# Equivalence Partitioning as a Basis for Dynamic Conditional Invariant Detection

Worakarn Isaratham

Dissertation 2015
Erasmus Mundus MSc in Dependable Software Systems

Department of Computer Science
Maynooth University, Maynooth
Co. Kildare, Ireland

A dissertation submitted in partial fulfilment
of the requirements for the
Erasmus Mundus MSc Dependable Software Systems

Head of Department: Dr Adam Winstanley
Supervisor: Dr Rosemary Monahan
June 9, 2015

Word Count: 21036

## Declaration

I hereby certify that this material, which I now submit for assessment of the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of other save and to the extent that such work has been cited and acknowledged within the text of my work.

Worakarn Isaratham

**Acknowledgements**

First of all I would like to thank my supervisor, Dr Rosemary Monahan, who let me run with my own ideas, and has provided constant support and encouragement throughout the project. I also thank Dr Stephen Brown for his invaluable inputs.

Special thanks to Inthraporn Aranyanak and Professor Ronan Reilly, who helped me settle down during my first days in Ireland.

## Abstract

Program invariants are statements asserting properties of programs at certain points. They can assist developers and testers in understanding the program, and can be used for automated formal verification of the program. However, despite their usefulness they are often omitted from code. Dynamic invariant detection is a technique that discovers program invariants by observing execution of the program. One type of invariants that presents challenge to this technique is *conditional invariants*, which are considered to be computationally infeasible to be computed exhaustively. We present a new approach to assist conditional invariants detection, by analysing test suites used to drive the execution of the programs for their use of *equivalence partitioning* – a very common testing technique – and inferring conditional invariants from this information. A prototype implementation, named *Yacon*, is developed to work in conjunction with a mature dynamic invariant detection tool *Daikon*. Given a set of *splitting conditions*, Daikon can use them to infer conditional invariants. Yacon attempts to recover partitioning information from a given test suite, producing splitting conditions as a result. We introduced two strategies to recover partitioning information, one based on the presence of boundary value analysis testing technique; the other based on invariants within the test suite itself. We evaluated the effectiveness of each recovery strategy and the approach as a whole, and found that our approach can help make Daikon perform significantly better. However, the two recovery strategies only work well in limited circumstances, suggesting possible improvement in finding more effective recovery strategies.

# Contents

# 1 Introduction

Objective: this chapter states the background and motivation of this work, and identifies the research question. The structure of this dissertation is also laid out.

## 1.1 Background

We are living in an age where society is becoming more and more dependent on computers in all aspects of life, ranging from entertainment, transportation, finance, health care, telecommunications and even life-critical systems. One of the great challenges the software industry is currently facing, is to make sure that the computer systems on which our lives depend, are really dependable. We want to know that the software products we make are working as intended.

The traditional way of ensuring that software works as intended is to *test* or *simulate* the software. The industry has been investing huge amount of money and efforts into software testing, and these numbers are on a rising trend. A recent study [13] found that the budget spent on testing has increased from 18% in 2012, to 26% in 2014. Yet despite all the efforts, the final products are not always reliable. On the contrary, software products are constantly shipped with defects. Some are harmless, but others can cause devastating effects. As an example, in August 2012, a trading firm Knight Capital almost went out of business due to a loss of $440 million over a period of 30 minutes, caused by a defect in its trading software [49]. In an even worse catastrophe, during 1985–1987, six people died because of an overdose from the radiation therapy machine Therac-25, due to software error [42]. How could people continue shipping out products with such a huge, life threatening mistake? It is possible that these products were not appropriately tested. But even if they were well-tested, we still cannot rule out a chance of errors, since there is an inherent problem with software testing. In all but the most trivial cases, testing can only '[show] the presence, not the absence of bugs', as Dijkstra famously pointed out [12, p. 16]. No matter how well we test, we can never be 100% certain that a piece of software is free from defects.

An alternative method of ensuring correctness is formal verification. Whereas software testing aims at showing the lack of empirical evidences of errors, formal verification tries to use mathematical reasoning to prove program correctness [36]. Under this approach, it is possible to say that a function behaves correctly for all its possible inputs, without having to exhaustively test for each of them. There exist a lot of different techniques that are classified as formal verification, including well-established approaches such as *theorem proving* and *model checking* [14]. Theorem proving consists of a spectrum of different systems, from highly automated ones to more interactive systems. These techniques express both the software and its specifications as formal languages, then attempt to prove the specifications correct by deducing them from some formal axioms and rules. Model checking [5] builds models of a system, then proves that the desired properties hold for all state spaces.

## 1.2 Motivation

Despite sharing the same goal – to make software more reliable – testing and formal verification were once viewed as incompatible rivals. This perception has changed a lot in recent years, so that the two approaches are now seen as complementary [9]. It is then natural to ask, how can the two approaches help support and improve each other? So we looked at past and current research in this area, and found a number of different ways that this question could be addressed. [9] explores a large amount of number of these works. To name a few:

- Use formal specification to generate test cases.
- Derive test oracle from formal specification.
- Use existing test cases to verify formal specification.
- Use formal specification as a code coverage criterion.
- Derive formal specification from test cases.

While researching these areas, the topic of *dynamic invariant detection* caught our interest. It is a technique that can be classified under 'deriving formal specification from test cases', one of the research direction we explored. The technique works by generating program invariants from actual program executions. In this context, program invariants are properties that always hold at certain points during all program executions. Dynamic invariant detection fits our interest in connecting testing and formal verification together because *a*) it requires observing program execution, which is typically driven by running a *test suite* (a collection of test cases), and *b*) the invariants it generates are useful in a lot of different areas, one of them being *formal verification* [20].

Program invariants are useful in many areas, both to humans and tools. These include software development activities such as design, documentation, coding, testing, formal verification, and maintenance [20]. In ideal situation, programmers would follow formal-method oriented practice, such as *Design by Contract*, and annotate programs with some form of contracts or invariants. Regrettably, that is not the case, as a lot of software developers never use formal method tools and techniques, and those who know about them often consider them to be irrelevant in real-world situations [41]. Therefore the invariants need to be discovered – or rather, 'recovered'. This is the purpose of dynamic invariant detection. If it can be done, then it will be beneficial to many people in the software industry, including developers, designers, and testers. For example, invariants can help avoid bugs introduced by program maintenance [20].

While programmers usually do not use formal methods, in contrast, testing is established as a standard practice [1]. A lot of software already exist with comprehensive test suites. The recent surge in Test-Driven Development pushes this even further, as under this software process, new code will never be written unless a test has failed [4]. The combination of this contrast in popularity with the fact that dynamic invariant detection is usually driven by a test suite, as already stated above, leads us to explore – *could there be some useful information to be harvested from test suites, to assist or improve discovery of invariants?*

## 1.3 Problem Statement

There are 3 parts to this question: '*could there be some useful information to be harvested from test suites, to assist or improve discovery of invariants?*'

1. What aspects of dynamic invariant detection are to be improved?

2. What information is to be extracted from test suites?

3. Can the answers to these two questions complement each other?

For the first part, we found a selection of program invariants that are considered more difficult to detect than others. These are referred to as *conditional invariants*. While invariants typically need to hold for all possible executions, conditional invariants only hold under certain circumstances. These conditional invariants describe properties such as 'if list `l` is empty then function `foo` return 0', or 'the first `n` elements of list `l` are sorted after `n` iterations'. These conditions occurs naturally on any non-trivial programs. They are more difficult to detect because there are so many possible conditional predicates to consider, that trying all of them is computationally infeasible [19]. Clearly, some guidance is needed. Previous attempts at providing policies to help detecting conditional invariants, which will be covered in section 2.3, are mostly some variants of source code or execution traces analysis. None of them, to the best of our knowledge, make use of any information from the test. This makes it a good candidate as our answer to sub-question 1.

For part 2, we focused on extracting information from black-box testing techniques[1], rather than white-box/structural ones, since they are being used more in the industry [31]. We settled on equivalence partitioning, which is a testing technique that recommends testers to divide a domain variable (a method parameter or a class variable) into different partitions – each partition contains data that are processed in the same way, and create one test case for each partition [10]. In other words it advises testers to create one test case for each way a domain variable can *conditionally* be processed, which is a good match for our desire to improve detection of conditional invariants. Additionally, equivalence partitioning is a testing technique used by most testers [15], if not explicitly then informally without even realizing it [30]. In a survey in 2011, it was used by 60.9% of the respondents, being one of the most-used specification-based testing techniques [31].

Part 3 then becomes our main research question. We define the problem statement of this work as answering the research question:

> How can equivalence partitioning assist dynamic detection of conditional invariants?

[1]testing techniques based solely on the specification and not the code

## 1.4 Aims and Objectives

The aim of this project is to study the possibility of using equivalence partitioning to assist dynamic invariant detection in recovering conditional invariants. To accomplish this, the following objectives have to be met:

- The concept of using information from equivalence partitioning to help dynamic invariant detection has to be clarified – what exactly is the information from equivalence partitioning, and how can it be obtained.
- A working prototype has to be implemented as a proof of concept.
- The research question is to be evaluated for its effectiveness, by using the prototype solution.

## 1.5 Outline

The rest of this dissertation is organised as follows:

- Chapter 2 provides the background on researches and theories related to our research question, including equivalence partitioning and dynamic invariant detection.
- Chapter 3 elaborates the design of our solution.
- Chapter 4 details the implementation of the prototype solution, as well as the issues we experienced during development.
- Chapter 5 presents the evaluation of the work in terms of its effectiveness and performance.
- Chapter 6 concludes the work and suggests future research directions.
- The appendices provide more details on mathematical theories used, instructions on building and executing our solution, and the specific settings used in our evaluation.

## 1.6 Summary

This chapter identified the research question. We described the background and motivation that lead us to the research question ‘How can equivalence partitioning assist dynamic detection of conditional invariants?’. To answer this we will build a prototype solution to help us validate and evaluate the question.

# 2 Related Work

Objective: this chapter describes the context of the research question in detail, explaining existing approaches to the problem and how our approach differs.

As discussed in the previous chapter, this dissertation explores how we can use equivalence partitioning testing technique to improve dynamic invariant detection. Here we explore these areas in more details focusing on *a*) software testing techniques and theories related to equivalence partitioning, and *b*) dynamic invariant discovery techniques. We then focus on the problem of conditional invariant detection and how it is being handled so far.

## 2.1 Equivalence Partitioning and Related Testing Techniques

This section focuses on testing techniques related to equivalence partitioning, and theoretical background supporting these techniques.

### 2.1.1 Testing Techniques and Test Selection Hypothesis

First we consider the reason for testing techniques to exist.

The objective of software testing is to test that a piece of program behaves correctly *for all possible inputs*. Without testing techniques, testers do not have other choices but to test all the inputs in order to meet the objective. This is known as *exhaustive testing*. It is considered to be generally impossible, except for trivial cases [30]. Some input domains are infinite, thus testing all values is clearly impossible. Even for finite domains it is considered infeasible, since domains are usually too large that it would take too much time to test for every input [10].

As a result of the impossibility of exhaustive testing, testing techniques are needed to make it feasible. Every technique relies on some heuristics to select only a finite test set from the possibly infinite input and output domains. These heuristics are based on assumptions, such that if they hold true, they will produce a resulting test set that has good *bug-revealing power*. Such assumptions are known as *test selection hypotheses* [25, 28].

### 2.1.2 Domain-Based Testing

A *domain* of a variable is defined as a 'set that includes all possible values of a variable of a function' [39, p. 36].

A family of testing techniques bases their test selection hypotheses on properties of domain variables. We use the term 'domain-based testing techniques' to call any techniques with this property. Two of the most well-known such techniques are *equivalence partitioning* and *boundary value analysis*. Both are included in most of modern software testing textbooks [10, 30, 35, 38, 46] and standards [1, 55], often as the very first specification-based technique. They are also cited as 2 of the most used techniques in the black-box category, with more than 60% adoption each [31]. In the next sections we discuss about these 2 techniques, as equivalence partitioning is one of the main focus or the research question, and boundary value analysis is very closely related to equivalence partitioning and will be used in our prototype solution.

### 2.1.3 Equivalence Partitioning

*Equivalence partitioning*[1] (EP) is a black-box testing technique, meaning that it works on the specification of programs rather than the source code. Its test selection hypothesis is that the input and output domains can be partitioned into *equivalence classes*, where all members of the same class should behave in the same way. Therefore, only a single member of a class is sufficient to represent the whole class. If the system pass the test on a sample, then it is assumed that the system works correctly for all inputs from the class that test sample belongs to. Conversely, if the test fails, then it is assumed that it will fail for all members of that class.

[1]also called: *equivalence partition testing, equivalence class partitioning, equivalence class testing, equivalence classing*

The test selection hypothesis described above is known as the *uniformity hypothesis*. Gaudel [26] formally defined it as follows:

**Definition 2.1.** Given a specification *SP* and a system under test *SUT*.

$$\bigcup_{1 \leq i \leq p} UTS_i = \texttt{Exhaust}(SP), \textit{ and}$$

$$\forall i : 1 \leq i \leq p \, \forall t : t \in UTS_i, SUT \, \texttt{passes} \, t \Rightarrow SUT \, \texttt{passes} \, UTS_i$$

where $\texttt{Exhaust}(SP)$ is the exhaustive test set based on the specification, and $UTS_i$ are equivalence classes.

Equivalence partitioning has its root in the concepts of *equivalence relation, equivalence class,* and *partition*, in mathematics[2]. Given these concepts, the uniformity hypothesis can then be seen as the assumption of the existence of an equivalence relation that divides input and output domains into partitions.

By the mathematical definition of partition, every member of a domain must be in an equivalence class, and no two equivalence classes can overlap. These are two properties that are desirable when applied to software testing. The first property gives us a satisfying sense of completeness – that the entire test set is well covered. The second means that there is no redundancy in test cases [38]. However, the definition of the uniformity hypothesis as formally stated previously does not require the non-overlapping property to hold. This reflects the fact that overlapping partitions are common in practice [52]. Therefore, the resulting products of equivalence partitioning are not partitions, in the strict mathematical sense. Rather, to be accurate they should be called sub-domains, and the technique should be called 'sub-domain testing' instead [32, 52]. Nevertheless, since 'equivalence partitioning' is a long-standing term being regularly used in textbooks and standards, we will continue using it in this dissertation. The word '*partition*', however, can be ambiguous since it can be used to refer to both a single sub-domain and the set of all sub-domains. For clarity, for the rest of this dissertation we use the terms in the mathematical sense, as defined in appendix A.1. A single sub-domain is called an **equivalence class**, or simply a **class**, and the word **partition** means a set of equivalence classes.

In the context of our research question, each equivalence class when considered individually should yield its own unique set of invariants. We can say that these invariants are also true for the entire domain, if each of them is contextualised by a conditional statement. In other words,

[2]See appendix A.1 for definitions

*the existence of equivalence classes implies conditional invariants.* Since equivalence partitioning advises testers to create test cases according to these equivalence classes, it is reasonable to see test suites written using this technique as a source of conditional invariants.

Equivalence partitioning is also a very well-known testing technique included in most software testing textbooks, is mandatory knowledge for the most basic software testing certification [30], and is cited as one of the most used techniques in the black-box category [31]. It is also said to be used by most testers even without realizing [30]. Craig and Jaskiel stated that '[it] is a technique that is intuitively used by virtually every tester we've ever met.' [15, p. 162] It follows that given an existing test suite, there is a high probability that equivalence classes do exist in some form – with formal documentation or otherwise, in the test cases design stage, which is a good thing for our intended purpose of recovering the partitioning information.

### 2.1.4 Boundary Value Analysis

Boundary value analysis (BVA) is another domain-based testing technique closely related to equivalence partitioning. It is based on the same test selection hypothesis – uniformity hypothesis, assuming that input and output domains can be divided into equivalence classes. It adds another assumption that *different members of a class have different bug-revealing power.* It argues that errors tend to occur at or near the edge of the classes (the *boundary values*), because in most cases programmers have to explicitly write a conditional statement (an `if` clause or a loop) for each boundary [38]. Since every program statement can introduce bugs, mistakes in these conditions might produce errors at the extreme values. Therefore the boundary values have more bug-revealing power than other values in the same equivalence class – in other words, among all members of a domain, it is more probable to find an error at the boundary values. The conclusion is that boundary values should be preferred when selecting a class representative.

As an example, a common mistake is the use of wrong inequality operator, such as `<=` and `<`. Consider a function containing an integer domain `x` and a condition using inequality operator, `x < 10`. In this case, the boundary values of `x < 10` are 9 and 10, because they are two consecutive values that fall on different sides of the condition. The technique of boundary value analysis advises testers to write a test case for each of these two values. Suppose further that the condition `x < 10` is mistakenly written as `x <= 10`, which might occur due to a logical error on the programmer's part, or could be simply a typo. It can be seen that the only value of `x` that can reveal the bug is 10, which is covered by the test cases developed using boundary value analysis technique.

