Department of Computing

Imperial College London

Loupe – Discovering the Impact of Program Patches

Author: Artur Spychaj

Supervisor: Sophia Drossopoulou *Second marker:* Alastair Donaldson

Co-supervisor: Timothy Wood

June 16, 2015

Abstract

Programmers write code patches in order to make an application better, some of which change behaviour of the application. Sometimes these behaviour changes are unexpected and not what the programmer expected. A tool called `SVM` combines two techniques – symbolic execution and trace equivalence analysis – to find scenarios where unexpected behaviours are introduced. `SVM` does not scale with the size of the application as larger applications have more paths and each of the paths takes more time to execute.

We present our tool `loupe` that extends `SVM` by abstracting a user provided list of classes making the application we are analysing simpler. `Loupe` processes the original source code and creates an a class that approximates the sequence of method calls and return values of the non abstract class.

We evaluated the performance gain achieved by the use of abstractions. We show that by making abstractions over-approximate branch and loop conditionals symbolic execution explores scenarios 30 times faster. We show that this results in `loupe` being able to discover unexpected behaviour changes for twice as many benchmark examples within the same time limit. Furthermore we show that the presence of abstractions does not generate many false positives.

Acknowledgements

I would reallt like to thank Prof. Sophia Drossopoulu and Tim Wood for their guidance and thoughtful feedback throughout the project. Finally I would also like to thank my parents for all of their support and help.

Contents

1 Introduction

Software is engineered through a continuous and joined effort of developers. As new features are required, code is modified over and over again. However, as the project gets large it also gets more brittle – changes which seem to solve one problem usually end up causing many more. In fact 15-25% of bug fixes do not completely fix the issue and often introduce new bugs [1].

Unit and integration tests are used in order to prevent bugged code changes. Nevertheless, while they give a rough estimate of what works, they are insufficient to catch subtle bugs [2].

These testing methods do not find all bugs because writing tests that catch all bugs takes a lot of effort. It would require writing tests that specify the entire behaviour of an application. It seems that a technique that looks specifically for unexpectedly introduced behaviours could do better. The problem of catching bugs introduced by code changes could be solved by running tools that allow us to express and verify in code whether **what we though we changed** is **what actually changed**.

We call such testing/specification methods which compare two versions of a program, *equivalence analysis*. The goal of this project is to investigate techniques to enhance performance of an existing equivalence analysis tool called *SVM*. The tool `SVM` takes two versions of a program and a specification of the maximum behavioural difference between the versions, and then checks that the actual behavioural difference is indeed at most the behavioural difference specified. The behavioural difference specification (*bds*) is given as a predicate which partitions all the objects that are instantiated during execution into the ones whose behaviour is allegedly affected by the modification and the ones whose behaviour is allegedly unaffected by the modification.

This bds predicate induces a partition of the program stack and heap at each execution step. Objects that are allegedly unaffected by the modification and any stack frames with a `this` pointing to such an object are said to be in the *unaffected part* of the program state, the rest of the objects and stack frames are said to be in the affected part of the program state.

The bds-specification is valid, if, for all program scenarios, at each execution step, the *unaffected part* of both versions are indeed equivalent. Equivalence means that the only difference between the unaffected parts of the state is the exact memory location that each object is allocated at – the shape of the pointers and the values of any primitives in stack variables or fields must be the same.

From the programmer's perspective such bds are useful for the following reason: If the bds is satisfied then the allegedly unaffected objects will indeed behave identically in each version. If the bds is violated then some allegedly unaffected object changed its behaviour.

However, checking whether bd-specifcations are violated is computationally expensive: it requires to search across a potentially infinite number of program scenarios, and for each scenario, compute differences between the two program versions. `SVM` reduces the number of scenarios that need to be considered though the employment of symbolic execution, and the complexity of comparison by only comparing frames (i.e. the receiver and arguments of method calls) rather than frames and heaps. The latter reduction has been shown sound in [3].

Because programs can have an infinite number of states `SVM` will scan through as many scenarios as it can given a time limit. This makes the tool unsound: if the tool does not find a violation within a time limit, violations may or may not be present in unexplored scenarios.

In this project we investigated whether it is possible to improve performance of `SVM` without excessively affecting its accuracy. We adapted ideas from mock objects [4], and replaced some objects by their approximations.

We called our new tool `loupe`. `Loupe` automatically generates approximations for unaffected objects indicated by a programmer. When run by `SVM` these approximations have the same behaviour.

Our hypothesis was that a) by carefully choosing the objects that we approximate, and the approximation that we use, that we can decrease the time to detect a bds violation compared to precisely symbolically executing everything, b) that since we introduce the same extra behaviour into the unaffected objects of both programs, both programs should respond in an equivalent way without violating a bds and thus the rate of incorrectly detected bds violations would be limited.

Indeed, we have applied our ideas to a test suite of 6734 lines and have deduced that approximations can bring a speedup of up to 30 times and introduced only a single false positive. Furthermore, by using abstractions we were able to reduce false negative rate from 66% to 33%.

1.1 Motivating example

The motivation example is an adapted example from the Program Equivalence through Trace Equivalence paper[3]. Listings 1 to 3 shows different versions of an application which was given the following setting:

> The program awards three prizes to the top three eligible students ordered by grade, and also logs which students were considered for prizes. Users complain that it is hard to see who was and was not considered for a prize in a list ordered by grade. The modified program logs students in name order instead. Unfortunately the programmer makes a mistake and the modified program award prizes to the wrong students.

Using `SVM` it is possible for the programmer find this unexpected behaviour change. Given this patch the programmer would proceed to define the bds. The affected objects include all instances of `Logger` since it now sorts the students. However, while other classes' source code was not changed the following instances of classes are also affected: `PrintStream` since it now prints an ordered list and `List`[1] because its items get reordered. It is important to note that the programmer would expect the `Prizes` class instance to remain unaffected.

However, this bds would be violated. Listing 2 changes the behaviour of the `Prizes` class. In particular `Prizes` expects to receive the same top students to award. Since the list gets sorted under some scenarios the returned students change.

The modification in listing 3 corrects the problem. This version copies the list before sorting it, and so the `Prizes` object will receive an equivalent list of students to reward as in version 1.

The main issue comes from the fact that the `Main` method makes a call to the `StudentsDb` class (`orderedByGrade` method). Because `SVM` uses symbolic execution this can cause the following troubles. Firstly some instructions might be overly complex to explore – for instance the program might compute checksums of packets in order to verify the validity of received data. Secondly tests become dependent on the state of the environment – for instance the database would return a list of students for which the violation does not occur. Thirdly many database queries are irreversible making it impossible to explore all possible scenarios.

[1]Also internal classes used in the representation of `Lists`

```java
interface ProcessDataSet {
  void process(List<Student> students); }

class Main {
  StudentDb db = new StudentDb();
  Logger logger = new Logger();
  Prizes prizes = new Prizes();

  void main() {
    db.orderedByGrade(students -> {
      logger.considered(students);
      prizes.awardTo(students); })}}

class Prizes {
  void awardTo(List<Student> students) {
    award(students.get(0));
    award(students.get(1));
    award(students.get(2));}
  void award(final Student student) {
    /* ... */ }}

class Student {
  String name() { /* ... */ }}

class StudentDb {
  void orderedByGrade(ProcessDataSet pds) {
    /* talk to DB */ }}

class Logger {
  void considered(List<Student> students) {



    for (Student s : students) {
      println("considered:␣" + s.name()); }}}
```

Listing 1: Version 1. This Java program awards prizes to top students. It also logs which students were considered for a prize.

```java
interface ProcessDataSet {
  void process(List<Student> students); }

class Main {
  StudentDb db = new StudentDb();
  Logger logger = new Logger();
  Prizes prizes = new Prizes();

  void main() {
    db.orderedByGrade(students -> {
      logger.considered(students);
      prizes.awardTo(students); })}}

class Prizes {
  void awardTo(List<Student> students) {
    award(students.get(0));
    award(students.get(1));
    award(students.get(2));}
  void award(final Student student) {
    /* ... */ }}

class Student {
  String name() { /* ... */ }}

class StudentDb {
  void orderedByGrade(ProcessDataSet pds) {
    /* talk to DB */ }}

class Logger {
  void considered(List<Student> students) {


    sort(students, compareStudentsByName());
    for (Student s : students) {
      println("considered:␣" + s.name()); }}}
```

Listing 2: Version 2. This Java program is a modification of the program in 1. Students are now logged in name order. However, this version awards prizes to the wrong students because the sort in Logger unintentionally mutates the list. The modified parts are highlighted.

```java
interface ProcessDataSet {
  void process(List<Student> students); }

class Main {
  StudentDb db = new StudentDb();
  Logger logger = new Logger();
  Prizes prizes = new Prizes();

  void main() {
    db.orderedByGrade(students -> {
      logger.considered(students);
      prizes.awardTo(students); })}}

class Prizes {
  void awardTo(List<Student> students) {
    award(students.get(0));
    award(students.get(1));
    award(students.get(2));}
  void award(final Student student) {
    /* ... */ }}

class Student {
String name() { /* ... */ }}

class StudentDb {
  void orderedByGrade(ProcessDataSet pds) {
    /* talk to DB */ }}

class Logger {
  void considered(List<Student> students) {
    List<Student> studentsCopy =
      new ArrayList<>(students);
    sort(studentsCopy,compareStudentsByName());
    for (Student s : studentsCopy) {
      println("considered:␣" + s.name()); }}}
```

Listing 3: Version 3. This Java program is a modification of the program in 1. Students are now logged in name order. This version awards prizes to the right students by copying the list of students before sorting it. The modified parts are highlighted.

`Loupe` solves those issues. If a programmer decides that `StudentDb` should be abstracted `loupe` would generate a class that has the same interface as `StudentDb`. However, instead of performing IO it would invoke methods or return symbolic values. In this example, we could approximate the effect of calls to the method `StudentDb.orderedByGrade` by returning a noop. However, given such an approximation no bds violation would be found. An abstraction generated by `loupe` would call the `ProcessDataSet.process` method passing it a symbolic `List` of `Student` objects, allowing us to find the bds violation without having to execute complex library code or perform IO operations.

The approximated objects behave the same in executions of both versions, so error states introduced by the approximation are often introduced into both versions. In this example a `List` of `Student` objects with less than 3 elements will cause an `IndexOutOfBoundsException` in the `Prizes.awardTo` method of each version. However, since both versions exhibit this same extra behaviour no additional spurious bds violations will be reported.

1.2 Objectives

The objective of this project is to generate abstractions given their concrete implementations.

1. Automatic generation of code abstractions - Make a tool that automatically generates abstractions that closely resemble the concrete implementation. The abstractions need to be able to deal with IO interactions.
2. Validate performance improvement - Evaluate whether or not abstracting the program increases the speed with which bugs can be found.
3. Validate accuracy - Evaluate whether or not abstracting a program generates false positives compared to using just a concrete version.

1.3 Contributions

1. Created a tool that makes it easy to perform equivalence analysis between the two versions.
2. Created library that generates abstractions. This library over approximates classes in order to increase the rate with which unexpected behaviour changes are found.
    - The most prominent feature is an efficient abstraction API used to express non deterministic behaviours. Being designed with performance in mind it minimises the number of SMT queries made during symbolic execution. As a result for several examples unexpected behaviours were detected faster whilst abstractions were used.
3. Made several optimisations to the `SVM` library in order to handle more complex examples.
4. Verified the effect of using abstractions on performance and accuracy of the tool. I show that using abstractions results in `loupe` being able to find violations in twice as many benchmark examples.

1.4 Report outline

- Background (page 15) covers the concepts necessary in order to understand how a trace equivalence engine works and how abstractions fit into checking of partition violations.
- Implementation section (page 38) discusses the implementation details that went into building of the abstraction generation. The section gives a general introduction to the front facing UI of the tool. Later the architecture section (page 41) details the general architecture of the tool that performs the abstraction and does trace equivalence analysis while generating abstractions section (page 45) details the process in which the abstractions are generated automatically.
- Limitations section (page 68) discusses some limitations that the current version of the analyser has and discusses ways in which some of these limitations could be mitigated.
- Evaluation section (page 71) discusses the methodology in which the hypothesis whether abstractions are useful in a multiple version analysis tool was evaluated and discusses the collected results.
- Conclusion section (page 82) gives the insights summarising the entire project. In addition it includes the future work section which contains ideas worth pursuing in order to make the tool scale even better and make it more approachable to the general user.

2 Background

This section gives a detailed introduction to techniques and approaches taken that are used in equivalence analysis. Each subsection will also mention the extent to which each technique is applied in this specific task and will summarize its benefits as well as limitations.

Section 2.1 discusses all of the core concepts on which this project is based.

Section 2.2 lists the main libraries that are used in this project.

Sections 2.3 to 2.5 discuss different alternative techniques and approaches that are applicable to the project.

2.1 Core concepts

2.1.1 Java internals

Java is a high level language. It tries to give the capability of writing source code that can be run on any platform. Making this goal achievable is possible since every Java program runs inside of a VM that interprets and executes Java code called Java Virtual Machine (JVM).

To minimise the complexity that would be necessary to port all Java semantics to each platform JVM interprets a lower level instruction set called Java bytecode. Bytecode is a stack based instruction set which is much less complicated compared to Java. For the JVM's instruction set does not have a notion of variables. Instead it requires objects to be stored at specific locations. For instance an operation `STORE 3` which means store at stack location 3.

The JVM instruction set in called Java bytecode and in some cases differs from Java. A few important to note differences are:

1. Bytecode has no notion of generics – since Java 1.5 Java allows code to be annotated with additional type information that allows to express such data structures as lists of objects of type `T`. The type system then verifies that the method calls satisfy the type and makes the language safer. However when the code gets compiled types get erased. Because of that they cannot be accessed when reading bytecode.

2. Bytecode contains `goto` statements – in Java all branch instructions are expressed with different control flow statements such as if/then/else, while, do-while or switch statements. However, in bytecode all of these are unified into a GOTO instruction.

3. Bytecode does not have complex instructions – it is common in Java to write a single instruction which makes multiple method calls that take multiple parameters. For instance it is not uncommon to see code like `System.out.printf("SomeText%s", text)`. However, such a complex operation is simply not permitted in bytecode. Instead, each argument is pushes onto a stack in a separate instruction. Once all parameters are pushed to the stack the method is called.

4. Bytecode has `NEW` and `<init>` methods instead of the constructor – in Java constructors are defined by creating a method named just like a class. In bytecode that method is renamed to `<init>`. Furthermore, in bytecode object construction is split into two bytecode instructions `NEW` and `<init>`. The `NEW` instruction allocates memory for the new object.

The `<init>` method called the constructor methods. Thus in the example below the Java code on the left would translate to the bytecode on the right.

```java
new ArrayList(5);
```

```
NEW java/util/ArrayList
DUP
ICONST_5
INVOKESPECIAL ArrayList.<init> (I)V
```

2.1.2 Behaviour equivalence analysis

An execution of a program can be expressed as a sequence of instructions. However, many programs depend on external input or non deterministic choices. Therefore, if we are interested in comparing the behaviours of a program we need to take into account all of the possible scenarios[2] and behaviours that these scenarios give rise to. These are represented by a tree for example fig. 1.

Consider the following code in listing 4. This application writes the `Hello World!` message. Under a scenario when it is not passed any command line arguments it prints the message to stdout. However when it is passed at least one command line argument it prints the message to the specified file.

```java
public class WriteSomeLogsMain {
  public static void main(String[] args) throws IOException {
    if (args.length > 1) {
      System.exit(1);
    }
    PrintStream w = args.length == 0
      ? System.out
      : new PrintStream(args[0]);

    w.println("Hello␣world!");
  }
}
```

Listing 4: WriteSomeLogs main application.

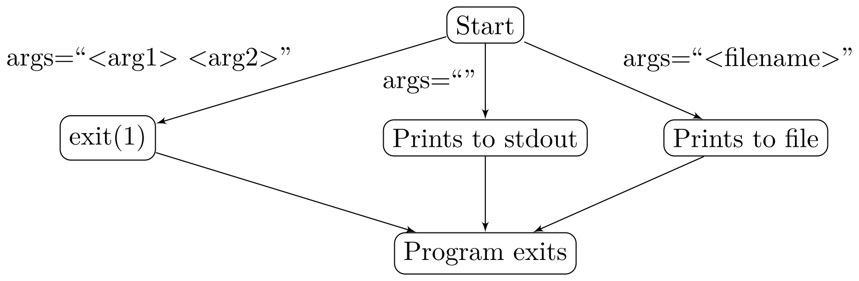


Figure 1: A simplified set of python executable behaviours.

Suppose a programmer would like to print `Hello World!` message in colour. This can be done by printing an escape code. For instance printing `\e[0;31m` will colour the text red. Thus the new version could look like listing 5.

[2]A scenario describes the set of inputs under which a program is executed.

```java
public class WriteSomeLogsMain {
  public static void main(String[] args) throws IOException {
    if (args.length > 1) {
      System.exit(1);
    }
    PrintStream w = args.length == 0
      ? System.out
      : new PrintStream(args[0]);

    w.println("\e[0;31mHello␣world!");
  }
}
```

Listing 5: WriteSomeLogs application. The new version makes the output coloured.

When comparing the behaviour of applications we need to consider all possible scenarios and see how the first application compares relative to the other. Behaviour analysis allows us to compare behaviours and check if they are equivalent. When comparing two versions the behaviour differences can be classified in three ways:

- Preserved behaviours – in this case both versions behave in the same way. For example when two arguments are passed both versions preserve the same behaviour – they exit with `System.exit(1)`.
- Expected behaviour change – in this case the applications behave in a different way. However, the difference has been expected by a programmer. For instance in the `WriteSomeLogsMain` the programmer wanted to colour the "Hello World!" message.
- Unexpected behaviour change – in this case the applications behave in different ways. However, the programmer did not expect such change. For example the new version will print `\e[0;31mHello world!` to a file. It is up to the programmer to decide whether this change was unintended.

There are two goals of this project. The first goal is to provide a specification format that allows to differentiate between expected and unexpected changes. The second goal is to efficiently find unexpected behaviour changes between the two program versions.

2.1.3 Symbolic execution

In order to build an equivalence analysis tool it needs to be able to explore different program scenarios and find the scenario for which the unexpected behaviour change occurs.

It is possible to use just the source code in order to find places where the behaviour is different[3]. Nevertheless, inferring information from the code is a difficult process as a lot of the contracts established by the source code are not written explicitly.

Symbolic execution is a technique which explores different program scenarios by simulating the source code. One of the earliest systems were the EFFIGY and SELECT systems [5][6]. Authors of EFFIGY describe symbolic execution in the following way:

[3]This is done by static analysis which infers properties from code. Code analyses are later explained in section 2.1.7.

> Instead of executing a program on a set of sample inputs, a program is "symbolically" executed for a set of classes of inputs. [. . .] The class of inputs characterized by each symbolic execution is determined by the dependence of the program's control flow on its inputs. [. . .] If the control flow of the program is dependent on the inputs, one must resort to a case analysis.

In order to express what this means let's consider the program from listing 5 again. `Args` is user provided, hence it could potentially be anything. Thus, it is treated as a symbolic variable. Instead of defining `args` with a single value `args` is defined in terms of a set of values.

When the program reaches line 3 the choice of whether it takes the then branch or the else branch depends on the symbolic input. At this point a case analysis needs to be performed, i.e. a symbolic execution must explore the case where `args.length > 1` and case where `!(args.length > 1)`. To do this a symbolic execution forks the program and explores both cases separately.

Executing the case where `args.length > 1` leads to a termination condition. On the other hand executing the case where `!(args.length > 1)` leads to another instruction which has its effect depend on the symbolic value of args. This is the instruction on lines 6-8. Thus both cases when `!(args.length > 1) && (args.length == 0)` and when `!(args.length > 1) && (!args.length == 0)` have to be explored.

As a result **all possible behaviours** for this program can be explored by checking 3 cases for the args value:

1. `args.length == 0`
2. `args.length == 1`
3. `args.length > 1`

It is not always the case that every branch can be taken. Consider the code in listing 6. In order to consider all program executions we have to consider the following cases:

1. `args.length < 0`
2. `!(args.length < 0)`

```
public void main(String args[]) {
  int x = args.length;
  if (x < 0) {
    assert false: "x␣should␣not␣be␣smaller␣than␣0";
  }
}
```

Listing 6: Program with an unreachable body.

What is important to note is Java semantics make arrays length always positive. Effectively the first case will never occur when executing a program. In order for the symbolic execution program to make such a deduction it sends the query to an SMT solver[4]. An SMT solver parses the query and returns whether a scenario is feasible. For this example the symbolic execution would have to augment the query with information about array length. Augmented query shown in fig. 2 would then be sent to the SMT solver.

[4]An SMT solver is a program that takes a query with a set of linear constraints and returns whether such constraints are satisfiable.

```
args.length < 0        &&        args.length >= 0
 case condition            Java arrays cannot have negative size
```

Figure 2: SMT query sent in order to check if one branch is feasible. Branch forms a part of the query. Since the condition concerns an array length symbolic execution must add a common fact that the length is non negative.

This fact that symbolic execution must send a query to an SMT solver to check if parts of the code are feasible is the first limitation of the technique. In general solving a constraint set is an undecidable problem. While the SMT solvers have improved in the past [7] solving complex constraint sets still takes a lot of time. Moreover, symbolic executors still spend most of their time querying the SMT solvers. A more modern symbolic execution program KLEE has made several improvements in order to make such queries more efficient [7].

Firstly KLEE checks for feasibility of getting to any branch. Because of this it does not waste CPU cycles exploring paths which could not happen in reality. For instance it would never simulate the line 4 as it cannot be reached.

Secondly KLEE caches SMT queries and simplifies them making. In effect it both sends less queries to the SMT solver and makes the queries simpler.

Thirdly it treats many variables concretely. For instance in this example by concretising the `args` variable the SMT query is no longer needed. The length of a concrete array is not smaller than 0. Hence, the branch condition would be equal to `false`.

The second limitation that symbolic execution needs to deal with is path explosion. Path explosion occurs because the number of cases increases exponentially with the number of branches through which the code goes through.

Consider the code in listing 7. It shows a program that prints all non null arguments in main. Unlike the array example there is no universal rule that would enforce array elements to be non null. Therefore at each branch the condition and its negation are both possible. The problem rises due to the fact that the program iterates 200 times. Because at each branch it can both take and not take the branch in total it will end up with 2^{200} unique paths. In effect the program will no longer be able to explore the entire program within any reasonable timeout. Furthermore most programs contain loops and have an infinite number of paths.

```java
public static void main(String[] args) {
  if (args.length < 200) {
    return;
  }
  for (int i = 0; i < 200; i++) {
    if (args[i] != null) {
      System.out.println(args[i]);
    }
  }
}
```

Listing 7: An example program for which the number of paths explode.

In order to deal with the path explosion problem symbolic execution engines are equipped with heuristic algorithms that choose the paths to explore. A good heuristic increases the likelihood of exploring paths that contain the interesting scenario (for KLEE a scenario which contains a bug).

2.1.4 Affected and unaffected objects

When a programmer makes a change in the code most of its behaviour is often preserved. As explained previously in section 2.1.2 we can classify behaviour changes in three different ways: preserved behaviours, expected behaviour changes and unexpected behaviour changes. However in that section we defined behaviour difference in terms of the observable output generated by the tool.

In general it is not necessary for the tool to generate different output in order to classify its behaviours as different. Thus we need a more general way of expressing code behaviours. A way that allows us to express the behaviour at the level of object instances classifying them as either *affected* or *unaffected*.

Unaffected objects are these objects which preserve their correspondence between two versions. At every execution point their correspondence holds and they will be in the same state. This might not hold for the *affected* objects. Thus for the `WriteSomeLogsMain` shown below we can classify objects in the following way:

1. `args` object (line 2) – unaffected since the arguments for any scenario correspond between each other.
2. `System.out` object (line 8) – affected since executing the line 11 by both versions causes the correspondence to be lost.
3. `new PrintStream(args[0])` object (line 9) – affected since executing the line 11 by both versions causes the correspondence between the objects to be lost.

```
 1  public class WriteSomeLogsMain {
 2
 3    String MSG = "Hello␣World!";
 4
 5    static void main(String[] args)
 6        throws IOException {
 7      if (args.length > 1) {
 8        System.exit(1);
 9      }
10      PrintStream w = args.length == 0
11        ? System.out
12        : new PrintStream(args[0]);
13
14      w.println(MSG);
15    }
16  }
```

Listing 8: WriteSomeLogs main application. Code differences with the second version have been highlighted.

```
 1  public class WriteSomeLogsMain {
 2    String RED = "\e[0;31m";
 3    String MSG = "Hello␣World!";
 4
 5    static void main(String[] args)
 6        throws IOException {
 7      if (args.length > 1) {
 8        System.exit(1);
 9      }
10      PrintStream w = args.length == 0
11        ? System.out
12        : new PrintStream(args[0]);
13
14      w.println(RED + MSG);
15    }
16  }
```

Listing 9: WriteSomeLogs application. The new version makes the output coloured.

Given this classification it is possible to differentiate between preserved behaviours, expected and unexpected behaviour changes. To do this a programmer making a patch would classify object instances as either *affected* and *unaffected*. A behaviour under a given scenario is preserved if all objects are *unaffected* for that particular scenario. A behaviour has made an expected change if there are *affected* objects however the programmer has classified all of them as such. Moreover, a behaviour change is unexpected if there are *affected* objects but they were not classified by the programmer as *affected*.

