FuzzDiff: A Program Equivalence Checker based on feedback-directed fuzz testing and semantic analysis

Akash Purushottam Patil

A Dissertation

Presented to the University of Dublin, Trinity College

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science (Intelligent Systems)

Supervisor: Prof. Vasileios Koutavas

August 2022

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Akash Purushottam Patil

August 19, 2022

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Akash Purushottam Patil

August 19, 2022

FuzzDiff: A Program Equivalence Checker based on feedback-directed fuzz testing and semantic analysis

Akash Purushottam Patil, Master of Science in Computer Science

University of Dublin, Trinity College, 2022

Supervisor: Prof. Vasileios Koutavas

Property-based fuzz testing with coverage guidance mechanism has proven to be very effective in finding bugs in high-level programs. This dissertation attempts to examine the possibility of proving functional equivalency of two Java programs using such feedback-directed fuzz testing techniques and semantic analysis, facilitated by a framework called JQF. An equivalence checker called FuzzDiff, in the form of a Maven project, is developed to compare the functional behaviour of two simple java programs having method with same signature and return type. JQF, which is based on JUnit QuickCheck, is used as the engine for generating feedback-directed random inputs being fed into these programs. A number of JUnit assertions determine the functional equivalence of the two input programs over the two stages. A Generic Generator class is also designed and implemented for generating random instances of custom Java objects. Moreover, the tool also applies additional tests for semantic analysis of the two input programs, where method invocations are traced and compared to comply with certain assertions. Finally, the tool is evaluated on a number of test programs in benchmarks used by tools like Hobbit and ARDiff, and the results are analyzed. Results show that FuzzDiff can effectively identify inequivalency but can be inconsistent in verifying equivalency of two Java programs. During evaluation, FuzzDiff also finds an incorrectly classified equivalent pair of program in ARDiff benchmark.

Acknowledgments

I would like to firstly acknowledge and thank my supervisor, Prof. Vasileios Koutavas for his constant guidance and providing his valuable knowledge in the field towards the development of the project. His role was fundamental to the completion of this dissertation and I learned a lot through the many interactions we had.

Secondly, I would like to thank my parents and my sister for their persistent support and words of encouragement throughout the course of the project.

I would also like to acknowledge Prof. Andrew Butterfield for sharing his valuable feedback during the demo.

Akash Purushottam Patil

University of Dublin, Trinity College
August 2022

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background and Motivation

Program Equivalence checking, a part of a discipline of formal verification in Computer Science, is the scientific problem of formally proving that two methods or programs exhibit the same behaviour functionally. It is a long-standing and key research problem in the world of formal verification with several techniques and methodologies proposed to address it. Program equivalence checking can be an undecidable problem depending on the complexity of programs, and hence it is fairly difficult to devise an out-and-out equivalence checking procedure for a pair of programs [27]. It is further challenging to create a tool that would serve as an implementation to the checking procedure.

Applications of Program equivalence include verification of correctness of compilers [15], Program synthesis [11] and correctness verification of code refactoring [32]. This study focuses on the ultimate application i.e detecting program equivalence or in-equivalence for verifying refactored programs. Code Refactoring is the process of re-structuring part of a code or program without affecting its existing behaviour or functionality with respect to other components of the program. As per a survey in 2019 [9], software engineers spend 22% of their time in code refactoring tasks, which also includes validating if the behaviour of the code is unchanged. In practice, the validation is done using regression testing which involves manually creating a test suite that covers all possible scenarios and then executing it to see if any failures occur. This process can be very time-consuming and resource-dependent. If we can provide a utility that can be easily integrated into a software development pipeline, it can improve the overall process of detecting defects in code refactoring tasks. This forms the core motivation of the study: attempting to simplify program verification post code refactoring.

Program equivalence checkers are tools that compare the behaviour of two programs based on features related to a program. Some make the judgement based on the syntax, while others compare the functional behaviour and semantics. There are some existing tools like SymDiff [22] and Hobbit [21] which try to decide equivalence using algorithms based on differential verification and symbolic execution respectively. However, little has been done in the field to address the same problem using property-based testing techniques, specifically random sampling or fuzz testing. Moreover, the majority of the tools available online are based on languages like OCaml, C++ and C. Very few are based on object-oriented languages like Java and Scala. This paves the motivation for designing a Java-based program equivalence checker, which would also contribute to the research community working on Java verification tools.

When researching program equivalence checkers, it was observed that very few utilize testing-based techniques for verification. Junit is a testing framework in Java that allows developers to write and run unit test cases to test the behavior of methods inside classes they have written. In industrial applications, Junit is heavily used and relied on for quality assurance. These test cases execute user-provided mock objects as input to these methods and verify if the output is expected as actual. A set of such test cases can cover a range of scenarios (both positive and negative) where the program may exhibit different behaviour, and hence is referred to as test coverage of a class. There is a possibility that such a framework can be used as instrumentation for testing the behaviour of a refactored method and then comparing its results with the original. But this would require the user to provide manually curated inputs to both these methods, which can be time-consuming and inefficient. This is where fuzz testing can come into play.

Fuzz testing is a testing technique that involves providing randomly generated inputs into a computer program. It is a very popular technique, mainly used in automated program monitoring in large-scale systems. JUnit QuickCheck [4] was found to be one of the popular libraries that use the concept of fuzzing in JUnit testing by making use of generators to feed randomly generated inputs into parameterized test methods. This tool is inspired by the QuickCheck tool originally developed for random testing of Haskell programs [16]. JQF [30], a tool built on top of JUnit QuickCheck, uses an algorithm called Zest [31] that tunes these generators to produce "semantically valid inputs" and has proven to maximize code coverage in random testing.

Hence, through this study, we attempt to design a program equivalence checker which

would apply fuzz testing using JQF to verify and compare the functional behaviour of a pair of Java programs. The study mainly focuses on prototyping such a tool and testing its effectiveness in program verification following real-world refactoring tasks.

The tool designed is Maven based and uses a suite of dependencies and plugins for providing a command line interface for the user to input the two programs and method name to be fuzzed. The tool can be found at `https://github.com/akashpatil7/FuzzDiff`

We hope that researchers and developers find it convenient and productive to implement new verification ideas using this flexible tool.

1.2 Research Question and Aim

This section outlines the research questions that this study is addressing and also proposes hypotheses in terms of the potential of this tool. As discussed in the previous section, the research aim of this study mainly revolved around solving the problem of program equivalence. But the research questions we are trying to answer are:

Question 1: Can property-based testing techniques involving random sampling or fuzzing be used to identify program equivalence of two Java methods?
Question 2: If yes, can a viable solution in the form of a tool be implemented to leverage this methodology?
Question 3: To what level would its usage be effective in real-world scenarios?

With several techniques being devised to address program equivalence, it will be interesting to see if fuzz testing proves to be one of the solutions that can be not just effective but useful to compare two Java programs.

The hypothesis of this study is that : Though it is viable to develop such a tool but it may not be the most efficient technique given the number of test executions on the target method. While dealing with highly complex programs, the technique might not be very viable as well, since there will be a range of scenarios to be considered in the overall workflow of the tool. Hence, it is important to define the scope of the tool to support elementary Java programs consisting of plain algorithms and using plain java objects and data types like primitives and arrays.

The research aim is to create a prototype that implements a novel methodology of com-

paring two programs for equivalence and lays the foundation for further integration in the future.

1.3 Report Structure

This section gives a brief overview of how the study work is organized into different chapters:

Chapter 2 discusses the state-of-the-art techniques in formal verification for program equivalence. Few of the techniques also come in the form of a tool supporting a wide range of languages. It will be essential to focus on tools built on JVM-based languages like Java. It also covers the literature review of some important papers which expand on the problem statement and provide solutions widely recognized in the research community.

Chapter 3 describes the problem statement in detail and covers important concepts and definitions related to functional equivalence and property-based testing. This includes in-depth analysis of tools like JQF and Junit QuickCheck. It also describes the design and implementation of a Generic Generator which addresses the shortcomings of QuickCheck-based generators.

Chapter 4 covers the groundwork of the tool from design to the implementation phase, expanding on different components of the system, functional requirements, decision-making criteria and the limitations of the model. It also covers few examples of equivalent and non-equivalent programs in context with the tool. it also shows the usage of the proposed tool with step-by-step details of the user's interactions with the system, outlining the pre-requisites for the usage and the criteria for valid and invalid inputs.

Chapter 5 shows the evaluation of the tool and describes the data sets and methodology used for completing the evaluation of the tool. This also includes the different experiments performed to evaluate the viability and effectiveness of the tool. It also outlines the results of the experiments, analyses the outcomes and expands on the observations drawn from the results.

Chapter 6 discusses the possible explanations of the answers to the research questions and the hypotheses. It concludes the study and covers the future scope of the tool in detail, providing recommended directions for future research and improvements.

Chapter 2

Literature review

This chapter of the dissertation provides a thorough background of the problem statement and takes a deep dive into the different state-of-the-art techniques and tools for determining program equivalence. It also discusses in detail the methodology used in the backend of FuzzDiff and how it enables the tool to check for equivalence.

2.1 Tools

A wide range of techniques have been introduced in past few years, each providing a different perspective to the problem of program equivalence. It has been widely researched over the years and has a long history in verification and model checking applications.

One of those verification techniques being discussed in [10] specifically focuses on code refactoring of sequential programs to their parallel counterpart. It attempts to tackle the challenge of verifying program parallelization using symbolic execution, taking into account that the conversion of sequential programs to its parallel counterpart may lead to synchronization issues. In real-world scenarios, producing error-free parallel code may lead to changes in behaviour or functionality of the sequential program. Using symbolic interpretation and mathematical modelling, the two programs i.e sequential and parallel version can be proven functionally equivalent more effectively as compared to testing based techniques. The model checker uses control flow graph (CFG) built using a symbolic interpreter for C/C++ called ExpliSAT. For each change in the refactoring, it is verified if the function produces the same output for every input. The result is analyzed and the outcome is determined based on any violations or interleaving of threads.

Other software verification tools like CPAChecker [14] provides technique to implement

configurable program analysis (CPA) for model checking.

2.1.1 PatEC

PatEC [17] is another such tool which checks for functional equivalence between serial and parallelized programs. However, this tool identifies the differences based on parallel design patterns of OpenMP parallelizations. OpenMP is an application programming interface which allows shared memory multiprocessing programming in C, C++ and Fortran.

However, in context to the research question, the tools that can be considered state-of-the-art are the ones which takes into account two programs irrespective of their nature. Some of these tools are: PEQTest, ARDiff, SymDiff, Hobbit. Each of these tools take a different approach in determining program equivalence, and hence these form as the reference point for the design and implementation of FuzzDiff in the dissertation.

2.1.2 PEQTest

PEQTest [19] is again a solution proposed to detect change in functional behaviour of two programs after code alternation or refactoring. PEQTest is testing based technique to check for equivalence inspired by the techniques discussed in this section like SymDiff [22] and PEQCheck [18] which convert the two programs into a test program. It uses existing test generators for functional equivalence checking but executions are done in a more efficient manner. The main motive of this tool is to reduce the computation in testing redundant code of the test program which remains the same in all executions. To achieve the same, the refactored code segment in the test program is detected and then specific lines of code is injected into the program which will store, restore and compare the modified variables in the refactored segment. These segments of code are essentially checkpoints to preserve the value of variables which would eventually help in determining the equivalence. If no execution leads to invalid scenario in the comparison of variables, the two programs are considered equivalent. The evaluation of this tool show that it outperforms PEQCheck, on which it's core is mainly based on. The benchmark used for evaluation were the tasks manually selected from the DataRaceBench [25] suite containing 26 equivalent and 106 in-equivalent tasks.