It can also be seen that test suites developed using boundary value analysis can be useful in the context of the research question. In particular, the boundary values can more accurately point out the condition part of conditional invariants.

We have discussed equivalence partitioning and its related concepts, which forms the first half of our research question. Next we consider the other half, dynamic invariant detection and conditional invariants.

## 2.2 Dynamic Invariant Detection

Dynamic invariant detection is a technique used to recover invariants dynamically by observing program execution. An *invariant* is a property that always hold at certain points during all program executions [22]. They include method preconditions, method postconditions, class invariants, among others. The following statements are examples of invariants: $y = 5 \times x - 1$, $x \times y > 1$, 'list $a$ is sorted', and 'graph $g$ contains no cycles' [20]. These invariants are useful information that can help programmers and tools in various software development activities, including design, documentation, coding, testing, formal verification, and maintenance [20]. Dynamic invariant detection use real program execution to deduce the invariants, as opposed to static program analysis, such as symbolic execution [40] or model checking [5], which uses inspection of source code or binary code to arrive at the invariants. One advantage of dynamic techniques is that the source code of target program is not required. There are 2 steps in dynamic invariant detection [20]:

1. Collecting trace data of the target program, and
2. Infer properties from the collected trace data.

The rest of this section presents a number of existing dynamic invariant detection tools, and the reasons for us to select one of them for our prototype solution.

### 2.2.1 Daikon

Daikon [21,22] is the first and primary tool in the area of dynamic invariant detection. It was first developed in 1998, and has been in continuous improvement and maintenance ever since, with a new stable release every month. Daikon pre-calculates possible invariants at various program points, then runs the target program, observes the values, and eliminates invariants that do not conform with the executions. What remains are the discovered invariants.

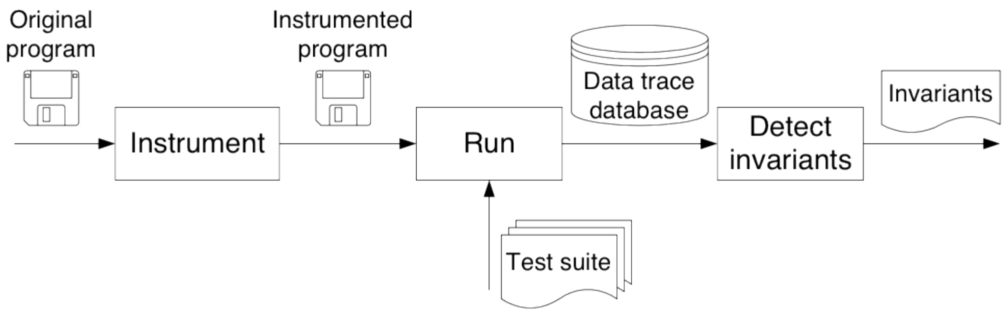

Figure 2.1: Daikon flow [20]

The workflow of Daikon is shown in figure 2.1. First, a subject program is *instrumented* to be able to produce information necessary for analysis. The resulting program is then exercised, typically by running a *test suite*, under control of a *front end* program. This produces a trace database, which is passed on to be analysed by the *invariant detector*. Finally the detector reports its finding – the detected invariants of the subject program.

### 2.2.2 Agitator

Agitator [8] is a commercial product by Agitar software. The main purpose of Agitator is to detect dynamic invariants ('agitate') in order to inform users about tests, to enable them to write better test code. The product focuses on invariants of the following type: equality, range, and null checking.

### 2.2.3 DIDUCE

DIDUCE [34] is a highly scalable invariant detection tool with the main purpose to find software bugs. It works on-line, that is, it detects the invariants as the target program is being executed. DIDUCE is specialised in detecting unexpected violation of invariants. The tool clearly separates its operations into two modes: the training mode deduces all possible values of the invariants, and the checking mode detects cases where the invariants are violated – i.e. new values are observed. The tool is especially helpful in detecting and explaining difficult-to-reproduce bugs.

### 2.2.4 DySy

DySy [16] is a tool for dynamic invariant detection that, in addition to dynamically execute the given programs, it also uses symbolic execution to enhanced the quality of dynamically inferred invariants. DySy differs form Daikon in the source of invariants templates – while Daikon use predefined templates, DySy derives them from symbolic execution.

### 2.2.5 IODINE

IODINE [33] is a dynamic invariant detector that discovers design properties of hardware designs. The invariants it produces are in the form of state machine protocols, request-acknowledge pairs, and mutual exclusion between signals. It executes test vectors on a simulation of the real hardware, to learn about any resulting properties emerged from the design.

## Tool Selection

We analysed these tools and decided to base our prototype implementation on the tool Daikon, because of the following reasons:

- It is the most mature one, with active development and support.
- It works generally for any types of invariants instead of being specialised to only specific types.
- It is open-source, which enables us to inspect the inner working of the tool.
- It can analyse programs written in many programming languages such as Java, C, C++, C#, and Perl. It can also be extended to work on other languages easily, by following the process described in the developer's manual [2] to develop a front end specifically for that language.
- It has built-in support for conditional invariants – the subject of our research question.

Now we turn to conditional invariants, to see why they are difficult to detect, how Daikon currently detects them, and how our prototype solution can fit into the picture.

## 2.3 Conditional Invariant Detection

Conditional invariants are properties in the implication form $p \Rightarrow q$. Disjunction can also be considered a form of implication, because of the identity $p \lor q \equiv \neg p \Rightarrow q$. While program invariants need to hold true for all circumstances, the consequence part of conditional invariants needs to be true only under certain conditions. This type of invariants arises naturally in any programs with conditional statements, such as `if` and `while` in Java.

Since trying all combinations of predicates is computationally prohibitive [21], Daikon relies on predicates called *splitting conditions* to split the execution trace of a programs into parts. For each splitting condition, it divides the executions into two groups – one that satisfies the condition and the other that does not. The two groups are then processed separately, to detect invariants on each side. These invariants, combined with the splitting conditions, are then inferred to produce implications.

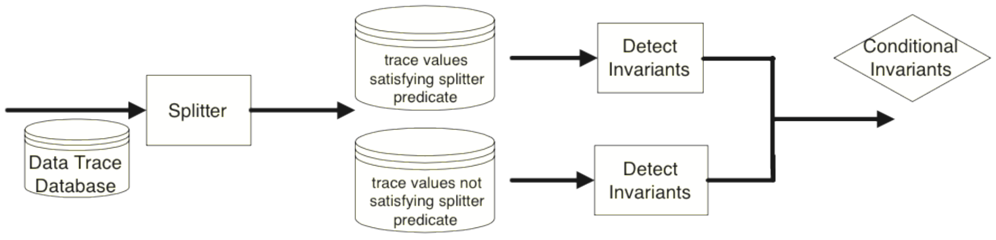
Data Trace Database
Splitter
trace values satisfying splitter predicate
trace values not satisfying splitter predicate
Detect Invariants
Detect Invariants
Conditional Invariants

Figure 2.2: Detecting conditional invariants through splitting conditions [18]

The quality of the resulting conditional invariants depends on how good the splitting conditions are. For example, if a condition creates two groups that produces the exact same invariants, then that condition is completely useless. Strategies to select good splitting conditions are needed.

For Daikon, these strategies are called *splitting policies.* They are described as splitter files (`.spinfo`), which can be created manually, or by using some other means. The standard distribution of Daikon provides a number of tools to accomplish this task:

**Procedure Return Analysis.** This is the built-in mechanism Daikon provides. It looks at procedures that return binary values, and creates splitting conditions based on the return values. Another source of conditions is from procedures that have multiple return statements [18].

**Static Analysis.** This approach takes all boolean conditions in the program, such as the `if` and loop statements, as the source of splitting conditions. To use this policy, run the `daikon.tools.jtb.CreateSpinfo` command line program. The command takes Java source files as arguments and create `.spinfo` files as its output. [3]

**Cluster Analysis.** This is a statistical analysis approach to divide execution traces into clusters, where similar data points are grouped together. To use this policy, run the `runcluster.pl` Perl script. The command receives execution trace files (`.dtrace`) and produce `.spinfo` files as its output. Three clustering algorithms are currently supported: k-means, x-means and hierarchical. [3]

**Random Sampling.** This is done by randomly creating a number of subsets of data, find invariants in each subset, and compare the result with the invariants in the overall dataset. Invariants presented in any of the subsets but not in the overall can be taken as splitting conditions. This method works better for unbalanced data since the chance of them being discovered would be higher [19]. To use this policy, run the `daikon.tools.TraceSelect` command line program, with the number of subsets, number of runs, and execution trace files as parameters. [3]

Furthermore, there exists a number of methodologies other than those distributed with Daikon.

**Data Mining.** Two standard data mining techniques, *Association Rule Mining* and *Decision Tree Learning*, have also been proposed to be used to generate splitting conditions from execution traces [23]. No analysis on the effectiveness of this approaches are publicly available.

**Sub-Cases Analysis.** This is a log-based technique; the program under analysis is instrumented to produce logs. These logs are partitioned into sub-cases, each of the sub-case can then be passed to Daikon to infer the pre- and post-conditions. The research [50] did not explicitly structure the resulting conditions as conditional invariants, but it can be done in a trivial fashion.

For our research question of using equivalence partitioning information to assist detection of conditional invariants, since Daikon has provided a way for users to specify the splitting condition in order to help detecting conditional invariants, in the form of `.spinfo` files, the task of our prototype implementation is reduced to creating a new splitting policy to produce these files from equivalence partitioning information.

## 2.4 Summary

In this chapter we introduced the testing technique of equivalence partitioning and related concepts. We also discussed dynamic invariant detection and the current tools in this research area, and the specifics of conditional invariants. We conclude this chapter by stating that it should be possible to extract information necessary for detecting conditional invariants from equivalence partitioning, because of the following reasons:

- Equivalence partitioning implies in-class invariants, as discussed in page 9.
- Equivalence partitioning is ubiquitous in test suites, as discussed in page 10.
- Test suites are used to drive Daikon. To perform its analysis, Daikon needs multiple trace runs of the subject program. While it is not required so, it is most intuitive to drive the execution from a test suite, as implied from the flow diagram (figure 2.1).

Combine these observations together, we might conclude that for any existing Daikon's subject program, it is highly probable that invariants information should already exist in its driving test suite, hence our proposal to extract invariants from equivalence partitioning.

# 3 Solution Design

Objective: this chapter presents the design of the prototype solution to answer the research question of using equivalence partitioning to assist dynamic invariant detection.

We describe in this chapter the detailed design of *Yacon*[1], our prototype solution.

## 3.1 High-Level Design

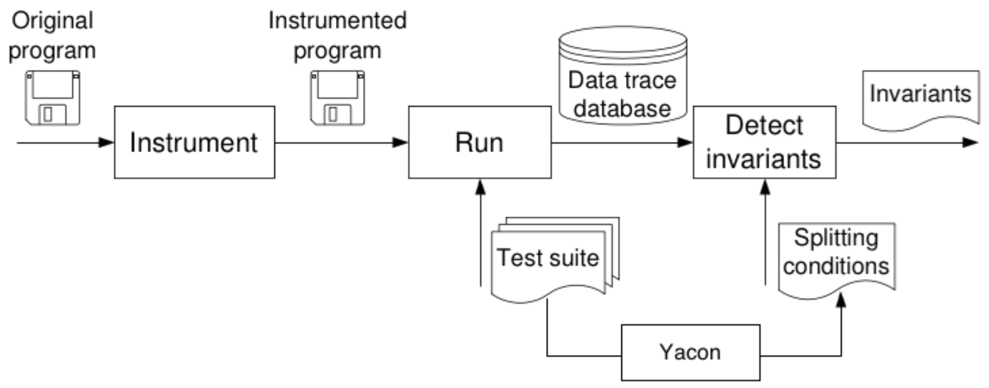

Figure 3.1: Integration of Yacon with Daikon

[1]The name is a reference to the vegetable *yacón*, to conform with Daikon's conventional practice of naming things after root vegetables!

In the previous chapter, we decided to base our prototype solution on **Daikon** since it is mature, well-researched, and well-supported by active development communities. We also identified that the purpose of our solution is to draw equivalence partitioning information from test suite, in order to produce *splitting conditions*, in the form of Daikon's `.spinfo` files. In this design, Yacon exists as an independent component of the Daikon system, as shown in figure 3.1.

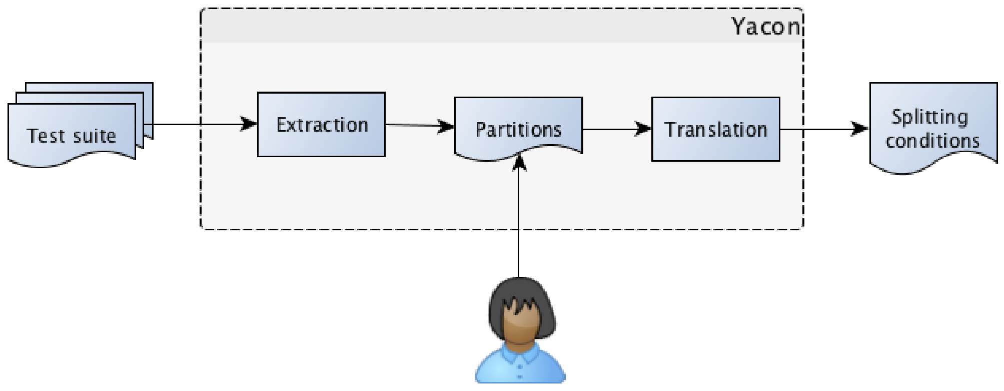

Figure 3.2: High-level flow diagram of Yacon

The work of Yacon is divided into two phases, *a*) *extraction* and *b*) *translation*. An intermediate file, which will be in a format we called *partitioning file format*, will be used to communicate partitioning information from the extraction phase to the translation phase, as depicted in the flow in figure 3.2.

The use of intermediate files as information medium, instead of passing it directly in-memory, is a design choice we settled upon because we recognised that extraction will be implemented using a heuristic-based approach. Therefore, the partitioning information produced would not be perfect. Using intermediate files permits users to inspect, verify, and modify the partitioning information.

To be precise, the two phases of Yacon have the following responsibility:

1. The **extraction phase** extracts partitioning information from test suites, and create a partitioning file.

2. The **translation phase** processes the partitioning file, producing splitting conditions.

This is summarised in the use case diagram in figure 3.3.

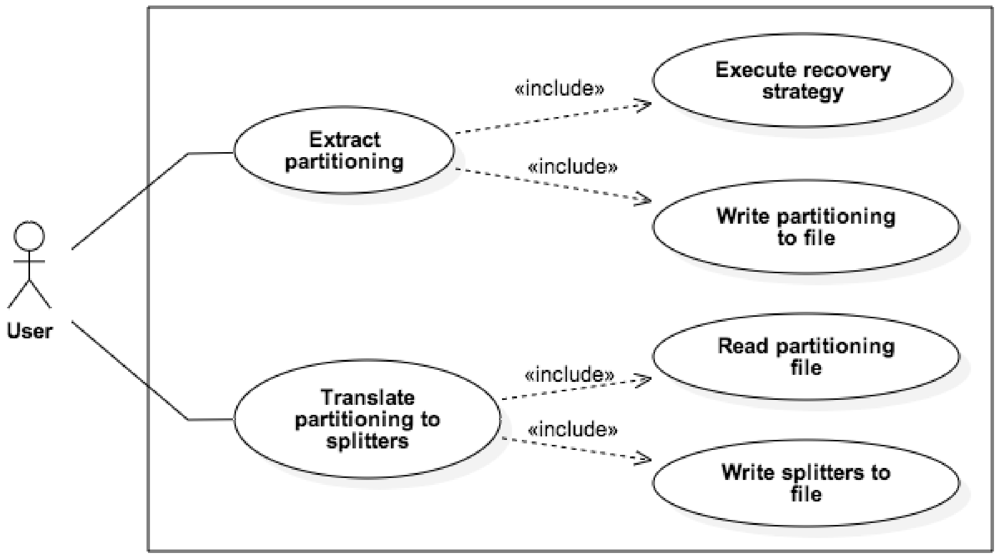

Figure 3.3: Use case diagram of Yacon

The rest of this chapter explores the design of each component of Yacon, starting with the extraction phase (section 3.2), following by the partitioning files (section 3.3), and then the translation phase (section 3.3).

## 3.2 Extraction Phase

This phase is responsible for extracting partitioning information from test suites, and passed it on to translation phase, in the form of a partitioning file. This section expands on two parts: first we clarify the concept of 'partitioning information', then we describe two *recovery strategies* – methods to recover partitioning information from test suites.

### 3.2.1 Partitioning Information

Partitioning information consists of the following :

1. A set of all the domain variables within the target program being subjected to partitioning.
2. The partition of each domain variable. By definition, a partition is a set of disjoint subsets, known as equivalence classes, with their union equals to the domain.