2.1.5 Trace equivalence

In object oriented programming code is split into classes that contain their own data and define methods. Methods can manipulate their data of a class instance. Furthermore a method can call a method on other objects.

To see how can we tell *affected* and *unaffected* objects apart consider the code below. Is the `Rectangle` object an *affected* or *unaffected* object? On the first glance you might think that `Rectangle` should be an *unaffected* object since we print its `area()` and `size()`. However methods can have arbitrary code. This would not be the case if `area()` method would be defined like in listing 11.

```java
public class Shape {
  static void main(String[] args) {
    Rectangle shape =
      new Rectangle(25);
    System.out.println(shape.size());
  }
}
```

```java
public class Shape {
  static void main(String[] args) {
    Rectangle shape =
      new Rectangle(25);
    System.out.println(shape.area());
    System.out.println(shape.size());
  }
}
```

```java
public class Rectangle {
  private size = 12;
  public Rectangle(int size) {
    this.size = size;
  }
  public int size() {
    return size;
  }
  public int area() {
    // Some implementation.
  }
}
```

Listing 10: Rectangle example.

```java
public class Rectangle {
  private size = 12;
  public Rectangle(int size) {
    this.size = size;
  }
  public int size() {
    return size;
  }
  public int area() {
    size /= 2;
    return size * size;
  }
}
```

Listing 11: A possible rectangle implementation.

Objects cannot lose their correspondence if they are called with the same sequence of method calls. Thus *affected* from *unaffected* objects can be distinguished from each other by comparing the sequence of method calls they receive along with parameters passed and values they return. Such a sequence of method calls is called a *program trace*.

Consider the listings 12 and 13. Until line 3 we can make say that the `Rectangles` are equivalent under an equivalence `{(Rectangle@1, Rectangle@2)}`. This means that if `Rectangle@1` is equivalent to `Rectangle@2` then all method calls performed on the Rectangle are equivalent. This is true since both Rectangles get constructed with the same value (25) and both return an equivalent address.

```
1 new Rectangle(25)
2 return Rectangle@1
3
4
5 Rectangle.size(Rectangle@1)
6 return 25
```

Listing 12: Possible trace of method calls on the `Rectangle` object of the first version of the `Shape.main` method. Each object is defined in terms of a unique address (Rectangle@1).

```
1 new Rectangle(25)
2 return Rectangle@2
3 Rectangle.area(Rectangle@2)
4 return 625
5 Rectangle.size(Rectangle@2)
6 return 12
```

Listing 13: Possible trace of method calls on the `Rectangle` object of the second version of the `Shape.main` method.Each object is defined in terms of a unique address (Rectangle@2). The method calls that are different from the first version are highlighted.

We can no longer make a correspondence between objects if two versions do not make an equivalent method call. An equivalent method call would mean that the instance gets mutated in an equivalent way preserving the correspondence. However, when a different method is called then the object **may** get mutated in an inequivalent way, losing the correspondence. The latter case is shown in listing 13 where the `Rectangle.size()` returns a different value.

What *Wood et al* have shown is that in order to detect whether a behaviour is preserved with classification of objects as *affected* and *unaffected* it is only necessary to trace all method calls at the *partition boundary* [3]. This means that only method calls made from *affected* objects to *unaffected* objects and *unaffected* to *affected* objects need to be traced. This works because in order for an *unaffected* object to lose correspondence it must either receive a not equivalent value in the other version or be the receiver of a non equivalent method. By definition *unaffected* objects never will make non equivalent method, thus do not need to be considered.

However this technique has a major limitation. It becomes very restrictive and reports objects as *affected* even if they have the same behaviour. Take a look at a different implementation of the rectangle. The trace of method calls receiver by the `Rectangle` object is shown in listings 14 and 15. It can be noticed that the behaviour of the Rectangle is preserved despite the fact that the `Rectangle.area` method is called. However, without checking the source code it is impossible to tell if the rectangles correspond to each other after line 4.

```java
public class Rectangle {
  private size = 12;
  public Rectangle(int size) {
    this.size = size;
  }
  public int size() {
    return size;
  }
  public int area() {
    return size * size;
  }
}
```

```
new Rectangle(25)
return Rectangle@1


Rectangle.size(Rectangle@1)
return 25
```

Listing 14: Possible trace of method calls on the `Rectangle` object of the first version of the `Shape.main` method. Each object is defined in terms of a unique address (Rectangle@1).

```
new Rectangle(25)
return Rectangle@2
Rectangle.area(Rectangle@2)
return 625
Rectangle.size(Rectangle@2)
return 25
```

Listing 15: Possible trace of method calls on the `Rectangle` object of the second version of the `Shape.main` method.Each object is defined in terms of a unique address (Rectangle@2). The method calls that are different from the first version are highlighted.

2.1.6 Control flow graph

Programs written by programmers are stored in files. They form a sequence of characters that combined with the semantics of a language define a behaviour of a program. However when programs need to process source code, control flow graphs serve a much more convenient representation.

Check the program in listing 16. In order to process the source code a program would store a 179 character long string. However, this representation is very difficult to comprehend. For instance the string does not have any semantics attached to it. Compilers or IDEs convert then the source code into an AST, a tree structure that represents the source code. An example of an AST is shown in figure fig. 3. However the AST representation is not the most convenient. To process an AST a program needs to specially handle every AST node corresponding to any statement such as an if statement, while loop or a switch statement.

To deal with that issue a more generalised control flow graph structure is used. When CPU executes a program it executes a sequence of instructions. A control flow graph is a directed graph which defines the order in which these instructions are executed by a program. Because of this it abstract over all control flow statements. The program from listing 16 would have a control flow graph given in fig. 4.

```
public static void main(String[] args) {
  String command;
  if (args.length == 0) {
    command = "run";
  } else {
    command = args[0];
  }
  System.out.println(command);
}
```

Listing 16: Simple application

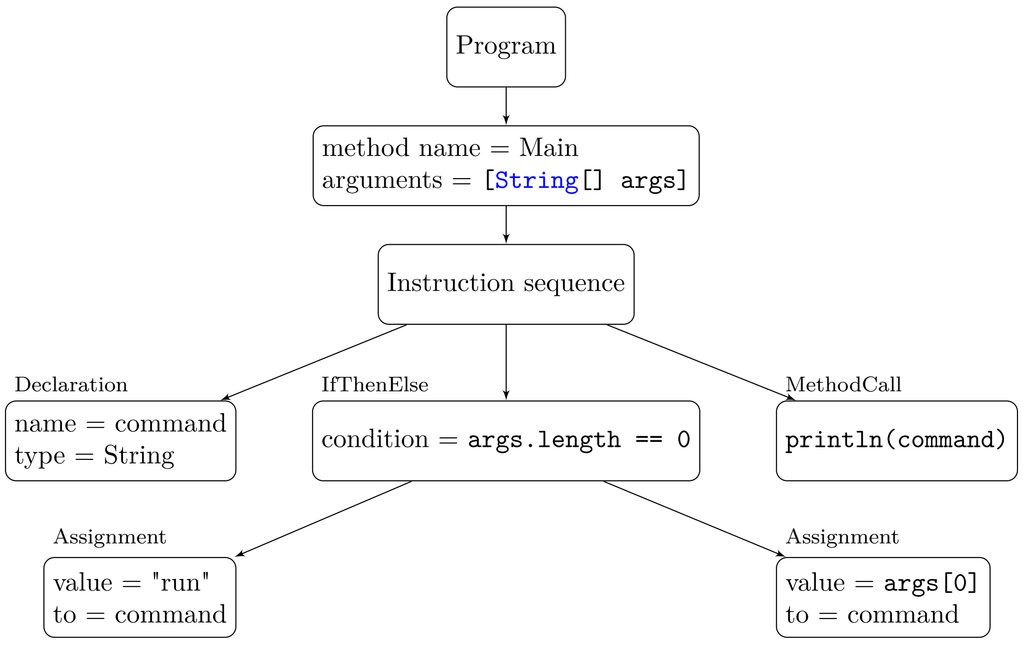


Figure 3: AST of the listing 16.

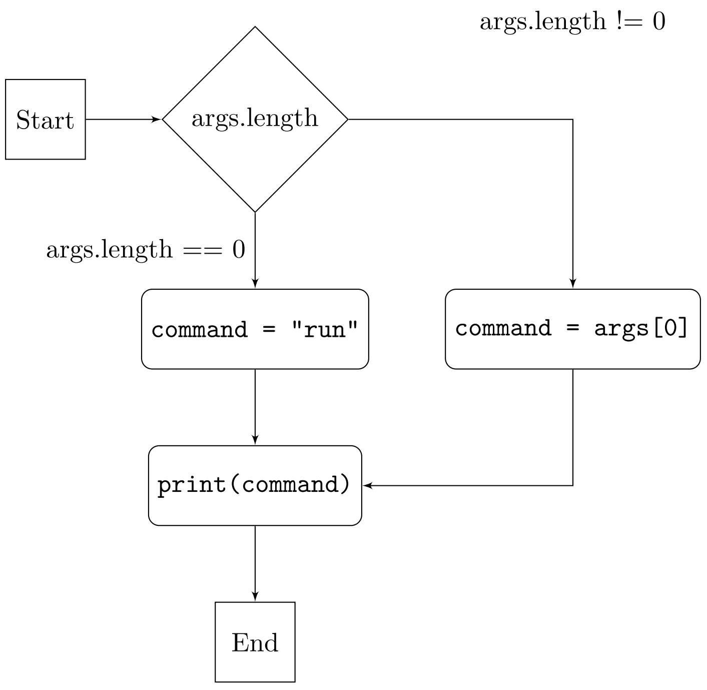


Figure 4: Control flow graph of the listing 16.

2.1.7 Code flow analysis

Source code is a text representation of a program behaviour. Since Java follows an imperative paradigm the method bodies define a sequence of instructions that the machine should execute. Each instruction then makes a corresponding change to the memory of the application.

Code flow analysis is a standard method of inferring properties of code represented by a control flow graph. In a code flow analysis the properties about the execution are propagated along the graph and combined during branch joins. Consider an analysis that would compute the set of variables with an even value. Take as an example the control flow graph shown in fig. 5.

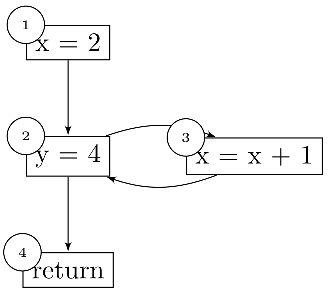


Figure 5: An example control flow graph. Circle nodes denote labels of statements.

From code inspection we could conclude with the following analysis. At statement 1 x is known to be even. At all other statements y is known to be even.

Code analysis is performed by splitting all statements into entry and exits. The analysis at the statement entry defines properties of the program that are reached when the program reaches a given statement. Analysis at statement exits defines properties of the program once the statement is executed. For instance the analysis of entry of statement `2` will return the set of variables that are even when the statement `2` is reached, i.e. right after statements `1` or `3` execute. However the statement exit states the properties after the assignment `y = 4` is made.

In order to compute the analysis at statement entries the results of the predecessors are merged. Consider statement `2`. In statement `1` only x is even. In statement `3` x is not even. In order to be certain that x is even at the entry to statement `2` it would need to be even when reached from all predecessors. Thus at the entry no statement is known to be even.

In order to compute the analysis at statement exits the analysis result at statement entry is processed with respect to the statement. In the case of statement `2` the entry returns no set of even variables. On the other hand after `y = 4` is executed the set of variables to be even is `{ y }`. This is the result of the analysis at the exit.

2.1.8 Points-to analysis

One of code flow analysis extensively used by this project is the points-to analysis. Points-to analysis is a technique which “tries to compute an accurate information about the behaviour of pointers” [8]. While Java does not have pointers per se it has references which are assigned dynamically and so require the same computation.

When creating programs Java uses a concept of variables in order to assign data to certain names. This allows programs to later refer to data by a previously defined name. Because of this for more complex algorithms variables get assigned values constructed in many places.

Consider the `WriteSomeLogsMain` program again. The example is shown below. In this example a variable `w` is created. It holds the object responsible for printing to the output. What we expect the points-to analysis to compute is to define the mapping from variables to the places where they get located. For instance we expect it to tell us that `w` can either be a file (`new PrintStream(args[0])`) or stdout.

```java
public class WriteSomeLogsMain {
  public static void main(String[] args) throws IOException {
    if (args.length > 1) {
      System.exit(1);
    }
    PrintStream w = args.length == 0
      ? System.out
      : new PrintStream(args[0]);

    w.println("\e[0;31mHello␣world!");
  }
}
```

Points-to analysis works just like any code flow analysis and propagates the assignments in order to find the sources of variables. For each variable, array, field assignment it adds a constraint adding the value to the variable mapping. It unions all of the computations and results with all possible locations for the variable definitions. This allows for instance to check if two variables might refer to the same object.

It is possible to apply the points-to analysis either locally and find all of the places within the context of the method body where the variable can be defined. On the other hand it is possible to perform a whole program analysis and check for variable assignments coming from the entire program. The advantage of the first analysis is that it is far faster than the second one. The advantage of the second analysis is that it is far more precise in defining the mappings.

A limitation of the points-to analysis just like any other static analysis is that it often over approximates the results and does not check for context. For instance consider the code below. A points-to analysis will infer that `s` can either be defined at lines 7 or 8. Furthermore it will infer that `w` can be defined either at lines 10 or 11. However it will be unable to distinguish the fact that whenever `s` is defined by line 7 then `w` is defined by line 10.

```java
 1  public class WriteSomeLogsMain {
 2    public static void main(String[] args) throws IOException {
 3      if (args.length > 1) {
 4        System.exit(1);
 5      }
 6      Scanner s = args.length == 0
 7        ? new Scanner(System.in)
 8        : new Scanner(new File(args[0]));
 9      PrintStream w = args.length == 0
10        ? System.out
11        : new PrintStream(args[0]);
12
13      System.out.println(s.next());
14      w.println("\e[0;31mHello␣world!");
15    }
16  }
```

2.1.9 Dominators

The dominator is defined in a control flow graph in the following way:

> Let the flow graph contain the starting node `r`. A vertex `v` dominates another version `w` such that `v != w` if every path from `r` to `w` contains `v` [9].

In programs this means as much as an instruction `v` dominated `w` if the following holds: whenever `w` executes `v` must have already been executed. What is important is not always equal to the predecessor. Consider the `WriteSomeLogsMain` example again (listing 4). Its control flow graph is defined by fig. 6. In this example the `w.println` statement is dominated by the program start and both if statements. However it is *not* dominated by either the w assignments.

Such a definition allows us to detect branches and loops inside of the code. Branches can be detected by checking if the predecessor is the dominator. If not then the dominator defines the branch instruction. Loops can be detected by checking if an instruction is dominated by its successor.

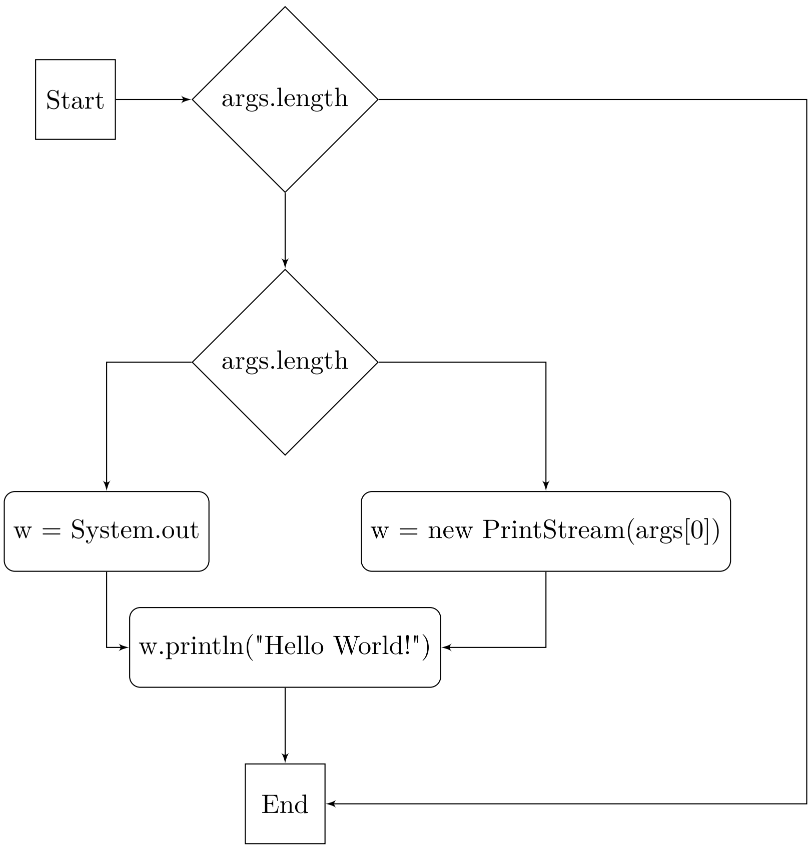


Figure 6: Control flow graph of the WriteSomeLogs example.

2.1.10 Most general context

A good analyser should provide only true positives and true negatives. Therefore it should not find “problems” that cannot happen in a real program’s execution. However it also should not miss any problems (false negatives) that will occur in a real program.

Static driver verifier (SDV) tries to verify that a windows driver will not violate the assertions made by the OS APIs [10]. In order to deal with false negative issues SDV models the OS APIs and implements a hostile model. The hostile model defines all behaviours that the API calls could cause, both the common ones as well as the uncommon ones.

A model deals with the fact that concrete execution (against the OS) is costly. Also it allows SDV to find rare bugs that do not occur often.

The approach introduced by Welsch generalises the approach by which models can be generated [11]. Welsch et al introduce a notion of a *most general context* that defines all possible behaviours of the environment. The *most general context* is being defined as a simple non deterministic loop. On every iteration it perform the following iterations:

1. Create new objects
2. Interact with existing objects it has access to
3. Return any objects it has access to

While the *most general context* allows to compare two libraries behaviour extensively it checks for far more than the observable behaviour. This is because actual behaviour of a class does not create arbitrary objects or call arbitrary method on them.

This is why other researchers usually take a far more cautious approach where the contexts are either generated manually (for instance in case of SDV) or some real contexts (such as the application main code or unit tests) are modified in order to generate the contexts for the application.

However, this limitation applies mainly to analysers that verify only a behaviour of a single version. When comparing two versions between each other the fact that both versions in most cases will respond in the same way when executed under the *most general context*.

2.1.11 Decompilation of code

A technique that reverses the compilation procedure is called a decompilation. `Dava` is one of decompilers for Java and is part of the Soot framework [12].

A decompiler needs to perform many transformations on the sources in order to produce high-level code. It needs to be able to interpret the low-level code into an internal representation. Then the decompiler needs to replace simple stack based operations with instructions. For instance it needs to be able to retrieve variable information.

The second stage is the restructuring state. This is the only decompilation stage that is performed by the `loupe` tool. Restructuring converts a code which can contain arbitrary flow statements (such as goto and break) into structured statements (such as if/else, switch).

The fundamental relation used by code structuring algorithms is a dominators relation. It is used to detect both loops and branch successors. Loops are detected by checking when a statement can jump to a statement it is dominated by. The fact that it is dominated by a statement means that at the same time it had to come from that statement. Branch successors are detected by finding a node which is dominated by the branch entry. In case `if/else` statements some of these nodes will overlap. These are the nodes which follow the branch and should not be a part of the then code block or the else code block.

The `Dava` decompiler uses generates Structured Encapsulation Trees (SETs) in order to create an initial structure on top of bytecode [12]. It performs step by step structure detection. Initially it detects and created do-while nodes. Then it locates try/catch blocks and finally if/else statements.

What normally limits the decompiler is the fact that compilation is not a reversible procedure. Thus decompiled code might not look like code written by a programmer. Despite this flaw decompiled code is still more readable than binary encoded bytecode.

2.2 Libraries used in this project

This section lists the most important libraries used by `loupe`.

2.2.1 Soot framework

Soot framework is a framework designed to process Java bytecode [13]. This framework provides many different transformations and analyses that can be applied to Java bytecode representations. It allows to represent Java code with multiple representations (jimple, grimp and shimp) all of which are defined in terms of control flow graphs.

The most important features of Soot for this project are:

1. Computes the control flow graph from bytecode.
2. Compute dominators of a control flow.
3. Compute the loops of a control flow.
4. Contains a common framework for creating and performing control flow analyses.

2.2.2 CodeModel

Code model is a library that allows programmatic creation of source code [14]. It provides a simple API to define modules, classes, methods making it easy to generate good looking Java code without having to do String manipulation manually.

2.2.3 IntelliJ plugin framework

IntelliJ is a Java IDE which allows to code, refactor and run Java applications from a single interface. It provides a set of plugin APIs that allow programmers to extend it with additional functionality. It this project these APIs are used in order to allow the analysis to be run from within the IDE.

2.3 Alternative approach used to run Java programs

Build tools such as maven or gradle allow the programmer to define the dependencies of the project, as well as the way in which it should be built, run and tested. This allows to install, run and test with a single command. For instance `mvn test` will run all tests in a maven project. This allows other tools able to automatically detect the entry point of a Java application without the necessity for the programmer to manually write it down.

This project does not support build tools. However, in order to make the setup effortless it integrates with IntelliJ "run configurations" that also define the way in which Java programs are run. Therefore if a Java project is run using a "run configuration" it can also be run by `loupe`.

2.4 Alternative approaches to code simplification

Class abstraction is a technique which allows to reduce the complexity of a class. There are however other techniques which also make methods less complex and allow thorough exploration of other bits of code.

These are method summaries (section 2.4.1) and uninterpreted functions (section 2.4.2). Method summaries describe how methods bodies can be replaces with a much less computationally complex summaries. On the other hand uninterpreted functions are functions that are not run at all.

2.4.1 Method summaries

Method summary is a technique which deals with the scalability issues for the programs by computing a disjunction of input effect pairs and has been used by Person et al [15]. It needs to be noted that the effect can in general also include object mutations.

$$\bigvee_i (precondition_i \wedge effect_i)$$

The preconditions ($precondition_i$) define the conditions that the input variables and the environment must pass for the program to have a certain effect while the effects ($effect_i$) state the statements which should be executed. In a complete method summary the union of preconditions would have to cover all possible input cases. However, it might be difficult to compute therefore we are often interested only in partial summary that only explores some input values.

The approach highlighted by Person computes complete method summary [15]. To decrease the initial overhead of summary computation Godefroid computes the method summaries lazily [16].

For example a factorial function shown below could be replaced by the following partial summary:

$$summary = \{n = 1 \wedge result = 1, n = 2 \wedge result = 2, n = 3 \wedge result = 6, \ldots\}$$

```java
public int fact(int n) {
  return n == 1 ? 1 : n * fact(n-1);
}
```

2.4.2 Uninterpreted functions

An uninterpreted function is a function that returns itself and the arguments passed instead of a value. Both Anand and Currie have discussed the use of uninterpreted functions in order so that they do not have to be executed [17] [18].

Later in order to compare the behaviours of the two program executions the uninterpreted function return values are equal if they were called with the same parameters.

It is similar to the way that symbolic values are treated in a symbolic execution. For instance consider an expression `args[0].length() + 1 == (args[0].length() + 2) / 2` the example below. If `args` is symbolic and the `length` function is pure then the expression always returns true. However, the expression `args[0].length() == args[1].length()` would not since the length method was called on a different receiver[5].

The disadvantage of this method is that it only allows to approximate objects that do not have side effects.

2.5 Alternative approaches to equivalence analysis

Trace equivalence analysis validates checks for behaviour differences against a specification. This section describes alternative approaches which also try to find the behavioural differences between the two versions of code.

2.5.1 Overview

Known under many names: *incremental testing*[19], *differential analysis*[20], *differential static analysis*[21], *incremental program testing*[22] differential analysis was being studied for at least 25 years.

There are many uses of differential testing, all of them with a goal to improve the testability of software. Lahiri et al lists more uses [21]:

> Although regression test generation has often been thought of as the ultimate goal of differential analysis, we highlight several other applications that can be enabled by differential static analysis. This includes equivalence checking, semantic diffing, differential contract checking, summary validation, invariant discovery and better debugging.

The two most largely tackled use cases are *differential* equivalence checking and *behaviour* diffing because they give a quick insight on the code change and do not require contracts to be written. Equivalence checking is an approach which validates if the behaviours of two programs are equivalent. In effect the result can be summarised by a single boolean (equivalent, not equivalent). Semantic diffing on the other hand should show the inputs for which the program has a different output.

The table 1 lists the different problems as well as already proposed solutions.

[5]This holds as long as `args` variables cannot alias one another.

Problem	**Technique** - How it works
Verify code changes	**Reverse refactorings** [23] - generate dependency graphs. Find atomic changes and detect if they were correctly applied **Trace equivalence** [3] - generate traces. Validate that they are not violating the change specification.
Compute adequacy of tests	**Control flow graph** [24] - An example of static analysis. Compute the *affected* paths. Using code coverage determine if the tests cover the *affected* paths well
Prioritize tests	**Symbolic execution and control flow graph** [24] - **Call graphs** [24] - split the changes into smaller atomic changes. For each atomic change construct the call graph and find affected tests.
Generate regression tests	**Automatic test input generation** [25] - get the changed classes. Create test inputs for these classes. **Symbolic execution** [26] - create a wrapper and use it somewhere else **Symbolic execution** [27] - use symbolic execution to guide the program into the branches which were not executed before.
Compute changed outputs	**Method summaries** [15] - use symbolic execution in order to explore potential method executions. Use the executions to build partial summaries and compare them across versions. **Automatic test input generation** [25] - get the changed classes. Create test inputs for these classes. **Use automatic test generation** [28, 20] - the authors check how well do the automatic test generators handle regressions. In how many mutations do they fail. **Dependency graph** [29, 22] - for each variable assignment a mapping from the variable name to free variables in the expression is made.
Improve efficiency of testing	**Method summaries** [15] - inline all method calls and loops to make these methods execute in $O(1)$ time. **Uninterpreted functions** [16] - do not execute unnamed functions. Instead log their method calls and compare such logs (or traces) to detect equivalence.