2.1.3 SymDiff

SymDiff [22] or Symbolic Diff is another state-of-the-art equivalence checker which is based on symbolic execution of test programs written in C. It complements to tools

such as WinDiff and GNUDiff which are based on "syntatic" differentiation. SymDiff utilizes an intermediate verification language known as Boogie [13], and hence it requires programs of any language to be first translated to Boogie. Boogie has its own set of specifications for control-flow and assignment operations. SymDiff's user interface takes two Boogie programs (which must be free of loops) and a configuration file which contains all the required mapping of procedures, global variables between the two programs. It then generates new set of Boogie procedures based on the mappings provided by the user and checks for partial equivalence for all pairs. All these resulting procedures are then fed into a modular verifier which checks for assertions and returns the combined output. In case of any assertion failure, SymDiff reports a set of intraprocedural paths to user to identify the incorrectness in the behaviour of the two programs. With regards to the proposed tool in this dissertation, SymDiff helps define the assertions required as part of the verification tasks. However, we go beyond just comparing the final output of the two procedures.

2.1.4 ARDiff

The creators of ARDiff [12] argue that tools or techniques which use symbolic execution are cost heavy and not very scalable given the complex nature of real world programs. Hence, ARDiff aims to improve the scalability of such tools by abstracting the common code in the two input methods. These segments of common code can occupy majority of the method body, hence pruning it can help the model checking be more efficient without impacting its effectiveness. For this refinement, ARDiff iteratively applies a set of three heuristics to decide whether the two versions of method are equivalent or not. At every iteration, all the syntactically similar statements are pruned and then symbolic execution is performed. The first two heuristics narrows down the method to statements relevant in proving or disproving equivalence, while the third heuristic performs code-level analysis to remove ambiguity from the code. With this approach, ARDiff reduces the complexity of the input program and refines the decision-making process.

The creators of ARDiff have also curated a new benchmark [5] for equivalence checking, consisting of real-life examples of refactoring with varying complexity. Since the examples are Java-based, this benchmarks makes the most ideal choice for evaluation of the proposed tool in this dissertation. Hence, we use few equivalent and non-equivalent code samples from this benchmark for evaluation and compare the results with that of ARDiff.

2.1.5 Hobbit

Another state-of-the-art verification tool for contextual equivalence checking is Hobbit [21], which implements bounded model checking to prove equivalences for higher order programs. These programs are based on a subset of a language called OCaml [1], which allows object-oriented style of programming in a simplified manner. The bounded model checker implements "symbolic environment bisimulations" to explore potentially infinite computation trees and then finitize them for equivalence check. The dataset [26] on which Hobbit is evaluated is relevant to this dissertation as the OCaml programs contain local state which can be verified by testing-based approaches. Hence, for second part of the evaluation of the proposed tool, we first convert few of these OCaml programs to equivalent Java classes storing state as global variables, and then use this manually curated dataset as input programs to the tool. This will also allow us to compare the effectiveness of the tool with respect to Hobbit.

Chapter 3

Background

In this section, we first discuss in detail the main engine of FuzzDiff i.e JQF to gain some background over coverage-guided fuzzing mechanisms and the role it plays in the design of the tool. We then discuss an important limitation of JQF and how FuzzDiff attempts to address it.

3.1 JQF

FuzzDiff uses JQF [30] as the engine for generating coverage-guided random inputs of structured data types and executing the test cases with these inputs.

JQF is an open-source testing platform which allows fuzz testing of java programs more efficiently as compared to other fuzz testing platforms. JQF is a proven bug finder, discovering 42 unknown bugs in few of the most stable open-sources packages on the internet including OpenJDK, Apache Commons, Apache Maven Google Closure Compiler and Mozilla Rhino. Hence, JQF makes an ideal utility for fuzz testing in FuzzDiff.

JQF is built on top of JUnit QuickCheck [4], a library that enables property-based testing in JUnit. JQF uses coverage guidance for generating random inputs for fuzzing. In FuzzDiff, we use the default guidance mechanism i.e semantic fuzzing using Zest for coverage-guided fuzzing, but we also discuss other guidances that ship with JQF like AFL [8], PerfFuzz [24] and RLCheck [33]. This ability of JQF to plugin any custom fuzzing algorithm and be compatible with other existing guidance mechanisms is what sets it apart from other fuzz testing platform like Kelinci [20] which only supports AFL fuzzing algorithms. There are other utilities like Randoop [28] and FuzzChick FuzzChick [23] which also use coverage guided, property-based testing in their implementation. Hence,

JQF is also referred to as a modular framework for fuzz testing.

JQF programmatically instruments Java classes representing program under test using

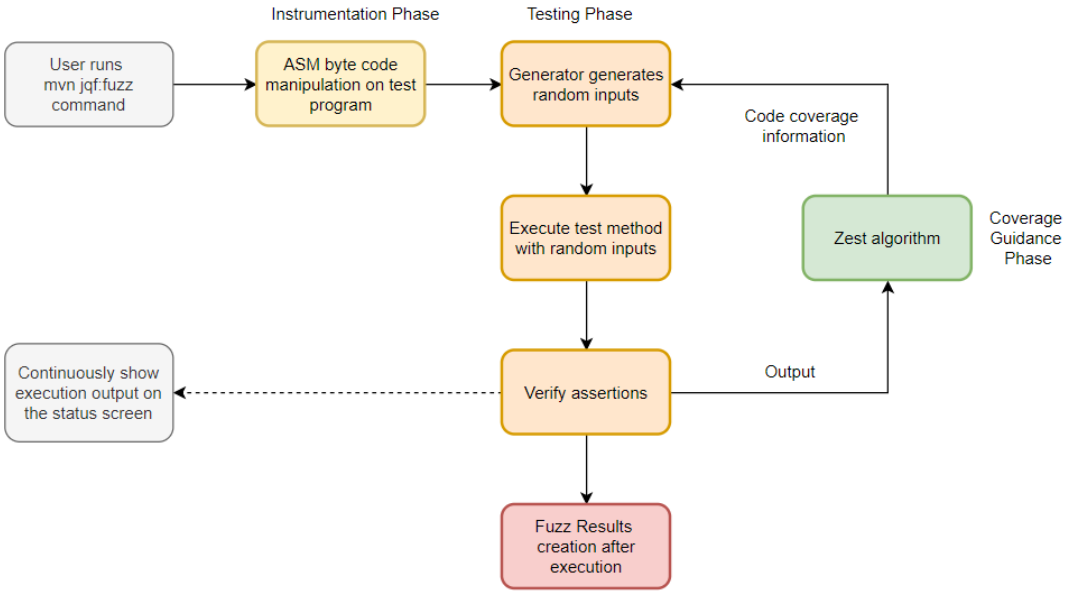

Figure 3.1: JQF Workflow Diagram

the ASM bytecode manipulation framework [3]. This instrumentation enables JQF to inject code coverage events like the branches covered and invocation of method calls into the test program. When the test program is executed, these events are emitted and the information is conveyed to the generator instance. Figure 3.1 shows the end-to-end flow of how JQF works in phases.

3.1.1 Coverage-guided Fuzzing (CGF)

Coverage-guided fuzzing [31] or CGF is a powerful testing technique for generating inputs that provide maximum code coverage. It is also referred to as greybox fuzzing technique [6] because it uses partial knowledge of the test program to determine which parts of the program are covered by the input fed into it. The fuzzer or the random input generator monitors the execution of program under test and utilizes this information to make decisions about the next set of inputs.

Figure 3.2 shows the underlying algorithm of coverage-guide fuzzing described in [31]. Initially, a set of random inputs are fed into the program under test and the results of code coverage are stored with the inputs. Based on this information, the next set of inputs are mutated by using an algorithm to maximize code coverage. If a random input

Algorithm 1 Coverage-guided fuzzing.

Input: program p, set of initial inputs I
Output: a set of test inputs and failing inputs

```
1:  S ← I
2:  F ← ∅
3:  totalCoverage ← ∅
4:  repeat
5:    for input in S do
6:      for 1 ≤ i ≤ NUMCANDIDATES(input) do
7:        candidate ← MUTATE(input, S)
8:        coverage, result ← RUN(p, candidate)
9:        if result = FAILURE then
10:         F ← F ∪ candidate
11:       else if coverage ⊈ totalCoverage then
12:         S ← S ∪ {candidate}
13:         totalCoverage ← totalCoverage ∪ coverage
14: until given time budget expires
15: return S, F
```

Figure 3.2: Coverage-guided Fuzzing [31]

leads to a increase in code coverage, it is stored for future reference. If an input causes a crash, a bug is found. In this way, the inputs keep involving with each test execution and the newly generated inputs are able to expose uncovered parts of the program. With this approach, CGF algorithms have successfully found bugs and security vulnerabilities in many open source libraries effectively.

JQF's Guidance interface allows researchers to implement their own coverage guidance mechanism and run JQF using the static method GuidedFuzzing.run().

3.1.2 Usage

In practice, JQF [29] is a Maven plugin or package published on the maven central repository. This plugin can be injected as a dependency into any maven project, which makes its easy to use and setup. JQF is also made available as a jar for command line interface (CLI) usage. For production-ready usage, JQF also provides integration with CI/CD pipelines on Gitlab. When a pull request is merged into master branch on Gitlab, a suite of regression tests are executed containing the fuzz targets and bugs are reported if any.

To use JQF [29] in an existing maven project, @RunWith(JQF.class) annotation is used to annotate the test class using JQF Runner and the test method is annotated with @Fuzz to declare method as a parameterized test case. Figure 3.3 shows an example of one such test class TrieTest which has a test method testMap2Trie annotated with @Fuzz, indicating that its parameters will be fuzzed with random values and the test method will be executed either until the user forces it to stop, a user-specified timeout occurs or until a

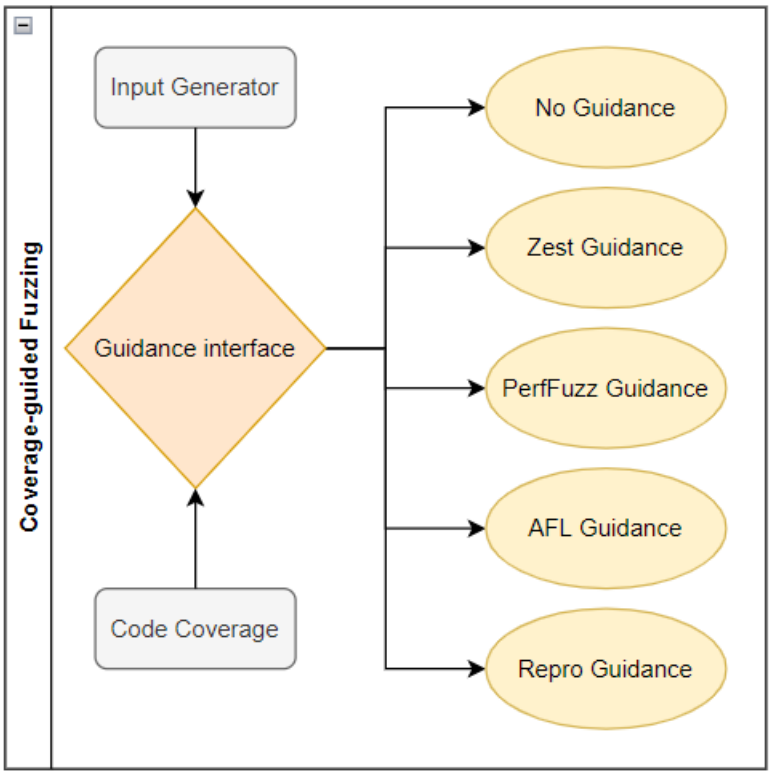

Figure 3.3: JQF - Coverage-guided Fuzzing using Guidance interface

failure occurs. At every execution, the property is verified on a set of assertions, which checks if the input value satisfies the condition specified by the user. The output of these assertions define the next set of inputs.

```
1  @RunWith(JQF.class)
2  class TrieTest {
3    @Fuzz /* Arguments are generated randomly by JQF */
4    public void testMap2Trie(String key,
5                             Map<String,Integer> map){
6      assumeTrue(map.containsKey(key));
7      Trie trie = new PatriciaTrie(map); // Map2Trie
8      assertTrue(trie.containsKey(key));
9    }
10 }
```

Figure 3.4: Simple property test using JQF [30]

To run JQF for this test method, the user needs to execute the following maven command:

```
mvn jqf:fuzz -Dclass=TrieTest -Dmethod=testMap2Trie
```

On running this command, it invokes the method repeatedly executes a number of test execution on the specified test class and test method. JQF by default uses the Zest algorithm [31] for generating the inputs and it can execute more than 5000 tests in 5

seconds. During all the executions, JQF provides a status screen on the terminal showing the statistics in the form of information such as total number of executions, number of valid inputs, unique failures, total coverage etc. The user can end the execution at any time and see the final output of the status screen.