In simple terms, partitioning information describes how the domain variables are to be divided into equivalence classes. The set of domains is straightforward to represent, while partitions are more complex. To find a way to express the equivalence classes, we utilise the facts that $a$) an equivalence class is a set, and $b$) every set can be defined by its *builder predicate*. For example, for the set of all even integers, its builder predicate is $P(x) = [x \in \mathbb{N} \wedge \exists y \in \mathbb{N} : x = 2y]$. More generally, a set $s = \{x|P(x)\}$ has $P(x)$ as its *builder predicate*.

Since there is no limitation to how builder predicates can be defined, the partitioning file format should support any general predicates. At the same time, we also recognise from the standard practice and textbooks on equivalence partitioning, that a few special cases of equivalence classes do arise frequently, such that they deserve special treatment separate from normal builder predicates. We explore two such cases here.

**Interval-based equivalence class** For an enumerable single-dimension domain, such as integers or floating point numbers, a usual way to partition the domain into classes is to divide it into intervals. For example, an integer domain could be divided by the sign into *negative*, *zero*, and *positive* classes. The partitioning in this case is $\{[Min..-1], \{0\}, [1..Max]\}$, which is indeed an interval-based partitioning.

**Complementary equivalence class** This is the class containing all domain members that do not belong to any other classes in the same domain.

The textual representation of partitioning information in the partitioning file format, as will be described later in section 3.3, must support these two special cases of equivalence classes, as well as any generic classes.

In the next part we described two strategies to extract partitioning information from test suites.

### 3.2.2 Recovery Strategies

One of our assumptions is that testers use equivalence partitioning technique when writing tests, either consciously so or otherwise. The objective of this phase is to recover the intents of the testers from written test suites. We recognise that in the process of converting intents into test cases, there is likely be loss of information – the intent is not be retained in full. This implies that it is generally impossible to recover the information perfectly. Any recovery strategies need to be based on some more assumptions on the properties of the test suites, such as the techniques and styles that the testers follow. The result of these strategies would be *potential* equivalence classes. Here we explore two strategies: *a*) boundary value recovery strategy, and *b*) test suite invariants recovery strategy.

#### 3.2.2.1 Boundary Value Recovery Strategy

This strategy is based on the boundary value analysis testing technique (see section 2.1.4). It states that the values at the edge of equivalence classes, known as boundary values, are more error-prone, and therefore should be preferred when selecting test data. If we assume that the testers who write the test suite followed this technique, then we would expected to find *adjacent* values – the values that differ by the smallest unit – as test parameters. Looking at this from the other direction, if we observe all the values at a certain program point and found some adjacent pairs, then these values can be used to deduce class boundaries. These boundaries can then be used to define *interval-based equivalence classes*, a special case of equivalence classes described previously.

This approach should work for any enumerable single-dimension domain, such as integers or floating point numbers. As for more complex domains, the concept of boundary values is not well-defined [10]. However, even for the single-dimension domain, there remains one question: *how do we determine that two given values are adjacent or not?* This can be answered by defining a predicate, which we called the *adjacency predicate*, such that it receives two values, and evaluates to *true* if and only if the two given values are considered adjacent.

##### Adjacency Predicates for Integers

The adjacency predicate for integer domains is trivial. since we know that two integers are adjacent when they differ by exactly 1. So the adjacency predicate is $adj(x, y) = (|x - y| = 1)$.

##### Adjacency Predicates for Floating Point Numbers

Things are trickier when considering floating point numbers. Technically there is a well-defined concept of adjacency. Floating point numbers are discrete approximations of real numbers, there is always a clear answer on the smallest number larger than a given one. Java provides the functions `Math.nextUp(double)` and `Math.nextUp(float)` precisely for this task. So, the adjacency predicate can be defined as $adj(x, y) =$ `Math.nextUp(x) == y || Math.nextUp(y) == x`.

Careful considerations are needed, though. Humans usually deal with numbers in decimal system, and so we expect that software requirement will often use decimal numbers as well. However, most decimal fractions cannot be represented exactly in binary, so there is built-in imprecision in floating point numbers. Conventionally, when comparing floating point numbers, a small error tolerance is allowed [17]. We suspect that most testers will adopt this practice of allowing for tolerance when applying boundary value analysis to floating point numbers. This means that, for the value of 'the smallest `double` greater than 1.0', we expect to find decimal values such as 1.01, 1.001, or 1.0005, more than the precise value of `Math.nextUp(1.0)`, which is $1 + 2^{-52}$, which approximates to 1.0000000000000002 in decimal.

Also, recall that we want to find adjacent pairs – a pair of values that *differ* by the smallest unit. By this definition, equal values are considered *not adjacent*. But comparing if two values are equal also suffers from the same imprecision problem. So there has to be another tolerance – if the difference between two values is less than this tolerance then they are considered equal.

To accommodate both tolerances, the adjacency predicate becomes $adj(x, y) = \epsilon_0(x, y) < |x - y| \land |x - y| < \epsilon_1(x, y)$. The functions $\epsilon_0(x, y)$ and $\epsilon_1(x, y)$ represent the equality tolerance and adjacency tolerance, respectively. The functions might be a fixed constant (such as 0.001) or varying based on the value of $x$ and $y$ (such as '1% of the smaller value'). The content of these tolerance functions should be configurable by the users.

Floating point also supports special values of positive infinity, negative infinity, and `NaN` (Not a Number). Each of these should be considered as belonging to separated equivalence class not subjected to boundary value analysis.

### Adjacency Predicates for Other Domains

The practice of letting the user define the tolerance function themselves could be extended to support any other arbitrary *enumerable* domains. Instead of defining the tolerance function, users have to define the adjacency predicate itself.

This strategy requires observation of all the arguments and return values of the subject program. This can be done statically, by performing some kind of analysis on the test suite code, or dynamically, by observing actual test runs. We decided in favour of the dynamic approach, because of the following reasons:

1. Static analysis is a non-trivial task. It could even be impossible in some cases, where not all the logic is contained in the source code itself – some test suites read the test data from external resources, such as spreadsheets or database. Dynamic approach is much more straightforward.

2. Daikon itself is a dynamic analysis tool, as the task of observing program values is precisely what the front-end component does. It should not be difficult to adapt the front end to work for our purposes.

To summarise, the implementation of boundary value recovery strategy would consist of *a*) using a Daikon front end to observes the values being passed to, and return from, the program under test, then *b*) select the boundaries from all the observed values, and finally *c*) translate these boundaries into equivalence classes, in the form of a partitioning file.

#### 3.2.2.2 Test Suite Invariants Recovery Strategy

One of the reasons that testing techniques exist is to reduce the number of test cases into manageable number. For equivalence partitioning, the required number of test cases is one case per equivalence class. However, in practice it is typical for a class to have more than one test cases. This could be because testers also employ other techniques alongside equivalence partitioning, resulting in more required test cases. Another possible reason is that the test suite might run fast enough that achieving minimal number of test cases is not the top priority, so testers opt to trade some efficiency for more confidence, by using more test cases per equivalence class. Regardless of the reasons, if there exist multiple test cases that exercise the same equivalence class, then the test logic should be exactly the same, with the test data as the only differences.

When implementing a test suite, testers could simply create one test method for each test case. But if the test logic for these methods are exactly the same, then there will be some redundancy in the test code. As in any other software, redundancy makes code – the test suite, in this case – difficult to maintain. One solution to this problem is to factor out the common parts of the code into another entity, such as a utility method that receives the input, expected output, or both, as parameters. Tests that use such utility methods are called *parameterized tests* [45]. The approach is a common practice, such that JUnit and TestNG, two very well-known Java test frameworks, both provide explicit support for parameterized tests.[2]

Assuming that testers implement test methods belonging to the same equivalence class as some form of parameterized tests, then the subject program will be invoked several times from the same point in the test suite. If we can derive some properties from the parameters and return values at these points, then it makes sense to use these properties as the builder predicates of equivalence classes. This is why we call this strategy the *test suite invariants recovery strategy.*

As in the previous strategy, this one also requires observing values in the test suite, so the arguments of using dynamic approach, and Daikon front end in particular, also apply. Moreover, it also requires inferring properties from the observed values, which is precisely what Daikon is built to do, so the invariant detector part of Daikon could be utilised as well, but instead of running it directly on the subject program, the test suite itself is what being examined.

However, there is one problem with the idea of using the Daikon front end to record values at invocation points: Chicory, the Java front end component, does not include the invocation points as the program points to be observed. To clarify, Chicory does observe values at the method entries and exits, but what we need to know is the *invocation points* – the point where target methods are called. This is a task for the prototype implementation to extend default behaviour of Chicory to work with these extra program points. This can be accomplished by any of the following approaches:

1. Modify the source code of Chicory to support the needed behaviour.

2. Build a new front end that uses Chicory as an API.

3. Build an transformer to modify test suites into a form that Chicory can works on directly.

4. Build a new front end from scratch.

The first 2 approaches are very tightly coupled with Daikon. Approach 1 requires merging and recompilation of source code for every new version of Daikon. For approach 2, it might not be possible to use only Chicory's public methods to achieve our goals, so a hack into private methods might be required, which means the implementation could break down at any

[2]JUnit provides `Parameterized` test runner [56, p. 17], and TestNG provides `@Parameters` and `@DataProvider` annotations [7, p. 42].

Daikon updates. Approach 4 is the cleanest one, but it needs a lot more effort compared with other options. So we decide that approach 3 is the most preferable one that the Yacon should follow.

To summarise, the implementation of test suite invariants recovery strategy would consist of *a*) building a new front end, or modifying the Java front end Chicory, to observe program invocations from test suite, *b*) inferring invariants from the observed invocations, then finally *c*) translating the invariants into builder predicates of equivalence classes, in the form of a partitioning file.

#### 3.2.2.3 Support for User-Defined Strategies

It is conceivable that there could be many more viable recovery strategies than the two we have introduced, therefore it is desirable to build the extractor in a way that allows for more user-defined strategies. To accomplish this, the implementation must declare a common interface that all recovery strategies have to implement. It must also have a mechanism to load user-defined strategy at runtime.

At the end of the extraction phase, a partitioning file is to be produced. The next section describes the format of this file.

## 3.3 The Partitioning File Format

Partitioning files are generated by the extraction phase and processed by the translation phase. We create requirements for the format, as follows:

- Partitioning files must be *human-readable text files*. As discussed previously, human users should be able to inspect, verify and modify partitioning information. For this reason, the files should be human-readable. Additionally, it is desirable, but not required, that the format is in conformance with Daikon file format convention [2].

- Partitioning files must be able to accurately express partitioning information of any Java domain types.

- Partitioning files must be able to express associations between partitioning of a domain type and actual domain variables.

The partitioning file format is henceforth defined to accommodate all these requirements.

### 3.3.1 Structure

A partitioning file is composed of several sections, separated by one or more empty lines. There are 3 types of sections: IMPORTS, PARTITION, and CLASS, in this order.

A partitioning file is to be processed in a line-by-line fashion. Leading and trailing whitespace characters are ignored, therefore indentations can be used for decorative purposes.

A line that starts with the character '`#`' is treated as a comment, which is completely ignored. Note that a comment line is to be distinguished from an empty line, which is used to signify the end of a section.

Readers can refer to appendix B. for the grammar of the file format. The rest of this section provides details of each of the IMPORT, PARTITION , and CLASS sections.

### 3.3.2 IMPORTS section

The partitioning information needs to refer to Java classes throughout the file. For example, we need to know the type (class) to which a partitioning can be applied. One way to make these references is to use the *fully qualified name* [29] of the classes. However, fully qualified names are composed of the package and class names, which can result in very long, hard-to-read names. The purpose of IMPORTS is to provide a way to use short aliases for class names. This section is optional, and can only be placed as the first section. Also, there cannot be more than one IMPORTS sections.

Here is an example of an `IMPORTS` section:

```
IMPORTS
List=java.util.List
Foo=package1subpackage1.subpackage2.Foo
Bar=package2.Foo
```

With this declaration, any reference to the standard `java.util.List` class can be shortened to a simple '`List`', similar to the effect of `import` statements in Java source files. The difference is that the alias does not have to be the simple name of the class, rather, it can be anything, as demonstrated above by the use of '`Bar`' as the alias of `package2.Foo`.

The fully qualified names of the classes on the right hand side of the '`=`' symbols do not have to be unique; it is conceivable that a single class could have more than one aliases. On the contrary, the aliases on the left hand side must be unique.

The classes in the package `java.lang`, (e.g. `String`, `Object`, `System`, `Math`, `Integer`, etc.), do not have to be specified in this section – their simple names can be used directly.

### 3.3.3 PARTITION section

A PARTITION section describes how a domain can be divided into equivalence classes. One PARTITION section then consists of its name, its domain type, and a list of rules to construct equivalence classes. The domain type can be either a fully qualified name of a type, or an alias defined in the IMPORTS section.

Each partition is built upon *class construction rules*. 3 types of them are supported: `EQClass`, `IntervalClasses`, and `ComplementClass`.

#### 3.3.3.1 EQClass Rule

This rule is the most generic one – it can be used to express any equivalence classes, by stating the builder predicate (see page 20) of the class using the following syntax:

```
EQClass <predicate>
```

A special keyword `$value` is used to represent the variable in the predicate. For example, a class of even integers may be represented by `EQClass $value % 2 == 0`.

#### 3.3.3.2 IntervalClasses Rule

This rule supports the special case of interval-based equivalence classes (see page 20), by using the `IntervalClasses` syntax:

```
IntervalClasses [<transform>] <predicate> <mode> <boundaries>
```

In this rule, the `boundaries` list is a comma-separated expressions that are evaluated to valid members of the current domain type. The *optional* `transform` function is used to represent a mapping that transforms all domain members into other values, which can be the same domain or a different one. The keyword `$value` is again used to represent the variable domain member within the transformation context. An example of this optional transform part is `Func[100.0 * $value]: double`, which describes a function that transforms values from a numeric domain (which can be integers or floating points) to a floating point domain `double`.

The value `mode` determines the type of boundaries. Imagine a sorted sequence of all domain members as a one-dimensional line, with the lowest value on the left and the highest value on the right. Each boundary value then marks the end of one equivalence class and the beginning of another. The boundary itself must also belong to one of the classes it divides. If it belongs to the class to the left, then we say it is the *maximum* boundary (equivalently, the left interval is right-closed and the right one is left-opened), otherwise it is a *minimum* boundary (the left interval is right-opened and the right one is left-closed), as depicted in figure 3.4.

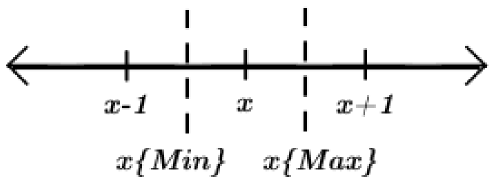

Figure 3.4: Minimum and Maximum boundaries for an integer $x$, denoted `x{Min}` and `x{Max}` respectively.

If the mode is `Minima`, then the boundary values listed are all minimum boundaries. If the mode is `Maxima`, then the boundary values are all maximum boundaries. The other option is `Mixed`, where each boundary value has to be accompanied by a suffix `{Min}` or `{Max}`, to indicate the type of each one individually.

As an example, consider the partitioning of negative, zero, and positive integers once again. There are 3 equivalence classes, therefore there are 2 boundaries, one that divides -1 from 0 and the other that divides 0 from 1. The following are four different ways to represent these boundaries using the `IntervalClasses` syntax:

- `IntervalClasses Minima(0, 1)`
- `IntervalClasses Maxima(-1, 0)`
- `IntervalClasses Mixed(-1{Max}, 1{Min})`
- `IntervalClasses Mixed(0{Min}, 0{Max})` Notice that it is perfectly valid to have both `{Min}` and `{Max}` associated with the same boundary value.

For integer domains it is concise to use the `Minima` or `Maxima` mode. However, for floating point numbers it might not be easy to use these two concise modes, hence the need for `Mixed` mode. Consider the equivalence class of *valid probabilities* – any real number between 0.0 and 1.0 *inclusive*. It is easy to represent the boundary at 0.0 as a minimum boundary

`0.0{Min}`, and the boundary at 1.0 as a maximum boundary `1.0{Max}`. However, suppose we want to use the `Minima` mode, then the boundary `1.0{Max}` has to be rewritten as a minimum boundary, and the smallest domain member greater than 1.0 is needed. That number, $1 + 2^{-52}$ for the `double` domain, is not trivial to compute nor represent as decimal[3]. So it makes more sense to keep both boundaries as they are and allow both types to mix, instead of forcing all boundaries to be of the same type.

#### 3.3.3.3 ComplementClass Rule

This rule supports the special case of complementary equivalence classes – the classes of members not belong to any other classes. It has a very simple syntax: a single word `ComplementClass`.

The complementary class is meaningless if it exists alone – therefore it can only be declared in a program point that has at least one other classes. It also does not make sense to have more than one complementary classes in the same partition.

