Recording Execution Traces through Java Bytecode Programs

Gwenyth Rooijackers

MInf Project (Part 1) Report

Master of Informatics
School of Informatics
University of Edinburgh

2021

Abstract

Dynamic analysis is a valuable tool to understand the behaviour of a program when executed. The focus of the project was execution tracing, a common method used for dynamic analysis. The implemented tracing tool presented in this report instruments a compiled Java bytecode program by inserting logging statements using the Soot framework. When running the instrumented program, execution traces are produced recording the method names, arguments and return values of method invocations.

The feasibility of the tracing tool was shown by tracing the tests of four real Java libraries, Joda-Time, JFreeChart, Commons-Lang and Commons-Math. The overhead introduced by the instrumentation was large and depended on the size of the produced traces. The usefulness of the traces was evaluated by tracing tests and using these traces in an existing classifier tackling the test oracle problem. The performance of the classifier was poor for all but one test program, but the usefulness experiment was successful in showing that the traces could be applied in an existing setting. A small usability study showed that the volunteers were happy with the features but the tool could be easier to use.

Acknowledgements

First a big thank you to my supervisor Dr. Ajitha Rajan for the ongoing support, frequent meetings and helpful feedback on the report.

Thank you, Chao Peng for the help to get me started with Soot and Foivos Tsimpourlas for his expertise on the test oracle.

I also want to thank the volunteers in the usability study.

Table of Contents

Chapter 1

Introduction

Static analysis evaluates the code of a program to derive its inherent properties that hold over all executions. Dynamic analysis, on the other hand, examines a running program and provides insights into the behaviour of the program. Dynamic analysis cannot provide the properties of a program like static analysis can, but by examining one or a few executions of a program, it can find instances when the desired properties of the program do not hold. [1]

An advantage of dynamic analysis is precision of information: it allows us to tune exactly what information we want to record during execution. [1] Precision of information will be an important concept for this project, as it will allow us to design our execution traces.

To record execution information for dynamic analysis, small bits of code can be inserted into the program's executable files, a process known as instrumentation. [2]

The goal of this project was to instrument Java bytecode to produce execution traces for dynamic analysis. Java bytecode was chosen for this project in particular to extend the work of Tsimpourlas et al. [3] They instrumented the LLVM intermediate representation, which allowed them to trace programs such as C/C++, gather traces of tests through buggy programs and attempt to automatically classify the traces as coming from passing or failing tests.

The advantages of instrumenting the compiled bytecode over the source code are that (1) a program can be traced even when you only have access to the compiled program and not the source code and (2) the framework also works for programs in languages other than Java that compile to Java bytecode, such as Groovy, Kotlin and Scala. Android apps, which are developed in Java and so compile to Java bytecode, are also possible target programs for the tool.

1.1 Overview of Report

Tracing tool. The main part of the project presented in this report was building a tracing tool to record execution traces of Java programs. The traces record the exit

points of method calls and include the names of the calling and called methods, any arguments and any return values. The tracing tool is deployed on a compiled program. Details of the approach are described in chapter 3 (Capturing the Execution Traces).

The result of running the tool is the instrumented program. When running the instrumented program with a given input, it has the same functionality as originally, but the method invocations will be recorded in an execution trace that is printed to the standard output.

Evaluation. The evaluation of the tracing tool is described in chapters 4 and 5 (Experiments and Results). Four aspects of the tool were considered for the experiments: feasibility, overhead, usefulness and usability.

The first aspect of the experiments was **feasibility** of applying the tool under real settings, on four Java libraries: Joda-Time, JFreeChart, Commons-Math and Commons-Lang from the Defects4J database [4]. To achieve this, buggy versions of the programs were instrumented to gather the traces produced by running their tests.

The runtime of the instrumentation and the **overhead** introduced into the programs from the instrumentation were then evaluated on the same programs.

Failing and passing traces were created by dividing the traces of each program into two sets, those produced by failing tests and those produced by passing tests. The **usefulness** experiment aimed to assess whether an existing classifier could accept the produced traces as its input. Two types of usefulness experiments were included. The first type mirrored the experiment performed in the paper by Tsimpourlas et al. [3], but using Java programs rather than C/C++ programs. This involved training their neural network on a subset of the traces of the tests of a program and evaluating how well the classifier could classify the remaining test traces of the same program.

The second usefulness experiment was not performed in the study by Tsimpourlas et al. [3] and it involved training a classifier on the traces from a program and evaluating its classification performance on the traces of a different version of the same program. This aimed to test the ability of the classifier to generalise to a different program version with a different bug than what it was trained on. This experiment was possible because the Defects4J database provides multiple program versions with different bugs.

The final evaluation aspect of this project was centered around **usability**. The usability testing involved unfamiliar users testing the tracing tool presented in this report and answering questions about the experience.

In summary, the contributions presented in the report are:

- Making a tracing tool for compiled Java programs which instruments the programs to record method invocations.
- Evaluating the feasibility of the tool by gathering the traces of four existing programs.
- Evaluating the overhead incurred by the tracing instrumentation.

- Evaluating the usefulness of the traces by performing experiments that involved running the gathered traces through an existing classifier that attempts to tackle the test oracle problem.
- Performing a small usability study of the tool.

1.2 Impact

The results of the first usefulness experiment on Commons-Lang were included in a manuscript which has been submitted to a journal and is currently under review. [5] More specifically, this involved the tracing tool used for tracing the tests of Commons-Lang and the performance of the neural network for classification on the traces of Commons-Lang.

Chapter 2

Background

2.1 Execution Tracing

Execution traces are generated by logging events in a program while it is running. Strictly speaking, execution tracing is different from logging. Execution tracing collects information for software developers or maintainers while logging refers to collecting information for system administrators or operators. [6]

Execution traces are used for understanding and debugging programs. It is especially important in Object-Oriented systems as the complexity can prove difficult for static analysis. [7, 8] For example, dynamic binding is a challenge for static analysis as it is resolved at runtime rather than at compile time (static binding). When dynamically bound, the exact method that is invoked depends on the type of the object it was called on, meaning that static analysis cannot capture the behaviour. As execution tracing is a kind of dynamic analysis, it can detect the behaviour of the program given a test input, but may not capture all the possible behaviours, those not covered by the test cases. [8]

Examples of uses of execution traces are automatically separating passing and failing tests (test oracle) [3, 9], automated fault localization [10], recovering collaborations from existing code [11] and finding and understanding the features of an existing program [12].

2.2 Java Bytecode Instrumentation

Not only Java but also for example Scala and JRuby are compiled to Java bytecode [13] and can then be interpreted by the Java Virtual Machine (JVM). Bytecode instrumentation involves making changes to the bytecode that can be used to change functionality or monitor behaviour of a programs. Bytecode instrumentation for monitoring behaviour is important in dynamic program analysis on compiled programs. [14] Examples of dynamic analysis which can be done with bytecode instrumentation are execution tracing, profiling, [14] and data race detection [13].

There are many tools for analysing and instrumenting Java source- or bytecode. How-

ever, many are specialised to specific tasks. Here some tools are presented which can make general instrumentation on Java bytecode programs.

2.2.1 Java Instrumentation API

The standard API for bytecode instrumentation has been available since Java 5. It is meant only for monitoring the behaviour of programs, not for changing the behaviour.

The instrumentation can be performed dynamically, when the program is being run in the JVM, or statically, at the JVM startup. [15] To perform the instrumentation, a special agent class must be implemented, packaged in a Java Archive (JAR), and registered with the JVM at startup or during runtime. [15] By adding a transformer to the agent, all classes which are loaded after the agent has been added can be examined and instrumented. [16]

2.2.2 ASM

ASM is focused on runtime instrumentation and is therefore designed to be fast and small in memory. [17] It uses the visitor patter: a class visitor looks at one class at a time rather than representing the classes as a graph. [17] Because it only keeps track of one class at the time it is not aware of class hierarchies. [17]

Jahromi and Hona compared the use of ASM and Soot as a front end for call graph construction. They found that Soot is slower than ASM as it has a larger overhead, but Soot has better support for extracting information such as parameters and return values, a task that requires much more coding when using ASM. [18] For the intended use in this project, the modified bytecode of a program only needed to be compiled once. Therefore, simplicity in coding and access to more complicated functionality like class hierarchy information was prioritised over fast execution.