Table 1: Summary of alternative techniques and their use cases.

Even though the technique exploits the fact that a code patch constitutes only a small portion of the source code all of these cases have a problem with scalability - they cannot run for programs with a few thousand lines of code.

2.5.2 Dependency graph

One of the main ways in which a program behaviour can be defined is in terms of how data flows through the program. Assignment expressions can be used to define the dependencies between the variables where the assigned to variable depends on all of the free variables in the expression. This approach has been used by Jackson and Bates [29] [22]. For example in the expression `var1 = a + b - c` the variable `var1` depends on `a`, `b` and `c`.

In listings 17 to 19 the return value depends on `x` and `y`. In the first and the second program `x` depends on `ints[0]` and `y` depends on `ints[1]`. However in listing 18 the value of `y` depends on `ints[2]`.

```
void run(int[] ints) {
    int x = ints[0];
    int y = ints[1];
    return x + y;
}
```

Listing 17: First program

```
void run(int[] ints) {
    int x = ints[0];
    int y = ints[1];
    return y + x;
}
```

Listing 18: Second program

```
void run(int[] ints) {
    int x = ints[0];
    int y = ints[2];
    return x + y;
}
```

Listing 19: Third program

After having a list of dependencies we can construct a directed graph where the nodes are variables and the edge would represent that variable depends on another variable. This approach can be used to explain the data flow [29] [22].

This graph is easy to compute and allows to capture basic program behaviours and find the outputs which have a new behaviour. For instance it can detect the difference between the first two programs and the last one.

A problem with this naïve approach is that variables can depend on external inputs such as file reads as well as variables can be redefined. Also `x = a; x = x + y` has a different behaviour than `x = x + y; x = a` which will not be caught if we use a naïve notion of a dependency. In order to deal with that Bates et al use different sorts of dependencies to model the data flow including the read dependency that models the variable dependency on file content, def-order dependency which models a variable override and flow dependency to model simple assignments.

Even with this change there are potential flows of describing program behaviour using dependencies. Firstly this approach cannot deal with aliases. For example if we knew that `ints[1]` and `ints[2]` actually aliased the same value then the dependency graph will falsely show a behaviour difference. In addition Jackson et al have noticed that when two graphs have the same dependencies they do not necessarily have the same behaviour. Both `x = x + y` and `x = x * y` have the same dependency graph but they compute two different things [21].

2.5.2.1 Concolic execution Concolic execution is a technique closely related to symbolic execution. Both try to explore all program scenarios. However a concolic engine initialises variables to concrete values and then explores the code. It then collects branch constraints and uses them to generate a new set of inputs. It sends SMT queries to check if it can cover new branches.

The concolic executor uses a few techniques in order to make sure that a lot of states are being explored efficiently. Programs are explored in a *bounded depth-first search* so that the paths of boundless depth are not explored forever [30]. On each if statement the outcome of the branch condition is predicted. If the prediction is incorrect the program is restarted with a fresh input

[30]. This makes the symbolic executor guide the execution of the program by guessing the inputs for which a program would be executing a certain path.

Just like symbolic execution this approach also suffers the problem of path explosion as the number of feasible paths in the program is the same.

However it can be more opportunistic as it can simply create a random assignment to the variables and see in what state does the program end up. This is especially useful when the branch would have a condition like `hash(x) == 0`. In this case an SMT solver would be unable to find an answer and would potentially block the execution of the analyser.

2.5.3 Reverse refactoring

Many code changes only refactor code. A reverse refactoring technique validates if the programmer completed a refactoring change successfully. [23]. For example this technique checks if a programmer renamed **all** instances of a variable. Consider the following code:

```
public class Barometer {
  public void value() {}
}
public abstract class Valuable {
  public abstract int value();
}
public class HundredPounds extends Valuable {
  @Override
  public int value() { return 100; }
}
public class TwoHundredPounds extends Valuable {
  @Override
  public int value() { return 200; }
}
public class Pocket extends Valuable {
  private Valuable[] valuables;

  @Override
  public int value() {
    return valuables.stream().mapToInt(v -> v.value()).sum();
  }
}
```

The method `value()` has been mentioned in multiple places. There are two definitions of the method. One definition is made in `Barometer`. A second definition is made in `Valuable` and is then implemented in `HundredPounds`, `TwoHundredPounds` and `Pocket`. In addition there is also the call to a value method namely `v.value()` which refers to `Valuable.value`.

It is safe to rename the first method as there is only one instance referring to this definition. However for the second method there are 5 places which **all** need to be renamed atomically for the refactoring to be correct. More formally if we rename a method `foo` to `bar` we need to ensure that:

1. All overriding methods must be accordingly changed to `bar`.
2. All methods which it overrides must be renamed accordingly.
3. Any places where the method `foo` was called must be replaced with a call to `bar`.

Having this set of rules and two versions it is possible to find and validate such refactorings. With the example of the rename refactoring the presence of a method `foo` being removed and another method named `bar` being added both having the same code it is a valid assumption that this method was renamed [23].

In case the software finds any places where one of the rules above is violated it can raise warnings that the change is not a pure refactoring change. Of course there are limitation of analysing reverse refactorings. One is that finding refactorings does not always work [23]. Another limitation is caused by reflection.

For example a programmer can write the following code:

```
public static void main(String command) {
  Program p = new Program();
  Method method = p.getClass().getMethod(command, null);
  method.invoke(p, null);
}
```

This method takes the name of the command to be run and then calls a function named this way in the program.

So if we call the program with the argument run then the `run(null)` method could be called. Because the method call is dynamic, static analysis will be unable to directly link `method.invoke(p, null)` to the `command` method. This would most likely fail even in the case if we would call this by `main("run")` command. Under such circumstances the IDE would most likely fail to rename the method safely as well as validate if a manually done refactoring was indeed correct.

This limitation also applies to other types of static analysis and is one of the major problems because of which such analysis could omit invalid code changes. The upside is that most of the software being written does not need to make use of reflections in order to be implemented.

Another limitation of this approach is that usually code changes are not pure, i.e. many refactorings are applied to the code at the same time. For example a method can be renamed and its body may be changed. In this case the static analysis tool may be unable to determine what refactorings have been applied and in effect cannot validate their correctness.

2.5.4 Analysis through unit testing

An alternative to *checking* whether code adheres to bd-specs is to *generate bd-specs* out of code changes. Approaches in this section generate tests that demonstrate different behaviour. These tests therefore create a list of properties which hold true for some version of the application. Such tests are named Characterisation Tests [20] as they can be used to characterise what the program is doing. Evans proposes a method where these tests can be generated for the version before and after the patch is applied [20]. This results in a series of properties that the application is checked for. The properties which are passing for both versions are properties that the program preserves. What is useful in regression testing however are the newly passing and newly failing properties.

Newly passing or failing properties are used as the output for the differential tests. In addition this technique can validate if unit tests cover these properties. Of course running the test suite methods once poses a challenge for programmers to ensure that their tests cover different software behaviour. The challenging problem here is that the accuracy of the analysis will depend on how many tests have been written.

Bacchelli shows a study that compares the accuracy of such automated test generators [28]. In the paper a couple of programs are examined: Randoop, JUnit factory, JCrasher that can automatically generate unit tests for a program. The paper has shown that the unit test generators (in particular JUnit factory) was able to generate useful tests. In case when the code would be mutated to have a different behaviour from the original one the at least one of the tests would fail in 70% of the cases.

Jin created regression tests tailored to the code change being made [25]. This allows the approach to make tests execute faster; most unit tests pass as code patches are expected to preserve behaviour. Just like in the previous example, the refactoring tests are being run on the original and the new version in order to show which tests started failing or passing since the original version.

A similar approach has been shown by Ren where the entire code change is first analysed and split into smaller changes [24]. The tool assumes an existing test suite and for each atomic change the tests that might have been affected by such change.

The major disadvantage of automated test generation is that these techniques pre-compute a sequence of tests results. Ideally the test generation techniques would generate tests most likely to show the behaviour change. However if inadequate tests would be produced they could generate more.

2.5.5 Control flow analyses

2.5.5.1 Coverage analysis A method that tries to deal with the problem of how extensively the code is tested is control flow coverage analysis [31].

Dinh-Trong [31] looked at how can we measure the adequacy of tests and proposed to use the test code coverage as a good metric. Because different behaviours occur when the program takes different code paths, checking whether the unit tests have checked a lot of paths allows to determine if they have checked a lot of behaviours.

For instance if the program contained an assertion `assert(x > 0)` then the test suite should contain an input where `x <= 0`. If the test suite did not have such a test then the test suite would be inadequate not covering all behaviours.

Being able to measure how well are the regression tests cover the new behaviours helps to assure the programmer that introduced behaviours will not surprise him.

In order to be complete, the tests should cover all possible path interleavings. Under such circumstances we would try any combination of branches taken which allows us to simulate potential code behaviours. By decreasing the way in which code coverage is measured, we could potentially get behaviours faster.

The simplest coverage criterion is by *Blocks on Scoped Impacted Paths coverage (BSIPC)*. With this criterion coverage is defined in terms of blocks which have been covered. This is equivalent to line coverage which is a widely used metric for determining the accuracy of test suites.

The next criterion is *Branches on Scoped Impacted Paths coverage (BrSIPC)*. This criterion requires all of the paths within the affected code to be taken.

The final criterion is *Scoped Impacted Paths coverage (SIPC)*. This criterion is the strictest of them all. In order for tests to pass this criterion they must explore all feasible program executions that also execute modified code. Such tests have potential to find all different behaviours.

2.5.5.2 Finding affected tests One of the problems of large regression test suites is that they take a long time to run. Having to run all of the tests can be very painful especially given that most of them are not impacted by the code change and will therefore still be passing.

Ren et al try to deal with this issue by computing the control flow graph for each of the tests and then determining which of the tests can be potentially affected by the change [24]. This tests what methods did a test depend on.

In the end, only the tests which depend on modified code need to be re-run. This control flow analysis can also be used to detect poorly tested new behaviours in the case where the number of tests which depend on modified code would be suspiciously small.

2.5.6 Symbolic diff

Symbolic execution can be used to compare the program behaviours. This method is called *symbolic diffing* [15] [27]. The method tries to find a set of inputs for which the behaviours will be different between the two program versions. Symbolic execution is used in order to explore different paths that programs can execute.

Technique used by Person computes method summaries (a technique outlined in section 2.4.1). As a result methods can be summarised by a number of precondition effect pairs. Symbolic diffing is then performed by comparing these pairs in order to find some precondition for which the first version would have a different effect to the second version.

On the other hand Taneja uses symbolic execution to find externally visible behaviour difference. It uses the assumption that in order for there to exist a visible difference a new version of the code must be executed and then must infect the state and become externally visible [27]. Symbolic execution is thus a tool that guides the program in order to go though the affected code sooner. This generates tests for which the inputs which are more likely to have been affected by the code change.

3 Implementation of Loupe

`Loupe` is a tool created in this project that performs equivalence analysis. The following sections detail the implementation details of the `Loupe` tool. The sections discuss the current design, architecture, performance considerations and limitations of the tool.

`Loupe` is split into an IntelliJ plugin and a CLI application. In this report the IntelliJ plugin will be referred to as an `IntelliJ plugin` while the CLI program will be simply referred to as `loupe`.

The IntelliJ plugin acts solely as a front end interface for the CLI application. Its main purpose is to make `loupe` easier to use. `Loupe` itself is not tied to IntelliJ and could be integrated with other IDEs. Screenshots of the IDE plugin and details of its design are discussed in section 4.1.

The underlying CLI application is explained in section 5.1. `Loupe` takes the behaviour difference specification and looks for any partition violations between two program versions. `Loupe` is expected to return a JSON formatted output with the findings it made. `Loupe` relies on a symbolic execution library called `SVM` that explores the paths of both versions of the program [32].

4 Using Loupe

4.1 IntelliJ plugin for Loupe

`Loupe` is designed to analyse Java programs. Java programs are predominantly developed inside of an IDE which takes care of dependency handling, finding code sources and setting up command line arguments. Therefore, in order to make `loupe` fit right into the workflow of Java developers the IntelliJ plugin was made.

Running the analysis as part of the IntelliJ plugin is very simple. The programmer is expected to perform the following steps:

1. Open IntelliJ with your project.
2. Load both versions of your code. One before and after the change you want to analyse.
3. Create a Java class for specification class. It's format is explained in section 5.2. So far the project structure should look similarly to the one in fig. 7.
4. Create a run configuration. How to do that is explained in section 4.1.1.
5. Run the analysis in the same way as Java programs or JUnit tests are run in IntelliJ.

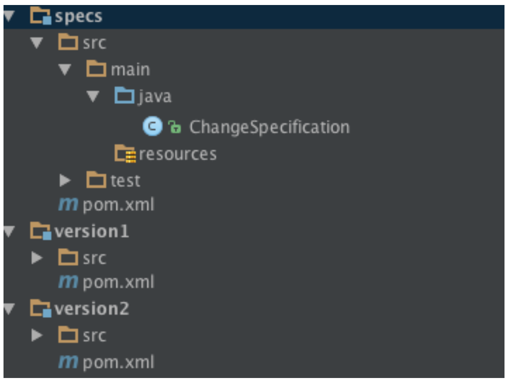

```
▼ specs
  ▼ src
    ▼ main
      ▼ java
          ChangeSpecification
        resources
    ▶ test
    pom.xml
▼ version1
  ▶ src
    pom.xml
▼ version2
  ▶ src
    pom.xml
```

Figure 7: Example IntelliJ project setup required to run behaviour difference analysis.

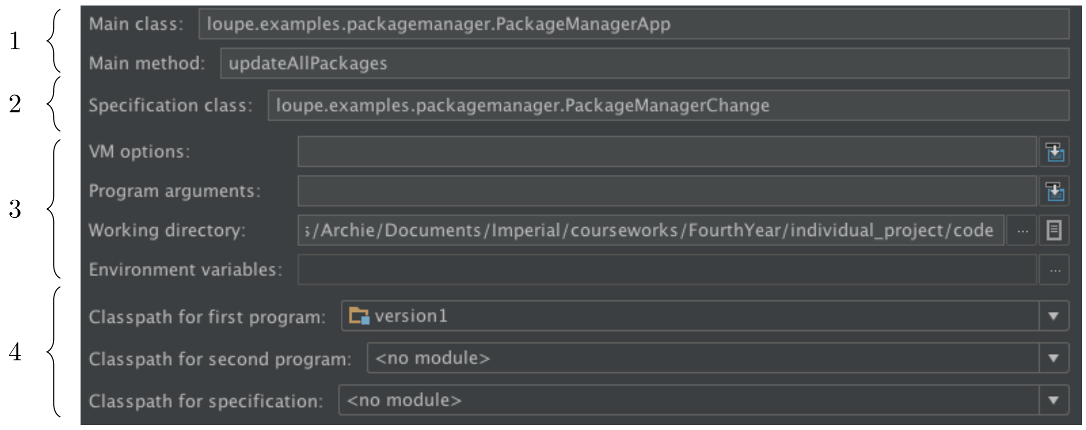
4.1.1 Running the loupe tool in IntelliJ

Any program that a programmer wants to run needs to be configured. In a terminal this is done by passing command line arguments. In an IDE an analogous operation is done by creating a run configuration. Run configurations encapsulate all information necessary to run a program. They contain type of program to be run (for instance Java application or a JUnit test) as well as the parameters required to run it. This allows IDEs to run the program without any further set up. Hence, it is possible to start running or debugging a program with a single keyboard shortcut.

To make it convenient to run the `loupe` analysis the IntelliJ plugin provides its own run configuration[6]. The configuration has a `Differential Run` type and contains all information necessary to run the analysis.

A necessary prerequisite required to run the equivalence analysis is that an IntelliJ project must contain **both versions** of the program that needs to be analysed.

The run configuration is created by opening the configuration editor, creating a new `Differential Run` configuration and specifying the analysis options. Opening the configuration editor can be done by selecting the `Run > Edit configurations...` menu option. Creating a new `Differential Run` configuration can be done by clicking on the `+` button and selecting the `Differential Run` from options. This creates a fresh configuration for analysis which needs to be configured. The configuration options are explained below and an example of a configured configuration is shown in fig. 8.



Main class: loupe.examples.packagemanager.PackageManagerApp
Main method: updateAllPackages
Specification class: loupe.examples.packagemanager.PackageManagerChange
VM options:
Program arguments:
Working directory: s/Archie/Documents/Imperial/courseworks/FourthYear/individual_project/code
Environment variables:
Classpath for first program: version1
Classpath for second program: <no module>
Classpath for specification: <no module>

Figure 8: An example run configuration.

1. **main class and main method** – these fields define the entry point to the program. Typical programs execute `public static void main(String[] args)` method and do not require a main method field. However `loupe` can execute any static method as an entry point. The only requirement is that the main class should only have a single method with that name.
2. **bds class** – this field should include a full name of a class that defines the partitioning. `loupe` will try to find a scenario for both programs that violates this specification. The specification format is explained in greater detail in section 5.2.

[6]The configuration is simplified by using project settings provided by IntelliJ. For instance `loupe` extracts project's classpath and module information.

3. **vm options/program arguments/working directory/environment variables** – these are standard settings that configure the running program.

4. **modules for both programs** – this fields should be set with the modules for the first and second version as well as the specification. These are used in order to determine the exact classpath that should be used in order to run each version.

4.1.2 Running the CLI tool

To run it as a CLI program simply invoke the `java loupe.inspector.Inspector` command. The application accepts the options described below. What is worth noting is that these options correspond directly to the configuration options required by the IntelliJ plugin.

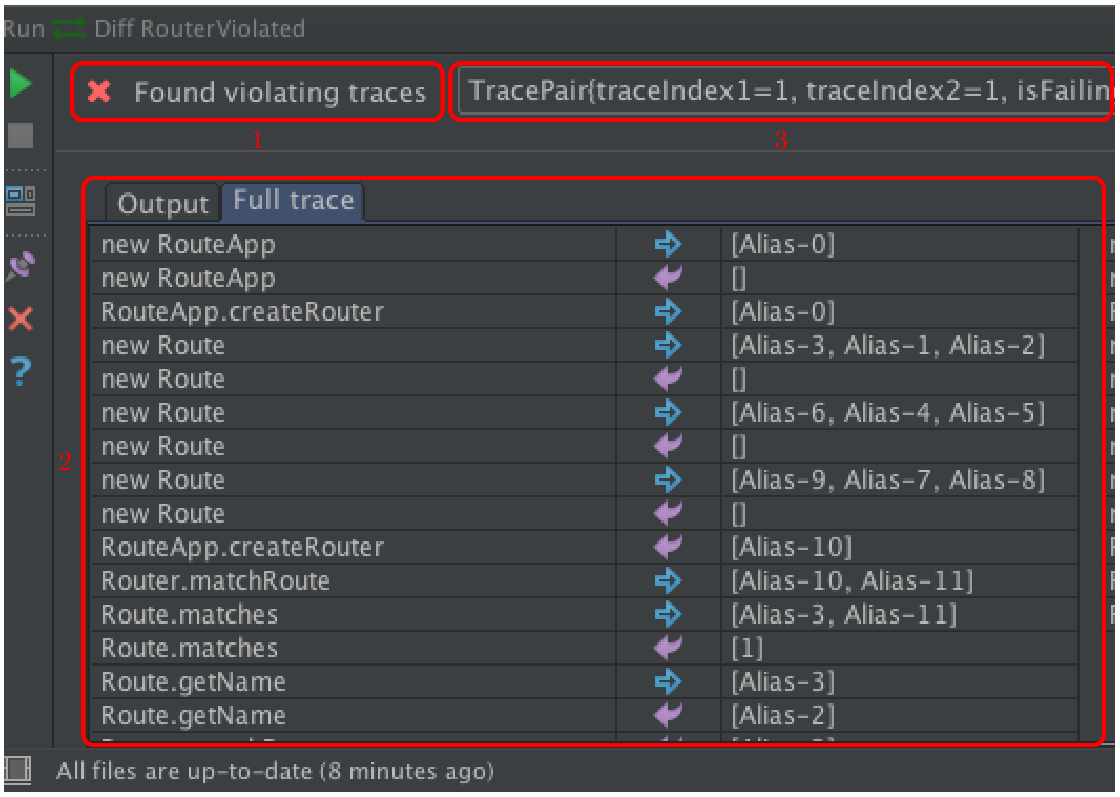
```
--className         Name of the main class to execute.
--mainMethod        Name of the main method to execute.
--methodSignature   Signature of the method to execute.
--specification     Class name of change specification.
--cpSpecification   Classpath of the change specification.
--cpV1              Classpath of the first program version.
--cpV2              Classpath of the second program version.
--help
```

4.1.3 Processing loupe IntelliJ plugin output

Once the analysis completes the IntelliJ plugin will display the output. An example output is shown in fig. 9.

The UI consists of the following areas:

1. Summary – shows whether or not any violations were found.
2. Traces – whenever a specification violation is found it might be useful to see the method call that caused a violation.
3. Trace selection – in case many trace pairs were found the selection box allows to navigate between them. However since the analysis terminates as soon as it finds a bug not all correct trace pairs might be explored.



Found violating traces

TracePair{traceIndex1=1, traceIndex2=1, isFailin

```
new RouteApp            →   [Alias-0]
new RouteApp            ↩   []
RouteApp.createRouter   →   [Alias-0]
new Route               →   [Alias-3, Alias-1, Alias-2]
new Route               ↩   []
new Route               →   [Alias-6, Alias-4, Alias-5]
new Route               ↩   []
new Route               →   [Alias-9, Alias-7, Alias-8]
new Route               ↩   []
RouteApp.createRouter   ↩   [Alias-10]
Router.matchRoute       →   [Alias-10, Alias-11]
Route.matches           →   [Alias-3, Alias-11]
Route.matches           ↩   [1]
Route.getName           →   [Alias-3]
Route.getName           ↩   [Alias-2]
```

All files are up-to-date (8 minutes ago)

Figure 9: Screen shot of the 'loupe' application.

5 Loupe architecture

5.1 Architecture overview

Loupe's design is split into multiple stages to make the process modular. To prepare for the analysis it performs a sequence of transformations and analyses on the source code. Loupe makes it easy to add or modify the existing filters in order to tweak its behaviour. For example it is possible to disable abstraction in order to see its effect on the performance. All stages are enabled by default and are highlighted in fig. 10. These stages are:

Partition building (section 5.2) – the first stage of the application converts the bds provided in a user format into object classifiers that are actually used by SVM.

Abstraction generation (section 6) – the second stage is the main concern of the project. It is responsible for automatic generation of abstractions of concrete classes. Abstraction generation makes use of the Soot framework in order to generate and process control flow of method bodies. The stage is expected to analyse these method bodies in order to generate accurate summaries. The resulting summaries are then used to generate Java code for the abstractions which are used in their compiled form by SVM.

Equivalence analysis – at this stage loupe prepares the SVM symbolic engine to run both versions of the program and check for any partition violations. This is done by doing the following steps:

- load both versions of the program into `SVM`.
- load all of abstraction classes. Instrument `SVM` classloader to use abstractions instead of the original classes (section 6.1.4).
- load the partition classifier so that `SVM` will generate the relevant trace.
- set the maximum time for which `SVM` should be running. This is necessary since most programs have infinite paths and so will never terminate.

After this is done, the `loupe` starts the `SVM`. It then waits until `SVM` times out or finds a violation.

Output processing – at this stage the result of the analysis is printed to the standard output. Since the result of the application is mainly used by other plugins the output of the application is generated in a machine readable format. To do so the results from `SVM` are converted into a data structure that is later serialized into JSON using Jackson serialization library[7].

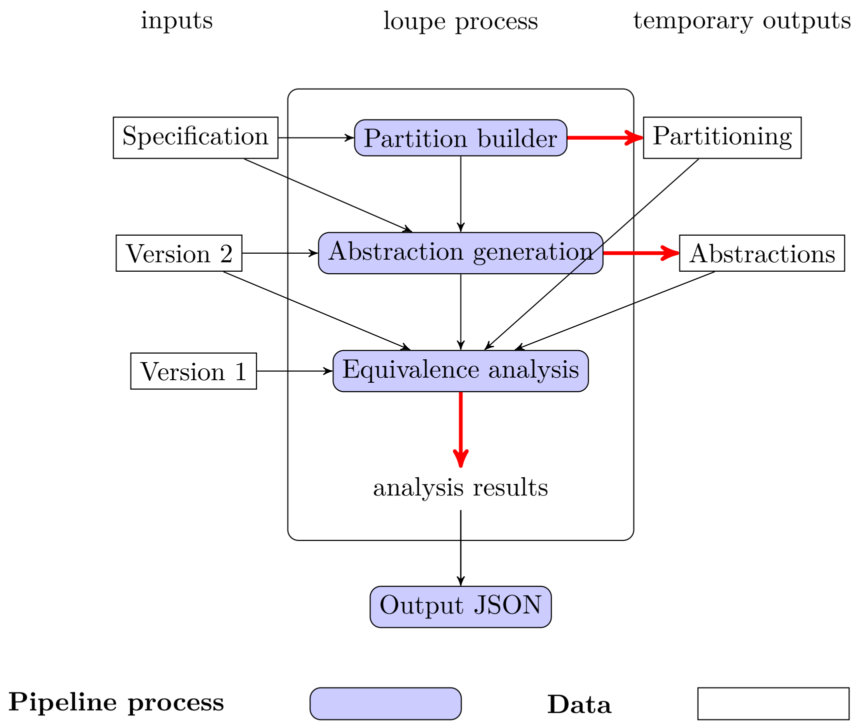


Figure 10: Overview of 'Loupe' architecture. Since behaviour of classes to be abstracted is equal only version 2 is required by abstraction generation.

