Automated Test Generation through Runtime Execution Trace Analysis

Thomas Rooney
Imperial College London
tr111@ic.ac.uk

Department of Computing

Imperial College London

Supervisor: Alastair Donaldson
Second Marker: Sophia Drossopoulou

Abstract

The trend for software systems is one of increasing complexity [1]. One of the ways software engineers manage this complexity is the practice of software testing: it is estimated that as much as 50% of software development time is spent on software testing [2]. This high cost has led to many attempts to *automate* testing. Despite this, attempts to transfer automated test generation tools from research to practice have posed difficulty [3]. One of the reasons identified for the slow uptake is that test inputs generated are not representative values for what occurs during practical usage of an application.

This report documents the development of a system that enables the automatic generation of unit tests to detect regression, directly utilising information gathered during actual use of an application through instrumentation. As such, test input values become representative of what occurred during practical usage. Our tool is targeted at the Java language, and generates tests for the popular testing and mocking frameworks *JUnit* and *Mockito*. Our tool needs no user guidance – the inputs it requires are the program to test, and an entrypoint and arguments to run it with.

We evaluate our tool with a wide range of open source examples, and show that our tool is capable of consistently achieving similar code coverage in its generated tests to the code coverage in the exercised program. We demonstrate that we can generate tests within a *practical* time period. We provide reasoning and data to show that the generated tests can run at greater speed than the original entrypoint; tests can be run in parallel due to the abstraction of external resources and internal interfaces into *mock* objects. Finally, we explore several use cases for which our technique can be applied to today to provide useful value and aid the diagnosis of bugs.

Acknowledgements

I would like to thank my supervisor, Dr. Alastair Donaldson, for taking time out of his busy schedule to discuss this project in our meetings; providing support, motivation and guidance throughout. He has constantly remained confident in my ability to deliver, and provided ideas, advice and constructive criticism. I would like to express my gratitude to Dr. Sophia Drossopoulou for asking the important and hard questions, and providing guidance on how best to move forward. Finally I would like to thank my friends and family for their ongoing support throughout my four years at Imperial College.

CONTENTS

I Introduction 6

I-A Demonstration 7

I-B Core Contributions 12

I-C Report Outline 12

II Background 14

II-A Object Oriented Programming 14

II-B Java 15

II-C JVM 16

II-D How a debugger works 17

II-D1 Debugging Native Code (C/C++) 18

II-D2 Debugging Java 19

II-E Software Testing 20

II-F Automatic Test Generation 21

II-G Deterministic Execution 23

II-H Eliminating non-determinism in system calls 25

II-I Mocking 25

II-J Related Work 28

II-J1 AgitarOne 28

II-J2 CodePro AnalytiX 28

II-J3 Randoop (aka Palus) 28

II-J4 Daikon 29

II-J5 Test Factoring 29

II-J6 Alternative Tracing Techniques 29

III Implementation 30

III-A Requirements 30

III-B Overview 31

III-C Recorder Module :: Trace Collection 31

III-C1 Architecture 32

III-C2 Java Platform Debugger Architecture (JPDA) 33
III-C3 Lifecycle 35
III-C4 Following Control Flow 36
III-C5 Usage 37
III-D Storage Module :: Graph Database 38
III-E Generation Module :: Test Code Generation 39
III-E1 Architecture 39
III-E2 Neo4j Graph Reader 41
III-E3 Call Graph Partitioner 41
III-F Static Analysis 42
III-F1 Sub Graph Analysis and Filtering 44
III-F2 Building Unit Tests 44
III-F3 JCodeModel 45
III-F4 Naming Variables 45
III-F5 Arrays 46
III-F6 Primitives and Immutable Objects 46
III-F7 Mocking Complex Objects 47
III-F8 Usage 47
III-G Implementation Walkthrough by Example : ABC 48
III-H Usage Walkthrough by Example 53

IV Limitations and Utility 56
IV-A Limitations 57
IV-A1 False Positives and False Negatives: Brittleness 57
IV-B Utility 57
IV-B1 Additional Information for Failure Diagnosis 58
IV-B2 Performance Improvements in Integration Test Suites 58
IV-B3 Bootstrapping manual test production 58
IV-B4 Automating UI Testing 58

V Evaluation 59
V-A Test Configuration 59
V-B Benchmark Suite 59
V-C Coverage Evaluation 60
V-C1 Results 61
V-C2 Analysis 61
V-C3 Validity of Results 62
V-D Performance Evaluation 63
V-D1 Results 63
V-D2 Analysis 64
V-D3 Abnormalities 65
V-E Further Experiments 65

VI Future Work 65
VI-A Test Trees 66
VI-B Generating tests for testing validity of mock objects. 70

VII Conclusion 71

VIII Appendix 72
VIII-A Demonstration 1 :: Main 72
VIII-B The execution time of the `Recorder` module without tracing. 72
VIII-C Expression Evaluation Example 72
VIII-D Generating our own test suite 79
VIII-E Coverage Data Tables 80
VIII-F Performance Data Tables 83
VIII-G Process Interaction 86

I. Introduction

In the 1970s, it was observed by Lehman and Belady that real world software follows several laws as it changes over time. Amongst others, they include that *"software must be continuously adapted or it becomes progressively less satisfactory"*, and *"as software is adapted, its complexity increases unless work is done to maintain or reduce it"* [4]. These two laws have been empirically evaluated multiple times [5] [6], and look to remain supported in the near future [1].

This complexity increase has a very noticeable effect on the economics of software development. As much as 50% of time [2] is spent in the QA and testing phase of software development, and the cost of discovering and fixing a bug increases with the size and complexity of an application [7]. To mitigate this, software developers have a wide range of techniques, ranging from formalised *verification*[1] of software through to the *test suite*. A software system's *test suite* relates to the set of programs written to verify that the software system has the intended behaviour. These can either test smaller software components or perform larger, system level behaviour tests. This project is focused on the augmentation of the project's *test suite*.

It is desirable for a test suite to contain small, fast, deterministic and focused[2] tests [8]. By having small and focused tests, when a problem is detected, the potential causes are isolated to a small number of components, speeding up diagnosis of the problem. A Unit Test is one example of this kind of test. A Unit Test exercises and tests the behaviour of a small number of components, such as a single class or method. It is often the case that the method's dependencies are replaced with simulated versions (otherwise known as *Mock Objects* discussed in Section II-I), which isolate the verification of the component from those dependencies.

Mock Objects enable isolation from different components and external interactions, meaning unit tests can be run in parallel. Their use also isolates errors to a small amount of code, easing the debugging process when they fail by focusing a developer's attention to a small set of places that can be the source of the error.

In industry, such focused tests are not always available. It is often the case that the focus on testing and long term reliability is sacrificed to ensure business objectives are met. Unit tests can take significant time to write[3] – they might be too expensive. Instead, more overarching tests or procedures might be used. For example, a development team might deploy their code to a *staging area*, where it would be manually tested to verify correctness of behaviour, before forwarding to production. A more automated example might be *system* or *integration*[4] tests – end-to-end tests that exercise a significant part of the functionality of the entire system. Because they test larger portions of the system, there are often fewer of them, and they are generally easier to manage and maintain. They are less brittle to software changes, for example changes in internal interfaces. They also tend to be more comprehensive, by covering more code and its interaction in synchrony. However they take longer to run, and do not ease debugging in

[1] Formal verification relates to the use of mathematical proofs to verify that software works as desired, directly showing a mapping between the specification of behaviour desired, and the implementation of the software used to provide this behaviour.

[2] A Focused test is a test that verifies the behaviour of a small module – for instance a single method call.

[3] In industrial case studies, it has been evaluated that writing unit tests costs a ~15-35% extra upfront developer time cost for writing the tests. It has also been shown this process does cause fewer bugs to go to production. See [9] for more details.

[4] The term *system test* or *integration test* are often used interchangeably to mean the same thing – larger tests which bring together and test multiple components

the way that a unit test does – when a system test fails, the number of possible components which are the cause of this failure is significantly larger than an equivalent failure in a unit test.

Because unit test suites are expensive to build manually, it would be great if we could generate them automatically. However, *The Oracle Problem* (Section II-F) poses a fundamental issue to the generation of testing code: "*how might a machine distinguish between working and non-working code*". We sidestep this problem by asserting that one version of a program "works", and changes on this version that break previously made internal class interactions (Section II-A) are "non-working" code.

Such tests would become *regression* tests – they would not look at desired behaviour of a program (which would require some form of *specification*), but at regressing *changes* in runtime behaviour, within the same code paths that an execution of a previous version walked through. As at least 15-25% of software updates to fix bugs are implemented incorrectly and cause issues to end users [10], this would still have significant benefit.

The execution of an application will cause a sequence of interactions with smaller components when it is run. These sequences of interactions can be collected into what we call an *execution trace*. This *execution trace* can be filtered to look at the interactions with a single object. We postulate that a "*successful*"[5] integration test run is equal in meaning to the associated execution traces' *interactions* with its objects also being *successful*, or correct, and from these interactions, we postulate that we can build a series of unit tests that replay the *interactions* with those objects in such a way that it *verifies* the behaviour of each object alone.

The core objective of this project is to investigate and implement the automatic generation of unit tests in Java with the assumptions defined above. We investigate whether the generated unit tests provide more information (i.e. location and test data input of a method) on the cause of a regression than the original process which allowed us to generate the tests. We demonstrate how our tool scales (Section V-D), and the utility of its generated tests (Section IV-B). By doing this, we investigate and show that we can provide some of the benefits of focused and isolated unit tests without as significant a development time cost.

We make the assumption that we have the ability to exercise a correctly running system, either by manual testing or integration test suites. We do not aim for *soundness* of tests produced – each test may "Pass" or "Fail" for a detected change in behaviour without that behaviour being necessarily a bug or its lack. Instead we aim to generate tests similar to how developers would write them themselves. We attempt to provide a system which is directly applicable to a number of use cases, and we describe these in Section IV-B. We provide a simple use case below, to illustrate the core behaviour of our application.

A. Demonstration

In this example we aim to show the use of the tool in producing a unit test for a database interaction. This test would abstract away the database, and allow for the logic to be tested *without* actually connecting to the database itself. In doing so, the test becomes holistic – it only needs itself and the program under test to run. As such, the execution can be reproduced on any machine, not just one with a configured database.

[5]By *successful*, we mean the *"pass"* result (as opposed to *"fail"*) that occurs running a set of verifications on the result of running the integration test.

In Listing 1, the `DataAccessObject` class is interacted with via the `getUserDetails` method. This method performs a SQL query with a SQL Connection object, and returns the result wrapped up in `User` object. The SQL `Connection` object contains underlying logic that interacts with the database itself, but this is hidden behind the `Connection` interface.

```
1  public class DataAccessObject {
2      private final Connection connection;
3
4      public DataAccessObject(Connection connection) {
5          this.connection = connection;
6      }
7
8      public User getUserDetails(String person) throws SQLException,
9                                                       DAOException {
10         PreparedStatement stmt = connection.prepareStatement(
11             "SELECT name, occupation FROM people WHERE name = ?"
12         );
13         stmt.setString(1, person);
14         ResultSet results = stmt.executeQuery();
15         if (results.next()) {
16             return new User(results.getString(1), results.getString(2));
17         } else {
18             throw new DAOException(String.format("No user %s", person));
19         }
20     }
21 }
```

Listing 1. The class for which we aim to generate tests for.

In Listing VIII-A we show our `Main` method, but for the purposes of tracing, we only track its two interactions with this class:

1. The object was constructed, passing in the `Connection` interface which happened to be implemented by an `SQLLiteConnection` object.
2. A single call to the `getUserDetails` method was made with parameter `"Gandhi"`. For this example, we wrap this interaction into a `Main` method, (Listing VIII-A). In a real program, this interaction may occur upon a UI interaction. What triggers the interaction has no meaning to our tool – merely that the component we desire to test is interacted with whilst running under our instrumentation process.

We desire to generate tests for the `getUserDetails` method. As such we have to make the interactions with the external database deterministic (Section II-G). As the database is interacted with through the `Connection`, `PreparedStatement`, and `ResultSet` interfaces (Figure 1), to abstract it away we must *mock*[6] these interfaces (Figure 2), and ensure that the `getUserDetails`'s dependencies are replaced with the *mock* objects.

These *mock* objects are configured with a sequence of *expected* interactions and what it should respond for each of these interactions. They also record *actual* interactions locally, and can provide verification that specific interactions took place as expected after the method is run. This configuration is shown

[6]*Mock* objects and their implementation are expanded upon in Section II-I, and can be understood as a simulated or stubbed version of an object, with hard-coded return values for method calls.

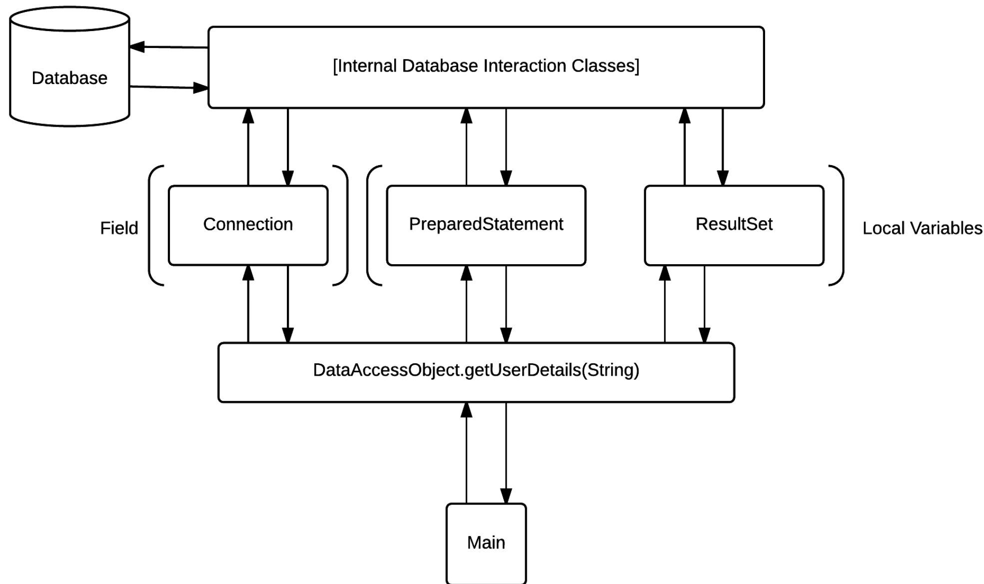

Fig. 1. The object interactions inside `DataAccessObject.getUserDetails(String)`

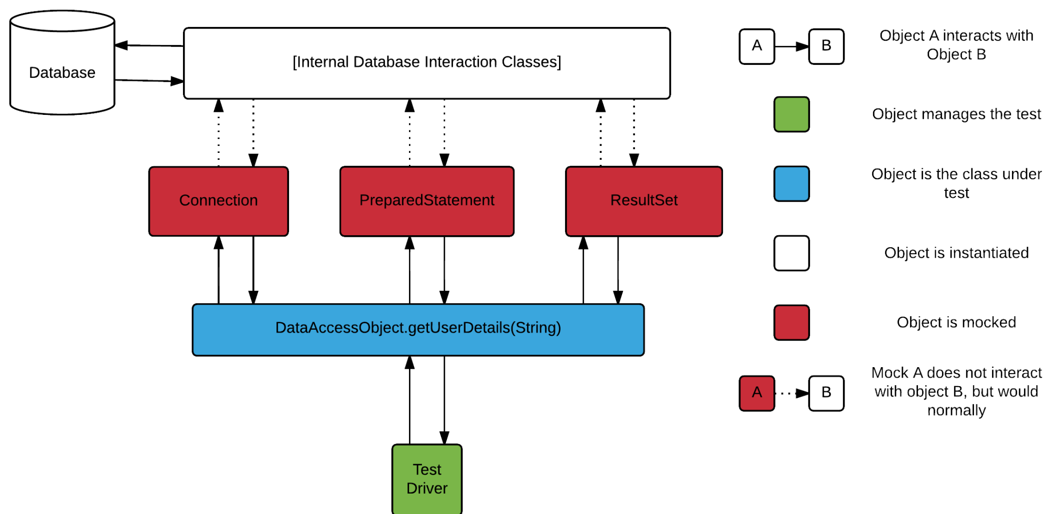

Fig. 2. The mocks necessary to abstract the Database from the `DataAccessObject.getUserDetails(String)` method

in our generated code, **lines 19-23** and lines **31-36**. The *expected* interactions are collected through interception of `MethodEntry` and `MethodExit` events, through the JPDA Debugging API (Section III-C2).

This result of the generation process can be seen in Listing 2. This generated code was produced by our `generator` module Section III-E after tracing the execution of the program with our `recorder` module

(Section III-C).

In the source code, the *static class* **Mockito** refers to an API used for construction and verification of *Mock Objects* – further detail on this can be found at Section II-I.

```
1  public class TestDataAccessObject {
2      @Test
3      public void testGetUserDetails()
4          throws SQLException, DAOException
5      {
6
7          // 1. Generate the fields
8
9          SQLiteConnection mockconnection = Mockito.mock(SQLiteConnection.class);
10         JDBC4PreparedStatement mockstmt = Mockito.mock(JDBC4PreparedStatement.class);
11         JDBC4ResultSet mockJDBC4ResultSet = Mockito.mock(JDBC4ResultSet.class);
12
13         // 2. Construct the DataAccessObject class
14
15         DataAccessObject dataAccessObject = new DataAccessObject(mockconnection);
16
17         // 3. Link Mocks
18
19         Mockito.when(mockstmt.executeQuery()).thenReturn(mockJDBC4ResultSet);
20         Mockito.when(mockJDBC4ResultSet.next()).thenReturn(true);
21         Mockito.when(mockconnection.prepareStatement(Mockito.eq("SELECT name,
       occupation FROM people WHERE name = ?"))).thenReturn(mockstmt);
22         Mockito.when(mockJDBC4ResultSet.getString(Mockito.eq(1))).thenReturn("Gandhi");
23         Mockito.when(mockJDBC4ResultSet.getString(Mockito.eq(2))).thenReturn("politics"
       );
24
25         // 4. Invoke the method
26
27         User user = dataAccessObject.getUserDetails("Gandhi");
28
29         // 5. Assert that our method interactions are what we expect.
30
31         Mockito.verify(mockstmt, Mockito.times(1)).executeQuery();
32         Mockito.verify(mockstmt, Mockito.times(1)).setString(Mockito.eq(1), Mockito.eq(
       "Gandhi"));
33         Mockito.verify(mockJDBC4ResultSet, Mockito.times(1)).getString(Mockito.eq(1));
34         Mockito.verify(mockJDBC4ResultSet, Mockito.times(1)).getString(Mockito.eq(2));
35         Mockito.verify(mockconnection, Mockito.times(1)).prepareStatement(Mockito.eq("
       SELECT name, occupation FROM people WHERE name = ?"));
36         Mockito.verify(mockJDBC4ResultSet, Mockito.times(1)).next();
37     }
38 }
```

Listing 2. The generated test case

The output of the tool in Listing 2 shows five distinct steps in our generated tests.

1. **Generate the fields** We construct the complex objects that the class interacts with *through* a field. In this case, there are three objects – a `Connection`, a `PreparedStatement`, and a `ResultSet`. The `PreparedStatement` and `ResultSet` objects are accessed through interactions with the field `private final Connection connection` in Listing 1.

2. **Construct the DataAccessObject class** We construct the class, passing in the *root mock*[7] – the `Connection` class. We refer to it as the root mock as it is through interactions with this object that the other objects are interacted with.
3. **Link Mocks** We set up each of the mocks' return values. Each operation on an object which returns something must be explicitly configured in the mock. This is done through the Mockito library – a popular mocking framework. Further information on Mocking can be seen in Section II-I. These return values come from the recorded trace information. For primitives, their value is returned explicitly as it occurred in the original program. For objects references, they are returned in the form of an appropriate mock (should it be *interacted with* by this method call).
4. **Invoke the method** We invoke the method, with the same parameters as were initially passed in during the execution of our program.
5. **Assert that our method interactions are what we expect.** We verify that the objects *effects*, in terms of object interactions and return values (the return value is an object in this case, and so there is no verification beyond type), are the same as that under the original execution. The *effects* of a method are explained in further detail in Section II-B.

This test code is similar to what a human might produce for this method [11]. There are several possible reasons we think a human might write this:

- The existence of a test like this allows for one to refactor the implementation of a program whilst maintaining a high level of confidence that it hasn't been broken. For instance, one can imagine that there could exist multiple users in the database with the *same name*. Perhaps a future programmer might want to modify the code such that it uses `Lambda` expressions to simplify the code. In this case, running this test suite and it returning `PASS` would provide confidence that he hasn't broken this use case[8].
- When changing any code across a code base, the running of a large suite of unit tests allow programmers a degree of confidence that their changes haven't broken anything. The mere existence of a test suite with methods like this provides this comfort, especially to new developers, regardless of whether or not it will catch their bugs. This process can generate tests for an entire, large code base with very low effort.
- The writing of this test might constitute a design task – by building up the expected inputs and outputs to the function as a series of tests, the programmer can ensure that the design of his program *makes sense*. Furthermore, this allows for the production of a program in parallel by multiple developers. By stubbing out expected dependencies before they are written, each programmer can work in parallel and test their work before the entire program is written and *system-level* behaviour can be verified. This utility cannot be replicated by our tool, as it requires a *working* system before it can generate the tests.

This consider further applications and limitations of the tests generated in Section IV-A.

[7]We use the term *root mock* to refer to the mock through whose interactions other *mocks* are accessed. For instance, in Listing 1 the `PreparedStatement` is accessed through the `PreparedStatement Connection.prepareStatement (String)` method. Likewise, the `ResultSet` is accessed through the `PreparedStatement.executeQuery()` method. This can be considered a *tree* of mocks, whose nodes are mock objects and children which are the mock objects who are accessed through that given mock.

[8]It is worth noting that common IDE refactoring tools would also modify the existing test case in synchrony with the original code – such that if he were to rename referenced fields or even change the return value many IDE's (Eclipse/IntelliJ) would provide the option of modifying the test suite automatically, or at least make it obvious so as what to change to ensure that the type checks pass

B. Core Contributions

This project makes the following contributions:

- **Collecting Execution Traces** We designed and implemented a tool for collecting execution traces in Java, and building it into an indexed graph. This graph contains the core information for building *regression tests* from captured runtime information. This graph also contains information that would allow one to build other applications for use in software engineering. These other potential applications are described in Section VI.
- **Building Regression Suites** We designed and implemented a tool for building regression suites from the collected execution traces. These test suites take the form of JUnit and Mockito built unit tests in Java. They allow a deterministic and fast running verification that a method interacts with the environment around it in the same way that it did when the tests were generated. The utility of these tests is described in Section V.
- **Challenges and Limitations** We enumerate the engineering challenges we have had to overcome to convert our technique from theory to practice, as well as the some of the limitations we have run into with our design. We do this in hope that we can help provide a solid foundation for those who work on similar projects in the future.
- **Use Case Evaluation** We evaluated our system with a series of walked through examples, describing how it can aid the discovery and reproduction of bugs in a production environment.
- **Performance Evaluation** We evaluated the scalability of our tools by performing a variety of tests on the SF100 corpus [12] as well as a variety of synthetic tests designed to illustrate the strengths and weaknesses of our approach.

C. Report Outline

- Section I contains a brief description of the core problem that we are trying to solve – the improving of developer productivity via the automatic generation of unit tests from integration test suites or manual testing. It also contains a summary of the core contributions of this project and a description of the report outline
- Section II contains the background research on which this project relies. It summarises the current techniques used in software testing, automatic test generation, and debugging, alongside descriptions of some of the core concepts for understanding the implementation of our tool. We describe a variety of the related projects to this work which inspired it and a description of the alternative approaches we could have taken.
- Section III describes the implementation of the application itself, along with the core design decisions that led to its current performance and limitations. Furthermore, we provide a walkthrough use of our system for a simple, synthetic example to demonstrate the data flow and usage of our application.
- Section IV-A describes some of the core limitations of our approach, as well as a description of some of the limitations inherent in our techniques used to implement this approach, under what we define as the theoretically perfect implementation.
- Section V describes the methodology that we used to go about testing the application, and analysis of the results generated. We try to quantitatively evaluate the strengths and weaknesses of our approach on the SF100 test corpus, as well as on custom test cases built to show the core strengths and weaknesses of our implementation.

- Section VI contains our thoughts on other applications of the tools we have implemented here, as well as how they might be evolved to aid a number of other use cases.
- Section VII contains our concluding thoughts on this project.

II. Background

A. Object Oriented Programming

One of the core assumptions made in the technique we've used in this project is that program under test is modularised. What this means is that the way a part of a program interacts with the wider program is separated by explicit interfaces. By making this assumption, we can start overriding these interfaces with fake implementations, which we talk about in Section II-I.

Pure Object Oriented Programming languages provide this isolation guarantee. A pure object oriented programming language is described as a language where everything is treated consistently as an object. An object is a data structure which contains data and code, with programs being constructed as a *web* of connected, communicating objects.