2.2.3 Soot

Soot is a framework for Java optimisation and instrumentation developed by the Sable Research Group at McGill university [19] and was released in 2000. [20] Soot is published under the GNU Lesser General Public License v2.1 and the source code is available at https://github.com/Soot-oss/Soot. [21]

Soot is a compiler framework for Java and it has three main intermediate representations (IRs): [22]

1. Baf is a stack-based IR for optimisation and analysis that work better on stack-based code. It is also used when producing a Jimple representation. [22]
2. Jimple is a typed 3-address code designed to simplify optimisation. [22]
3. Grimp is used for simplifying decompilation and is the most similar to Java. [22]

Soot accepts class files in Java bytecode compiled by a compiler such as javac [22, 23], certain versions of Java source code, Jimple (Soot IR) or Jasmin. [19] The input class is translated to Jimple via Baf. [22] It is then analysed and optimised in Jimple.

[22] The Jimple representation can be transformed back to Java bytecode via Grimp or Baf and can subsequently be interpreted and executed by the JVM at runtime. [22, 23] Alternatively, the Jimple representation of the program can be further optimised or compiled to a different language such as C. [22, 23] Soot can also output the compiled class files as Android bytecode, Jasmin or Jimple, regardless of the format of the input classes. [19] This project analysed and instrumented Java bytecode.

In terms of program analysis, examples of what Soot can be used for include call-graph construction, definition-use chains or points-to analysis. [19] Soot performs static instrumentation, meaning that a compiled instrumented version is produced, rather than the instrumentation being inserted during execution.

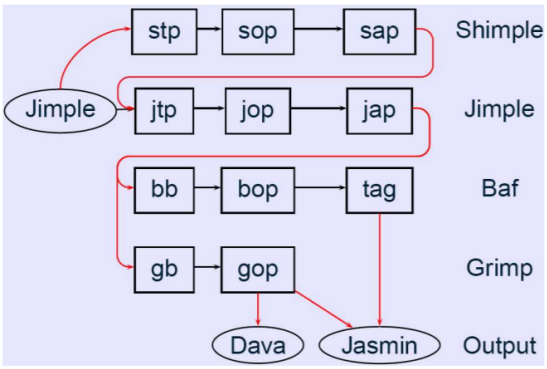


Figure 2.1: Soot phases, packs, for processing a Jimple class. Image from Vallée-Rai et al. [23]

Soot processes classes in phases, called packs in Soot, represented as rectangles in figure 2.1. Figure 2.1 shows the packs which operate after the Jimple representation of a class is produced in the Jimple body creation pack. There are a few packs for each intermediate representation, shown in the rows of the figure and by the first letter of the pack names. The second letter of each pack name in figure 2.1 represents its function: body creation (b), transformation (t), optimisation (o) and annotation (a). For inter-procedural analysis, the packs in figure 2.1 are preceded by packs responsible for call graph generation, whole Jimple transformation, whole Jimple optimisation and whole Jimple annotation. [23]

To instrument the bytecode of a program, a user-defined transformation can be added to the transformation packs. The transformation should extend `BodyTransformer` to transform method bodies (inserted into the Jimple transformation pack) and `SceneTransformer` for a whole program (inserted into the whole Jimple transformation pack). The user-defined transformation should provide an implementation of the `internalTransform` method, which is the entry point to instrumenting the Jimple representation. In `internalTransform` it is possible to iterate over all the statements and specify modifications to be inserted into the Jimple IR. When the program is compiled back to bytecode, the modifications can be used for analysing the program while it is run. [22, 23] This way dynamic program analysis can be performed on any compiled class, without changing the source code.

2.3 Test Oracle

Much research and effort has gone into automating testing to make the testing process cheaper and faster. There are ways of automatically generating test inputs, but the bottleneck of test generation is checking these against the expected behaviour. This is the task of the test oracle, the procedure that can distinguish correct and incorrect behaviours. [24]

The test oracle problem is relevant in this report as it will be the domain of the usefulness experiments, with the purpose of testing if the traces can be applied in a useful setting.

Test oracles can be divided into four categories: [24]

1. Specified test oracles which base their judgement on formal specifications that form a mathematical model for aspects of the system's behaviour.
2. Derived test oracles that decide if the system's behaviour is correct or incorrect based on artefacts such as documentation, system execution, previous versions of or properties of the program.
3. Implicit test oracles focus on occurrences that almost always are errors: the events that cause a crash or execution failure such as buffer overflows.
4. The human oracle problem involves minimising the effort of the test developer when determining if the program's behaviour under the tests is correct or not. This is useful when it is not possible to automate a test oracle.

In the experiments in this project, a test oracle derived from system execution was considered, by feeding it the produced execution traces. The test oracle was proposed by Tsimpourlas et al. [3] and is a neural network (NN) approach to the test oracle problem. They gathered execution traces as a sequence of method invocations of tests of C/C++ programs that were labelled as passing or failing. Most of the buggy programs used for their study were injected with faults, rather than having real-life bugs. They embedded the traces as fixed length vectors and used a subset (15 %) of the traces as training input. The NN model was evaluated on the remaining traces and high precision, recall and specificity/true negative rate were achieved, averaging 89 %, 88 % and 92 % respectively. The work in this project is an extension of the work presented by Tsimpourlas et al. [3], with the goal of producing execution traces matching theirs but for Java programs. Their NN will in this project be used only to evaluate the usefulness of the produced traces.

Using neural networks to create a test oracle is an active area of research [25, 26, 27, 28, 29, 30]. These suggested neural networks operate on only the inputs and outputs of test cases. The reasoning behind this is to teach the NN to become a simulated model of the software application, based on the inputs and outputs. The unique aspect of Tsimpourlas et al. [3] is that they used execution traces as input to the NN, hypothesising that the execution traces hold useful information for identifying failing tests.

Chapter 3

Capturing the Execution Traces

The tool presented here instruments a program's bytecode using the strategies outlined in the background and further in the upcoming methods chapter to record the method invocations.

In short, the strategy for doing this is extending Soot Main with `TracerTransform` (figure 3.1) in the Jimple transformation pack (described in the background, section 2.2.3). When running Soot on a program, it will apply the added `TracerTransform` to the program. `TracerTransform` will get a Jimple representation of the methods in the program classes and iterate over its statements to find method invocations. It can then insert Jimple printing statements into the Jimple intermediate representation to print the names of the current and called methods, the arguments and return values of the method invocation.

When the transformation is finished, Soot will convert the program's Jimple representation back to bytecode and the inserted printing statements will be included. When running the program after performing this instrumentation, the information will be printed. Because the sequence of method invocations and the values sent as arguments and return values depend on how the program is run (for instance which parts of the program are run, control paths, and input data), the produced traces are dynamic.

This section will also outline the optional functionality of converting the values of the arguments and return values to unsigned numbers. This is the first part of encoding needed for the usefulness evaluation of the traces presented in the experiments, section 4.4.

3.1 Execution Trace Format

The tool was designed to find all method invocations in the code and insert printing statements to print the name of the calling method, the method it calls, its arguments, and its return values. The traces produced when running the instrumented program will contain one trace line per method invocation. The format of the trace lines, unless otherwise specified using the options in section 3.6, is:

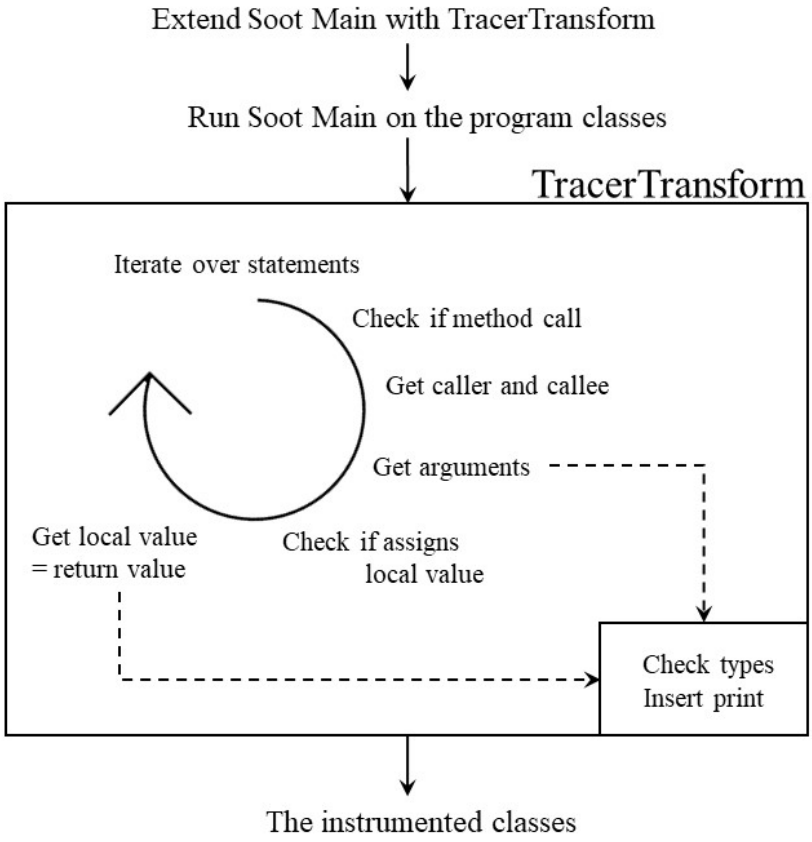


Figure 3.1: Overview of the steps taken to instrument the classes to capture the execution traces in `TracerTransform`.

```
caller callee, arguments, return values
```

The caller is the method in which the method invocation occurs and the callee the method that is called. The values of the arguments and return values are space separated. The trace line is printed after each method return, meaning that the tool performs a postorder traversal over the called methods.