Figure 3.5 shows the status screen for Semantic fuzzing of testMap2Trie method de-

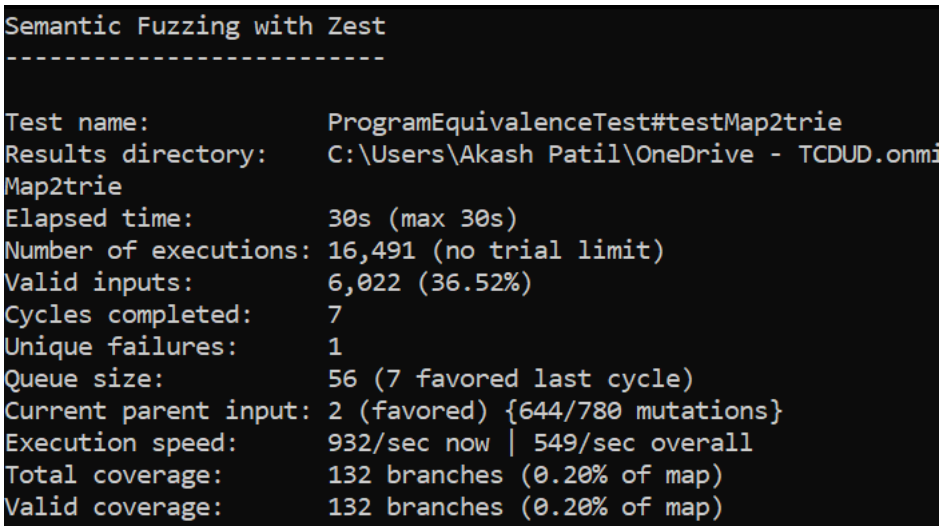

Figure 3.5: JQF status screen for testMap2Trie method

scribed in Figure 3.4. By default, Zest algorithm is used for the fuzzing unless explicitly specified by the user. It can be observed that the report shows one unique failure (or assestion violation) in the test execution, while only 36.5% of the inputs passed the test. A failure can be considered as the test scenario which leads to an assertion violation. This shows how effective Zest algorithm combined with fuzzing is in identifying bugs in the implementation of Trie in an open source library. It was able to generate and run more than 16000 inputs on the test method within a period of 30 seconds. The same set of executions without Zest does not lead to any failure as zest explores a special corner case which random sampling with no guidance cannot usually find. In the next section, we will discuss how Zest was able to find the violation.

After the end of execution, JQF also stores the test inputs for unique failures (if any) in the target/fuzz-results/failures directory. Apart from the failures, JQF also saves the corpus of successful inputs in the target/fuzz-results/corpus directory. These test cases can be executed again by running the command:

```
mvn jqf:repro -Dclass=TrieTest -Dmethod=testMap2Trie
              -Dinput=target/fuzz-results/failures/idn_00000
```

The output will be a trace log of the AssertionError which contains the input for which the violation occured and the line number at which the failure occured. This feature allows the developer to easily fix the bug by executing all the unique failures.

3.1.3 Semantic Fuzzing with Zest

JQF by-default supports the Zest algorithm [31] for guidance in fuzzing, which also becomes the default choice for fuzzing in FuzzDiff. Zest is a technique designed specifically for property testing i.e it combines the power of coverage-guided fuzzing with the property-based random testing facilitated by JUnit QuickCheck [4]. It uses semantic knowledge from previous execution to determine the next ideal input candidate for maximum code coverage.

Zest is a structure-aware algorithm i.e it operates on structured inputs and leverages the knowledge of input format to generate inputs which are valid syntactically. On the other hand, tools like AFL or libFuzzer [2] operate on sequence of bytes on compiler level.

Input: program p, generator q
Output: a set of test inputs and failing inputs

1: $\mathcal{S} \leftarrow$ {RANDOM}
2: $\mathcal{F} \leftarrow \emptyset$
3: $totalCoverage \leftarrow \emptyset$
4: $validCoverage \leftarrow \emptyset$
5: $g \leftarrow$ MAKEPARAMETRIC(q)
6: **repeat**
7: **for** *param* in $\mathcal{S}$ **do**
8: **for** $1 \le i \le$ NUMCANDIDATES(*param*) **do**
9: $candidate \leftarrow$ MUTATE(*param*, $\mathcal{S}$)
10: $input \leftarrow g(candidate)$
11: $coverage, result \leftarrow$ RUN(p, *input*)
12: **if** *result* = FAILURE **then**
13: $\mathcal{F} \leftarrow \mathcal{F} \cup candidate$
14: **else**
15: **if** $coverage \nsubseteq totalCoverage$ **then**
16: $\mathcal{S} \leftarrow \mathcal{S} \cup \{candidate\}$
17: $totalCoverage \leftarrow totalCoverage \cup coverage$
18: **if** *result* = VALID and $coverage \nsubseteq validCoverage$ **then**
19: $\mathcal{S} \leftarrow \mathcal{S} \cup \{candidate\}$
20: $validCoverage \leftarrow validCoverage \cup coverage$
21: **until** given time budget expires
22: **return** $g(\mathcal{S})$, $g(\mathcal{F})$

Figure 3.6: Zest algorithm [31]

Zest first converts a random input generator like JUnit QuickCheck into a custom parameteric generator g which acts on parameter sequences rather than seeded inputs. For the first iteration, theses sequences are random and are updated with each iteration. In addition to total coverage, Zest also maintains valid coverage which is the set of coverage

points covered by the valid inputs in g. Once the program is executed on an input, the result variable stores whether the input is valid or not. If the input leads to a violation then it is considered invalid. If result is valid and it covers code statements which have not been covered by previous valid test, then the validCoverage variable is updated and that input is included in parametric sequence for the subsequent mutation. This loop continues until the user interrupts the execution and it returns the corpus of successful inputs g(S) and failure inputs g(F).

In this manner, Zest synthesizes the inputs to converge towards complete code coverage and guides or biases QuickCheck style generators towards generating structured inputs. This also means that Zest is dependent on a structured-input generator in contrast to AFL [8] which performs byte-level mutations on the input string. Hence, Zest's effectiveness in coverage-guided fuzzing may vary based on the quality of the generator used.

3.1.4 Binary Fuzzing with AFL

AFL or American Fuzzy Loop [8] is a very popular CGF fuzzing tool for C/C++ programs. It is best suited for finding syntactic bugs in and has discovered thousands of bugs and security vulnerabilities in programs that parse binary data, such as image decoders and media players. The creators of AFL define it as a brute force but rock-solid instrumentation approach to coverage-guided fuzzing. AFL integrates its instrumentation with the target compiler by adding small snippets of code everywhere to help collect coverage information when the code is eventually executed.

AFL [7] takes the test case specified by the user and puts it into a queue. The binary program, instrumented with AFL, then takes the test case as an input and then mutates the data trying to find interesting inputs. The repeated mutation is done using a wide variety of strategies. In case any of the mutations leads the binary into a new state, the binary reports back with new coverage information. The information contains which edges are discovered newly and which input led to it. These inputs are added as new entry into the queue. This helps the AFL fuzzer to mutate the input to further explore new edges.

AFL's mutation technique utilizes different heuristics that are compatible with programs that parse fixed-size binary files. AFL does not explicitly differentiate between INVALID and FAILURE results unlike Zest. As a result, JQF's AFLGuidance proves to be highly effective when used with test methods that take only InputStream type as argument.

AFL specializes in fuzzing binary data, and hence integration with JQF means that it is only applicable for Java programs using streams of bytes instead of structured data types.

3.1.5 Complexity Fuzzing with PerfFuzz

PerfFuzz [24] is a another technique for feedback-directed fuzzing supported by JQF, which as the name suggest, uses performance feedback to mutate the inputs. PerfFuzz is essentially an extension of AFL, where its extends the code coverage map to consider performance counters. If a mutated input either leads to newly explored edges or finds interesting information related to execution time, it saves the input for further mutation.

Using this strategy, PerfFuzz is able to effectively produce test inputs that maximize performance counters and discover performance bottlenecks. In terms of implementation in JQF, PerfFuzz Guidance is defined as a subclass of AFL Guidance, as it inherits the underlying mechanism for mutation. Post evaluation, PerfFuzz was found to be effective at generating inputs that exhibit different time or space complexity vulnerabilities.

3.1.6 Reinforcement learning with RLCheck

RLCheck [33] is an AI based approach for generating valid test inputs for property-based testing of a program. It uses reinforcement learning for guiding generators towards producing diverse set of sematically valid inputs. The main goal of RLCheck is to produce large set of valid inputs in very short span of time for effective property testing. To achieve this, the problem is first formalized into a diversifying guidance problem and then reinforcement learning algorithm like on-policy Monte Carlo Control is applied to guide the generator.

3.2 JUnit QuickCheck

JQF leverages the JUnit QuickCheck framework to produce structured inputs. In practice, JUnit QuickCheck [4] is a library that allows writing and running property-based JUnit test cases. JUnit QuickCheck is Java implementation of QuickCheck [16] tool originally designed for property-based testing of Haskell programs.

When we say property-based testing, it means that the test case verifies a property of the program, after executing the code against a number of randomly generated inputs. These characteristics or properties of code are verified using a set of JUnit based assertions, which repeatedly checks if the property holds true for the input fed into the test

program. This provides us some guarantee that upon subjecting the program to wide range of inputs, the property of the program is preserved. Hence, it will be interesting to see if this verification technique can be used to instead check for equivalence of two programs having the same set of input parameters.

JUnit QuickCheck uses the underlying mechanism of the JUnit testing framework to make property-based testing compatible with Java programs. Typical JUnit test cases are parameterless i.e the user is expected to mock the input being fed into the test program. But with JUnit QuickCheck, the framework allows writing parameterized test cases where the user can specify the data type of the input which is to be verified in the body. Such test cases need to be annotated with @Property in order for the framework to identify a "Property Test".

During the execution of the test case, its parameters are subjected to built in generators which randomly generate values based on the data type of the parameter. The random input generation, however, does not use any feedback mechanism, and hence are unlikely to find a corner case even after thousands of executions. The library provides a collection of generators for most common data types in Java. The test driver can also be directed to create generator automatically by pointing to constructors or public data members of class T. In case a generator is not available for a custom type, the user can provide its own hand-written generator by extending the Generator<T> class. It is an abstract class which provides abstract method called generate() that can be implemented to randomly sample a new instance of T.

All Generator instances use SourceOfRandomness object which is an API to generate random instances of objects in a non-deterministic manner. This SourceOfRandomness object is subjected to an output stream of bytes by default, however, JQF overrides this implementation to use the byte stream facilitated by Guidance.getInput() method. In this way, JQF utilizes the generators to produce feedback-directed semantically valid inputs. It can be also said that using JQF with No Guidance is equivalent to using plain JUnit QuickCheck.

QuickCheck also provides a range of Java annotations liek @InRange, @Size, @NotAllowed etc. to restrict or control the size of input being generated. This proves useful especially for recursive functions where large inputs may lead to long running programs defeating the purpose of "Quick check". The mechanism of controlling inputs is carried over from QuickCheck for Haskell discussed in the next section.

