590.0	1.0
b_p[ChangeParamProb|TourSize]	0.01	0.261	-0.414	0.406	0.002	0.002	28569.0	16066.0	1.0
b_p[Crossover|TourSize]	0.011	0.253	-0.391	0.407	0.002	0.002	23829.0	15824.0	1.0
b_p[ChangeParamProb|TestChangeProb]	0.014	0.255	-0.393	0.407	0.002	0.002	24223.0	16034.0	1.0
b_p[TestInsertionProb|TourSize]	0.018	0.254	-0.365	0.444	0.002	0.002	26180.0	15525.0	1.0
b_p[ChromLen|RandPert]	0.02	0.256	-0.397	0.409	0.002	0.002	26026.0	16041.0	1.0
b_p[RandPert|TestChangeProb]	0.02	0.256	-0.369	0.431	0.002	0.002	30362.0	15672.0	1.0
b_p[Elite|StatemInsertProb]	0.021	0.251	-0.386	0.406	0.001	0.002	31512.0	16052.0	1.0
b_p[Crossover|Elite]	0.022	0.255	-0.396	0.417	0.002	0.002	27548.0	14725.0	1.0
b_p[StatemInsertProb|TourSize]	0.023	0.26	-0.385	0.432	0.002	0.002	26520.0	15458.0	1.0
b_p[ChromLen|Elite]	0.024	0.253	-0.365	0.429	0.002	0.002	26795.0	16600.0	1.0
b_p[ChangeParamProb|Elite]	0.026	0.26	-0.367	0.457	0.002	0.002	25819.0	15151.0	1.0
b_p[Elite|Pop]	0.027	0.261	-0.393	0.432	0.002	0.002	29947.0	16539.0	1.0
b_p[Pop|StatemInsertProb]	0.027	0.254	-0.353	0.447	0.002	0.002	28977.0	16802.0	1.0
b_p[ChromLen|TestInsertProb]	0.028	0.258	-0.367	0.44	0.002	0.002	28873.0	15267.0	1.0
b_p[ChangeParamProb|RandPert]	0.028	0.259	-0.386	0.423	0.001	0.002	34426.0	15384.0	1.0
b_p[TestChangeProb|TestInsertProb]	0.029	0.255	-0.361	0.437	0.002	0.002	23646.0	15386.0	1.0
b_p[RandPert|TestInsertProb]	0.03	0.258	-0.367	0.448	0.002	0.002	25567.0	16316.0	1.0
b_p[Pop|TestInsertProb]	0.031	0.26	-0.384	0.44	0.002	0.002	26630.0	15901.0	1.0
b_p[Crossover|TestInsertionProb]	0.04	0.261	-0.351	0.459	0.002	0.002	27142.0	15624.0	1.0
b_p[StatemInsertProb|TestChangeProb]	0.043	0.259	-0.372	0.435	0.002	0.002	29157.0	16094.0	1.0
b_p[Crossover|TestInsertProb]	0.058	0.262	-0.33	0.475	0.002	0.002	24631.0	15690.0	1.0
b_p[TestDeleteProb|TestInsertionProb]	0.059	0.264	-0.348	0.477	0.002	0.002	26332.0	15592.0	1.0
b_p[ChangeParamProb|Pop]	0.06	0.262	-0.329	0.487	0.002	0.002	24157.0	16234.0	1.0
b_p[ChangeParamProb|TestInsertProb]	0.064	0.264	-0.317	0.498	0.002	0.002	24846.0	15780.0	1.0
b_p[Pop|TestInsertionProb]	0.067	0.27	-0.334	0.494	0.002	0.002	20633.0	15865.0	1.0
b_p[ChromLen|TestChangeProb]	0.068	0.27	-0.341	0.495	0.002	0.002	27859.0	15534.0	1.0
b_p[TestInsertProb|TestInsertionProb]	0.089	0.277	-0.305	0.534	0.002	0.002	23458.0	15531.0	1.0
b_p[Elite|TestInsertionProb]	0.098	0.277	-0.309	0.538	0.002	0.002	21099.0	14899.0	1.0
b_p[RandPert|TourSize]	0.101	0.281	-0.308	0.545	0.002	0.002	18577.0	15378.0	1.0
b_p[StatemInsertProb|TestInsertionProb]	0.141	0.303	-0.262	0.622	0.002	0.002	17121.0	15704.0	1.0
b_p[RandPert|TestInsertionProb]	0.184	0.324	-0.238	0.673	0.003	0.002	13787.0	14635.0	1.0
b_p[ChromLen|TestInsertionProb]	0.206	0.339	-0.202	0.74	0.003	0.002	13426.0	15827.0	1.0

Table F.6: A summary table containing 40 interaction effect (γ_{mp}) posterior distributions for the multi-parameter branch coverage growth rate model. These 40 are a subset of the total 1584 distributions and include the 20 with the smallest mean and the 20 with the largest mean. See Appendix B for full names of modules and parameters.