Consider the example program in figure 3.2. It is a simple program that accepts two numbers from the input and calculates the square of their sum. It has two methods to perform the two mathematical operations, summing two values and squaring one value, and a method for printing the results. The trace produced by running the program with the inputs 12 and 8 after having instrumented it with the tracing tool is shown in figure 3.3a. Each line in the trace represents one method invocation. The first trace line in figure 3.3a highlights which parts of the trace line is the caller (main), callee (sum, the method being called), the arguments (12 and 8, the inputs to the program) and the return value (20, the sum of the arguments). The other trace lines have the same structure.

Running the same instrumented program with the inputs 44 and 67 produces the trace in 3.3b. The difference in the values in the two traces highlights the dynamic aspect of the traces. In more complex programs, the traces will also differ in which methods are

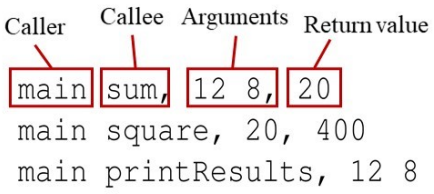
```
class Math {
  public static void main(String[] args) {
    //Parse the input
    int x = Integer.parseInt(args[0]);
    int y = Integer.parseInt(args[1]);
    //Calculate (x+y)^2
    int s = sum(x, y);
    int sq = square(s);
    printResults(x, y, sq);
  }

  public static int sum(int x, int y) {
    return x + y;
  }

  public static int square(int x) {
    return(x * x);
  }

  public static void printResults(int x, int y, int result) {
    System.out.printf("The result of (%d + %d)^2 is %d",
                      x, y, result);
  }
}
```

Figure 3.2: Example program for tracing that calculates the square of the sum of the input numbers x and y, $(x+y)^2$, using the three methods `sum(int x, int y)`, `square(int x)` and `printResults(int x, int y, int result)`.

called and in what order they appear in the traces.

This trace format was chosen because it is similar to that used in the previously published paper by Tsimpourlas et al. [3] when tracing their C/C++ programs. This will make the usefulness experiments of the traces possible, where the traces are fed into their neural net for classifying passing and failing tests. Tsimpourlas et al. [3] found that including more information in the traces introduced too much overhead in the classification which provided a natural and realistic scope to this project,

There is room for further work with the execution trace format in specifying the granularity of the traces to include more or different events in the execution traces. For instance, also recording which branches are taken in the control structures or global variable modifications might be useful for applications other than classification with the neural network.



(a) Inputs 12 and 8 (b) Inputs 44 and 67

Figure 3.3: Traces of the maths program calculating the square of the sum of the input numbers x and y, $(x+y)^2$. Each line in the traces represents one method invocation.

3.2 Extending Soot Main

To instrument the bytecode of the program, Soot is used for transformation to an intermediate representation (IR). The IR of interest for the static instrumentation performed for this project is Jimple. As described in the background, Soot creates a Jimple body of the bytecode classes and additional transforms to be applied to the classes can be specified by adding them to the Jimple transform pack (jtp). The `PackManager` is called to add a user-defined transform to the jtp:

```
PackManager.v().getPack("jtp").add(new Transform("jtp.myTransform",
    TracerTransform.v()));
```

Where the class `TracerTransform` is the user-defined transformation. It extends `BodyTransformer` because the goal is to apply the transformations on the method bodies of the original program:

```
class TracerTransform extends BodyTransformer {...}
```

To perform Soot's transformations on Jimple and to insert the user-defined transformation into the Jimple transformation pack, the Soot `Main` class' main method is called on the classes to be instrumented. The classes to be instrumented and the Soot options are given as arguments `args` when running the tracer tool's main method. They are then passed to the Soot `Main` class:

```
soot.Main.main(args);
```

3.3 Finding Invoke Expressions, Arguments and Return Values

The `TracerTransform` class which specifies the user-defined transformations that should be added to the Soot transformation must implement the `internalTransform` method:

```
@Override
protected void internalTransform(Body b, String phaseName,
    Map<String, String> options) {}
```

In the `internalTransform` method, the `Body b` will contain the body of the method

under instrumentation. Calling `getUnits()` on the body returns a `Chain` of the Jimple statements in the method. Any changes in the returned `Chain` will be preserved in the transformed Jimple class.

```
Chain<Unit> units = b.getUnits();
Iterator<Unit> stmtIt = units.snapshotIterator();
```

The iterator `stmtIt` is used to iterate over the statements in the method. If a statement contains an invoke expression, meaning it calls a method, it is recorded in the trace. The invoke statements can be identified using:

```
stmt.containsInvokeExpr()
```

Additionally, I only wanted to include calls to methods that are defined in the instrumented classes, the application classes, as to exclude calls to outside libraries. These classes are identified by Soot's `isApplicationClass()`:

```
stmt.getInvokeExpr().getMethod().getDeclaringClass()
                                              .isApplicationClass()
```

When iterating over the statements in the classes, statements which call `java.io.PrintStream` are removed to suppress the original output of the program and avoid the program output from interfering with the traces.

The arguments are found by getting the invoke expression (`InvokeExpr invExpr`) of the statement and using `invExpr.getArgCount()` to iterate over the arguments and `invExpr.getArg(int i)` to get a reference to them.

Return values are found by checking if the statement is an instance of `AssignStmt`, meaning the same statement that calls the method also assigns a local value. If that is the case, the statement is cast to an `AssignStmt` and the left operator can be extracted: `((AssignStmt) stmt).getLeftOp()`. The assignment is only preserved in the bytecode if the assigned local variable is used at a later point in the method and so return values are only included in the traces if they are later used in the original program.

The types of the arguments and return values are then retrieved using Soot's `value.getType()`. Depending on the types, appropriate Jimple statements to print the values are created and inserted into the chain of units/statements of the method. When Soot compiles the Jimple IR back to bytecode, these statements will be included.

3.4 Output: Data Types in the Execution Traces

The `printController` is the method of the transformation class that determines the type of a value, and calls the appropriate methods in the tool to insert statements into the Jimple unit chain to print the value. This section details which data types for arguments and return values are included in the produced traces.

Primitive types, identified by being an instance of Soot's `PrimType` or `IntType`, are included in the traces by inserting the matching version of `System.out`'s `print` method. As primitive data types cannot be null, no checks are needed before printing.

Strings: As strings are a reference type, they can be assigned null. Before attempting to call any methods on or print a returned or passed string, an if statement is inserted to ignore the value if the string equals to Soot's `NullConstant`. Next, any `\n` or `\r` symbols are removed from the strings to avoid them breaking up the traces. This is done by inserting a call to `java.lang.String`'s `replace` method, replacing any instances of the symbols with the empty string. Finally, resulting strings can be printed by inserting the standard `System.out.print(String s)` method.

Primitive and atomic wrapper classes are unwrapped by inserting a call to the wrapper class' method for extracting the value inside and then printing it using the printing instrumentation of the resulting primitive type. As with strings, a null check is inserted before calling any methods.

Arrays are printed by iterating over the elements using its length and indices and recursively calling the printing instrumentation on the elements, resulting in the values in arrays or nested arrays of supported types being included in the traces.

Application classes: For application classes only public fields are included, as these are the only fields that can be accessed from outside the class and the only fields that would normally be available when sending application classes as arguments or return values. After inserting a null check, Soot's field iterator is used to iterate over the fields. For each public field, statements are inserted to access non-static fields by referencing the object (which is an instance of the class) and static fields by referencing the class. The `printContoller` is then called again on the returned value to evaluate its type and insert statements accordingly.