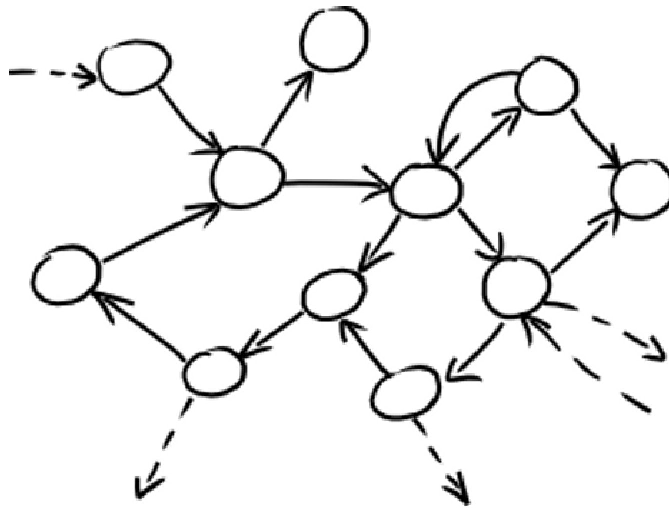

Fig. 3. An object oriented program can be thought of as a web of communicating objects. [Freeman and Pryce [11]].

This is important because to construct our unit tests around an object, we must provide a simulation of the environment around a method call. In an object oriented program, this environment is constrained to mean three things. (1) the **arguments** that are passed to that method call. (2) the objects that this method can access through **fields**. (3) any **global** objects or methods that this method can access.

Through controlling (1) and (2), we can build a test environment for most methods. However the lack of control over (3) means that any test generation tool we build is limited to not being able to handle all possible program statements – which we describe as *tainting* the test. For example, there is no way for the environment around an object to be fully closed when a method creates and then interacts with an object. In Listing 3, the returned value of the method is the result of the `HTTP GET` to `http://ipecho.net/plain`, which is the external IP of the calling machine. Should one want to build a test around this, we would fail to make this run deterministically and without making the call – there is no (easy[9]) way to override the ipEchoStream variable.

[9]Though not really feasible, this could be done with `Agent Libraries`, custom JVMs or Debugger interception methods. These could not be deployed to a developers machine without third party libraries however, so we do not consider them further

```
1  String untestableGetExternalIP() {
2    InputStream ipEchoStream = new URL( "http://ipecho.net/plain" ).openStream();
3    return IOUtils.toString( ipEchoStream );
4  }
```

Listing 3. An example of an untestable method

In [13], Java is described as being an object oriented language, however, unlike languages like Smalltalk it is not considered a true object oriented programming language. For instance its implementation of Arrays and Primitives does not encapsulated them as an object. This yields complexities in the implementation of our test generation tool. Each of these primitive object types must be handled and managed explicitly, in both generation of the source code and data collection.

B. Java

Our tool is targeted at the Java language. A familiarity with Java language syntax is presumed. We enumerate the attributes that led to our decision to use it below:

1. **All programmatic constructs are encapsulated as objects** This means that all logical elements which *control flow*[10] passes through are *methods* in *objects*. There are *almost* no exceptions[11].
2. **Data constructs take the form of objects, primitives, and arrays of [objects or primitives or arrays of [. . .]]** In our implementation, we must store some representation of what *data* and *value* means. This is because we must instantiate objects and pass them into *fields* and *arguments*, and so we have to store data constructs in a way that we can reconstruct them. This means that we need maintain only three[12] internal representations for a Value: `Primitive`, `Array`, `ObjectReference`[13]
3. **Control flow changes on Method Calls, Exceptions, Branches, and Method Returns** A Control flow begins at the start of a Thread – the first of which is instantiated by JVM. From this origin point, there are 3 ways it can change. A Method call will transfer control flow to the callee, and then back again once that callee has finished executing. An Exception will rewind control flow to the a *catch* block, and *branches* determine different code *blocks* to be executed on a condition. To follow control flow through a deterministic execution (Section II-G) , all evaluated *branch points* will take the same choice. As such, when we trace and generate tests for an execution, we must keep track of *Method Calls* and *Exceptions*.
4. **Java Bytecode has a high correspondence with the source code** This is built into the Java specification, because it means that a JVM can give good error messages when something goes wrong (e.g. the Stack Trace). As our tool runs on compiled code this decision means we can map *traced* control flow changes to the original source code. There are a few exceptions to this. The `new` method call is one of them. In Java Bytecode, this is broken down into a **new** instruction

[10]Control flow relates to an ordering of which segment of an application is currently executing – there is a flow of *control* throughout an application as different instructions get executed and affect *state*.

[11]The internals of *native* methods are the main exception, but esoteric examples have been demonstrated which use *unsafe* methods to create and execute arbitrary instructions. These are fortunately rare, and we ignore them as there is no way we think feasible to intercept them.

[12]We internally maintain a fourth – a representation for an `ObjectReference` which `implements Throwable`, i.e. an `Exception`. This simplifies our representation and generation of Mock Objects

[13]though these representations must allow reference to children defined to capture full specification - i.e. fields, supertypes, array elements, etc.

which allocates memory for a new object, and a `<clinit>` and `<init>` method which perform *static initialisation* and call the constructor appropriately.

C. JVM

A JVM is an abstract virtual machine. What this means is that it is a program which is implemented on many platforms, and can run other programs which conform to the JVM's bytecode specification. This bytecode specification is similar to native specifications like X86. However, unlike X86, it is not machine specific, having no concept of a limited register set or machine specific operands.

In this project, what is important are the methods used to collect information from a JVM at runtime, and the effect these methods have on performance. As such, we give a general overview of how the JVM achieves high performance, along with some of the internal details for how objects are created. This provides grounding knowledge for where the limitations of our technique originate from.

At runtime, the JVM can be considered to be made up from a number of application threads, a heap, and a registry of instruction code in the form of Java Bytecode.

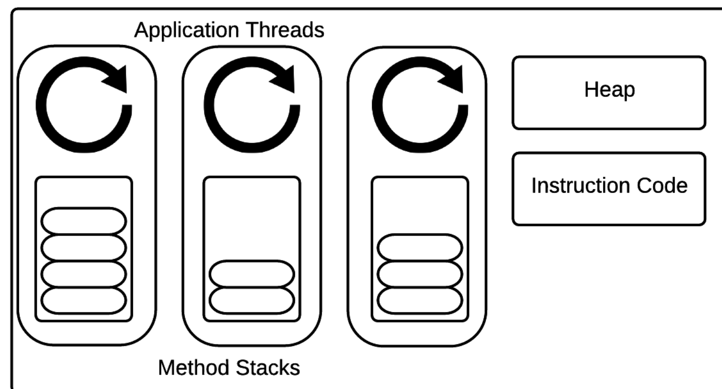

Fig. 4. A simplified runtime structure diagram of the JVM

Java bytecode is arranged into the form of objects, with a stream of instructions per method in that class. This is interpreted and executed when a method is invoked during the execution of the program.

Each of the JVM's stack based instructions consists of a one-byte opcode, followed by zero or more operands. The primitives that can be manipulated with the stack are *byte*, *short*, *integer*, *long*, *float*, *double*, *char*, *object*, and *return address* data types. On top of this, additional primitives in the form of the array form for each of these are defined – each array is **not** wrapped up as an *object*.

Internally, stack variables are contained in *frames* – unique objects on the heap which manage all local variables defined in a method call. The JVM is a stack based machine, therefore all operations on data occur through this stack, whether they are referenced through fields or constant pools.

Memory is dynamically allocated to executing programs from a garbage collected heap using the *new* operator in Java. This *new* operator takes the form of a six step process to creating an object.

1. The class loader is invoked with the object's type signature, called via ClassLoader.loadClass(String typeSignature). This returns a Class object which contains the necessary details of the classes internal structure.
2. The Security manager is invoked to ensure there are no violation in execution constraints.
3. [Optional] The class's `<clinit>` is called, which performs any static initialisation within the class.
4. The class's `<init>` method is invoked. This is the call to the appropriate constructor of the object based on the arguments type signatures.
5. Any superclasses have their `<init>` method called as the first statement in their subtypes `<init>` method[14].
6. The constructor "returns" an instance of a constructed object of the specified type.

After an object is created, it has an associated **uniqueID** (unique identifier) which is assigned to it for the course of the JVM's lifetime. We use this *uniqueID* to map runtime invocations of objects across an execution, along with the **type information** to collect a coherent model of an objects behaviour and functionality. The JVM maintains a high performance by JIT'ing the code with runtime guided optimizations, such as automatic unboxing and inlining techniques [14]. These techniques pose a problem to us when we are performing tracing activities. For instance, one of the core sources of unsoundness in our trace is caused by the automatic inlining of small methods. From our perspective, what appears to happen is that sometimes a method is entered into that has *no* stack frame. This is important to us, because to work out *which* object that was entered into we need to halt the JVM and look at the `this` object's `uniqueID`. Inlined methods are amongst what we denote our **opaque** set of methods. For these we either have to ignore the interaction, or make a guess at which object it was. For instance, we can look through the *type* information of in scope variables, and try to deduce which of these was the one invoked. However this methodology is fundamentally unsound. As such, in many cases we simply tag such an interaction with the *opaque* flag, and give up on this test case.

There are another large set of interactions that we denote as *opaque*. These are the *native method calls*. A Native method call is handled by copying memory across from the JVM to an isolated execution sandbox for the native method call. As such, all JVM functionality to trace information is no longer accessible. However, as methods *into* and *out of* the native execution area occur through a JVM-land wrapper class, a call to a native method does not stop one from generating mocks for these wrapper classes. However it does mean that we cannot generate tests *for* native methods, as we have no knowledge as to what occurs internally.

The APIs which we use to trace JVM object interactions are collectively known as the Java Platform Debug Architecture (JPDA), whose use of which we discuss in Section III-C2. These are not one of the only ways to collect the trace information we need – there are also several other methods that could be used for similar effect. Each technique comes with it a set of limitations which we outline in Section II-J6.

D. How a debugger works

The approach this project takes to collecting traces is highly related to the way in which a debugger is implemented. To capture traces we require gathering information from a running process. As such,

[14]This is actually optional, according to the Java bytecode, but it is defined as part of the Java languages specification, and so non-optional. However it is possible to rewrite a classes bytecode instructions such that its supertypes `<init>` methods are called later, or not at all!

an understanding of how debuggers works is useful to understanding the approach we take and its limitations.

1) Debugging Native Code (C/C++): Native Debuggers generally comprise of a toolchain which embeds additional information into a binary, and a tool that can interpret this information to control execution and read information from the running process.

In Unix based systems, this generally means embedding *DWARF*[15] debugging information into the binary. *DWARF* is a standard language which tags registers and memory locations with source identifiers, such that a debugger can *join the dots* between machine code and source code.

For example, consider a block to add two numbers:

```
[...]
3: int a = 5;
4: int b = 4;
5: int sum = a + b;
[...]
```

When compiled in an unoptimized way, this converts to the assembler:

```
$ gcc test.c -O0 -o test
$ objdump -d test
[...]
movl   $0x5,-0x14(%rbp)
movl   $0x4,-0x18(%rbp)
mov    -0x14(%rbp),%edi
add    -0x18(%rbp),%edi
[...]
```

When compiled with DWARF debugging information, this adds the following file headers

```
$ gcc test.c -O0 -g2 -o test
$ objdump --dwarf=decodedline test
[...]
<1><72>: Abbrev Number: 5 (DW_TAG_subprogram)
<73>   DW_AT_external    : 1
<74>   DW_AT_name        : (indirect string, offset: 0x20): main
<78>   DW_AT_decl_file   : 1
<79>   DW_AT_decl_line   : 2
<7a>   DW_AT_prototyped  : 1
<7b>   DW_AT_type        : <0x57>
<7f>   DW_AT_low_pc      : 0x4004b4
<87>   DW_AT_high_pc     : 0x4004dd
<8f>   DW_AT_frame_base  : 0x0  (location list)
<93>   DW_AT_sibling     : <0xda>
[...]
<2><b3>: Abbrev Number: 7 (DW_TAG_variable)
<b4>   DW_AT_name        : a
<b6>   DW_AT_decl_file   : 1
<b7>   DW_AT_decl_line   : 3
<b8>   DW_AT_type        : <0x57>
<bc>   DW_AT_location    : 2 byte block: 91 64  (DW_OP_fbreg: -28)
<2><bf>: Abbrev Number: 7 (DW_TAG_variable)
<c0>   DW_AT_name        : b
<c2>   DW_AT_decl_file   : 1
```

[15]DWARF is a debugging information format standard, which is formally defined at http://www.dwarfstd.org.

```
25    <c3>   DW_AT_decl_line   : 4
26    <c4>   DW_AT_type        : <0x57>
27    <c8>   DW_AT_location    : 2 byte block: 91 68  (DW_OP_fbreg: -24)
28    <2><cb>: Abbrev Number: 7 (DW_TAG_variable)
29    <cc>   DW_AT_name        : sum
30    <d0>   DW_AT_decl_file   : 1
31    <d1>   DW_AT_decl_line   : 5
32    <d2>   DW_AT_type        : <0x57>
33    <d6>   DW_AT_location    : 2 byte block: 91 6c  (DW_OP_fbreg: -20)
34    [...]
35    CU: test.c:
36    File name                          Line number    Starting address
37    test.c                                       2            0x4004b4
38    test.c                                       3            0x4004bf
39    test.c                                       4            0x4004c6
40    test.c                                       5            0x4004cd
41    test.c                                       6            0x4004d8
42    test.c                                       7            0x4004db
```

With this information, when a debugger is instructed to break execution on a line (e.g. "`b test.c:6`") , it can convert that to the address of the first instruction on that line (`0x4004d8`), and interrupt execution at that address. It would then utilise the DWARF information (`DW_TAG_variable`) to discover which variables are in scope, their type, and memory locations. With this information, it could then provide those variables to an interested user on the appropriate instruction with the identifiers that were in the original source code.

There are two main ways that a native debugger uses to interrupt a running program. These are loosely defined in terms of "hardware" breakpoints and "software" breakpoints.

Hardware breakpoints are sometimes made available for debugging by the processor, in a way that's built in to a the instruction set of a program. For instance, a chip might have a dedicated register in which a breakpoint address could be stored. When the PC[16] matches a value in a breakpoint register, the CPU transfor control to a registered handler, such as a debugger.

Software breakpoints generally relate to injection of code into an application to cause it to suspend execution at a certain location, passing control to a debugger. As soon as the program is stopped, the original instructions overwritten are restored.

Once execution is past to the debugger, it can utilise the DWARF debugging information to provide a human usable interface to an internal application data, as it relates to the source code identifiers - for instance printing the data inside a structure given a variable name that is in scope.

2) Debugging Java: In Java, the situation is much more simple. The Java Virtual Machine (JVM), operates on `Java Byte Code`, analysing and executing it with a plethora of optimizations. `Java Byte Code` is already tightly coupled with the underlying source code in `Java`[17], and does not require injected specification information – it is already within the Java Byte Code specification. With the ability to intercept changes in control flow and read data values, we can extract the way each class interacts with the environment on a per-method basis. This is done through an API named the Java Platform Debug Architecture, and discussed fully in Section III-C2.

[16]PC stands for Program Counter – it is a register which corresponds to the instruction which is next to execute
[17]This is less the case for languages like `Scala` and `Closure` which also compile to `Java Byte Code`

E. Software Testing

Software testing is both highly important [15], and very labour-intensive and expensive – in 1976 it was evaluated to account for approximately 50% of the cost of the development cost of a software system [2], and there is evidence that the complexity of software systems has increased since then, causing further necessity for software testing practices [16].

The testing process can be classified to several different areas. These classifications were introduced for the Waterfall methodology (Figure 5) [17] of software development, but are applicable to more recent development trends today [18].

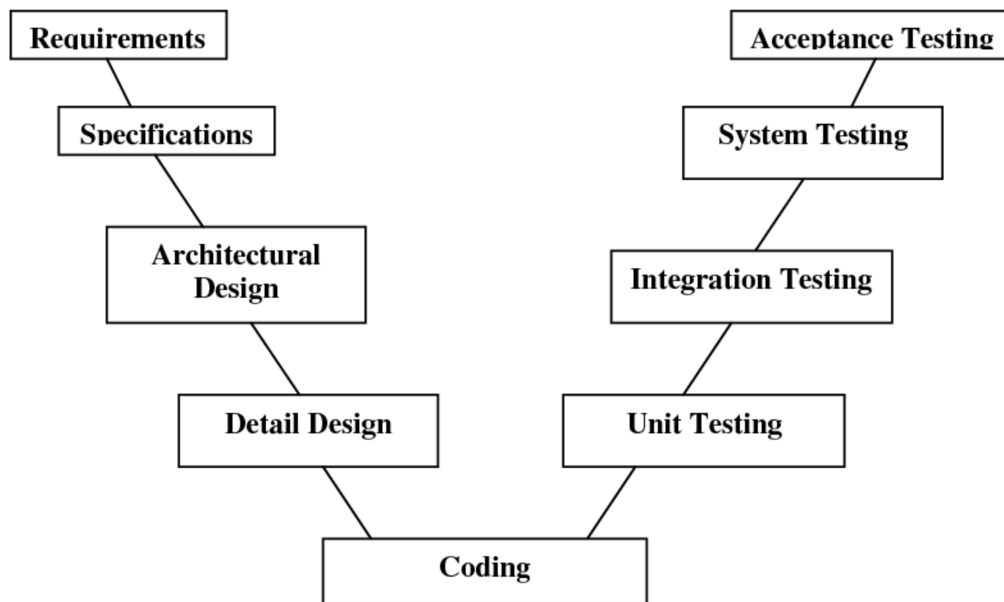

Fig. 5. The V Model as proposed by William E Perry

This classification of tests (right hand branch of Figure 5) is a hierarchical structure which executes from the bottom up – where the higher levels assume successful and satisfactory completion of the lower level tests. The purpose of this is to increase the speed of discovering bugs – the *lower* levels of tests exercise and test less code per test, whilst the higher levels bring together more components and test more higher level integration logic and behaviour. As such, tests on the lower level limit the location of defects to a smaller area of code, and in practice allow for bugs to be diagnosed and found faster.

In practice, unit and integration tests are generally performed in a programmer's IDE (Integrated Development Environment), and system tests are done in a simulated environment [19].

1. Unit testing is the lowest testing level. It is often even performed before implementation code has been written – a practice called TDD (Test Driven Development), and popularised by the *Extreme Programming* software development practice [20]. Unit testing seeks to discover if the implementations satisfies some functional specification. As this functional specification does not

often exist[18], unit testing is generally reduced to looking at the effects of running a particular implementation on some data, as specified by a programmer. In an object oriented program, these effects are limited to the interactions with object dependencies and the return values. By using mocks (Section II-I), these dependencies can be removed, so that each test is highly *focused*.
2. In Integration Testing a developer starts to combine the different components of the code to form working subsystems. These subsystems are then interacted with to verify correctness of behaviour.
3. System Testing (aka Acceptance Testing) is done to prove that the systems implementation meets a particular set of system requirements. These tests form the entire system into its production configuration, and perform tests in line with what occurs in a real life scenarios.

In Figure 6, the test driver is applied to a class with a tree of dependencies. As each of these dependencies is instantiated, this test has both side effects through interactions with the File System and Database module, and is slow – it incorporates a computationally intensive operation.

This could be considered an *Integration Test* as it brings together a submodule through instantiating multiple components, and so tests for the overall behaviour of that submodule.

Through the use of *Mocking* (Section II-I), these dependencies are reduced such that the test driver is now only testing the logic of the *Class Under Test*. This can be considered a *Unit Test*. This process can be applied to each of the modules, mocking out their dependencies, such that when a problem now occurs, its diagnosis can now be focused to that of the particular class under test, rather than with the inclusion of all that class's dependencies.

It is important to note that between System Testing and Unit testing lies a wide spectrum of these integration tests – the number of possible combinations of subsystems is combinatorially large.

This can be visualised with Figure 7 – each of the mocks could be replaced with an instantiated copy in turn, of which some, or all of that instantiated copies are mocked.

We consider generating this series of tests in our section on Test Trees (Section VI-A), as this spectrum is prohibitively expensive to be fully built manually by developers.

F. Automatic Test Generation

As Software Testing is very expensive, there has been a number of efforts to automate this process. If it could be even partially automated, the cost of developing software would be significantly reduced. However there is a fundamental problem with automating the software testing process. This is known as *The Oracle Problem* [22].

An oracle is any (human or mechanical) agent which decides whether a program behaves correctly in a given test, and accordingly produces a verdict of "pass" or "fail". The problem is that there is not enough information in a program alone to state whether it is working correctly.

There are several approaches to producing an automated oracle. We consider three of them:

1. A programmer can encode the *specification* for an applications or components behaviour as constraints. These allow for a category of tools which check that these constraints are not violated [21]. A violation is an "incorrect" behaviour, whereas satisfiability is "correct" behaviour.

[18] the exception to this would be in the use of annotation languages like EscJava [21] to provide formalised specification of the behaviour of a class or method.

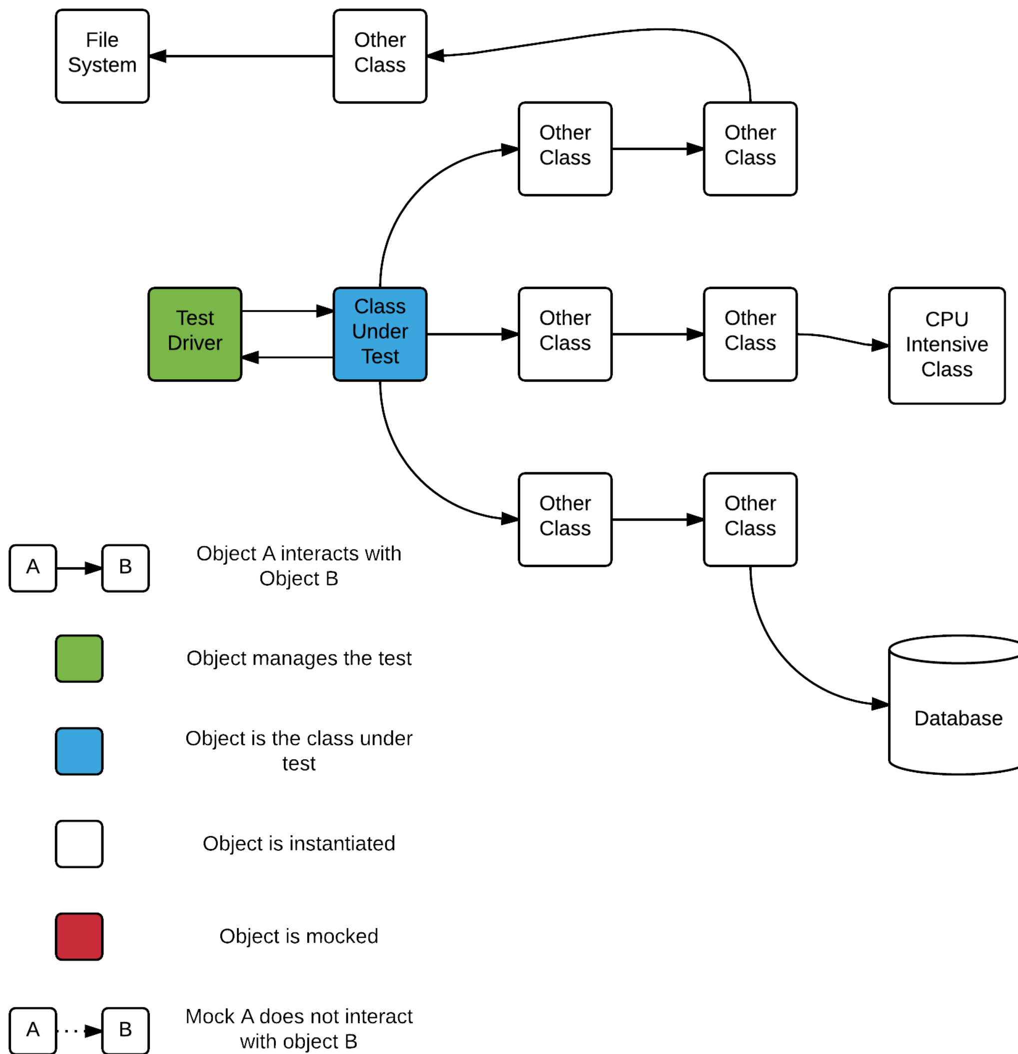

Fig. 6. A visualisation of a hypothetical test with instantiated dependencies

2. Oracles can be built which check for terminating conditions of a program given symbolic input. For instance, null pointer dereferences, API usage violations, deadlocks [23] [24]. A terminating state not a particular location is "failing" behaviour, otherwise "passing" behaviour.
3. Oracles can be built around the concept which states one particular *version* of an application is "correct", and as such any regression in this behaviour is "incorrect".