The PARTITION sections define a part of partitioning information. To complete the information they must be applied on domain variables. This is done using the CLASS sections.

### 3.3.4 CLASS section

A CLASS section lists all the domain variables to be partitioned within the context of a single Java class. To apply partitions on more than one class one would need an equal number of CLASS sections.

As in the PARTITION section, domain types can be specified either as fully qualified name of Java types, or aliases defined in the IMPORTS section. This includes the type declared in the first statement of the section, which sets the context of all the domain variables in its section.

For a domain variable $d$ and partitions $p1$ and $p2$, which must be previously defined as PARTITION sections, we write `d{p1, p2}` to state that these partitioning are associated with the type.

The type declaration syntax in this section consistently uses the postfix style `<name>:<type>`, similar to programming languages such as Pascal, Ada, Scala, etc. Field declarations start with a modifier (`STATIC` or `INSTANCE`), followed by the name and type of the field, such as

```
INSTANCE list: java.util.List
```

[3]The value is computed according to *IEEE 754 Standard* [60]

If a partition $p0$ is to be applied on this field (and every field should have at least one partitioning, otherwise there would be no reason to declare the field) then we can write

```
INSTANCE list: java.util.List{p0}
```

Similarly for method declarations, a Java method `int f(String s, double d)` is referred to as

```
f(s: String, d: double): int
```

Partitions could be applied on any of the argument types and return types. So we can write

```
f(s: String{p1, p2}, d: double): int{p3}
```

for the same method with partition $p1$ and $p2$ applied on the `String` argument, no partitioning on the `double` argument, and partition $p3$ applied on the return value. Note that the return type of Java methods that return nothing (a `void` method) is `` `void' ``, and partitions cannot be applied on this type.

Typically a field declaration would represent a partition at a class-level program point. However, method executions can depend on the values of some fields, which means that there exist invariants at method-level program points that are conditional to the value of some fields. To represent this dependency, field declarations can be added to a method-level program point after a `DEPENDS` clause, which is to be placed immediately after a method declaration. For example, the following method declaration depends on the value of a non-static field `n`, which is an integer domain with partition $p4$:

```
METHOD f(): void
DEPENDS
INSTANCE n:int{p4}
```

### 3.3.5 Example File

Here is an example of a partitioning file based on the class `baz.Bar` in figure 3.5, that demonstrates all the aspects of the file format.

```
# This is a comment

IMPORTS
Bar=baz.Bar

# predicate-based classes and complementary class
PARTITION SimplePartition: int
EQClass $value == 0
EQClass $value < 10 && $value >= 1
ComplementClass

# interval-based classes
PARTITION Sign: int
IntervalClasses Minima(0, 1)

PARTITION SignDouble: double
IntervalClasses Mixed(0.0{Min}, 0.0{Max})

# Mixed type interval-based classes
PARTITION Probability: float
IntervalClasses Mixed(0.0f{Min}, 1.0f{Max})

PARTITION Numeric: char
IntervalClasses Mixed('0'{Min}, '9'{Max})

PARTITION Alphabetic: char
IntervalClasses Mixed('A'{Min}, 'Z'{Max}, 'a'{Min
 }, 'z'{Max})

# Transformation
PARTITION RadiusByArea: int
IntervalClasses Func[3.14159 * $value * $value]:
 double Minima(0, 100, 10000)

CLASS Bar
STATIC c:char{Numeric}
INSTANCE f:Float{Probability}
METHOD f(n:int, d:double{SignDouble}): int{Sign}
DEPENDS
#applies more than 1 partitioning
STATIC c:char{Numeric, Alphabetic}

```

## 3.4 Translation Phase

The goal of this phase is to generate splitting conditions from partitioning information. In Daikon system, splitting conditions are captured in files with `.spinfo` extension, called the *splitter* files. We covered the existing methods to generate these files in section 2.3.

In order to be able to generate these files, we need to understand their format first. This section expands on the format of splitter files, then explains why it is possible to translate a partitioning file into a splitter file.

### 3.4.1 Splitter Files

A simplified version[4] of the format of splitter files can be described as follows: they are textual, composed of *sections* separated by one or more empty lines. There are 2 types of sections, *program point sections* and *replacement sections*.

#### Program Point Section

A program point section starts with the name of the *program point*, follows by one or more conditions associated with that program point. A condition may have optional *formatting options*, which describe textual representation of the condition in different formats. The structure of a program point section is illustrated in this following syntax:

```
PPT_NAME <program point name>
<splitting condition 1>
    <formatting option 1.1>
    <formatting option 1.2>
    ...
<splitting condition 2>
...
```

A program point refers to one or more specific places in the program source code. At the time of writing, the current version (5.2.2) of Daikon produces 4 types of program points: method entry, method exit, instance-level and class-level. To help illustrate, examples of program points are provided based on a hypothetical Java class `Bar` defined in figure 3.5:

[4]For full description, see section 6.2.1 in Daikon's user manual [3].

```
 1  package baz;
 2  public class Bar {
 3  static char c;
 4  Float f;
 5  public int foo(int x, double y) {
 6  if(someCondition())
 7  return x;
 8  else
 9  return (int)y;
10  }
11  ...
12  }
13
```

Figure 3.5: A hypothetical Java class baz.Bar

| Program point | Example | Invariant type |
|---|---|---|
| Method entry | `baz.Bar.foo(int, double):::ENTRY` | Method precondition |
| Method exit | `baz.Bar.foo(int, double):::EXIT7` | Method postcondition at line 7, among multiple return statements |
|  | `baz.Bar.foo(int, double):::EXIT` | Method postcondition for all return statements |
| Instance | `baz.Bar:::OBJECT` | Instance invariant |
| Class | `baz.Bar:::CLASS` | Static invariant |

A precondition is a condition that is true before the execution of the method. A postcondition is true after the method is executed. The instance invariants and static invariants are more generally known as *class invariants*. The difference is that, as the names suggested, an instance invariant is a property related to one or more *instance variables* that stays true for all states of an object instance, while static invariants are related to *static variables*.

Each splitting condition in a program point section is a Java expression that evaluates to a boolean value. A condition expression can refer to any variables that it has access to at that program point. For example, a condition at class-level program point (a static invariant) can only refer to static fields of that class, while a condition at the beginning of an instance method (a precondition) can refer to any method arguments or fields of that class. However, method calls are not allowed in the condition expression. There is an exception to this rule; methods that are listed in replacement sections can be called.

#### Replacement Section

The replacement section lists the methods that can be used by splitting conditions. It starts with the keyword `REPLACE`, follows by one or more pairs of lines, the first line is a method name, and the second is a Java expression, called the *replacement*.

```
REPLACE
<method 1>
<replacement 1>
<method 2>
<replacement 2>
...
```

### 3.4.2 Translation From Partitioning Files to Splitter Files

It can be seen that a partitioning file contains all the information needed to generate a splitter file. This includes:

- Program point names. A partitioning file has references to methods and fields in the class sections. These can be directly converted into program point names.
- Splitting conditions. Each of the equivalence class in partitions associated with a domain variable can be translated into a splitting condition under the program point containing that domain variable. An equivalence class is a set, and every set has a builder predicate (see page 20), which can be used as a splitting condition.

Note that a builder predicate can be any valid Java expression that evaluates to a boolean value, while a splitting condition needs to be free of method calls, other than those appear in the Replacement section. Therefore the implementation of the translation phase need to extract the method calls within builder predicates into the Replacement section in order for these builder predicates valid as splitting conditions.

## 3.5 Summary

This chapter provided the design of Yacon, the prototype solution to the research question of using equivalence partitioning information to assist dynamic invariant detection. Yacon is designed to work in two phases, the extraction phase extracts partitioning information from test suites and writes it to an intermediate partitioning file; the translation phase parses that file and produces a splitter file, which can be of assistance to Daikon in detecting conditional invariants. Two recovery strategies for extracting partitioning information were introduced – the boundary value recovery strategy and the test suite invariants recovery strategy. We also described the format of the partitioning file in detail, and explained how it should be possible to translate from a partitioning file to a splitter file. The next chapter will build upon the design in this chapter to produce a concrete implementation of our prototype solution.

# 4 Implementation

Objective: this chapter presents the concrete implementation of Yacon, the prototype solution to the research question of using equivalence partitioning to assist conditional invariants detection.

## 4.1 Structure

Yacon is developed completely in Java, using Eclipse as the IDE. To maximise compatibility with Daikon, the same version of the language, Java 7, is selected.

The final product of Yacon is a number of Java classes, packaged as two Java `.jar` files, one with dependencies included[1] and one without. It is intended to be run from a command line. See appendix C for build and execution instructions.

The Java classes we implemented are divided into the following packages:

1. `me.arkorwan.yacon.extractor` – for the classes responsible for the extraction phase. Each of the recovery strategies are put into its own package, which is a sub-package of this one.

2. `me.arkorwan.yacon.translator` – for the classes responsible for the translation phase.

3. `me.arkorwan.yacon.model` – for the common data models shared between both phases.

[1]Daikon is also a dependency but it is expected to be installed locally, so it is included in none of the `.jar` files.

4. `me.arkorwan.utils` – for generic utility classes not specific to the purposes of Yacon.

The only exception is the class `daikon.Yacon`. This is the main class, containing the entry point of the program. Since the tool is intended to be used from the command line, in the same fashion as all other tools in the Daikon system, the user has to type in the fully qualified name of the main class to start execution. We decided that using a short, simple name would be more convenient for the users.

The rest of this chapter explores the implementation of models classes, the extraction phase, and the translation phase, in this order. After that we also present the test coverage of Yacon and the problems we have faced during implementation.

## 4.2 Data Models

The representation of information necessary to the workflow of Yacon centred around the concept of *program points*, which is already covered in section 3.4.1. See figure 4.1 for the class diagram of the model classes.

The functions and purposes of each class are provided in the following paragraphs from the ground up, starting from classes with no dependency on other models classes, until the class representing program points is reached.

**`DomainType`** (not present in the class diagram) This class encapsulates the types that can be used as the *domain* for equivalence partitioning, which are all Java types, including the primitives.

**`JavaExpression`** (not present in the class diagram) This class represents mapping functions that transform values from one type to another. A `JavaExpression` consists of an input type, an output type, and a String that must be a valid Java expression that takes an input variable named '`$value`' and evaluates to the output type. The expression is validated by using a `ScriptEvaluator`, from the *Janino* library[2], to build a return statement from the String '`return <expression body>;`'.

We also extended the expression syntax to support another special keyword, `$func`, as an 'escape' mechanism to represent parts of the expression that will not be validated. This is useful for expressions that references

[2]`http://docs.codehaus.org/display/JANINO/Home`

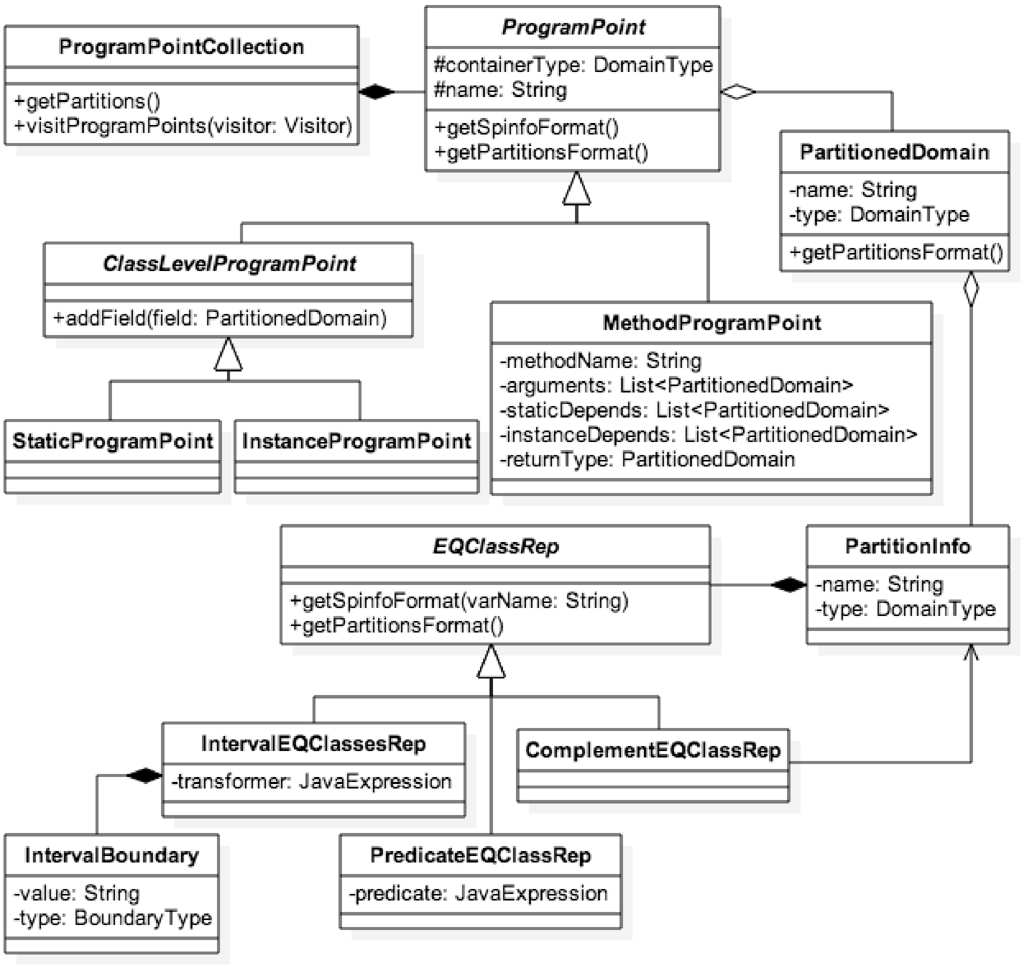

Figure 4.1: Model classes diagram

other methods of variables unknown to the validator. We rearrange the expressions to be validated so that the validator sees these escaped expressions as method arguments of unknown values. The syntax for this escape expression is

```
$func:<type>{<escaped expression>}
```

This can be used in the predicate part of an `EQClass` declaration, or the transformation part of an `IntervalClasses` declaration (see page 87).

**`IntervalBoundary`** This class represents a single boundary between two interval-based equivalent classes. A boundary is described by a value and the type of the boundary, which can be either *closed-on-left* or *closed-on-right*. For a closed-on-left boundary, the interval to its left is closed at the given value – in other words, the given value belongs to the interval on its 'left' side, which means the value is the maximum of the interval. By the same logic, if the boundary is closed-on-right, then the given value is the minimum of the interval on the 'right' side. Note that the left-right notion refers to the direction of the closed interval relative to the position of the boundary.

**`EQClassRep`** This abstract type describes all kinds of equivalence classes representation. The subclass `PredicateEQClassRep` uses a `JavaExpression` (with return type always fixed to `boolean`) as builder predicates describing equivalence classes. `IntervalEQClassesRep` uses a list of `IntervalBoundary` objects to describe the interval-based equivalence classes, and `ComplementEQClassRep` uses a reference to all other equivalence classes belonging to the same partition to describe the complementary equivalence classes.

**`PartitionInfo`** This represents a partition – a collection of equivalence classes, with a name and a type associated with the partition.

**`PartitionedDomain`** An instance of this class is an application of partitions to a domain, such as a method argument, a method return value, or a field. The object is composed from a name (variable name in the cases of method arguments and fields), the type of the domain, and a list of partitions to be applied.

**`ProgramPoint`** This is an abstract class that represents all types of program points. Recall that there are 4 of them, *a*) method entry, *b*) method exit, *c*) instance program point (for instance variables), and *d*) static program point (for class variables). Every program point has common attributes – its name and its declaring class – the class in which the field or method described by the program point is declared, and also a list of `PartitionedDomain`s.

For our purposes, there are no distinctions between a method entry and a method exit, so the two types are combined, with the class `MethodProgramPoint` – a direct subclass of `ProgramPoint` – as their representative. This class defines more attributes to capture information necessary for identifying a Java method, i.e the name, arguments, and return type of the method. Also defined here is the dependency of a method program point on field variables.

The abstract class `ClassLevelProgramPoint` – the other direct subclass of `ProgramPoint` – represents the instance and static program points combined.

**`ProgramPointCollection`** This is simply a collection of `ProgramPoints`, with a method to extract all relevant partitions. It also offers a way to traverse the program points in a well-ordering manner. This feature is implemented using the *visitor pattern* [24]. Clients can extend the abstract class `ProgramPointCollection.Visitor` to receive callbacks when each type of program point is visited, or the enclosing class is changing (leaving the enclosing class and entering a new one). This will be explained in more detailed when it is used in the extraction phase.

## 4.3 Extractor

The purpose of the extractor is to produce a partitioning file. A *recovery strategy* is selected and executed to obtain necessary information to create a partitioning file. We implemented the two strategies that were discussed in the design phase, and also a mechanism that allows users to create their own recovery strategies.