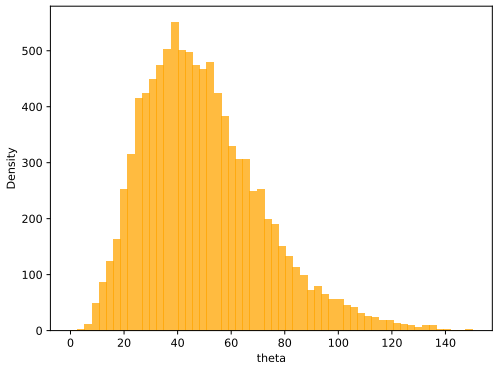
	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
g_mp[return_from_generator x ChromLen|TestDeleteProb]	-1.521	1.472	-3.763	0.324	0.03	0.021	2435.0	8932.0	1.0
g_mp[return_from_generator x RandPert|TestInsertProb]	-1.13	1.245	-3.073	0.485	0.022	0.016	3189.0	10427.0	1.0
g_mp[return_from_generator x Elite|TestChangeProb]	-1.105	1.24	-3.036	0.504	0.022	0.015	3334.0	9022.0	1.0
g_mp[return_from_generator x ChromLen|TourSize]	-0.906	1.129	-2.77	0.555	0.017	0.012	4620.0	10123.0	1.0
g_mp[return_from_generator x StatemInsertProb|TestDeleteProb]	-0.889	1.136	-2.641	0.694	0.017	0.012	4972.0	9942.0	1.0
g_mp[return_from_generator x TestInsertProb|TourSize]	-0.858	1.116	-2.608	0.686	0.017	0.012	4694.0	9973.0	1.0
g_mp[return_from_generator x Elite|TestDeleteProb]	-0.781	1.098	-2.526	0.737	0.015	0.011	5236.0	9706.0	1.0
g_mp[yield_from x TestChangeProb|TestInsertionProb]	-0.734	1.055	-2.43	0.74	0.015	0.01	5783.0	11703.0	1.0
g_mp[return_from_generator x Crossover|RandPert]	-0.639	1.025	-2.273	0.864	0.013	0.009	6751.0	10850.0	1.0
g_mp[return_from_generator x TestInsertProb|TestInsertionProb]	-0.619	1.008	-2.241	0.842	0.012	0.009	7539.0	12148.0	1.0
g_mp[return_from_generator x Elite|TourSize]	-0.603	1.007	-2.246	0.845	0.012	0.009	8231.0	12098.0	1.0
g_mp[return_from_generator x ChromLen|Pop]	-0.6	1.011	-2.227	0.87	0.012	0.009	7882.0	11166.0	1.0
g_mp[return_from_generator x Pop|TestChangeProb]	-0.566	1.004	-2.2	0.916	0.011	0.009	8947.0	10965.0	1.0
g_mp[return_from_generator x RandPert|TestChangeProb]	-0.566	1.01	-2.231	0.877	0.011	0.009	9260.0	10696.0	1.0
g_mp[return_from_generator x Pop|TestDeleteProb]	-0.557	1.009	-2.292	0.825	0.011	0.009	8958.0	11660.0	1.0
g_mp[yield_from x Elite|TestChangeProb]	-0.546	0.982	-2.045	0.971	0.012	0.009	7868.0	11661.0	1.0
g_mp[yield_from x Crossover|RandPert]	-0.524	0.982	-2.117	0.925	0.01	0.009	10308.0	10894.0	1.0
g_mp[yield_from x ChromLen|Pop]	-0.517	0.979	-2.075	0.958	0.011	0.009	9222.0	10307.0	1.0
g_mp[return_from_generator x ChangeParamProb|Crossover]	-0.513	0.996	-2.199	0.915	0.01	0.009	10817.0	11517.0	1.0
g_mp[return_from_generator x TestDeleteProb|TestInsertionProb]	-0.5	0.977	-2.138	0.871	0.01	0.008	9677.0	12025.0	1.0
...	...	...	...	...	...	...	...	...	...
g_mp[return_from_generator x Elite|StatemInsertProb]	0.633	1.02	-0.785	2.329	0.013	0.009	6835.0	9941.0	1.0
g_mp[yield_from x StatemInsertProb|TestInsertionProb]	0.634	1.029	-0.736	2.392	0.014	0.01	6208.0	9647.0	1.0
g_mp[packages x Crossover|RandPert]	0.65	1.031	-0.757	2.373	0.013	0.01	6593.0	11159.0	1.0
g_mp[positional_validation x TestDeleteProb|TestInsertionProb]	0.653	1.03	-0.829	2.327	0.014	0.01	6711.0	10323.0	1.0
g_mp[positional_validation x ChromLen|TestChangeProb]	0.661	1.035	-0.772	2.355	0.013	0.009	7485.0	11049.0	1.0
g_mp[dict_unpacking x ChangeParamProb|TestChangeProb]	0.68	1.043	-0.828	2.359	0.014	0.01	6582.0	9541.0	1.0
g_mp[return_from_generator x Pop|RandPert]	0.694	1.042	-0.676	2.48	0.013	0.01	6898.0	10209.0	1.0
g_mp[positional_validation x Elite|TestInsertionProb]	0.732	1.071	-0.776	2.445	0.015	0.011	5577.0	9704.0	1.0
g_mp[packages x Pop|TestInsertProb]	0.743	1.069	-0.784	2.416	0.015	0.011	5291.0	9747.0	1.0
g_mp[positional_validation x RandPert|TestChangeProb]	0.84	1.116	-0.69	2.616	0.016	0.012	4903.0	11141.0	1.0
g_mp[return_from_generator x StatemInsertProb|TestInsertionProb]	0.91	1.146	-0.604	2.741	0.018	0.013	4280.0	10608.0	1.0
g_mp[positional_validation x Pop|StatemInsertProb]	1.016	1.187	-0.569	2.878	0.019	0.014	3932.0	10276.0	1.0
g_mp[return_from_generator x Crossover|TestInsertProb]	1.03	1.199	-0.564	2.911	0.02	0.014	3554.0	9365.0	1.0
g_mp[return_from_generator x ChromLen|RandPert]	1.059	1.212	-0.589	2.895	0.021	0.015	3461.0	9907.0	1.0
g_mp[return_from_generator x RandPert|TourSize]	1.155	1.264	-0.531	3.049	0.022	0.016	3236.0	10725.0	1.0
g_mp[return_from_generator x StatemInsertProb|TestChangeProb]	1.157	1.258	-0.475	3.086	0.022	0.016	3188.0	9296.0	1.0
g_mp[packages x RandPert|TestInsertProb]	1.207	1.296	-0.509	3.175	0.024	0.017	2861.0	8225.0	1.0
g_mp[return_from_generator x ChromLen|TestInsertionProb]	1.249	1.318	-0.416	3.293	0.024	0.017	3150.0	9999.0	1.0
g_mp[return_from_generator x RandPert|TestInsertionProb]	1.332	1.362	-0.43	3.412	0.026	0.018	2722.0	8998.0	1.0
g_mp[packages x TestInsertProb|TestInsertionProb]	1.519	1.475	-0.344	3.78	0.029	0.021	2451.0	9208.0	1.0

G

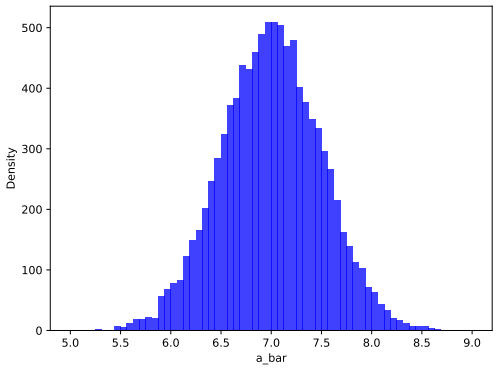
Overhead Model

Figure G.1: Prior checks for the overhead model (part 1).

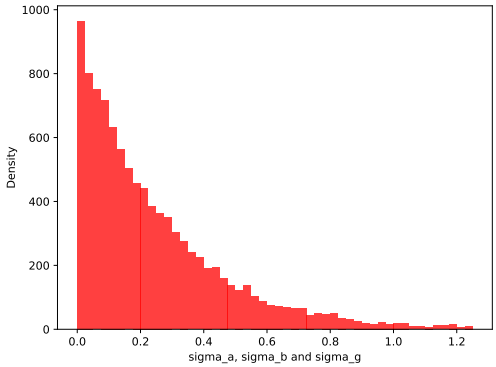
(a) Prior check for θ.



(b) Prior check for $\bar{\alpha}$.



(c) Prior check for σ_α, σ_β,and σ_γ.



(d) Prior check for α_m.