For our purposes, we aim to generate a *regression* suite of tests, and so follow this third variety of oracle. We generate tests from a *working* version of a program, and as such **all of our tests should pass on this version of the program**. We then run the changed version on these tests, which allow

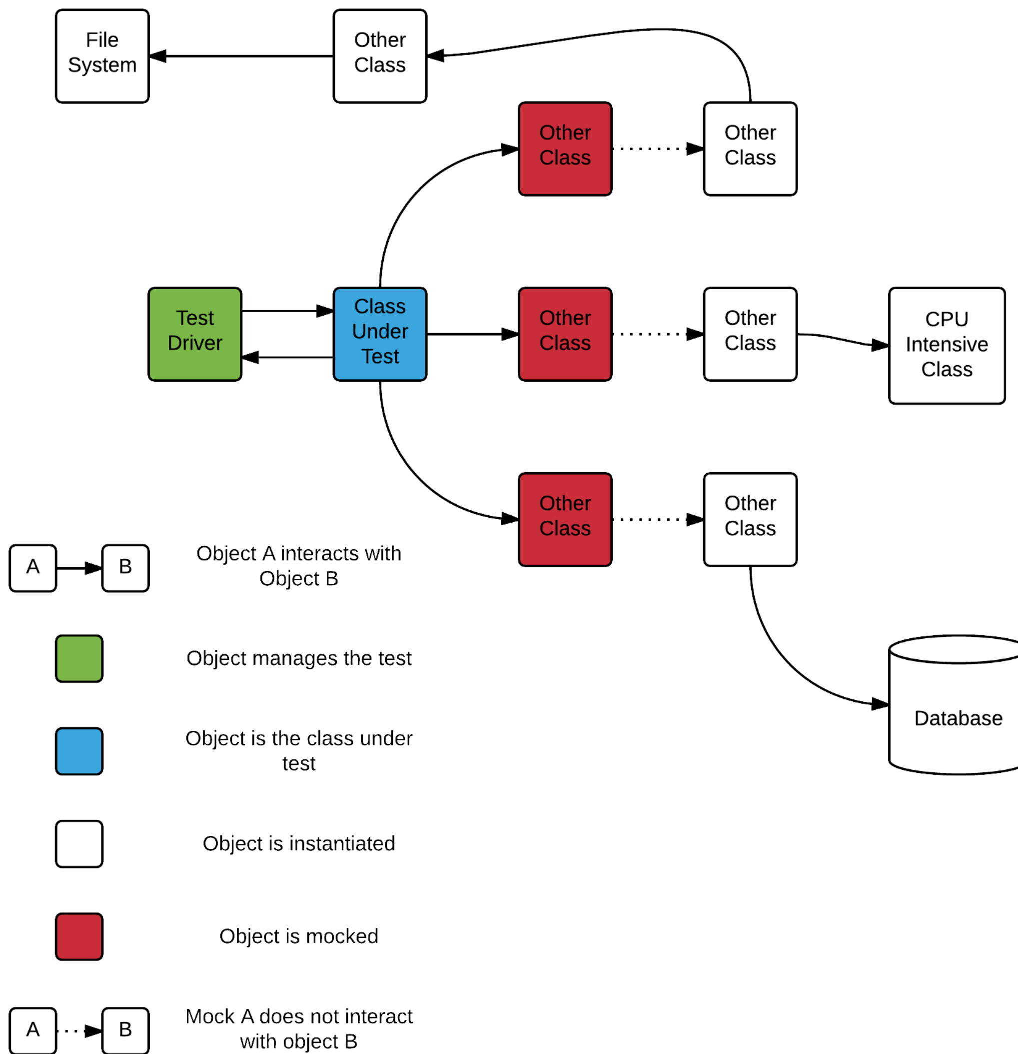

Fig. 7. A visualisation of a test with mocked dependencies

us to discover regressions. False negatives are theoretically removed through the use of *Test Trees* (Section VI-A), however we have no evaluation of this concept, only subjective reasoning.

G. Deterministic Execution

One of the core qualities we aim to build into our tests is that of *determinism*. A *deterministic* test or execution is one that *always* has the same result, for a given version of the application. More formally, two entities are considered to be deterministic if, when they start from the same initial state and apply

the same sequence of operations, they then reach the same final state. This should hold true for any execution of the same method with the same input data, no matter the machine that it is run on.

In practice, this is not always the case. There are several sources of non-determinism in an application. Barring the more unusual ones such as alpha particle strike [25], there are two main categories of non-determinism in application code:

1. System or environmental Interaction
   - System calls that return host-specific information
   - gettimeofday(), gethostname(), ...
   - Random number generators
   - Environmental (third-party) interaction
   - Interaction with human through graphical interface
   - Interaction with shared memory, I/O, etc.
2. Scheduling/Control Flow
   - Multithreading
   - Asynchronous Events
   - Interrupts
   - Exceptions
   - Signals

In this project, we ignore the scheduling/control flow causes of non-determinism, because these – in practice – do not occur in small units of code, only in the interactions of more complex, multi-threaded constructions. Since our objective is to generate unit test code, this approach has not caused us any issues during the project. As such, furthermore, when we refer to non-determinism, we refer to that caused by system or environment interaction.

When writing tests, non-determinism causes big barriers to successful automatic *verification* and *testing* of code. The two below code fragments illustrates the issue with such a test.

```
1 class Dice {
2     Random random = new Random();
3     public int rollDice() {
4         return random.rand(0,7);
5     }
6 }
```

Listing 4. Rolling dice

The method in Listing 4 is intended to return an integer between `1` and `6`, as determined by the random number generated `Random`. A developer might have the presumption that the `random.rand(int a, int b)` method returns a number between `a` and `b` exclusively, i.e. within the set `{a+1, a+2, ... b-1}`. A human might write a test for this method that looks like Listing 5, ensuring that result of the method call is *reasonable*, to what he perceives as expected output.

```
1 public int testRoll() {
2     Dice dice = new Dice();
3     int roll = dice.roll();
4     assert(roll <= 6 && roll >= 1);
5 }
```

Listing 5. Deterministic Test Case

Now consider that the `random.rand(1,6)` does not produce a number within the bounds of the arguments, *exclusively*, but rather *inclusively* – i.e. it will produce a number in the set `{0,1,2,3,4,5,6,7}` instead of a number in the set `{1,2,3,4,5,6}`. Now this test has a non-deterministic result, failing `25%` of the time, and passing `75%` of the time.

Whilst this example is a trivial one, in more realistic scenarios non-determinism can be much more subtle. Fortunately, there are strategies for mitigation of both of these issues.

H. Eliminating non-determinism in system calls

In Listing 4, the cause of non-determinism is the use of a *Random Number Generator*. Random Number Generators function through collecting sources of entropy: most commonly delegating to system calls whose results are non-deterministic, and have a large possible set of values. For instance, a common *seeding*[19] strategy is to make a system call to access a hardware random number generator[20].

Thus the way to cause determinism in the dice roll test is to ensure the *seed* is an explicitly provided value in the test.

There are several ways of generically doing this, regardless of what the system call is. Perhaps the most generic way we found of doing this was an attempt to make a recording of an entire VM execution, and then allow a user to restart the VM at any point during the execution – this idea was commercialised by VMWare [26], though they've since abandoned the product offering.

A more actively developed and maintained way of doing this is Mozilla's `rr` project [27]. Mozilla attempts to do this on *any* program execution on a Linux operating system through *recording the system calls and responses made during a non-deterministic execution*, and then allowing a developer to replay and debug a deterministic execution of the original program, through running the program in a sandbox and *intercepting system calls, replacing their responses with the original responses* during the *recorded* execution.

This project has been successful – allowing developers to reproduce hard to find bugs in complex software. It does have a fundamental limitation however – in that with this type of non-determinism the sandbox that re-runs the original execution must emulate a single-core machine with a deterministic scheduler. Without this limitation, all the sources of non-determinism in from scheduling/control flow return.

The approach we have taking towards ensuring non-determinism in our tests is provided through our limitation in our technique to the Java programming language. This is not a fundamental limitation – we could use a technique similar to that of `rr`, and intercept and replay system calls in any application code. However, through this limitation, we can simplify such our implementation through using *Mock Objects* (Section II-I).

I. Mocking

Extreme Programming Examined introduced [20] the concept of mock objects to both ensure non-determinism of system calls, and simplify tests with dependencies on non-trivial objects. The concept

[19]Seeding is the name associated with initialising a pseudo random number generator with random values, from which sequences of pseudo random numbers can be generated

[20]A hardware random generator functions through collecting input sources from the environment that are so chaotic so as to be considered random – such as thermal noise, or quantum sources

is to replace the code under test's (hereafter referenced as the *domain code*) dependencies with dummy implementations that both emulate the real functionality and allow the enforcing of assertions about the behaviour of the domain code.

These mock objects are passed to the target domain code which they are testing from outside, and when the domain code is invoked, it is intended to interact with the mock objects in the *same way* as with the instantiated objects.

Recalling our dice example in Listing 4, one might imagine one way to remove the non-determinism would be to replace the `random.rand(0,7)` return value with an explicitly provided one.

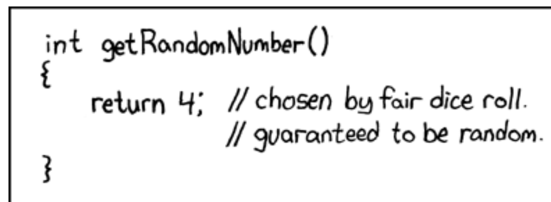

Fig. 8. XKCD 221: Random Number Generator

```
1 class Dice {
2     Random random = new Random();
3     public int rollDice() {
4         return random.rand(0,7);
5     }
6 }
```

Listing 6. Non-Deterministic Dice Roll

```
1 class FakeRandom extends Random {
2     @Override
3     public int rand(int a, int b) {
4         return 4;
5     }
6 }
7
8 public int testRoll() {
9     Dice dice = new Dice();
10    Random mockRandom = new FakeRandom();
11    dice.random = mockRandom;
12    roll = dice.rollDice();
13    assert(roll <= 6 && roll >= 1);
14 }
```

Listing 7. Deterministic Dice Roll

In Listing 7, the random implementation used by the `Dice` class is replaced with a `Mock` object. In most languages, there is a way to create these objects dynamically, without the generation of source code for the mock. In building them this way (Listing 8), the simplicity of test classes is reduced.

```
1 public int testRoll() {
2     Dice dice = new Dice();
3     Random mockRandom = mock(Random.class);
4     when(mockRandom.rand(0, 7)).thenReturn(4);
5     dice.random = mockRandom;
6     roll = dice.rollDice();
```

```
7     assert(roll <= 6 && roll >= 1);
8 }
```

Listing 8. Deterministic Dice Roll with Mock Objects

Underlying this scene, the mock object has created what is called a *Dynamic Proxy Object*[21] . This object is configured with a set of interactions and responses, and returns these configured responses when interacted with in the specified way. One can imagine that the `mock`, `when`, and `thenReturn` calls have created an object like Listing 9, at runtime.

```
1     class MockRandom extends Random {
2         int DEFAULT_INT_RESPONSE = 0;
3 
4         @Override
5         public int rand(int a, int b) {
6             if (a == 0 && b == 7) {
7                 return 4;
8             } else {
9                 return DEFAULT_INT_RESPONSE;
10            }
11        }
12    }
```

Listing 9. Internals of a Mock Object

Furthermore, these objects record their interactions, and allow for verification that such an interaction has occurred. This lets us write a test that ensure a run of the test code has the same *interactions* as what we expect. For instance, in Listing 10 shows very similar code to what we would generate through our tool – the addition of line 8 being the verification step.

```
1     public int testRoll() {
2         Dice dice = new Dice();
3         Random mockRandom = mock(Random.class);
4         when(mockRandom.rand(0, 7)).thenReturn(4);
5         dice.random = mockRandom;
6         roll = dice.rollDice();
7         assert(roll == 4);
8         verify(mockRandom, times(1)).rand(0, 7);
9     }
```

Listing 10. Mock object with method call verification

This change would then allow us to ensure that the `random.rand(0,7)` call was actually made once. The utility of this additional verification might be to ensure that a future developer not make a breaking change (regression) without a failing test informing him that something is not quite right – for example, Listing 11. If the test above was ran on the below code, the failing test would inform him that the `Random` field is expected to be interacted with via `Random.rand(0,7)` at least once:

```
1     class Dice {
2         Random random = new Random();
3         public int rollDice() {
```

[21]A Dynamic proxy object is an object whose type and values are dynamically realised and constructed at runtime to be an overridden extension or implementation of an existing interface. It has all the functionality of the proxied object, but allows for each method and field to be overridden programmatically and JIT'd in to the application. In Mocking, each method is overridden with a default implementation which - depending on the method's return type, returns either a valid but arbitrary value, or `null` for object references, until configured otherwise

```
4            return 4; // chosen by fair dice roll.
5                      // guaranteed to be random
6        }
7    }
```

Listing 11. Bugged implementation of ‘rollDice‘.

J. Related Work

Because of the potential utility of automated test generation, there have been many attempts to automate it in the past. Many of these have been successful commercial products. We enumerate some of the most common works here, as well as some similar work in the past which uses the same techniques that we do for collecting traces but for a different purpose.

1) AgitarOne: AgitarOne is a JUnit test generator. It claims to allow for the generation of unit-level tests for Java code with no user input. They also use the same *oracle* that we do – a developer just states a single version of his code is the “working” version and the tool generates test cases around this.

In practice it appears to use try to generate a series of random test inputs for each class’s methods using static analysis. It then filters these down to those that *pass* on the version provided.

The difference in techniques is that our test data is collected from actual runtime usage of the application, and so we judge it more likely to correspond to those that a developer would write.

They charge prices “starting at” $10 per class.

2) CodePro AnalytiX: CodePro is a system supported by Google, which provides both Static Analysis and JUnit Test generation. Its price is unspecified, but it is targeted at enterprise customers.

It appears to provide a Wizard interface for the generation of unit tests. It allows a user to select types and methods, and then attempts to use static analysis to generate test data.

It is more limited than our tool. It does not support generating tests for private elements, fields, initialisers, inner types, super types or nested types.

Our tool has the necessary logic to generate tests for each of these cases.

It does however use control flow analysis to try to generate high coverage unit tests. Our technique differs in that we will generate them if we `record` an execution which uses them. We do not think either approach is necessarily better than the other, but we expect we avoid a significant amount of complexity in our implementation.

3) Randoop (aka Palus): Randoop uses Random test generation. This means it randomly, but with a clever heuristic, generates sequences of methods and constructor invocations for the classes under tests, and then uses these sequences to create tests. These tests are then executed, and the results of this execution is used to create assertions that capture the behaviour of that program.

Randoop is free.

It is hard to compare the two approaches. Our tool looks to find breaking changes that occur during production usage. Randoop looks to find bugs, following branches that might never actually get executed in the original program.

4) Daikon: Daikon is a tool to discover likely program invariants. It does this by running a program, observing the values that the program computes, and reporting properties which were true across the observed executions.

These properties may be examined by a human or used as input to a tool. It's mechanism to generate traces is similar to what we utilise, but it tries to further augment data by simplifying to the likely invariants of a method call or field.

Whilst this is potentially very useful for tool input, the results do not correspond well with test data generation. In our tool we do not search for likely invariants, but use the traces to guide test data input.

5) Test Factoring: Test Factoring is the closest body of academic work to what we have achieved in this project. In [28], the implementation and evaluation of an Automatic Test Factoring engine for Java is described.

Test Factoring is the creation of fast, focused unit tests from slow system-wide tests; each new unit test exercises a subset of the functionality of the originating test. Through the augmenting of a test suite with factored unit tests, errors could be caught earlier in the test suite run.

It is unfortunate that we discovered this paper after our implementation was complete – it would have significantly increased our progress and we might have gone in a differing direction if we had found it earlier. Despite this we present contributions that are lateral to this paper:

a) The use of debugging APIs to collect trace information: The implementation that [28] uses to collect traces overloads of the `classloader` to rewrite class byte code and inject instrumentation calls. They describe limitations in this approach which can often prevent capture from completing – for instance the use of *native* methods or *reflection* libraries. In addition to this, it means that the technique is limited to running on a particular Virtual Machine version. These limitations do not apply to our technique, because we use a standardized API which removes these complexities.

The core advantage of their technique is that it is faster – the overhead that is caused by such instrumentation is lower than that of using the Debugger APIs.

b) Generated Test Code: Test factoring is aimed at producing a faster run of the integration suite. Its generated test code is machine readable, but not intended for human use. We have put significant engineering effort into ensuring that we generate tests which can be placed directly into a developers code base, are of similar style to human written tests, and use APIs used for manual test production (Mockito/JUnit).

6) Alternative Tracing Techniques: There were two other core techniques that we could have used to collect test data.

In retrospect, Byte Code Interleaving (Paragraph II-J6a) might have been a better choice to achieve higher performance of our application. This is used by [28] and [29], though they did describe significant engineering complexities that must be overcomes for the use of *native* methods, *reflection* libraries and *static* objects and methods. We avoid these complexities by using Debugging APIs.

a) Byte Code Interleaving: This utilises a similar technique to a native debugger, rewriting the byte code to include additional instructions that allow a process to collect traces. This was utilised in the paper [29].

The reason why we do not choose to utilise this technique is that it would cause our software to then rely upon knowledge of a particular byte code format. This format, whilst generally maintaining compatibility, can break between different JVM versions. Also, this technique is more complex than utilising provided APIs, and is expected to be less performant.

b) Java Agents: By attaching a Java agent to a JVM, control and analysis of internal JVM properties can be achieved in a lower level way than the JPDA. For instance, the classloader can by dynamically intercepted to provide different implementations of classes created via `new`. This is potentially useful should one be looking to detect regression.

III. Implementation

In this chapter I describe the core implementation of the application. This discussion includes explicitly stating and providing insight into why some of the design decisions were made during development, and some possible alternatives that could have been taken instead.

For convenience, to differentiate between discussion of the *application* we have built, and the *program* for which tests are generated, when we use the word *Application* we are referring to the test generation system we have built, and when we use the word *Program* we are referring to the program that the *Application* is exercising and generating tests for.

Firstly, before we delve into the particulars, it is important to highlight the core requirements of our application, as that has driven the design.

A. Requirements

Wide Variety of Input Programs: We believe that to maximise the utility of our tool, we should aim to produce tests for as wide as possible a subset of programs. This requirement puts a number of constraints with regards to the possible approaches we can take.

Performance: To maximise the utility of our tool, we aim to ensure that it works on large programs just as well as small programs. This limits our approaches to those which have linear performance decrease with program complexity.

Reproducibility: For problems to be diagnosed and resolved, the ability to reproduce any faulty behaviour is essential. We endeavour to make full use of the techniques discussed in Section II-G to manage this.

Soundness: Failing tests for applications which are correct waste time, and limit the usability of test generation tools. We aim to ensure that we minimise both false positives and false negatives in generated tests.

Usability: One of the core advantages for research into generation of unit tests is that they fit directly into developers workflow. If the tests generated are unreadable, and far away from what a developer would write normally, then they have less utility. As such, this is something to be avoided. Furthermore, the generated tests should be isolated to the extent where a developer can pick and choose the tests to drop into their development environment. This constrains our application to generate tests with well known test harnesses and frameworks.

B. Overview

The application consists of three core modules. The first is loosely categorised as the `Recorder` module. This module instruments and controls a subprocess *under test*. This subprocess is a Java application that we aim to generate tests for. The `Recorder` module collects traces on class interactions and forms these into a graph.

The second core module is categorised as the `Storage` module. It is a set of interfaces to and from a `Graph Database`. The interface to this graph database collects `events`, and appropriately forms `relationships` between these `events`. The core purpose of this is to appropriately create relationships between `events` that might not necessarily have occurred in a chronological ordering in the `trace`. For instance, a `Method Call` event might be closely connected to the `Constructor` of the original object for which a method call was made upon, despite the method call occurring significantly after the construction of the original object. These relationships are necessary to produce tests – allowing for the `Scope` of an object to be closed efficiently regardless of the time period between interactions to a single object.

The third core module is categorised as the `Generator` module. This module has the purpose of producing unit tests from the information stored in the `Storage` module. It contains program generation logic which utilises the `Storage` module's `Graph`, as well as static analysis of the bytecode and `Reflection`.

A typical run of the application will take the form of two Java applications being run. The first will run the `Recorder` module, and will be passed the command line arguments to run the program as well as several options to manipulate the way it generates test. For instance, the command below will run the `Recorder` module to generate tests from the `test.test1.Main` entrypoint. It will provide this entrypoint the command line argument array of `String[]{"10"}`. It would output these traces to the storage module configured at `/tmp/db`. The detailed usage of the CLI application used to record traces is described in Section III-C5.

```
1 java -cp recorder.jar -db /tmp/db -target test.test1.Main -args 10
```

The second application to be run is the test generation module. This below command would simply produce as many test cases as possible, and output them to the command line. It would do so from the storage module configured at `/tmp/db`. The detailed usage of the CLI application used to generate tests is described in Section III-F8.

```
1 java -cp generator.jar -db /tmp/db -dump
```

In the following sections, we describe the three stages of test generation (Figure 9) in greater detail.

C. Recorder Module :: Trace Collection

The `recorder` module instruments and controls a Java executable subprocess, collecting instrumentation.

In an overarching sense, this module has the responsibility of working with JPDA APIs to extract out the necessary information to generate tests from. This information consists of a sequence of *events*. These *events* are described in Section III-C2. A key analogy for the events that this module collects and processes are that they are *equivalent* to the events that a Java Debugger collects and processes. A Java Debugger will place *Breakpoints* on particular lines when a human indicates internal information on the

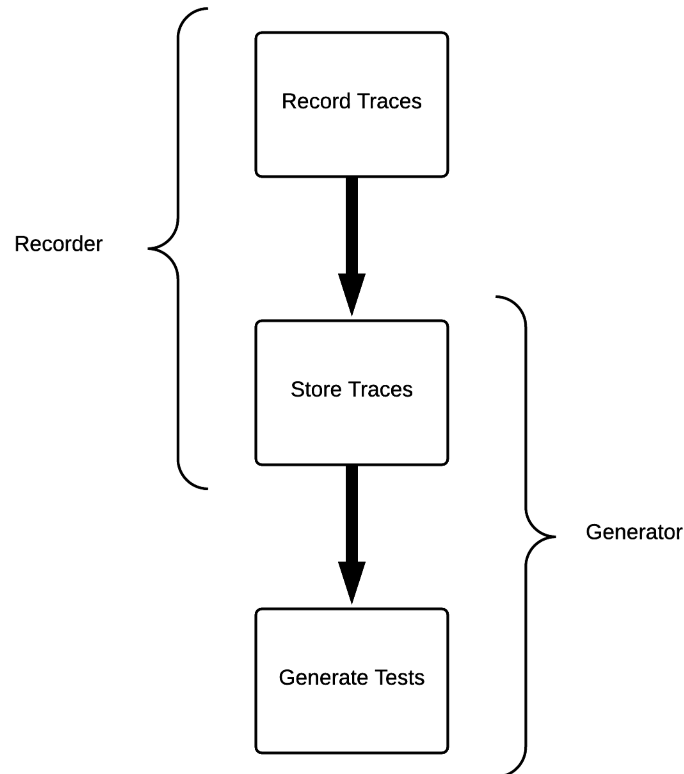

Fig. 9. An overview of the three stages of test generation

state of an application is interesting. What our application does is place and delete these breakpoints in such a way to *sample* the changes in state of an application before and after method calls of certain classes, as indicated by command line arguments.

1) Architecture: The `recorder` module is architected as a pipeline (Figure 10). Events of form defined in Section III-C2 pass through the pipeline, getting *filtered* out when they have no meaning, and *augmented* with additional information (such as concrete values for arguments) until they reach the end of the pipeline, which stores them into the graph database.

These events are partially ordered. What this means is that the sequence of events that pass through the pipeline for *some* events is tied to the causal ordering of these events in the JVM. For instance, if a method is entered into, then throws an exception, which gets caught at a particular location, the sequence of events would respect that ordering. However, *field access* and *field change* events do not fully respect this ordering – they are only ordered so as to be between associated *Method Entry / Method Exit* events. For instance, if a method is entered which changes field A followed by field B, then it may occur that we record field A is changed followed by field B.

The reason for this ordering/partial ordering mix is to maximise the performance of the application.

When we query for the **value** of a variable or field, we *must* have the associated thread in the JVM **halted** at the precise location where that variable enters or leaves scope. This allows us to look up the variable's value at the point where it changed. Likewise, when a method call is made, we need to keep track of the order in which it occurs such that we can reconstruct what happened at a later stage when producing our *mock* objects. However, this halting of the virtual machine causes **significant slowdowns**[22] on the tracing of the application, such that we try to only do it when absolutely necessary.