NullType: Instances of `NullType` are always skipped as they carry no values.

3.5 Output: Encoding Return and Argument Values

The evaluation of the tool considered the usefulness of the traces. The usefulness was evaluated by tracing tests and deploying a neural net acting as a test oracle with the traces as its input. The neural net classifier wanted the arguments and return values to be sequences of unsigned primitives. For the sake of meeting the format required by the neural net for the experiments, the tool has the option of also inserting statements into the bytecode with the goal of converting the values in the traces to unsigned numbers. Using the option, the method names remain as normal but strings and negative numbers in the arguments and return value sections of the trace lines are represented as unsigned numbers.

More specifically, strings in the arguments or return values are encoded as byte arrays. Negative values stored in small numeric data types, byte (8 bits), short (16 bits), int (32 bits), are converted to their 64-bit two's complement (which corresponds to one 64-bit unsigned long). Values stored in the data type long (64 bits) are converted to two unsigned longs. For instance, -6 is converted to 4294967290 which corresponds to the 64-bit two's complement and -1099512676352 is converted to 4294967039 4293918720 which corresponds to -256 -1048576, the 128-bit two's complement represented as 2 longs. The floating point data types float (32 bits) and double (64 bits) remain signed

because the trace processor used in the experiments accepted such formats.

3.6 Using the Tool

Tracing using the tool is performed in two steps. The first step is to instrument the program using the tracing tool. The second step is to run the instrumented program like normal, using the instrumented class files, which will produce the execution traces while running.

The command to run the tracing tool on the program class files is:

```
java -jar Tracer.jar [trace options] [soot options]
```

Soot options are used to specify which class files to instrument, most easily done using the Soot option `-process-dir` to instrument all class files in a directory. For instance, to instrument everything in the folder `program/classes`:

```
java -jar Tracer.jar [trace options] -cp .program/classes
    -process-dir .program/classes
```

The Soot classpath needs to point to the folder which contains the classes to be instrumented. When instrumenting a package or classes in a package, the Soot classpath should point to the folder that contains the base of the package. For more Soot options see Soot command-line options [31].

Trace options: There are two methods of tracing. Not specifying the method (leaving the trace options blank) will result in tracing method invocations with arguments and return values, the format described in the rest of the report. But there is also the option of tracing only the methods in the order that they are visited. The tracing methods and trace options are:

1. Method 1 traces the method invocations in the order that they return to the caller method (postorder traversal). Options:
   - *invocations* — Include the caller and callee method names in the trace.
   - *arguments* — Include the values of the arguments in the trace.
   - *return-values* — Include the return values in the trace.
   - *encode-unsigned* — Encode in the arguments and return values the strings as byte arrays and convert numbers to their unsigned representation, excluding floats, as described in section 3.5.
2. Method 2 prints the method names in the instrumented classes when they are executed (preorder traversal). This method cannot be combined with arguments and return values in the traces. Options:
   - *visited-methods* — Include the function name of the visited methods in the order they are executed.

An option that can be used with both methods:

- *include-class* — Write the methods as: `Class.method name`.

Redirecting the output to file: The instrumented programs print the trace lines directly to the standard output using `System.out.print`. As `System.out` is line buffered by default [32], it has the advantage of printing the traces in real-time. However, efficient runtime may be preferred over real-time updates for the purposes of tracing. That is why `System.out` was redirected to a file print stream using the constructor `PrintStream(File file)` when running the experiments, which buffers the output before writing to the file without automatic line flushing. This strategy provides the freedom of designing your own experiments and deciding which files to redirect the traces to.

Chapter 4

Experiments

The experiments aimed to evaluate four aspects of the tracing tool: feasibility, usefulness, overhead and usability.

Feasibility aimed to test whether the tool scales to large real programs, as a way of testing if the tool works under real-life settings

Overhead and runtime experiments were conducted to evaluate the instrumentation cost and the overhead introduced by tracing a program.

Usefulness aimed to evaluate how well the traces produced by the tool could be used together with other systems. Usefulness was evaluated by the compatibility of the traces with an existing test oracle classifier which takes execution traces of tests as its input.

Usability testing involved evaluating the experience of running the tracing tool and if the functionality matched the expectations of participants previously unfamiliar with the tool.

4.1 The Test Programs

The test programs came from the Defects4J database, which is a database of open-source programs with real, reproducible bugs. [4] The database contains many versions of each program and each version of the program comes in two versions: a buggy version, caused by a real fault found in the program, and a fixed version where the bug has been resolved. The programs are also accompanied by developer JUnit tests. [4] For all the experiments, Defects4J 2.0.0 was used to access the programs from the database and compile the test programs, Java 1.8.0 was used to perform the instrumentation using the Tracer tool and Java 1.8.0 and JUnit Platform Console Standalone 1.7.0 of JUnit 5 to eventually run the tests and gather the traces.

The potential test programs were those Java programs considered by Papadakis et al. [33] in a study of the relationship between mutants and real faults, evaluated on real programs with real faults. Four of the five programs were used in the experiments,

Program	Version	Lines of Code	# Total Tests	# Failing Tests
Joda-Time	12	5800	3936	8
	14	5795	3906	8
JFreeChart	4	19646	2179	22
	26	17118	1591	22
Commons-Lang	34	3149	1670	27
	50	3271	1720	8
Commons-Math	6	17242	4174	28
	43	13646	2687	6

Table 4.1: The Defects4J test programs used for the experiments.

because the fifth program (Closure) had a high number of the types that were not covered by the tool, see section 3.4.

The chosen test programs were:

- Joda-Time, the standard date and time processing library before Java 8. [34]
- JFreeChart, a Java library for making charts. [35]
- Commons-Lang, a library from Apache Commons with utility classes for the `java.lang` API, the package which includes important classes such as `Object` and `Class`. [36]
- Commons-Math, a library from Apache Commons providing utilities for mathematics and statistics that are not available in the Java programming language or Commons-Lang. [37]

Many of the test sets from the Defects4J programs have a few thousand passing tests, but only a few failing tests (see table 4.1). To gather as many failing tests as possible for the NN model to train on, the two buggy program versions from Defects4J with the highest number of failing tests that produced non-empty traces for each of the four programs were included in the experiments.

4.2 Feasibility

The feasibility of the tracing tool was evaluated by instrumenting the test programs from the Defects4J database and tracing the execution of the JUnit tests through the target programs. This experiment tested if the tracing could scale from the small dummy programs used during development to large, real programs. The feasibility experiments tested the two steps of tracing: instrumenting a large program and then running the instrumented program (by running its tests) to produce the execution traces.

Producing large traces proved to be slow. Some tests produced traces of multiple GBs in size, meaning that the full trace sets would not have fitted on the available storage. Therefore, a timeout was introduced that stopped a passing test if it exceeded a certain trace size and removed that trace to avoid having incomplete traces. The failing tests

did not have the same check as they were deemed more important given the ratio of failing to passing tests. For Joda-Time the size limit was 20 MB because that was the first program that was traced. Subsequent programs had a limit of 2 MB. The size timeout was used for the traces in the feasibility and usefulness experiments.

4.3 Overhead

The aim of the next experiment was to establish how much overhead was incurred from the tracing instrumentation.

The runtimes and overheads were measured for two versions of each of the four test programs. To evaluate the overhead when running the instrumented programs, 100 tests were randomly sampled from the test suites that accompanied each program. The sampled tests were run 5 times on the original program, un-instrumented, and 5 times after instrumenting the program. The averages of the measured runtimes are presented in section 5.2. To estimate the runtime of the instrumentation, the average runtimes of instrumenting each program 5 times are also presented.

4.4 Usefulness

The next aspect of the tool, the usefulness, was evaluated by running the traces through an existing neural net (NN) designed to classify traces as passing or failing, distinguishing correct from incorrect behaviour in the program as a test oracle. [3] The NN was applied in two settings. In Experiment 1, the training and test set were traces from the same program version while Experiment 2 evaluated the generalisation performance for the traces of a different version (with a different bug) of the same program. The focus of the classification was not the performance but rather the compatibility of the traces and the existing classification system.

4.4.1 Neural Net for Classification

The NN was the classifier used in the experiments to classify traces as passing or failing (acting as a test oracle) and thereby evaluating the usefulness of the traces and tracing tool. The encoding and NN model described in this section were created and published by Tsimpourlas et al. [3].