3.2.1 QuickCheck for Haskell

As discussed in the previous section, JUnit QuickCheck is inspired by the QuickCheck tool for Haskell programs [16]. QuickCheck was a tool originally designed to help Haskell programmers perform property testing of Haskell programs. Haskell is a high-level statically typed functional programming language based on mathematical functions. Functional programs are well suited for automated random testing, and hence QuickCheck intends to provide a mechanism to test these functions in a shorter period of time. In QuickCheck, the Haskell programmer is required to provide a specification of the program, which are the properties that must be satisfied by the functions. QuickCheck then randomly generates large number of inputs and checks if the properties hold true for each input. The specification can be conveyed using set of combinators provided by QuickCheck, which can also be used to notice the distribution of test data.

Random testing proves to be most effective in cases where wide distribution of test data is required. Hence, QuickCheck facilitates test data generators for common types in Haskell, and gives the control to the user to decide how uniform the distribution of data must be. It also provides a novel way of controlling or restricting the size of input especially for recursive functions.

QuickCheck, when used for testing programs on real Haskell systems showed that it can prove to be a very lightweight tool for automatic testing which is otherwise very costly on Haskell systems. For smaller Haskell programs, it is very likely that the random inputs will cover all branches of the programs exhaustively and hence fine grain testing also becomes the ideal approach for testing large Haskell programs.

3.3 Limitations

In the previous section, we discussed how effective JQF combined with Zest is in verifying properties of a program using fuzz testing. Incorporating a coverage-guidance mechanism improves the ability of test generators to produce semantically valid inputs, helping explore every corner of program under test. The verification task becomes more optimal with use of feedback-directed fuzzing.

Being an extension of QuickCheck, JQF comes bundled with a collection of generators for most common datatypes and collection interfaces in Java. It also provides generators for concurrent implementations and functional interfaces like lambda expressions. Table

3.1 shows few examples of JUnit QuickCheck generators for most commonly used types in Java.

Category	**Generators**
java.lang	StringGenerator IntegerGenerator LongGenerator DoubleGenerator CharacterGenerator BooleanGenerator ByteGenerator
java.math	BigDecimalGenerator BigIntegerGenerator
java.time	ClockGenerator LocalDateTimeGenerator YearGenerator ZonedDateTimeGenerator MonthDayGenerator InstantGenerator
java.util.concurrent	CallableGenerator
java.util.function	FunctionGenerator PredicateGenerator SupplierGenerator BinaryOperatorGenerator IntFunctionGenerator LongFunctionGenerator
java.util	ArrayListGenerator HashMapGenerator HashSetGenerator HashtableGenerator LinkedListGenerator SetGenerator StackGenerator

Table 3.1: Examples of Inbuilt Generators provided by JUnit QuickCheck [4]

Based on the type of parameter, JQF automatically picks up the corresponding Generator from the library and sets it for fuzzing random inputs of that type. While the list covers most commonly used types and data structures, JUnit QuickCheck still cannot fuzz custom objects containing properties of simple data types. In most programming scenarios, whole Java objects are passed as parameters into methods rather than individual attributes. This is mainly to achieve abstraction between two java classes. For instance, suppose a programmer wants to write a method that checks if a person's age is

above 18 and the Person object contains a number of attributes including age. Following the design principles of object oriented programming, the programmer is more likely to define a method with Person object as a parameter rather than an integer variable storing the age. This ensures that Person's age is encapsulated in an object and instructions in method can then call getter method to get the age of the person.

Generating random instances of such custom objects is not possible using JUnit QuickCheck and hence it becomes a limitation in its usage for fuzz testing. To address this shortcoming, We propose the design and implementation of a Generic generator which supports the fuzzing of a simple custom Java object created by the user. The usage of the Generic generator remains the same as the other readily available generators, but the user needs to decide which generator supports their requirements best.

3.4 Generic Generator

The Generic Generator aims to provide a fuzzy generator that can create random instances of custom objects. The design of Generic Generator follows the underlying mechanism of a Generator<T> instance, but instead of specifying the type T of Generator while extending its abstract class, we use the parent Object class. This makes the Generator compatible with all classes as in Java, by default all custom classes extend the Object class. The design heavily relies on the use of Generics and Java reflection API to avoid any manual changes in the implementation.

Moreover, the Generic Generator has been designed to provide flexibility to the user in terms of controlling the size of inputs for specific data types. The user can specify maximum and minimum value of the Integer types, the default for which is Integer.MAX and Integer.MIN. The user can set the number of iterations for array types, for which default value is set to 10. And lastly, the user can specify the length of the string to be generated, with 5 set as the default length. The user can specify these requirements through parameters in the command line interface, which is explained in detail in Usage Section.

Working

Similar to implementation of handwritten Generators, we define a Generic Generator class which extends Generator<T> abstract class. This class implements the abstract method generate(). The constructor of the class takes inputs as a Class<Object> and

sets its value to a global variable cls of type Class <Object>. This sets the class of the input parameter being fuzzed. Next, in the generate() method, we first check if the class is primitive, wrapper or array using inbuilt methods in java.lang and then use Java reflection API to create an instance of this class. The user may have to make changes in the code where declared constructor of the class has been determined using reflection. Once the correct constructor is identified, the class is then instantiated with specified arguments (incase of all argument constructor) or no arguments (incase of no argument constructor). Next, all the declared fields in the class are determined using reflection. Then, using a for loop, we iterate through each field, get its type, generate random value for that type and finally set the random value in the field. In this manner, we ultimately get a random instance of the custom class object, which is returned back to the test driver. The other private methods in the class provide utility for generating random values using SourceOfRandomness instance, taking into consideration user specified restrictions.

Limitations

The Generic Generator has its own limitatons in terms of the data types that can used in custom objects. In its first version, it only supports types which are primitive, wrapper class or Array instances of primitive and wrapper classes. Attempts were made to support Collection data structures like ArrayList, HashSet, LinkedList etc. but since the type of collection object is determined at runtime in JVM, the Generic Generator fails to generate random instances using SourceofRandomness. However, the user can always use the inbuilt CollectionGenerator for fuzzing Java Collection objects.

Moreover, the Generic Generator is not entirely generic as it may require code modification if the the class to be fuzzed contains an All Argument Constructor, which makes it impossible to instantiate the class using java reflection as there is no way of guessing the number and type of arguments in the constructor. This becomes a major limitation of the Generic Generator and remains one of the points to be addressed in the future versions.

Chapter 4

FuzzDiff

In this section, based on the design choices, we define the functional requirements of FuzzDiff. We then describe the architecture of FuzzDiff and the various components involved in the design. And finally, we describe the testing-based approach utilized in FuzzDiff and then review the implementation in detail.

4.1 Functional Requirements

FuzzDiff aims to solve program equivalence using the powerful utility that JQF [30] provides for coverage-guided fuzzing. All the features of JQF discussed in previous sections contribute towards devising a novel approach of checking two programs for equivalence. Focusing on this objective, we introduce a tool that performs program verification after code refactoring keeping JQF at the core of it. It is however important to define the scope of the tool by defining the requirements.

These evaluated requirements of the equivalence tool are:

1. The input to the tool is a pair of two java programs (original and refactored) containing the two public methods to be tested for equivalence. The name and the signature of the methods must be same. The tool should be able to take the three inputs i.e Original class, refactored class and method name from the user through command line interface. The input classes may contain global state in form of globally declared variables. Though generally global data members are private and are accessed via public getters and setters to achieve abstraction, for this tool, it is assumed that the data members are accessible from any class.

2. The tool should compare the output of the two methods along with the global state of the two classes.

3. The tool must compare the state of the input before and after the execution of the program. In case of multiple parameters, iteratively compare the state of all inputs.

4. If the methods return type is void, then compare only the global state of the two classes.

5. If the methods have no parameters, compare only the final output and state of the program as fuzzing is not possible for that method.

6. The tool should also semantically compare the calling behaviour of the two methods.

7. If the two methods make method invocations to other classes, provide an interface for the user to provide those dependent classes.

8. The tool should support a number of options to control the fuzzing mechanism. It should also allow user to provide its own custom generator and use it for fuzzing.

9. As an output, the tool must provide user with the result of the verification task i.e if the two programs are equivalent or not. If they are not equivalent, then notify user which input led to in-equivalency and which line of the program resulted in it.

Success Criteria for Equivalence

Before considering design choices for implementation, it is important to define the criteria for equivalence and non-equivalence. The two input programs can be considered equivalent if -

1. The state of input(s) remains same before and after execution in terms of the value they hold.

2. The output of both the methods after execution is equal by value.

3. The global state i.e the value of all global data members is equal after execution.

4. The calling behaviour of the two methods is similar i.e the number of method invocations and the order in which invocations are made is equal.

4.1.1 Example of equivalent and non-equivalent programs

Consider two programs Program1 and Program2 returning the maximum element in an integer array. These two programs shown in Figure 4.1 and 4.2 are to be checked for equivalence using FuzzDiff. The original class Program1 uses iterative approach to compute the largest element, while the refactored class Program2 first sorts the array and

```java
package classes;

public class Program1 {
    public int largest(int[] arr){

        // Initialize maximum element
        int max = arr[0];

        // Traverse array elements from second and
        // compare every element with current max
        for (int j : arr) {
            if (j > max) {
                max = j;
            }
        }
        return max;
    }
}
```

Figure 4.1: Program1.java

```java
package classes;

import java.util.Arrays;

public class Program2 {
    public int largest(int[] arr) {

        // Sort array
        Arrays.sort(arr);
        return arr[arr.length - 1];
    }
}
```

Figure 4.2: Program2.java

returns the last element of the sorted array. Both are different approaches to solving the same problem and can be considered as a real world example of refactoring.

The two programs, as per the criteria defined above, are **inequivalent** because the state of input, which is the integer array in this example, is not same at the end of execution of both the programs. The Array.sort() method in Program2 sorts the input array, while Program1 uses local variables to determine largest element without affecting the state of the input array. Hence, despite the final output of the two method being equal, the inequality of input state after execution makes the two programs contextually inequivalent.

Now consider another program Program3 shown in Figure 4.4, which similar to Program1 and Program2, computes the largest element in an Integer array but uses a different approach which largely resembles to that of Program1 (Figure 4.3). It first stores all the elements of the array in an ArrayList and then returns the maximum element in the List using Collections.max() method. This approach can be considered as a minor refactoring of Program1 as the algorithm is mostly similar. As per the success criteria defined in previous section, the two programs, based on verification, prove to be equivalent as the final output of the two programs and the input state after execution, both are equal.

For demonstration purposes, lets assume both programs store the largest element in a global variable max_num, initially set to Integer.MIN_VALUE. The two methods also make invocations to two methods testMethod1() and testMethod2() with the intention to only verify the number of method invocations and the order of those invocations. As

```java
public class Program1 {
    public int max_num = Integer.MIN_VALUE;

    public int largest(int[] arr){

        int max = arr[0];

        // Mock call invocations for demonstration purposes
        Main.testMethod1(max);
        Main.testMethod2(max);

        // Traverse array elements from second and
        // compare every element with current max
        for (int j : arr) {
            if (j > max) {
                max = j;
            }
            max_num = max;
        }
        return max;
    }
}
```

Figure 4.3: Modified Program1.java

```java
public class Program3 {

    public int max_num = Integer.MIN_VALUE;

    public int largest(int[] arr){

        // Initialize an ArrayList
        List<Integer> list=new ArrayList<>();

        Main.testMethod2(max_num);
        Main.testMethod1(max_num);

        // Add all array elements to the list
        for (int j : arr) {
            list.add(j);
        }

        max_num = Collections.max(list);
        return Collections.max(list);
    }
}
```

Figure 4.4: Program3.java

per the equivalence criteria, based on structural analysis, these two programs prove to be inequivalent as the order of method invocations to methods testMethod1() and testMethod2() is not the same.

4.2 Design and Implementation

Considering all the functional requirements and the equivalence criteria, we make design choices for the tool and devise a novel approach to perform equivalence checking using fuzz testing.