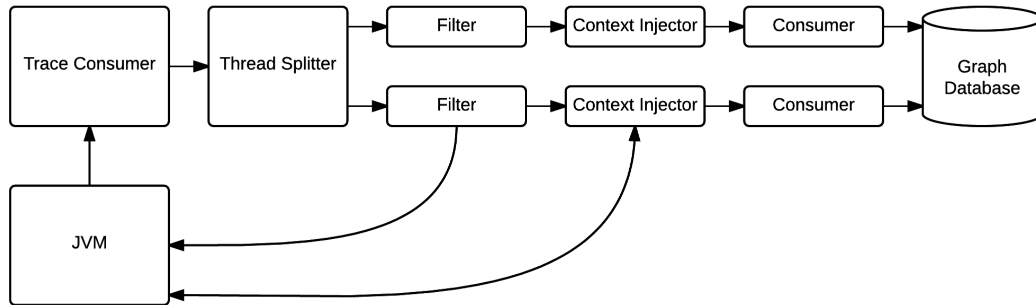

Fig. 10. The recorder module's architecture

2) Java Platform Debugger Architecture (JPDA): The Java Platform Debugger Architecture (JPDA) is a JVM-provided set of interfaces that allows for programs to listen to, and control an executing Java Process. This is the interface which we use to perform our instrumentation[23]. This interface is implemented by the `Java HotSpot VM`, which is the Virtual Machine we use in our Evaluation – though is a standard across JVMs[24]. These allow a connecting process to send requests to the VM, such as a request for a notification on *Method Entry* or *Method Exit* inside the Java Process as in Listing 12.

```
1      private void sendEventRequests(VirtualMachine vm) {
2          EventRequestManager mgr = vm.eventRequestManager();
3          MethodEntryRequest menr = mgr.createMethodEntryRequest();
4          menr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);
5          menr.enable();
6          MethodExitRequest mexr = mgr.createMethodExitRequest();
7          mexr.setSuspendPolicy(EventRequest.SUSPEND_NONE);
8          mexr.enable();
9      }
```

Listing 12. The mechanism we use to request events

This is generally the mechanism via which Java Debuggers provide an interface to running code for a human operator. Through an event loop (Listing 13) an attaching process can be provided events in real time as they happen on instrumented code. The JVM connected to also dynamically slows down

[22]The slowdown for halting the VM can be seen by contrasting Figure 30 and Figure 22. It is of the order of a 30x slowdown, which when compounded by the 10x slowdown of an empty tracer reaches ~300x. This limits the applicability of our technique to running tracing processes on an external services to avoid significantly slowing down a developers machine for large periods of time.

[23]Alternative approaches discussed in Section II-J6

[24]This *implies* that our technique should work on Android with little modification, as we are using JVM API calls rather than bytecode rewriting (Section II-J6). However, this is an untested claim.

should these events not be consumed fast enough. This architecture results in a significantly decreased complexity in the implementation of a debugger for Java, as opposed to the equivalent in a native language (Section II-D1).

```
public void run() {
    EventQueue queue = vm.eventQueue();
    while (connected) {
        EventSet eventSet = queue.remove();
        EventIterator it = eventSet.eventIterator();
        while (it.hasNext()) {
            handleEvent(it.nextEvent());
        }
        eventSet.resume();
    }

}
```

Listing 13. An event consumer for JPDA provided events

The event requests can be assigned with *filters*, and *suspend policies*. A *Filter* allows for us to only halt on a certain condition being met. The core filters we use are:

1. Thread Filters – this instructs the breakpoint to only be reported if the thread is the explicitly provided one. This allows us to handle multi threaded code without significant performance losses.
2. Class Filters – these allow the breakpoint to only occur on a location within the provided class. This class can be given in glob form, i.e. com.test.* to indicate all (recursively defined) subclasses within the com.test package.
3. Count Filter – these allow the breakpoint to only be reported after it has been hit

Suspend policies allow us to ensure that the JVM performs one of 3 halting actions.

1. The thread which hit the breakpoint is suspended immediately, until the event reported is handled: `EventRequest.SUSPEND_EVENT_THREAD`.
2. The entire JVM is halted until the event reported is handled: `EventRequest.SUSPEND_ALL`.
3. The VM reports the event but does not halt: `EventRequest.SUSPEND_NONE`.

As any suspension causes a drastic slowdown in the VM's operation, we try to avoid it as much as possible. However, to collect data from the running JVM, the thread it reaches must be suspended – otherwise data values may change and become out of scope before we can ascertain data values. For instance, when we enter a method that we are interested in generating tests for, we need to collect its arguments so we can either pass them in during our generated test (should they be primitive), or reconstruct them as mock objects (should they be an object reference) as necessary.

There are 11 events that we get from the JVM explicitly as part of our tracing activities, and we augment these events or create new ones as necessary. The below events are the ones provided by the event loop in Listing 13, and are augmented as necessary in our `recorder` module.

1. **Method Entry Event**. This event has no arguments, so the JVM must be halted and stack frame inspected to collect the **Arguments**.
2. **Method Exits**, with associated **Return Values**.
3. **Field Accesses**, with the associated **Field Identifier**. The **Field Values** are extracted via stack frame inspection after JVM halt.

4. **Field Changes**, with the associated **Field Identifier**. The **New Field Values** are extracted upon the **Method Exit**, as tests generated only need know about Post-Method-Call values, to perform the associated `Assert`.
5. **Exception** events occur when an exception occurs, *not* when it is caught. As such we halt the JVM on an exception, perform one **STEP** or single instruction execution of the JVM, then try to identify the **Catch Location** from identifying the number of dropped *stack frames*.
6. **Class Preparations**, with the associated **Class Identifier**.

On top of these 6 Object interaction specific events, there are 5 more global events that we use to track an Applications lifecycle.

1. **Virtual Machine Start**, which indiciates the beginning of the JVM's lifecycle.
2. **Virtual Machine Death**, which indiciates the end of the JVM's lifecycle.
3. **Virtual Machine Disconnect**, which indiciates an abnormality in our connection to the Virtual Machine.
4. **Thread Start**, which indicates the start of a new threads lifecycle.
5. **Thread Death**, which indicates the end of a threads lifecycle.

3) Lifecycle: The architecture in Section III-C1 Figure 10 represents the sequence of functional components that events pass through. The lifecycle of the application is:

1. The subprocess VM is instantiated with the provided command line arguments. A series of Event Requests for the initialising events[25] are sent to the VM. The VM is then launched, starting the underlying application.
2. `VMStartEvent` and `ThreadStartEvents` propagate through the pipeline, initialising the tracing machinery.
3. When a `ClassPrepareEvent` occurs for a class which we desire to *trace* – as indicated by command line arguments – appropriate `MethodEntry` breakpoints are set for each method.
4. When the class is actually instantiated[26], we set the `FieldAccess` and `FieldChange` watchpoints to collect information on each field change. We also `trace` the constructor – as this allows us to generate tests for a method call whilst instantiating using the *same* constructor and values as the traced object was initialised with in the trace.
5. When a method is entered which we are interested in, we start tracing for each method call out which that method calls. This allows us to work out how to mock each of the objects that these method calls are made on with their appropriate return value. However we do not[27] trace the calls made out[28], merely logging their arguments and return values in order.
6. Breakpoints created in (5) are removed upon a `MethodExit` event. We log the *return value* of the method, and the *final values of each field* that has been changed. This allows us to produce appropriate assertions for our tests.

[25] These events are the global events described in Section III-C2, and the `ClassPrepareEvent`, which occurs on the first instantiation of a class with `new`

[26] `ClassPrepareEvents` are global, but when the `new` call actually occurs the VM sends a `MethodEntry` event with `<init>` in its name, indicating the constructor has been called.

[27] Except for when the *indirection constant* > 1, for the Test Trees concept

[28] But for when we are also *interested* in the objects called into

7. After the subprocess under trace finishes, we get a `VMDeathEvent` after which we exit the application.

The logic illustrated above allows us to filter for events that we note as *interesting* through the command line arguments. These *events* pass through the `Context Injector` submodule. This submodule halts the VM, and queries it for the actual data values of each argument and return values. If these values are primitive, then we record in the storage module the `actual values`. If they are object references, we store a `UID` with type information of this underlying object. This `UID` is unique to a particular instantiation of the object. By indexing this `UID` in our `storage` module, we minimise the information on the execution of an object to a map of all interactions with that object. if we are also *interested* in objects of this type, then we collect a map of all interactions the said object makes with other objects during its own method calls.[29]

The end result of this is a series of events which enter into our graph database, referred to as our `storage` module as it is programmatically separated by an interface for event input and querying.

4) Following Control Flow: When implementing a tracer, one particular complexity you must manage is following execution paths through exceptional conditions.

One might imagine, as we did initially, that a simple tracer could be formed through a single Stack object – upon *Method Entry*, a `push` to this stack is made, and upon *Method Exit*, a `pop` to this stack is made. If you were to try to form a Graph of methods which call each other, one would collect a map of interactions by, on *Method Entry*, create a new node in our graph, `peek` the top of the stack to get the caller, join our new node to the callers node, and then `push` our new node to the top of the stack. Likewise, on method exit, just `pop` from this stack. For instance, Listing 14.

```
1  class EventToGraph {
2      Graph<Node> graph = new Graph<>();
3      Stack<Node> methodStack = new Stack<>();
4      @Override
5      public void methodEntryEvent(event) {
6          Node callee = new Node(event);
7          Node caller = methodStack.peek();
8          graph.addRelationship(caller, CALLS, callee);
9          methodStack.push(callee);
10     }
11
12     @Override
13     public void methodExitEvent(event) {
14         methodStack.pop();
15     }
16 }
```

Listing 14. A Naive implementation of a tracer

Unfortunately, this does not work. It relies on the assumption that every method that is entered also exits at some point. With a language that provides exceptions, this is **not true**. The *MethodExit* event provided by the JPDA does not trigger on a method exiting through an exception.

[29]If a `-n` argument of `>1` is provided, then we selectively trace further out into methods that we judge `un-interesting` (i.e. command line arguments for interesting do not include these objeect types) – this allows us to replace the mock objects an additional level of indirection out with an actual instantiation of this object with particular *state*

What this means is that instead, one needs to get the location where an exception is caught, and `pop()` until that point. This yields yet more complexities throughout an application which provides `filters` – in which not every event that occurs should necessarily get persisted to the graph. With our application, filters are necessary, as otherwise the size of the trace produced becomes unmanageable[30]. As such, additional logic is necessary to manage this.

Furthermore, any such logic is inherently unstable – if a single event is missed or throws an exception due to our implementation not accounting for a particular edge case, then the graph gets `corrupted`. In practice, what this means is that the `Stack<Node>` might miss a node, and from then onwards, each method call is connected to an invalid one.

For this reason, our implementation ended up with a large number of branches so that we could handle every edge case we ran into – there was simply no option to iterate on a partially working solution until one reaches a point where it works completely. It either all works or all doesn't work (or we give up at a point in an application's lifecycle when the graph becomes corrupt). When using the JPDA's partially undocumented APIs, for which the best documentation was *Eclipse* or *IntelliJ* source code, this led to significant code complexity.

For instance, sometimes when trying to read a suspended JVMs memory to map object references to our own representation of the object graphs, the JVM would throw an Exception[31]. In these instances, we have to `taint` a number of methods and objects around the failed object, as these are now impossible to generate tests for. Moreover, we have to send events throughout our pipeline such that they can cleanup any expectation of an event that is now no longer going to occur.

5) Usage: The Recorder module is invoked as a Java executable which can be run with the following command line arguments:

```
1  $ java -cp $JAVA_HOME/lib/tools.jar:recorder.jar recorder.Main
2   -args VAL                              : Arguments to use for launched program
3   -c VAL                                 : Class Trace Filter
4   -v                                     : Verbose output mode
5   -filter [DEBUG | TRACE_ALL |           : FilterMode
6   TRACE_TEST1 | TRACE_ONLY_NON_STD |
7   TRACE_LEVEL1_WITHOUT_CLASSLOADER |
8   TRACE_LEVEL1_NON_STD]
9   -input [LAUNCHER | REMOTE]             : Select how the program is connected to
10  -n N                                   : Levels of indirection away to record
11                                           for from the filtered classes
12  -db DATABASE_LOCATION                  : The location of the database to output to
13  -output [STDOUT | NEO4J]               : Output Mode
14  -target VAL                            : Main method to run
15  -threshold N                           : Threshold after which to stop
16                                           sampling a method
```

There is only one required argument, `-target` to provide an entry point to the program – i.e. the class for which whose `static void main(String[])` method will be ran.

The `-args` argument allows for the arguments to this main method to be provided; e.g. `-args "- 1 2 3"`

[30]in some of our early experiments, the trace reached 150Gb, and took 3/4 days to produce.

[31]JVM Exceptions simply take the form of `JVM Internal Exception: Error Code <Number>` – we ran into `Error Code 22` quite a few times, with no available documentation found online on why or for mitigation strategies

By default, the `recorder` module performs a `dry run` on the given method entry. This means that it sends *sampled* events from the target program to standard out. To change this behaviour, the arguments `-output NEO4J` should be provided, along with `-db <some database location>` to select the location of a Neo4j database to be created. This database is then populated with the trace of the application. To generate tests, the `generator` module is ran with this same configured trace directory.

By default, the program tries to sample every method call that is not in the Java standard library[32]. This can be very slow when external libraries are included in the application, and this slow down is mitigated via passing an explicit `class trace filter` argument with `-c`. This filter then only traces methods in and around the explicit package or class provided such that tests are only generated for the given filter. This has a significant speedup in the trace generation process[33]. For example, in the provided example at Section I-A, by avoiding tracing through the actual database connection logic, and instead just tracing the methods around the query logic, we achieved a speedup of approximately 500 [34].

This filtering is augmented with a *sampling threshold*. When set with the `-threshold N` argument, this disables collecting traces for method calls after `N` method calls have passed. What this means is that even for long applications, if each method call itself does not take too long, then the total execution time should *tend* to an *almost native* execution time. The reason why this does not tend to *native* execution times is that the virtual machine still acts under under a *debug* mode, which means that it does not `JIT` compile the Java Byte Code to machine code, but instead acts as an `interpreter`. This means that the `recorder` module is fundamentally limited to running at less than native speeds.

The argument `-n` allows for more complete graphs to be made from filtered graphs. This is referred to as the indirection coefficient. The default value of `1` contains enough information to generate deterministic executions/test cases for each traced method, using *mock* objects for every dependency. However, if we want to use *instantiated objects* for the method under test rather than mock objects, we must instead mock everything a level an additional level of indirection away. This option is used for experimentation with the *Test Trees* concept discussed in Future-Work.

D. Storage Module :: Graph Database

The `storage` module is a set of interfaces for event input and querying of execution graphs. Our current implementation relies on the *Neo4j database* to persistently storage and index this information.

There are several key benefits to using a graph database from this information. The first is that the data model of an executing application naturally forms into a graph, rather than tabular data.

To generate tests, we need to look at the interactions a particular method call makes during one execution. These interactions naturally form into a graph of nodes and relationships – nodes representing the objects which are interacted with, with relationships being the interactions made, and details thereof.

The second key benefit of using the graph database is that it is fast and simple to follow relationships to different data nodes. In a tabular database the equivalent functionality would be provided by the use

[32]The argument `-filter DEBUG` allows one to sample every method call, including the standard library. This is slow – for example, a *Hello World* program consists of over 500,000 events, which takes approximately 30 minutes to collect on modern machine

[33]For a formal evaluation of all speedups, please see Section V for more information

[34]~407.289s in this initial, naive implementation to ~0.832s for our current, partially tracing one.

of indexes. An Index lets one move from one row of data to another quickly, through querying some identifying value in the originating row. However, as our schema requires a large number of related yet separate data values, as well requiring fast lookup given a particular index, the equivalent SQL schema would be highly complex.

The full schema is not formally defined, only having being written in code (Neo4j not requiring any formal schema) – it must encapsulate a large portion of the information encoded when control flow moves and state changes in an application.

The end result is that to generate tests for a particular method can be managed through looking through a series of nodes around the `METHOD` call node, which represent `METHOD INSTANCES` – instances of time in which a thread executed a particular method. These `METHOD INSTANCE` nodes contain relationships to other `METHOD INSTANCE` nodes, representing calls made to and from that `METHOD INSTANCE`. They also have relationships to nodes which contain the state, and changes in state made of data structures in scope at the time of the method call. This process of crawling the graph is conservative, and only reaches at most 2 relationships[35] away from the initial seed node – making the generation of tests from this graph a linear process on the complexity of the method call itself, not requiring analysis of the dependencies of each method call itself[36].

Working with Neo4j graphs has another core benefit – they are human readable and understandable to a degree that a relational format is not. Through looking at these we can understand what went on at runtime, and were able to gradually augment this graph with additional information as needed by our test generator. Neo4j allows for visualisations of the graph through the web browser, allowing for problems in our implementation to be debugged quickly to understand which side of the application is at fault for any issue – the `recorder` or the `generator` modules.

E. Generation Module :: Test Code Generation

The `generator` module has the responsibility of generating the test code.

The input it has available to it are the traces gathered in the `recorder` module, along with the Java bytecode of the program under test. With this bytecode, it can use Java's standard APIs to perform simplistic static analysis, which is augmented by the traces as the test *input data*.

Once this analysis is complete, it uses Java's `JCodeModel` library [37] to programmatically generate test code with the aid of `JUnit` and `Mockito` – two popular testing and mocking libraries for Java.

1) Architecture: The `generator` module, like the `recorder` module, is architected as a pipeline. This internal architecture can be seen in Figure 11.

[35]This is true but for the exceptions of recursion method calls and method calls which cannot be mocked – instead their dependencies are *inlined* into the method call, potentially requiring an arbitrarily large crawl through the graph. However, this crawl does not repeat itself, has no loops, and in practice is relatively fast even for deep recursive calls.

[36]This is true for the *Mock Everything* strategy that we describe here, but not true for the *Test Trees* strategy described in Section VI.

[37]`JCodeModel` is the set of in-built standard APIs to generate Java code. It is provided by package `com.sun.codemodel.JCodeModel` in sun's `tools.jar` – a library located at the root of a JDK. To use it, this `tools.jar` need only to be added to the `classpath`

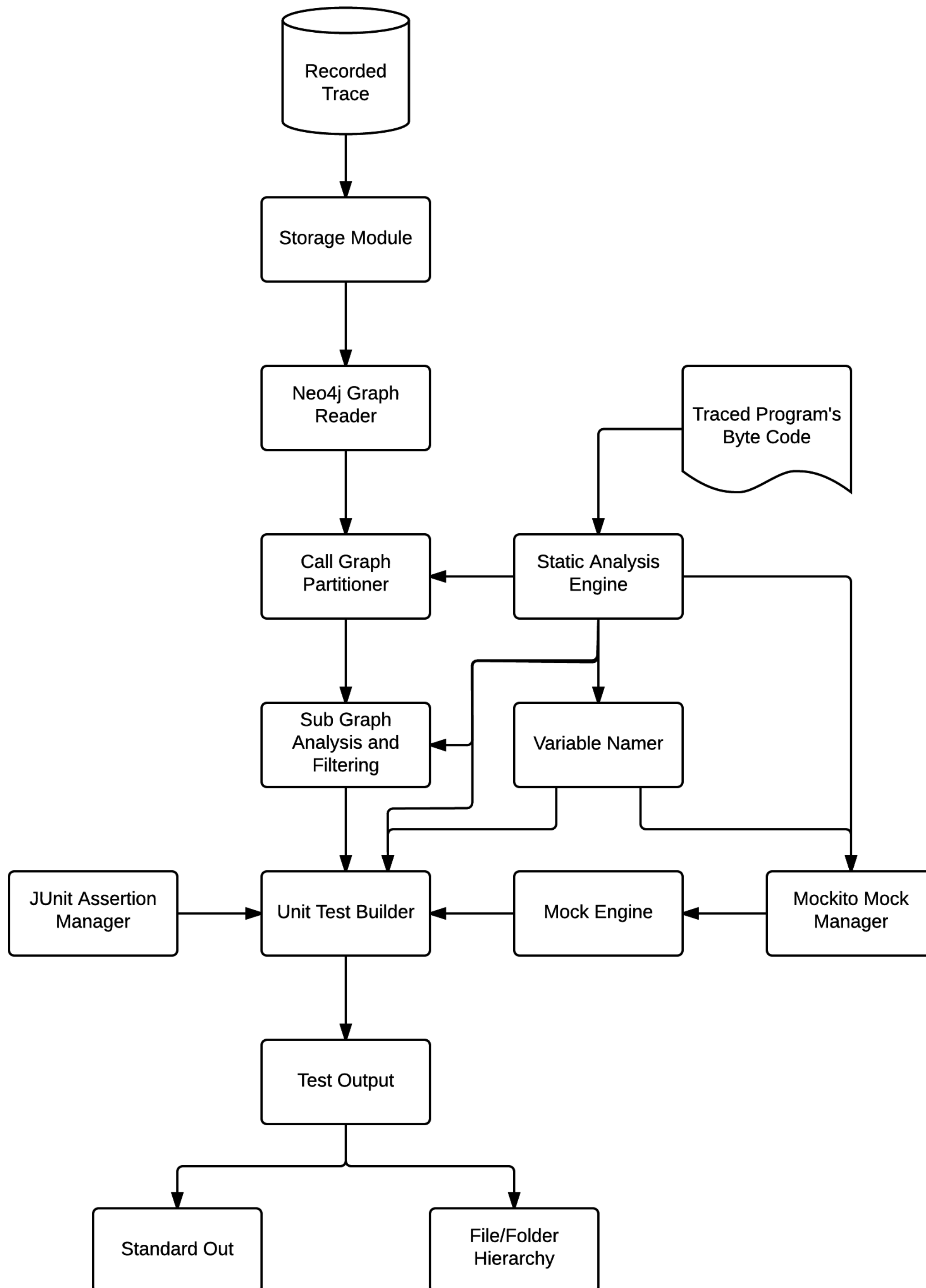

Fig. 11. The internal architecture of the generator module

The process begins by collecting a list of classes for which to generate tests for, from the Neo4j database through the API and Schema provided by the `storage` module. After this, each method and associated *Method Instance*[38] – is filtered by these predicates:

1. The method call and the calls it makes must not be *opaque*. I.e. we must have managed to successfully trace the method call. If a filter was used in the recorder stage for a particular class or set of classes, this will remove all method calls outside that filter.
2. The method must not be a **`<clinit>`** method. `<clinit>` methods are the static initialiser methods – they occur the first time a class is instantiated which has static fields. *static* fields interactions are recorded and generated in the smae way as standard field interactions. This means that we will set them to a mock whenever necessary, otherwise ignore them. In principle, the use of *static* fields means there will be unsoundness if multiple unit tests which utilise that same field on the same class are run in parallel as mocks get overridden. However, as we only generate one test case (though with multiple test-methods) for each class, and mocks are used when there are object references, in practice this will not be a problem.
3. **Optionally** If we have seen a method before, we will not generate another test for it. This means that no matter how many times a method call is traced, there will only be one test for it. The other behaviour is to generate tests for each method call we traced. As the maximum number of times we trace a method is a configurable option, this is the default behaviour.

When we decide to generate tests for a given method instance, we push that method through the pipeline in Figure 11. Each stage is described in the Sections III-E2, III-E3, III-F1, III-F2, III-F7, with intermediary sections providing some of the supporting modules such as Variable Naming Section III-F4.

2) Neo4j Graph Reader: The first step of the test generation process is to cache all associated information from the `storage` module's `Neo4j` graph into memory – into a set of treelike data structures that represent:

1. All information related to the class and method such that *Reflection* utilities in Java can locate the raw class associated with the given method to be tested.
2. The arguments of the method call.
3. The field pre-conditions for each *accessed* field in the method. I.e. what was their value prior to the method been called.
4. The field post-conditions for each *modified* field in the method call. I.e. what was their value at the very end of the method call.
5. The call graph, and associated object interactions. These interactions are indexed on *uniqueID* of the objects.
6. The Constructor that was used to initially instantiate the object, what was its arguments, and how each argument maps to the fields set at the end of the call.

3) Call Graph Partitioner: The second step of the test generation process is to partition the call graph into a tree of Objects and their children – i.e. if an object was *returned by* an object interaction, than the original object is the *parent* of the returned object. For instance, for the method `Mouse.answerTheQuestion(Earth)` in Listing 15, the result would be the Tree in Figure 12.

[38]Described in Section III-D, method instances are the node associated with *traced* method calls.

Each method call on a complex object is recorded in order, along with its (Optional) return value. This is because multiple method calls of the same method can sometimes return different values.

```
public class UltimateAnswer {
  final private int answer;
  public UltimateAnswer(int answer) {
    this.answer = answer;
  }f

  public int getAnswer() {
    return answer;
  }

  public String getQuestion() {
    throw new NotImplementedException();
  }

}

public class Earth {
    public int calculateAnswer() {
        return new UltimateAnswer(42);
    }
}

public class Mouse {
    public String answerTheQuestion(Earth earth) {
        UltimateAnswer answer = earth.calculateAnswer();
        return "answer: " + answer.getAnswer()  + ", question: " + answer.getQuestion()
    ;
    }
}

public static void main(String [] args) throws Exception {
    System.out.println(new Mouse().answerTheQuestion(new Earth()));
}
```

Listing 15. A fragment of source code to help demonstrate the tree formed of object interactions used to build mocks.

F. Static Analysis

Basic Static Analysis of the object is performed lazily as necessary to collect the following information through Java Reflection APIs:

1. Any *modifiers* of fields, methods, or classes. Different modifiers relate to different generated code to set them up for non-determinism. For instance, if there is a *field precondition* which is *private*, then code along the lines of Listing 16 must be generated – which forces its *set*. Likewise, for *final* fields, their Modifier must be reset to remove the Final and the class re-compiled, inside the generated test to set it to an appropriate value. A fragment to achieve this is shown in Listing 17.
2. **Object constructors and their associated arguments**, for cases where we have not traced the constructor, yet desire to generate methods for it either way.
3. The associated **throws** information for each method call so that they can be added to the *testMethod*.
4. **Getter and setter methods** are collected so we can set *private* fields without the use of reflection such as Listing 16.