[7]http://jackson.codehaus.org/

5.2 Specification

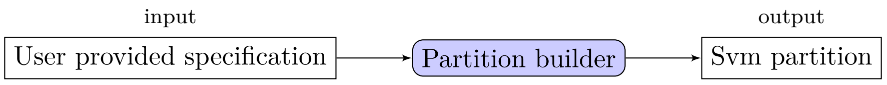


Figure 11: High level overview of partition builder.

When adding code behaviours programmers want to ensure that the program behaves correctly. As stated in the background section 2.1.2 whenever code is changed some behaviours are preserved, some changes are expected and some are unexpected. It's up to the programmer to decide which changed behaviours are expected and which ones are not.

When making unit tests a programmer can test the behaviour of the new version by making expectations about program behaviour in form of test suites[8]. A programmer can write multiple scenarios and add assertions he believes that the code should pass.

In `loupe` behaviour difference specification (bds) serves the same purpose for equivalence analysis as the test suite as unit testing. A programmer may write the assertions made on the behaviour difference. By doing so he can specify what behaviour changes he expects to happen. An unexpected behaviour is discovered if the specification is violated for some program scenario.

This section explains the format of a user provided specification.

5.2.1 Creating and using a behaviour difference specification

The bds's design mirrors that one of the JUnit tests. Each specification is defined in a single Java class file. The class needs to implement the `ChangeSpecification` interface. In addition it must provide the expected classification of objects on the heap, i.e. which objects are expected to be affected and which ones are expected to be unaffected.

In order to find any specification violations it is only necessary to check the behaviour of objects at the boundary between affected and unaffected objects. Therefore the main intention of the `ChangeSpecification` class is to form that boundary.

To add a classifier that tags an object on the partition boundary as either affected or unaffected it is necessary to write a method that returns a predicate. In order to distinguish between unaffected and affected predicates the unaffected predicate should be annotated with `@BehaviourUnchanged` while the affected predicate should be annotated with `@BehaviourChanged`. An advantage of this approach is that reclassifying an object requires just an annotation change.

An example use of predicates is shown in listing 20. For any constructed object `o` if `packageManagerChanged().match(o)` returns true then the object is tagged as affected and when `upToDatePackagesShouldNotChange.match(o)` returns true then the object is tagged as unaffected.

By default `SVM` tags all objects created by an affected object as affected and all unaffected objects created by an unaffected object as unaffected. With such a default behaviour only objects on the partition need to be classified. This makes specifications much smaller and easier to write.

[8]In Java a commonly used library is JUnit. Tests are created by making methods that assert behaviour of the code. The test methods are then annotated with a `@Test` annotation

```java
public class Change implements ChangeSpecification {
    @BehaviourChanged
    public Matcher<? super CallContext> packageManagerChanged() {
        return receiver(klassIn(PackageManager.class));
    }

    @BehaviourUnchanged
    public Matcher<? super CallContext> upToDatePackagesShouldNotChange() {
        return allOf(
                receiver(klassIn(Package.class)),
                anyOf(
                        calleeParameter(2, value(0)),
                        calleeParameter(3, value(1))
                )
        );
    }
}
```

Listing 20: Example change specification. It expresses a classification where every PackageManager instance is affected. Moreover every instance of Package which gets passed 0 as the second parameter or 1 as the third parameter is unaffected.

Predicates are represented as Hamcrest matchers[9]. `SVM` provides the following predicates:

1. `receiver(Matcher<Receiver> matcher)` – this predicate returns true if the receiver of the method call matcher the `matcher` predicate.
2. `klassIn(String... classes)` – with this matcher we can classify certain classes. This is a commonly used matcher as usually we want to limit the impact of our code change to a certain class(es). this predicate returns true if the receiver's class is one of the `classes`.
3. `calleeParameter(int index, Matcher<Value> matcher)` – with this matcher we can differentiate affected from unaffected objects based on constructor parameters. This predicate returns true if the `index` parameter of the method (where 0 is the index of the receiver) matches a `matcher` predicate.
4. `value(Matcher<? super Object> matcher)` – with this matcher we match against an actual value on the heap according to the `matcher` predicate.

Figure 12 shows a possible heap classification, based on a program run from appendix A.1.

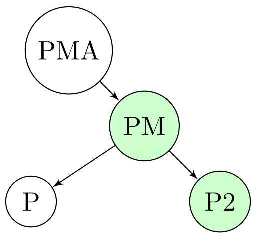


Figure 12: Example heap. Package Manager (PM) and Package (P2) are tagged as affected objects. PMA stands for PackageManagerApp. PM stands for PackageManager. P stands for Package.

[9]The wiki page contains the list of common matchers including `anyOf` and `allOf` http://hamcrest.org/JavaHamcrest/javadoc/1.3/org/hamcrest/core/package-summary.html

6 Generating abstractions

Automatic generation of abstractions is necessary in order to make the tool more usable for the programmer. Manually written abstractions are known as models and have been often used in order to model the effect of IO method calls [7]. The main problem with such an approach is that for every new model an abstraction needs to be written manually. Another problem is that the models mock the non abstract implementation precisely. In effect the implementations might turn out to be just as complex as the non abstract counterparts.

This section explains the automatic abstraction generation algorithm. The overview of the algorithm is shown in fig. 13. From a high level the abstraction generation takes the bytecode of classes to abstract and produces bytecode with their abstractions that can be run by `SVM`.

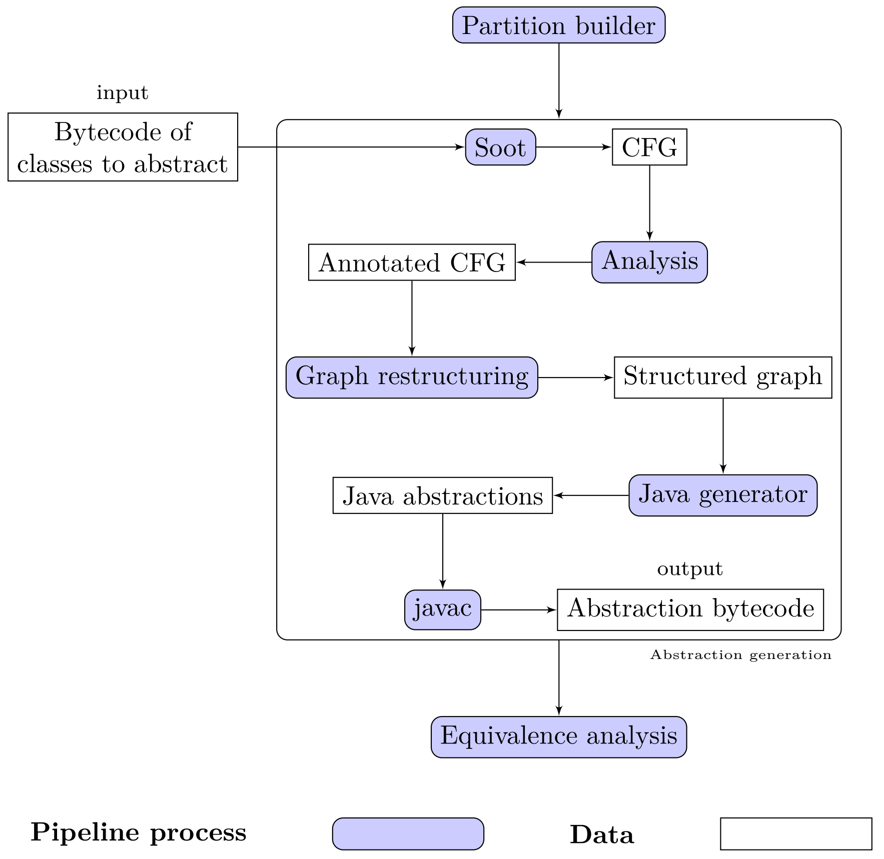


Figure 13: Overview of the abstraction generation pipeline.

To run abstractions under `SVM` they need to be generated in the form of bytecode. Abstracting behaviours section (section 6.1) specifies the output necessary in order to generate bytecode capable of expressing abstract behaviours. The section goes into detail how abstractions are implemented in `SVM`. The most noteworthy section is section 6.1.5. It discusses the implementation of abstract APIs which has **the largest impact on the performance of svm**.

In order to produce a valid Java output the original bytecode must go through multiple stages. The remaining sections describe these stages in more detail.

The first stage of the algorithm transforms the bytecode using the Soot library into a CFG. A CFG is a more general way to reason about code and makes it easy to apply several analyses over the code.

The second stage of the algorithm annotates the CFG with information that is not present directly in the source code as discussed in section 2.1.6. In particular all variables in the CFG are annotated using points-to analysis. Each variable is tagged with all places where objects they get assigned are constructed. The annotations make it possible to over approximate the code. Section 6.2 discusses code analyses that are incorporated into `loupe` in order to improve the accuracy of abstractions.

At this point the tool produces an annotated CFG. However, the CFG must still be converted into Java bytecode. This meant either directly generating bytecode or generating Java code and then compiling it. `Loupe` does the latter. It is difficult to generate valid Java code from the CFG. This is because CFG contains unstructured `GOTO` jumps while Java requires structures statements such as if/then/else, switch or branch statements. In order to convert `GOTO` statements to their structured equivalent `loupe` applies a restructuring transformation explained in section 6.3. By doing so it creates a structured graph where if/then/else, switch and loop statements are expressed as graph nodes.

The final stage takes the structured CFG and generates valid Java code. How it is done is explained in section 6.3.5.

6.1 Abstracting behaviours

Under symbolic execution the program is being explored for all of the possible behaviours. Thus, given enough time, symbolic execution will try to explore all paths of every method body. Abstractions are classes that approximate method definitions provided by the non abstract versions.

Non abstract implementations have different behaviours on inputs they receive. However because abstract classes do not hold onto the concrete state they need to choose between different behaviours non-deterministically. Sections 6.1.1 and 6.1.2 discuss an API designed in order to write method definitions that perform non-deterministic choices.

Section 6.1.3 discusses how abstractions can be defined in Java using the abstractions API.

Section 6.1.5 discusses how abstraction API was implemented in `SVM`.

6.1.1 Retained behaviours

Abstractions are over-approximations of concrete classes. In many cases an unexpected behaviour might only occur when the non abstract class exhibits a certain behaviour. To do that abstractions need to be able to perform the following actions:

1. Create objects – It is possible that the abstraction needs to create new objects on the heap. For example factories create new instances of object.
2. Call methods – Some classes invoke methods on other objects which can potentially mutate their state. While in some cases like calling IO methods this can be replaced by a no op, sometimes bds violations occur only if the class makes a relevant method call.

3. Perform non-deterministic actions – Some actions can be performed deterministically only given knowledge of the state. Take for example the following method call `new Scanner(System.in).nextLong()`. Its return value is dependent on user input. Therefore, the `next` method should non deterministically return a symbolic `long` or throw an `InputMismatchException`.

These actions were chosen as they are akin to the concept of the Most General Context (MGC) introduced by Welsch et al [11] (explained in background section 2.1.10). The only interaction missing in this project are field writes. Nevertheless Java promotes encapsulation where all fields are kept private. Since an abstraction does not contain any state, it does have to set any fields either.

The first two of these actions are common to concrete implementations. To make an abstraction which fulfils these actions it is only necessary to make it capable of making the same method calls. We later call this *mirroring* of method calls. However behaviour that cannot be expressed by concrete objects is a non-deterministic choice for which the symbolic execution will take many paths. For example when `Math.random()` is called the symbolic execution will return **one** random result and finish the execution. However what an abstraction needs is to be able to simulate **all** possible results, i.e. return a symbolic double in the range $[0, 1]$.

This creates a requirement for an API supporting non-deterministic choices. Programs running under JVM cannot have non-deterministic actions for which all behaviours are explored. Therefore, abstractions cannot run outside of a symbolic execution engine.

6.1.2 Abstraction API

The abstraction API contains functions that allow the program to perform non-deterministic choices. The main considerations when designing the API was to make the abstract version fast and readable. Therefore the API exposes a "standard library" for expressing non-deterministic behaviours.

The `SymbolFactory` class contains the following methods:

1. `newIntSymbol` and `newBooleanSymbol`

   these methods create new symbolic values. In cas the concrete integer value of the object is not known it can always be approximated by a symbol which can have take any integer value. For instance we can over-approximate `Math.random() > 0.5` by exploring the case when it is true and when it is false.

```
boolean booleanSymbolExample() {
  return Math.random() > 0.5;
}
```

```
boolean booleanSymbolExample() {
  return newBooleanSymbol();
}
```

2. `randomChoice` and `selectState`

   sometimes the code requires us to take a branch. For example the code block `if (*) { A(); } else { B(); }` will either call `A()` or `B()`. `randomChoice` allows us to fork the state of the program in which the first state will take the then branch and the other will take the else branch. `selectState` generalises this approach to n different branches.

```java
void main(String[] args) {
  if (args.lenght == 0) {
    System.out.println("Args!");
  } else {
    System.out.println("None!");
  }
}
```

```java
void main(String[] args) {
  if (randomChoice()) {
    System.out.println("Args!");
  } else {
    System.out.println("None!");
  }
}
```

```java
void main(String[] args) {
  switch (args.length) {
    case 0:
      System.exit(2);
      break;
    case 1:
      System.exit(1);
      break;
    default:
      break;
  }
}
```

```java
void main(String[] args) {
  int choice = selectState(3);
  if (choice == 0) {
    System.exit(2);
  } else if (choice == 1) {
    System.exit(1);
  } else {
  }
}
```

3. `getArgument`

   sometimes we cannot be sure of the argument a given variable might have. To return one of possibilities `getArgument` is called. For example running the abstract version of the `getArgumentExample` method below would result in two states. In the first state `getObject1().isInstalled()` would be called and in the second state `getObject2().isInstalled()` would be called.

```java
boolean getArgumentExample() {
    Package o = getObject1();
    if (o == null) {
        o = getObject2();
    }
    return o.isInstalled();
}
```

```java
boolean getArgumentExample() {
    Package o = getArgument(
      getObject1(),
      getObject2());
    return o.isInstalled();
}
```

4. `getPassedParameter`

   the `getArgument` method is useful when the instances which could be passed as arguments are known statically in advance. However this is not always the case. Take for example the code snippet below. Without knowing how the `HashMap` is implemented the best over-approximation of the `map.get(key)` call would return one of the values passed into the object. Thus whenever an `addPackage` method is called the set of possible return values of `getPackage` increases.

   Calling `getPassedParameter` returns any instance from a list. Therefore it allows objects to keep track and return **all** instances of a specific type when passed into a class.

```java
public class PackageManager {
  private HashMap<String, Integer> map;
  public void addPackage(String key, Integer value) {
    map.put(key, value);
  }
  public Integer getPackage(String key) {
    return map.get(key);
  }
}
```

```java
public class PackageManager {
  private ArrayList<String> strings = new ArrayList<>();
  private ArrayList<Integer> integers = new ArrayList<>();

  public void addPackage(String key, Integer value) {
    strings.add(key);
    integers.add(value);
  }

  public Integer getPackage(String key) {
    strings.add(key);
    return getPassedParameter(integers);
  }
}
```

6.1.3 Using the abstraction API to build abstractions

The abstraction API allows to define methods that have abstract behaviour. The API itself is written as a Java class akin to a Java interface. Because of this, any bytecode that calls the API will trigger non-deterministic behaviour. Hence, if need be (for instance `loupe` cannot generate an abstraction automatically) a programmer could write it by hand and use in `SVM`.

There are many ways in which the behaviours can be implemented. `Loupe` generates abstractions as Java code and calls the abstraction API. It could, however also compile the abstractions directly into bytecode or it could even create data structures which combine abstract class definitions with abstraction API implementation. The main reason why `loupe` does the former is that it allows programmers to look directly at the code of abstractions. It also makes abstractions less brittle to any svm changes.

6.1.4 Running abstractions inside of svm

Once a Java class with an abstraction is created it needs to be provided to `SVM` in order to execute its method bodies. In order to provide a seamless experience `loupe` ensures that no code changes are required in order to load the abstractions. In particular programmers are not required to replace the concrete implementation with an abstraction.

This seamless experience is possible as `loupe` dynamically modifies the `SVM` class loading behaviour. Normally during method calls or new instructions a class definition needs to be loaded. Whenever that happens `SVM` calls the `loadKlassFor(className)` method in order to find the relevant class definition. This makes a class loader fetch the .class file from a disk and process class's bytecode.

When abstractions are enabled before `SVM` starts `loupe` loads all abstraction .class files and creates a map with abstraction definitions. `Loupe` then adds an *abstraction instrumentation* and wraps each bytecode operation into an abstraction operation. For non abstract classes the abstraction operation proxies the `loadKlassFor` method to the default classloader. For abstract method abstraction operation returns the initially loaded abstraction definitions instead.

6.1.5 Implementing abstraction API

Since abstractions are written in Java they behave just like any other classes. When one of their methods is called the object class is loaded and the called method is fetched executing instruction one by one.

They behave non deterministically by calling the abstraction APIs. These APIs expose `static native` methods, methods which need to be implemented by the VM that interprets the code. Thus whenever a native method is called `SVM` looks up its own list of definitions for native methods and executes custom code.

The project has added to `SVM` a method definition for each abstract API method. The way in which the method affect the program state are explained below:

1. `newIntSymbol` and `newBooleanSymbol`

Both of these instructions push a new `Symbol` to the stack. However, in order to be synchronised, equivalent symbols need to be named in the same way across versions. To do this both of these methods keep count of the number of created symbols and name each symbol by the counter.

2. `randomChoice()` and `selectState(n)`

This is the most performance critical of the non deterministic instructions. Using this instruction makes abstraction classes outperform non abstract classes.

`randomChoice()` is equivalent to the call `selectState(2) == 0`. Both of these instructions fork the execution state. To each state they push one of the return values. A `randomChoice()` instruction will create two states, with `false` and `true` values pushed to the states respectively. `selectState(n)` creates n states with values from `0` to `n - 1` pushed to the states. What the `selectState()` method must ensure is that only states that made the same choice are compared.

An initial implementation of the `selectState()` method added disjoint path constraints to each of the forked states. This ensured that only paths with the same selection would be compared. At the same time it made abstractions really slow as `SVM` had compare all states pairwise to check which ones made the same choice. This required $O(n^2)$ SMT queries where n is the number of states.

The improved implementation of `selectState()` improves the speed by removing SMT queries entirely. Furthermore exploration of abstractions became faster compared to non abstract versions due to the fact that SMT queries take substantial time during regular symbolic execution.

In order to avoid any SMT queries the implementation indexes each selection. The abstraction will have the same behaviour if the index is equal. `selectState()` groups states based on this index and ensures that paths are only compared if they belong to the same group. Because of that extra constraints no longer need to be added to different paths.

To see why this method outperforms the non abstract execution look below at the comparison of a code that uses a `newBooleanSymbol()` vs code that uses `selectState` method. Both versions end up in two states, one calling `doA()` and another calling `doB()`. However, the first version forks at line 3 during the if statement. To do this is needs to verify the feasibility of path constraints `s == false` and `s == true` making expensive calls to the SMT solver. On the other hand version that uses `selectState(2)` method will fork at line 2 on the `selectState(2)` method. Since the value of `s` will be equal to a concrete integer `0` or `1` the branch condition will evaluate to `true` or `false` respectively. As a result both lines 4 and 6 are reached without making an SMT query.

```java
public void main() {
  boolean s = newBooleanSymbol();
  if (s) {
    doA();
  } else {
    doB();
  }
}
```

```java
public void main() {
  int s = selectState(2);
  if (s == 0) {
    doA();
  } else {
    doB();
  }
}
```

3. `getArgument(argsList)`

This instruction is a syntax sugar operation that makes it easy to pick one element from a list of objects. It is implemented by selecting an arbitrary array index by calling `selectState(argsList.length)`. Then to each of the states it pushes a list element.

4. `getPassedParameter(list)`

This instruction is implemented exactly like the `getArgument` list except that it works for lists instead of arrays.

6.2 Improving abstraction precision by code analysis

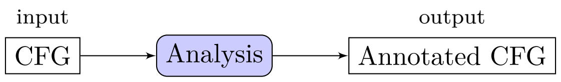


Figure 14: High level overview of the code analysis stage.

Using the abstraction API makes it possible to simply write Java code by hand that replaces concrete implementations of classes with their abstract counterparts. Take for example the `Safe` class defined below.

```java
class Alarm {
  public void ring() {
    System.out.println("Intruder␣tries␣to␣steal␣your␣money.");
  }
}
class Safe {
    private ArrayList<Integer> cash = new ArrayList<>();
    private Alarm alarm;
    private int pin;

    public Safe(int coins, int pin, Alarm alarm) {
      this.alarm = alarm;
      this.pin = pin;
      addCoins(coins);
    }

    public void addCoins(int coins) {
      this.cash.add(coins);
    }

    public int getCash(int code) {
      if (code != pin) {
        alarm.ring();
        return 0;
      }

      int total = 0;
      for (int i = 0; i < cash.length; i++) {
        total += cash[i];
      }
      return total;
    }
}
```

Without the automatic `loupe` abstraction generation a programmer could write the following abstraction:

```
class Alarm {
  public void ring() {
    System.out.println("Intruder␣tries␣to␣steal␣your␣money.");
  }
}
class Safe {
    public Safe(int coins, Alarm alarm) { }
    public void addCoins(int coins) { }
    public int getCash(int code) { return SymbolFactory.newIntSymbol(); }
}
```

While the abstracted version simplifies the complexity of the `getCash` method and would make the tests run faster it is very imprecise. In particular with this abstraction the alarm is never rung. This means that some partition violations that would occur when an alarm is rung would not be found.

The problem with the naïve implementation is that it does not over-approximate the behaviours. In this example it does not capture the `alarm.ring()` call.

This section explains how this project infers information about the method code in order to generate abstractions that capture the behaviours and are capable of making the same method calls. For the rest of the section the process of making the abstraction make equivalent method calls is called mirroring. In order to achieve this method body is analysed. The following section summarises strategies that are executed in order to make abstractions closely resemble their concrete counterpart.

Section 6.2.1 discusses the first strategy which creates a generic abstract state for all abstractions. In case of the `Safe` example this means ensuring that the `alarm` parameter passed in the constructor will be available in the `getCash` method without the presence of an `alarm` field.

Section 6.2.2 discusses the second strategy which mirrors the method calls and `new` instructions. In particular it discusses different options in which `new` could be mirrored and the trade-off between accuracy and complexity it creates.

Section 6.2.3 discusses the third strategy which uses points-to analysis. This analysis is used to ensure that as methods are mirrored their parameters are also accurately over approximated. For instance in the example above the `Safe` constructor calls the `addCoins(coins)` method. Points-to analysis will ensure that points-to variable can only refer to the first parameter of the method call.

Section 6.2.4 discusses the fourth strategy which mirrors the control flow of the method body. This ensures that the method calls executed by a method body will reflect the ones that could feasibly be executed.

6.2.1 Dealing with parameters

In object oriented programming class instances contain both data (instance state) and code. As parameters are passed into a class method they are often stored in a field or a container (for instance a `HashMap`) for future use. Look at the example below. When the `main` method is run it constructs the configuration. `Config` uses the `configuration` field to store the mapping from configuration names to configuration values.

When abstractions are created they no longer use concrete fields. However, abstractions still need to access parameters that might have been stored for future use. For example the `Config.getConfig` method is called the abstraction should be able to return one of the passed values by the `Config.setConfig` method.

```java
public class Config {
  private Map<String, String> configuration = new HashMap<>();
  public Config(String[] args) {
    for (int i = 1; i < args.length - 1; i+= 2) {
      c.setConfig(args[i], args[i+1]);
    }
  }

  public void setConfig(String key, String value) {
    configuration.put(key, value);
  }
  public String getConfig(String key) {
    return configuration.get(key);
  }
}

public class Program {
  public static void main(String[] args) {
    Config c = new Config(args);
    System.out.println(c.getConfig("verbose"));
  }
}
```

We can store all parameters entering a class by using a `getPassedParameter(list)` method. In order to ensure that **all** possible instances of a class instance can be returned the `list` variable must be populated with all instances entering a class. Hence in the example above the `Config` class would be abstracted by the code below.

```java
public class Config {
  private ArrayList<String[]> strings = new ArrayList<>();

  public Config(String[] args) {
    strings.add(args);
    for (randomChoice()) {
      c.setConfig(getArgument(args), getArgument(args));
    }
  }

  public void setConfig(String key, String value) {
    strings.add(key);
    strings.add(value);
  }
  public void getConfig(String key) {
    strings.add(key);
    return getPassedParameter(strings);
  }
}
```

This approximation is very imprecise as it allows many more parameters to be returned than possible. Nevertheless, it ensures that the abstraction over approximates the original class behaviour. It is possible to make more complex analyses which specialise some cases and deal with them accordingly. For instance in case of getters and setters it might be beneficial to keep the field instead of the abstraction state.