4.2.1 Architecture

Figure 4.5 shows the proposed System Architecture for FuzzDiff. The two programs (original and refactored) are taken as input from the command line interface and fed into the fuzz checker. The underlying working of fuzzer has already been discussed in Chapter 3. We use the JQF [29] framework for coverage-guided fuzz testing using Zest guidance algorithm and incorporate it with a set of verification tasks defined by JUnit assertions. These set of assertions check for equivalence based on criteria defined in Section 4.1. These verification tasks occur sequentially and in sync with JQF fuzzer. The Fuzz Checker by default uses the Generic Generator for fuzzing but the user can provide its own handwritten generator as input. We use Java reflection API to automatically get declared method of programs under test, invoke them and perform other class-level comparisons.

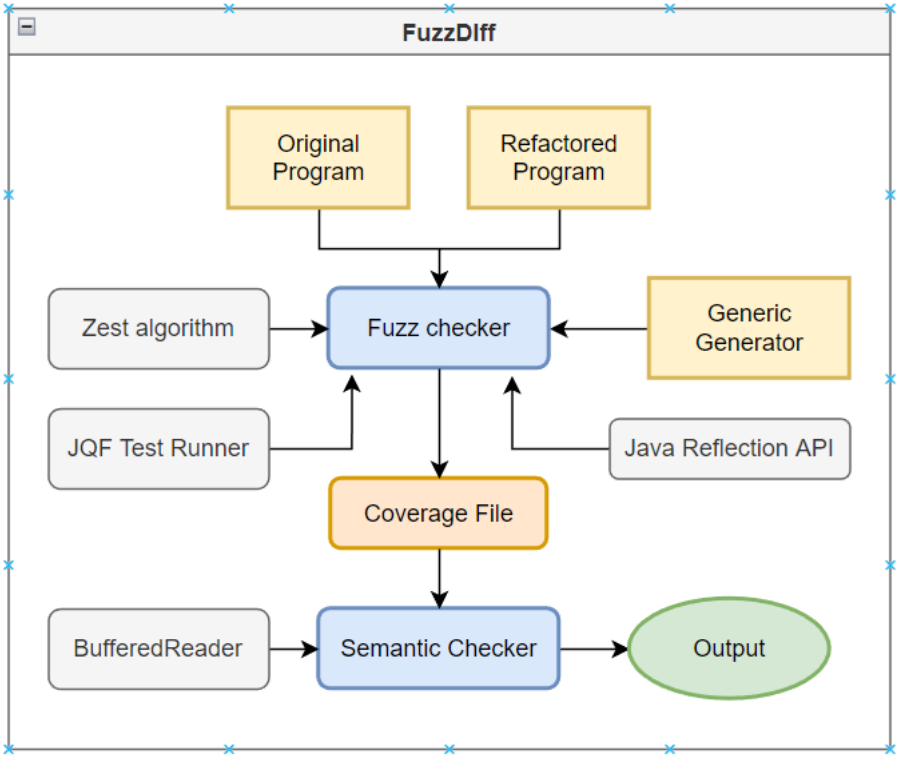

Figure 4.5: FuzzDiff - System Architecture

The JQF-based Fuzz Checker then uses the reproduced coverage file to perform semantic check by analysing the coverage file and parsing it to fetch necessary information regarding method invocations. The final output is then computed after all checks are computed and presented to the user.

4.2.2 Implementation

FuzzDiff is made available as a Maven-based tool. Apache Maven is a build automation and project management tool made primarily to support the packaging Java programs. This implementation choice limits its usage but as JQf is Maven-based utility, it makes sense to build FuzzDiff as Maven project. It provides many complementary plugins that assist in building large scale Java-based applications. The Maven project contains a pom.xml file where we can version the tool and inject dependencies published on maven's central repository. One of these dependencies is jqf-fuzz, which gives us access to all the features of JQF including the annotations and interfaces. JQF internally uses JUnit QuickCheck as a maven dependency. We also define three custom system property variables namely original, refactored and methodName in maven-surefire-plugin in pom.xml. This configuration allows all the classes in the application to recognize the input classes

and method under test.

The Maven project is structured in the following manner to satisfy the functional requirements of the tool:

- The "classes" package in src/main stores all the input programs and its dependent classes. This is the location where the user is supposed to place the input classes, change its package name and resolve imports if any.
- The "generators" package in src/main stores all the custom generators provided by the user, which may be required for fuzzing custom objects. It also houses the Generic Generator proposed as part of this tool.
- The main class "EquivalenceChecker" contains the main method which serves as the entry point of the application.
- The test directory contains the two test classes and ProgramEquivalenceTest and AdditionalChecksTest responsible for performing the equivalence check. The former corresponds to the fuzz checker component in the architecture, while the latter corresponds to the semantic checker component. These two classes contains the JUnit test cases which can be directly executed from the command line or can be run in IDE itself by the developer.
- The project also contains the run.sh script which provides the interface for running the tool.

Class diagram

Figure 4.6 shows the UML class diagram, notating all the classes in FuzzDiff. It describes the structure of the system by showing the system's classes, their attributes, methods and the relationship between classes.

The user first provides FuzzDiff with the two input programs i.e the original and refactored class containing same state variables and methods with same name and signature. These two classes are placed in the classes directory along with its dependent classes if any. In case the user wants to provide a custom generator for the two input classes, it is placed in the generators directory. Based on the type of declared constructor of the two classes, the user makes changes to the Generic Generator class so that the classes can be instantiated using reflection. If the constructor contains arguments, then the user

also needs to make few changes in the fuzzTestForEquivalence() method to support the instantiation of the two classes.

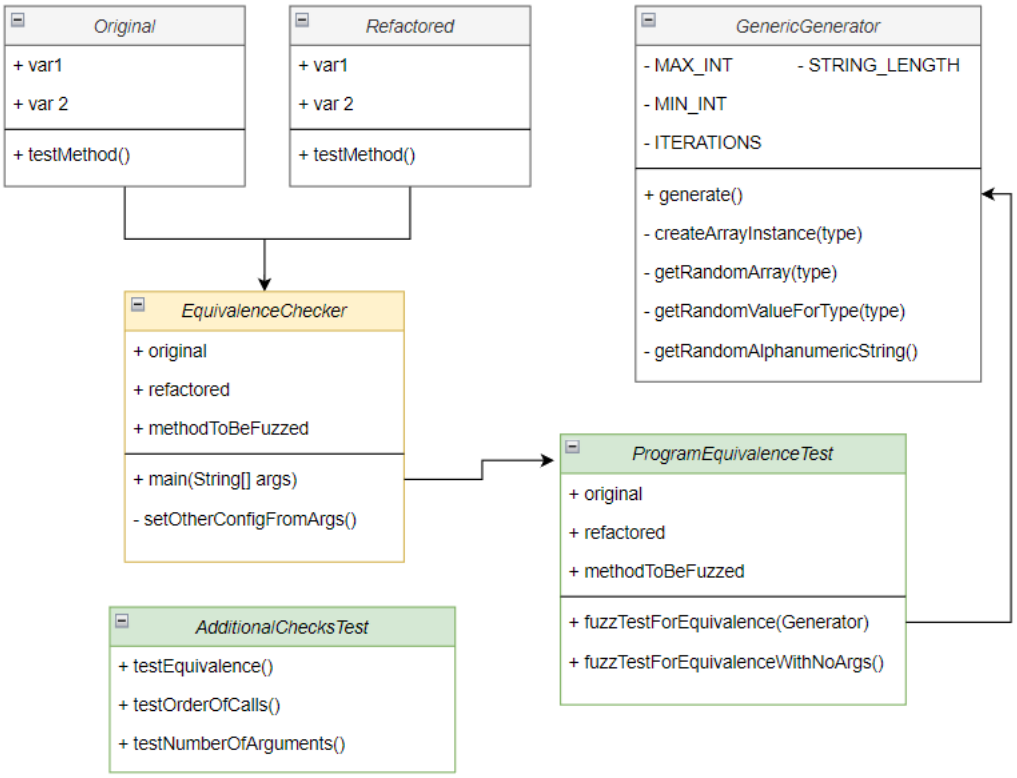

Figure 4.6: FuzzDiff - UML Class Diagram

Next, the user runs the shell script where the two class names and the method name to be fuzzed are provided as input by the user. The script executes the main method in EquivalenceChecker class by running the following maven command:

```
mvn exec:java -Dexec.args="$original $refactored $method_name"
```

where $original, $refactored and $method_name are the three inputs form the user. The main method first validates the three arguments by checking if the number of arguments are 3 and if they are not null. If not null, the String inputs are then converted to Class<?> objects and then all the declared methods in those classes are fetched and stored in two arrays. If none of those declared method names match the input method name, an exception is thrown saying "METHOD_NOT_FOUND". If a match is found, the parameter count of that method is determined using reflection. If the parameter is zero, it means that the method requires no fuzzing and it only subjected to verification tasks defined explicitly in fuzzTestForEquivalenceWithNoArgs() test method in ProgramEquivalenceTest class. If the parameters exist for the method, then it qualifies for fuzz testing

carried out by test method fuzzTestForEquivalence() in ProgramEquivalenceTest class.

In order to execute the above test methods directly from the main class, a different process is started within the same thread. A Runtime object is initialized which executes the following maven command onthata seperate process using the Runtime.exec() method:

```
mvn.cmd jqf:fuzz -Dclass=ProgramEquivalenceTest
                 -Dmethod=fuzzTestForEquivalence
                 -Dtime=10s
                 -Doriginal=n$original
                 -Drefactored=n$refactored
                 -DmethodName=n$methodToBeFuzzed
```

The argument options -Doriginal, -Drefactored and -DmethodName have been defined as system property variables in pom.xml. This allows us to transfer information from EquivalenceChecker class in main directory to ProgramEquivalenceTest class in test directory, which is not otherwise possible through normal object passing. The fuzz time has been set to 10 seconds, which means the fuzzing will automatically stop after 10 seconds. Before the process is started, the setOtherConfigurationMethod() sets the custom properties the user may provide to control the size of inputs.

Once the command is executed, control goes to ProgramEquivalenceTest class, where one of the two test methods is ran. However, before the tests are ran, we define a setup() method annotated with @BeforeClass, which picks up the three inputs set in the system property variables and initialize them on a global level. This annotation instructs the test class to execute the setup method before any test method could be executed.

The two test methods fuzzTestForEquivalence() and fuzzTestForEquivalenceWithNoArgs() serves the logic for the proposed novel approach of program equivalence. The former method is a typical JQF parameterized test driver annotated with @Fuzz, which instructs the test class to perform fuzzing on the parameters of the test method. Since the parameters of the input method cannot be dynamically determined beforehand, it requires the user to manually add the type and variable name of the parameters in the brackets along with the type of generator being used for those parameters (By default it will be inbuilt JUnit QuickCheck generators). On making these changes, the equivalence checker is now ready to be executed. Once it is executed by the mvn:jqf command, it first instantiates the the original and refactored class.

To verify if the state of input(s) remain same before and after execution, we first create a clone of the parameter objects. Then, we create a Method object based on the method name and the type of parameter (s) (as method overloading is a possibility). Using java reflection, we then invoke the original method by passing the clone object as parameter and store the final output of the invocation in Object x. We repeat the same for refactored method and store the output in Object y. Using simple JUnit assertions, the two clone objects are checked for equality. In case of multiple parameters, user is required to add more assert statements to check for equivalence.

Once the input state is verified, we then compare the output x and y using assertEquals() method. However, we only do this verification if the return type of the methods are not void.

The final fuzz test is verifying if the global state of the two objects is equal or not after the execution. To perform the verification, we first check if the classes even have a global state. No declared fields in the two classes indicate that this step is not required. Presence of any declared public fields indicates that they must be checked for equivalence. Initially, we check if both the class have same number of declared fields using assertEquals() method. This remains one of the conditions to be satisfied by the classes; that both must have same number of globally declared fields. If they are same, we store the values of those fields in two ArrayLists (one for each object) and then finally check if both the lists are equal using assertEquals() method. This step concludes the fuzz testing of the two methods as per the criteria defined for equivalence in Section 4.1.