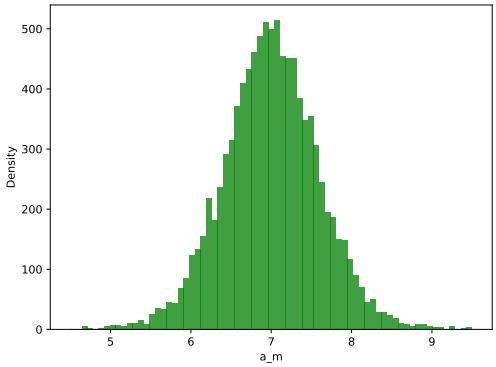
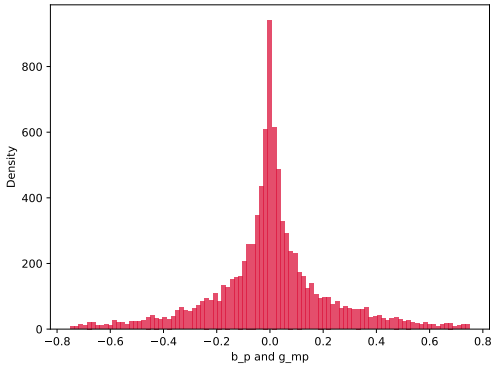
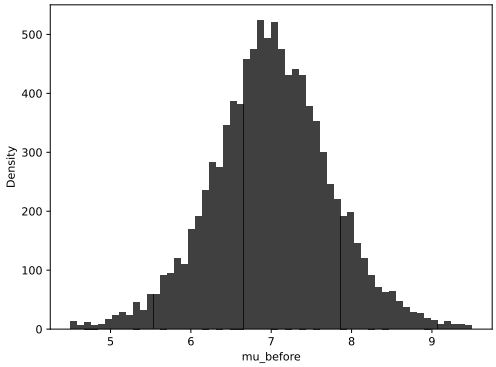


Figure G.2: Prior checks for the overhead model (part 2).

(a) Prior check for β_p and γ_{mp}.



(b) Prior check for $log(\mu)$.



(c) Prior check for μ.

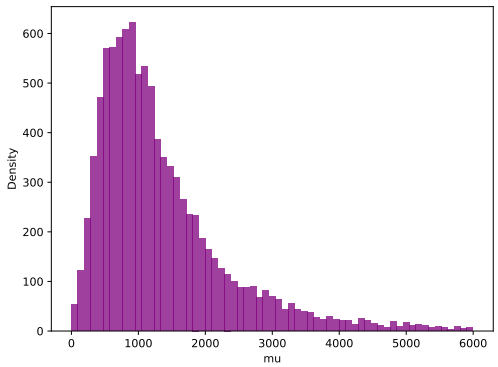


Figure G.3: A prior predictive simulation for the overhead model.

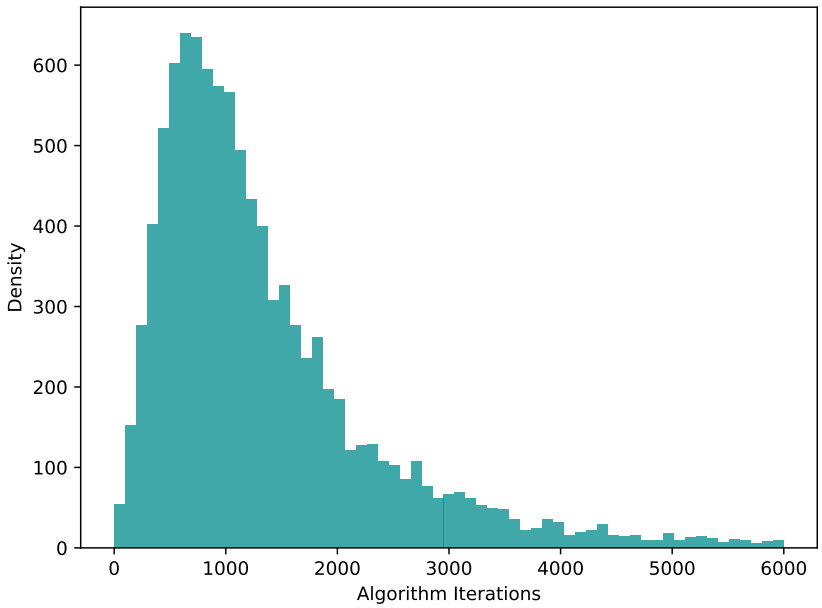


Table G.1: A summary table for the module intercept (α_m) posterior distributions for the single-parameter overhead model (see Appendix B for full names of modules).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
a_m[da]	6.429	0.013	6.409	6.45	0.0	0.0	21759.0	18090.0	1.0
a_m[config]	6.723	0.013	6.703	6.744	0.0	0.0	22122.0	18252.0	1.0
a_m[dict_unpacking]	6.909	0.013	6.889	6.93	0.0	0.0	23024.0	18401.0	1.0
a_m[return_from_generator]	6.987	0.013	6.966	7.007	0.0	0.0	22187.0	17821.0	1.0
a_m[py_base]	7.005	0.013	6.984	7.025	0.0	0.0	23819.0	17087.0	1.0
a_m[maybe]	7.05	0.013	7.03	7.071	0.0	0.0	23305.0	17988.0	1.0
a_m[immutable_list]	7.113	0.013	7.092	7.134	0.0	0.0	22682.0	16916.0	1.0
a_m[yield_from]	7.119	0.013	7.099	7.14	0.0	0.0	22789.0	17341.0	1.0
a_m[cmd]	7.135	0.013	7.114	7.155	0.0	0.0	23124.0	18198.0	1.0
a_m[timer]	7.166	0.013	7.146	7.186	0.0	0.0	23156.0	17556.0	1.0
a_m[namedtupleutils]	7.258	0.013	7.237	7.278	0.0	0.0	23553.0	16791.0	1.0
a_m[lazy_import]	7.302	0.013	7.282	7.322	0.0	0.0	22062.0	17294.0	1.0
a_m[packages]	7.32	0.013	7.299	7.34	0.0	0.0	22457.0	17790.0	1.0
a_m[journaling]	7.333	0.013	7.312	7.353	0.0	0.0	21327.0	16176.0	1.0
a_m[signals]	7.337	0.013	7.317	7.357	0.0	0.0	22265.0	16206.0	1.0
a_m[decorators]	7.429	0.013	7.409	7.45	0.0	0.0	22397.0	16684.0	1.0
a_m[h_base]	7.437	0.013	7.416	7.457	0.0	0.0	22658.0	17174.0	1.0
a_m[py_helpers]	7.597	0.013	7.577	7.617	0.0	0.0	23114.0	18014.0	1.0
a_m[headers]	7.65	0.013	7.63	7.671	0.0	0.0	23348.0	18318.0	1.0
a_m[s_helpers]	7.676	0.013	7.656	7.696	0.0	0.0	21027.0	17612.0	1.0
a_m[validation]	7.703	0.013	7.683	7.723	0.0	0.0	23710.0	17743.0	1.0
a_m[pgzero_frontend]	7.876	0.013	7.856	7.896	0.0	0.0	22589.0	18206.0	1.0
a_m[python]	8.06	0.013	8.04	8.081	0.0	0.0	23342.0	18653.0	1.0
a_m[positional_validation]	8.173	0.013	8.152	8.193	0.0	0.0	22353.0	18929.0	1.0