4.4.1.1 Encoding the Traces

The values of the arguments and return values in the traces were represented as unsigned numbers, byte arrays or unchanged floats by the tracing tool, as described in section 3.5. The step of converting the values to a sequence of primitives was thus handled by the tracing tool presented in this report.

The implementation provided by Tsimpourlas et al. [3] further encoded the traces because the NN needed a fixed-length vector as its input. First, each component of the trace line was encoded. The sets of arguments and return values in each trace line were

converted to fixed-length binary vectors by the Long Short-Term Memory (LSTM) network `ValEnc`. The method names were encoded as a one-hot vector by `1Hot` with a unique representation for each method name except for rare methods which were all collected in a special Unknown category and name. [3]

To see how these encodings were applied to the trace lines, let (n_p, n_c, a, r) be the trace lines produced by the tool, where n_p is the caller (parent) method's name, n_c the called method's name, a the arguments and r the return values. Applying these encodings to the values and method names turns the trace line into
$\mathbf{t_i} = [ValEnc(r), ValEnc(a), 1Hot(n_p), 1Hot(n_c)]$ which is a fixed-length binary vector. [3]

When the individual trace lines were encoded into the fixed-length binary vectors as in the example above, each whole trace was encoded to a single fixed-length vector using another LSTM. The final vector could be accepted by the classifier. [3]

4.4.1.2 Classifying the Traces

The classification was performed by a multi-layer perceptron with a single output which gave the probability that a trace belonged to a failing test given the encoded traces. The whole network of encoding and classification was trained together in a supervised manner. More details about the network can be found in the original paper [3].

Like Tsimpourlas et al. [3], the training was conducted in 20 epochs and the epoch that performed the best on the training set was evaluated on the validation set. To reduce the length of the traces to a more manageable size, only the first 150 and last 150 lines of each trace were included in the experiments. Any traces less than 10 lines in size were also removed because they cluttered the dataset and removing them somewhat remedied the class imbalance. The traces with between 10 and 300 lines were kept in their original form.

4.4.2 Performance Measures

In the two-class classification problem, the failing traces were given a positive label and passing traces were given a negative label. This choice of labelling meant that true positives (TP) were the correctly classified failing traces, false negatives (FN) the failing traces incorrectly classified as passing, true negatives (TN) the traces correctly classified as passing traces and false positives (FP) the passing traces incorrectly classified as failing. Using these definitions, the performance measures used for the classification were:

- Precision $= \frac{TP}{TP+FP}$ which measures the ratio of correctly classified failing traces against all traces labelled as failing by the classifier.
- Recall $= \frac{TP}{TP+FN}$ which measures how many of the failing traces were correctly classified as failing by the classifier.
- True Negative Rate (TNR) $= \frac{TN}{TN+FP}$ (also called the specificity) which measures how many of the passing traces were correctly classified as passing by the

classifier.

4.4.3 Experiment 1 - Using a Single Program Version

The first usefulness experiment was performed on a single program version, training the classifier on a subset of the traces and classifying the unseen traces as passing or failing. The outline is illustrated in figure 4.1. The purpose of the first experiment was to stage a situation where there is a large set of tests, likely auto-generated, but one only wants to manually label each test in a subset as a pass or fail and have a classifier label the remaining traces.

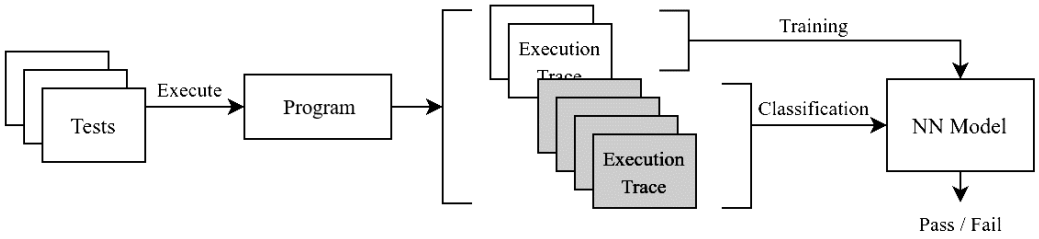


Figure 4.1: Outline of usefulness Experiment 1. Image from Tsimpourlas et al. [3]

For the first experiment, the NN described in section 4.4.1 was trained on 40% of the traces and tested on the remaining 60% of traces of the same program. The 40/60 train to test ratio was followed for both the set of passing and failing test traces. These numbers were chosen to give the NN enough failing traces to learn from, considering the small number of failing tests and traces and subsequent class imbalance of the programs.

4.4.4 Experiment 2 - Training and Classifying on Different Program Versions

The second type of usefulness experiment was training the NN model on all the traces of one program version and testing it on all the traces of another version of the same program, as presented in figure 4.2. The purpose of this experiment was to test the ability of the NN to generalise to a different program version containing a different bug. This type of experiment was not performed in the paper of Tsimpourlas et al. [3] and was possible because Defects4J offers multiple buggy versions of the same program.

As discussed in section 4.1, the second program version was chosen because it also had a high number of failing traces, providing enough material for the NN to classify. In the cases where the failing traces were empty or had less than 10 lines, another program version with a high number of failing tests was included instead. Selecting the program versions with the highest number of failing tests to include in the experiment meant that the version numbers used could diverge substantially. The large divergence in program versions was not ideal under the assumption that the program differed the

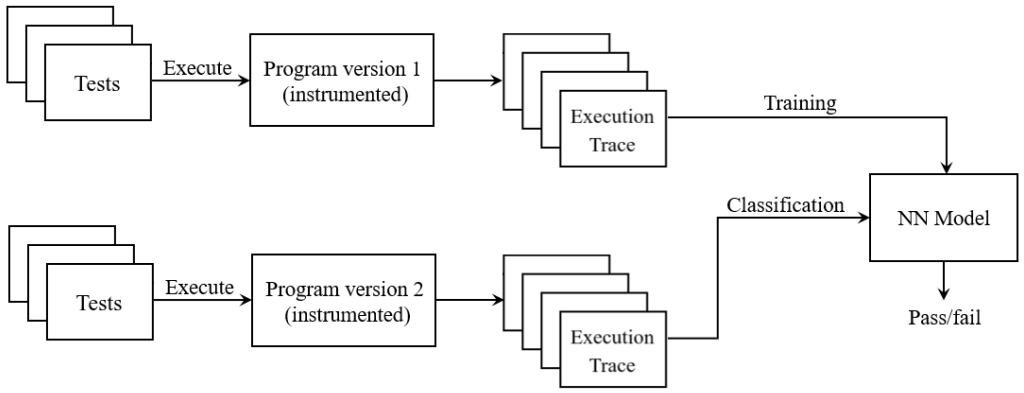


Figure 4.2: Outline of usefulness Experiment 2.

least for adjacent program versions and more for distant versions which could affect the patterns in the execution traces. This method was still used to ensure that the classifier had more than one or two failing traces in the set of a few hundred to thousands of traces.

4.5 Usability

The usability testing aimed to evaluate how easy the tracing tool is to use. It involved having 10 participants run the instrumentation and tracing. It evaluated only the tracing tool: no classification was performed after the tracing.

4.5.1 The Test Programs

The participants were given two example programs to instrument. Example program 1 was a small program with three class files from Well House Consultants that processes railway statistics, found online[1]. Example program 2 was Commons-Lang version 34 used throughout the experiments of this report and was accompanied by the set of JUnit tests that produced traces smaller than 2 MB and a class RunSomeTests that ran the tests and saved the trace of each test to a file. Both the example programs displayed the runtimes to the participants.

4.5.2 The Testing Procedure

The participants of the user testing were mostly unfamiliar with the project and the goal of the tool, bar the description given as part of the instructions. All participants were students within the School of Informatics and as such familiar with using a command line and running Java programs.

The tracer tool was packaged as a JAR that included Soot. It came with a set of instructions outlining the steps needed to produce the traces. In summary, those steps

[1] http://www.wellho.net/resources/ex.php4?item=j713/d2x.java

were:

1. Download the tool and example programs.
2. Try running both example programs before instrumentation.
3. Instrument the example programs using the tracing tool. This step included a description of the program options described in section 3.6 of this report.
4. Run the instrumented programs to produce the traces.

After following the instructions, the participants were asked to answer some questions in a web-based form about the experience.

For each example program, the form included the questions:

- Was the tracing successful?
- Did the output match your expectation considering the goal of the tool and the chosen program options?
- Did the runtime meet your expectations?

The questions about the experience overall included:

- How easy was it to use?
- Does it offer the features you would expect?
- Optional suggestions.