6.2.2 Mirroring method calls and dealing with new

Besides reusing the already passed in parameters non abstract objects often create new objects. However this means that abstractions also need to mirror object construction. Take a look at the `ClassRoom` class below. The `addGrade` method adds a grade to a student. However if a `Student` for a given name did not exist then a new `Student` is created. In addition the class creates a new `HashMap` on initialisation. Thus those two constructions need to be mirrored in order to accurately abstract away the behaviour.

```java
public class Student {
  private List<Integer> grades = new ArrayList<>();
  public void addGrade(int grade) {
    grades.add(grade);
  }

  public int getGPA() {
    if (grades.length == 0) {
      return 0;
    }

    int total = 0;
    for (int grade: grades) {
      total += grade;
    }
    return total / grades.length;
  }
}

public class ClassRoom {
  private Map<String, Student> students = new HashMap<>();

  public Iterable<String> getStudentNames() {
    return students.keySet();
  }

  public void addStudent(String name, Student student) {
    students.put(name, student);
  }

  public void addGrade(String name, int grade) {
    Student student = students.get(name);
    if (student == null) {
      student = new Student();
```

```
        addStudent(name, student);
      }

      student.addGrade(grade);
    }
}
```

Listing 21: Students grading system example.

To deal with `new` method calls we have to decide whether to treat the objects abstractly or not.

If the object is not treated abstractly then the `new` method call is exactly mirrored. This creates a requirement for all method calls to be mirrored as well.

If the object is treated abstractly the `new` method call instead of creating a concrete object should create an abstraction instead. However in some cases the method calls do not need to be mirrored. This is especially true in case of container classes such as `HashMap`, `Set` or `List`. In this it is not necessary to create an actual `HashMap`. Instead it can become lazy and when any methods are called on a `HashMap` it can return an abstract value corresponding to the type.

In the example above (listing 21) the `getStudentNames` method returns `students.keySet()`. If the method would be mirrored then the `HashMap` instance would need to be actually created. However if the methods are not mirrored then the `students.keySet()` call is allowed to return an abstract instance of `Set<String>` instead.

6.2.3 Points-to analysis

Concrete implementation of a class can call methods that have parameters. While in a concrete implementation the parameter can have a single source this is no longer the case in an abstraction. In listing 21 in order to create an accurate abstraction we need to reflect the method calls. In case of `addStudent` method the abstraction would have to call `HashMap.put` method. In case of `addGrade` method the abstraction would have to call `HashMap.get`, `ClassRoom.addStudent` and `Student.addGrade` methods.

What points-to analysis provides are all possible locations where a variable was defined. The points-to analysis in this project is implemented as a forwards analysis that propagating the Map from variables to their assignments. For each assignment a new mapping is added to the map. The analysis returns a mapping from each local variable to the constants, fields, parameters and complex expressions that could define it.

Other points-to analysis techniques exist as well. They all provide a trade-off between performance and accuracy. The complexity of local points-to analysis used in this project depends only on the size of code to be abstracted. On the other hand the complexity of a full program control flow analysis provided by `Soot` depends on the size of the entire program.

The points-to analysis result is used in order to decide the closest overapproximation for a method call parameter. In order to decide which approximation should be chosen the following heuristics are used:

- In case if the parameter is concrete or has been passed around as a parameter of the callee method then this parameter is used. For instance abstraction of the `grade` variable in `student.addGrade(grade)` is the `grade` variable.

- In case if the parameter is passed from a variable then all of the instances it could be passed around are determined. For instance abstraction of the `student` receiver in the `student.addGrade(grade)` method can either be a student found in a `HashMap` or a newly created student. Thus we could approximate the `student.addGrade(grade)` call with the following code block.

```
public void addGrade(String name, int grade) {
  Student s1 = students.get(name);
  Student s2 = new Student();
  getArgument(s1, s2).addGrade(grade);
}
```

- In case if the parameter has a primitive type – the parameter is replaced by a symbol. For instance if we would abstract the statement `return total / grades.length` in `Student.getGPA()` method then it would be replaced by `return newIntSymbol()`.
- In case if the parameter is not a primitive and has a type `T` – it is replaced by any instance of `T` passed into the class. This can be done by calling `getPassedParameter(listOfTypeT)` method.

6.2.4 Keeping the control flow

Precision of abstractions can be highly improved by keeping the control flow. `Loupe` does this by making analysis annotations directly on top of the bytecode CFG. As a result an annotated CFG will have the same control flow.

Without keeping the control flow the abstract method definition would no longer precisely replicate the same sequences of method calls. A possible abstraction of the `addGrade` method in the ClassRoom example (listing 21) that completely disregards the control flow is shown below.

```
public void addGrade(String name, int grade) {
  Student student = null;
  while (randomChoice()) {
    switch (selectState(4)) {
      case 0:
        student = students.get()
        break;
      case 1:
        student = new Student();
        break;
      case 2:
        addStudent(name, students)
        break;
      default:
        student.addGrade(grade);
        break;
    }
  }
}
```

While this abstraction clearly has can execute any sequence of instruction that non abstract object. However, the non abstract class can only execute two method call sequences shown in fig. 15. On the other hand the abstraction has an infinite number of sequences of method calls.

```
students.get(name);
new Student();
addStudent(name, student);
student.addGrade(grade);
```

```
students.get(name);
student.addGrade(grade);
```

Figure 15: Two possible method call sequences that addGrade method can generate.

6.3 Generating Java code

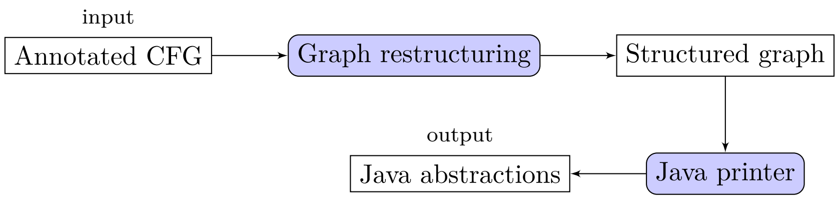


Figure 16: High level overview of code generation stages.

Abstractions are expected to be expressed as Java code. This is problematic since static analysis produces an annotated CFG. Thus, lower level bytecode needs to be restructured into valid Java code. For instance goto operations need to be converted into if/else, switch, loop statements.

The problem of reverse engineering the original code from compiled code is known as decompilation. Doing so is a challenge on its own however `loupe` only needs to perform the restructuring phase of decompilation. In addition `loupe` does not need to extract branch and loop conditionals making the algorithm simpler.

A CFG contains arbitrary `GOTO` statements and many statements can be reached from multiple paths. This we cannot generate code by recursively traversing the graph. The result of restructuring creates nodes representing branch and loop statements. Thus, after restructuring the graph has a tree structure and can be converted into Java by recursive traversal.

Consider that we would try to generate abstractions for the `Router` example appendix A.2. The source code is shown by listing 22, which by this stage would be represented by the CFG[10] shown in fig. 17 and a dominator graph shown in fig. 18.

```
public void route(Request req, Response res) {
  for (int i = 0; i < routeCount; i++) {
    RouterItem item = items[i];
    if (req.getPath().equalToString(item.format)) {
      item.route.route(req, res);
      return;
    }
  }
  LOG.warning("Route␣not␣found␣%s", req.getPath());
}
```

Listing 22: Original `route` method taken from appendix A.2.

[10]Actually this is a simplification of the graph as some instructions would get compiled to multiple Java instructions.

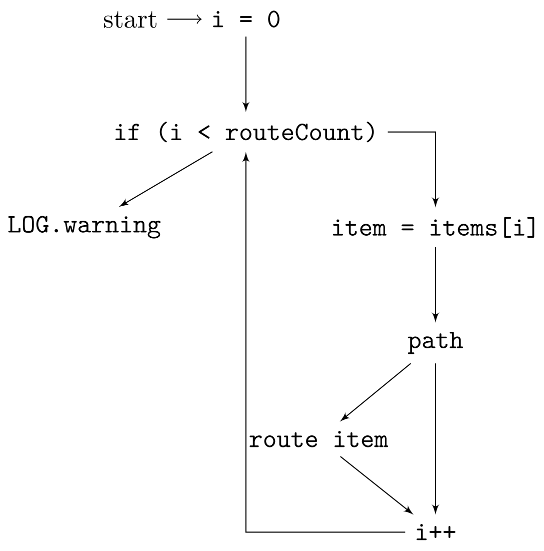


Figure 17: Control flow graph of listing 22.

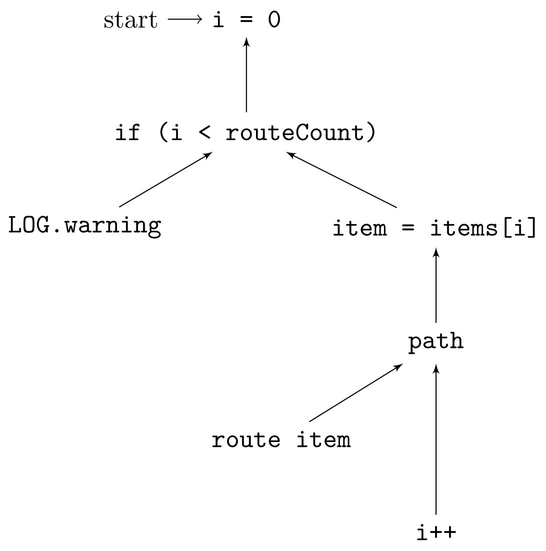


Figure 18: Dominators graph. An arrow from one node to another means that one node is dominated by the other node. For instance `i++` is dominated by `path`.

Algorithm that performs reconstruction of the code uses the unstructured graph (as the one in fig. 17) and a dominators tree (fig. 18). It produces a structured graph where the nodes represent the Java structure (such as fig. 19). The structured graph consists of the following node types:

- Branching nodes – these nodes represent any branching statements such as if/else statements or switch statements. Because an over-approximation is allowed to take any of the branches branch node does not contain the condition.
- Loop nodes – these nodes contain the loop instructions. Equivalent to `do while` statements.
- Exit nodes – these nodes are equivalent to `break` statements and allow jumping out of the loop.
- Block nodes – these nodes represent a sequential computation.
- Statement nodes – these nodes contain the actual statement being executed. Represented by a statement name. Empty statements represent noops.

With these node types the expected structured graph for the method would look like fig. 19. This closely resembles original method except that the `while` statement was changed into a `do while` statement. Nevertheless given the structured graph it is very easy to create the final Java code as explained in section 6.3.5.

6.3.1 Overview of the algorithm

The main idea behind the algorithm it to create the structure as shown in fig. 19.

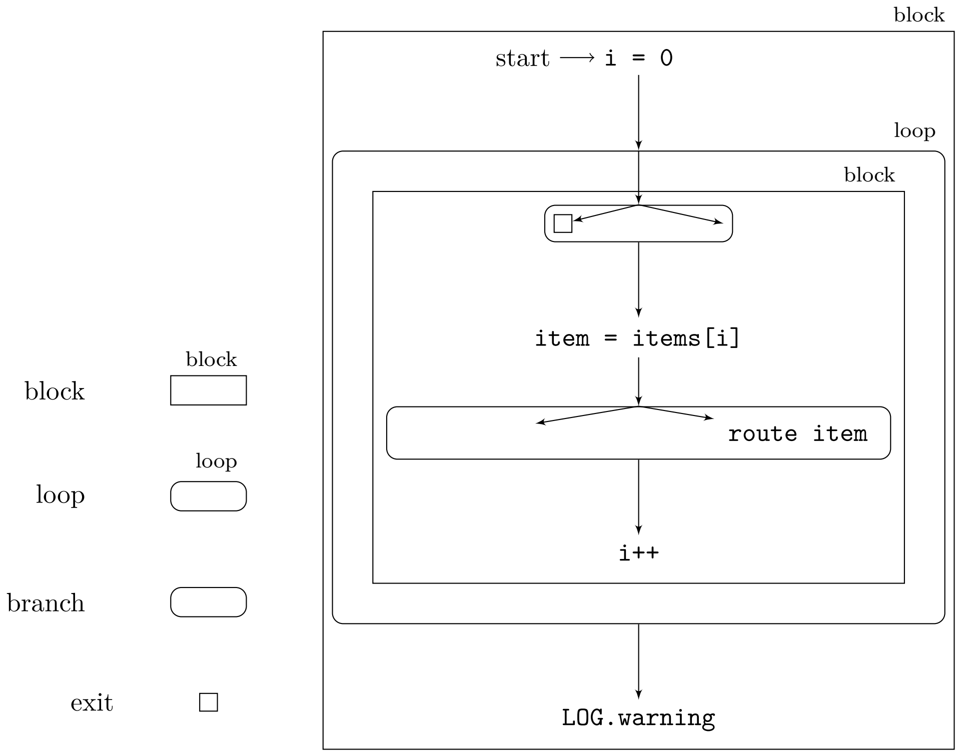


Figure 19: Structured control flow of the method.

When the algorithm is initialised it creates the initial structure of the graph. In this structure each statement is wrapped in a block where the successor of a statement is a branch node. The idea being it is that all instructions which should follow a given statement should become one of the branches of the branch node.

The initial structure for each node would look like fig. 20. However for larger graphs it becomes difficult to understand the graph. A branch node with three choices would be represented by a fig. 21. Because the choices themselves would also be made of a block a statement and a branch node. To make the graphs more readable in sections explaining branch and loop restructuring a simpler notation will be used as shown in fig. 22. The complex statements will be styled in bold.

The algorithm then proceeds to do a breadth first traversal of the control flow graph starting at program exits. Then for each node being visited restructuring algorithms are applied. The restructuring algorithms connect the node to its predecessor in a structured way.

The first step of the restructuring algorithm retrieves the dominator of the node and based on the dominator decides whether loop or branch structuring should be applied. If the dominator is a loop header or the dominator is a part of the loop while the process node isn't then loop structuring is used[11]. A loop header is a node which starts a loop and other statements will jump back to it. A loop exit is a node from which the loop can be exited.

In order to ensure that such loops do not cause the breadth first traversal to loop infinitely the

[11]The case when the node has a successor that leaves a loop is also called a loop exit. However in this report an exit node is a structured AST node type and has a class `ExitNode`.

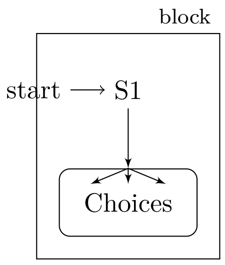


Figure 20: Structured graph after initialisation.

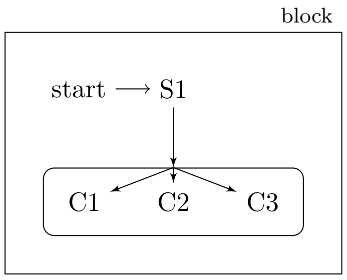


Figure 21: Simplified representation of a structured graph after initialisation.

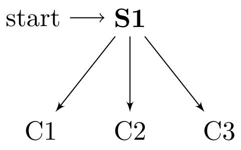


Figure 22: Simplified representation of a structured graph after initialisation.

algorithm keeps a set of all traversed nodes and visits each node just once.

6.3.2 Branch structuring

The main outcome of branch structuring is to convert the paths in the control flow graph into branches (`if (*) { S1 } else { S2 }`) and sequential execution (`S1 S2`).

Look at the simplest if/else statement shown in fig. 23. Intuitively we can see that the statement `path` can either branch to `route item` or can exit the branch. Thus the statements `route item` and a noop statement should be nested inside a branch node while `i++` should follow the branch node.

In the example in fig. 23 after initialisation the first item to process `i++`. In branch structuring processing is done by looking up this node's dominator. In this case the dominator is the `path` node. Then we create a branch node and connect all nodes together. To do this the branch node is added as one of dominators choices and process node follows the branch node. This is shown in fig. 24. Furthermore, because `i++` is also the successor of `path` a noop is added as one of the choices.

Then we add the predecessors of a node to the traversal queue. However the desire of the algorithm is to put **all paths** between the dominator and a process node into the branch node. Such a structure is shown in fig. 25. To do this the dominator graph is modified to add the branch node and make it dominate the initial instructions of branching paths. In our example this would mean making the branch node dominate `route item` instruction.

6.3.3 Loop structuring

`Loupe` uses the `LoopNestTree` class in order to perform loop detection which is part of the `Soot` library. From the `LoopNestTree` is retrieves an iterator of loops. In the `Router` example we have

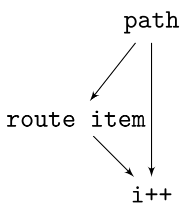


Figure 23: If/else unstructured flow.

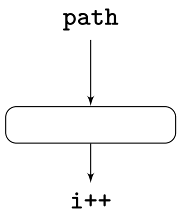


Figure 24: Partial restructuring of the branch by adding an branch path.

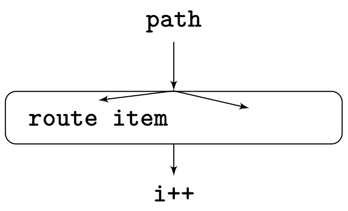


Figure 25: Completed restructuring by adding branch paths into the branch node.

to restructure the `for` loop. The `if` condition is both the loop header and a loop exit[12].

`Loupe` uses the `Loop` class instances in order to structure the loops accurately. These instances are able to return both the loop header and its exits. From this information `loupe` classifies the dominator.

If the processing node is not in the loop but the dominator is then two things happen. Firstly an exit node is added as one of dominator's choices. Secondly the process node is added as one of loop header's choices. This is illustrated with fig. 26.

If the processing node is a loop header then a loop node is attached as a choice of the dominator node. In addition the processing node is attached as a loop body of a loop node. This is illustrated with fig. 27.

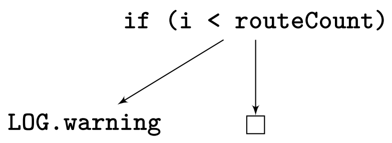


Figure 26: Loop structuring in case processing node exits the loop.

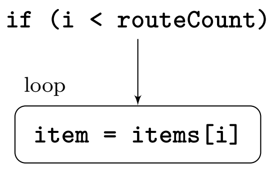
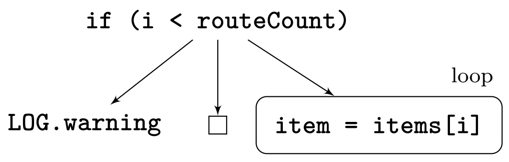


Figure 27: Loop structuring in case dominator is the loop header.



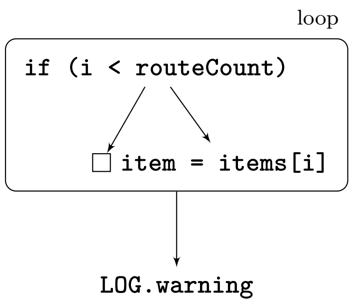


Figure 28: Loop structuring before (left) and after (right) of LoopVisitor execution.

Once the algorithm that performs the breadth first traversal a `LoopVisitor` is executed. The

[12] In bytecode the if condition would actually take multiple instructions. We use a single node to simplify the diagram.

`LoopVisitor` creates the final loop that encapsulates the loop header. The transformation is illustrated by fig. 28.

6.3.4 Simplifying structured graph

The structured graphs generated by the technique outlined above contains a lot of redundant nodes. The transformations in the previous section represented the graph using a simplified representation shown in fig. 22. For instance the actual representation of fig. 25 is shown in fig. 29.

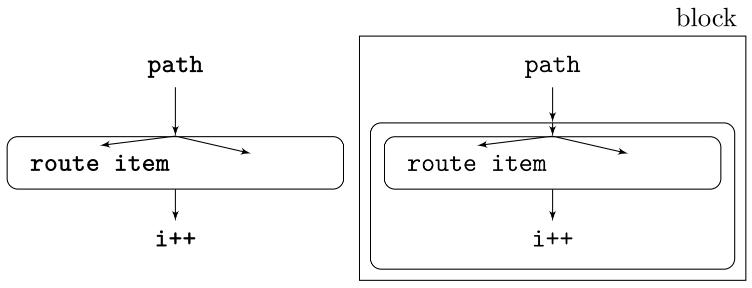


Figure 29: Simplified structured if/else node from fig. 25 (shown on the left) and actual representation (shown on the right).

The representation above allowed the code to be generalised. However to make handling of the structured graph easier for the remaining algorithms the graph is simplified using two steps.

First step of graph simplification creates a block structure. It ensures that the branch node and the following node are added to the same code block. For our example this will result in the inner branch node being inlined making `path`, branch and `i++` nodes in the same block.

Second step of graph simplification removes redundant nodes. It performs the following transformations.

1. Removes no ops from code blocks.
2. Replaces a code block with a single statement or a branch with a single choice statement by that statement.
3. All statements in the nested code block are added to the parent code block.
4. Removes any empty choices from branches.

After these simplifications the graph has a very compact structure making it very easy to process it further.

6.3.5 Generating Java from structured graph

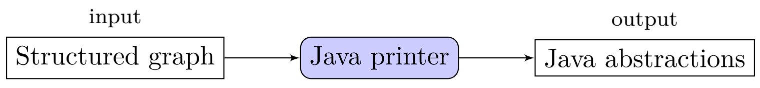


Figure 30: High level overview of Java code generation.

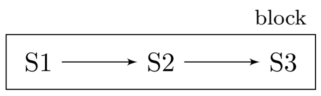
The structured graph that is generated by code restructuring has a one-to-one mapping to Java code. The `block` node corresponds directly to a sequence of statements, `loop` node corresponds directly to a loop statement, `exit` node to a break statement. Thus we can generate Java code by applying the following transformation recursively (graph is shown on the left side while the generated code is shown on the right side):

- Statement node

`i++`

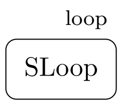
```
i++;
```

- Block node



```
S1;
S2;
S3;
```

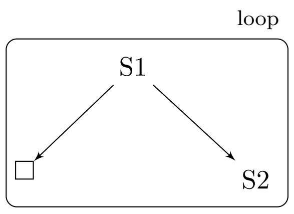
- Loop and exit nodes – loop nodes are converted into do while loops. Exit nodes are converted into break statements. Both of these transformations are shown below.



```
do {
  SLoop;
}
```

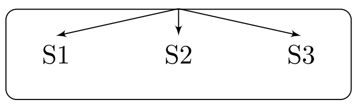
```
break;
```

By combining these two it is possible to generate loops that can be exited. An example is shown below.



```
do {
  S1;
  if (randomChoice()) {
    S2;
  } else {
    break;
  }
}
```

- Branch node



```
int choice = selectState(3);
if (choice == 0) {
  S1;
} else if (choice == 1) {
  S2;
} else {
  S3;
}
```

7 Svm

This section discusses the symbolic execution library used in the project called SVM. It goes into detail about how it works.

Section 7.4 describes the performance improvements that this project contributes to svm in order to simulate Java bytecode faster.

7.1 Svm overview

Svm is an open source symbolic execution engine. Its source can be found on github [32]. It works like a typical symbolic execution engine and explored all possible scenarios that can be produces by an application under analysis. The overview is shown in fig. 31.

The vm is a Java bytecode interpreter. It provides its classloader, operations that simulate bytecode instructions and provides implementation for native methods. Furthermore in order to support some bytecode instructions SVM tags each state with meta information that is not accessible from within a jvm. For example it tags each state with information about its path constraints.

Since a program can exhibit multiple scenarios SVM keeps track of all program states and then executes instructions separately for each one of them. Whenever the execution could follow different paths under different program scenarios is forks the program state and specifies the constraints that would lead to either of these states.

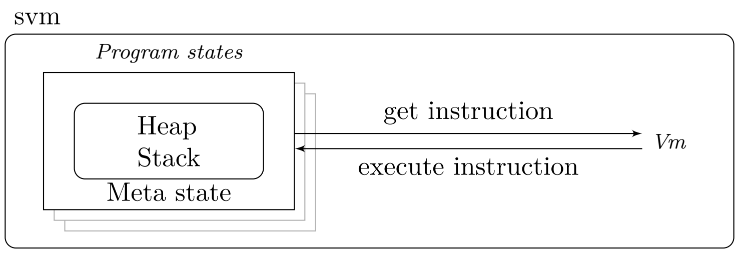


Figure 31: svm execution.

7.2 Svm performance

SVM was designed with equivalence analysis in mind. It comes with support for running multiple versions of a program at the same time. Moreover it can use information from execution of a single version to guide the exploration of another version. In effect it is expected to find violations faster. For instance whenever a program adds a trace item SVM checks whether it violates with traces created by another version.

7.2.0.1 In flight equivalence checking An already existing feature of SVM extends search strategies by checking for equivalence of traces on the fly. Doing so can potentially guide the exploration better. However it also ensures that the partition violation is found as soon as two traces which cause the violation are executed. This limits the complexity of equivalence checking afterwards.

7.2.0.2 Parallelism An intrinsic feature of symbolic execution is that it is a problem that can be highly parallelised. Therefore performance limitations can always be addressed by increasing the amount of hardware on the machine. This gives possibility to explore the entire code by running it on a large machine cluster. Machines would be allocated disjoint code paths.

Exploration of different code paths is a problem that can be parallelised. Consider the following method body:

```
public static void main(String[] args) {
  if (randomChoice()) {
    A();
  } else {
    B();
  }
}
```

When an if branch is taken the code will either execute the `A()` method or the `B()` method. Symbolic execution will fork the state of the program. This makes `A()` and `B()` method calls independent of one another. In effect both of these paths could be run in parallel.

7.3 Search strategy

One of the main configuration parameters that affect the performance of a symbolic execution tool is the choice of a search strategy. If a specification violation exists then it is only necessary to explore two paths to find it (one per program version). However, many programs have an infinite number of paths and only a few paths might actually cause a violation.