Table G.2: A summary table for the parameter effect (β_p) posterior distributions for the single-parameter overhead model (see Appendix B for full names of parameters).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b_p[TestInsertionProb]	-0.241	0.011	-0.259	-0.224	0.0	0.0	22030.0	18128.0	1.0
b_p[Pop]	-0.216	0.011	-0.233	-0.198	0.0	0.0	21824.0	18524.0	1.0
b_p[TestInsertProb]	-0.075	0.011	-0.092	-0.058	0.0	0.0	21483.0	17126.0	1.0
b_p[TestChangeProb]	-0.074	0.011	-0.092	-0.058	0.0	0.0	22367.0	16757.0	1.0
b_p[StatemInsertProb]	-0.071	0.011	-0.087	-0.053	0.0	0.0	21943.0	17611.0	1.0
b_p[Elite]	-0.07	0.011	-0.086	-0.052	0.0	0.0	23760.0	17331.0	1.0
b_p[Crossover]	-0.069	0.011	-0.086	-0.052	0.0	0.0	23511.0	18658.0	1.0
b_p[TestDeleteProb]	-0.065	0.011	-0.082	-0.048	0.0	0.0	22249.0	17005.0	1.0
b_p[ChangeParamProb]	-0.057	0.011	-0.075	-0.04	0.0	0.0	23562.0	16917.0	1.0
b_p[RandPert]	-0.053	0.011	-0.07	-0.035	0.0	0.0	23022.0	17155.0	1.0
b_p[TourSize]	-0.053	0.011	-0.07	-0.036	0.0	0.0	22951.0	17961.0	1.0
b_p[ChromLen]	-0.052	0.011	-0.07	-0.036	0.0	0.0	22932.0	17590.0	1.0

Table G.3: A summary table containing 40 interaction effect (γ_{mp}) posterior distributions for the single-parameter overhead model. These 40 are a subset of the total 288 distributions and include the 20 with the smallest mean and the 20 with the largest mean. See Appendix B for full names of modules and parameters.

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
g_mp[positional_validation x TestInsertionProb]	-0.034	0.021	-0.067	-0.001	0.0	0.0	32378.0	14159.0	1.0
g_mp[python x TestInsertionProb]	-0.034	0.021	-0.066	0.0	0.0	0.0	29527.0	13739.0	1.0
g_mp[namedtupleutils x Elite]	-0.033	0.021	-0.068	-0.002	0.0	0.0	32056.0	14225.0	1.0
g_mp[s_helpers x Crossover]	-0.031	0.021	-0.064	0.002	0.0	0.0	29970.0	14383.0	1.0
g_mp[namedtupleutils x TourSize]	-0.031	0.021	-0.064	0.002	0.0	0.0	33985.0	14953.0	1.0
g_mp[decorators x Pop]	-0.03	0.021	-0.065	0.002	0.0	0.0	33054.0	13605.0	1.0
g_mp[python x Crossover]	-0.029	0.021	-0.062	0.004	0.0	0.0	30049.0	14704.0	1.0
g_mp[config x TourSize]	-0.027	0.02	-0.059	0.005	0.0	0.0	36215.0	14802.0	1.0
g_mp[return_from_generator x ChromLen]	-0.027	0.02	-0.058	0.007	0.0	0.0	35838.0	14611.0	1.0
g_mp[py_helpers x Pop]	-0.026	0.02	-0.058	0.007	0.0	0.0	36058.0	15025.0	1.0
g_mp[s_helpers x Elite]	-0.026	0.021	-0.058	0.007	0.0	0.0	33953.0	13914.0	1.0
g_mp[s_helpers x TestDeleteProb]	-0.025	0.02	-0.057	0.007	0.0	0.0	35630.0	14719.0	1.0
g_mp[positional_validation x Pop]	-0.025	0.02	-0.057	0.008	0.0	0.0	32999.0	15478.0	1.0
g_mp[da x TestChangeProb]	-0.023	0.02	-0.055	0.009	0.0	0.0	33006.0	16328.0	1.0
g_mp[maybe x Crossover]	-0.023	0.02	-0.055	0.009	0.0	0.0	34930.0	14592.0	1.0
g_mp[validation x Elite]	-0.023	0.02	-0.054	0.011	0.0	0.0	33405.0	14458.0	1.0
g_mp[packages x Pop]	-0.022	0.02	-0.055	0.01	0.0	0.0	35093.0	14322.0	1.0
g_mp[da x ChangeParamProb]	-0.021	0.02	-0.052	0.013	0.0	0.0	40662.0	12894.0	1.0
g_mp[yield_from x TestChangeProb]	-0.021	0.02	-0.053	0.011	0.0	0.0	40504.0	13956.0	1.0
g_mp[packages x TestInsertionProb]	-0.02	0.02	-0.053	0.011	0.0	0.0	38705.0	14353.0	1.0
...	...	...	...	...	...	...	...	...	...
g_mp[py_helpers x TestChangeProb]	0.019	0.02	-0.013	0.05	0.0	0.0	44805.0	14646.0	1.0
g_mp[namedtupleutils x StatemInsertProb]	0.02	0.02	-0.013	0.051	0.0	0.0	34256.0	14549.0	1.0
g_mp[maybe x TestChangeProb]	0.02	0.02	-0.012	0.051	0.0	0.0	34676.0	14874.0	1.0
g_mp[validation x TourSize]	0.021	0.02	-0.011	0.054	0.0	0.0	33647.0	14897.0	1.0
g_mp[config x Elite]	0.021	0.021	-0.011	0.054	0.0	0.0	38507.0	15032.0	1.0
g_mp[config x Pop]	0.021	0.02	-0.011	0.054	0.0	0.0	40220.0	14351.0	1.0
g_mp[pgzero_frontend x Pop]	0.023	0.02	-0.009	0.054	0.0	0.0	39913.0	14661.0	1.0
g_mp[signals x Pop]	0.023	0.02	-0.009	0.054	0.0	0.0	37888.0	14244.0	1.0
g_mp[da x Pop]	0.023	0.02	-0.009	0.055	0.0	0.0	36689.0	14280.0	1.0
g_mp[immutable_list x Pop]	0.026	0.02	-0.006	0.059	0.0	0.0	36094.0	14335.0	1.0
g_mp[maybe x TestDeleteProb]	0.026	0.02	-0.006	0.059	0.0	0.0	37265.0	14474.0	1.0
g_mp[timer x RandPert]	0.027	0.02	-0.004	0.06	0.0	0.0	35738.0	13922.0	1.0
g_mp[maybe x TestInsertionProb]	0.027	0.02	-0.006	0.059	0.0	0.0	31476.0	14033.0	1.0
g_mp[maybe x Pop]	0.03	0.021	-0.003	0.063	0.0	0.0	31405.0	15316.0	1.0
g_mp[config x TestInsertionProb]	0.031	0.021	-0.002	0.064	0.0	0.0	31446.0	14594.0	1.0
g_mp[py_helpers x Crossover]	0.033	0.02	0.001	0.065	0.0	0.0	30651.0	15147.0	1.0
g_mp[s_helpers x ChromLen]	0.036	0.02	0.004	0.069	0.0	0.0	30412.0	14819.0	1.0
g_mp[s_helpers x TestInsertionProb]	0.039	0.021	0.006	0.072	0.0	0.0	30369.0	14067.0	1.0
g_mp[validation x Crossover]	0.039	0.021	0.006	0.071	0.0	0.0	29603.0	14541.0	1.0
g_mp[da x TourSize]	0.04	0.021	0.005	0.071	0.0	0.0	28623.0	15138.0	1.0