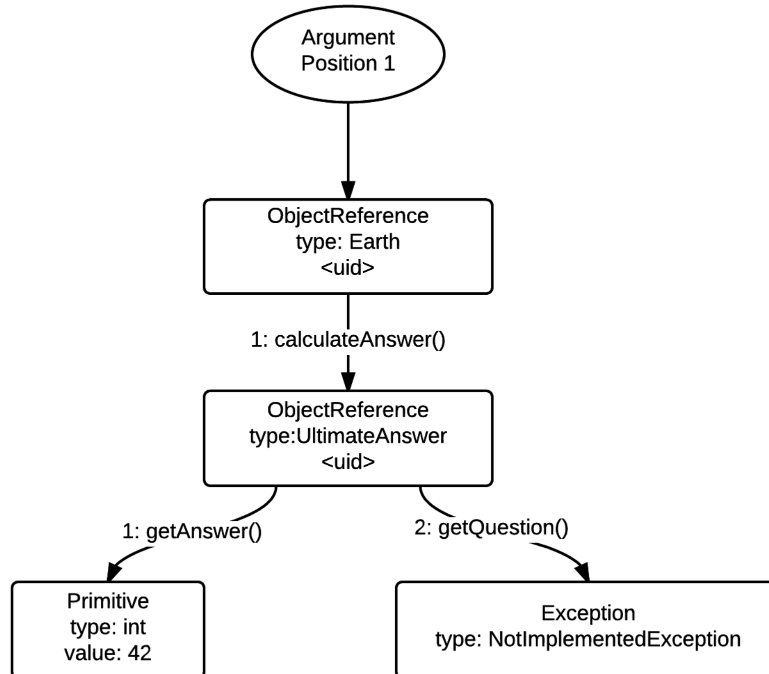

Fig. 12. The generated call graph for argument Earth.

This information is used to help generate code which compiles and correctly removes non-determinism. It is also used to try to make the naming of variables *better* – i.e. a little closer to what a human would name them (see: Section III-F4).

```
1 void setField(String fieldID, Object objectInstance, Object desiredFieldValue) {
2   Field field = objectInstance.getClass()/*.getSuperType().getSuperType() as necessary
      */.getDeclaredField(fieldID)
3   field.setAccessible(true);
4   field.set(objectInstance, desiredFieldValue>);
5 }
```

Listing 16. Generated code to force a field set on a private field

```
1 void setField(String fieldID, Object objectInstance, Object desiredFieldValue) {
2   Object desiredFieldValue;
3   Field field = objectInstance.getClass()/*.getSuperType().getSuperType() as necessary
      */.getDeclaredField(fieldID)
4   field.setAccessible(true);
5   Field modifiersField = Field.class.getDeclaredField("modifiers");
6   modifiersField.setAccessible(true);
7   modifiersField.setInt(field, field.getModifiers() & ~Modifier.FINAL);
8   field.set(objectInstance, desiredFieldValue>);
9 }
```

Listing 17. Generated code to force a field set on a final field

1) Sub Graph Analysis and Filtering: There are a number of edge cases for particular types of call graphs. These need to be correctly handled so we can generate tests for as many objects as possible.

1. **Method calls in the same instance** When one method in an object calls another method in that same object, we are unable to intercept. We must *inline* that method's dependencies so, whilst we can't stop it being executed, we can still correctly handle the method's dependencies. This is the same edge case as *Recursion* – the recursed call's dependencies gets inlined into the original.
2. **Some Opaque Method calls can be recovered** When an *opaque* method exists in a call graph that we have desired to trace, it indicates something went wrong whilst tracing it. This could be because the method has no **stack** – i.e. it was returned a constant or was otherwise particularly small. It could also be one of a selection of JPDA bugs that occur rarely, but when dealing with large traces (>500000 traced methods), occur more often then not in a test generation pass. In these cases, we want to still generate tests as well as we can. As such, if possible, we use static analysis to determine what **opaque** methods must have been called with, given their type signature and the types and values in scope at the time of the call.

Some call graphs are *untestable*. I.e. they invoked methods that we cannot intercept. There are four options we have considered to manage these call graphs.

1. We inline the *untestable* method's own call graph, and try to mock each of its own dependencies. This strategy *might* work in some cases, such as the use of *final* classes or *pure* static methods. However, it might have side effects, such as the utilisation of a Final class that interacts with the File System.
2. We use classloader interception to mock it. This would mean tests generated have to be run with an additional parameter (**"-agentlib"**) or under a modified JVM. This option would stop us from running generated code on a developers own test runners.
3. We generate tests without mocking this object, hoping this will be okay. Much of the time this will result in the generated test giving a *NullPointerException*, but this would work correctly for *pure* methods.
4. We give up and move onto a different method.

Our implementation uses option 4 for all *untestable* graphs. This is a tradeoff between our **Usability** and **Wide Variety of Input Programs** requirements (Section III-A). We think more intelligent static analysis and the use of Option 1 would be a better option, but did not complete this due to time requirements.

2) Building Unit Tests: By the point before we start generating code, we have collected the following information on the method to be tested:

1. **Field Pre Expectations** These are the values that each field used in the *method under test* has *before* the method's execution. In the case where these fields are Object References, this Value is in the form of a *Call Graph* (Figure 12), with the root node being the field itself, and each leaf being a local variable that is made into scope through an interaction with this field.
2. **Field Post Expectations** These are the values that each field used in the *method under test* has *after* the method's execution.
3. **The Constructor** of the class under test, the value of each argument and *effects*[39].

[39]In terms of fields set, and how they relate to the constructor's arguments.

4. **The Arguments** of the method call under test. If these arguments are object references, these are call graph structures alike Field Pre Expectations.
5. **The Return Value** of the method call under test.
6. **Any New Objects** created during the method call, and their interactions

This information is directly translated into a unit test. Each Pre-Condition of the method instance is generated in the *test code* for the class in the test code (Field values, constructors, arguments), the method is invoked, and the post-conditions (Field Values, Interactions, Return Values) are verified.

The code generation is managed through use of the `JCodeModel` API Section III-F3, with the use of several algorithms and techniques for each of the data structures we construct.

3) JCodeModel: The `JCodeModel` API is a programmatic way of generating source code. It contains methods that allow for the building of an Java AST programmatically. Once the AST is constructed as desired, we can output this into structured and styled source code – which we either send to `stdout` or a file hierarchy as determined by command line options (Section III-F8).

4) Naming Variables: **Usability** is highly important to the utility of our tool, and one of the core complaints against other test generation tools are their variable naming policies. An anecdotal example of this comes from [3] : a conversation between a developer and a team at Microsoft Research (Pex) trying to promote use of their automated test generation tool.

```
1 Developer: "Your tool generated a test called Foo001. I don't like it.""
2 Pex team: "What did you expect?"
3 Developer: "Foo_Should_Fail_When_The_Bar_Is_Negative."
```

Users need to interact with generated tests, and thus the naming conventions of the generated tests should follow that of manually written ones.

When developers write test methods manually, they use meaningful naming conventions for declared variables and test methods. This significantly aids the readability of the tests by joining the meaning and implementation of each variable.

With this in mind, we have 3 strategies that we use to assign names to declared variables.

1. If the referenced object for the local variable is ever set to a field in the target test class, then its name becomes the same as the field name. We call this strategy **Field Propagation** because it looks forwards and backwards in time for assignments of this object to a field.
2. If we have information on local variable names assigned to object references that we pass around, then this becomes the name of the variable. This strategy is lower priority than (1).
3. If we have no such information, we name the variables based on their type signature in camel case; e.g. `Foo foo`, `int i`, `FooBar fooBar`.

These strategies are augmented with additional information should we use them in a particular way or in case of conflicts.

1. If a variable is set to a mock object in our test, then its name becomes `camelCase("mock" + originalName)`. I.e. `Foo mockFoo = mock(Foo.class)`.
2. If a variable name is conflicted by another variable declaration, then an ID is applied to the one used later. This ID is an incrementing integer.

5) Arrays: Array datatypes are similar to complex objects, in that we represent them as a tree to handle deeply nested arrays (see Figure 12 for reference on how we handle objects).

With an array, we try to be conservative in our generated code. If we only access particular indices of our array, then every indice but for the accessed one is made to be a default value (e.g. `null` for object references).

For instance, if, in our exercising of the code a method call was made to foo like: `foo ([["1","2"],["1","2"],["1","2"]])`, but no value was accessed: `void foo(String[][] args){ this.val = args;};`, then the generated test code would look like Listing 18.

```
1  public void testFoo() {
2
3    // Generate the arguments
4
5    String[][] val = new String[3][];
6    String[] stringArray = new String[2];
7    stringArray[0] = null;
8    stringArray[1] = null;
9    String[] stringArray1 = new String[2];
10   stringArray1[0] = null;
11   stringArray1[1] = null;
12   String[] stringArray2 = new String[2];
13   stringArray2[0] = null;
14   stringArray2[1] = null;
15   val[0] = stringArray;
16   val[1] = stringArray1;
17   val[2] = stringArray2;
18
19   // Construct the A class
20
21   A a = new A();
22
23   // Invoke the method
24
25   a.foo(val);
26
27   // Assert the fields are what we expect
28
29   assertEquals(a.val, val);
30 }
```

Listing 18. Generating array test input

6) Primitives and Immutable Objects: Primitive and Immutable objects are treated differently to complex objects. Primitives and Strings[40] can be instantiated directly, without calling the original object. Likewise other immutable objects do not need to be made into a *mock* – we can instantiate them directly should they be flagged as such. This lowers the *Brittleness* of the code (Section IV-A1).

Automatically detecting immutable objects is a possible extension to simplify the generated tests. This could be done through a simplistic static analysis to ensure that all fields are *final* and all methods only return a field.

[40]Strings are immutable in Java

7) Mocking Complex Objects: Generating mocks for complex objects is one of the more convoluted algorithms in our applications. The principle however is quite simple. In Figure 12 we show the data structure that is used for complex objects. Generating a mock on this data structure can be made into a 5 step process, that operates on a *node* in this tree.

1. We **recursively** build all the objects that are children of the object/node we desire to build. From this recursive call, we collect a *reference* that is associated with the variable declaration that the child objects will be generated into.
2. We declare and instantiate the *mock* variable associated with the current node. If this variable is a primitive or string reference, we generate it and stop. If it is an array, we instantiate that with syntax in Section III-F5.
3. We collect the *Access Events* associated with the object, as necessary, and merge these into a `Map<AccessEvent, List<ReturnValue>>` structure. An `Access Event` is a method call with a unique set of object values (or array reference). I.e., if an object `class Foo {int bar(String);}` has three calls made to `bar` during a single *test method* of `bar("1"),bar("1"),bar("3")` returning `1,2,3` respectively, then this map would look like `{bar("1")->[1,2];bar("3")->[3]}`.
4. Once we have collected all Access events for a given method call, we generate a series of `whens` on each *mock* object from this `Map<AccessEvent, List<ReturnValue>>` structure. I.e. `{bar("1")->[1,2];bar("3")->[3]}` would become `when(mockFoo).bar("1").thenReturn(1).thenReturn(2);when(mockFoo).bar("3").thenReturn(3);`

8) Usage: The Generator module is invoked is a Java executable with the following command line arguments:

```
$> java -cp generator.jar:example-program.jar generator.Main
 -all                            : Produce all possible tests, as opposed to one
                                   per method
 -classes VAL                    : Input a list of classes to try to generate
                                   tests for
 -d                              : Debug mode
 -db VAL                         : Database Location
 -dump                           : Dump all tests
 -input [Neo4j]                  : Input Mode
 -mock [Mockito]                 : Mocking mode
 -output [STDOUT | BOTH | FILE]  : Output mode
```

There are just *three* important command line options necessary to generate tests. `-dump`, `-output`, and `-db`. The other options allow for a little extra configuration in the format of generated tests.

`-dump` indicates that we actually want to produce tests. It is a necessary flag to start the process, used because it will potentially write a lot of files to disk.

`-output` indicates the preferred output mode of the tests. `-output STDOUT` indicates that they should be simply printed to standard out. `-output FILE` indicates that they should be written to disk in the equivalent folder hierarchy to each tests package. I.e. if a *class under test* is within the `com.mytest` package, then the test will be generated in the `./com/mytest` folder. `-output BOTH` indicates that both of these should occur – the class will be put into standard out and written to disk.

`-db` indicates the trace database location. Like the `recorder`, this could be of form `-db /tmp/db` to indicate that it will read from the database existing in the `/tmp/db` folder.

On top of these options, there are 2 options which currently do not do anything (`-input` and `-mock`), as they have only one configurable value thus far. `-d` can be used to provide verbose printing

G. Implementation Walkthrough by Example : ABC

To illustrate the implementation of a tracer, we look at the graph generated for the following toy example, named *ABC* (this was in fact our earliest test case). The source to this toy application is available in Listing 19.

```
1  public class Main {
2      public static void main(String [] args) throws Exception {
3          A a = new A();
4          B b = new B();
5          C c = new C(a,b);
6          c.callsBWithA();
7          System.out.println("Everything went better than expected");
8      }
9  }
10
11 public class A {
12     public int makes_a_nice_call() {
13         return 42;
14     }
15 }
16
17 public class B {
18     public int makes_a_almost_nice_call(A a) {
19         return 1 + a.makes_a_nice_call();
20     }
21 }
22
23 public class C {
24     private final B b;
25     private final A a;
26
27     public C(A a, B b) {
28         this.a = a;
29         this.b = b;
30     }
31
32     public int callsBWithA() throws Exception {
33         int answer = b.makes_a_almost_nice_call(a);
34         if (answer < 42) {
35             throw new Exception("Not really a good answer");
36         }
37         return answer;
38     }
39 }
```

Listing 19. The source code for the ABC example

In the provided code, three classes are instantiated. The first two, `A` and `B` take no arguments, and have no fields. The third `C` takes an `A` and a `B` object, and sets them to a final fields. Then, the method `callsBWithA` is called, which then calls `b.made_a_almost_nice_call(a)`, which then calls `a.makes_a_nice_call()`, who return `42`, `43`, `43` respectively. After this a call to `System.out.println` is made with the argument `"Everything went better than expected"`.

If we walked through the source code from the entry point, looking line by line at what will happen, we could build the *sequence diagram* shown in Figure 13

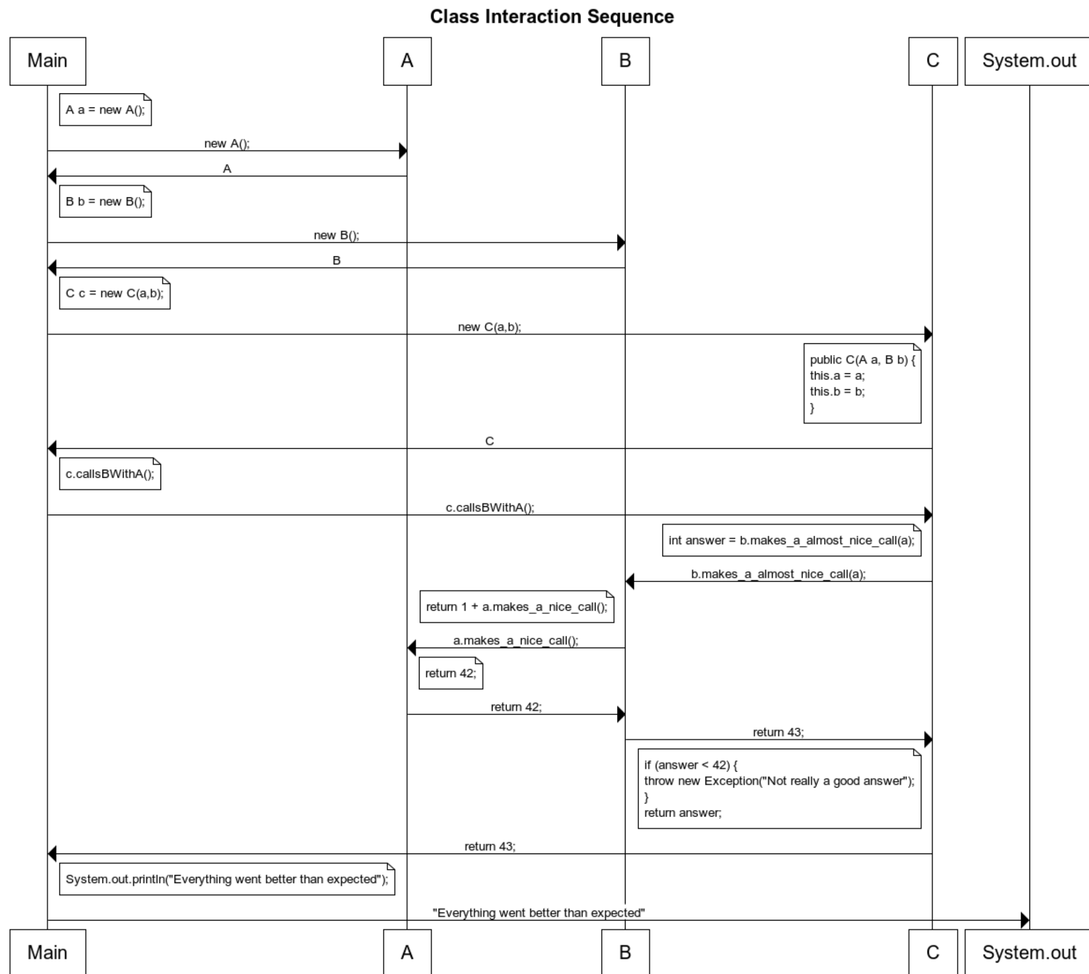

Fig. 13. The sequence diagram generated through walking through the entirety of the ABC Test

The tracers objective is to walk through a sequence of events like that given in the sequence diagram above, and build a graph of interactions, that contains enough information to generate tests from:

Each node in the Figure 14 is what we denote a *Method Instance* – i.e. it represents a method call in the original program. The node **3** can be seen to be the entry point, as it only calls other objects, and never gets called itself. The **3->9->14** chain is next, which illustrates the **new A()** call. The first node, **9**, represents the `A<init>()` call, which firstly calls its supertype constructor `Object<init>()`. Likewise the `3->20->23` chain represents the `B<init>()` default constructor. The `3->29->32` chain is the `C<init>(A,B)` constructor. The `3->39->42->45` chain represents the sequences of calls through `C.callsBWithA()` and the `3->52` chain represents the call to `System.out.println(String)`.

If each node were expanded to visualise the other node types, we would also see the `Arguments` to each function, the `Return Value`, and the value of each object instance (that which is referred to by

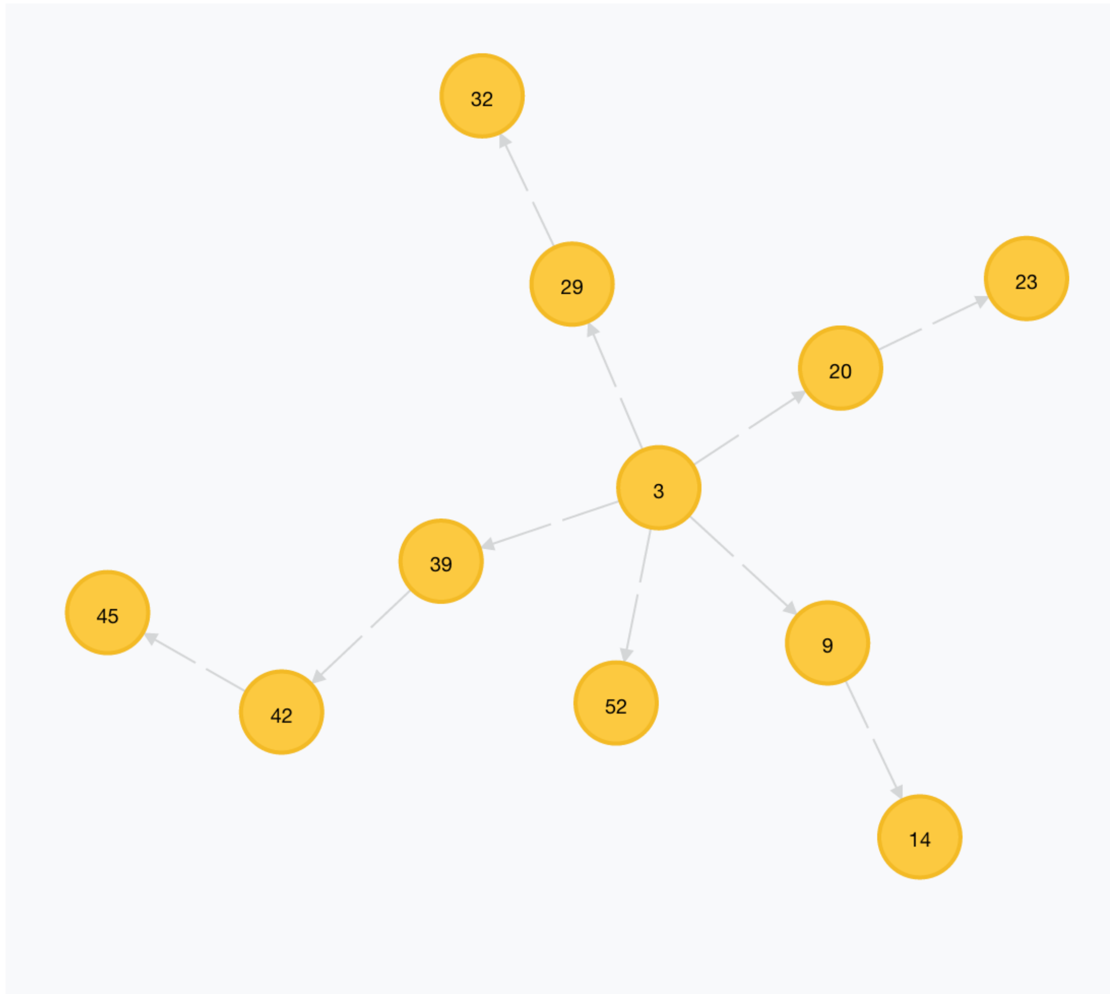

Fig. 14. The method call graph generated from exercising the ABC Example

the `this` keyword) – the object which each method is called on. These Values are indexed – for each object interacted with in the JVM, it only ever has a single node. This allows us to directly follow the sub-trace of a method call through looking only at the *method instances* which it directly calls, and their arguments, *return value* and *this parameter*. If we seek to construct one of these objects, we follow through the objects *method instance* to its *this* parameter. This *this* parameter has a reference to all methods called upon it, the first of which will be the *constructor*. As such, we can get the *constructor* each of an objects dependencies was initially called with in only 3 steps. This full graph is provided in Figure 15 for illustration: we will not enumerate the meaning and values of each node for brevity.