Once the extractor starts, it selects a recovery strategy based on a configuration value. All strategies are subclass of the abstract type `RecoveryStrategy`. The strategy object is then instantiated and executed, resulting in an instance of `ProgramPointCollection`. It would then be passed on to a `PartitioningFileWriter`, which writes the information down to a file in the partitioning file format, completing the process. `PartitioningFileWriter` is an abstract class, and currently there is only one concrete implementation – `SimplePartitioningFileWriter` – which writes all relevant partitions and program points using machine-generated name for the partitions and fully qualified names for all references to Java types.

Next we discuss implementation details of the two recovery strategies.

### 4.3.1 Boundary Value Recovery Strategy

This strategy assumes that testers use the boundary value analysis testing technique. The procedure will be explained in two steps, as laid out in the design phase:

1. Execute Chicory.

2. Analyse Chicory result for *adjacent pairs* of values.

### Step 1: Running Chicory

This step is simply an execution of Chicory from within a Java program. Chicory has to be run as an external process because of a technical limitation[3]. The class `JavaProcessRunner` is created to facilitate execution of external Java programs in a different process. Internally it utilises the built-in class `java.lang.ProcessBuilder` to do the task. To use this class, client code needs to submit a *runner class* – an implementation of the interface `JavaProcessRunner.JavaRunnable`, which declares a single method, `run`, that receives a list of String as its argument. An instance of the given runner class will be created at runtime via reflection, and it is required that the runner can be instantiated using the empty constructor.

In this case, the runner class is `ChicoryRunner`, which simply relays the arguments to the main method of `daikon.Chicory`. After execution, the resulting trace data will be written to an archived trace file with the given file name.

### Step 2: Reading Trace Data and Finding Adjacent Pairs

It is recommended to use method `daikon.FileIO.read_data_trace_files` to read the trace file [2]. The method receives a list of trace files, and a processor, which must be a subclass of `daikon.FileIO.Processor`. Note that references to trace files has to be in file URI format ('`file://`' follows by full path to the file), for the reason given in section 4.6.5.

Our implementation of processor is the class `TraceCollector`. It scans through all samples and grouped them by program points. It then applies a filter to select domain variables to analyse. All the following conditions have to be met in order for a domain variable to be selected:

- The domain belongs to a class whose name matches a pattern of classes we want to observe. The pattern is defined by the configurable value `boundary_values.patterns`.

- There is an *adjacency predicate* that can process the type of this domain variable. The predicates for Java's primitive types (`boolean`,

[3]See section 4.6.4 for details.

byte, short, int, long, float, double) and their wrapper counterparts are provided by default. To analyse any other types, custom adjacency predicates are needed.

For each selected domain variable, all its values found in the trace are collected into a sorted list, and each pair of consecutive values in the list is examined. If the pair is not equal (according to the *equality predicate*), but adjacent (according to the adjacency predicate), then it is identified as a possible boundary.

Depending on the way the test suite is written, this algorithm can produce too many false positives. This has the effects of reduced quality of resulting partitioning information, as well as slowing down the process. One way to decrease the number of false positives is to apply the experimental technique we called *second-order boundaries*. It keeps track of the adjacency detection result as a list. If there exist a run of successful detection, we only keep the first and the last of the run. This is actually an application of the boundary value analysis technique on the boundaries themselves, hence the name 'second-order boundaries'. This is optional and can be turned on/off using the configuration boundary_values.second_order_trigger.

Each of the remaining pairs of adjacent values is identified as a boundary. If there is at least one boundary detected for a domain variable, then we create a PartitionInfo with an IntervalEQClassesRep to encapsulate this partition. The PartitionInfo is then attached to the ProgramPoint object representing this particular program point.

At the end of the process, all instances of PartitionInfo are collected into a PartitionInfoCollection, which is the final result of the strategy.

**Providing custom adjacency predicates.** We have stated that if users want to apply this strategy on types other than the numeric primitive types, the adjacency predicates for those types have to be provided. To do that for an arbitrary type $T$, users have to create a subclass of the abstract type AdjacencyPredicate<U>, and the type parameter $U$ must be assignable from $T$ – that is, any instance of $T$ can be cast to $U$. An AdjacencyPredicate<U> must implement the method compareTo, to compare two objects as follows:

- If the two objects are equal, then return 0.
- If the two objects are different, but are close enough to be considered 'almost equal' – that is, if the equality predicate should be true, then return 1 (if this object is greater) or -1 (if the specified object is greater).

- If the two objects are *adjacent* – that is, if the adjacency predicate should be true, then return 2 (if this object is greater) or -2 (if the specified object is greater).
- Otherwise, return any integer greater than 2 if this object is greater, or any integer less than -2 if the specified object is greater.

To let the strategy know of the existent of any custom `AdjacencyPredicates`, users can specify the name of the `AdjacencyPredicate` class, and the name of the class to which this predicate can be applied, in the configuration value `boundary_values.adj_predicates`.

### 4.3.2 Test Suite Invariants Recovery Strategy

Test suite invariants recovery strategy assumes that if a program expression in a test suite is called several times with different parameters, then those parameters should belong to the same equivalence class. As concluded in the design phase, the first thing we need to do is to find a way to observe program points at those line of codes, by either creating a completely new front end, or extending the default Java front end Chicory to support this kind of program points.

However, our actual implementation went in a different route. We left the Chicory tool unmodified, and applied the changes to the test suites instead. The idea is that, since Chicory can only observe method entries and exits, then if we transform every program expression we want to observe into a *proxy* method, while preserving the flow of the program, then Chicory can be used as is. We call this the process of *proxification.*

We divided the procedure into the following steps:

1. Proxify the test suite.
2. Compile the proxified test suite.
3. Run Chicory and Daikon using the proxified test suite.
4. Collect the result.

We will go through each of these steps in detail.

#### Step 1: Test Suite Proxification

There are two main options when trying to modify existing Java code – to do it at source code level, or at bytecode level. For our purpose it is preferable to modify the source code, since we want to preserve the intention of the code writer, which could be lost during compilation, by means of compiler optimisation. We chose *Spoon* [48] as the library to assist this task. The main reason we selected Spoon is the ability to easily insert code snippet as a Java String into the *abstract syntax tree* (AST) data structure generated from parsing a Java source file.

Given a Java source file, Spoon builds and traverses the AST of the source file. It provides a callback method that is invoked every time a specified type of AST node is being traversed. We can override the method to modify the AST. At the end of the traversal, the AST is written as a new source file.

For our purposes, we observe two types of AST nodes, one for method invocation, the other for field assignment. The nodes are filtered for invocations or assignments on the classes we want to analyse, then passed on to the class `MethodProxifier` to do the proxification.

**Proxy method placeholder class.** For every AST node that `MethodProxifier` receives, it attempts to create a new proxy method for the node, and replace the original invocation by an invocation of the new method. All proxy methods are put into a static nested class of the original class, named `ẎȧċȯṅProxifier`. Notice that the class name contains characters Y, ȧ, ċ, ȯ, and ṅ, which are not in the *Basic Latin* unicode block i.e. the ASCII characters. This is because we wanted the name to be as unique as possible, so that we could instruct Chicory to observe only the classes with this name, using the option `--ppt-select-pattern`. With a unique name like this, the chance of spending time observing unwanted program points would be very low.

The structure of this proxifier class is presented hereafter. Notice there are two fields defined, `ẏȧċȯṅTarget` and `ẏȧċȯṅInstance`, whose names also contain unicode characters to reduce the chance of accidental duplication. Their purposes will be discussed shortly after.

```java
public static class ẎȧċȯṅProxifier {
  Object ẏȧċȯṅTarget;
  static ẎȧċȯṅProxifier ẏȧċȯṅInstance = new
   ẎȧċȯṅProxifier();

    static ẎȧċȯṅProxifier proxify(Object o) {
        ẏȧċȯṅInstanceẏȧċȯṅ.Target = o;
        return ẏȧċȯṅInstance;
    }

    ... //proxy methods
}
```

**Proxy method signature.** We want to preserve all the information of the original invocations within the proxy methods, in order to be able to refer back to the original. Our implementation transforms the names of the methods and their parameters, simplifies the access modifiers, but tries to keep everything else the same.

We broke down the components of a method signature, and describe each of them translated into the proxy method.

- **Access modifier.** Since the only place a proxy method will be accessed from is its original enclosing class, which is obviously in the same package, so this component is always set to *default*, for simplicity.

- **Static modifier.** This is the keyword `static` that declares the method as a class method when present, or an instance method when absent. Static modifier is present in a proxy method if and only if it is also present in the original method.

- **Return type.** This is can be a Java type, or `void` if the method does not return any values. The return type of a proxy method is the same as its original method.

- **Method name.** The name of proxy method is required to be unique for each occurrence of the original invocation/assignment in the source code, because if the same method is invoked at different places in the test suite, then we want to observe them separately. We also need the encode the declaring class name into the proxy method name, resulting in the following form:

  ```
  m_<original name>_<declaring class>_<unique id>
  ```

  The letter `m` at the beginning signifies that the origin of this proxy method is a method invocation, not a field assignment. <declaring class> is the fully qualified name of the class that declares the method, with all occurrences of the period character ('.') replaced by '¤'. This is because periods are not allowed in a valid Java method name. <unique id> is a unique identifier for each invocation. We use the position of the original invocation within the source file as this identifier.

- **Method parameters.** Parameters of a proxy method have the same types and order as the original method. We also want the name of each parameter to be the same, but for technical implementation reasons, it will be more efficient when forming `PartitionInfoCollection` from Daikon result if we know the index of any parameter just from its name, otherwise we would have to look up and compare from the actual method implementation. Therefore, we use a scheme that embeds the index of every parameter into its name – a method parameter `arg` at index *i* becomes `Γarg$iℲ`. The enclosing characters 'Γ' and 'Ⅎ' are used to establish the beginning and the end of an argument value.

  Note that parameters names are only available when compile using `-g` option to retain debug information. This is the same requirement that Daikon puts on its subject program.

- **Exceptions.** Checked exceptions are a part of method signature. It tells what kind of non-runtime exceptions might be thrown from the method. Proxy methods copy the exact same exceptions from their original methods.

- **Generics type parameters.** A method can declare its own generic type parameters as part of its signature. For example, the method `<T> int length(List<T> list)` has a reference to type parameter `T`. This is a complex subject that will be discussed along with other aspects of generic types in section 4.6.2.

As an example, if the following method, declared in class `foo.Bar`, is invoked at position 42 in a test class:

```
public static String f(int n, double[] d) throws
   EOFExpcetion, FileNotFoundException { ... }
```

It would produce a proxy method with the following signature:

```
static String m_f_foo¤Bar_42(int Γn$0Ⅎ
   , double[] Γd$1Ⅎ  throws EOFException,
   FileNotFoundException { ... }
```

As for field assignment, the signature follows a similar set of rules, with a few differences:

- The name of the method uses prefix '`f`' to indicate instance field assignment, or '`sf`' to indicate static field assignment.

- There is always exactly one parameter. Its type is the same as the field type. Its name is the name of the field, enclose by the characters ‘Γ’ and ‘Ⅎ’, without the index indicator ‘$<id>’.

- The return type is the same as the type of the field. This might be surprising to some readers, since field assignment is used to *set* a new value to a field, as oppose to getting the current one. However, it is also valid to use the value on the right-hand-side of the assignment as part of larger expression. This is might be frequently seen in loop construct, for example:

  ```
  while((this.line = reader.readLine()) != null
   ){ ... }
  
  ```

  The expression `this.line = reader.readLine()` is a valid field assignment that is also used as part of the null checking. If the assignment is replaced by a `void` method, then the result would not be a valid program. Instead, the replacing method must return a value of the same type as `this.line`.

As an example, a public static field `s` of type `String`, declared in class `foo.Bar`, and has an assignment at position 64 in a test class, would produce a proxy method with the following signature:

```
static String sf‿s‿foo¤Bar‿64(int Γsⅎ ) { ... }
```

**Proxy method body and the replacement invocation.** The task of proxy methods consists of relaying the arguments to the right method (or assigning the right value), and relaying the result back to the caller.

For *static methods*, this is very simple. We can just forward the argument to the original method directly. The body of a proxy method generated from `static int Foo.f(int n)` would be simply `return Foo.f(n);` (Omit the keyword ‘`return`’ for `void` methods.) On the caller side, the invocation `Foo.f(123)` would be replaced by `ŸȧċȯṅProxifier.<proxy method name>(123)`.

Similarly, the content of proxy methods for *static fields* would be of the form `return(<class>.<fieldname>=<argument>;)` The replacement on the caller side takes exactly the same form as in the case of static methods.

It is a little more complicated in the case of instance members. While we can directly refer to the target class in static case, which is known at compile time, for instance members we need a reference to the target instance. This is accomplished by the following process:

- On the caller side we have an invocation of method $f$ on object $o$ with arguments $args$.
- A replacement invocation is built, starting by calling the static method `ŸȧċȯṅProxifier.proxify(o)`, putting the target object $o$ in as the argument. This will set the value of the `ÿȧċȯṅTarget` field to the target object.
- `proxify` returns a singleton instance of the class `ŸȧċȯṅProxifier`. We can use this instance to further call the proxy method generated from $f$, supplying $args$ as the arguments. This is the replacement invocation.

For the content of the proxy methods, start with the target object, which we know is assigned to `ÿȧċȯṅTarget`. Because the type of this field is plain Object, a cast back to the original type is needed. Then we can progress to the same steps as when processing static members, by forwarding the arguments to the original method, assigning field values, and relaying return values.

For example, the invocation `o.g(123)` on method `int Foo.g(int n)` would produces `ŸȧċȯṅProxifier.proxify(o).g(123)` on the caller side, and `return ((Foo) ÿȧċȯṅTarget).g(<argument>);` on the proxy side.

The proxification process as described here works successfully for most cases. The process becomes more complicated when generics are involved, with one problem remains unsolved. The issue is described in detail in sections 4.6.2 and 4.6.3.

### Step 2: Compilation

The proxification process produces a set of manipulated source files. This step is simply to compile them into binaries. We used the built-in type `javax.tools.JavaCompiler` for this task. The generated source files are put into a specific folder, so we search that folder for all files with '.java' extension and pass them to the compiler. The compiler option `-g` is also set, in order to retain debug information necessary for Daikon. The resulting classes are put into another folder, which we will call the *proxified binaries folder*.

### Step 3: Running Chicory and Daikon

This step executes Chicory and Daikon using the proxified classes from the previous step. Users can specify the arguments to pass to Daikon in the configuration file.

Chicory and Daikon have to be executed as a separate process[4]. As in the boundary value recovery strategy, the class `JavaProcessRunner` can be used to facilitate running external Java process. The runner class for this strategy is (`DaikonRunner`. Its task is to execute Chicory and Daikon in on-line mode, focusing only on program points whose name contain the name of the proxy class ‘YȧċȯṅProxifier’. We specifies the command line option ‘–format java’ to tell Daikon to generate the resulting invariants using Java format, as is required by the partitioning file format.

While Daikon has an option to write the result as a file, we found that the option does not work well in conjunction with the on-line mode, so the option is not specified. By doing this, Daikon will use its default behaviours, which is to write the result to the standard output. We can then capture the result by redirecting the standard output to a file, by redefining Java’s `System.out` as a `java.io.PrintWriter`.

For Chicory and Daikon to be able to run successfully, the classpaths given to them have to be set up in the right way, especially the one containing proxified classes. Since these classes are compiled twice – before and after the proxification process, there will be two versions of the binaries for them. (Unless the proxified binaries folder is set up to be at the same location as the original, in which case the proxified binaries will replace the original.) The program sets up the classpaths by always putting the proxified folder in front of all other classpaths. Since the classloader searches for class binaries from each specified classpath in the order of appearance [47], this setup ensures that the proxified folder has precedent over the original.

### Step 4: Daikon Result Collection

This step reads from the file generated from the previous step, and parse it into a `ProgramPointCollection`, using class `InvariantCollector` and its concrete subclasses – `ClassLevelInvariantCollector` to handle class-level invariants (static and instance), and `MethodInvariantCollector` to handle method-level invariants. The parser filters out some of the excessive invariants and program points, such as those concerning properties of the

[4]see section 4.6.4 for the reasons.

auxiliary class `ŸȧċȯṅProxifier` itself, or the invariants that Daikon could not format into a Java expression.

For clarity, in this part we use the term ‘invocation point’ to refer to program points on proxy methods, and reserve the term ‘program point’ itself to refer to program points on the subject program. In this sense, one program point can encompass any number of invocation points, while an invocation point is always associated with a single program point.

When the parser scans the input file and encounters an invocation point, it can infer the associated program point from the information we embedded into the name of the invocation point. A new instance of `InvariantCollector` subclass, which internally holds a reference to a `ProgramPoint` object, is created for every *new* program point. The parser then passes on the invariants on each invocation point to the associated `InvariantCollector`, which keeps track of the invariants, as well as the number of times each unique invariant occurs.