All the above questions had a 1-5 answer scale except the optional suggestions (text box) and whether the tracing was successful (yes/no).

Chapter 5

Results

5.1 Feasibility

The feasibility of the tracing was tested by tracing the execution of the tests of a selection of large programs from the Defects4J database of real programs. All programs were successfully traced (table 5.1).

Program	Version	Successfully traced (Y/N)
Joda-Time	12	Y
	14	Y
JFreeChart	4	Y
	26	Y
Commons-Lang	34	Y
	50	Y
Commons-Math	6	Y*

Table 5.1: Outcome of the feasibility experiment for the selected programs and versions. *Commons-Math was successfully traced, but with some difficulty.

Commons-Math version 6 was successfully traced (with timeout for the largest traces) but with some difficulty. Only one version of Commons-Math was fully traced and included in table 5.1, because the many large traces caused the tracing to take a long time and fill up the storage. This was an important result as it showed a limitation of the tool. This limitation became increasingly apparent in the overhead experiment (section 5.2).

5.2 Overhead

All instrumentation leads to some overhead and the aim of this experiment was to establish how much overhead was caused by the tracing instrumentation.

Program	Version	Avg runtime original	Avg instrumentation time	Avg runtime instrumented	# Trace lines
Joda-Time	12	2.4 seconds	3.0 seconds	46.6 seconds	314 937
	14	2.2 seconds	2.0 seconds	15.4 seconds	120 685
JFreeChart	4	2.2 seconds	3.8 seconds	5.6 seconds	38 157
	26	2.0 seconds	3.2 seconds	7.6 seconds	55 406
Commons-Lang	34	1.4 seconds	1.2 seconds	3.0 seconds	5 204
	50	2.8 seconds	1.0 seconds	91.2 seconds	208 392
Commons-Math	6	2.2 seconds	7.4 seconds	197 minutes	234 681 982
	43	1.2 seconds	4.0 seconds	47.7 minutes	54 479 897

Table 5.2: The runtime of instrumentation and the overhead of the instrumented programs presented as average (avg) per program version. The number of trace lines are the total number of lines in the traces of 100 tests to give an indication of how the size of the produced traces impacts the overhead.

The number of trace lines in table 5.2 is the sum of the number of trace lines produced by 100 tests for each program version and corresponds to the number of method invocations to an application class recorded by the tracer. It is included to give an indication of the size of the produced traces as this will affect the overhead. However, it does not consider the width of the lines (how many arguments and return values were in the traces).

Table 5.2 shows that all the programs had a significant overhead. All original runtimes were similar, between 1.2 and 2.8 seconds. The post-instrumentation runtimes all showed an increase and varied a lot between the programs. The overhead was large, the smallest increase was 114 % after instrumentation (Commons-Lang 34 going from 1.4 to 3.0 seconds) and the largest was a 537900 % increase in runtime (Commons-Math version 43 going from 2.2 seconds to 197 minutes). The overhead heavily depended on the behaviour of the tests. This is apparent from the overhead on Commons-Math which produced long traces, meaning it had a large amount of method invocations and had to execute a large number of added statements from the instrumentation.

The increased runtime can be handled by using a timeout (based on time or size of traces) when tracing to stop the execution. The traces up until the timeout will still be available.

5.3 Usefulness

The usefulness of the traces was considered to be how well they could be used in an existing system. In order to test the usefulness, the traces were used in two classification experiments, as described in section 4.4.

5.3.1 Experiment 1 - Using a Single Program Version

The results of the NN's classification on a subset of the test traces are presented in table 5.3. The classifier did not perform well for Joda-Time, JFreeChart and Commons-Math, which was not surprising considering their class imbalance. The low precision of

Program (version id)	# Traces	# Failing traces	Precision	Recall	TNR
Joda-Time (12)	3216	8	0.0055	0.60	0.72
JFreeChart (4)	1654	22	0.071	0.82	0.88
Commons-Lang (34)	586	27	0.71	0.94	0.98
Commons-Math (6)	885	12	0.083	0.75	0.87

Table 5.3: Performance of the NN classifying traces in Experiment 1. Experiment 1 trained on 40 % of the traces and tested on 60 % of traces from the same program version.

the classifier in Joda-Time, JFreeChart and Commons-Math was because many passing traces were misclassified as failing. For instance, in Joda-Time 540 negative traces (passing) were misclassified as positive (failing) while there were only 3 true positives (correctly classified failing traces). Even if the classifier classified all failing traces correctly, but did not improve the performance for correctly classifying passing traces, the precision would only improve to 0.0092. Overall, this means that the classifier was not able to find distinguishing features between the traces of the two classes for these programs.

The NN classifier performed better on Commons-Lang with 71% precision, 94% recall and 98% TNR, presented in table 5.3, despite having only a slightly better class ratio. Manually looking at the failing traces of Commons-Lang, most of them included a call from and to a string conversion method, `toString()`, towards the end of the trace, which is likely used by the classifier to discern the classes.

Tsimpourlas et al. [3] achieved an average of 89 % precision, 88 % recall and 92 % TNR with the same NN classifier on their traces of C/C++ programs and tests. The difference in performance can depend on many things such as the test programs, the type of bug, the quality of the tests, the information captured by the traces and/or the choice of NN parameters.

5.3.2 Experiment 2 - Training and Classifying on Different Program Versions

Program	Training version	# Traces	# Failing Traces	Validation version	# Traces	# Failing Traces	Precision	Recall	TNR
Joda-Time	12	3216	8	14	3198	8	0.0055	0.125	0.94
JFreeChart	4	1654	22	26	1014	23	0.019	0.17	0.80
Commons-Lang	34	586	27	50	511	4	0	0	1.00

Table 5.4: Performance of the NN classifying traces in experiment 2. In Experiment 2 the classifier was trained on all the traces of one program version and tested on all traces of a different version of the same program.

In Experiment 2, the classification performance on Commons-Lang when trained and testing on the traces from different program versions was unfortunately not as good as in Experiment 1. This was likely for the same reason that it did so well on the first experiment. The failing traces of program version 34 were distinct, all having

calls to `toString()` towards the end of the traces. The failing traces of version 50 did not have this common attribute. Instead, these traces were very small and tended to call methods that included the word ”Any”, such as `indexOfAny()`, or the method `containsNone()`. This suggests that the neural network fitted to the trace patterns corresponding to the bug rather than learning the normal or abnormal behaviour of the program.

The classification performance for the other programs was once again poor. This was not surprising as the classifier was not able to distinguish failing from passing traces from the same program version and the second experiment was a leap from that.

5.3.3 Overall Usefulness

The performance of the NN for classification was presented and briefly discussed in the two previous sections (sections 5.3.1 and 5.3.2). The performance of the classification can depend on many things apart from the quality of the traces. For instance, the test programs, the number or quality of the tests or the neural network. No attempts at tuning the experiments or improving the performance are included in the report as the main aim of the experiments was evaluating the usefulness of the traces in form of the compatibility of the traces and the existing NN-based classification system. This was successfully shown because the classifier can be run with the traces produced by the tracing tool as its input and further supported by the success of classifying the traces of Commons-Lang which shows that the traces hold useful information.

Thereby, the usefulness of the traces was shown by the classification experiments.

5.4 Usability

The replies to the user survey are presented in the figures below. Figure 5.1 shows the replies to the question ”Did the output match your expectation?” for the two test programs. The replies were mainly on the positive side. The reply of 1 on the 5-point scale (”Not at all”) for the two programs were from the same participant who seemed to have been able to run the tool to instrument the test programs but for some reason was not able to produce/see/find the traces when running the instrumented programs. As the form was anonymous, it was not possible to identify what caused the difficulties.

The number of replies were different for the two programs as all questions were optional and participants could skip questions. This mainly happened in the questions that required some more knowledge about the goal of the tool. In particular, if the output matched the expectation and if the tool offered the expected features.