Table G.4: A summary table for the module intercept (α_m) posterior distributions for the multi-parameter overhead model (see Appendix B for full names of modules).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
a_m[da]	6.688	0.014	6.665	6.708	0.0	0.0	1781.0	3683.0	1.0
a_m[config]	6.697	0.014	6.675	6.718	0.0	0.0	1701.0	4035.0	1.0
a_m[maybe]	6.999	0.014	6.977	7.021	0.0	0.0	1713.0	3432.0	1.0
a_m[cmd]	7.082	0.013	7.061	7.104	0.0	0.0	1679.0	3234.0	1.0
a_m[immutable_list]	7.115	0.013	7.093	7.136	0.0	0.0	1698.0	3977.0	1.0
a_m[timer]	7.124	0.014	7.101	7.144	0.0	0.0	1688.0	4244.0	1.0
a_m[namedtupleutils]	7.193	0.014	7.171	7.214	0.0	0.0	1722.0	3797.0	1.0
a_m[dict_unpacking]	7.219	0.013	7.198	7.241	0.0	0.0	1753.0	3831.0	1.0
a_m[packages]	7.242	0.013	7.22	7.263	0.0	0.0	1703.0	3700.0	1.0
a_m[lazy_import]	7.262	0.014	7.24	7.283	0.0	0.0	1728.0	3611.0	1.0
a_m[signals]	7.302	0.013	7.281	7.324	0.0	0.0	1669.0	3547.0	1.0
a_m[return_from_generator]	7.303	0.013	7.281	7.324	0.0	0.0	1662.0	3953.0	1.0
a_m[journaling]	7.314	0.014	7.292	7.335	0.0	0.0	1673.0	3313.0	1.0
a_m[py_base]	7.321	0.013	7.299	7.342	0.0	0.0	1675.0	3752.0	1.0
a_m[decorators]	7.401	0.014	7.379	7.423	0.0	0.0	1697.0	3388.0	1.0
a_m[yield_from]	7.489	0.014	7.468	7.511	0.0	0.0	1684.0	3585.0	1.0
a_m[py_helpers]	7.574	0.014	7.553	7.596	0.0	0.0	1682.0	3415.0	1.0
a_m[s_helpers]	7.624	0.014	7.603	7.646	0.0	0.0	1754.0	3756.0	1.0
a_m[validation]	7.624	0.013	7.602	7.645	0.0	0.0	1706.0	3781.0	1.0
a_m[h_base]	7.683	0.014	7.662	7.705	0.0	0.0	1738.0	3908.0	1.0
a_m[pgzero_frontend]	7.815	0.013	7.794	7.837	0.0	0.0	1695.0	3942.0	1.0
a_m[headers]	7.884	0.013	7.863	7.906	0.0	0.0	1658.0	3546.0	1.0
a_m[python]	8.001	0.013	7.979	8.022	0.0	0.0	1714.0	4137.0	1.0
a_m[positional_validation]	8.059	0.013	8.038	8.081	0.0	0.0	1734.0	3486.0	1.0

Table G.5: A summary table for the parameter effect (β_p) posterior distributions for the multi-parameter overhead model (see Appendix B for full names of parameters).