At the end of the execution, the occurrence of each invariant is examined. If it is equal to the number of invocation points belongs to that `InvariantCollector`, it means the invariant is universally true for all invocation points, which is not interesting since we want to find *splitting conditions*. Therefore, these universally true invariants are discarded.

For the rest, we try to extract variables from the invariant, by looking for substrings that are enclosed by the characters ‘Γ’ and ‘Ⅎ’. Since we want to build predicate-base equivalence classes from these invariants, and every equivalence class must be tied to a single domain, therefore we are only interested in the invariants where the number of variables is exactly 1. If an invariant has no references to variables, then there is no domain. If there are more than one variables, then it might be an invariant over relationship between multiple domains, which is not expressible in the partitioning file format.

Invariants that survive the above process are grouped by the variable, and built into `PredicateEQClass` objects under a `PartitionInfo`. A `ComplementEQClassRep` is also automatically added into any `PartitionInfo` containing more than 1 equivalence classes. The `PartitionInfo`s are then added to `ProgramPoints`. All the `ProgramPoints` are then collected into a `ProgramPointCollection`, which is the end result of the whole strategy.

### 4.3.3 User-Defined Strategies

In addition to using the two strategies we implemented, users can opt to create their own strategy. This can be done by the following steps:

1. Creating a subclass of the abstract type `RecoveryStrategy` and implement the abstract method `execute()`, which returns an object of type `ProgramPointCollection`. The subclass must provide either a public empty constructor, or a public constructor that receives a single argument – a configuration object (an instance of `com.typesafe.config.Config`), or both. Note that the classes `BoundaryValueStrategy` and `JavaSuiteInvariantStrategy`, which are the implementation of our two strategies, also follow this same pattern, and can be used as reference.

2. Compile the subclass. Typesafe's config library and Yacon need to be accessible in the classpath for the compilation to be successful.

3. Set the configuration value `yacon.extraction.strategy.class` to the *fully qualified name* of this `RecovertStrategy` subclass. If the strategy has configurations, then put them into the main configuration file, and set the value of `yacon.extraction.strategy.name` to the top-level name of the strategy configuration.

As a minimal working example, consider the following class `EmptyStrategy`.

```java
import me.arkorwan.yacon.extractor.RecoveryStrategy;
import me.arkorwan.yacon.model.
    ProgramPointCollection;

public class EmptyStrategy extends RecoveryStrategy
    {

  @Override
  public ProgramPointCollection execute() {
    System.out.println("Hello, world!");
    return new ProgramPointCollection();
  }

}
```

This strategy does nothing but printing 'Hello, world!' to the standard output, then returns an empty `ProgramPointCollection`. It does not implement any explicit constructors, which means the default constructor `public EmptyStrategy()` is automatically provided.

Suppose we also have the following configuration file named `example.conf`:

```
yacon : {
  extraction : {
    strategy : {
      class : EmptyStrategy
    }
    output_file : example.ycnp
  }
}
```

After compile the class and make sure it is in accessible from Java classpath, then the following command would produce the text ‘Hello, world!’ to the standard output:

```
java daikon.Yacon extract example.conf
```

See appendix C for detailed description of the configuration file and how to execute Yacon from command line.

## 4.4 Translator

The sole responsibility of the translator is to convert a file from the partitioning file format into a splitter info file. We identified 2 subtasks. Firstly, the input file has to be parsed into data models, and secondly, the models are transformed and printed to text output. The main class of the translator is the class `PartitionTranslator`, which contains two static methods for each subtask.

### Parsing

Parsing starts by creating an instance of `YaconStatementEmitter`, which is an iterator that reads the given partitioning file and sequentially produces a statement upon request, until the end of file is reached. It basically produces the statement in a line-by-line fashion, with 2 exceptions:

1. Comment lines (lines that has ‘#’ as the first non-space character) are ignored.
2. Consecutive empty lines (lines that contain no non-space character) are treated as a single empty line.

The class `YaconStatement` represents statements. Its singleton subclasses `EmptyStatement` and `FinishedStatement` represent empty lines and the end of file, respectively.

The YaconStatements are not directly processed by the main translator class. Instead, PartitionExtractor itself only inspects the current statement and assigns the appropriate processor to handle the next section. The processors are subclasses of the abstract class SectionProcessor, which repeatedly processes the next input statements until either an EmptyStatement or a FinishedStatement is encountered. There are 3 such subclasses, one for each of the IMPORTS, PARTITION, and CLASS sections.

- ImportsSectionProcessor consumes the IMPORTS section, and stores the type aliases to be used further as a mapping from Strings to DomainTypes.
- PartitionSectionProcessor processes PARTITION sections, each section is parsed and stored as an instance of PartitionInfo class containing one or more EQClassReps, for later references by CLASS sections.
- ClassSectionProcessor processes CLASS sections, each section produces a collection of ProgramPoint instances, with appropriate relation to PartitionInfos.

## Printing Splitter Info

At the end of the parsing process, all the information will be collected as a list of ProgramPoints. A splitter info file can then be produced by iterating though all the equivalence classes (EQClassReps) in each program point, and convert the information into condition expressions, by using the method getSpinfoFormat(), which each of the subclasses of EQClassRep implements differently.

For example, consider a PartitionInfo based on this partition, bound to a variable x:

```
PARTITION SimplePartition: int
EQClass $value == 0
EQClass $value < 10 && $value >= 1
ComplementClass
```

There are 3 EQClassReps in this partition. Their getSpinfoFormat() method would produce the following strings, respectively:

- x == 0
- x < 10 && x >= 1
- !((x == 0) || (x < 10 && x >= 1))

## 4.5 Testing

There is a comprehensive set of test cases for most of the code, implemented using the test framework TestNG [7]. They were developed along with the product code itself. The build process is set up in a way that all the test cases have to be passed in order to create a successful build. While this cannot ensure correctness of Yacon, it can improve confidence level we have on the solution implementation.

We used EclEmma to measure code coverage, to make sure that the code is tested thoroughly, resulting in the following statistics:

| Coverage Metric | Total | Covered | % coverage |
|---|---|---|---|
| Instructions | 8,482 | 6,862 | 80.9% |
| Lines | 1,971 | 1,608 | 81.6% |
| Branches | 788 | 507 | 64.3% |
| Methods | 327 | 263 | 80.4% |
| Types | 62 | 56 | 90.3% |
| Cyclomatic complexity | 724 | 444 | 61.3% |

## 4.6 Challenges and Issues

This section details a number of technical challenges and issues we have encountered over the course of development.

### 4.6.1 Handling Multi-Catch Clauses with Spoon

We selected Spoon to assist the task of source code manipulation, a part of test suite invariants recovery strategy. It serves out purposes very well. But we did find a big obstacle during development.

We were using version 3.0 of Spoon when a defect was discovered – the generated sources failed to resolve fully qualified names of the types in Java 7's multi-catch clause. For example, instead of writing `catch (java.io.EOFException | java.io.FileNotFoundException e)`, it produces `catch (EOFException | FileNotFoundException e)`. And since the configuration of Spoon was set up to use no `import` statement, the files containing multi-catches failed to be compiled. A natural solution to this problem is to upgrade Spoon the latest released version (3.1 at the time). But the problem still persisted.

Another approach to solve the problem is to configure Spoon to always use `import` statements. This solution was able to fix the issue, but another issue arose in its place: Spoon tried to `import` the generated nested class `ŸȧċȯṅProxifier` into its enclosing class – which is both unnecessary and illegal. The source files still failed to be compiled.

We then looked at the code repository[5] and found that the multi-catch issue has been reported and *fixed*, but not released yet. So we tried to upgrade the library to several commits on the main branch of the repository. The latest commit at the time already made a lot of changes to the API that are not backward-compatible, so that our working client code failed. On the other hand, the commit that fixed this particular issue contained a failed test case that prevented the library to be successfully built. After some trials and errors, we finally found a particular commit[6] that worked for our purposes.

### 4.6.2 Proxification of generic type parameters

The proxification process as described in section 4.3.2 works correctly, as long as generics type parameters are not involved. Methods containing references to type parameters introduce difficulties to our process. We have stated that the proxification process tries to create proxy methods that mimic their origins as close as possible. Ideally we would like to apply this principle to type parameters as well, by also using generics for the proxy methods. To do that, we need the information of typed parameters in detail. However, because of *type erasure*[7], we could not find a way to obtain all the needed information *reliably*.

To demonstrate the problem, consider the following `singletonList` generic method, defined in class `G`.

```
public static <T extends Number> List<T>
    singletonList(T item)
```

And suppose the method is invoked like this:

```
List<Integer> list = G.singletonList(1);
```

This method receives an instance of any object type that is a subclass of `java.lang.Number`, and returns a list parameterized to that type. Applying the proxification process on this method, we would like to generate a proxy method like the following:

```
static <T extends Number> List<T>
    m_singletonList_G_42(T ⌈item$0⌋
    ) {
```

[5]`https://github.com/INRIA/spoon`
[6]commit 13bc4becb2603f7558fc454c28cfb8b2c0344766
[7]See `https://docs.oracle.com/javase/tutorial/java/generics/erasure.html`

```
return G.singletonList(⌜item$0⌟ );
}
```

But because we do not have the information of generic type `T`, it is not possible to use the parameter type `T`, nor the return type `List<T>`. They must be replaced by some other non-generic types.

For an arbitrary type parameter $T1$ , we need a type $T2$ that satisfies the following conditions:

1. The actual argument can be assigned to type $T2$.
2. Instances of type $T2$ can be assigned to type $T1$.

This means any types in the whole type hierarchy from the argument type up to $T1$ can be used. In our implementation, the argument type is tried first. This covers most of the cases, for it trivially satisfies both conditions. The only case this tactic fails is when the argument is null, which means the argument type is *undefined*. When that happens, we try to fall back on the type of the parameter as stated in the method signature. This fall-back mechanism is not guaranteed to succeed since the second condition could be violated. For the `singletonList` example, the type of the parameter is `T extends Number`, which becomes `Number` after type erasure. Since `Number` cannot be assigned to `Integer`, the second condition is violated, and the generated code will fail to compile.

To work around the issue, users are advised not to use `null` directly as method arguments. Instead, refer to a variable or method with the desired type and has null as its value. We provide a utility class `me.arkorwan.testng.utils.Null` that contains a single static generic method `value()` for this purpose. To continue with the `singletonList` example, instead of writing

```
List<Integer> list = G.singletonList(null);
```

users are encouraged to substitute `null` with a call to the `Null` class, as follows:

```
List<Integer> list = G.singletonList(Null.<Integer>
    value());
```

In the case of return type, we simply use the result type of the original invocation. This is not always the same as the return type of the original method, when the return type references type parameters. In our example, while the original method `singletonList` returns a value of type `List<T extends Number>` (which becomes `List<Number>` after type erasure), the result type of the invocation `G.singletonList(1)` is a `List<Integer>`.

Applying the above tactic to the example, the actual proxy method produce by our process is:

```
static List<Integer> m_singletonList_G_42(int
   Γitem$0⅃
   ) {
return G.singletonList(Γitem$0⅃ );
}
```

As for the effect of generics on field assignment, we simply resolve the generic field type to concrete type, and use the result as both the parameter type and return type of the proxy method. As an example, for the following generic field:

```
public class Foo<T>{
T field;
}
```

with this assignment:

```
Foo<Integer> f = new Foo<>();
f.field = 1;
```

our process would produce the following proxy method:

```
Integer f_field_Foo_42(Integer Γfield⅃ )  {
return ((Foo) ẏȧċȯṅTarget).singletonList(Γfield⅃ );
}
```

There is one limitation that we have found, concerning the combination of generics and exceptions.

### 4.6.3 Proxification process limitation on combination of generics and exceptions

There is one issue remain unresolved from the proxification process, concerning exceptions and generics. According to the Java specification, all subclasses of `java.lang.Throwable` cannot be generic [29, p. 186], so an expression like `throws CustomException<T>` is not allowed. However, there is still a way to refers to type parameter in the `throws` clause. Consider the following example:

```
static <E extends Exception> void f(E e) throws E {
  throw e;
}
```

This is a valid construct. The method simply throws the exception it receives. Also recall that checked exceptions need to be either caught or re-thrown, otherwise the compilation would failed. Suppose the above method is invoked with an exception subclass, and is handled by a catch clause, as in the following:

```java
try {
  f(new EOFException());
} catch(EOFException e) {
}
```

When this method is proxified, the process tries generating a `throws` clause to mirror the original method. Current implementation uses reflection to extract this information from the actual method implementation, so the `throws` clause of the proxy method will be '`throws Exception`'. The problem is that, once the invocation is replaced by the a call to the proxy method, the existing catch clause only catches the more specific exception subclass, so from the compiler's point of view there can be 'leaked' exceptions. Therefore the proxified code would fail to compile.

To solve this problem, the proxification process must be able to infer the correct exception type. This is impossible from the method invocation alone, so we need to search the enclosing scope for the correct catch or throws clause. Since this solution is a difficult process, and the problem only occurs for a very specific case, we decided to leave this as a known limitation for the time being.

### 4.6.4 Running Chicory from within a Java Program

Both of the recovery strategies we implemented rely on running the tool *Chicory*, the Daikon front-end for Java. Like all of Daikon various tools, it is intended to be used from a command-line interface. However, to fit the recovery strategies within the extraction framework, Chicory needs to be run from within a Java program. This should be a trivial task, that is, it could be accomplished by calling the `main()` method of the class `daikon.Chicory`. Then we could execute Chicory from within a Java program, supplying appropriate command line arguments and JVM properties, then either interpreting the resulting trace file or pass the information on to Daikon tool.

However, unlike some other tools in the Daikon ecosystem, Chicory is not designed to be run from other programs. The problem is that it always makes a call to `System.exit()`, which terminates the running JVM. Therefore, if Yacon directly calls the main method of Chicory, then when Chicory finishes, Yacon will terminate as well. Because in most cases the program terminates *successfully*, meaning that there is no error message being produced, it was quite difficult to detect the problem.

Once we knew the root cause, we identified two possible solutions:

1. Prevent`System.exit()` from terminating the JVM.

2. Run Chicory as an external process.

The first solution can be implemented by using a `SecurityManager` to detect a call to `System.exit()`, then throw an exception, to prevent the JVM from termination [54]. While this solution works quite well, we considered it as a hack, since it uses the bad practice of throwing Exceptions in a non-exceptional case. Moreover, we are not certain if there are any consequences of exploiting the `SecurityManager` in this way.

The second solution, in contrast, is more straightforward. Java already has several built-in mechanisms of dealing with external processes, such as the class `java.lang.Runtime` or `java.lang.ProcessBuilder`. So this is the solution we chose, using the `ProcessBuilder` class.

### 4.6.5 Reading Trace File Produced by Chicory in the Correct Encoding

Once again, both recovery strategies require reading trace information produced by Chicory. In the case of boundary value recovery strategy, the traces are needed to be read and processed by our tool, while in the case of test suite invariants recovery strategy, the traces are to be passed on to the tool Daikon.

In our development process we first focused on the second case, where we need to passed the trace information to Daikon. Normally Chicory produces a result trace into a file, which can be passed directly as an argument to Daikon. There exists an alternative, which is to specify the command line option `--daikon` or `--daikon-online` to instruct Chicory to run Daikon automatically. The difference for these 2 options is that, the first one is just a convenient way to run Chicory and Daikon consecutively, while in the second one the two tools are being run concurrently, with trace data being communicated incrementally via a socket, therefore does not involve writing or reading trace files.

We first attempted to use the option `--daikon` since it would be more convenient than running two tools by ourselves, and the online option is unnecessary. However, it was quickly found that the resulting invariants from this process are partially garbled, also known as '*mojibake*' [51], as shown in figure 4.2. This is most likely a text encoding problem, when a string of characters are converted into bytes using one encoding, and then

```
 Foo$áº È§Ä È¯á¹ Proxifier.mâ ¿gâ ¿Fooâ ¿1423(int
):::ENTER
 Foo$áº È§Ä È¯á¹ Proxifier.instance.target !=
null
 n == 2147483647
```

(a) read trace data from file

```
Foo$ẎȧċȯṅProxifier.m‿g‿Foo‿1423(int):::ENTER
Foo$ẎȧċȯṅProxifier.instance.target != null
n == 2147483647
```

(b) receive trace data from Chicory

Figure 4.2: Comparison of Daikon outputs from difference input sources

converted back to characters using a different encoding. We examined the trace file produced by Chicory and found that the output encoding is correct, therefore the defect must lie within the input reading of Daikon. Usually this would not be a big problem since most programs use only characters in ASCII range for class and method names. However, we deliberately used the uncommon Unicode characters *Ẏ, ȧ, ċ, ȯ, ṅ, ‿, ꭥ, Г* and *Ꟶ* in the source code transformation phase of the strategy. Therefore, while this approach works well in most circumstances, it becomes a big obstacle for our purposes.