To test an object's method call, we look at that objects *this* parameter in the graph, and jump to its initial constructor. We can then instantiate that object with the same constructor call, set each of the fields to their *state* just before the method call, and set the arguments to that method call to their value at the time of the original call. Any complex objects are replaced with *mock objects* which replay the

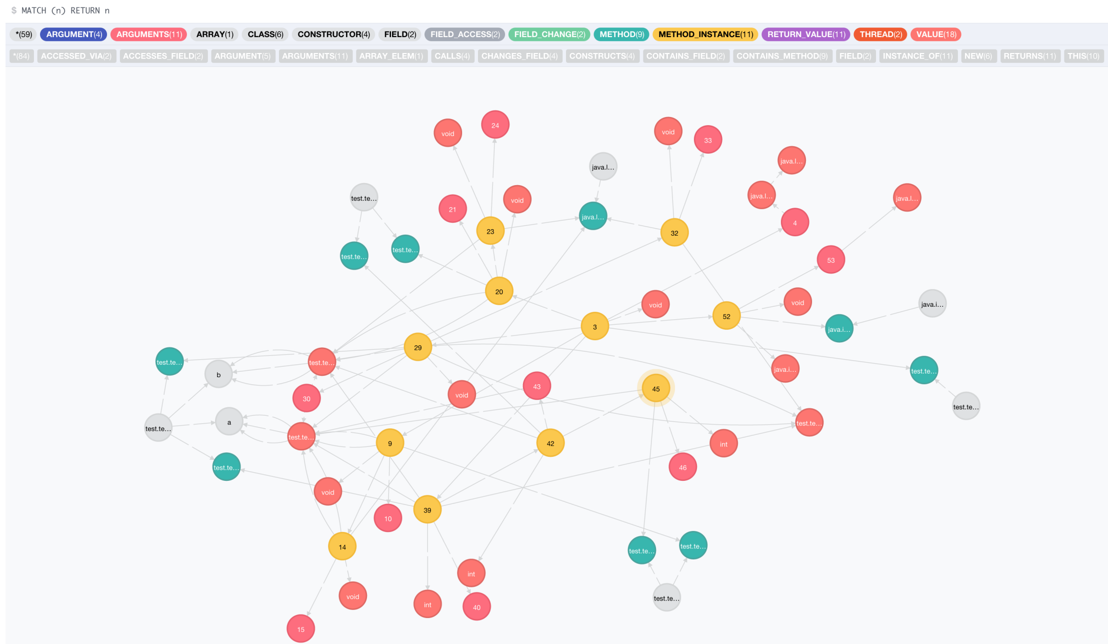

Fig. 15. The full graph generated from exercising the ABC Example

interactions made with the original object. After this, the test is fully deterministic given the assumption that the original method is *testable* – i.e. does not create/use non-deterministic objects with *new* or use non-deterministic static methods. See Listing 20 for the full generated test suite.

```
public class TestA {
    @Test
    public void testConstructor() {
        f
        // Invoke the method

        A a = new A();
    }

    @Test
    public void testMakes_a_nice_call() {

        // Construct the A class

        A a = new A();

        // Invoke the method

        int i = a.makes_a_nice_call();

        // Assert the return value

        Assert.assertEquals(42, i);
    }
}

```

```
27 public class TestB {
28
29
30     @Test
31     public void testConstructor() {
32
33         // Invoke the method
34
35         B b = new B();
36     }
37
38     @Test
39     public void testMakes_a_almost_nice_call() {
40
41         // Generate the arguments
42
43         A mocka = Mockito.mock(A.class);
44
45         // Construct the B class
46
47         B b = new B();
48
49         // Link Mocks
50
51         Mockito.when(mocka.makes_a_nice_call()).thenReturn(42);
52
53         // Invoke the method
54
55         int i = b.makes_a_almost_nice_call(mocka);
56
57         // Assert the return value
58
59         Assert.assertEquals(43, i);
60
61         // Assert that our method interactions are what we expect.
62
63         Mockito.verify(mocka, Mockito.times(1)).makes_a_nice_call();
64     }
65
66 }
67
68 public class TestC {
69
70
71     @Test
72     public void testConstructor()
73         throws IllegalAccessException, NoSuchFieldException
74     {
75
76         // Generate the arguments
77
78         A mocka = Mockito.mock(A.class);
79         B mockb = Mockito.mock(B.class);
80
81         // Invoke the method
82
83         C c = new C(mocka, mockb);
84
85         // Assert the fields are what we expect
```

```
86
87         Field field = c.getClass().getDeclaredField("a");
88         field.setAccessible(true);
89         A a = ((A) field.get(c));
90         Assert.assertEquals(mocka, a);
91         Field field1 = c.getClass().getDeclaredField("b");
92         field1 .setAccessible(true);
93         B b = ((B) field1 .get(c));
94         Assert.assertEquals(mockb, b);
95
96         // Assert that our method interactions are what we expect.
97
98     }
99
100    @Test
101    public void testCallsBWithA()
102        throws Exception
103    {
104
105        // Generate the fields
106
107        A mocka = Mockito.mock(A.class);
108        B mockb = Mockito.mock(B.class);
109
110        // Construct the C class
111
112        C c = new C(mocka, mockb);
113
114        // Link Mocks
115
116        Mockito.when(mockb.makes_a_almost_nice_call(Mockito.eq(mocka))).thenReturn(43);
117
118        // Invoke the method
119
120        int i = c.callsBWithA();
121
122        // Assert the return value
123
124        Assert.assertEquals(43, i);
125
126        // Assert that our method interactions are what we expect.
127
128        Mockito.verify(mockb, Mockito.times(1)).makes_a_almost_nice_call(Mockito.eq(
    mocka));
129    }
130
131 }
```

Listing 20. The code generated by running the ABC trace through the `generator` module.

H. Usage Walkthrough by Example

```
1 public class A {
2     public int change(String a) {
3         int i = new Integer(a);
4         i *= 100;
5         return i;
6     }
7 }
```

```
8  public class Main {
9      public static void main(String[] args) {
10         int i = new A().change(args[0]);
11         System.out.println("Out: " + Integer.toString(i));
12     }
13 }
```

This toy example involves the use of one class and an entrypoint. `Class A` has a single method which we want to test. It takes in a String argument, converts it to integer form, multiplies it by 100 and returns it.

It is exercised by `Class Main`, which passes `Class A.change(String)` the first command line argument, then prints the result to standard out.

Firstly, the application is run under instrumentation with the `recorder` module.

The effect of this command is a `standard output` trace of what happened, along with a populated Neo4j database containing the graph of what occurred.

```
1  $> java -cp recorder.jar:examples.jar recorder.Main -d -output Neo4j -db /tmp/db -
       target test.test6.Main -args 15
2
3  ThreadStartEvent in thread main
4  | MethodEntryEventContext to Method[test.test6.Main.main(java.lang.String[])] with
       Arguments[java.lang.String[] args = ["15"|UNIQUEID=126]]
5  | | MethodEntryEventContext to Method[test.test6.A.<init>()]
6  | | | MethodEntryEventContext to Method[java.lang.Object.<init>()]
7  | | | MethodExitEvent@java.lang.Object:37 in thread main
8  | | MethodExitEvent@test.test6.A:6 in thread main
9  | | MethodEntryEventContext to Method[test.test6.A.change(java.lang.String)] with
       Arguments[java.lang.String a = "15"]
10 | | | MethodEntryEventContext to Method[java.lang.Integer.<init>(java.lang.String)]
       with Arguments[java.lang.String str_0 = "15"]
11 | | | MethodExitEvent@java.lang.Integer:868 in thread main
12 | | | MethodEntryEventContext to Method[java.lang.Integer.intValue()]
13 | | | MethodExitEvent@java.lang.Integer:893 in thread main
14 | | MethodExitEvent@test.test6.A:10 in thread main
15 | | MethodEntryEventContext to Method[java.lang.StringBuilder.<init>()]
16 | | MethodExitEvent@java.lang.StringBuilder:90 in thread main
17 | | MethodEntryEventContext to Method[java.lang.StringBuilder.append(java.lang.String)]
        with Arguments[java.lang.String str_0 = "OUT: "]
18 | | MethodExitEvent@java.lang.StringBuilder:137 in thread main
19 | | MethodEntryEventContext to Method[java.lang.Integer.toString(int)] with Arguments[
       int int_0 = 1500]
20 | | MethodExitEvent@java.lang.Integer:403 in thread main
21 | | MethodEntryEventContext to Method[java.lang.StringBuilder.append(java.lang.String)]
        with Arguments[java.lang.String str_0 = "1500"]
22 | | MethodExitEvent@java.lang.StringBuilder:137 in thread main
23 | | MethodEntryEventContext to Method[java.lang.StringBuilder.toString()]
24 | | MethodExitEvent@java.lang.StringBuilder:407 in thread main
25 | | MethodEntryEventContext to Method[java.io.PrintStream.println(java.lang.String)]
       with Arguments[java.lang.String str_0 = "OUT: 1500"]
26 | | MethodExitEvent@java.io.PrintStream:809 in thread main
27 | MethodExitEvent@test.test6.Main:10 in thread main
28 ThreadDeathEvent in thread main
29 ThreadStartEvent in thread DestroyJavaVM
30 ThreadDeathEvent in thread DestroyJavaVM
31
```

```
32 ----------
33 Time: 0.756790721 seconds
34 ----------
```

By looking at this trace, we can see what happened under the scenes. There are a few interesting things to note:

1. In the trace the values of each argument and return value is given. This will occur for all methods under instrumentation, and is important as it is our *input test data* for each unit test. However, this can be prohibitively expensive to print to standard out, so it is disabled by default – the `-d` flag re-enabled it.
2. In line 12 of the trace one can see that there is a *MethodEntry* into `Integer.intValue`, for the `int i = Integer.getInteger(a);` (line 3) in the originating source. This is an implicit conversion from the boxed type `Integer` to the `int` primitive – though there is in the Java language a 1:1 correspondence between the two as Integer itself has immutable value. This highlights some of the edge cases we have to handle. By default, if we attempt to get the `Value` of the `Integer` class, we get a `UniqueID` – a unique value that we use to keep track of object references so we can simulate them later. When generating tests, this causes an issue because a method might return or take an Integer value, and we would like to compare this against a particular primitive. To do this, we actually have a special case for the `Integer` object and look into its `private final int value` field to get the actual value. Similar edge cases exist for all other primitives, strings, and arrays.
3. Lines 15 through 24 of the trace show that `"Out: " + Integer.toString(i)` expression is simply syntactic sugar for the creation of a `StringBuilder` object to add these two strings together. It is effectively converted into `new StringBuilder("Out").append(Integer.toString(i)).toString()`.
4. This example takes `0.75` seconds to run, which is almost all spent in VM initialisation.

The `generator` module converts this trace into the test for class `A`.

```
1 $> java -cp generator.jar:examples.jar generator.Main -db /tmp/db -dump
```

```
1  public class TestA {
2      @Test
3      public void testConstructor() {
4
5          // Invoke the method
6
7          A a = new A();
8      }
9
10     @Test
11     public void testChange() {
12         // Construct the A class
13
14         A a = new A();
15
16         // Invoke the method
17
18         int i = a.change("15");
19
20         // Assert the return value
21
```

```
22          Assert.assertEquals(1500, i);
23      }
24
25  }
```

As no external object interactions occur, this test merely becomes a test of the data-in of `"15"` to data-out of `1500`.

It is worth noting that the initial implementation of this test had a bug. The following fragment shows the bugged version (change at line 4):

```
1  public class A {
2      public int change(String a) {
3          // int i = new Integer(a);
4          int i = Integer.getInteger(a);
5          i *= 100;
6          return i;
7      }
8  }
```

The `Integer.getInteger(String)` method does not actually convert the String to an Integer, instead looking up the given `String` in the system *environment variables* map to get an Integer value. As such, it causes a null pointer exception when the `intValue()` method is called.

The test generated reflects this, expecting the `java.lang.NullPointerException.class`, and passes.

```
1  public class TestA {
2      @Test
3      public void testConstructor() {
4
5          // Invoke the method
6
7          A a = new A();
8      }
9
10     @Test(expected = java.lang.NullPointerException.class)
11     public void testChange() {
12
13         // Construct the A class
14
15         A a = new A();
16
17         // Invoke the method
18
19         a.change("15");
20     }
21
22 }
```

IV. Limitations and Utility

In this section I shall explore some of the key limitations inherent in both the theoretically perfect implementation of the ideas presented here, as well as some of the limitations of our current implementation. I also examine some of the possible reasons why developers would want the tests generated by our tool.

A. *Limitations*

The perfect implementation of our ideas is a direct translation from a higher level test suite into the form of many lower level tests. In this section we detail the most important limitation to this technique – one of *Brittleness*. On top of this there are two further limitations that we do not believe are solvable without extending the technique

1. **Large Test Inputs** One of our test cases for our tool was a student project for a machine learning library. It took as input a large `Integer[][]` (~10000 elements in size). We managed to successfully generate a test input using this test data, but we do not consider such a test highly *useful* because it is not of the sort that a developer would produce manually[41]. It would be good if we could dynamically reduce the size and complexity of test inputs in such a way that the *coverage* of our tests was not affected, however this would break one of the core advantages of our technique; using test data from an already existing execution ensures that the test data is of form that may occur during actual execution.
2. **Untestable methods** If a method includes non-deterministic method calls that cannot be intercepted, non-determinism will be introduced in any generated test.

1) False Positives and False Negatives: Brittleness: An important question to ask is can this translation produce *false positives* or *false negatives*. A *false positive* would occur when the generated test **passes** yet the test which it is generated from **fails**. A *false negative* would occur when the generated test **fails** yet the test which it is generated from **passes**.

Each of these attributes would limit the utility of our technique. In principle, for *testable* methods (see Listing 3 in Section II-A for a definition of *testable*) our tests do not have false positives or false negatives **on the version for which tests are generated**. However, when run on a different version of the code, these tests might become out of date. For instance, the type signature of a method might change, or it begins to interact with an additional field or take another parameter. When this occurs, our tests will start producing **false negatives**, because they test and provide all information for the *initial* version of a method, but not the *update* version.

We call this attribute *Brittleness*, because these tests are not very tolerant of changes. The system as a whole might have correct behaviour, but our tests indicate that something is wrong.

This problem is one which occurs whilst building tests manually, just as it does when building them automatically. The difference is that when tests are built manually, it would take significant effort to rebuild them to allow for a change. One can imagine a workflow where developers would use our tool to avoid this – if the behaviour of the program does not change according to the system tests, the broken tests are simply regenerated.

We believe this property of *Brittleness* would be mitigated with the use of *Test Trees* (Section VI-A), however this is an idea we have not validated yet.

B. *Utility*

An important question to consider is the *utility* of tests generated. We describe four applications where we believe our technique can be applied to increase developer productivity.

[41]To test this kind of input, the developer would use much smaller sample sizes such that the test executes in reasonable time and writing the test does not take too much effort

1. **Additional Information for Failure Diagnosis**
2. **Speeding up integration test suites**
3. **Automating UI testing**
4. **Bootstrapping manual test production**

1) Additional Information for Failure Diagnosis: When an integration test fails, the error message will indicate one of two things:

1. The integration test reached a terminating state during execution. For example, a `NullPointerException`. This will be generally be accompanied with a *stack trace* for locational information on the problem.
2. Some property which the integration test was testing was not correctly achieved.

In the case of (1) – the *cause* of the bug will often be quite far from the exception location. With the use of our technique, divergences in the execution trace from a *working* version can be identified. This may help to narrow the search space a developer would look through to discover the bug.

In the case of (2) – the search space to look through to find the bug is the entirety of the integration test, potentially filtered by looking at *differences* in source code. With our technique, this would isolate the set of *changes* to a discrete set of locations, and provide a quick way of re-running each of them to be manually verified for correctness.

2) Performance Improvements in Integration Test Suites: Integration tests can often take a long time to run. The tests that our tool generates from the integration test allow for this integration test to be effectively run *in parallel*, with each *parallel* section executing the logic within one class. We imagine that one could calculate if a *modified* version of a program should pass its integration test, by looking at the changes in the interactions between its components in parallel.

With the addition of the *Test Trees* concept (Section VI-A), false "failures" could be removed by walking and verifying higher level tests which exercise more components. False "positives" could be removed by generating tests to ensure that each *mocked* interaction can still occur. We believe all this is is achievable with our technique, though it would take non-trivial engineering effort to ensure that our implementation was able to achieve this.

3) Bootstrapping manual test production: Many IDEs perform a simplistic test generation process which automatically creates, stubs out and names test methods for a given class. One of the low-hanging-fruit to using our work would be to enhance this stubbing by providing our tests. Developers could then modify test input data through changing *mock configurations* and arguments. They could also write more meaningful assertions based on the intended behaviour of the code. Starting from our generated tests might be a better baseline to bootstrap manual testing than what is currently provided.

4) Automating UI Testing: One of the experiments we performed was to show that we could generate an automatic UI test from interactions with a *traced* GUI. We ran a student project – a simple Java game[42] – under our `recorder`, manually interacted with it, then closed it. UI interactions pass control to user code through method interactions at defined handlers. Our generated test code replicated these method calls – with *mock* event objects instead of the instantiated ones. As we also traced the internal APIs used to construct the GUIs, tests for these were also constructed. Running these tests caused a

[42]This application used the SWING Java GUI package.

sequence of UIs to be created and destroyed for each interaction we made in the original recorded execution.

Our generated tests do not follow the *lifecycle* of an object – the sequence of all method interactions with it across its lifetime. Many UI libraries require a sequence of method calls in a precise ordering to start and configure any UIs. Because of this ordering, and because we replace internal state with mock objects, this is not sufficient to capture and replace UI interactions for every library. However we consider it conceivable to build an extension to our project which captures and replays entire orderings of event calls across an objects lifecycle – such as every call through a View interface in a MVC (Model View Controller) application. This might allow one to produce a series of automations just from manual interaction with a Java program.

V. Evaluation

In this chapter we intend to answer three questions:

1. How much of an execution trace can we transform into deterministic test programs?
2. How does the tool scale?
3. How good are the tests?

We ask these questions because they directly effect the usefulness of our technique.

Questions (1) and (2) can be evaluated quantitatively: (1) can be answered by looking at the *coverage* information of the originating execution trace and the *coverage* of the test cases generated from that (Section V-C) . (2) can be answered with a performance evaluation (Section V-D).

Question (3) is difficult to be evaluated quantitatively. One measure of looking at how good the tests are can be done through looking at *coverage* information. However, as our coverage directly comes from the originating execution trace we consider this not a good measure. Another quantifiable question would be: “can we speed the discovery of bugs in comparison to an already existing integration test”. This is a difficult metric to collect – it would require a user testing study. With respect to time-constraints, we present a purely subjective evaluation. Throughout this report we show several examples of programs and their generated tests – further examples are also found in Section VIII. In Section IV-B we describe some use cases for our generated tests, which would require more engineering effort to implement and evaluate. We belief these demonstrations and analysis consitute

A. Test Configuration

All measurements were performed on a 2.3 GHz Intel Core i7 processor (8 cores) with 16 GB 1600 MHz DDR3 memory. The operating system was OSX version 10.10.2. The Java version was 1.8.0_25 using Java HotSpot(tm) 64-Bit Server VM.

B. Benchmark Suite

The benchmark suite we used contains a mix of projects selected randomly from SourceForce (the SF100 corpus [12]), miscellaneous student projects, and a number of synthetic examples we custom built to show the strengths and weaknesses of the tool.

Of those in the SF100 corpus, we filtered them down to 9 test cases. This was due to the constraint of our tool: we must have a way to *exercise* the project. As such, we filtered the corpus to those that had pre-existing *entrypoints* (main methods) which acted as integration tests.

In our own testing we demonstrate the generation of unit tests from UI interactions. This interaction is not discussed or quantified, as we did not have the capability to automate it.

C. Coverage Evaluation

In this section, we aim to answer the question: "How much of an execution trace can we transform into deterministic test programs?"

To do this, we aim to quantify the difference between the coverage of the original execution trace and the coverage of our generated tests.

Code coverage is a metric for how much of the program has been exercised. It is calculated by tracing in a similar manner to how we do it in Section III-C, but without halting the VM. We use the coverage runner `jcov` – one maintained by *Oracle*, to collect this data.

The specific metrics we collect for each of our test cases are:

- Number of Covered Classes : How many classes were instantiated during the exercise
- Method Coverage : How many methods were entered into during the exercise
- Block Coverage : How many blocks were entered into during the exercise
- Branch Coverage : The number of conditional branches walked down during the exercise
- Line Coverage : The number of lines of source code that were executed.

A positive evaluation for us is when each of the metrics is *similar* in the generated unit test and the original test execution which spawned it. A non-nonsensical evaluation would be one where we achieve greater code coverage than the original execution.

For 2 of our test cases, we do not have the *Line* coverage metric, because it requires the source code along with the compiled program.

For 1 of our test cases (`fizzbuzz`), we do not have *Block* or *Branch* coverage. This is because we used the `Emma` coverage runner on this test case, as `jcov` would not correctly work with its `Spring` dependency[43].

Our corpus for these tests excludes our own hand-written tests. This is because our these tests were used as our own test cases during the building of our implementation, and we have ensured that we are at the *theoretical optimum* implementation for each of those. To eliminate any bias, they are not included in the resultset.

Instead, we use 6 tests from the SF100 corpus – those that we managed to run through `jcov`, and `fizzbuzz enterprise edition` – an open source `Spring` application.

To collect code coverage results we performed the following steps:

- Run the program with `jcov` – a Java Coverage Analyser – with the specified entrypoint

[43]Spring is a Java library used for (amongst other things) managing dependency injection with XML.

- Run the program with our test generator to output a series of test files
- Run each test with `jcov` to collect the code coverage for the generated tests.
- Filter the coverage results to remove the effect of the entrypoint.[44]

1) Results: We present our code coverage results in Table I and Figure 31.

The table shows that, for 85% of method calls that are executed in the original trace, we have been able to generate a test which provides the same branch/line coverage deterministically.

In Figure 18, it can be seen that line coverage is similar for each evaluated project. Likewise branch coverage can be compared in Figure 19.

Figure 16 and Figure 17 show the overall class and line coverage in each test compared to the execution trace.

These figures plot the proportion of coverage results achieved against the maximum possible coverage on the *y-axis*. On the *x-axis* each project that was evaluated for that metric is listed.

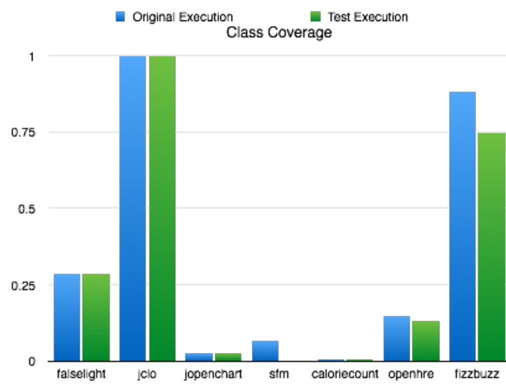

Fig. 16. The differing class coverage results proportionally

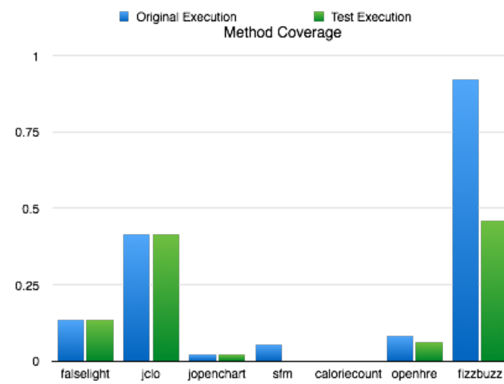

Fig. 17. The differing method coverage results proportionally

2) Analysis: The results in Table I and the graphs above show the differing coverage results in our test corpus.

Graphically, these do show that our generated tests are *similar*. In 4 of 7 cases we have managed to create tests at our theoretical optimum – for every method call there is an equivalent test which deterministically runs it. In 1 of 7 cases we do not manage to generate tests at all, and in our largest project we have achieved test cases which provide 85% of the coverage of the original execution.

In general, we think these results show that we have achieved our aim of generating tests from execution traces successfully. However, there are a few anomalies. We enumerate these below:

a) Low Coverage in Original Executions: The first thing to note is that the covered items in the original execution is much lower than we expected. This is due to the sample corpus. The original corpus (SF100) comprises of random open-source projects from SourceForge. From these we filtered

[44]This step is performed because our test generator will generate a test for the `Main` method. This otherwise causes *bias* in our results because we effectively cover each method at least twice – once inside the Main method test (which effectively re-runs the entire integration test again), and once in the isolated test.

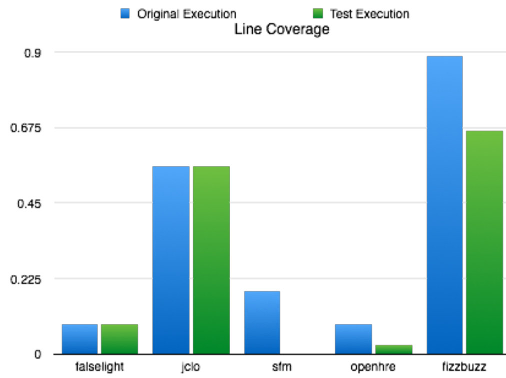

Fig. 18. The differing line coverage results proportionally

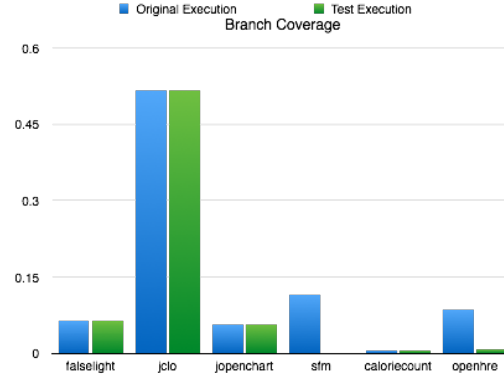

Fig. 19. The differing branch coverage results proportionally

them those that include of an entrypoint named something like "Test" or "Example". Whilst they fulfil the quality of being an unbiased representation of open source programs, this means they were not necessarily built with good coverage in mind. In fact, when looking into their source code, most of them are only exercise small utility methods.

b) Difference in Code Coverage: In most cases, the tests generated have *exactly* the same code coverage as the original execution. This is the theoretical optimum, and what we should achieve when we do not run into one of our implementation's edge cases.

The cases where we do not manage to do this are `sfm`, `openhre`, and `fizzbuzz`. In `sfm`, one of the classes that was generated failed to compile. This was due a bug in our implementation of variable name declarations (it tried to name a variable "[B"). This bug has since been fixed.

In the case of `openhre` and `fizzbuzz`, the reason is primarily due to `untestable` methods (see Section II-A). Rather than generate a non-deterministic test, we just don't generate a test.

3) Validity of Results: The results generated here have two main arguments for the validity of our conclusion that "our tool is capable of consistently achieving similar code coverage in its generated tests to the code coverage in the exercised program":

1. They demonstrate the effectiveness of the tool on open source code using a testing corpus which our tool has not been previously exposed to.
2. They provide a quantitative means of measuring this conclusion: "We can test ~82.5% of methods called from a trace exercising them".

However, there are several reasons why just this result is not enough to justify our conclusion

1. The testing corpus is not large enough. In total we have only ran the tool over 82 classes (177 methods), and many industrial Java applications have many times this amount[45].
2. 80% cannot be considered good enough to say we have "similar" code coverage.