	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
b_p[Pop|TestInsertionProb]	-0.344	0.017	-0.371	-0.317	0.0	0.0	2442.0	5456.0	1.0
b_p[TestChangeProb|TestInsertionProb]	-0.218	0.017	-0.245	-0.19	0.0	0.0	2562.0	5112.0	1.0
b_p[TestInsertProb|TestInsertionProb]	-0.211	0.017	-0.238	-0.184	0.0	0.0	2470.0	5299.0	1.0
b_p[ChangeParamProb|TestInsertionProb]	-0.207	0.017	-0.234	-0.18	0.0	0.0	2633.0	5827.0	1.0
b_p[ChromLen|TestInsertionProb]	-0.204	0.017	-0.231	-0.176	0.0	0.0	2627.0	5930.0	1.0
b_p[RandPert|TestInsertionProb]	-0.198	0.017	-0.225	-0.172	0.0	0.0	2543.0	6367.0	1.0
b_p[Crossover|TestInsertionProb]	-0.197	0.017	-0.224	-0.17	0.0	0.0	2692.0	5177.0	1.0
b_p[Elite|TestInsertionProb]	-0.196	0.017	-0.223	-0.169	0.0	0.0	2575.0	5581.0	1.0
b_p[TestDeleteProb|TestInsertionProb]	-0.192	0.017	-0.218	-0.164	0.0	0.0	2567.0	6166.0	1.0
b_p[StatemInsertProb|TestInsertionProb]	-0.191	0.017	-0.219	-0.164	0.0	0.0	2621.0	5842.0	1.0
b_p[TestInsertionProb|TourSize]	-0.187	0.017	-0.216	-0.161	0.0	0.0	2609.0	5743.0	1.0
b_p[Pop|TestInsertProb]	-0.185	0.017	-0.212	-0.158	0.0	0.0	2613.0	5825.0	1.0
b_p[ChromLen|Pop]	-0.176	0.017	-0.203	-0.149	0.0	0.0	2486.0	5273.0	1.0
b_p[Pop|StatemInsertProb]	-0.176	0.017	-0.203	-0.149	0.0	0.0	2483.0	5957.0	1.0
b_p[Pop|TestDeleteProb]	-0.173	0.017	-0.201	-0.146	0.0	0.0	2559.0	6162.0	1.0
b_p[Crossover|Pop]	-0.164	0.017	-0.191	-0.137	0.0	0.0	2573.0	5916.0	1.0
b_p[ChangeParamProb|Pop]	-0.164	0.017	-0.191	-0.137	0.0	0.0	2534.0	5212.0	1.0
b_p[Elite|Pop]	-0.162	0.017	-0.188	-0.135	0.0	0.0	2606.0	6126.0	1.0
b_p[Pop|TourSize]	-0.158	0.017	-0.184	-0.131	0.0	0.0	2499.0	6141.0	1.0
b_p[Pop|TestChangeProb]	-0.158	0.017	-0.185	-0.132	0.0	0.0	2659.0	6437.0	1.0
b_p[Pop|RandPert]	-0.143	0.017	-0.171	-0.118	0.0	0.0	2576.0	6202.0	1.0
b_p[StatemInsertProb|TestChangeProb]	-0.039	0.017	-0.065	-0.012	0.0	0.0	2409.0	5469.0	1.0
b_p[Crossover|TestChangeProb]	-0.038	0.017	-0.065	-0.011	0.0	0.0	2555.0	5396.0	1.0
b_p[Crossover|TestDeleteProb]	-0.036	0.017	-0.062	-0.009	0.0	0.0	2585.0	5912.0	1.0
b_p[StatemInsertProb|TestInsertProb]	-0.034	0.017	-0.062	-0.008	0.0	0.0	2444.0	5875.0	1.0
b_p[RandPert|TestDeleteProb]	-0.033	0.017	-0.06	-0.006	0.0	0.0	2526.0	6165.0	1.0
b_p[TestChangeProb|TestInsertProb]	-0.032	0.017	-0.06	-0.006	0.0	0.0	2559.0	5568.0	1.0
b_p[StatemInsertProb|TestDeleteProb]	-0.031	0.017	-0.058	-0.004	0.0	0.0	2647.0	5777.0	1.0
b_p[Elite|StatemInsertProb]	-0.03	0.017	-0.057	-0.004	0.0	0.0	2554.0	5640.0	1.0
b_p[TestDeleteProb|TestInsertProb]	-0.028	0.017	-0.054	-0.001	0.0	0.0	2538.0	5506.0	1.0
b_p[TestChangeProb|TestDeleteProb]	-0.028	0.017	-0.054	-0.001	0.0	0.0	2548.0	5827.0	1.0
b_p[Crossover|TestInsertProb]	-0.025	0.017	-0.053	0.0	0.0	0.0	2454.0	5986.0	1.0
b_p[ChromLen|TestDeleteProb]	-0.025	0.017	-0.052	0.002	0.0	0.0	2568.0	6549.0	1.0
b_p[ChromLen|StatemInsertProb]	-0.023	0.017	-0.05	0.004	0.0	0.0	2587.0	5535.0	1.0
b_p[ChangeParamProb|TestChangeProb]	-0.022	0.017	-0.049	0.004	0.0	0.0	2472.0	5800.0	1.0
b_p[Elite|RandPert]	-0.022	0.017	-0.049	0.004	0.0	0.0	2526.0	5871.0	1.0
b_p[RandPert|TestChangeProb]	-0.021	0.017	-0.047	0.006	0.0	0.0	2584.0	5381.0	1.0
b_p[Elite|TestInsertProb]	-0.02	0.017	-0.047	0.007	0.0	0.0	2458.0	5785.0	1.0
b_p[ChromLen|Crossover]	-0.018	0.017	-0.045	0.008	0.0	0.0	2444.0	6105.0	1.0
b_p[Crossover|RandPert]	-0.018	0.017	-0.044	0.009	0.0	0.0	2551.0	5961.0	1.0
b_p[Crossover|Elite]	-0.018	0.017	-0.045	0.008	0.0	0.0	2458.0	5694.0	1.0
b_p[RandPert|StatemInsertProb]	-0.017	0.017	-0.045	0.009	0.0	0.0	2525.0	5321.0	1.0
b_p[TestInsertProb|TourSize]	-0.014	0.017	-0.041	0.012	0.0	0.0	2403.0	5339.0	1.0
b_p[ChromLen|TestChangeProb]	-0.011	0.017	-0.038	0.016	0.0	0.0	2664.0	7070.0	1.0
b_p[ChangeParamProb|TestDeleteProb]	-0.011	0.017	-0.037	0.016	0.0	0.0	2487.0	5975.0	1.0
b_p[StatemInsertProb|TourSize]	-0.01	0.017	-0.036	0.017	0.0	0.0	2522.0	5698.0	1.0
b_p[ChangeParamProb|Crossover]	-0.01	0.017	-0.037	0.017	0.0	0.0	2530.0	6164.0	1.0
b_p[TestDeleteProb|TourSize]	-0.01	0.017	-0.037	0.016	0.0	0.0	2651.0	6551.0	1.0
b_p[RandPert|TestInsertProb]	-0.009	0.017	-0.036	0.017	0.0	0.0	2622.0	6063.0	1.0
b_p[Crossover|StatemInsertProb]	-0.009	0.017	-0.035	0.019	0.0	0.0	2505.0	6348.0	1.0
b_p[RandPert|TourSize]	-0.008	0.017	-0.034	0.02	0.0	0.0	2508.0	5717.0	1.0
b_p[Elite|TestDeleteProb]	-0.006	0.017	-0.032	0.021	0.0	0.0	2561.0	6059.0	1.0
b_p[Elite|TestChangeProb]	-0.006	0.017	-0.033	0.021	0.0	0.0	2540.0	5847.0	1.0
b_p[ChromLen|TestInsertProb]	-0.006	0.017	-0.031	0.021	0.0	0.0	2551.0	5177.0	1.0
b_p[ChangeParamProb|Elite]	-0.004	0.017	-0.03	0.023	0.0	0.0	2588.0	5814.0	1.0
b_p[Elite|TourSize]	-0.003	0.017	-0.03	0.023	0.0	0.0	2599.0	6409.0	1.0
b_p[ChangeParamProb|TestInsertProb]	-0.003	0.017	-0.03	0.023	0.0	0.0	2509.0	5268.0	1.0
b_p[ChangeParamProb|ChromLen]	-0.002	0.017	-0.029	0.025	0.0	0.0	2555.0	5369.0	1.0
b_p[ChromLen|TourSize]	0.0	0.017	-0.027	0.026	0.0	0.0	2556.0	5893.0	1.0
b_p[ChromLen|RandPert]	-0.0	0.017	-0.027	0.026	0.0	0.0	2519.0	5869.0	1.0
b_p[ChangeParamProb|RandPert]	0.001	0.017	-0.027	0.027	0.0	0.0	2532.0	6370.0	1.0
b_p[ChromLen|Elite]	0.002	0.017	-0.024	0.029	0.0	0.0	2515.0	5741.0	1.0
b_p[Crossover|TourSize]	0.003	0.017	-0.024	0.029	0.0	0.0	2474.0	5834.0	1.0
b_p[TestChangeProb|TourSize]	0.006	0.017	-0.021	0.033	0.0	0.0	2577.0	6435.0	1.0
b_p[ChangeParamProb|TourSize]	0.007	0.017	-0.019	0.035	0.0	0.0	2579.0	6252.0	1.0
b_p[ChangeParamProb|StatemInsertProb]	0.013	0.017	-0.013	0.04	0.0	0.0	2589.0	5893.0	1.0

Table G.6: A summary table containing 40 interaction effect (γ_{mp}) posterior distributions for the multi-parameter overhead model. These 40 are a subset of the total 1584 distributions and include the 20 with the smallest mean and the 20 with the largest mean. See Appendix B for full names of modules and parameters.