We deduced that changing the method from running the `--daikon` option to separately running Chicory and Daikon would also suffer from the same problem, since it is essentially the same process. So instead we tried the other approach, using the command line option `--daikon-online`. As expect, because the process does not need reading data from the trace file, the mojibake does not occur, and we continued development of the strategy using this approach.

However, the problem itself was not solved, just evaded for the time. So when we turned our focus to the boundary value recovery strategy, the problem resurfaced. Since this strategy requires only trace data and not invariants, reading the trace file is inevitable. Fortunately, this is not as critical as in the other strategy, for the chance of encountering the problem is much lower since no Unicode characters are injected into the subject program. However, it would still be better to be able to avoid the problem completely.

```java
// Open the reader stream
if (raw_filename.equals("-")) {
  // "-" means read from the standard input stream
  Reader file_reader = new InputStreamReader(System.in, "ISO-8859-1");
  reader = new LineNumberReader(file_reader);
}
else if (raw_filename.equals("+")) { //socket comm with Chicory
  InputStream chicoryInput = connectToChicory();
  InputStreamReader chicReader = new InputStreamReader(chicoryInput);
  reader = new LineNumberReader(chicReader);
} else if (is_url) {
  URL url = new URL (raw_filename);
  InputStream stream = url.openStream();
  if (raw_filename.endsWith (".gz")) {
    GZIPInputStream gzip_stream = new GZIPInputStream (stream);
    reader = new LineNumberReader (new InputStreamReader (gzip_stream));
  } else {
    reader = new LineNumberReader (new InputStreamReader (stream));
  }
} else {
  reader = UtilMDE.lineNumberFileReader(raw_filename);
}
```

Figure 4.3: Part of Daikon's class `daikon.FileIO` responsible for reading trace files

Daikon follows its own recommendation of using the method `FileIO.read_data_trace_files` to read trace data, and this is indeed the source of the mojibake problem. We discovered the part of the source code that is responsible – the construction of the reader object is in the class `daikon.FileIO`, shown in figure 4.3[8]. It can be seen that there are mainly 4 sources of input.

1. The standard input. In this case the encoding is specifically set to ISO-8859-1 or Latin-1. (line 1418)

2. From Chicory. In this case the encoding is not explicitly stated. The reader is constructed using the constructor `InputStreamReader (InputStream)` (line 1423), which '[c]reates an InputStreamReader that uses the default charset', as stated in its documentation[9].

3. From a URL. This case is similar to the previous one – the reader is construct using the same constructor. (lines 1430 and 1432).

4. From a simple file. In this case the construction is deferred to a library call `UtilMDE`. (line 1435)

From our earlier tests we knew that case 2 succeeded, while case 4 failed, for unicode input. Because cases 2 and 3 construct the input reader in the same way, it is expected that the encoding problem should not happen in case 3. To make a reference to a file as a URL, one can simply use the file URI scheme [6] in the form `file://<file absolute path>`. We tried this solution and found that it succeeds as expected.

## 4.7 Summary

The implementation of Yacon, the prototype solution to the research question of using equivalence partitioning to assist dynamic invariants detection, was described in detail in this chapter. We introduced the data model classes, and the implementation of the extraction and translation phases of Yacon. Test coverage statistics of the solution was also provided. During development a number of issues came up. In the last section we recorded the problems and how they were solved, or why they remain unsolved.

[8]Source code and line numbers are from Daikon version 5.2.2

[9]`http://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html`

# 5 Evaluation

Yacon was developed as a tool to help answering our research question of 'how can equivalence partitioning assist dynamic detection of conditional invariants?' In this chapter we evaluate Yacon for its effectiveness as a solution to the research question. Also evaluated is the performance, in terms of execution time of the solution.

## 5.1 Effectiveness Evaluation

### 5.1.1 Experimental Setup

The process for effectiveness measurement is as the following outline:

1. Select a number of programs for evaluation. (Section 5.1.2)
2. Create a test suite for each program, aiming to achieve full branch and statement coverage, ensuring that all possible branches of each decision point and all statements are executed. We will refer to these test suites as the 'full coverage test suites'.
3. Create a partitioning file for each program manually. These files serve as the expected result of the extraction phase of Yacon.
4. Run Yacon to produce the intermediate partitioning file and the resulting splitter file for each program and each extraction strategy. This step is performed once using the full coverage test suites we wrote ourselves in step 2, and another using the test code from the textbook (we will refer to them as 'textbook test suites'), if available.

5. Compare how close the generated partitioning files are to the reference file. (Section 5.1.3)

6. For each pair of program and test suite, run Daikon with and without the help of Yacon, resulting in different sets of generated invariants. Evaluate each invariant for its *a*) correctness, *b*) usefulness, and *c*) significance. Use the result to calculate the effectiveness of Yacon. (Section 5.1.4)

See appendix D for the details of evaluation environment and configurations.

### 5.1.2 Program Selection

In total, 7 programs taken from programming textbooks are used for evaluation:

- `StackAr` and `QueueAr` from [58]
- `BinaryHeap` and `BinarySearchTree` from [59]
- `Earthquake` from [37]
- `ComputeTax` from [43]
- `Insurance` from [10]

The first two programs, `StackAr` and `QueueAr`, are included with standard Daikon releases. Prior work on Daikon often relied on them as benchmark programs, including work on conditional invariants [19, 21]. Together with `BinaryHeap` and `BinarySearchTree`, they are examples of recursive data structures, which naturally include conditional invariants, for they need to process the base case and the recursive case differently [21]. The textbooks also provide small demonstration programs that can be used as test suite.

The last three programs are selected because they have suitable structure for equivalence partitioning testing technique – that is, they contain code that process data in the same domains in different ways depending on values. `Earthquake` is the simplest among the three programs, with the output conditionally depends on only a single input domain, while `ComputeTax` and `Insurance` depend on more than one inputs. Unlike the data structure programs, there are no test suites provided for these.

### 5.1.3 Recovery Strategy Effectiveness

For each pair of sample program and test suite, we have written a reference partitioning file, and we have generated 2 partitioning files using Yacon, one for each recovery strategy. In this section we evaluate the effectiveness of these strategies.

#### 5.1.3.1 Metrics

Comparison of partitioning is a common problem in the area of data cluster analysis. A *clustering* is analogous to a partitioning, and a *cluster* is an equivalence class. There are a number of traditional metrics of comparing clusters in the literature [44], but most of them are designed to measure distance between disjoint clusters, equivalent to non-overlapping partitioning. Since Yacon allows for overlaps, we instead used the *best match algorithm* from [27], which is more suitable for overlapping clusterings.

The best-matched distance $d$ between two clusterings $C_1$ and $C_2$ is defined as:

$$d(C_1, C_2) = \sum_{i=1}^{n} \min_{j=1..m} \delta(S_i, S'_j) + \sum_{j=1}^{m} \min_{i=1..n} \delta(S'_j, S_i)$$

where $C_1$ consists of $n$ clusters $S_1..S_n$ and $C_2$ consists of $m$ clusters $S'_1..S'_m$. The *difference function* $\delta(S_i, S_j)$ computes difference between any two clusters, and must be defined in order for the definition of best-matched distance to be complete. In [27], two choices of this difference function are given, one is the number of moves necessary to convert one cluster to another, and the other is based on the concept of *entropy* from information theory. In both approaches, the size of clusters is a major component in the computation. However, we are more interested in the *structure* of the partitioning than the size, so we defined the difference function using only the relationship between clusters, as follows:

$$\delta(S_i, S_j) = \begin{cases} 0, & \text{if } S_i = S_j \\ w_s, & \text{if } S_i \subset S_j \text{ or } S_j \subset S_i \\ 1, & \text{otherwise} \end{cases}$$

where $0 < w_s < 1$.

In simple terms, the function evaluates to 0 if the two clusters are perfect match, $w_s$ if there is a hierarchical relationship between them, and 1 if there is not.

Using this definition of difference function, the best-matched distance is 0 for two identical clusterings, and $n + m$ for two maximally different clusterings with $n$ and $m$ clusters respectively. We can further define the *normalised best-matched distance* $d_{\text{normal}}$ by dividing the best-matched distance by $n + m$, so that the result will always be a number between 0 and 1 inclusive, a form more suitable for comparison.

For the evaluation of our generated partitioning files, we measure the normalised best-matched distance for each of the domains present in the reference partitioning, using 0.5 as the value of $w_s$. If a partitioning is missing from the generated file, we use the domain itself as the only equivalence class. Notice that in this case, the best-matched distance will always be $w_s$. This means that the value 0.5 can serve as the threshold that separates good and bad partitionings – a good one should have its distance less than 0.5.

We further define the **recovery effectiveness** of a partitioning as follows:

$$\text{effectiveness} = 1 - 2 \cdot (\text{average best-matched distance of all domains})$$

The effectiveness ranges from $-1$ to 1. A perfect match has the effectiveness of 1, empty partitioning has the effectiveness of 0, and maximally mismatched partitioning has the effectiveness of -1.

#### 5.1.3.2 Experimental Results

We measure the recovery effectiveness of the boundary value strategy, the test suite invariants strategy, and the combination of them (the combined partitioning consists of the union of equivalence classes in the same domain from both strategies).

The results for full coverage test suites and textbook test suites are presented in figures 5.1 and 5.2, respectively.

It can be seen that, for the full coverage suites:

- There is no dominant strategy – out of 7 sample programs, boundary value generates better result for 3 of them, and test suite invariants is better for the rest. In 3 instances (`Earthquake`, `BinaryHeap` and `BinarySearchTree`), one strategy produced some results while the other failed completely.
- The combination of the two strategies created the best result most of the time (6 out of 7).

Overall, the combined strategy is effective to varying degrees for the manually-written suites, with the values ranging from 0.05 to 1.0. In contrast, the result for textbook test suites are poor: effectiveness of all

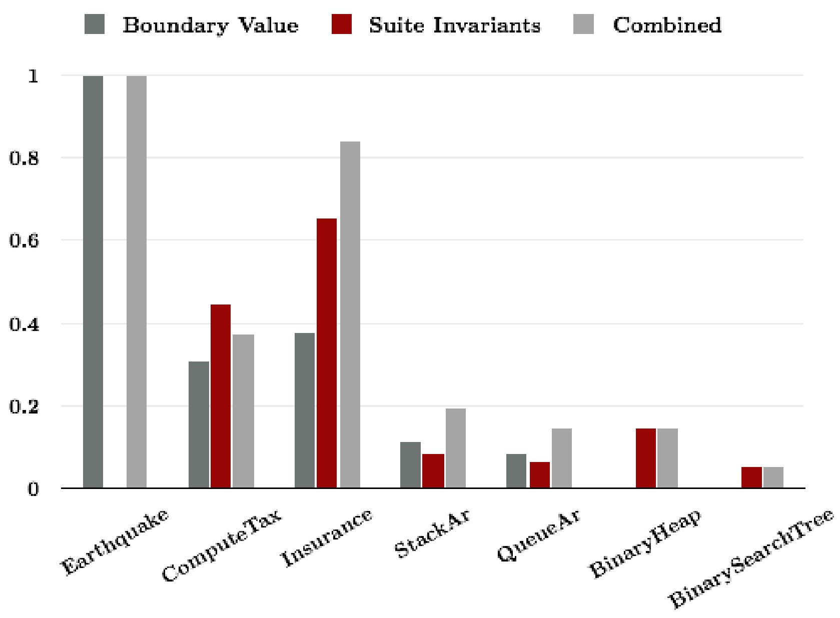

Figure 5.1: Recovery effectiveness for full coverage test suites

| | Recovery effectiveness | | |
|---|---|---|---|
| Program | Boundary Value | Suite Invariants | Combined |
| StackAr | 0.0 | 0.0 | 0.0 |
| QueueAr | 0.0 | 0.0 | 0.0 |
| BinaryHeap | 0.0 | 0.0 | 0.0 |
| BinarySearchTree | 0.0 | 0.0 | 0.0 |

Figure 5.2: Recovery effectiveness for textbook test suites

the strategies executed on all 4 programs are measured as 0.0, meaning that they are ineffective for these test suites. We found 2 possible explanations for this performance:

1. The 4 data structure programs are not well-suited for equivalence partitioning technique. This is supported by the fact that the effectiveness for these 4 programs are also lower than the other 3 on the manually-written test suites. The conditional processing for these programs mostly depends on properties of underlying data storage rather than the input/output domains themselves. And while the partitioning file format supports conditions that depend on other domains outside of the input/output, current implementation of recovery strategies is not able to produce this kind of conditions yet.

2. In our opinion, the textbooks ‘test suites’ are not test suites in the traditional software testing sense. Their main purpose is to demonstrate usage of the data structures in various different situations, rather than to find program defects or ensure correctness of the program. With this different mindset, our assumption that equivalence partitioning is usually used when writing test suites might be invalid for these cases.

To prove these claims we need more data, which could be obtained from applying the evaluation process on other programs suitable for equivalence partitioning with ‘traditional test suites’.

### 5.1.4 Invariants Detection Effectiveness

In this section we measured the effectiveness of Yacon as a solution, to see if it can assist Daikon on invariants detection.

For each subject program and test suite, Daikon was run three times with different inputs, producing three sets of invariants:

- *Baseline* - the invariants produced by Daikon without assistance of any splitter files.
- *Actual* - the invariants produced by Daikon with the two splitter files generated by Yacon, one file from each recovery strategy.
- *Reference* - the invariants produced by Daikon with the splitter file generated from running Yacon on the reference partitioning. This represents the potential of the idea of using domain partitioning to assist invariants discovery, regardless of the effectiveness of recovery strategies used.

#### 5.1.4.1 Metrics

To measure the effectiveness of a set of invariants, each invariant is manually examined for their correctness, usefulness, and relevance, defined as follows:

- **Correctness:** an invariant is correct if it is always true. Incorrect invariants are a symptom of *overfitting* – they are true for all the present data, but not for some of the absent ones. They can be eliminated by adding counterexamples to the test cases.

- **Usefulness:** an invariant is useful if it helps programmers, testers, or any other parties in some ways.[1]
- **Relevance:** an invariant is relevant if it can demonstrate a unique characteristic or objective of the program.

These definitions are inherently subjective. Therefore we developed the following guidelines to make the judgements as objective and consistent as possible:

- A useful invariant is required to be correct.
- A relevant invariant is required to be useful.
- An invariant is relevant if it meets our pre-determined expectation.
- For any two equivalent invariants, only one of them can be relevant.
- A correct invariant that can be implied by another non-equivalent invariant is useful but not relevant.
- An invariant that is obviously true (for example, $x = x$, or $x > 0 \rightarrow x > 1$) is not useful.
- An invariant that relates properties that should not be relate (for example, relationship between the array size and the content of the array) , even when correct, is not useful.
- A conditional invariant whose condition clause is always true (for example, $(x = 0 \vee x \neq 0) \rightarrow y > 0$) is not useful.

The following metrics are then defined for a set of invariants:

- **Correctness** – the ratio of correct invariants by reported invariants.
- **Usefulness** – the ratio of useful invariants by reported invariants.
- **Precision** – the ratio of relevant invariants by reported invariants. This is a well-known metric in the field of information retrieval [11].
- **Recall** – the ratio of relevant invariants by expected invariants. Another well-known metric from information retrieval.

[1]This definition is close to *relevance*, as defined by Ernst in [20].

Out of these 4 metrics, the most important for our situation is *recall*, since the purpose of Yacon is to help recover invariants missing from normal Daikon execution – in other words, we want to improve the recall value. So we will use recall as the main indicator of invariants detection effectiveness, while the other metrics serve as indicators of how the effect this process might have created to the quality of generated invariants.

#### 5.1.4.2 Experimental Results

We measure the correctness, usefulness, precision, and recall of each set of invariants generated from Daikon. Note that some of the sample programs produced invariant sets too large for our manual assessment. We instead limit our analysis of these sample programs to only a subset of invariants associated with a selected method. The subjects for analysis and their size, in terms of number of methods and number of instructions[2] are presented in the following table:

| | Program Size | | | Selected Size | |
|---|---|---|---|---|---|
| Program | lines | methods | Selected Methods | lines | methods |
| Earthquake | 46 | 1 | (all) | | |
| ComputeTax | 244 | 1 | (all) | | |
| Insurance | 45 | 1 | (all) | | |
| StackAr | 117 | 9 | (all) | | |
| QueueAr | 122 | 9 | `enqueue (Object)` | 26 | 1 |
| BinaryHeap | 247 | 12 | `percolateDown (int)` | 56 | 1 |
| BinarySearchTree | 245 | 16 | `find (T, BinaryNode<T>)` | 25 | 1 |

We counted the number of reported invariants, then measured the correctness, usefulness, precision, and recall of each set of invariants. The results are presented in the tables in figures 5.3 to 5.7.