Since it is impossible for a program to explore all code paths it needs to use a heuristic by which it will pick the paths which are most likely to contain the violation.

The two simple strategies would include going breadth first search and depth first search.

In the depth first search `SVM` would execute a single path until it terminates and then would proceed with other paths. However if the path itself does not terminate then `SVM` would also not terminate.

In breadth first search in order to explore a path which took `n` branches all paths which took `n-1` branches would have to be explored. This removes the possibility of not exploring one path because another does not terminate. However this algorithm also suffers from path explosion as the algorithm would have to always try all possible paths with `n` branches.

Better strategies include heuristics that guide the program towards breaking paths. `SVM` uses two strategies for which the performance is evaluated in this report. However, due to time constraints this project does not include any new strategies.

The first strategy prioritises the states based on their trace. This is because states that make additional trace elements are more likely to violate with others.

Another strategy uses lines of code to guide the algorithm. The reason for this choice is that it is more likely that the violation will be found by exploring not yet explored code. In addition this heuristic also handles with infinite loops and will try to escape from the loop as soon as possible.

What is important to note is that most strategies highly benefit from randomisation. The main reason to add randomisation is due to time constraints. A good search strategy might be able to find the violation in an hour. However if the timeout is set to 10 minutes no matter how many times the analysis is run the violation will not be found. However, if randomisation is used every time the analysis is run the chance that a violation is found increases.

7.4 Contributions to svm library

In order for symbolic execution to work efficiently the interpreter needs to pose as little overhead over the actual program runtime. Furthermore a symbolic execution might have to handle millions of states that need to be executed. Thus, a lot of `SVM` code is both performance and memory critical.

This section describes contributions made to the svm library in order to make it capable to run more complex examples.

7.4.0.3 Adding timeout to program execution Many programs can run forever. In effect it is important to define a timeout after which `SVM` will say that no violation is detected. In order to make the timeout efficient, the timing logic is placed on a separate background thread and does not cause any performance issues.

7.4.0.4 Adding support for more bytecode instructions. Because Java is a complex language creating an interpreter capable of supporting all of bytecode would take a staggering amount of time. `SVM` supports most commonly used bytecode instructions. In effect it is capable of running more complex Java programs. Nevertheless, it is still incapable of running many simple Java applications.

In order to be able to run more complex examples partial of complete support was added for the following bytecode operations:

- IDIV, IREM – these operators perform division and modulo operations. They and their symbolic counterparts have been implemented into `SVM`.
- INSTANCEOF and CHECKCAST – while `SVM` already supported these operations its implementation was not bug free. In particular `INSTANCEOF <class>` and `CHECKCAST <class>` expected an instance of a class `<class>` to be already initialised by the vm. However, for many programs this is not the case.
- LDC – this is a class load operation that occurs whenever a class is loaded explicitly. For instance an instruction `System.out.println(Object.class)` will require to load the Object class definition. Again previously this operation required the class to be already loaded. However, due to the lazy nature of Java this is not always the case.
- ATHROW – this bytecode instruction is called whenever the program throws an exception. This instruction has not been implemented completely as catching the exception is not supported. However `SVM` will now terminate the paths which throw exceptions instead of crashing the entire `SVM`.

7.4.1 Caching SMT results

A symbolic execution engine spends a lot of its time executing branch conditions. For each branch it needs to determine if it could be taken under some scenario. With symbolic values it does it by making a query to an SMT solver and checking whether the branch conditions are satisfiable.

Querying the SMT solver is a costly operation on its own and can potentially degrade the performance of the symbolic engine. Branch execution can be sped up by caching the queries that were sent to the SMT solver and reusing their results. Relatively sophisticated methods are discussed in the KLEE paper [7].

Whenever a symbolic execution needs to execute a branch instruction it needs to determine whether the then and the else branches are feasible. In some cases both of these branches can be taken conditionally. Take the following snippet as an example:

```
public class SMTExample {
  public static void main(String[] args) {
    if (args.length == 0) {
      System.out.println("No␣args!");
    }
  }
}
```

It can take both branches. For instance calling `java SMTExample` would print the "No args!" message. However calling `java SMTExample 0` would not. In order to determine whether there exists a scenario under which a program would take a branch `SVM` sends the branch condition (in this case `args.length == 0`) to an SMT solver. This is a very costly operation.

By caching the results of SMT queries two cases are optimised. Firstly when the branch condition evaluates to True or False (which happens for most branches) `SVM` no longer makes the SMT query. Secondly, if both versions make the same queries the query is sent to the SMT solver just once.

7.4.1.1 Fast state selection When profiling the code one of the largest performance bottlenecks was the state selection. After diagnosis I noticed that the states were kept in an array list. However, once a state was selected it was removed from a list. Originally this required both a list scan and an array copy. In effect the state selection algorithm had a $O(States)$ complexity. That made the speed dramatically decrease once `SVM` had to handle a lot of states.

Fast selection algorithm made two changes in order to reach an $O(1)$ complexity. Firstly an element from an array list was removed by swapping it with the last one and removing the element from the end of the list. In effect this removed the costly array copy operation. Secondly when the element was selected from the list the state selection algorithm remembered the index at which the item was stored in the list. This removed the requirement to scan the list.

7.4.2 Caching of data

Some objects are continuously created by the programs. One such example are symbols. In order to avoid the overhead caused by redundant memory allocation `SVM` now pre allocates many data structures. Then when a program wants to allocate some data it gets an already prepared instance.

7.4.3 Memory sharing

Since symbolic execution expects to handle many states it cannot have a separate heap for each state. This would be both inefficient as the fork would be costly but would also make `SVM` quickly run out of memory. SVM already did heap sharing so that the heap would be shared between different program states.

This project had went further and implemented data sharing in more places. It adds sharing for the stack of processes only cloning the stack frame for the method call on the top of the stack. Furthermore it uses the Copy on Write technique to share all of the meta state between different versions. Only when one of the states actually modifies the data does it create its own copy.

8 Limitations

There are a few limitations both to the equivalence checking technique as well as the tools used to implement it. This section discusses those limitations and potential solutions to deal with them.

8.1 Project integration

IntelliJ plugin makes it possible to integrate the use of the `loupe` tool inside of an IDE. However, the tool can still not be used out of the box. In order to actually run the analysis the project must contain both version of the program. This is normally not the case as instead the programmer keeps just one version in his working directory. Other versions are available, nevertheless they can only be accessed from git.

8.2 State explosion

One of the major limitations of any symbolic execution technique is that large applications even without abstractions have a lot of paths to explore. These paths increase in size exponentially on branch points. As a result only clever search strategies are able to reach interesting paths (in equivalence analysis an interesting path is the one that causes a partition violation). The problem this causes is that most of the paths remain unexplored.

8.3 Abstraction performance

Equivalence analysis can only be useful for the programmer if it can quickly find violations. Abstractions can immensely change the performance of SVM. Their implementation greatly affects the complexity of classes and the number of states that they generate.

More imprecise abstractions can potentially increase the number of possible code paths that a program can execute. Consider the following two code snippets below, before and after abstraction. The non abstract version the `config.get` method can be explored in one way. On the other hand with abstractions the same method can result in `args.length * 2` different outcomes. This can potentially increase the number of states that SVM needs to explore in order to find the violation[13]. In effect finding the performance might degrade.

```java
public interface StringMap {
  public void putValue(String key, String value);
  public String getValue(String key);
}
```

[13]This depends on the search strategy making it's choice even more important

```
public void main(String[] args) {
  StringMap config = new StringMap();
  for (int i = 0; i < args.lenght - 1; i+= 2) {
    config.put(args[i], args[i+1]);
  }
  System.out.println(config.get("verbose"));
}
```

```
public class StringMap {
  private ArrayList<String> strings = new ArrayList<>();

  public void putValue(String key, String value) {
    strings.add(key);
    strings.add(value);
  }
  public String getValue(String key) {
    return getPassedParameter(strings);
  }
}

public void main(String[] args) {
  StringMap config = new StringMap();
  for (int i = 0; i < args.lenght - 1; i+= 2) {
    config.put(args[i], args[i+1]);
  }
  System.out.println(config.get("verbose"));
}
```

8.4 Parallelism

Typical Java programs do not execute sequential code and in order to boost their performance use multiple threads. The problem that this creates is that the traces could be out of order and thus even running the same version of the program twice could create a violation.

8.5 Svm Java support

`SVM` is a library that simulates bytecode instructions in a symbolic environment. This in order to function it needs to accurately simulate all bytecode instructions. At the moment of writing there are several instructions that are not implemented. For instance no instructions from Java 8 are implemented. Therefore `SVM` cannot execute an arbitrary program.

8.6 Specification language

Specification language forms the basis of equivalence checking as it defines ways in which we can partition the heap. The predicates that current specification language provides to the user are not so expressive. In effect it is tricky in some circumstances to define a partitioning which does not tag too many unaffected objects as affected.

The reason why the specification language cannot be always expressive is due to the way that partitioning is used. Because we want to avoid any backtracking we want to classify objects during bytecode execution when it gets constructed.

There is very little information we can obtain about the object when it is constructed in bytecode. Take for instance the following constructor:

```
new Package(newIntSymbol(), newBooleanSymbol(), newIntSymbol())
```

Its bytecode representation is shown in listing 23. In order to avoid any backtracking we would like to classify the object during the `NEW` bytecode operation (line 1). However at this point we do not have any access to the object's state or even constructor parameters. These parameters are only available at line 6.

In order to mitigate it `SVM` already has an in built algorithm that scans for matching `NEW` and `INVOKESPECIAL.<init>` calls. This way it can tag the object on the `<init>` call once it knows the constructor parameters.

```
1 NEW loupe/examples/packagemanager/Package
2 DUP
3 INVOKESTATIC lexicalscope/.../SymbolFactory.newIntSymbol ()I
4 INVOKESTATIC lexicalscope/.../SymbolFactory.newBooleanSymbol ()Z
5 INVOKESTATIC lexicalscope/.../SymbolFactory.newIntSymbol ()I
6 INVOKESPECIAL loupe/examples/packagemanager/Package.<init> (IZI)V
7 ARETURN
```

Listing 23: Bytecode representation of the new Package construction.

However even with this improvement the specification language is still limited. This is because constructor parameters, call stack and the object which makes the `INVOKESPECIAL.<init>` call are the only pieces of information we have about the objet. Thus we still cannot tag objects based on its field at construction. Consider the implementation of the class below. The current specification language still does not allow to make all packages affected if the `version` field would get instantiated to a positive value.

```java
public class Package {
  public Package() {
    this.version = SymbolFactory.newIntSymbol();
  }
}
```

8.7 Retaining object properties

Abstracted objects usually loose a lot of properties that they inherently had. For instance an ordered list would loose the elements order, a hash map would loose the mapping from keys to values. Being able to detect and preserve those properties would potentially allows to make the abstractions even more precise.

Nevertheless making the abstractions too precise has the potential of decreasing the performance of the tool and actually might make it more difficult to find the bug.

9 Evaluation

This section evaluates the quality of abstractions generated by the `loupe` tool. It assesses two quality criteria of abstractions:

1. Speed with which violations are found.
2. The number of false positives generated by abstractions.

The environment which was prepared for the benchmarks is discussed in section 9.1. Specifically it describes actions taken in order to make the results comparable.

The data collected by the benchmark suite is listed in section 9.2.

The evaluation has been performed by contrasting different techniques (section 9.3). The techniques differ by varying the search strategy and the level of abstraction. Each technique is then evaluated against a set of example programs (section 9.4).

9.1 Testing environment

The testing environment in which the tests were run was controlled in order to give reliable measurements.

Firstly the benchmark was performed on the machine with an `i5-4250U` processor running at `1.30GHz`[14]. The machine was fitted with 8GB of RAM. In addition typical background processes were closed in order to ensure an uninterrupted execution of the benchmark. Furthermore, the benchmark was run against the JVM `1.7.0_45` version on `Mac OS X 10.10.3`.

Secondly the benchmarks have been executed in two steps. The first step run the application multiple times. This has been done in order to ensure that JVM has warmed up. Without this step the second execution would normally take far less time as the code was compiled by the JIT. The second step run the benchmark multiple times and then calculated the average performance.

It is important to note that the symbolic exploration strategy uses randomisation to pick the relevant paths underneath. This causes disparity between successive runs and is the main reason for averaging the results.

Adding randomisation makes the benchmarks return a different result every time it is run. Nonetheless, there are multiple reasons for the use of randomisation both in the benchmark process. Firstly a random strategy is used by `SVM` when finding violations. Therefore by using randomisation we can benchmarks the expected performance of `SVM`. Secondly the search strategies often execute heuristically. Adding randomisation allows the strategies to avoid pathological edge cases.

9.2 Data collected

When benchmarks are run the benchmarking tool executed the `SVM` instances in the same way that `loupe` would run it. After `SVM` finishes the analysis the benchmarking tool collects several metrics that can be used to understand the performance behaviour of the application. The metrics are explained in table 2.

[14]Full CPU specification: http://ark.intel.com/products/75028/Intel-Core-i5-4250U-Processor-3M-Cache-up-to-2_60-GHz

Metric name	Metric description
Time	Time in which svm completed its analysis. This is the main metric used to evaluate performance. The time will be shown as “TO” when a timeout occurred. Because some examples run orders of magnitude faster the time is shown on a logarithmic scale.
Finished states	The number of states which completed execution. This is one potential reason for the time difference between two versions.
Queries	The number of different queries sent to the SMT solver. The queries are generated in order to check the feasibility of different scenarios within one version. They also check if there is a common scenario between two states in two versions. This is one potential reasons for the time difference between two versions.
Violation detected	Shows whether the tool detected a partition violation. Used to detect any false positives/negatives generated by an analysis.

Table 2: Metrics collected by the benchmarking tool.

9.3 Svm configurations

Equivalence analysis can be made more or less efficient by configuring the way that `SVM` explores the program. The main configuration parameter being evaluated by this project is the degree to which the code is being abstracted. Another configuration parameter considers in the search strategy which defines how does `SVM` explore the paths.

The search strategies that svm runs in this benchmark are:

1. Tree strategy – This is a simple strategy which picks one of existing states almost at random. In order to improve performance it groups the pending states by their trace. Then it selects one of the groups at random and then one of the states from within the random group.
2. Line coverage strategy (LOC) – This strategy tries to optimize the search by tracking instruction coverage. When it tries to pick a state to execute it prioritizes states that would be executing previously unreached code. This heuristic can potentially speed up the search since violations are likely to occur on previously unexplored code.

The different abstraction policies that svm is configured with are:

1. Non abstract implementation – This is the class implementation that was a part of the program.
2. Shallow – Naïve abstractions have their method bodies replaced by a single return statement. The return statement returns a symbolic object of a type expected by the method.
3. Deep – This abstraction extends the naïve abstraction. The method bodies also contain method calls to other objects. However, all objects present in the abstractions are symbolic.

4. Mnemonic – This is the abstraction automatically generated by the `loupe` tool. It reuses objects an instance gets passed in method calls. We expect that it should preserve more properties of the classes.

Making the non abstract implementation required no additional work since the class definition was included in the sources. Similarly the mnemonic abstraction policy also required no work since it was generated by `loupe`. However, both shallow and deep abstractions were written by hand. They were created for comparison to see how other potential abstractions would perform.

The IO in the benchmarked examples could not be execute concretely since `SVM` does not model the IO interactions. Therefore it has been replaced by abstractions that return symbolic values. In effect program will receive symbolic inputs.

9.4 Choice of benchmarking examples

Initially real world Java applications were candidates for the benchmarking. However, they were either unable to run on `SVM` that supports a Java subset and required a change specification. This made it impossible to simply apply the technique to existing source code.

To solve this problem the benchmarking examples were hand written and tested different aspects of the tool. We created benchmarks with three aims in mind:

1. Check abstraction instrumentation overhead (results in section 9.5.2) – As explained in section 6.1.4 all bytecode instructions are instrumented and wrapped into an abstraction operation. One set of benchmarks (called `C1`) was created to test whether or not the presence of abstract operations cause an overhead. In case of `C1` benchmarks no classes were listed for abstraction. For each example we run `SVM` with and without the instrumentation enabled.
2. Check abstraction performance (results in sections 9.5.3 and 9.5.4) – Abstraction policy and search strategy can affect the performance with which violations are found. Therefore a series of examples `C2` was created. All of these examples contained a specification violation and `SVM` was expected to find it. Furthermore for these examples some classes were abstracted according to all abstraction policies.
3. Check accuracy
   - Check false negative rate (results in section 9.5.5) – False negatives occur if `SVM` fails to find a violation when it exists. This happens whenever `SVM` times out. Since `C2` examples all contained violations they are used to find the false negative rate.
   - Check false positive rate (results in section 9.5.6) – False positives occur then `SVM` finds a violation while the code change does not violate a change specification. Therefore examples `C2` could not be used to test the false positive rate. Thus examples 5 examples `C3` were created that did not violate the specification.

In total 19 examples were generated and contained 6734 lines of code. Some of the examples were really small and contained just 20 lines of code. The largest example was made of 450 lines. A lot of these examples make use of a standard library that was also created. Because some `java.util` containers used methods not supported by `SVM` I created similar classes as a part of the benchmark. The classes are: `ArrayList`, `HashMap`, `Trie`, symbolic string `Str`, `Logger`.

Selected examples have been added to the appendix section for reference.

9.5 Results

This section gives an analysis of the results. Raw table of benchmark results can be seen in appendix B. These results have been processed into figures and tables and are discussed in the following sections.

9.5.1 Automatic generation of abstractions

The classes meant to be abstracted have a varying complexity in terms of the control flow. Some examples contain method bodies comprised of just a single statement while others have more complicated branching instructions. Benchmark examples contain if/else statements, loops, switch statements. For all of these examples `loupe` is capable to automatically generate abstractions and run them in `SVM`.

One limitation of automatic generation is that the source code generated does not always turn out to be pretty or fully optimised. Hand crafted abstractions can have a much simpler logic. Potential optimisations of the generated code are left for future work. An example which shows potential simplifications is shown in listings 35 and 36.

Another side of abstraction generation is the aggressiveness of over approximations. Other examples in the benchmark have a higher over approximation. However, making such abstractions automatically is non trivial. These abstractions were built with the knowledge of concrete behaviour in mind. This is needed especially when a method returns an `Object` instance while it returns `String` objects.

Another limitation of the automatic generation is that `loupe` does not handle exceptions. A potential extension would add support for try/catch cases. Despite it limiting the classes that could be abstracted it did not make examples poor. Firstly a lot of classes do not throw exceptions. Secondly `SVM` does not completely handle exceptions either.

9.5.2 Overhead of abstraction instrumentation

This series of benchmarks runs tests to check the overhead of abstraction instrumentation (defined in section 6.1.4). The results of these benchmarks figs. 32 to 34 indicate that the overhead is minimal.

The timings shown in fig. 32 show that the time to find the violation with and without the instrumentation is comparable. Running abstraction instrumentation added at most a 10% overhead for the examples that did not time out[15]. Surprisingly for some of them searching for the violation with abstraction instrumentation improved performance. However this can be attributed to the random nature of exploration.

The same conclusion can be reached when comparing for the unique queries computed and the number of states that reached termination (figs. 33 and 34). If we compare runs that use the same strategy abstractions generate 10% more queries and traverse roughly 10% less states. This disparity holds even for the examples for which the analysis timed out. Thus, even in those cases the analyses have made similar progress.

[15]When comparing the time it only makes sense to compare examples that did not time out. This is because the examples were stopped right after a minute and so it looks like they have terminated at the same time. However, it could still be possible for one of them to complete right after a minute while the other would complete after 2 minutes.

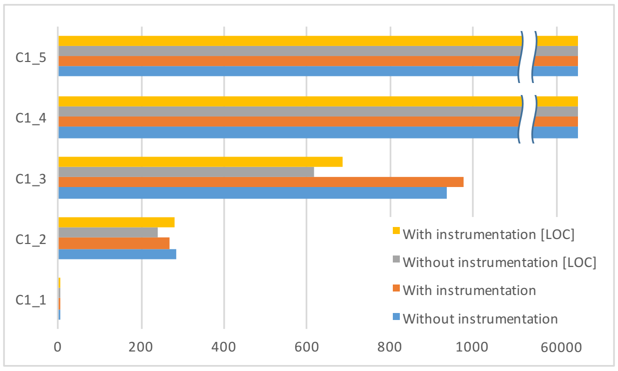


Figure 32: Comparison of time taken for concrete examples. It is important to note that expressions and webapp examples took 60000 milliseconds.

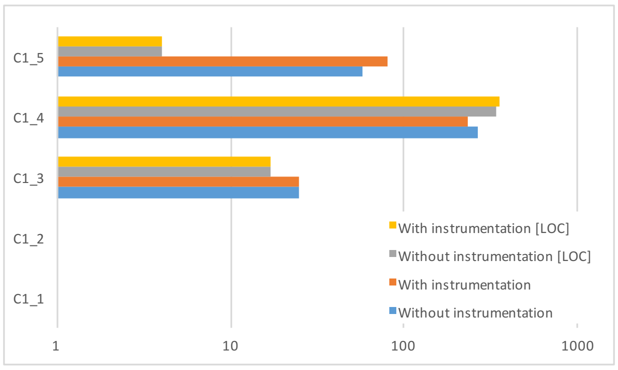


Figure 33: Comparison of states terminated for concrete benchmarks plotted on a logarithmic scale.

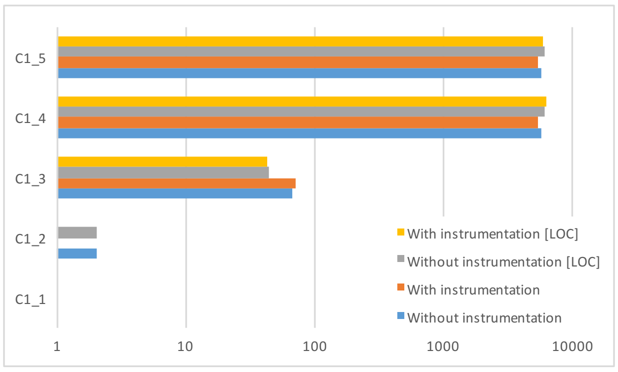


Figure 34: Comparison of the number of queries for concrete benchmarks plotted on a logarithmic scale.

9.5.3 Effect of abstraction policy on performance

In this section we compare the effect of abstraction policy on performance. Thus, we only compare timings which use the same search strategy.

Figure 35 compares the timings of the different runs. Unlike for the concrete examples here we can see a large disparity in performance. Because of this the results had to be plotted on a logarithmic scale. It is worth paying attention to that no strategy outperforms others for all examples.

For some examples there is a staggering advantage in using abstractions. For instance in case of `C2_3` and `C2_4` examples mnemonic abstractions perform 30 times faster compared to the non abstract search strategy which times out. Deep was also capable to find the violation with a similar time. On the other hand Shallow abstraction was unable to find the violation. This is because the violation was only triggered if the abstracted class made relevant method calls.

For some examples there is a big can see a big disadvantage in using abstractions. Look for example at the `C2_2` example in figs. 35 and 37. There are a few interesting observations we can make about the example. The non abstract version has reached 30 states in 2423 milliseconds. On the other hand the deep and mnemonic versions have reached 23049 and 42918 states in a minute. Thus, when the abstracted version is used `SVM` explores states 30 or 55 times faster. Nevertheless, `SVM` was still unable to pick a state which contained a violation. In addition we can notice that the naïve implementation completed just in 30 milliseconds. However unlike all other analyses it was unable to find the bug.

We can see that the speed improvement is caused by the lowered number of SMT queries. It is orders of magnitude lower compared to the non abstract classes as shown in fig. 37. This explains why abstractions perform faster by a factor of 30.

Because of this the mnemonic strategy highly increases the number of explored terminating states within the same time. For instance in the `C2_1` example it explores 60 times the number of states the non abstract strategy does. Thus, whether on not an abstract strategy finds a violation depends on whether abstractions generate more than 60 times the number of states the exploration of a concrete class would cause.

If we look at the `C2_6` example the mnemonic strategy makes 300 times less SMT queries (4500 queries). When we compare the number of states explored we can see on the other hand that abstractions explore 300 times more states.

9.5.4 Effect of search strategy on performance

Another useful comparison is to compare different search strategies between each other and see how they compare. In order to make the comparison we made the table 3. It shows the ratio between the time it took the tree strategy to find a violation over the time it took the tree LOC strategy to find a violation a particular example. Values larger than 1 mean that the tree strategy was slower. Values smaller than 1 mean that the tree strategy was faster.

What can be noticed is that neither of these techniques is always better.

For many examples the ratio equals to 1. This is caused by two reasons. Firstly some examples are small and the search strategy has no effect on the speed. Secondly some examples time out. For these examples both of the search strategies timed out.