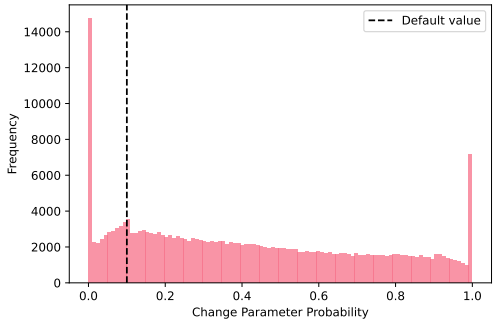
	mean	sd	hdi_5.5%	hdi_94.5%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
g_mp[config x ChangeParamProb|StatemInsertProb]	-0.008	0.359	-0.501	0.449	0.003	0.002	25527.0	15706.0	1.0
g_mp[lazy_import x ChromLen|Pop]	-0.008	0.356	-0.448	0.513	0.003	0.002	24263.0	15667.0	1.0
g_mp[signals x Elite|TourSize]	-0.007	0.356	-0.51	0.443	0.003	0.002	25257.0	16962.0	1.0
g_mp[dict_unpacking x Pop|TourSize]	-0.007	0.351	-0.485	0.451	0.003	0.002	24151.0	17386.0	1.0
g_mp[lazy_import x ChangeParamProb|TourSize]	-0.007	0.348	-0.479	0.488	0.003	0.002	23469.0	16884.0	1.0
g_mp[lazy_import x Pop|TestChangeProb]	-0.007	0.355	-0.512	0.455	0.003	0.002	24816.0	16523.0	1.0
g_mp[immutable_list x ChangeParamProb|Elite]	-0.007	0.353	-0.501	0.455	0.003	0.002	23712.0	17086.0	1.0
g_mp[positional_validation x Pop|TourSize]	-0.007	0.351	-0.486	0.449	0.003	0.002	24594.0	16103.0	1.0
g_mp[timer x Crossover|TestInsertionProb]	-0.006	0.355	-0.462	0.479	0.003	0.002	24312.0	16701.0	1.0
g_mp[dict_unpacking x StatemInsertProb|TestInsertProb]	-0.006	0.349	-0.475	0.462	0.003	0.002	24245.0	16984.0	1.0
g_mp[headers x StatemInsertProb|TestInsertProb]	-0.006	0.353	-0.471	0.49	0.003	0.002	23454.0	16659.0	1.0
g_mp[namedtupleutils x Crossover|TestInsertProb]	-0.006	0.351	-0.499	0.444	0.003	0.002	24351.0	15998.0	1.0
g_mp[immutable_list x TestDeleteProb|TestInsertionProb]	-0.006	0.355	-0.485	0.481	0.003	0.002	25008.0	16689.0	1.0
g_mp[dict_unpacking x TestChangeProb|TestDeleteProb]	-0.006	0.356	-0.512	0.46	0.003	0.002	24015.0	16640.0	1.0
g_mp[immutable_list x Pop|TestChangeProb]	-0.006	0.354	-0.477	0.483	0.003	0.002	22581.0	16463.0	1.0
g_mp[h_base x TestDeleteProb|TourSize]	-0.006	0.357	-0.482	0.488	0.003	0.002	22832.0	16644.0	1.0
g_mp[namedtupleutils x Elite|StatemInsertProb]	-0.006	0.348	-0.445	0.486	0.003	0.002	24100.0	16201.0	1.0
g_mp[return_from_generator x TestDeleteProb|TourSize]	-0.006	0.352	-0.475	0.489	0.003	0.002	24498.0	16683.0	1.0
g_mp[lazy_import x Crossover|RandPert]	-0.006	0.356	-0.507	0.459	0.003	0.002	23791.0	16474.0	1.0
g_mp[headers x Pop|TestInsertProb]	-0.006	0.346	-0.451	0.515	0.003	0.002	23022.0	17317.0	1.0
...	...	...	...	...	...	...	...	...	...
g_mp[immutable_list x StatemInsertProb|TourSize]	0.006	0.347	-0.491	0.464	0.003	0.002	23688.0	16085.0	1.0
g_mp[maybe x ChangeParamProb|Crossover]	0.006	0.35	-0.44	0.496	0.003	0.002	23372.0	17214.0	1.0
g_mp[positional_validation x TestInsertionProb|TourSize]	0.006	0.352	-0.489	0.456	0.003	0.002	24525.0	15928.0	1.0
g_mp[return_from_generator x ChangeParamProb|Crossover]	0.006	0.347	-0.464	0.484	0.003	0.002	24887.0	16662.0	1.0
g_mp[headers x Pop|TestDeleteProb]	0.006	0.35	-0.483	0.48	0.003	0.002	26373.0	15515.0	1.0
g_mp[cmd x ChromLen|Crossover]	0.006	0.354	-0.457	0.501	0.003	0.002	24131.0	16481.0	1.0
g_mp[positional_validation x Crossover|TestDeleteProb]	0.006	0.353	-0.502	0.446	0.003	0.002	23228.0	16118.0	1.0
g_mp[timer x ChangeParamProb|TourSize]	0.006	0.349	-0.48	0.474	0.003	0.002	23144.0	16883.0	1.0
g_mp[pgzero_frontend x Pop|TestDeleteProb]	0.006	0.353	-0.53	0.432	0.003	0.002	24546.0	16871.0	1.0
g_mp[py_base x Pop|RandPert]	0.006	0.354	-0.473	0.48	0.003	0.002	24631.0	16361.0	1.0
g_mp[da x ChangeParamProb|TestDeleteProb]	0.006	0.357	-0.478	0.486	0.003	0.002	23970.0	16443.0	1.0
g_mp[config x ChangeParamProb|Pop]	0.007	0.355	-0.446	0.51	0.003	0.002	23615.0	17749.0	1.0
g_mp[s_helpers x ChangeParamProb|TestChangeProb]	0.007	0.357	-0.484	0.467	0.003	0.002	22732.0	16681.0	1.0
g_mp[da x Pop|TourSize]	0.007	0.347	-0.451	0.488	0.003	0.002	23186.0	15756.0	1.0
g_mp[signals x Crossover|TestDeleteProb]	0.007	0.343	-0.476	0.471	0.003	0.002	22736.0	17672.0	1.0
g_mp[immutable_list x Pop|TourSize]	0.007	0.357	-0.461	0.494	0.003	0.002	24059.0	16399.0	1.0
g_mp[yield_from x ChromLen|TestInsertionProb]	0.007	0.351	-0.498	0.468	0.003	0.002	22962.0	16737.0	1.0
g_mp[timer x TestChangeProb|TestDeleteProb]	0.008	0.349	-0.486	0.461	0.003	0.002	25631.0	17025.0	1.0
g_mp[return_from_generator x ChangeParamProb|TestDeleteProb]	0.008	0.345	-0.442	0.512	0.003	0.002	23961.0	17091.0	1.0
g_mp[journaling x Elite|Pop]	0.013	0.345	-0.429	0.526	0.003	0.002	25674.0	16953.0	1.0