Figure 5.2 presents the replies to the question ”Did the runtime meet your expectation?” and shows that the users were happy with the runtimes and overhead. Example program 1 was very small and reported instrumentation times (running the tracing tool) in the usability tests spanned 0 to 0.3 seconds and runtime after instrumentation was 0 to 6 seconds. Example program 2 was Commons-Lang used throughout the other experiments but included only the tests that produced traces smaller than 2 MB to not put too much strain on or require too much time from the participants. This resulted in

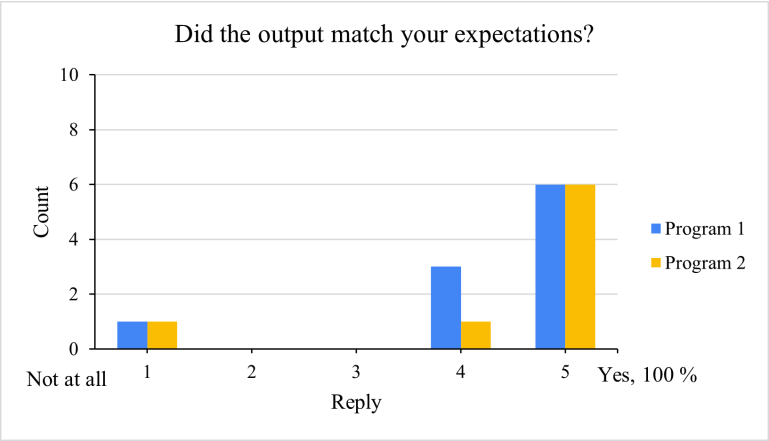


Figure 5.1: Replies to ”Did the output match your expectations?” for the two test programs.

the usability testing having a somewhat simplified tracing task compared to the tracing performed in the feasibility experiment (section 5.1) and overhead experiment (section 5.2). Reported instrumentation times for Commons-Lang were between 0 and 22 seconds and runtime after instrumentation 2 to 91 seconds with an average of 35.8 seconds. The form did not include a question about original runtime, pre-instrumentation.

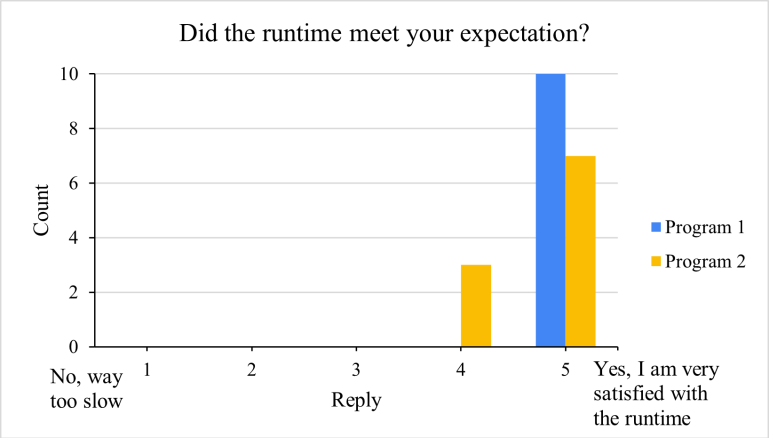


Figure 5.2: Replies to ”Did the runtime meet your expectation?” for the two test programs.

For the tool overall, the replies to ”How easy was the tool to use?” in figure 5.3 were very divided. This means that overall, the usability has room for improvement.

Figure 5.4 presents the replies to the question ”Does it offer the features you would expect?”. The replies were mostly positive or neutral and the one negative reply was from the same anonymous participant who answered a 1 - ”Not at all” when asked if the output matched their expectation. This participant seemed to be unable to produce the traces, which would understandably make it difficult to judge if the features were up to expectation.

Java versions used for the usability study included 8, 11, 14 and 15. The tool and

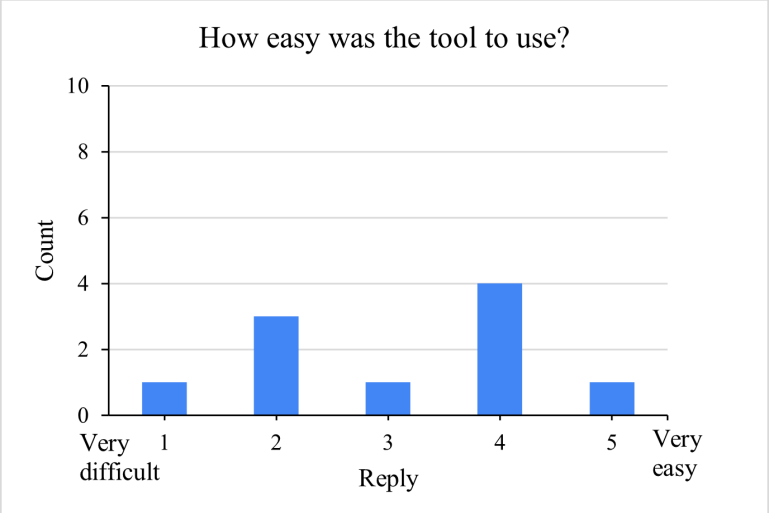


Figure 5.3: Replies to ”How easy was the tool to use?” for the tool overall.

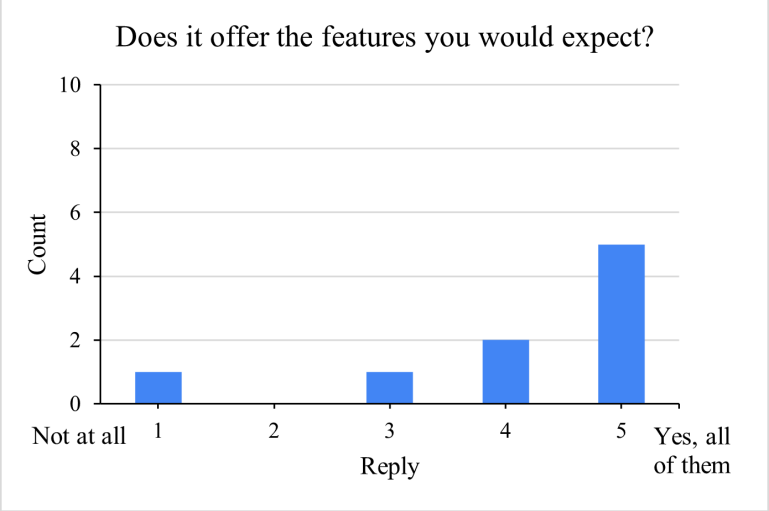


Figure 5.4: Replies to ”Does it offer the features you would expect?” with regards to the tool overall.

example programs were compiled with Java 8 and the tool was not compatible with Java 16 (as tested by the author).

The last question was an optional suggestion box. Notable suggestions included:

- Combining step 3 and 4 to perform the instrumentation and execution trace gathering in one step.
- Clarifying in the instructions that the output (the traces) is produced when running the instrumented program in step 4 rather than when running the tracing tool and performing the instrumentation in step 3.
- The traces of example program 1 were directed to the standard output (being displayed in the command line window where the tool was run from), while the traces of the JUnit tests from example program 2 were automatically saved to

files. The suggestion was to automatically save the traces of example program 1 to file as well.

- Highlighting the suggested commands in the instructions, especially those in the text.
- Making the instructions more concise.
- The tool did not handle typos in the program options gracefully.

Chapter 6

Conclusion

In summary, the implemented tracing tool instruments a compiled target program. It will insert logging statements into an intermediate representation to record method invocations. When the re-compiled program is run, execution traces will be produced.

Dividing the discussion of the evaluation into the four aspects:

Feasibility. The biggest threat to the feasibility was the overhead and size of large traces that were generated by some of the tests from the Defects4J programs' accompanying test suites.

Overhead. The overhead increased with the size of the produced traces rather than the runtime pre-instrumentation. This makes sense as a program with many method calls can normally run fast, but the instrumentation slowed the execution down with the need to log every method call.

Usefulness. The main aim of the usefulness experiments was showing that the traces produced by the tool provided a representation of the programs that could be used in a useful setting. The performance of the test oracle classification was poor for most test programs, but the results from Commons-Lang was enough to indicate that the tracing was useful and thereby also that the tracing was successful in capturing the programs' execution information.

Usability had some room for improvement. The tool seemed to be tricky to use at first but when figuring out how to use it, the participants were happy with the output, runtime and features of the tool.

Overall, the tool was successful in tracing the program execution but getting started with the tool required some initial effort from the user.

6.1 Further Work

The most immediate further work on the tracing tool is improvements to the content of the execution traces. This could include adding more types or adding more details in

the execution events to capture in the traces. Extending the program options to reflect new possible contents would provide the ability to personalise the traces.

A possibility for improvement is exploring ways of reducing the overhead. Possible ways of doing this include finding a faster way of collecting the trace contents before printing them or having fewer checks for the values to be included in the traces. The large traces are mainly the result of method calls inside loops. Another option for reducing the overhead is detecting loops with method calls and summarising them on the fly. This would however impact the format of the produced traces and the information contained in them.