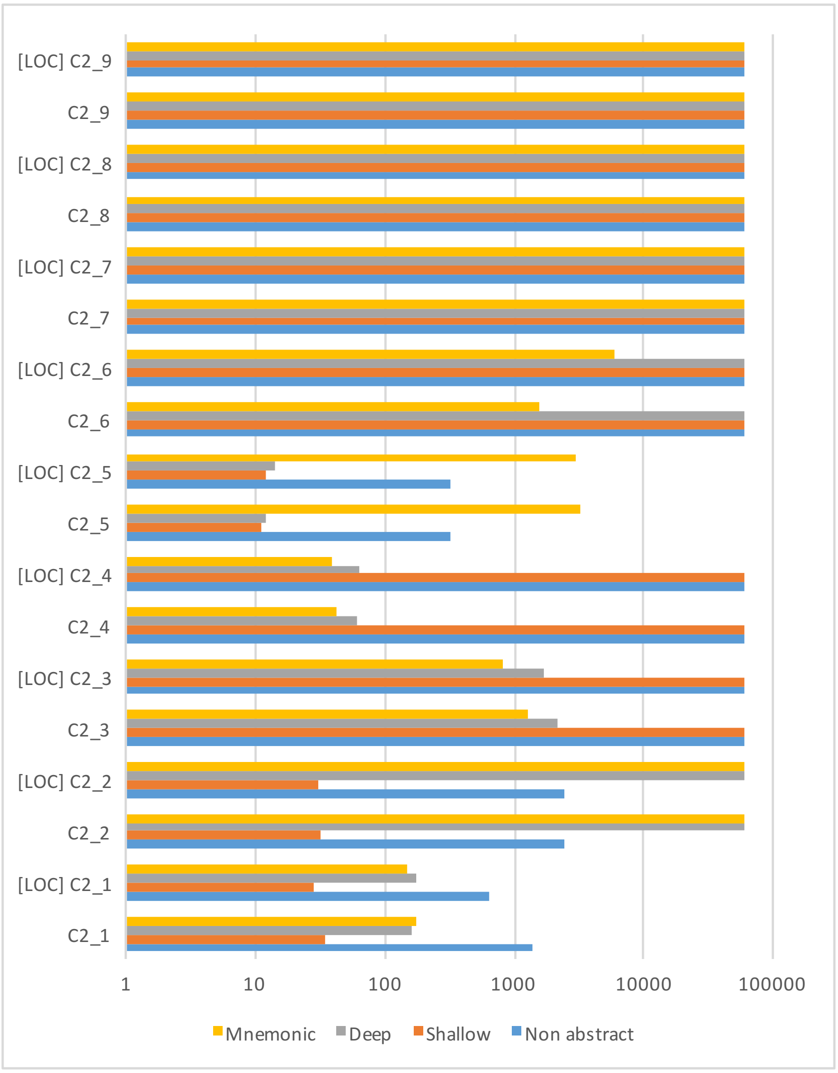


Figure 35: Comparison of time taken for abstract benchmarks grouped by examples and search strategy. It is plotted on a logarithmic scale. Since all examples above contained a violation runs that took a minute (60000 milliseconds) signify a missed violation (false negative). Examples prefixed with [LOC] were run using the LOC strategy.

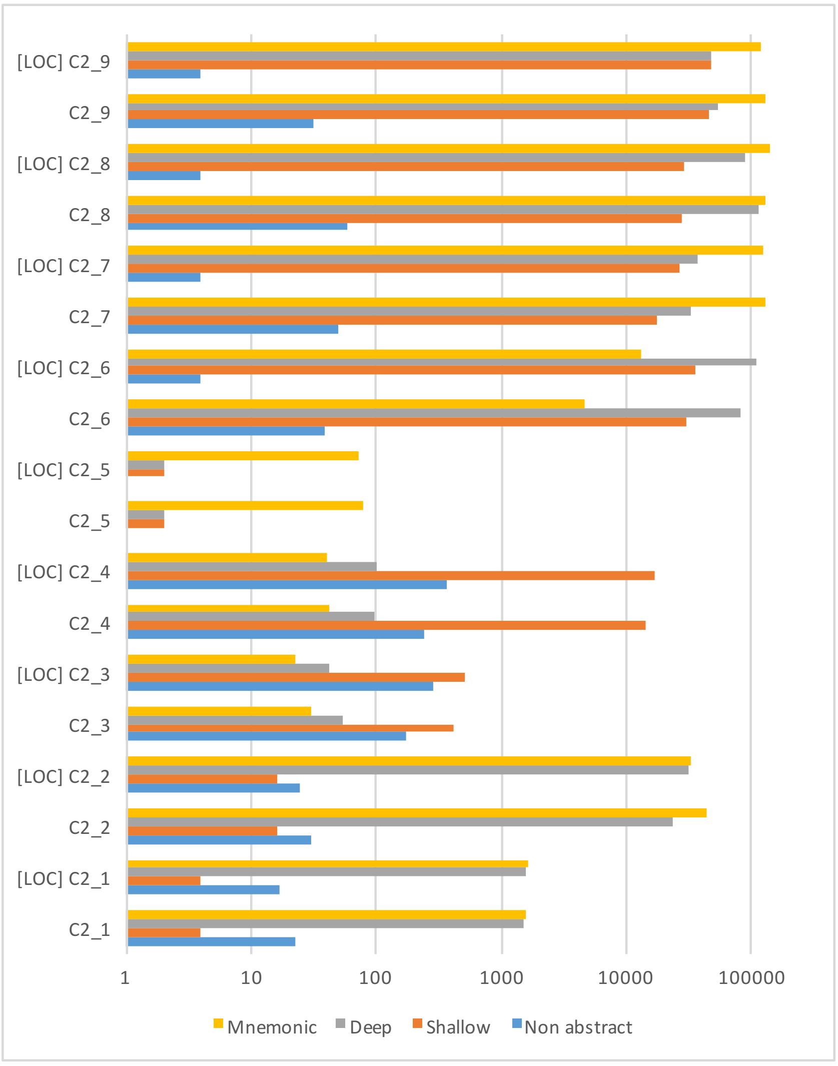


Figure 36: Comparison of states terminated for abstract benchmarks grouped by examples and search strategy. Examples prefixed with [LOC] were run using the LOC strategy.

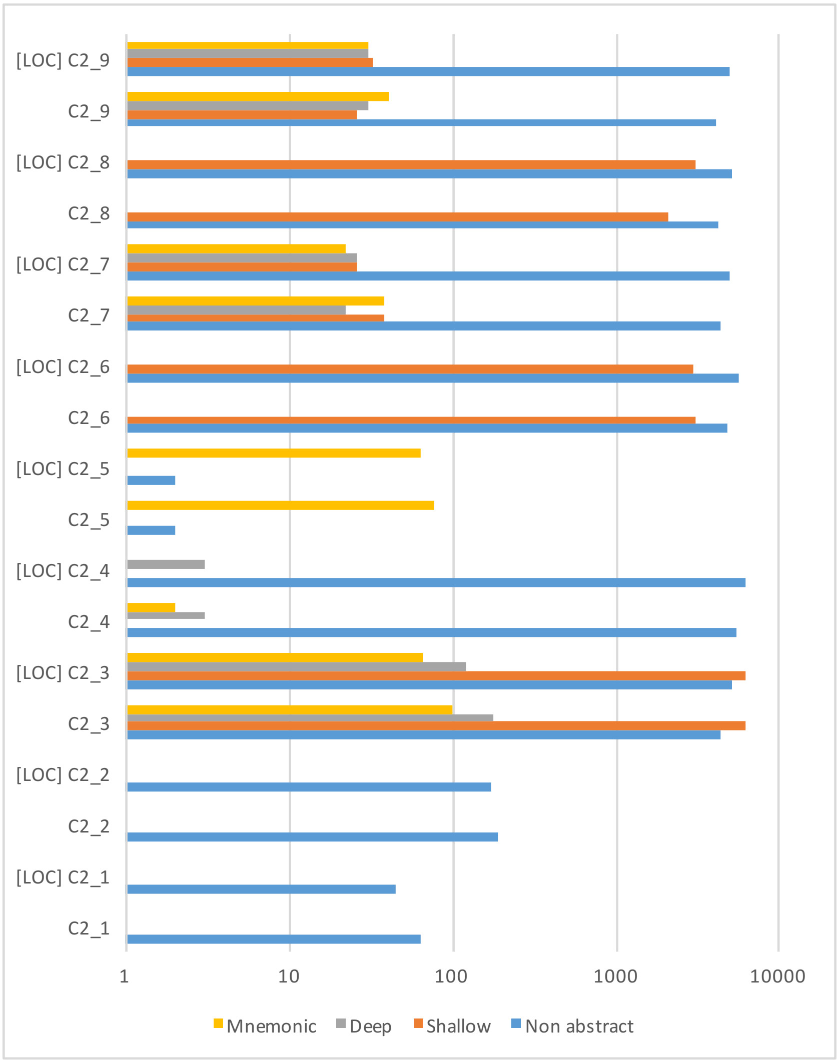


Figure 37: Comparison of the number of queries for abstract benchmarks. It can be noticed that for many examples the non abstract strategy is the only one that makes any queries. The mnemonic strategy makes most queries out of abstraction strategies. Examples prefixed with [LOC] were run using the LOC strategy.

Example	Non abstract	Shallow	AbstractionsMiddle	AbstractionsBest
`C2_1`	2.12	1.25	0.94	1.19
`C2_2`	0.98	1.07	1.00	1.00
`C2_3`	1.00	1.00	1.30	1.56
`C2_4`	1.00	1.00	0.98	1.08
`C2_5`	1.00	0.92	0.86	1.09
`C2_6`	1.00	1.00	1.00	0.26
`C2_7`	1.00	1.00	1.00	1.00
`C2_8`	1.00	1.00	1.00	1.00
`C2_9`	1.00	1.00	1.00	1.00

Table 3: Amount of time taken by the tree search over amount of time takes by the line of code search strategy.

For the majority of remaining examples the tree strategy is slower than the lines of code strategy. In most examples the difference is negligible and the ratio is small than 1.10. The `C2_1` example stands out where the tree strategy requires over twice the time. This is most likely caused by the fact that the violation happens when a method is called on an object of a different class. Therefore the lines of code strategy will prioritise this example.

Some examples actually execute faster using the tree strategy. For instance in the `C2_6` example the tree strategy finds the violation four times faster. The possible explanation of this phenomenon is that exploration of lines that were already explored is extremely penalised requiring **all other code paths** to be explored first. For instance if a loop needs to be iterated twice firstly all paths that do not iterate over the loop will be explored.

Other strategies do not completely omit states that try to explore already explored lines of code, just decrease the weight with which these states get picked. For instance authors of PEX describe the following strategy:

> In order to avoid getting stuck in a particular area of the program by a fixed search order, Pex implements a fair choice between all such unexplored branches of the explored execution tree. Pex includes various fair strategies which partition all branches into equivalence classes, and then pick a representative of the least often chosen class. The equivalence classes cluster branches by mapping them:
>
> – to the branch statement in the program of which the execution tree branch is an instance (each branch statement may give rise to multiple branch instances in the execution tree, e.g. when loops are unfolded),
>
> – to the stack trace at the time the brach was recorded,
>
> – to the overall branch coverage at the time the branch was recorded,
>
> – to the depth of the branch in the execution tree.
>
> Pex combines all such fair strategies into a meta-strategy that performs a fair choice between the strategies. [33]

What we can also notice is that different strategies seem to suit different abstraction levels more. While the simple abstractions performs faster in general using the tree strategy the abstract abstractions perform faster in general using the LOC strategy.

9.5.5 Effect of abstractions on false negative rate

Table 4 extracts the timings from fig. 35 and counts the number of found violations. Since all of these examples contained a violation we count all examples for which the time to find a violation was smaller or equal to a minute (time out).

We can notice that the mnemonic strategy has the highest detection of violations. It finds 2 more violations compared to the non abstract version. Other abstraction strategies lie in between non abstract and mnemonic strategy.

What is important to note is that the mnemonic strategy does not find all violations that the non abstract strategy does. In effect by running all strategies in parallel and reporting a violation if any of the strategies reported one we would find even more violations. The combined strategy is capable of finding twice the number of violations found by the non abstract version.

While the LOC and tree strategies impact the speed with which the violations are found there were no examples where one of them would time out while the other would not. In effect the number of found violations is the same for both strategies.

Abstraction strategy	Found violations for tree strategy	Found violations for LOC strategy
Non abstract	3	3
Shallow	3	3
Deep	4	4
Mnemonic	5	5
Combined	6	6

Table 4: Number of examples (out of 9 examples in total) for which a violation was found by svm. Running svm with abstractions increases the number of violations found within a time limit. The last row expresses the number of violations that would be found if a violation is found whenever one of the strategies reports a violation.

9.5.6 Effect of abstractions on false positive rate

False positive examples try to differentiate the false positive rate for different abstraction strategies. We expect the programs to time out as this means that the `SVM` did not find a violation. Because of this it is not useful to compare the timings as the more accurate strategies would be considered "slower". Therefore, the only metric we will compare on is the percentage of times that different techniques have mistakenly found a violation.

Abstraction policy	False positives for tree strategy	False positives for LOC strategy
Non abstract	0	0
Shallow	1	1
Deep	1	1
Mnemonic	1	1

Table 5: Number of times svm incorrectly reported a violation (false positives). There were 4 examples in total.

Table 5 shows the percentage of false positives. Because non abstract execution runs the program as is, when a violation is found it would occur in a real program. As a result non abstract classes do not generate any false positives. On the other hand running abstracted classes does make a small false positive rate.

However the false positive rate is not terrible for two reasons. Firstly the abstraction keeps track of its behaviour. Thus, it would be possible to run the program once again with a non abstract class and find whether a non abstract class would also have the behaviour that caused the bug. Secondly this is still a rare behaviour. The example is shown in appendix A.3 however the specification change was generated specifically to trigger a false positive.

Nonetheless, it is hard to predict how many developers would build such contrived scenarios. Scenarios where the specification would assume properties of the system and the paths it could take. And scenarios where the abstracted class would return an over approximation that affects the paths that the program may take.

9.5.7 Summary

Firstly it is worth noticing that there is no significant overhead caused by adding abstraction instrumentation.

Overall abstractions code is a technique which makes it quicker for symbolic execution engines to explore program behaviours. Typically abstractions generated by the `loupe` tool can be explored faster. When abstractions are used `SVM` terminates with about 30 times more states explored. This is caused by the fact that abstractions reduce the number of queries.

Because abstractions are faster they can find more violations within a specified time out. However they also raise false positives and detect violations not present in the actual program.

In some cases finding a violation in over-approximations requires far more states to be explored. In those cases the abstractions generated by `loupe` no longer find violations faster. However, less precise approximation strategies (such as the shallow) decrease the number of states that have to be explored. For instance when approximating a `Map<String, String>.get(key)` instead of returning all values passed to the map a shallow abstraction would return one symbolic string. While returning a symbolic primitive is easy it becomes more difficult when the abstraction needs to return a symbolic object.

10 Conclusions

This project presents the `loupe` tool which can generate abstractions used in equivalence analysis. As a result it finds more bd spec violations than an analyser without abstractions. It can generate abstractions automatically making the abstraction system really easy to use. In order to generate abstractions for classes with complex behaviours `loupe` makes the abstraction logic closely resemble the non abstract code. As shown by section 9.5.3 out of all abstraction strategies the one used by `loupe` finds most violations. Nevertheless, in many cases it is slower compared to other strategies.

The project demonstrates that abstractions can improve the speed with which the analysis is performed. For the majority of benchmarks abstractions improved the speed with which violations are found. The use of abstractions increases the speed of exploration by factor larger than 30. In effect if the number of states to explore does not increase by a larger factor the violations will be found faster.

The key implementation choice that made such performance possible is making exploration of abstractions generate almost no SMT queries. In fact the SMT queries often turned out be the largest bottleneck of symbolic exploration.

Another feature of `loupe` is that it can generate the Java code for abstractions. In doing so it restructures bytecode into higher level Java code. It does it by doing a single traversal of the control flow graph. While the generated code is made for abstractions the algorithm is suitable for use in a decompiler.

While useful the proposed solution has several limitations which prevent it from being used to test large programs at the moment. Firstly, because of the complexity of the Java language the `SVM` library does not yet support all bytecode instructions. In effect not all programs can be run yet on `SVM` to begin with.

Secondly the abstractions generated by `loupe` are not the fastest. Abstractions which over approximate the behaviour even further usually trade off the accuracy for the performance. However, it is possible that some strategies could improve the performance without sacrificing the accuracy. An example strategy could combine the mnemonic and shallow strategies and prioritise the search using the shallow strategy. Therefore there is still room for improvement of abstraction algorithm.

Thirdly the number of false positives is small despite abstractions having many behaviours that non abstract classes do not have. The small false positive rate can be attributed to the fact that `SVM` executes and compares the behaviours of two versions. Thus in a single version execution if an abstraction would throw a `NullPointerException` an incorrect behaviour would be found. On the other hand in case of a multi version execution this behaviour is fine as long as the other version also throws the `NullPointerException` for that scenario.

Overall the project makes several contributions. First and foremost it shows a potential use for class abstractions in a symbolic execution engine. By doing so it gives a possibility to find violations that would not be found as quickly without their use. Secondly the project shows a new way in which bytecode can efficiently be decompiled into abstractions in a single pass.

10.1 Future work

This section contains changes that could be made to the project in order to improve it. The changes try to address the limitations of the current implementation. They concentrate on improving the overall performance of `loupe` and making the equivalence analysis more flexible.

Features which are discussed in the subsections in more detail that could be introduced are:

- A way of removing pure functions from the traces to make trace equivalence more lenient.
- A more interactive, graphical way of creating behaviour change specifications.
- Performing optimisations of the abstraction logic.
- Perform different abstractions in parallel. Since different abstraction levels have better performance under different scenarios perhaps using all of them would have the overall tool find violations faster.

- A way of reducing the number of states required to thoroughly explore a number of code paths. While this feature the path merging algorithm discussed below could highly limit the path explosion its implementation requires many changes to the symbolic engine. Moreover it isn't certain whether the current search strategies are good enough that they pick the interesting paths anyway.

A useful extension to the project would integrate with build tools like ant or maven. This would make it possible to automatically test the equivalence with respect to the last checked in version to a version control system. A tool could simply check out the sources from a version control system to a temporary directory and use the build configuration in order to detect the necessary set up required to run the main application.

10.1.1 More lenient trace equivalence

Path equivalence analysis checks for any method calls made on the *affected/unaffected* boundary since it could potentially be responsible for a mutation of an unaffected.

As explained in the background (section 2.1.5) this creates a big limitation. In reality there are many programs that have exactly the same behaviour but produce different traces. For instance if pure methods are added to the trace they definitely cannot affect object state. Yet they will cause a partition violation if another version of the program does not also call the same pure method.

The result of such trace equivalence is that if add a new getter call to an object from an affected object both object suddenly become affected. This is despite the fact that calling the getter cannot affect an object in its intuitive sense. Let's consider the following program.

```
interface Drink {
  public void drink();
}

class Alcohol implements Drink {
  public void drink() {
    System.out.println("Drink␣it␣all");
  }
}

class SoftDrink implements Drink {
  public void drink() {
    System.out.println("So␣fizzy");
  }
}

class Person {
  private int age;
  public Person(int age) {
    this.age = age;
  }

  public int getAge() {
    return age;
  }
}
```

```
class Vendor {
  public Drink giveAlcohol(Person p) {
    return new Alcohol();
  }
}

class AdultBuyingAlcohol {
  static void main(String[] args) {
    int age = Integer.parse(args[0]);
    Person adult = new Person(age);
    Vendor v = new Vendor();

    v.giveAlcohol(adult).drink();
  }
}
```

```
class Vendor {
  public Drink giveAlcohol(Person p) {
    if (p.getAge() >= 18) {
      return new Alcohol();
    } else {
      return new SoftDrink();
    }
  }
}

class AdultBuyingAlcohol {
  static void main(String[] args) {
    int age = Integer.parse(args[0]);
    Person adult = new Person(age);
    Vendor v = new Vendor();

    v.giveAlcohol(adult).drink();
  }
}
```

Version 2 of the Vendor program represents the way that current vendors in many supermarkets function checking the age of people and selling alcohol to just people over the drinking age. For the change specification we can say that all instances of a `Vendor` were affected since the vendor can now prohibit some people from drinking. However if the use the original equivalence notion all instances of `Person` are also affected. If we would make the `Person` unaffected then the following traces would be captured:

```
new Vendor()
return 0
Vendor.giveAlcohol(alias-0)
  new Alcohol()
  return alias-1
return alias-1
Drink.drink(alias-1)
System.out.println("Drunk")
return
return
```

Listing 24: Trace created by executing the first version of vendor.

```
new Vendor()
return 0
Vendor.giveAlcohol(alias-0)
  Person.getAge()
  return symbol-0
  new Alcohol()
  return alias-1
return alias-1
Drink.drink(alias-1)
System.out.println("Drunk")
return
return
```

Listing 25: Trace created by executing the second version of vendor.

The reason why these traces are not equivalent is because the second version of a `Vendor` class calls the `getAge` method. It was added to the trace because potentially it could mutate the state of the `Person` object. However in this case `getAge` resembles a simple object getter and cannot affect the state of a `Person`.

This is why a possible extension of `SVM` would be able to get a list of method calls that are pure. These methods would then be left out of the trace. This would make the affected objects have a more intuitive meaning.

10.1.2 Easier generation of partitions

A big pain point of the equivalence analysis is that it requires a change specification for every code change made. One can argue that this corresponds to the way that programmer write unit tests. Nevertheless in case of a refactoring change it might not be necessary to make any changes to unit tests. On the other hand a bds needs to be defined for **every** code change.

Ideally an equivalence analysis program such as `loupe` could generate the partitioning automatically. While it is possible to make an algorithm that creates a partitioning for which no violation occurs one would have to be careful to make sure it finds a representative partitioning. It is highly likely that it would reach a local minimum instead. For instance it could classify all classes as *affected* which is not too useful for a programmer.

A reasonable solution would make the process more interactive only suggesting the initial partitioning to work on. For example, by default all modified classes between the two versions could be tagged as affected.

In addition to make it easier to alter the change specification the programmer could be presented with a GUI. The UI would show the dependency graph between the classes. Making changes to the specification would require clicking on the classes present on the graph.

Making such a UI would potentially make it quicker to define and change the change specification. At the same time it would not make the tool less useful as the programmer would still be able to provide an arbitrary specification.

10.1.3 Path merging

A significant problem with symbolic execution is the presence of path explosion. In this project it occurs whenever an abstracted method performs a dynamic non-deterministic choice and is called many times. Let's take an example of an abstract HashMap.

```java
public class HashMap {
  private ArrayList<String> strings = new ArrayList<>();
  private ArrayList<Integer> integers = new ArrayList<>();

  public void put(String key, Integer value) {
    strings.add(key);
    integers.add(value);
  }

  public Integer get() {
    return getPassedParameter(integers);
  }
}

public MapTester {
  @Test
  public testHashMapContainsElements() {
    HashMap hm = new HashMap();
    for (int i = 0; i < 100; i++) {
      hm.put(Integer.toString(i), i);
    }
    for (int i = 0; i < 100; i++) {
      assertNotEqual(hm.get(Integer.toString(i), null));
    }
  }
}
```

Since the abstraction keeps no true state about what objects get added the program tries to return one of the passed values. However, given that 100 values were passed and a method is called a 100 times then the total number of paths to explore is 100^{100}. This means that the effectiveness of symbolic execution will depend enormously on the search strategy. It will only be able to explore a fraction of all code paths.

Figure 38 shows the paths that can be executed by the test method. What can be observed about the code above is that the results of method calls for `hm.get` are **mostly independent** of one another. There are 100 paths where the method call `hm.get("0")` returns a non null value. At the same time all of these paths behave in the same way as the test no longer queries for `hm.get("0")` again. Therefore instead of the `hm.get("0")` method creating a 101 states it could create two: one where the result is null and one where it isn't.

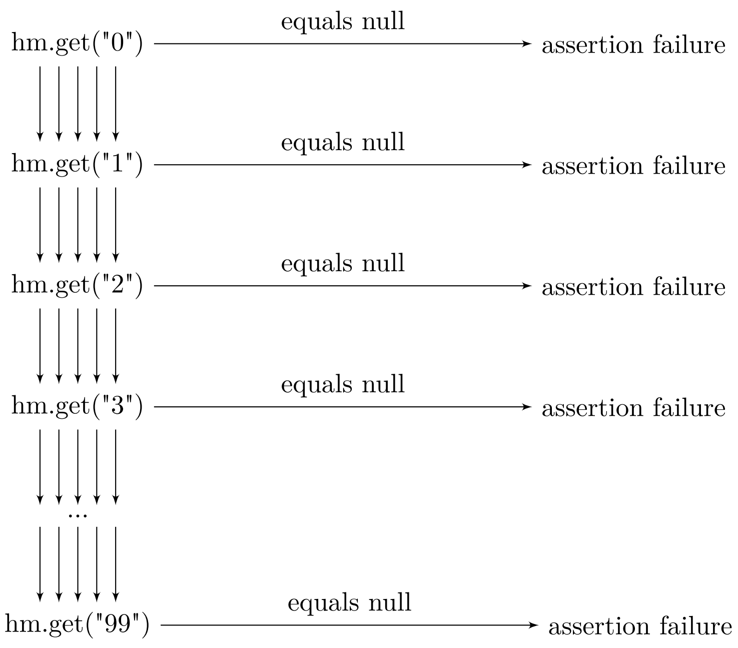


Figure 38: Paths of the testHashMapContainsElements test.

What this observation leads us to is a question: **can we explore all code paths without ending up with** 100^{100} **states?** By exploiting data independence we can limit state explosion to only occur the results so then we can save the program from falling into the exponential state explosion when it really is not necessary and would allow us to execute the bulk of the program in once doing a Single Instruction Multiple States execution.

Specifically by observing the fig. 38 I realised that the main issue is that once a state forks into two paths it stays that way forever. This happens even if both paths later execute the same instruction. As a result it takes twice the time for symbolic execution to explore that state.