H

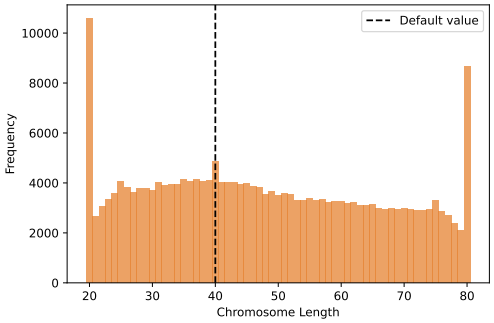
Parameter Assignment

Figure H.1: Histograms showing the parameter values chosen during the single-parameter experiment test generation process (part 1).

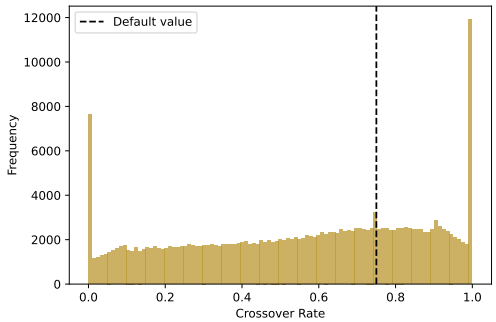
(a) Change Parameter Probability.



(b) Chromosome Length.



(c) Crossover Rate.



(d) Elite.

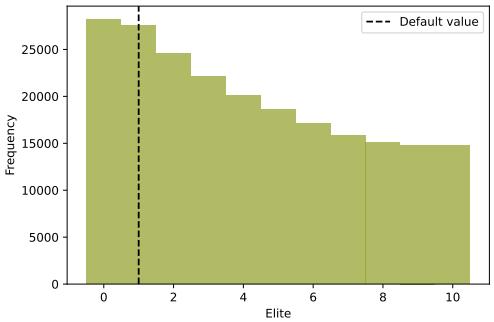
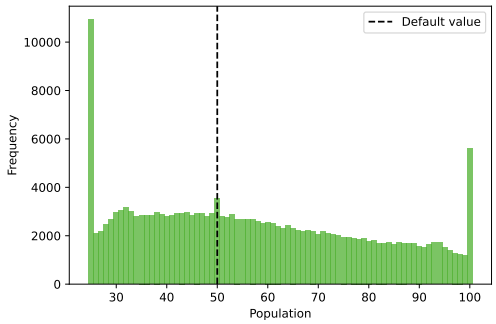
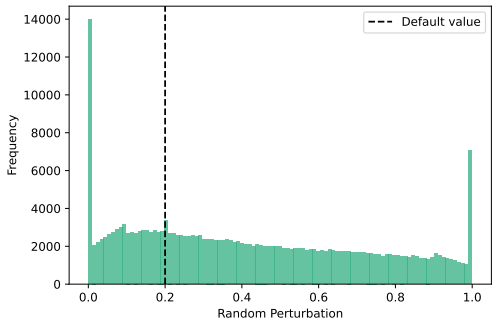


Figure H.2: Histograms showing the parameter values chosen during the single-parameter experiment test generation process (part 2).

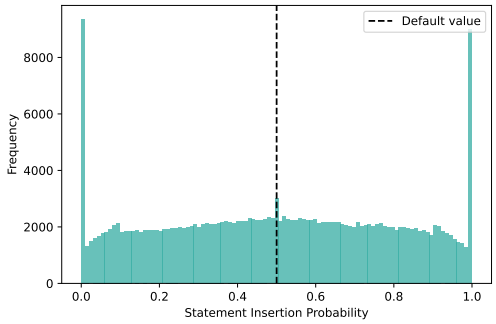
(a) Population.



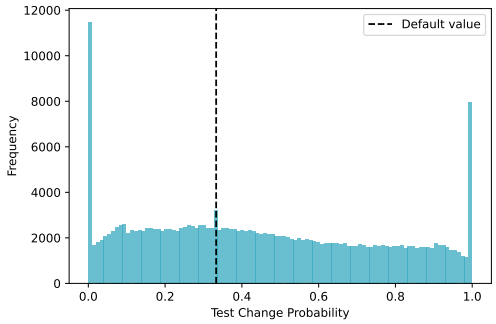
(b) Random Perturbation.



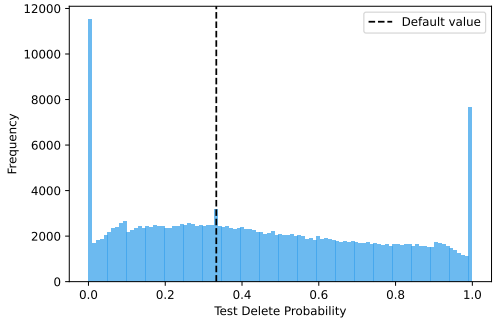
(c) Statement Insertion Probability



(d) Test Change Probability.



(e) Test Delete Probability.



(f) Test Insert Probability.

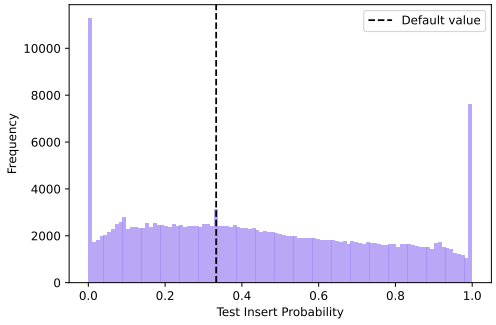
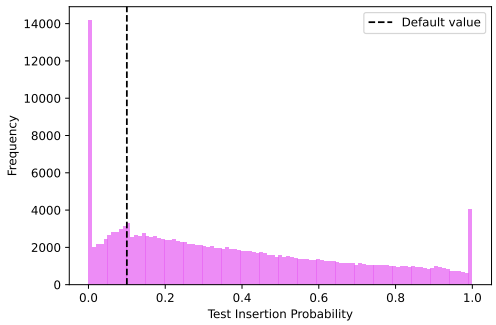


Figure H.3: Histograms showing the parameter values chosen during the single-parameter experiment test generation process (part 3).

(a) Test Insertion Probability.



(b) Tournament Size.