The fuzzTestForEquivalenceWithNoArgs() method performs the same operations but only checks for final output and global state once. As there no parameters are specified for the target method, no repeated execution is required for fuzzing. Hence, it is annotated with @Test to instruct the test class to execute it only once. All the testing takes place in background and the user is abstracted from the commands being executed in background.

The next step after fuzz checking, as per the proposed architecture, is semantic checking implemented in AdditionalChecksTest class. After the fuzz testing is over, JQF stores the test inputs that passed all checks and/or led to failures if any. If the fuzz testing led to no assertion errors, it means that the two classes are equivalent till now and no test inputs will be stored in the fuzz-results/failures directory. This is what we check in the first verification task of semantic checking. If there are any failures in that directory, skip

the semantic checking and inform user about the inequivalency of the two programs. Otherwise, we perform few more tests that check the calling behaviour of the two methods. If no failures are detected in the first step, the shell script reproduces the corpus results stored in fuzz-results directory to generate the coverage file provided by JQF. This is an optional feature of JQF and can be enabled by using the option -DlogCoverage in the jqf:repro command. This coverage file contains the trace log for method execution from the first line of the test class to the last line of the class being tested. The information in this file allows us to see the method invocations made during execution for each of the two methods. Figure 4.7 shows a sample coverage.txt file generated after reproducing fuzz results for Program1 and Program3 described in Section 4.1.1.

```
(000000000) #<unknown>():0 --> ProgramEquivalenceTest#setup()V
(2046828546) classes/Program3#largest():18 --> generators/Main#testMethod1(I)V
(2046828547) classes/Program3#largest():19 --> generators/Main#testMethod2(I)V
(2046828548) classes/Program3#largest():22 [0]
(2046828548) classes/Program3#largest():22 [1]
(2055217153) classes/Program1#largest():13 --> generators/Main#testMethod1(I)V
(2055217154) classes/Program1#largest():14 --> generators/Main#testMethod2(I)V
(2055217155) classes/Program1#largest():18 [0]
(2055217155) classes/Program1#largest():18 [1]
(2055217156) classes/Program1#largest():19 [0]
(2055217156) classes/Program1#largest():19 [1]
(914370562) ProgramEquivalenceTest#fuzzTestForEquivalence():48 --> classes/Program1#<init>()V
(914370564) ProgramEquivalenceTest#fuzzTestForEquivalence():49 --> classes/Program3#<init>()V
(914370567) ProgramEquivalenceTest#fuzzTestForEquivalence():54 --> classes/Program1#largest([I)I
(914370570) ProgramEquivalenceTest#fuzzTestForEquivalence():59 --> classes/Program3#largest([I)I
(914370575) ProgramEquivalenceTest#fuzzTestForEquivalence():65 [0]
(914370579) ProgramEquivalenceTest#fuzzTestForEquivalence():69 [1]
(914370596) ProgramEquivalenceTest#fuzzTestForEquivalence():78 [0]
(914370596) ProgramEquivalenceTest#fuzzTestForEquivalence():78 [1]
(914370602) ProgramEquivalenceTest#fuzzTestForEquivalence():82 [0]
(914370602) ProgramEquivalenceTest#fuzzTestForEquivalence():82 [1]
```

Figure 4.7: Sample Coverage file for Program1 and Program3

The testCallingBehaviour() method fetches this coverage file and stores the content in a BufferedReader. Using regex expressions, it then scans through the file to search for instances where method invocations have happened. For example, in the sample coverage file, line 2,3 and 6,7 show external calls to another class from the largest() method using arrows (–>). We store all such lines in an array, split the line based on "–>" string and store that string in a list. The list is splitted into two halves for comparison. The calling behaviour of the two methods is then tested by two assertions which first checks if the two methods have equal number of invocations. If yes, then the second checks if the order of those invocations is equal or not. Though the semantic check only comprises of two verification tasks, further improvements can be made to check for higher degree of equivalency.

After executing this test case, If all the checks pass, then the two Java classes are considered equivalent. Any failure in either of the two verification steps causes the tool to

return the opposite result. Moreover, by leveraging the features of JUnit framework, we can show the user the inputs which led to failure and at what line the bug needs to be fixed after refactoring. This novel approach of checking equivalence using fuzz testing and semantic analysis is made available as an open-source tool and can be found at the following Github repository: www.github.com/akashpatil7/FuzzDiff

4.2.3 Limitations

In this section, we discuss the limitations of FuzzDiff and the constraints it imposes to its real-world user. This tool is mainly targeted to developers who perform code refactoring tasks and which to check if the functional behaviour is intact after the changes have been made. Any bugs found can then be resolved quickly. FuzzDiff aims to explore the idea of solving program equivalence using feedback-directed fuzz testing and its implementation is a mere attempt to achieving the research goal.

With the development of its first version, FuzzDiff comes with a host of drawbacks and limitations that may impact the overall user-experience of using the tool. However, we expect the next versions to solve major bugs and limitations in usage. Firstly, it only supports simple Java programs with minimal complexity and computations. The two inputs classes are expected to have a public method with same signature and same return type. This restricts the utility to only certain type of java programs. But in reality, Java programs can contain complex algorithms and can have dependency on many different classes. FuzzDiff expects the use of same variable names for global state comparison, which in real-world should not be the case for two functionally equivalent programs. In any refactoring task, the functionality can remain the same even after a complete overhaul of the class. This fact is ignored by FuzzDiff and due its heavy reliance on Java reflection API, it makes certain compromises in providing a flexible equivalence checker tool to the user. FuzzDiff may not as flexible as other readily available equivalence checkers but its compatibility with Java, which is one of the most sought-out programming in the world, makes it a great value proposition.

Secondly, in certain use cases, FuzzDiff requires the programmer to make changes in the code for meeting the requirements of the tool. These manual changes are mainly caused due to inability of automatically identifying the declared constructor of the class and instantiating the class.

Thirdly, the assertions defined as part of the verification tasks are not very exhaustive,

and address the formal definition of functional equivalence. The semantic checker only perform two checks and does not completely check for semantic equivalence. This constraint is mainly due to the lack of detailed trace available after the test execution.

Lastly, in terms of its usage, FuzzDiff requires user to manually place the input classes in a certain directory and provide all the other necessary files and generators required to execute the program under test. This can be very time-consuming and tedious task for the user, whose preference otherwise would be to import the utility as a package in their own project, hence eliminating the need to transfer files from one directory to another.

Despite its limitations and shortcomings, FuzzDiff is still is in prototype phase and remains a work under progress. The research goal of this tool was ultimately to check if the proposed methodology is implementable in real world and if yes, how effective would it be. With several patches and alternations in implementation, FuzzDiff could prove to be a very effective equivalence checker for Java or any JVM based programming language.

4.3 Usage

In this section, we show the usage of FuzzDiff and the system requirements for its utilization. The pre-requisites of the tool are listed down and a step by step procedure is shown in the form of a user flow diagram, guiding the user in using the tool to its full potential. The code has been commented with TODOs which assist user amking decisions tha suit the requirements better.

FuzzDiff is an open source maven project available on Github. To start with, the user first needs to clone the repository locally and install all the maven dependencies by running the following command in the cloned directory:

```
mvn clean install -DskipTests
```

After executing the command, all the required packages and plugins (including JQF) are downloaded in the local repository. The project is now setup locally. Next, the user needs to open the project in an IDE or any editor where changes can be easily made to the code. Fig 4.8 illustrates step-by-step procedure from there onwards with the help of a user flow diagram.

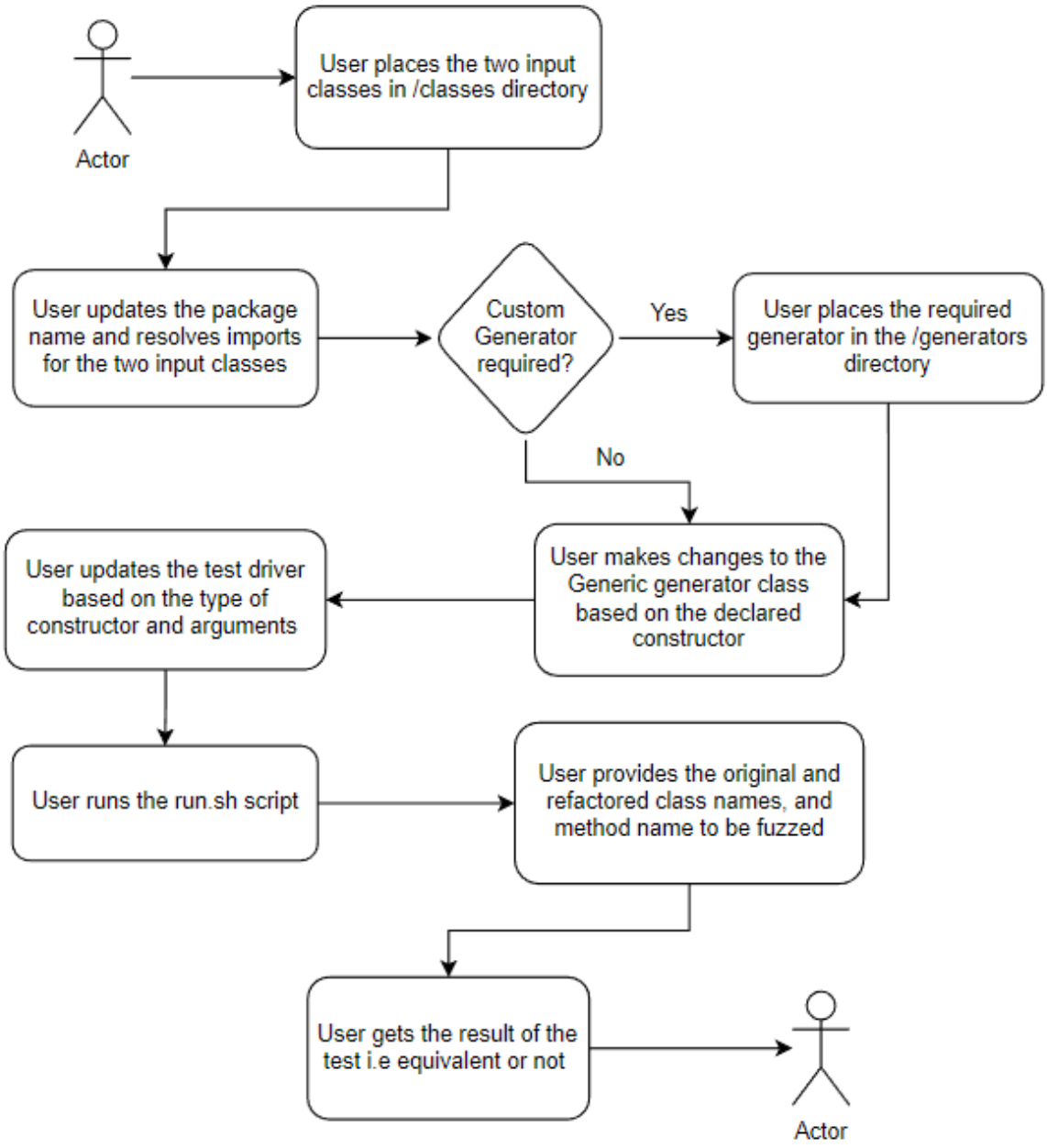

Figure 4.8: User Flow Diagram of FuzzDiff

4.3.1 Pre-requisites

In terms of pre-requisites for using the tool, it will useful for the user to understand the role and usage of JQF [29] and JUnit QuickCheck [4] in FuzzDiff. It will allow the user to troubleshoot and fix any JQF specific error quickly. With respect to the minimum system requirements, following installations are required to use the tool locally or on any server:

- Maven

- JDK 9+
- IDE or text editor
- Terminal or Command Prompt

4.3.2 Shell Script

FuzzDiff provides an user interface in the form of a simple shell script called run.sh, which when executed asks the user to provide three inputs: Original class name, Refactored class name and the name of the method to be tested. On entering these inputs, FuzzDiff executes different maven commands in background to perform operations in order to check for equivalence. The shell script can found in its raw form in the Appendix to understand how it orchestrates the test execution.