My hypothesis is that for most real applications the code that could be shared across different paths is high. If this is the case then it would be worthwhile merging paths, a technique which I though of.

Path merging is a technique in which symbolic execution would wait for different paths to reach the same instruction. Afterwards it could execute different paths under one state. Consider the code below. When line 2 is executed the state will be forked for cases when `args.length == 0` and case when `!(args.length == 0)`.

Once one of the paths executes line 2 it must wait for another path to complete. This needs to be done so that line 3 can be executed for both paths at once. After the second path reaches line 3 both paths can be merged. To make path merging possible the state that holds both cases would need to map the value of `x` to both 1 and 2. This allows then the line 3 to be executed by one state since all paths part of the state will compute the same result.

There are cases though when the merged state can compute different results. When line 4 is executed then the assertion fails for one state but passed for another. If this case the symbolic execution would split the path again and execute them separately.

```
1 public static void main(String[] args) {
2   int x = args.length == 0 ? 1 : 2;
3   System.out.println("Hello␣World!");
4   assert x + x == 2;
5 }
```

Path merging requires a lot of changes to the way that the engine executes instructions. The following changes are required:

1. Firstly as shown in the example above it is possible for a heap or stack object to have multiple values for one state. Thus each value in memory needs to be able to represent multiple values.
2. Bytecode instructions such as load, store, branches, arithmetic operations would need to support the fact that the memory location could have multiple values in memory. For the example above the expression `x + x` at line 4 would evaluate to 2 or 4 depending on the circumstances.
3. Most importantly in order to make this algorithm helpful symbolic execution needs to put some states on hold until other states reach the same program point and then efficiently merge them. In the HashMap example (fig. 38) ideally the symbolic execution would wait for all paths to complete the assertion after `hm.get("0")` is called. Then it would be able to merge 100 different states into 1. However, in general this is not a trivial problem as some states may throw exceptions, return from a method call or loop forever.

A Appendix: Code examples

A.1 Package Manager

The first example shows a possible implementation of a package manager with only the `update` method implemented. Listings 26 to 29 show the code for the first version (except for the PackageMap for the local repository).

The new version of the code tries to ensure that the package is installed both if it was not yet installed and also if a new version is available and the listing 30 shows how this class would change.

Listing 31 shows how the `PackageMap` class could be abstracted.

```java
package loupe.examples.packagemanager;

import com.lexicalscope.svm.j.instruction.symbolic.symbols.SymbolFactory;

public class Package {
    private final int versionNumber;
    private final boolean isInstalled;
    private final int isDeveloper;

    public Package(int versionNumber, boolean isInstalled, int isDeveloper) {
        this.versionNumber = versionNumber;
        this.isInstalled = isInstalled;
        this.isDeveloper = isDeveloper;
    }

    public void install() { }
    public int getVersionNumber() { return versionNumber; }
    public int isDeveloper() { return isDeveloper; }
    public boolean isInstalled() { return isInstalled; }
}
```

Listing 26: Package class. Shared between both versions.

```java
package loupe.examples.packagemanager;

public class PackageManagerApp {
    public static void main(String[] args) {
        updateAllPackages(args[0]);
    }

    public static void updateAllPackages(String packageName) {
        new PackageManagerApp().update(packageName);
    }

    private void update(String packageName) {
        PackageManager manager = new PackageManager();
        manager.updatePackages(packageName);
    }
}
```

Listing 27: Package Manager class. Shared between both versions.

```java
package loupe.examples.packagemanager;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.HashMap;

public class OriginalPackageMap {
    private HashMap<String, Package> packages;

    public OriginalPackageMap() {
        try {
            Connection connection = DriverManager.getConnection("localhost");
            Statement statement = connection.createStatement();
            String query = "SELECT␣name,␣versionNumber,␣" +
                         + "isInstalled,␣isDeveloper␣"
                         + "FROM␣packages;";
            ResultSet resultSet = statement.executeQuery(query);
            while (resultSet.next()) {
                String name = resultSet.getString(0);
                int versionNumber = resultSet.getInt(1);
                boolean isInstalled = resultSet.getBoolean(2);
                int isDeveloper = resultSet.getInt(3);
                put(name, new Package(versionNumber, isInstalled, isDeveloper));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }

    public void put(String key, Package value) {
        packages.put(key, value);
    }

    private Iterable<Package> getMembers() {
        return packages.values();
    }

    public Iterable<String> getKeys() {
        return packages.keySet();
    }

    public Package getValue(String key) {
        return packages.get(key);
    }
}
```

Listing 28: Concrete implementation of the package map.

```
package loupe.examples.packagemanager;

public class PackageManager {
    private PackageMap externalRepository = new PackageMap();
    private PackageMap locallyInstalled = new PackageMap();

    public void updatePackages(String packageName) {
        Package externalPackage = externalRepository.getValue(packageName);
        Package localPackage = locallyInstalled.getValue(packageName);
        int developer = localPackage.isDeveloper();
        int currentVersion = localPackage.getVersionNumber();
        int externalVersion = externalPackage.getVersionNumber();

        if (!localPackage.isInstalled()) {
            localPackage.install();
        }
    }
}
```

Listing 29: Package manager class for the first version.

After making the change

```
package loupe.examples.packagemanager;

public class PackageManager {
    private PackageMap externalRepository = new PackageMap();
    private PackageMap locallyInstalled = new PackageMap();

    public void updatePackages(String packageName) {
        Package externalPackage = externalRepository.getValue(packageName);
        Package localPackage = locallyInstalled.getValue(packageName);
        int developer = localPackage.isDeveloper();
        int currentVersion = localPackage.getVersionNumber();
        int externalVersion = externalPackage.getVersionNumber();

        if (!localPackage.isInstalled()) {
            localPackage.install();
        } else if (currentVersion < externalVersion) {
            localPackage.install();
        }
    }
}
```

Listing 30: Package manager class for the second version.

```
package loupe.examples.packagemanager;

import com.lexicalscope.svm.j.instruction.symbolic.symbols.SymbolFactory;

public class PackageMap {
    private ArrayList<String> strings = new ArrayList<>();
    private ArrayList<Package> packages = new ArrayList<>();

    public void put(String key, Package value) {
        strings.add(key);
        packages.add(value);
    }

    private Iterable<Object> getMembers() {
        return null;
    }

    public Iterable<String> getKeys() {
        return strings;
    }

    public Package getValue(String key) {
        return createPackage();
    }

    public static Package createPackage() {
        return new Package(
                SymbolFactory.newIntSymbol(),
                SymbolFactory.newBooleanSymbol(),
                    SymbolFactory.newIntSymbol());
    }
}
```

Listing 31: Abstracted version of the package map.

A.2 Web app example

A class hierarchy of the web app example is shown in fig. 39. For simplicity WebApp program uses a CGI server to handle the Http. As a CGI application the Main class receives the URL path and the query from the environment. The Main class creates then the Request and Response classes. The request class contains request information. On the other hand the Response contains methods that create a reply to the CGI server.

It follows a callback pattern where the `Router` class is expected to process the `Request` and decide which `Route` should handle the request. Then it should call the appropriate route method. A concrete implementation of the `Router` class is shown in listing 32.

The different levels of abstraction can be explained by comparing different `Router` implementations.

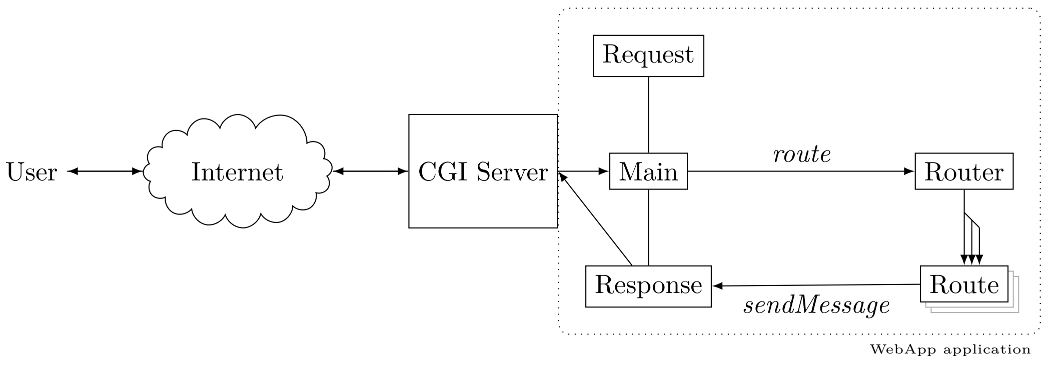


Figure 39: A web application overview.

A.2.1 Router abstractions

A.2.1.1 Concrete implementation concrete implementation of a `Router` routes the request to a `Route` class that matches on the `Route` path.

```
package loupe.examples.webapp.core;

import loupe.examples.utils.Logging;

public class Router {
  private static final Logging LOG = new Logging();
  private int routeCount = 0;
  private RouterItem[] items = new RouterItem[512];

  public void addRoute(String format, Route route) {
    items[routeCount] = new RouterItem(format, route);
    routeCount++;
  }

  public void route(Request req, Response res) {
    for (int i = 0; i < routeCount; i++) {
      RouterItem item = items[i];
      if (req.getPath().equalToString(item.format)) {
        item.route.route(req, res);
        return;
      }
    }
    LOG.warning("Route␣not␣found␣%s", req.getPath());
  }

  private class RouterItem {
    public String format;
    public Route route;

    public RouterItem(String format, Route route) {
      this.format = format;
      this.route = route;
    }
  }
}
```

Listing 32: Concrete implementation of the Router class.

A.2.1.2 The shallow abstraction such abstraction only needs to ensure that its return value conforms to the return type. Since the return types of both method is `void` the inner bodies will be replaced by no ops.

```
package loupe.examples.webapp.core;

import loupe.examples.utils.Logging;

public class Router {
  public void addRoute(String format, Route route) {
  }

  public void route(Request req, Response res) {
  }
}
```

Listing 33: Naïve implementation of a Router abstraction.

A.2.1.3 A deep abstraction such abstraction should allow a call to the `route` method missing from the naïve implementation. It over approximates any objects that could have multiple values (in this example the `route` variable) by creating a new object of a specific type. The implementation is shown below:

```
package loupe.examples.webapp.core;

import loupe.examples.webapp.routes.FindRoute;
import loupe.examples.webapp.routes.FormRoute;
import loupe.examples.webapp.routes.HelpRoute;
import loupe.examples.webapp.routes.InfoRoute;
import static com.lexicalscope.svm.j.instruction.symbolic
                    .symbols.SymbolFactory.*;
```

```
public class Router {
  public void addRoute(String format, Route route) {
  }

  public void route(Request req, Response res) {
    switch (selectState(4)) {
      case 0:
        new FindRoute().route(req, res);
        break;
      case 1:
        new FormRoute().route(req, res);
        break;
      case 2:
        new HelpRoute().route(req, res);
        break;
      default:
        new InfoRoute().route(req, res);
        break;
    }
  }
}
```

Listing 34: Simple implementation of a Router abstraction.

A.2.1.4 A mnemonic abstraction such abstraction preserves the context a class has between method calls. It uses the `SymbolFactory` method in order to capture the state of method calls. In addition it preserves the original control flow. The implementation is shown below:

```
package loupe.examples.webapp.core;

import static com.lexicalscope.svm.j.instruction.symbolic
                .symbols.SymbolFactory.*;

public class Router {
  private ArrayList<String> strings = new ArrayList<>();
  private ArrayList<Route> routes = new ArrayList<>();

  public void addRoute(String format, Route route) {
    strings.add(format);
    routes.add(route);
  }

  public void route(Request req, Response res) {
    do {
      if (randomChoice()) {
        break;
      }
      if (randomChoice()) {
        Route route = getPassedParameter(routes);
        route.route(req, res);
        return;
      }
    }
  }
}
```

Listing 35: Precise implementation of a Router abstraction.

The automated algorithm makes the code much more complicated than necessary. Manually it could be optimised to the following code:

```
package loupe.examples.webapp.core;

import static com.lexicalscope.svm.j.instruction.symbolic
                 ].symbols.SymbolFactory.*;

public class Router {
  private ArrayList<String> strings = new ArrayList<>();
  private ArrayList<Route> routes = new ArrayList<>();

  public void addRoute(String format, Route route) {
    strings.add(format);
    routes.add(route);
  }

  public void route(Request req, Response res) {
    if (randomChoice()) {
      Route route = getPassedParameter(routes);
      route.route(req, res);
    }
}
```

Listing 36: Manually written precise implementation of a Router abstraction.

A.3 Even change example

This example shows a case where creating an abstraction causes a false. The reason for this is that the bd spec violation happens on a path which cannot happen under the execution of a real program.

An even change program is a simple program that doubles a value and prints a message. The introduced change changes the message printed when the value is not even. The corresponding `EvenMain` classes are shown in listings 38 and 41. The `DoubleValue` class is shown in listing 39.

```
package loupe.examples.benchmark.falsepositive.evenchange;

import loupe.examples.utils.Logging;

public class EvenMain {
    private Logging LOG = new Logging();

    public static void main(int[] values) {
        new EvenMain().run(values);
    }

    private void run(int[] values) {
        if (new DoubleValue().doubleValue(values[0]) % 2 == 0) {
            LOG.info("should␣be␣even");
        } else {
            LOG.info("I␣don't␣even");
        }

    }
}
```

Listing 37: An old version of the EvenMain class.

```java
package loupe.examples.benchmark.falsepositive.evenchange;

import loupe.examples.utils.Logging;

public class EvenMain {
    private Logging LOG = new Logging();

    public static void main(int[] values) {
        new EvenMain().run(values);
    }

    private void run(int[] values) {
        if (new DoubleValue().doubleValue(values[0]) % 2 == 0) {
            LOG.info("should␣be␣even");
        } else {
            LOG.info("I␣don't␣even␣(%d)", 12);
        }

    }
}
```

Listing 38: A new version of the EvenMain class.

```java
package loupe.examples.benchmark.falsepositive.evenchange;

public class DoubleValue {
    public int doubleValue(int entry) {
        return 2 * entry;
    }
}
```

Listing 39: A DoubleValue implementation.

Under normal execution doubling an integer should never cause a number to become odd. Thus even though the `EvenMain` class was changed the output sent to the `Logging` class should not change. Therefore the bd spec shown in listing 40 would be applicable.

```java
package loupe.examples.benchmark.falsepositive;

import com.lexicalscope.svm.partition.spec.BehaviourChanged;
import com.lexicalscope.svm.partition.spec.BehaviourUnchanged;
import com.lexicalscope.svm.partition.spec.CallContext;
import com.lexicalscope.svm.partition.spec.ChangeSpecification;
import loupe.examples.benchmark.falsepositive.evenchange.EvenMain;
import loupe.examples.utils.Logging;
import org.hamcrest.Matcher;

import static com.lexicalscope.svm.partition.spec.MatchersSpec.*;
import static org.objectweb.asm.Type.getInternalName;

public class EvenChange implements ChangeSpecification {
    @BehaviourChanged
    public Matcher<? super CallContext> mainHasChanged() {
        return receiver(klassIn(getInternalName(EvenMain.class)));
    }

    @BehaviourUnchanged
    public Matcher<? super CallContext> loggingHasNotChanged() {
        return receiver(klassIn(getInternalName(Logging.class)));
    }
}
```

Listing 40: Bd spec for the EvenChange example.

A.3.1 Shallow abstraction

In this case the concrete implementation does not perform any method calls. Therefore the only thing that needs to be abstracted is the return value. Without performing an even/odd analysis the abstracted `DoubleValue` class would look like the listing 41.

Because the abstraction returns a symbolic integer the previously unreachable path suddenly becomes reachable. Since the code was modified only on the unreachable path a false positive is triggered. Nevertheless, this example seems contrived.

```
package loupe.examples.benchmark.falsepositive.evenchange;

import static com.lexicalscope.svm.j.instruction.symbolic
                  .symbols.SymbolFactory.*;

public class DoubleValue {
    public int doubleValue(int entry) {
        return newIntSymbol();
    }
}
```

Listing 41: Abstracted DoubleValue class.

B Appendix: Raw results

This section contains Raw output generated by the tool.

Legend:

- *Time* – total time spent by svm • *ST* – number of states that completed execution • *QS* – unique queries sent to SMT solver • *F* – violation detected

	Without instrumentation [Tree]				**With instrumentation [LOC]**				**Without instrumentation [Tree]**				**With instrumentation [LOC]**			
Name	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**
C1_1	6	1	0	T	6	1	0	T	6	1	0	T	6	1	0	T
C1_2	286	0	2	T	241	1	2	T	268	1	0	T	280	1	0	T
C1_3	935	25	67	T	618	17	44	T	978	25	70	T	687	17	43	T
C1_4	60000	264	5761	F	60000	341	5986	F	60000	234	5439	F	60000	350	6266	F
C1_5	60000	58	5646	F	60000	4	5990	F	60000	81	5312	F	60000	4	5800	F

Table 6: C1 examples benchmark results.

	Non abstract				**Shallow**				**Deep**				**Mnemonic**			
Name	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**
C2_1	1351	23	64	T	35	4	0	F	163	1524	0	T	177	1546	0	T
C2_2	2423	30	190	T	32	16	0	F	60000	23049	0	F	60000	42918	0	F
C2_3	60000	176	4416	F	60000	416	6146	F	2172	55	178	T	1260	30	99	T
C2_4	60000	244	5390	F	60000	14391	0	F	61	97	3	T	42	43	2	T
C2_5	326	0	2	T	11	2	0	F	12	2	0	F	3231	78	78	T
C2_6	60000	39	4776	F	60000	29679	3074	F	60000	83137	0	F	1545	4574	0	T
C2_7	60000	50	4312	F	60000	17640	38	F	60000	32628	22	F	60000	126533	38	F
C2_8	60000	58	4160	F	60000	27545	2104	F	60000	113963	0	F	60000	129664	0	F
C2_9	60000	32	4098	F	60000	45627	26	F	60000	53725	30	F	60000	130820	40	F

Table 7: C2 examples benchmark results for tree strategy.

	Non abstract				**Shallow**				**Deep**				**Mnemonic**			
Name	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**
C2_1	638	17	44	T	28	4	0	F	174	1565	0	T	149	1663	0	T
C2_2	2475	25	172	T	30	16	0	F	60000	31706	0	F	60000	32605	0	F
C2_3	60000	280	5122	F	60000	513	6238	F	1677	43	121	T	807	23	65	T
C2_4	60000	361	6146	F	60000	17048	0	F	62	100	3	T	39	40	1	T
C2_5	325	1	2	T	12	2	0	F	14	2	0	F	2969	73	64	T
C2_6	60000	4	5598	F	60000	35578	2968	F	60000	107958	0	F	6007	13057	0	T
C2_7	60000	4	4886	F	60000	26235	26	F	60000	37582	26	F	60000	122392	22	F
C2_8	60000	4	5122	F	60000	28728	3082	F	60000	88098	0	F	60000	141716	0	F
C2_9	60000	4	5020	F	60000	47914	32	F	60000	47796	30	F	60000	117242	30	F

Table 8: C2 examples benchmark results for LOC strategy.

	Non abstract				**Shallow**				**Deep**				**Mnemonic**			
Name	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**
C3_1	74	2	0	F	74	2	0	F	65	2	0	F	61	2	0	F
C3_2	255	2	2	F	204	0	3	T	182	0	2	T	168	0	3	T
C3_3	30000	95	2050	F	30000	163	2216	F	30000	1439	2200	F	30000	1362	2080	F
C3_4	15000	34	1138	F	15000	8216	1026	F	15000	33867	0	F	15000	40250	0	F
C3_5	15016	41	1380	F	15870	7292	882	F	15130	35435	0	F	15003	48018	0	F

Table 9: C3 examples benchmark results for tree strategy.

	Non abstract				**Shallow**				**Deep**				**Mnemonic**			
Name	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**	**Time**	**QS**	**ST**	**F**
C3_1	TO	2	0	F	65	2	0	F	65	2	0	F	68	2	0	F
C3_2	237	2	2	F	196	0	3	T	194	0	3	T	180	0	3	T
C3_3	30000	97	1832	F	30000	266	2788	F	30000	2589	2794	F	30000	1661	1966	F
C3_4	15000	4	1278	F	15000	10957	1026	F	15000	23806	0	F	15000	27807	0	F
C3_5	15022	4	1476	F	16341	5689	1026	F	18954	31409	0	F	15006	52865	0	F

Table 10: C3 examples benchmark results for LOC strategy.

References

[1] Zuoning Yin et al. "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.

[2] Napol Rachatasumrit and Miryung Kim. "An empirical investigation into the impact of refactoring on regression testing". In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE. 2012, pp. 357–366.

[3] Tim Wood and Sophia Drossopoulou. "Program Equivalence through Trace Equivalence". In: *Foundations of Object Oriented Languages, FOOL*. 2014.

[4] Tim Mackinnon, Steve Freeman, and Philip Craig. "Endo-testing: unit testing with mock objects". In: *Extreme programming examined* (2000), pp. 287–301.

[5] James C King. "Symbolic execution and program testing". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[6] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT&Mdash;a Formal System for Testing and Debugging Programs by Symbolic Execution". In: *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 234–245. DOI: 10.1145/800027.808445. URL: http://doi.acm.org/10.1145/800027.808445 (visited on 01/17/2015).

[7] Cristian Cadar, Daniel Dunbar, and Dawson R Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[8] Ondrej Lhoták. "Spark: A flexible points-to analysis framework for Java". In: (2002).

[9] Thomas Lengauer and Robert Endre Tarjan. "A fast algorithm for finding dominators in a flowgraph". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 1.1 (1979), pp. 121–141.

[10] Thomas Ball et al. "Thorough static analysis of device drivers". In: *ACM SIGOPS Operating Systems Review*. Vol. 40. 4. ACM. 2006, pp. 73–85.

[11] Yannick Welsch and Arnd Poetzsch-Heffter. "Verifying backwards compatibility of object-oriented libraries using Boogie". In: *Proceedings of the 14th Workshop on Formal Techniques for Java-like Programs*. ACM. 2012, pp. 35–41.

[12] Jerome Miecznikowski. "New algorithms for a java decompiler and their implementation in soot". PhD thesis. McGill University, 2003.

[13] Raja 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*. IBM Press. 1999, p. 13.

[14] Oracle. *CodeModel project*. https://codemodel.java.net/. 2015.

[15] Suzette Person et al. "Differential symbolic execution". In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM. 2008, pp. 226–237.

[16] Patrice Godefroid. "Compositional dynamic test generation". In: *Acm Sigplan Notices*. Vol. 42. 1. ACM. 2007, pp. 47–54.

[17] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. “Demand-driven compositional symbolic execution”. In: *Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2008, pp. 367–381.

[18] David Currie et al. “Embedded software verification using symbolic execution and uninterpreted functions”. In: *International Journal of Parallel Programming* 34.1 (2006), pp. 61–91.

[19] Mary Jean Harrold and ML Souffa. “An incremental approach to unit testing during maintenance”. In: *Software Maintenance, 1988., Proceedings of the Conference on.* IEEE. 1988, pp. 362–367.

[20] Robert B Evans and Alberto Savoia. “Differential testing: a new approach to change detection”. In: *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers.* ACM. 2007, pp. 549–552.

[21] Shuvendu K Lahiri, Kapil Vaswani, and C AR Hoare. “Differential static analysis: opportunities, applications, and challenges”. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research.* ACM. 2010, pp. 201–204.

[22] Samuel Bates and Susan Horwitz. “Incremental program testing using program dependence graphs”. In: *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM. 1993, pp. 384–396.

[23] Carsten Görg and Peter Weißgerber. “Error detection by refactoring reconstruction”. In: *ACM SIGSOFT Software Engineering Notes* 30.4 (2005), pp. 1–5.

[24] Xiaoxia Ren et al. “Chianti: a tool for change impact analysis of java programs”. In: *ACM Sigplan Notices.* Vol. 39. 10. ACM. 2004, pp. 432–448.

[25] Wei Jin, Alessandro Orso, and Tao Xie. “Automated behavioral regression testing”. In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on.* IEEE. 2010, pp. 137–146.

[26] Tao Xie et al. “Towards a framework for differential unit testing of object-oriented programs”. In: *Proceedings of the Second International Workshop on Automation of Software Test.* IEEE Computer Society. 2007, p. 5.

[27] Kunal Taneja et al. “Guided path exploration for regression test generation”. In: *Software Engineering-Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on.* IEEE. 2009, pp. 311–314.

[28] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. “On the effectiveness of manual and automatic unit test generation”. In: *Software Engineering Advances, 2008. ICSEA’08. The Third International Conference on.* IEEE. 2008, pp. 252–257.

[29] Daniel Jackson and David A Ladd. “Semantic diff: A tool for summarizing the effects of modifications”. In: *Software Maintenance, 1994. Proceedings., International Conference on.* IEEE. 1994, pp. 243–252.

[30] Koushik Sen, Darko Marinov, and Gul Agha. *CUTE: a concolic unit testing engine for C.* Vol. 30. 5. ACM, 2005.

[31] Trung Dinh-Trong et al. “Looking for More Confidence in Refactoring? How to Assess Adequacy of Your Refactoring Tests”. In: *Quality Software, 2008. QSIC’08. The Eighth International Conference on.* IEEE. 2008, pp. 255–263.

[32] Tim wood. *Java Bytecode Snapshot Vm and Symbolic Executor.* https://github.com/lexicalscope/svm. [Online; accessed 12-June-2015]. 2015.

[33] Nikolai Tillmann and Jonathan De Halleux. “Pex–white box test generation for. net”. In: *Tests and Proofs*. Springer, 2008, pp. 134–153.