[45]Originally, we intended to run our evaluation using *Apache hbase*'s end-to-end test suite. This test suite would have exercised approximately 2600 classes. Unfortunately, due to time constraints, we did not manage this.

D. Performance Evaluation

In this section we aim to answer the question "How does the tool scale?".

We do this through collecting results on the performance of our tool on our benchmark suite. We look at the speed of the various modules, and try to evaluate the performance *relative to the execution time of the original trace*.

We collect the following metrics:

- Time of Normal Execution
- Time to Trace Normal Execution using our `recorder` module.
- Time to Generate Tests

From these results, we perform the following transformations to reduce to a single metric:

1. We remove startup time (for large programs this startup time is irrelevant). This is done via the subtraction of a "Hello World" program from each metric.
2. We divide the time to run the two modules by the time of the normal execution. This will produce a relative slow down metric.

Our benchmark suite contains our filtered SF100 corpus, along with a 9 synthetic examples we hand wrote to test our tool. Some of these synthetic examples show anomalous results, which demonstrate the effectiveness and limitations of these measurements in answering our scalability question.

It is important to note that our *Traced Normal Execution* or *`recorder` module execution time*, is set in the mode of tracing everything. This is important to ascertain the overall performance hit, but does not demonstrate the effect of our *filters* (see Section III-C5, argument **-c**). Any code that is not traced runs at much greater speed (~0.1x original speed as opposed to ~0.01x). This is demonstrated in Section VIII-B.

Each time metric contains the mean execution time of 10 executions of the same command. Before this, two executions are made and discarded to reduce the *cold/hot* JVM effects[46]

1) Results: We present our performance testing results in Section VIII-F, Table II. Figures 20, 21, and 22 show the graphical representation of this result.

Our results table shows that the performance cost of running the `recorder` to generate tests is ~10-1000x per traced class. The mean cost of our `Recorder` module is a 286.5x slowdown. The mean cost of our `Generator` module is a 40x slowdown. Untraced classes have a mean cost of a ~33x slowdown (Figure 30).

To put this in proportion, An integration test that takes 30 seconds to run will take ~2.5 hours to generate tests for. An integration test that takes 30 minutes to run will take ~6 days to generate tests.

[46]The JVM gradually speeds up running code through profile guided optimization strategies. This means that if an execution is completed multiple times in a row, it is highly likely that the first result will be anomalously low, as it is before any optimizations have occurred. Hot is the name given to an optimized VM running at full speed. Cold is the name given before any such optimizations have been applied.

If however, only one small component was traced[47], then a 30 second integration test would take 15 minutes to generate the test, and a 30 minute integration test would take ~16 hours to generate the test.

The largest full scale test was to generate tests for a 40 minute (normal execution time) procedure. Generating tests for this procedure took 2 dedicated `Intel(R) Xeon(R) E5-4650` CPUs, 2-4Gb of RAM and 150Gb of disk space. It took approximately 3 days to collect the trace. It then took approximately 30 minutes beyond this to generate the tests. Further details can be seen in Section VIII-D.

Fig. 20. The differing execution times for each module

Fig. 21. Taking the baseline cost into account

2) Analysis: The question we generated our results to answer was "How does the tool scale?".

Out results show that the test production procedure can be performed in a matter of several hours for most integration test procedures. For longer executions, the procedure may take several days.

This is infeasible to be run on a developers machine.

Should this be a procedure that occurred on demand on a CI (Continuous Integration) server, we feel this is not unreasonable. Because the process needs no user interaction, we believe that it scales well enough to be usable for uses cases in Section IV-B1 and Section IV-B2. As the use case for UI

[47]One might imagine the *changed* component in a software project is having its tests regenerated rather than all tests are being regenerated

Fig. 22. Proportional cost of Recorder and Generator modules.

automation requires manual UI interaction into traced software, this process would be laborious under a UI which has ~300x slow down. If a `paint` of the UI screen took 50ms in normal usage, it would take 15s whilst being traced. This is not necessarily infeasible, but it does limit the utility of this use case.

3) Abnormalities: There are a few abnormalities in the results. The first one is the sharp dip to a 1.1x slowdown effect in Figure 22. This occurs because of a *blocking* call in the original execution. This blocking call comes from the interaction with an external resource – in this case a subprocess.readLine() method. This can be seen in Section VIII-G, Listing 27 line 18. The second is the large variance in the results. This is due to the native execution time is not tied as closely to the number of control flow changes as in our traced application. An application which delegates a common sequence of instructions to a method will take more time than one which has those instructions inline. The JVM under debug mode cannot take advantage of optimizations like automatic inlining of methods (Section II-C), and as such the performance varies significantly.

E. Further Experiments

We performed several other experiments whose results are not tabulated and quantified in this report. Two of these are enumerated here.

1. **Generated Tests run faster than the original execution** Generated tests abstract away external complexities and resources. Because the tests generated are isolated in their interactions with each other, they can be run in parallel – no matter how conjoined the interactions between objects are. As such, the generated tests can *in-principle* be ran at the time cost of the maximum test – individually.
2. **Swing UI Interactions can be extracted and replicated through out test suite** One of the projects we ran our tool on was a Java Swing GUI Application. We managed to successfully generate tests for it. After committing them back to the project, the maintainer provided a testimonial of “Brilliant tests man. Looks pretty neat and easy to read as well.”.

VI. Future Work

Whilst working on this project, we have encountered several limitations with the technique we have proposed. We believe some of these limitations can be overcome through the use of the below techniques.

A. Test Trees

One of the biggest limitations that we found with our technique was that of **Brittleness** in tests produced. This term is defined in Section IV-A1, and relates to the fact that our generated tests will often give a false negative – i.e. they will fail for small changes which do not effect the behaviour of the application. This limits the utility of the technique in providing useful information to diagnose failures – what we desire is to provide a useful set of test cases to quickly replicate and illustrate the regression between two versions of code. By providing the test cases in the form of source code, they could be easily given to a developer to run directly.

The concept of test trees was one envisioned whilst trying to minimise this *brittleness* property. It is the concept of forming gradually *larger* test cases until the failure of which is meaningful.

In Section II-E we describe a combinatorially large set of possible unit tests which mock out dependencies and greater and greater *distances* from the original spawning execution, until they are equivalent to re-running the original execution. Test trees is an extrapolation of this concept, asking the question: can we generate and explore this set of different tests to identify false positives.

Our intuition tells us that this is a valid idea. We have no further evaluation beyond intuition though. The intuition is described in Figures~23-27. These show our test generation process building tests for an execution of a pipeline-style application.

Figure 25 shows an overview of our generated tests for an application. If the test processs completes successfully, we should generate a sequence of tests for each class within the control flow of our execution. Each test should have its dependencies mocked out Figure 26. One can imagine that rather than mocking out direct dependencies, these dependencies could be instantiated and *their* dependencies mocked out Figure 27. If we enumerate through all the options with mocks further and further away from the originating tests we would reduce brittleness – rather than *assuming* method interactions of a mock class, we would be ensuring that behaviour of more than one class is maintained when they are composed in synchrony. As this process will eventually reach a form where all dependencies are instantiated, the end result is always the original test/execution which was used to generate the tests. If we are looking for bugs, this execution will fail, but along the way there will be smaller and smaller tests – some of which "fail" and some of which "pass". By focusing attention on smaller tests which "fail", developer attention is focused on a smaller set of places to look for cause of error than in original execution.

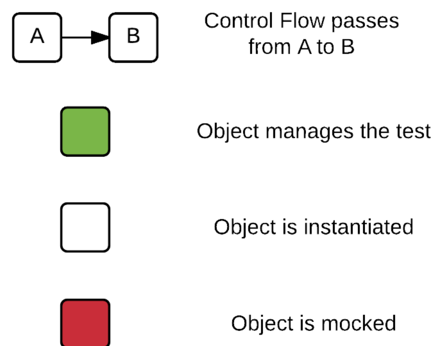

Fig. 23. The Legend for diagrams in @test-trees

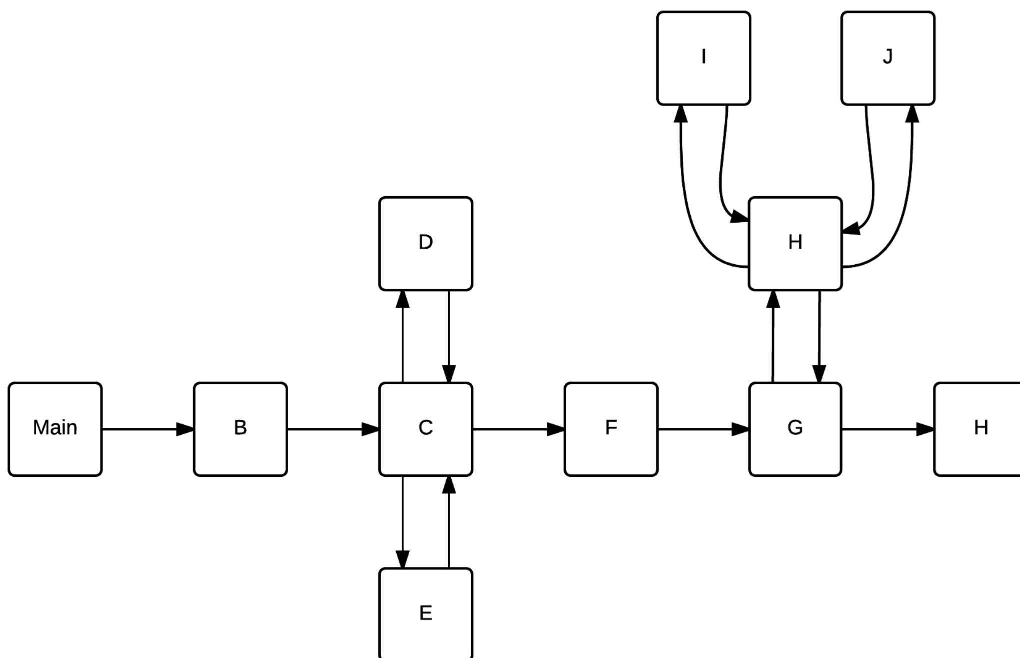

Fig. 24. A pipeline architected application

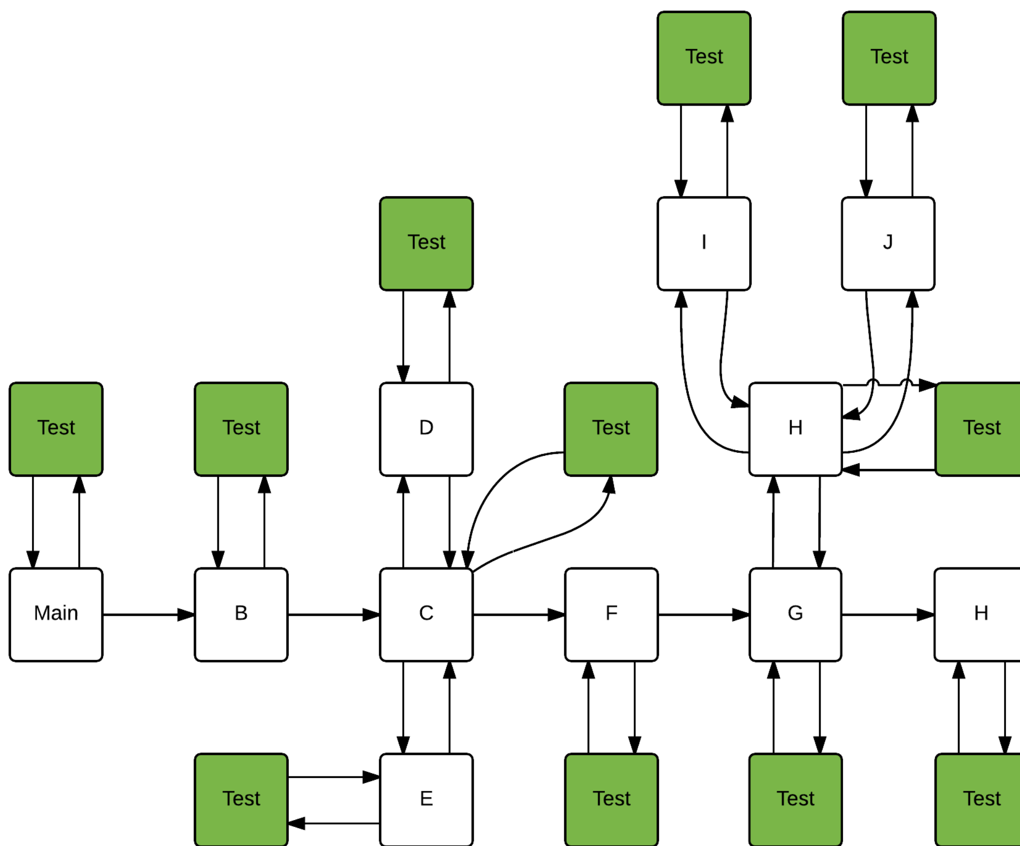

Fig. 25. The tests that will be generated by a full exercising

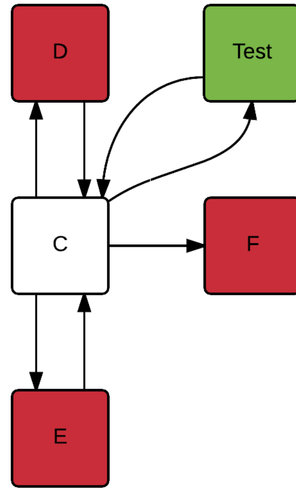

Fig. 26. Each test has its dependencies mocked out

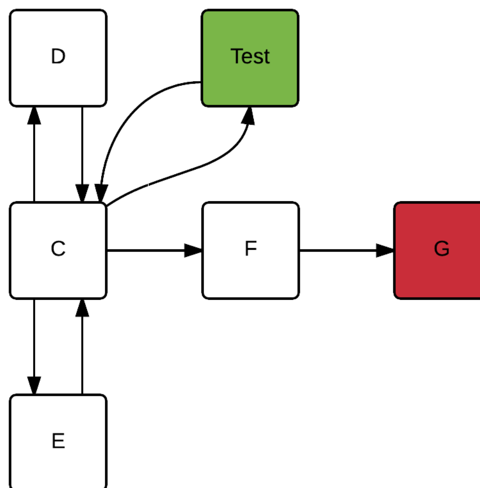

Fig. 27. Rather than mocking out dependencies, each dependency could be instantiated with its dependencies mocked out.

B. Generating tests for testing validity of mock objects.

In normal unit testing practices, mock objects are often used without any automated confidence that their interactions are still in-sync with the original program. To ensure that mocks are still valid would require using an instantiated object of the same type, and then interacting with it as the test does with the mock. As setting up such a harness would often require the use of more *mocks*, this process might be never ending.

Because our mechanism generates mock objects automatically and deterministically, it would be possible to generate tests for both sides of an interaction with low effort. By capturing state of external objects before and after their calls, we could generate additional unit tests *per mock* that verify that the mock approximation is a valid one. This could aid in the reduction of false positive results that are caused by incorrect mock specifications.

VII. CONCLUSION

We have produced a automatic unit test generator for Java. This tool requires no user input, and gathers test data from the execution of a known working version.

This tool generates unit tests that are *similar* to what developers produce manually, generating source code that uses two of the most popular testing and mocking frameworks for Java. We have shown that we can produce tests for the *majority* of an application, and we have shown that we can produce tests for large Java applications, though there is a high performance cost when doing so.

We have intentionally built our tool in such a way that it can be extended. Traces are generated and written to a database where they could be utilised be a multitude of services. Whilst we think that generating tests from them is one of the high value applications, we believe this there a wide variety of alternative uses where the execution graph could allow for useful metrics and visualisations of software, such as *producing sequence diagrams*, *complexity analysis*, *performance analysis* and *profile guided optimization*.

Fowler [18] suggests that when a bug is discovered by a high level test, such as a system or integration test, unit tests should be added that expose this bug. Doing this should help catch that bug faster should it ever be re-introduced. Our technique essentially performs that procedure, automatically. The tests it produces are *readable* and in exactly the same form that many Java developers are familiar with.

One of the main challenges of writing unit tests is ensuring that tests results are meaningful and rapidly lead a developer to the diagnosis of an error. Our technique can achieve the *minimum* test case for each method call, *mocking* all possible dependencies and isolating a bug to the smallest segment of code possible. Our evaluation shows that we can achieve this in a *reasonable* time frame, even over large code bases.

VIII. Appendix

A. Demonstration 1 :: Main

Here is the Main method used for our Data Access Object example, in Section I-A

```
1  public class DAOMain {
2      public static void main(String[] args) throws ClassNotFoundException, SQLException,
        DAOException {
3          Class.forName("org.sqlite.JDBC");
4          Connection conn =
5                  DriverManager.getConnection("jdbc:sqlite:test.db");
6          Statement stat = conn.createStatement();
7          stat.executeUpdate("drop table if exists people;");
8          stat.executeUpdate("create table people (name, occupation);");
9          PreparedStatement prep = conn.prepareStatement(
10                 "insert into people values (?, ?);");
11
12         prep.setString(1, "Gandhi");
13         prep.setString(2, "politics");
14         prep.addBatch();
15         prep.setString(1, "Turing");
16         prep.setString(2, "computers");
17         prep.addBatch();
18
19         conn.setAutoCommit(false);
20         prep.executeBatch();
21         conn.setAutoCommit(true);
22
23         DataAccessObject dao = new DataAccessObject(conn);
24
25         dao.getUserDetails("Gandhi");
26
27         conn.close();
28     }
29 }
```

B. The execution time of the `Recorder` module without tracing.

Figures 28, 29, and 30 show the execution time of the recorder module when it is not tracing – i.e. the baseline overhead that we cannot overcome without updates to the JDK to increase native debugging performance.

C. Expression Evaluation Example

Here is the short program used for our expression evaluation example. This was built for testing our tool, and consists of a Lexer to convert an expression string into a series of tokens, a Parser to convert this series of Tokens into an AST, and an Evaluator to evaluate this AST. It uses `BigDecimal` Operations to represent operations. The structure of this program is not directly relevant to our example, but it is provided here for the readers reference.

```
1  public class Main {
2      public static void main(String[] args) throws ParseException {
3          String expression = String.join(" ", args);
4          Lexer lexer = new Lexer(expression);
```

Fig. 28. The execution times when Recorder requests no events.

Fig. 29. Transforming Figure 28 by removing the baseline

```
        Parser parser = new Parser();
        Evaluator evaluator = new Evaluator();
        Iterator<Token> tokens = lexer.lex();
        ASTNode<Token> ast = parser.parse(tokens);
        BigDecimal result = evaluator.eval(ast);
        System.out.println(result);
    }
}

public interface Token {
    int precedence();

    boolean isOperator();


    class NUMBER implements Token {
        private final BigDecimal value;

        public NUMBER(BigDecimal value) {
            this.value = value;
        }

```

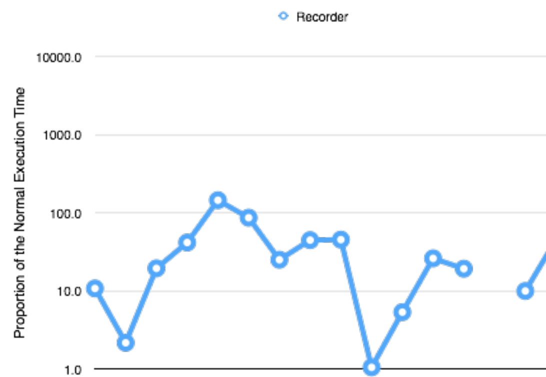

Fig. 30. Proportional Cost of the Recorder module not capturing a trace.

```
27          public BigDecimal getValue() {
28              return value;
29          }
30
31          @Override
32          public int precedence() {
33              return 1;
34          }
35
36          @Override
37          public boolean isOperator() {
38              return false;
39          }
40      }
41
42      class PLUS implements Token {
43          @Override
44          public int precedence() {
45              return 2;
46          }
47
48          @Override
49          public boolean isOperator() {
50              return true;
51          }
52      }
53
54      class MINUS implements Token {
55          @Override
56          public int precedence() {
57              return 2;
58          }
59
60          @Override
61          public boolean isOperator() {
62              return true;
63          }
64      }
65
66      class MULTIPLY implements Token {
```

```
67          @Override
68          public int precedence() {
69              return 3;
70          }
71
72          @Override
73          public boolean isOperator() {
74              return true;
75          }
76      }
77
78      class DIVIDE implements Token {
79          @Override
80          public int precedence() {
81              return 4;
82          }
83
84          @Override
85          public boolean isOperator() {
86              return true;
87          }
88      }
89
90
91      class LBRACKET implements Token {
92          @Override
93          public int precedence() {
94              return 0;
95          }
96
97          @Override
98          public boolean isOperator() {
99              return false;
100         }
101     }
102
103     class RBRACKET implements Token {
104         @Override
105         public int precedence() {
106             return 0;
107         }
108
109         @Override
110         public boolean isOperator() {
111             return false;
112         }
113     }
114 }
115
116
117 public class Lexer {
118     private final String expression;
119     private int i;
120
121     public Lexer(String expression) {
122         this.expression = expression;
123         i = 0;
124     }
125
```

```
126     public Iterator<Token> lex() throws ParseException {
127         LinkedList<Token> tokens = new LinkedList<>();
128 
129         while (i < expression.length()) {
130             char c = expression.charAt(i);
131             switch (c) {
132                 case ' ':
133                     i += 1;
134                     break;
135                 case '(':
136                     tokens.push(new Token.LBRACKET());
137                     i += 1;
138                     break;
139                 case ')':
140                     tokens.push(new Token.RBRACKET());
141                     i += 1;
142                     break;
143                 case '+':
144                     tokens.push(new Token.PLUS());
145                     i += 1;
146                     break;
147                 case '-':
148                     tokens.push(new Token.MINUS());
149                     i += 1;
150                     break;
151                 case '*':
152                     tokens.push(new Token.MULTIPLY());
153                     i += 1;
154                     break;
155 
156                 case '/':
157                     tokens.push(new Token.DIVIDE());
158                     i += 1;
159                     break;
160 
161                 default:
162                     if (c >= '0' && c <= '9' || c == '.') {
163                         BigDecimal number = eatNumber();
164                         tokens.push(new Token.NUMBER(number));
165                     } else {
166                         throw new ParseException(expression, i);
167                     }
168             }
169         }
170         return tokens.descendingIterator();
171     }
172 
173     private BigDecimal eatNumber() {
174         int numberStartIndex = i;
175 
176         while (i < expression.length()
177                 && (expression.charAt(i) == '.'
178                 || (expression.charAt(i) >= '0' && expression.charAt(i) <= '9')
179         )
180                 )
181             i += 1;
182         return new BigDecimal(expression.substring(numberStartIndex, i));
183     }
184 }
```

```
185
186 public class Parser {
187     Stack<ASTNode<Token>> operands = new Stack<>();
188     Stack<Token> operators = new Stack<>();
189
190     public ASTNode<Token> parse(Iterator<Token> tokens) {
191         tokenizing:
192         while (tokens.hasNext()) {
193             Token tok = tokens.next();
194             if (tok instanceof Token.LBRACKET) {
195                 operators.push(tok);
196             } else if (tok instanceof Token.RBRACKET) {
197                 while (!operators.isEmpty()) {
198                     Token popped = operators.pop();
199                     if (popped instanceof Token.LBRACKET) {
200                         continue tokenizing;
201                     } else {
202                         addNode(operands, popped);
203                     }
204                 }
205                 throw new IllegalStateException("Unbalanced right " +
206                         "parentheses");
207             } else {
208                 if (tok.isOperator()) {
209                     Token o1, o2;
210                     o1 = tok;
211                     while (!operators.isEmpty()) {
212                         o2 = o1;
213                         o1 = operators.peek();
214                         if (o1.precedence() >= o2.precedence()) {
215                             addNode(operands, operators.pop());
216                         } else {
217                             break;
218                         }
219                     }
220                     operators.push(tok);
221                 } else {
222                     operands.push(new ASTNode<>(null, tok, null));
223                 }
224             }
225         }
226         while (!operators.isEmpty()) {
227             addNode(operands, operators.pop());
228         }
229
230         return operands.pop();
231     }
232
233     private void addNode(Stack<ASTNode<Token>> stack, Token popped) {
234         ASTNode<Token> right = stack.pop();
235         ASTNode<Token> left = stack.pop();
236         stack.push(new ASTNode<>(left, popped, right));
237
238     }
239 }
240
241 public class Evaluator {
242
243     public BigDecimal eval(ASTNode<Token> ast) {
```

```
244         Token top = ast.getTop();
245         if (top instanceof Token.NUMBER) {
246             return ((Token.NUMBER) top).getValue();
247         } else if (top instanceof Token.PLUS) {
248             return eval(ast.getLeft()).add(eval(ast.getRight()));
249         } else if (top instanceof Token.MINUS) {
250             return eval(ast.getLeft()).subtract(eval(ast.getRight()));
251         } else if (top instanceof Token.MULTIPLY) {
252             return eval(ast.getLeft()).multiply(eval(ast.getRight()));
253         } else if (top instanceof Token.DIVIDE) {
254             return eval(ast.getLeft()).divide(eval(ast.getRight()), BigDecimal.ROUND_UP
    );
255         } else {
256             throw new NotImplementedException();
257         }
258     }
259 }
```

There are too many generated tests to reasonably demonstrate them all here. We have managed to successfully generate tests for complex expressions of many terms. One of these can be seen in Listing 21 – showing that for `a - b` the eval function computes `a.value subtract b.value` with the `BigDecimal` class.

```
1     @Test
2     public void testEval14() {
3 
4         // Generate the arguments
5 
6         ASTNode mockast = Mockito.mock(ASTNode.class);
7         test.test8.Token.MINUS mocktop = Mockito.mock(test.test8.Token.MINUS.class);
8         ASTNode mockast1 = Mockito.mock(ASTNode.class);
9         test.test8.Token.NUMBER mocktop1 = Mockito.mock(test.test8.Token.NUMBER.class);
10        BigDecimal mockvalue = Mockito.mock(BigDecimal.class);
11        BigDecimal mockbig_0 = Mockito.mock(BigDecimal.class);
12        ASTNode mockast2 = Mockito.mock(ASTNode.class);
13        test.test8.Token.NUMBER mocktop2 = Mockito.mock(test.test8.Token.NUMBER.class);
14        BigDecimal mockbig_01 = Mockito.mock(BigDecimal.class);
15
16        // Construct the Evaluator class
17
18        Evaluator evaluator = new Evaluator();
19
20        // Link Mocks
21
22        Mockito.when(mocktop1 .getValue()).thenReturn(mockvalue);
23        Mockito.when(mockvalue.subtract(Mockito.eq(mockbig_01))).thenReturn(mockbig_0);
24        Mockito.when(mockast.getRight()).thenReturn(mockast2);
25        Mockito.when(mocktop2 .getValue()).thenReturn(mockbig_01);
26        Mockito.when(mockast.getLeft()).thenReturn(mockast1);
27        Mockito.when(mockast.getTop()).thenReturn(mocktop);
28        Mockito.when(mockast2 .getTop()).thenReturn(mocktop2);
29        Mockito.when(mockast1 .getTop()).thenReturn(mocktop1);
30
31        // Invoke the method
32
33        BigDecimal bigDecimal = evaluator.eval(mockast);
34
35        // Assert that our method interactions are what we expect.
```

```
36
37          Mockito.verify(mocktop2, Mockito.times(1)).getValue();
38          Mockito.verify(mocktop1, Mockito.times(1)).getValue();
39          Mockito.verify(mockast2, Mockito.times(1)).getTop();
40          Mockito.verify(mockast1, Mockito.times(1)).getTop();
41          Mockito.verify(mockast, Mockito.times(1)).getRight();
42          Mockito.verify(mockast, Mockito.times(1)).getTop();
43          Mockito.verify(mockvalue, Mockito.times(1)).subtract(Mockito.eq(mockbig_01));
44          Mockito.verify(mockast, Mockito.times(1)).getLeft();
45      }
```

Listing 21. One of the many tests of the Evaluator method

D. Generating our own test suite

One of the interesting applications we used our tool for was to generate its own test suite. To do this we performed the following steps:

1. Run our `recorder` given a target main class of itself, and target arguments to trace the ABC example. Each `recorder` instance was setup to output its results to a different database.

```
java -cp recorder.jar:examples.jar -db /tmp/db1 -output Neo4j
      -target recorder.Main -c recorder
      -args "-db /tmp/db2 -target test.test1.Main
             -args 10 -c test.test1"
```

2. Run our `generator` module on the the traced results.

```
java -cp recorder.jar:generator.jar generator.Main -db /tmp/db1
     -dump -output FILE -all
```

In initial versions of our tool, these tests took ~2-3 days to produce. After a smarter filtering implementation (Section III-C), this was reduced to several hours. After our `sampling` strategy implementation (Section III-C3), this was further reduced to ~40 minutes.