4.4 Chapter Summary

To summarize the chapter, we present a novel approach to equivalence checking in the form of an open source utility called FuzzDiff. Based on the functional requirements and criteria of equivalence, we design a tool that attempts to address the research question which is Can property-based testing techniques involving random sampling or fuzzing be used to identify program equivalence of two methods. Based on the proposed implementation of FuzzDiff, the answer to this question is perspicuously yes. The prototype design satisfies all the functional requirements of the tool, and proves the equivalence or in-equivalence of two Java programs. Hence, it can be considered as a viable solution to the problem statement. The design and implementation of FuzzDiff comes with its set of flaws and limitations, however, these shortcomings can be addressed in the next versions of the product. In the next chapter, we look to answer the second research question which is determining the level to which it will be effective in real-world scenarios.

Chapter 5

Evaluation

In this section, we attempt to answer the final research question of this study i.e to what level is the FuzzDiff effective on real-world programs. The effectiveness of an equivalence checker can constructively be measured by its accuracy to judge the correct output of existing labelled dataset of java programs. Other way would be to exhaustively test the tool on open source softwares and observe if it can identify existing bugs in the programs. The latter approach was used by creators of JQF to evaluate the tool and it was successful in discovering 42 bugs in widely used open source softwares like OpenJDK, Apache Maven, Apache Commons etc. However, for FuzzDiff, we use the former approach for evaluation mainly because it is still in the prototype phase and comes with certain limitations in its usage.

5.1 Methodology

In this section, we describe the methodology used for evaluating the effectiveness of the tool on real-world Java programs.

5.1.1 ARDiff Benchmark

We first test the tool on the benchmarks created by the creators of ARDiff, which was eventually used for the evaluation of ARDiff. The dataset [5] contains 12 benchmarks, each containing multiple examples of original and refactored Java programs. Each example contains two pair of programs, one pair is labelled equivalent and other pair is labelled as in-equivalent. Both original and refactored program contains one public method with same signature and return type, and perform a set of operations on the input to give an output. Most programs also serve as quality test data for testing the effectiveness of the semantic checker as they contain invocations to other private methods in the same class.

On the flip side, none of the programs maintain a global state and store data only in local variables.

Nevertheless, the dataset still remains an ideal choice for evaluation of FuzzDiff as it provides real-life examples of refactoring and they can cover majority of the verification tasks defined in the fuzz testing. Moreover, the results can eventually be compared to that of ARDiff.

For each of the 73 examples of original and refactored pair of programs, we first identify the class names and method name to be given as input to FuzzDiff. Next, we makes changes to the test driver based on the type and number of method parameters. Finally, we execute the run.sh script providing the required inputs and then storing the output of the execution in a table. This experiment is carried out for all the 73 examples. During the experiment, the use of Generic Generator was not required as the input type for all programs was either primitive or wrapper class. Table 5.1 and Table 5.2 encapsulates the result of the simulations described above.

Result

It can be inferred from the table that out of 73 equivalent programs, 69 programs (95%) were rightly classified as equivalent, while 100% of the programs were rightly detected as in-equivalent. As per the definition of equivalency followed by FuzzDiff, the 4 equivalent programs which were identified as in-equivalent by FuzzDiff were found to be wrongly classified as equivalent. Out of those four programs, method "bessy1" was found to be inequivalent because the final output was not same during fuzz testing. A difference was found in the output value returned by the two programs in terms of the double precision accounted by the number of decimal places. The decimal precision of the double value returned by the original program was found to be one less than that of the refactored program. This can prove to be a bug if the refactored method is used for comparison in production systems. The other 3 methods namely "gser", "gamnnln" and "ran(0)" were evaluated by FuzzDiff as in-equivalent because the input state after execution was found to be unequal for the original and refactored program. It could be the case that these three programs were wrongly classified because ARDiff [12] followed a different definition of equivalency.

Benchmark	**Method**	**Output**	**Reason (if Inequivalent)**
ModDiff	Const	Equivalent	NA
	Add	Equivalent	NA
	Sub	Equivalent	NA
	Comp	Equivalent	NA
	LoopSub	Equivalent	NA
	UnchLoop	Equivalent	NA
	LoopMul2	Equivalent	NA
	LoopMul5	Equivalent	NA
	LoopMul10	Equivalent	NA
	LoopMul15	Equivalent	NA
	LoopMul20	Equivalent	NA
	LoopUnrch2	Equivalent	NA
	LoopUnrch5	Equivalent	NA
	LoopUnrch10	Equivalent	NA
	LoopUnrch15	Equivalent	NA
	LoopUnrch20	Equivalent	NA
Airy	max	Equivalent	NA
	sign	Equivalent	NA
Bess	bessi	Equivalent	NA
	bessi0	Equivalent	NA
	bessi1	Equivalent	NA
	bessj	Equivalent	NA
	bessj0	Equivalent	NA
	bessj1	Equivalent	NA
	bessk	Equivalent	NA
	bessk0	Equivalent	NA
	bessk1	Equivalent	NA
	bessy	Equivalent	NA
	bessy0	Equivalent	NA
	bessy1	**Inequivalent**	Final output is not same (difference in decimal points)
	dawson	Equivalent	NA
	probks	Equivalent	NA
	pythag	Equivalent	NA
	sign	Equivalent	NA
	sqr	Equivalent	NA
Caldat	badluk	Equivalent	NA
	julday	Equivalent	NA
dart	test	Equivalent	NA

Table 5.1: Results of experiment on different benchmarks used by ARDiff

Benchmark	**Method**	**Output**	**Reason (if Inequivalent)**
Ell	zbrent	Equivalent	NA
	brent	Equivalent	NA
	dbrent	Equivalent	NA
	rj	Equivalent	NA
	rf	Equivalent	NA
	rd	Equivalent	NA
	rc	Equivalent	NA
	plgndr	Equivalent	NA
	ell	Equivalent	NA
	ellpi	Equivalent	NA
Gam	betacf	Equivalent	NA
	betai	Equivalent	NA
	ei	Equivalent	NA
	erfcc	Equivalent	NA
	expint	Equivalent	NA
	gammp	Equivalent	NA
	gammq	Equivalent	NA
	gcf	Equivalent	NA
	gser	**Inequivalent**	The state of input is unequal after execution and order of method calls is not same
power	test	Equivalent	NA
Ran	bnldev	Equivalent	NA
	ran(1)	Equivalent	NA
	gasdev	Equivalent	NA
	poidev	Equivalent	NA
	gammln	**Inequivalent**	The state of input is unequal after execution
	expdev	Equivalent	NA
	ran(0)	**Inequivalent**	The state of input is unequal after execution
	gamdev	Equivalent	NA
sine	mysine	Equivalent	NA
tcas	altseptest	Equivalent	NA
	NonCrossingBiased Climb	Equivalent	NA
	NonCrossingBiased Descend	Equivalent	NA
tsafe	conflict	Equivalent	NA
	snippet	Equivalent	NA
	normAngle	Equivalent	NA

Table 5.2: Experiment on different benchmarks used by ARDiff

5.1.2 Hobbit test programs

As part of our second set of experiments, we evaluate FuzzDiff on a test-suite of 12 equivalent and 5 in-equivalent pair of programs originally used by an equivalence checker called Hobbit [21] for its evaluation. The dataset is readily available on Github [26] and proves to be an ideal choice for the evaluation of FuzzDiff because of the wide variety of programs it comprises of. Majority of programs maintain a global state, which allows us to test almost every verification task in FuzzDiff. However, one constraint of using using this dataset is that the programs are written in OCaml, which is a statically typed functional programming language with substantially different verbose compared to Java. Each test program is a .bils file that represents a pair of original and refactored program separated by "*jjj*" string. Since the programs were written in OCaml and FuzzDiff only supports Java, a fraction of the 108 equivalent and 68 inequivalent programs were manually translated to their Java counterparts.

The manual translation of the programs from OCaml to Java was done with reference to the official documentation provided by the OCaml Platform [1]. The dataset contains programs of varying complexity, however, for evaluation, we only consider simple OCaml programs that can be easily translated to Java. After translation, we follow similar process of evaluation as done on the ARDiff dataset. Few of the programs contained recursive operations on integers, as a result of which the input size was restricted from -10 to 10 or 0 to 10. This allowed us to utilize the custom options feature of FuzzDiff. The result of the experiments on the 12 equivalent and 5 in-equivalent pair of programs are presented in Table 5.3 and Table 5.4 respectively.

Result

It can be deduced from the two tables that only 1 out of the 12 equivalent programs was correctly classified as equivalent by FuzzDiff. The other 11 were found to be in-equivalent due to a variety of reasons including inequality in number of method invocations, dissimilarity in global state and inequity of final output after execution. On the contrary, all the 5 in-equivalent programs were correctly identified as in-equivalent, with majority of the executions failing due to inequity of final output. The positive inference from this experiment is that FuzzDiff was able to test all possible verification tasks and was subjected to all assertion failures. The negative inference is in fact the inaccuracy in the identifying equivalence. But the lack of correctness could possibly be justified by the assumptions made during translation from OCaml to Java.

Original Program	**Refactored Program**	**Output**	**Reason (if In-equivalent)**
Arrays	Arrays1	Inequivalent	Number of method invocations are not equal
Mult	Mult1	Inequivalent	Number of method invocations are not equal
Cell1	Cell11	Inequivalent	Number of global fields are not same
Cell11	Cell21	Inequivalent	Final output is not equal
Cell11	Cell41	Inequivalent	Global state is not same
Counter	Counter1	Inequivalent	Global state is not same
CounterV2	CounterV21	Inequivalent	Global state is not same
Mccarthy	Mccarthy1	Inequivalent	Number of method invocations are not equal
RecurFact	RecurFact1	Inequivalent	Final Output is not equal
Swap	Swap1	Inequivalent	The state of input is unequal after execution
TakeuchiKnuth	TakeuchiKnuth1	Inequivalent	Final Output is not equal for inputs b/w -100 and 100
Trivial	Trivial1	**Equivalent**	

Table 5.3: Results of experiment on equivalent programs used by Hobbit

Original Program	**Refactored Program**	**Output**	**Reason (if In-equivalent)**
ReveAckermannIneq	ReveAckermannIneq1	**Inequivalent**	Final output is not equal for inputs b/w 0 to 10
ReveAddHornIneq	ReveAddHornIneq1	**Inequivalent**	Final output is not equal for inputs b/w -10 to 10
ReveInliningIneq	ReveInliningIneq1	**Inequivalent**	Final output is not equal for inputs b/w -10 to 10
Cell11	Cell21	**Inequivalent**	Global state is not same after execution
Cell11	Cell41	**Inequivalent**	Final output is not not equal

Table 5.4: Results of experiment on in-equivalent programs used by Hobbit

5.2 Discussion

The experiments performed on the two datasets show that FuzzDiff, as an utility, can be conveniently used to test equivalaence for any given pair of program. FuzzDiff's easy usage and setup makes it favourable for any programmer to readily use it in day-to-day refactoring tasks. In terms of its effectiveness as an equivalence checker, it is still subjective to the type and complexity of programs. It proved to be very accurate in identifying equivalency in ARDiff benchmarks [5] but performed equally poor on the Hobbit test programs [26]. The two datasets were chosen because of their compatibility with the requirements of the tool. Program equivalence checkers follows varying set of rules and principles of equivalency, which makes it further difficult to curate a universal dataset that would meet the requirement of newly born equivalence checkers. Hence, finding the right dataset for evaluation could be a challenging task in the field of program verification.

Hobbit test programs further required program translation but the experiments showed that FuzzDiff can prove to be inaccurate in determining functional equivalency. However, on the flip side, it was able to identify a bug in one of the wrongly classified equivalent methods in the ARDiff dataset, which shows the effectiveness of fuzz testing and its capabilities in finding smallest of differences in behaviour of two given programs. It further shows its potential if subjected to real-world refactoring tasks.