The feasibility is threatened by the size of the produced traces which depends on the behaviour of the program (the number of methods calls, arguments and return values). Currently, the responsibility of stopping the program when it is producing the traces is on the user. Implementing a timeout as part of the tool was not a priority because it was not realistic in the time frame of the project and because the current tool gives more freedom in the tracing. An improvement could be implementing the timeout as part of the tool (and the instrumentation) and allowing the user to specify in the program options what criteria to use to stop the tracing (time or size) and when to stop the tracing.

Another area of improvement is usability. Improving usability requires further work in identifying what caused the difficulties in using the tool and how to simplify its use.

Based on this, the main plan for the second part of the MInf project is four-fold:

1. Develop the tracing tool to include more types. I will unpack more common data types to primitives to include in the traces and expand to object types with `toString()`. Non-primitive types in Java implement or inherit the methods of `Object`. All objects in Java have a `toString()` method which returns a textual representation of the object. This should be overridden to provide a human-readable representation of the object. If it is not overridden, the `Object` class returns the name of the class of the object and an unsigned hexadecimal representation of the hash code of the object. [38] Using `toString()`, the tool can also be extended to include the items of `Collection` types.

2. Develop the tracing tool to produce a different kind of trace. This may involve expanding the tool to optionally include global variable definitions or control flow.

3. Explore how the new information in the traces from the previous points can improve the classification of passing/failing tests based on their traces.

4. Explore other neural network architectures, beyond LSTMs with a multi-layer perceptron described in section 4.4.1, for encoding and classifying the execution traces.

Bibliography

[1] T. Ball. "The Concept of Dynamic Analysis". In: *Software Engineering — ESEC/FSE '99*. Edited by O. Nierstrasz and M. Lemoine. Springer Berlin Heidelberg, 1999, pages 216–234.

[2] J. R. Larus and T. Ball. "Rewriting Executable Files to Measure Program Behavior". In: *Software: Practice and Experience* 24.2 (Feb. 1994), pages 197–218. DOI: 10.1002/spe.4380240204.

[3] F. Tsimpourlas, A. Rajan, and M. Allamanis. "Learning to Encode and Classify Test Executions". In: *CoRR* (2020). arXiv: 2001.02444.

[4] R Just, D. Jalali, and M. D Ernst. "Defects4J: A Database of existing faults to enable controlled testing studies for Java programs". In: *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. 2014, pages 437–440. DOI: 10.1145/2610384.2628055.

[5] F. Tsimpourlas et al. "Embedding and Classifying Test Execution Traces using Neural Networks". In: *(submitted)* (2021).

[6] T. Galli, F. Chiclana, and F. Siewe. "Quality Properties of Execution Tracing, an Empirical Study". In: *Applied System Innovation* 4.1 (2021). DOI: 10.3390/asi4010020.

[7] A. Hamou-Lhadj et al. "Recovering behavioral design models from execution traces". In: *Ninth European Conference on Software Maintenance and Reengineering*. 2005, pages 112–121. DOI: 10.1109/CSMR.2005.46.

[8] N. Wilde and R. Huitt. "Maintenance support for object-oriented programs". In: *IEEE Transactions on Software Engineering* 18.12 (Dec. 1992), pages 1038–1044. DOI: 10.1109/ICSM.1991.160324.

[9] R. Almaghairbe and M. Roper. "Separating passing and failing test executions by clustering anomalies". In: *Software Quality Journal* 25.3 (2017), pages 803–840.

[10] W. Masri. "Chapter Three - Automated Fault Localization: Advances and Challenges". In: volume 99. Advances in Computers. Elsevier, 2015, pages 103–156. DOI: 10.1016/bs.adcom.2015.05.001.

[11] T. Richner and S. Ducasse. "Using dynamic information for the iterative recovery of collaborations and roles". In: *International Conference on Software Maintenance, 2002. Proceedings.* 2002, pages 34–43. DOI: 10.1109/ICSM.2002.1167745.

[12] Q. Xin et al. "Identifying Features of Android Apps from Execution Traces". In: *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering*

and Systems (MOBILESoft). 2019, pages 35–39. DOI: 10.1109/MOBILESoft.2019.00015.

[13] W. Binder et al. "Polymorphic bytecode instrumentation". In: *Software: Practice and Experience* 46.10 (2016), pages 1351–1380.

[14] J. Aarniala. "Instrumenting Java bytecode Seminar work for the Compilers-course , spring 2005". In: Department of Computer Science, University of Helsinki, Finland, 2005.

[15] Standard Edition 7 API Specification Java™ Platform. *Package java.lang.instrument*. 2020. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html. (accessed: 22.10.2020).

[16] Standard Edition 7 API Specification Java™ Platform. *Interface Instrumentation*. 2020. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html. (accessed: 22.10.2020).

[17] E. Bruneton. *ASM 4.0 A Java bytecode engineering library, Version 2.0*. 2011. URL: https://asm.ow2.io/asm4-guide.pdf.

[18] S. A. H. M. Jahromi and E. Honar. "A Framework for Call Graph Construction". Master's thesis. Sweden: Linnaeus University, 2010.

[19] Soot-oss. *Soot*. URL: https://soot-oss.github.io/soot/. (accessed: 20.10.2020).

[20] P. Lam et al. "The Soot framework for Java program analysis: a retrospective". In: 2011.

[21] Soot-oss. *About Soot - A Java optimization framework*. URL: https://github.com/soot-oss/soot. (accessed: 20.10.2020).

[22] R. Vallée-Rai et al. "Soot - a Java Bytecode Optimization Framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, page 13.

[23] Á. Einarsson and J. D. Nielsen. *A Survivor's Guide to Java Program Analysis with Soot*. 2008. URL: https://www.brics.dk/SootGuide/.

[24] E. T. Barr et al. "The Oracle Problem in Software Testing: A Survey". In: *IEEE Transactions on Software Engineering* 41.5 (2015), pages 507–525.

[25] H. Jin et al. "Artificial Neural Network for Automatic Test Oracles Generation". In: *2008 International Conference on Computer Science and Software Engineering*. Volume 2. 2008, pages 727–730. DOI: 10.1109/CSSE.2008.774.

[26] M. Vanmali, M. Last, and A. Kandel. "Using a neural network in the software testing process". In: *Int. J. Intell. Syst.* 17 (Jan. 2002), pages 45–62. DOI: 10.1002/int.1002.

[27] L. Ying and Y. Mao. "Oracle Model Based on RBF Neural Networks for Automated Software Testing". In: *Information Technology Journal* 6 (Mar. 2007). DOI: 10.3923/itj.2007.469.474.

[28] Y. Mao et al. "Neural Networks Based Automated Test Oracle for Software Testing". In: *Neural Information Processing*. Springer Berlin Heidelberg, 2006, pages 498–507.

[29] S. R. Shahamiri, W. M. N. Wan Kadir, and S. Ibrahim. "A Single-Network ANN-based Oracle to verify logical software modules". In: *2010 2nd Interna-*

tional Conference on Software Technology and Engineering. Volume 2. 2010, pages V2-272-V2–276. DOI: 10.1109/ICSTE.2010.5608808.

[30] S. R. Shahamiri et al. "Artificial Neural Networks as multi-networks automated test oracle". In: *Automated Software Engineering* 19 (Sept. 2012), pages 303–334. DOI: 10.1007/s10515-011-0094-z.

[31] *Soot command-line options*. URL: https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm. (accessed: 07.01.2021).

[32] G. McCluskey. *Articles: Tuning Java I/O Performance*. Oracle. Mar. 1999. URL: https://www.oracle.com/technical-resources/articles/javase/perftuning.html. (accessed: 17.03.2021).

[33] M. Papadakis et al. "Are Mutation Scores Correlated with Real Fault Detection? A Large Scale Empirical Study on the Relationship between Mutants and Real Faults". In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE '18. Association for Computing Machinery, 2018, pages 537–548. DOI: 10.1145/3180155.3180183.

[34] *Joda-Time*. Joda. 2021. URL: https://www.joda.org/joda-time/. (accessed: 25.03.2021).

[35] *JFreeChart*. JFree. 2021. URL: https://www.jfree.org/jfreechart/. (accessed: 25.03.2021).

[36] *Commons Lang*. Apache Commons. 2021. URL: https://commons.apache.org/proper/commons-lang/. (accessed: 25.03.2021).

[37] *Commons Math*. Apache Commons. 2016. URL: https://commons.apache.org/proper/commons-math/. (accessed: 25.03.2021).

[38] Java™ Platform Standard Ed. 8. *Class Object*. 2021. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html. (accessed: 07.04.2021).