The generated tests provided 72% line coverage for our `recorder` module. The test generator faired badly because many of the JPDA APIs that we use had static accessor methods, and we could not intercept those calls and replace them with mocked objects. Our utility methods did however generate well. We were provided a sequence of example input and output for each of them. For instance, Listing 22.

In practice however, we did not use these tests. This is because the project was developed by a solo developer, and was exploratory. This means our implementation changed significantly enough over time in non-breaking ways, and any breaking unit tests were noise. We hope that the Section VI-A concept would remove this noise in a meaningful manner.

If we were to provide this project to another developer, we believe that the generated unit tests have significantly more utility. Unit tests form a part of a projects documentation – they state precisely what one might expect for each given input and output of a method call. When dealing with an unfamiliar code base, they are one of the first areas developers look at to understand the *purpose* of each method or class.

If they don't exist, often developers step through the code in a debugger to provide insight. One can imagine this "stepping though the debugger" process is equivalent to looking through our generated tests, because they are generated through automatically "stepping through the debugger".

```
@Test
public void testIsInitMethod() {

    // Generate the arguments

    ConcreteMethodImpl mockmethod = Mockito.mock(ConcreteMethodImpl.class);
    Mockito.when(mockmethod.name()).thenReturn("<init>");

    // Invoke the method

    boolean b = FilterRules.isInitMethod(mockmethod);

    // Assert the return value

    Assert.assertEquals(true, b);

    // Assert that our method interactions are what we expect.

    Mockito.verify(mockmethod, Mockito.times(1)).name();
}
```

Listing 22. One of our generated tests for utility methods during *inception* test generation

E. Coverage Data Tables

TABLE I: The results for collected coverage information of original and generated executions.

Project	Entrypoint	Number of Classes	Method Coverage	Block Coverage	Branch Coverage	Line Coverage
falselight	falselight	2/7	4/30	4/99	1/16	16/182
falselight	Our Tool's Generated JUnit Tests	2/7	4/30	4/99	1/16	16/182
jclo	edu.mscd.cs.jclo.JCLO	4/4	22/53	97/193	60/116	135/242
jclo	Our Tool's Generated JUnit Tests	4/4	22/53	97/193	60/116	135/242
jopenchart	de.progra.charting.ChartUtilities	1/40	7/361	46/983	30/531	N/A
jopenchart	Our Tool's Generated JUnit Tests	1/40	7/361	46/983	30/531	N/A
sfm	com.hf.sfm.crypto.Base64	1/15	7/127	29/291	15/130	108/578
sfm	Our Tool's Generated JUnit Tests	0/15	0/127	0/291	0/130	0/578
caloriecount	com.lts.util.SortTest	5/925	21/7129	68/15575	40/7025	N/A
caloriecount	Our Tool's Generated JUnit Tests	5/925	21/7129	68/15575	40/7025	N/A
openhre	com.[...].TestExpresion	9/61	46/567	151/1608	89/1028	216/2439
openhre	Our Tool's Generated JUnit Tests	8/61	34/567	44/1608	9/1028	63/2439
fizzbuzz	com.[...].impl.Main	52/59	70/76	N/A	N/A	241/272
fizzbuzz	Our Tool's Generated JUnit Tests	44/59	58/126	N/A	N/A	200/301

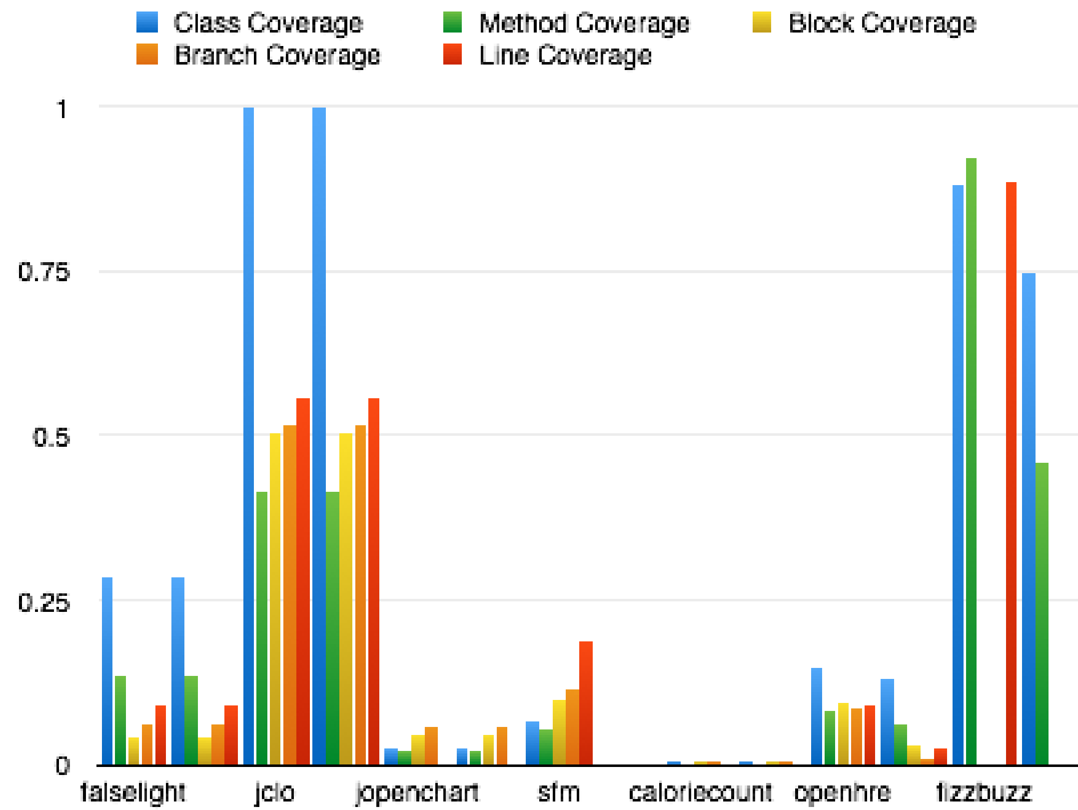

Fig. 31. The differing coverage results proportionally for all metrics

F. Performance Data Tables

Table II shows the collected data tables for our performance analysis. The discussion can be found in Section V-D

TABLE II: The collected performance information for each stage of the test generation pipeline.

Project Name	Normal Execution	Recorder Execution	Generator Execution	Normalised Normal	Normalised Generator	Normalised Recorder	Proportional Recorder	Proportional Generator
falselight	116.0	2724.6	11771.5	23.0	552.6	503.5	24.0	21.9
jclo	193.9	46061.8	12231.0	100.9	43889.8	963.0	435.0	9.5
jopenchart	109.9	9769.6	11909.8	16.9	7597.6	641.8	449.6	38.0
sfm	103.4	8217.3	11847.0	10.4	6045.3	579.0	581.3	55.7
caloriecount	95.2	3252.3	11735.2	2.2	1080.3	467.2	491.0	212.4
openhre	96.9	7385.8	11868.0	3.9	5213.8	600.0	1336.9	153.8
hft-bomberman	120.8	17762.1	11819.4	27.8	15590.1	551.4	560.8	19.8
imsmart	107.5	4345.2	11767.6	14.5	2173.2	499.6	149.9	34.5
test1.Main	105.6	2664.0	11767.5	12.6	492.0	499.5	39.0	39.6
test2.Main	3151.6	6954.3	11803.1	3058.6	4782.3	535.1	1.6	0.2
test3.DAOMain	234.0	2950.1	11785.1	141.0	778.1	517.1	5.5	3.7
test4.Main	108.5	7092.9	11733.7	15.5	4920.9	465.7	317.5	30.0
test5.Main	109.3	5059.7	11822.0	16.3	2887.7	554.0	177.2	34.0
test6.Main	96.0	2539.0	11689.3	3.0	367.0	421.3	0.0	0.0

Project Name	Normal Execution	Recorder Execution	Generator Execution	Normalised Normal	Normalised Generator	Normalised Recorder	Proportional Recorder	Proportional Generator
test7.Main	180.7	2937.0	11706.0	87.7	765.0	438.0	8.7	5.0
test8.Main	104.5	2248.0	11302.9	11.5	76.0	34.9	6.6	3.0

G. Process Interaction

This program and resultant generated tests is used for demonstration purposes.

```
public class ConnectivityDetector {
    private final PingFactory pingFactory;

    public ConnectivityDetector(PingFactory pingFactory) {
        this.pingFactory = pingFactory;
    }

    public boolean isConnected() throws IOException {
        Pinger p = pingFactory.makePinger();
        ProcessReader processReader = pingFactory.makeProcessReader();
        PingResults result = p.ping("8.8.8.8", processReader)
        return result.getPacketLoss() < 1;
    }
}
```

Listing 23. A class that aims to detect if the computer is connected to the internet through the *Pinger* class.

```
public class TestConnectivityDetector {
    @Test
    public void testConstructor()
        throws IllegalAccessException, NoSuchFieldException
    {

        // Generate the arguments

        PingFactory mockpingFactory = Mockito.mock(PingFactory.class);

        // Invoke the method

        ConnectivityDetector connectivityDetector = new ConnectivityDetector(
    mockpingFactory);

        // Assert the fields are what we expect

        Field field = connectivityDetector.getClass().getDeclaredField("pingFactory");
        field.setAccessible(true);
        PingFactory pingFactory = ((PingFactory) field.get(connectivityDetector));
        Assert.assertEquals(mockpingFactory, pingFactory);

        // Assert that our method interactions are what we expect.

    }

    @Test
    public void testIsConnected()
        throws IOException
    {

        // Generate the fields

        PingFactory mockpingFactory = Mockito.mock(PingFactory.class);
        Pinger mockPinger = Mockito.mock(Pinger.class);
        PingResults mockPingResults = Mockito.mock(PingResults.class);
        ProcessReader mockp = Mockito.mock(ProcessReader.class);

        // Construct the ConnectivityDetector class
```

```
39
40          ConnectivityDetector connectivityDetector = new ConnectivityDetector(
        mockpingFactory);
41
42          // Link Mocks
43
44          Mockito.when(mockpingFactory.makePinger()).thenReturn(mockPinger);
45          Mockito.when(mockPingResults.getPacketLoss()).thenReturn(0.0F);
46          Mockito.when(mockpingFactory.makeProcessReader()).thenReturn(mockp);
47          Mockito.when(mockPinger.ping(Mockito.eq("8.8.8.8"), Mockito.eq(mockp))).
        thenReturn(mockPingResults);
48
49          // Invoke the method
50
51          boolean b = connectivityDetector.isConnected();
52
53          // Assert the return value
54
55          Assert.assertEquals(true, b);
56
57          // Assert that our method interactions are what we expect.
58
59          Mockito.verify(mockPinger, Mockito.times(1)).ping(Mockito.eq("8.8.8.8"),
        Mockito.eq(mockp));
60          Mockito.verify(mockpingFactory, Mockito.times(1)).makeProcessReader();
61          Mockito.verify(mockPingResults, Mockito.times(1)).getPacketLoss();
62          Mockito.verify(mockpingFactory, Mockito.times(1)).makePinger();
63      }
64  }
```

Listing 24. The generated test for Listing 23

```
1   public class PingFactory {
2
3       public Pinger makePinger() {
4           return new Pinger(new PingCommand());
5       }
6
7       public ProcessReader makeProcessReader() {
8           return new ProcessReader();
9       }
10  }
```

Listing 25. A factory class to instantiate the Pinger and ProcessReader classes.

```
1   public class TestPingFactory {
2       @Test
3       public void testConstructor() {
4
5           // Invoke the method
6
7           PingFactory pingFactory = new PingFactory();
8       }
9
10      @Test
11      public void testMakePinger() {
12
13          // Construct the PingFactory class
14
15          PingFactory pingFactory = new PingFactory();
```

```
16
17          // Invoke the method
18
19          Pinger pinger = pingFactory.makePinger();
20      }
21
22      @Test
23      public void testMakeProcessReader() {
24
25          // Construct the PingFactory class
26
27          PingFactory pingFactory = new PingFactory();
28
29          // Invoke the method
30
31          ProcessReader processReader = pingFactory.makeProcessReader();
32      }
33
34  }
```

Listing 26. The generated test for Listing 25.

```
1   public class Pinger {
2       private final PingCommand pingCommand;
3
4       public Pinger(PingCommand pingCommand) {
5           this.pingCommand = pingCommand;
6       }
7
8       public PingResults ping(String s, ProcessReader p) throws IOException {
9           Process process = pingCommand.run();
10
11          BufferedReader stdInput = p.getBufferedReader(process.getInputStream());
12          BufferedReader stdError = p.getBufferedReader(process.getErrorStream());
13
14          Collection<Boolean> results = new LinkedList<Boolean>();
15          String final_line_1 = null;
16          String final_line_2 = null;
17
18          while ((s = stdInput.readLine()) != null) {
19              final_line_1 = final_line_2;
20              final_line_2 = s;
21          }
22
23          // read any errors from the attempted command
24          if (final_line_1 == null) {
25              return new PingResults(0, 0, 1f);
26          } else {
27              Pattern pattern = Pattern.compile("([0-9]+) packets transmitted, ([0-9]+)
        packets received, ([0-9]+.?[0-9]*)% packet loss");
28              Matcher matcher = pattern.matcher(final_line_1);
29              if (!matcher.find() || matcher.groupCount() != 3) {
30                  throw new RuntimeException("Unusual ping output");
31              }
32              int transmitted = new Integer(matcher.group(1));
33              int received = new Integer(matcher.group(2));
34              float loss = Float.parseFloat(matcher.group(3));
35              return new PingResults(transmitted, received, loss);
36          }
37      }
```

```
38 }
```

Listing 27. A class to run and control a *ping* subprocess and collect the results

```
public class TestPinger {
    @Test
    public void testConstructor()
        throws IllegalAccessException, NoSuchFieldException
    {

        // Generate the arguments

        PingCommand mockpingCommand = Mockito.mock(PingCommand.class);

        // Invoke the method

        Pinger pinger = new Pinger(mockpingCommand);

        // Assert the fields are what we expect

        Field field = pinger.getClass().getDeclaredField("pingCommand");
        field.setAccessible(true);
        PingCommand pingCommand = ((PingCommand) field.get(pinger));
        Assert.assertEquals(mockpingCommand, pingCommand);

        // Assert that our method interactions are what we expect.

    }

    @Test
    public void testPing()
        throws IOException
    {

        // Generate the arguments

        ProcessReader mockp = Mockito.mock(ProcessReader.class);
        BufferedReader mockBufferedReader = Mockito.mock(BufferedReader.class);
        BufferedReader mockBufferedReader1 = Mockito.mock(BufferedReader.class);

        // Generate the fields

        PingCommand mockpingCommand = Mockito.mock(PingCommand.class);
        Process mockUNIXProcess = Mockito.mock(Process.class);
        BufferedInputStream mockuni_0 = Mockito.mock(BufferedInputStream.class);
        BufferedInputStream mockuni_01 = Mockito.mock(BufferedInputStream.class);

        // Construct the Pinger class

        Pinger pinger = new Pinger(mockpingCommand);

        // Link Mocks

        Mockito.when(mockp.getBufferedReader(Mockito.eq(mockuni_0))).thenReturn(
    mockBufferedReader);
        Mockito.when(mockpingCommand.run()).thenReturn(mockUNIXProcess);
        Mockito.when(mockUNIXProcess.getErrorStream()).thenReturn(mockuni_01);
        Mockito.when(mockBufferedReader.readLine()).thenReturn("PING 8.8.8.8 (8.8.8.8):
     56 data bytes").thenReturn("64 bytes from 8.8.8.8: icmp_seq=0 ttl=52 time=4.560
    ms").thenReturn("64 bytes from 8.8.8.8: icmp_seq=1 ttl=52 time=5.209 ms").
```

```
    thenReturn("64 bytes from 8.8.8.8: icmp_seq=2 ttl=52 time=4.555 ms").thenReturn("
    64 bytes from 8.8.8.8: icmp_seq=3 ttl=52 time=4.514 ms").thenReturn("").thenReturn
    ("--- 8.8.8.8 ping statistics ---").thenReturn("4 packets transmitted, 4 packets
    received, 0.0% packet loss").thenReturn("round-trip min/avg/max/stddev =
    4.514/4.709/5.209/0.289 ms").thenReturn(null);
54      Mockito.when(mockp.getBufferedReader(Mockito.eq(mockuni_01))).thenReturn(
    mockBufferedReader1);
55      Mockito.when(mockUNIXProcess.getInputStream()).thenReturn(mockuni_0);
56
57      // Invoke the method
58
59      PingResults pingResults = pinger.ping("8.8.8.8", mockp);
60
61      // Assert that our method interactions are what we expect.
62
63      Mockito.verify(mockp, Mockito.times(1)).getBufferedReader(Mockito.eq(mockuni_0)
    );
64      Mockito.verify(mockBufferedReader, Mockito.times(10)).readLine();
65      Mockito.verify(mockUNIXProcess, Mockito.times(1)).getInputStream();
66      Mockito.verify(mockp, Mockito.times(1)).getBufferedReader(Mockito.eq(mockuni_01
    ));
67      Mockito.verify(mockpingCommand, Mockito.times(1)).run();
68      Mockito.verify(mockUNIXProcess, Mockito.times(1)).getErrorStream();
69  }
70
71 }
```

Listing 28. The generated test for Listing 27

References

[1] B. Boehm, “Some Future Trends and Implications for Systems and Software Engineering Processes,” *Syst. Eng.*, vol. 9, no. 1, pp. 1–19, Jan. 2006.

[2] D. S. Alberts, “The Economics of Software Quality Assurance,” in *Proceedings of the June 7-10, 1976, National Computer Conference and Exposition*, ser. AFIPS ’76. New York, NY, USA: ACM, 1976, pp. 433–442.

[3] N. Tillmann, J. de Halleux, and T. Xie, “Transferring an Automated Test Generation Tool to Practice: From Pex to Fakes and Code Digger,” in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE ’14. New York, NY, USA: ACM, 2014, pp. 385–396.

[4] M. Lehman, “Programs, life cycles, and laws of software evolution,” *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, Sep. 1980.

[5] M. M. Lehman, “Laws of Software Evolution Revisited,” in *Proceedings of the 5th European Workshop on Software Process Technology*, ser. EWSPT ’96. London, UK, UK: Springer-Verlag, 1996, pp. 108–124.

[6] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, “Metrics and Laws of Software Evolution - The Nineties View,” in *Proceedings of the 4th International Symposium on Software Metrics*, ser. METRICS ’97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 20–.

[7] B. W. Boehm and P. N. Papaccio, “Understanding and Controlling Software Costs,” *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1462–1477, Oct. 1988.

[8] R. Osherove, *The Art of Unit Testing: With Examples in .Net*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2009.

[9] T. Bhat and N. Nagappan, "Evaluating the efficacy of test-driven development: Industrial case studies," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ser. ISESE '06. New York, NY, USA: ACM, 2006, pp. 356–363.

[10] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM, 2011, pp. 26–36.

[11] S. Freeman and N. Pryce, *Growing Object-Oriented Software, Guided by Tests*, 1st ed. Addison-Wesley Professional, 2009.

[12] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *2012 34th International Conference on Software Engineering (ICSE)*, Jun. 2012, pp. 178–188.

[13] J. Lewis and W. Loftus, *Java Software Solutions: Foundations of Program Design*, 6th ed. USA: Addison-Wesley Publishing Company, 2008.

[14] I. H. Kazi, H. H. Chen, B. Stanley, and D. J. Lilja, "Techniques for Obtaining High Performance in Java Programs," *ACM Comput. Surv.*, vol. 32, no. 3, pp. 213–240, Sep. 2000.

[15] A. Bacchelli, P. Ciancarini, and D. Rossi, "On the effectiveness of manual and automatic unit test generation," in *The Third International Conference on Software Engineering Advances, 2008. ICSEA '08*, Oct. 2008, pp. 252–257.

[16] A. Mandal, "BRIDGE: A Model for Modern Software Development Process to Cater the Present Software Crisis," in *Advance Computing Conference, 2009. IACC 2009. IEEE International*, Mar. 2009, pp. 1617–1623.

[17] W. Perry, *Effective Methods for Software Testing, Third Edition*. New York, NY, USA: John Wiley & Sons, Inc., 2006.

[18] A. Hunt and D. Thomas, *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[19] R. D. Craig and S. P. Jaskiel, *Systematic Software Testing*. Norwood, MA, USA: Artech House, Inc., 2002.

[20] T. Mackinnon, S. Freeman, and P. Craig, "Extreme programming examined," G. Succi and M. Marchesi, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 287–301.

[21] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended static checking for java," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New York, NY, USA: ACM, 2002, pp. 234–245.

[22] P. Bourque and R. Dupuis, "Guide to the Software Engineering Body of Knowledge 2004 Version," *Guide to the Software Engineering Body of Knowledge, 2004. SWEBOK*, pp. –, 2004.

[23] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[24] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, "GPUVerify: A Verifier for GPU Kernels," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp.

113–132.

[25] A. A. Hwang, I. A. Stefanovici, and B. Schroeder, "Cosmic Rays Don'T Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 111–122.

[26] "Enhanced Execution Record / Replay in Workstation 6.5 | VMware Workstation Zealot - VMware Blogs."

[27] mozilla, "github.com/mozilla/rr : Record and replay framework."

[28] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic Test Factoring for Java," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 114–123.

[29] G. Pothier and E. Tanter, "Back to the future: Omniscient debugging," *IEEE Software*, vol. 26, no. 6, pp. 78–85, Nov. 2009.