We evaluate FuzzDiff on the two datasets using the default Zest algorithm configured in JQF. It will be interesting to see how well FuzzDiff performs on other coverage-guided fuzzing algorithms like AFL [8], PerfFuzz [24] for the same test programs.

Chapter 6

Conclusions & Future Work

6.1 Conclusion

In this study, we aim to examine the possibility of proving functional equivalency of two programs using fuzz testing and structural analysis. To achieve this, we attempt to build a program equivalence checker called FuzzDiff, which is based on JQF's feedback-directed fuzzing mechanism. It uses property-based testing technique and semantic analysis to verify if two Java programs are equivalent before and after refactoring. We first provide some background and motivation of the study, framing the research question on the basis of the hypotheses. We then expand on the problem statement, covering several formal definitions and discussing various state-of-the-art equivalence checkers like PatEC [17], PEQTest [19], ARDiff [12], SymDiff [22] and Hobbit [26]. We then throw some light on the underlying working of the two frameworks JQF [30] and Junit QuickCheck [4] and describe the functioning of fuzzy generators. A Generic generator is designed and implemented to address the shortcomings of QuickCheck based generators.

We then devise a novel approach of checking two Java programs for equivalency by defining a set of functional requirements and the success criteria for equivalency. Considering all the requirements, we propose the design of the prototype tool by describing in detail the several components involved in the architecture. We then discuss the implementation of the utility in detail and take a deep dive into the program verification process. The outlined limitations and drawbacks of the tool tell that FuzzDiff is still a work in process and requires several enhancements in terms of its implementation in order to provide support to wide range of applications. The assertions are tightly-coupled to certain type of programs and the tool requires manual intervention in certain scenarios. Yet, FuzzDiff, in its prototype phase, proves to be a viable solution to tackle most program equivalence

tasks.

Finally, we evaluate the tool on two benchmarks, one facilitated by the creators of ARDiff [5] and other by the creators of Hobbit [21]. We test a total of 73 pair of test programs in the ARDiff test suite, and 17 pairs in the hobbit test programs. We manually translate the 17 OCaml programs to Java for performing the evaluation on FuzzDiff. We then report the result of the evaluation which show that FuzzDiff performs better on the ARDIff dataset and equally poor on the OCaml. Moreover, it was able to identify a misclassified test program in the dataset due to rigorous random sampling of inputs. The effectiveness of the tool in solving program equivalence is still subjective to the type and complexity of the input programs. The experiments and inferred results may not be the ideal metric for judging the ability of the tool, but it proves to be effective in most day-to-day refactoring scenarios.

With this study, we answer all the research questions raised at the beginning and conclude it by correctly verifying the hypotheses. The promising results show that the tool is headed in the right direction and requires further development to achieve the aim with which it was originally designed for.

6.2 Future Work

In this section, we give a brief overview of the limitations of FuzzDiff and discuss its future scope in context to improvements which can improve its usability and efficiency. FuzzDiff is a prototype tool in its first version, and hence few compromises were made during the development phase in terms of its usability and compatibility to achieve the research goal. The tool shows great potential and can be further improved to make more capable and user-friendly.

The proposed Generic Generator comes with its own set of limitations in terms of the data types it can support. Hence, it can be made further generic to support Collection data structures like ArrayList, Hashset, LinkedList by integrating readily available QuickCheck Generators. Further, code changes are required if a parameterized constructor is used in the test class. These manual changes are mainly due to dependency on Java reflection API which needs to be reduced in general. The error handling can be improved in the Generic generator as well in the future.

As future work, further assertions and conditions can also be added to cover all possible

scenarios and add complexity in equivalence checking mechanism. Semantic checking can be developed further by conducting a more in-depth structural analysis of the method invocations in the test method. For instance, the parameter values of invoked method can be verified for both the programs and those values can be shown to the user in case the order of the invocations is not same.

There are several ways of writing a code with an object oriented programming language and Java being statically typed makes it further difficult to fully generalize and automate the verification tasks. And hence, there are a lot of instances where the user is expected to make changes manually in the code. However, based on the requirements of the user, the testing can be fully automated by dynamically generating the code using tools like ASM, javaassist, bcel etc.

In terms of support for Java programs, the tool can be instrumented further to be compatible with higher-order programs, taking into consideration object oriented principles like encapsulation, abstraction, polymorphism and inheritance.

In terms of usage, the tool can be further enhanced to provide even better user interface to user. The user can be provided with more options for tuning the fuzzing like execution time, number of executions etc. Additional flexibility can be provided by allowing the user to decide if they want either fuzz checking or semantic checking or both. The resultant output can be intuitively presented to the user by parsing the output trace and showing only relevant information if case of failures, helping developer identify the bug easily.

Bibliography

[1] Chapter 9 the ocaml language. https://v2.ocaml.org/manual/expr.html. Accessed: 2022-07-03.

[2] Llvm compiler infrastructure. 2016. libfuzzer. https://llvm.org/docs/LibFuzzer.html. Accessed: 2022-08-01.

[3] Ow2 consortium. 2018. objectweb asm. https://asm.ow2.io/. Accessed: 2022-08-03.

[4] Paul R. Holser, Jr. 2020 junit-quickcheck: Property-based testing, junit-style. https://pholser.github.io/junit-quickcheck/site/1.0. Accessed: 2022-05-03.

[5] The reliable, secure, and sustainable software lab - ardiff. https://github.com/resess/ARDiff/tree/main/benchmarks.

[6] Swen90006 software security testing. https://swen90006.github.io/notes/Coverage-Guided-Fuzzing.html. Accessed: 2022-08-03.

[7] M. Z. 2014. American fuzzy lop. https://lcamtuf.coredump.cx/afl/. Accessed: 2022-08-03.

[8] J. J. 2018. Binary rewriting approach [...] to fuzz java applications with afl-fuzz. https://github.com/Barro/java-afl. Accessed: 2022-08-01.

[9] C. G. 2019. How much time do developers spend actually writing code? https://thenewstack.io/how-much-time-do-developers-spend-actually-writing-code/. Accessed: 2022-08-03.

[10] M. Abadi, S. Barner, D. Pidan, and T. Veksler. Verifying parallel code after refactoring using equivalence checking. *International Journal of Parallel Programming*, 47:59–73, 2017.

[11] M. Alpuente. *Logic-Based Program Synthesis and Transformation: 20th International Symposium, LOPSTR 2010, Hagenberg, Austria, July 23-25, 2010, Revised Selected Papers*, volume 6564. Springer, 2011.

[12] S. Badihi, F. Akinotcho, Y. Li, and J. Rubin. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 13–24, 2020.

[13] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects*, pages 364–387. Springer, 2005.

[14] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In *International Conference on Computer Aided Veri cation*, pages 184–190. Springer, 2011.

[15] B. Churchill, O. Padon, R. Sharma, and A. Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1027–1040, 2019.

[16] K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fth ACM SIGPLAN international conference on Functional programming*, pages 268–279, 2000.

[17] M.-C. Jakobs. Patec: pattern-based equivalence checking. In *International Symposium on Model Checking Software*, pages 120–139. Springer, 2021.

[18] M.-C. Jakobs. Peqcheck: localized and context-aware checking of functional equivalence. In *2021 IEEE/ACM 9th International Conference on Formal Methods in Software Engineering (FormaliSE)*, pages 130–140. IEEE, 2021.

[19] M.-C. Jakobs and M. Wiesner. Peqtest: Testing functional equivalence. In *International Conference on Fundamental Approaches to Software Engineering*, pages 184–204. Springer, Cham, 2022.

[20] R. Kersten, K. Luckow, and C. S. Păsăreanu. Poster: Afl-based fuzzing for java with kelinci. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2511–2513, 2017.

[21] V. Koutavas, Y.-Y. Lin, and N. Tzevelekos. Hobbit: A tool for contextual equivalence checking using bisimulation up-to techniques.

[22] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Veri cation*, pages 712–717. Springer, 2012.

[23] L. Lampropoulos, M. Hicks, and B. C. Pierce. Coverage guided, property based testing. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.

[24] C. Lemieux, R. Padhye, K. Sen, and D. Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 254–265, 2018.

[25] C. Liao, P.-H. Lin, J. Asplund, M. Schordan, and I. Karlin. Dataracebench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[26] Y.-Y. Lin. Hobbit. `https://github.com/LaifsV1/Hobbit`.

[27] G. C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83–94, 2000.

[28] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, pages 815–816, 2007.

[29] R. Padhye. Jqf. `https://github.com/rohanpadhye/JQF`.

[30] R. Padhye, C. Lemieux, and K. Sen. Jqf: Coverage-guided property-based testing in java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 398–401, 2019.

[31] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 329–340, 2019.

[32] D. A. Ramos and D. R. Engler. Practical, low-effort equivalence verification of real code. In *International Conference on Computer Aided Veri cation*, pages 669–685. Springer, 2011.

[33] S. Reddy, C. Lemieux, R. Padhye, and K. Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1410–1421. IEEE, 2020.

Appendix

Here, we show the shell script that triggers the start of FuzzDiff. It seeks input from the user and sequentially executes a number of maven commands to provide user with a output indicating if the two input programs are equivalent or not.

```
#!/bin/bash
echo "Original class name: "
read original
echo ""

echo "Refactored class name: "
read refactored
echo ""

echo "Method name: "
read method_name
echo ""

echo "Fuzzing....."
OP=$(mvn exec:java -Dexec.args="$original $refactored $method_name")
echo ""

#echo "$OP"
sleep 5

SUB="ASSERTION FAILURES"
MNF="METHOD NOT FOUND"
CNF="CLASS NOT FOUND"
FAIL="BUILD FAILURE"
```

```
if [[ "$OP" == "$CNF" ]]; then
  echo "-------------------------------------------------------------"
  echo "Error: One or both class(es) not found in classes/ directory"
  echo "-------------------------------------------------------------"
  sleep 10
  exit
fi

if [[ "$OP" == "$MNF" ]]; then
  echo "-------------------------------------------------------------"
  echo "Error: Method $method_name not found in one or both classes"
  echo "-------------------------------------------------------------"
  sleep 10
  exit
fi

if [[ "$OP" == "$SUB" ]]; then
  echo "-----------------------------------------"
  echo "RESULT: The two programs are not equivalent."
  echo "-----------------------------------------"
  echo "There were failures in fuzz testing.
           Reproducing failures and generating coverage file ....."
  mvn jqf:repro -Doriginal=$original -Drefactored=$refactored
                -DmethodName=$method_name
                -Dclass=ProgramEquivalenceTest
                -Dmethod=fuzzTestForEquivalence
                -Dinput=target/fuzz-results/ProgramEquivalenceTest
                        /fuzzTestForEquivalence/failures
                -DlogCoverage=coverage.txt -DprintArgs
                -DdumpArgs=result/
  sleep 20
  exit
else
  echo "There were no failures in fuzz testing.
           Reproducing results and generating coverage file ....."
  COV=$(mvn jqf:repro -Doriginal=$original -Drefactored=$refactored
                -DmethodName=$method_name
                -Dclass=ProgramEquivalenceTest
                -Dmethod=fuzzTestForEquivalence
```

```
                -Dinput=target/fuzz-results/ProgramEquivalenceTest
                        /fuzzTestForEquivalence/corpus
                -DlogCoverage=coverage.txt -DprintArgs
                -DdumpArgs=result/)
fi
echo ""

sleep 5
echo "Executing semantic tests ..............."
echo ""

RESULT=$(mvn test -Dtest="AdditionalChecksTest")

SUC="SUCCESS"

echo "-----------------------------------------"
if [[ "$RESULT" == *"$SUC"* ]]; then
  echo "RESULT: The two programs are equivalent."
  echo "-----------------------------------------"
else
  echo "RESULT: The two programs are not equivalent."
  echo "-----------------------------------------"
  echo "$RESULT"
fi

sleep 120
```