We made the following observations from the results:

- The Yacon and Reference invariant sets are almost always larger than the Baseline set. This means our solution is *successful* in creating more candidates.

[2]As reported by the code coverage tool `eclEmma`.

| Program | Reported Invariants | | |
|---|---|---|---|
| | Baseline | Yacon | Reference |
| **full coverage test suites** | | | |
| Earthquake | 1 | 15 | 15 |
| ComputeTax | 22 | 206 | 151 |
| Insurance | 2 | 84 | 54 |
| StackAr | 140 | 269 | 311 |
| QueueAr | 26 | 109 | 199 |
| BinaryHeap | 20 | 91 | 61 |
| BinarySearchTree | 4 | 11 | 18 |
| **textbook test suites** | | | |
| StackAr | 97 | 123 | 205 |
| QueueAr | 24 | 24 | 156 |
| BinaryHeap | 30 | 88 | 88 |
| BinarySearchTree | 9 | 18 | 23 |

Figure 5.3: Number of reported invariants of Baseline, Yacon, and Reference test sets

| Program | Correctness | | |
|---|---|---|---|
| | Baseline | Yacon | Reference |
| **full coverage test suites** | | | |
| Earthquake | 100.00% | 100.00% | 100.00% |
| ComputeTax | 100.00% | 91.75% | 98.68% |
| Insurance | 100.00% | 63.10% | 68.52% |
| StackAr | 57.86% | 55.02% | 56.59% |
| QueueAr | 80.77% | 45.87% | 59.80% |
| BinaryHeap | 75.00% | 76.92% | 83.61% |
| BinarySearchTree | 100.00% | 100.00% | 88.89% |
| **textbook test suites** | | | |
| StackAr | 68.04% | 73.98% | 63.41% |
| QueueAr | 83.33% | 83.33% | 73.08% |
| BinaryHeap | 60.00% | 82.95% | 75.00% |
| BinarySearchTree | 66.67% | 72.22% | 78.26% |

Figure 5.4: Correctness comparison of Baseline, Yacon, and Reference test sets

| | Usefulness | | |
|---|---|---|---|
| Program | Baseline | Yacon | Reference |
| **full coverage test suites** | | | |
| Earthquake | 100.00% | 100.00% | 100.00% |
| ComputeTax | 81.82% | 58.25% | 56.95% |
| Insurance | 50.00% | 39.29% | 40.74% |
| StackAr | 57.14% | 41.64% | 54.66% |
| QueueAr | 80.77% | 33.03% | 37.19% |
| BinaryHeap | 75.00% | 72.53% | 72.13% |
| BinarySearchTree | 100.00% | 45.45% | 61.11% |
| **textbook test suites** | | | |
| StackAr | 65.98% | 66.67% | 55.61% |
| QueueAr | 83.33% | 83.33% | 42.31% |
| BinaryHeap | 46.67% | 67.05% | 59.09% |
| BinarySearchTree | 66.67% | 33.33% | 56.52% |

Figure 5.5: Usefulness comparison of Baseline, Yacon, and Reference test sets

| | Precision | | |
|---|---|---|---|
| Program | Baseline | Yacon | Reference |
| **full coverage test suites** | | | |
| Earthquake | 0.00% | 46.67% | 46.67% |
| ComputeTax | 13.64% | 1.46% | 1.99% |
| Insurance | 0.00% | 17.86% | 27.78% |
| StackAr | 26.43% | 14.13% | 12.22% |
| QueueAr | 19.23% | 4.59% | 3.02% |
| BinaryHeap | 30.00% | 6.59% | 9.84% |
| BinarySearchTree | 50.00% | 18.18% | 16.67% |
| **textbook test suites** | | | |
| StackAr | 28.87% | 22.76% | 14.63% |
| QueueAr | 20.83% | 20.83% | 3.85% |
| BinaryHeap | 23.33% | 7.95% | 7.95% |
| BinarySearchTree | 22.22% | 11.11% | 13.04% |

Figure 5.6: Precision comparison of Baseline, Yacon, and Reference test sets

| | Recall | | |
|---|---|---|---|
| Program | Baseline | Yacon | Reference |
| **full coverage test suites** | | | |
| Earthquake | 0.00% | 100.00% | 100.00% |
| ComputeTax | 11.11% | 11.11% | 11.11% |
| Insurance | 0.00% | 62.50% | 62.50% |
| StackAr | 97.37% | 100.00% | 100.00% |
| QueueAr | 62.50% | 62.50% | 75.00% |
| BinaryHeap | 60.00% | 60.00% | 60.00% |
| BinarySearchTree | 50.00% | 50.00% | 75.00% |
| **textbook test suites** | | | |
| StackAr | 73.68% | 73.68% | 78.95% |
| QueueAr | 62.50% | 62.50% | 75.00% |
| BinaryHeap | 70.00% | 70.00% | 70.00% |
| BinarySearchTree | 50.00% | 50.00% | 75.00% |

Figure 5.7: Recall comparison of Baseline, Yacon, and Reference test sets

- Let $R(S)$ represents the recall metric of an invariant set $S$. It can be seen that, for the same program/test suite, $R(\text{Baseline}) \leq R(\text{Yacon}) \leq R(\text{Reference})$. This means our solution always produces at least as good recall as the baseline. In the worst cases, $R(\text{Baseline}) = R(\text{Yacon}) = R(\text{Reference})$ – which means the help of splitter files failed to discovered more relevant invariants, but nothing is missing. In the best cases, all the relevant invariants are discovered by our recovery strategies.

- We inspected each of the additional recovered invariants found by the Reference set but not the Baseline, and found that all of them are conditional invariants, as expected.

- The Reference invariant sets successfully produced better recall than the baseline for 8 out of 11 cases. The Yacon sets performed worse, successfully producing better result for only 3 out of 11 cases. This seems to suggest that the equivalence partitioning approach can be useful, but our current recovery strategies are still inadequate.

- Correctness, usefulness, and precision of Yacon and Reference invariant sets are worse comparing to the Baseline set. This suggests that, by putting more candidates in order to uncover relevant invariants, we also let in even more irrelevant invariants, thus worsening these metrics. For precision and recall, it has been proved that trying to improve one metric will inherently worsen the other [11]. So this is an expected trade-off.

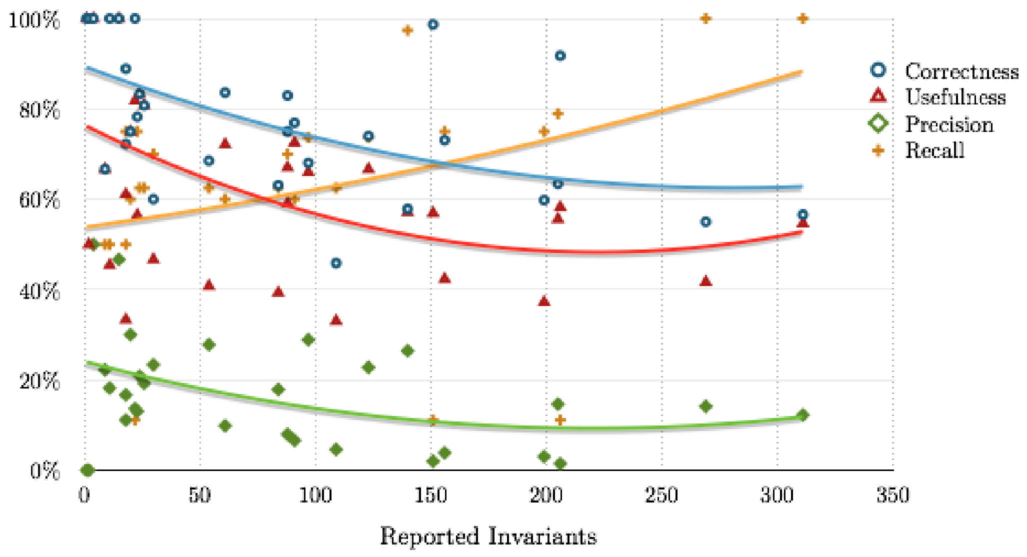

Figure 5.8: Trends of correctness, usefulness, precision, and recall, by number of reported invariants

To better visualise this relationship between recall and the other three metrics as we had speculated, we created a scatter plot relating each metric to number of reported invariants, with quadratic trend lines. (Figure 5.8). The graph shows how each recall improves with more invariants, while the other metrics worsens.

### 5.1.5 Correlation between Effectiveness of Recovery Strategy and Invariants Detection

One of our interpretation of the results in the previous section states that the equivalence partitioning approach has potential to be successful, but current implementation of the extraction phase is not able to realise that potential yet. In other words, better partitioning should lead to better invariant detection. This hypothesis is tested in this section.

We evaluated partitioning by measuring its *recovery effectiveness* (page 66). For evaluation of invariant detection, we utilised the discovered fact that recall of Yacon invariant set is always between those of Baseline set and Reference set (page 73). to define the *detection improvement coefficient*, as follows:

$$\text{improvement coefficient} = \frac{R(\text{Yacon}) - R(\text{Baseline})}{R(\text{Reference}) - R(\text{Baseline})}$$

where $R(S)$ represents the recall metric of an invariant set $S$. Note that if $R(\text{Reference}) = R(\text{Baseline})$, which is true in our evaluation for 3 out of 11 cases, then the improvement coefficient becomes $\frac{0}{0}$ – an indeterminate form. This is sensible to us since in this case there is no way to tell how much improvement the Yacon invariant set has over the baseline. Therefore we left them out of this analysis. For the other cases, we created a scatter plot correlating the detection improvement coefficient with the recovery effectiveness of the combined strategy, resulting in figure 5.9.

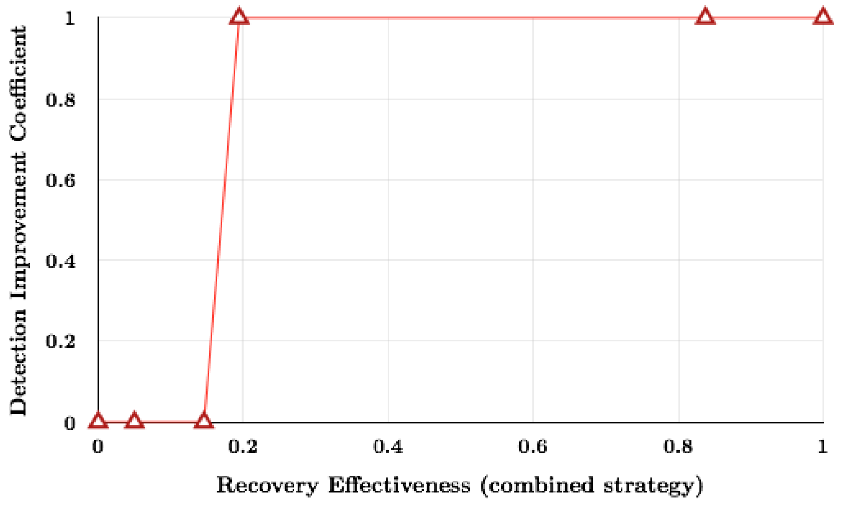

Figure 5.9: Correlation between improvement coefficient and recovery effectiveness

The correlation coefficient between these two variables is measured to be 0.8175, indicating strong positive correlation, seemingly confirming our hypothesis. What we did not expect is the strong polarity of the improvement coefficient – it is either 0 or 1, implying that the Yacon invariant set either achieves full improvement or not improving at all. There seems to be a tipping point on the recovery effectiveness, at around 0.15 - 0.20. We expected to see a more linear relationship, with the improvement coefficient gradually rises as the recovery effectiveness increases. Upon further analysis, we believe this is a symptom of small sample sizes. For half of the cases, the Reference invariant set managed to improve recall by discovering exactly 1 more relevant invariant, effectively limiting the possibility of improvement coefficient to be either 0 or 1, as we have seen. Because of this small sample size, we conclude that, while there is an evidence suggesting that better partitioning indeed leads to better invariant detection, it needs more experimental evidence on larger sample size for this hypothesis to be conclusive.

## 5.2 Performance Evaluation

Using the same settings and evaluation process as in the effectiveness evaluation, we also measured the execution time of Yacon, to assess the cost of our solution in terms of time resource.

For each pair of sample program and test suite we executed the effectiveness evaluation process 5 times, and measured the runtime of each step in the process. We are interested the execution time of *Yacon test set* in comparison with *baseline test set*.

The Yacon test set takes of the following steps:

1. Run Yacon extraction using test suite invariants recovery strategy.
2. Run Yacon translation on the partitioning generated by test suite invariants recovery strategy.
3. Run Yacon extraction using boundary Value recovery strategy. This step includes running Chicory as part of the strategy.
4. Run Yacon translation on the partitioning generated by boundary value recovery strategy.
5. Run Daikon.

The baseline test set takes of the following steps:

1. Run Chicory.
2. Run Daikon.

Notice that we optimised the Yacon process by reusing Chicory trace database for both boundary value extraction and Daikon.

### 5.2.1 Experimental Results

Figures 5.10 and 5.11 show the comparison of average runtime for each test set.

Figure 5.12 shows the average runtime of all programs and test suites, with details on how much time was spent in each phase. Note that translation and extraction of the same recovery strategy are grouped together, and Chicory is excluded from the runtime of boundary value recovery strategy.

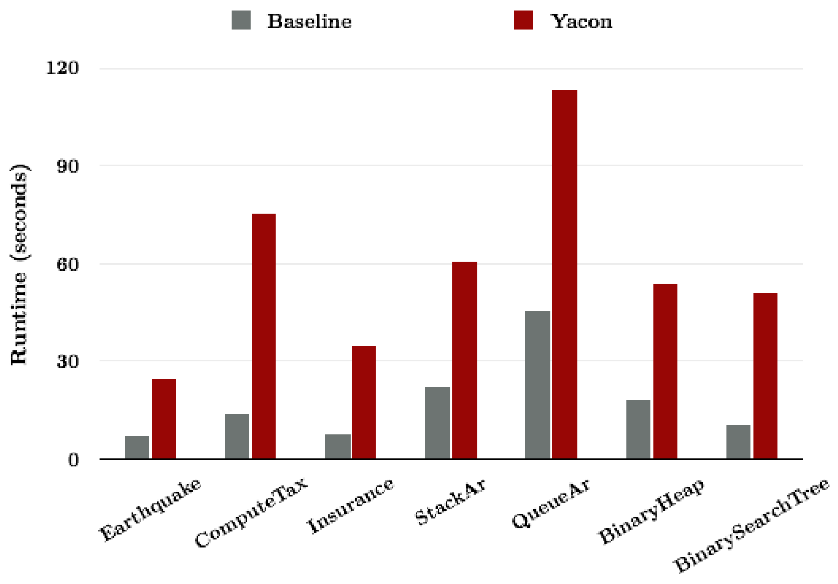

Figure 5.10: Runtime comparison for full coverage test suites

We made the following observations:

- Yacon test sets run about 1.5-5 times slower than the baseline sets. On average the ratio is about 3.25.
- Total runtime of the Yacon test set is dominated by running Daikon and Chicory with splitter files, rather than the time spent in producing those files.
- There is no difference in the runtime of Chicory.
- Daikon runs almost twice slower in the Yacon test set. This can be attributed to the time it needs to process splitter files.
- Test suite invariants strategy takes more time than boundary value strategy, running almost 3 times slower. This is an effect of the optimisation of reusing Chicory trace database. Without the optimisation the runtime of both strategies would be very close.

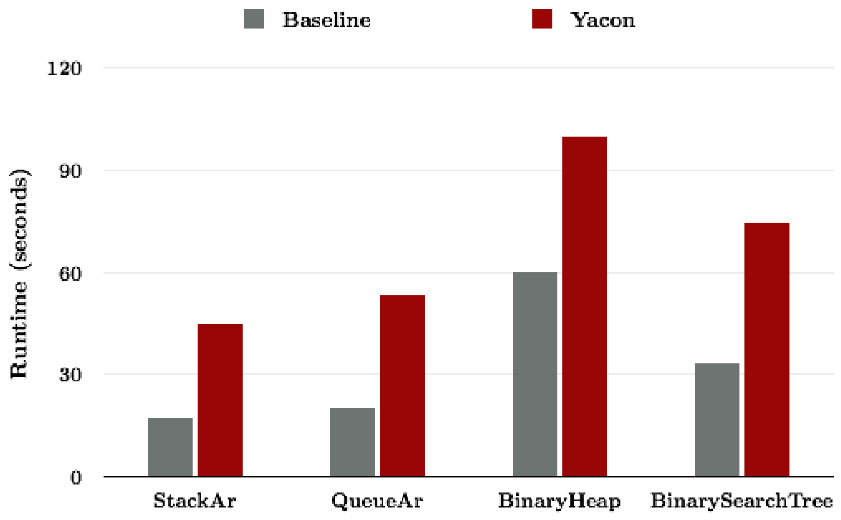

Figure 5.11: Runtime comparison for textbook test suites

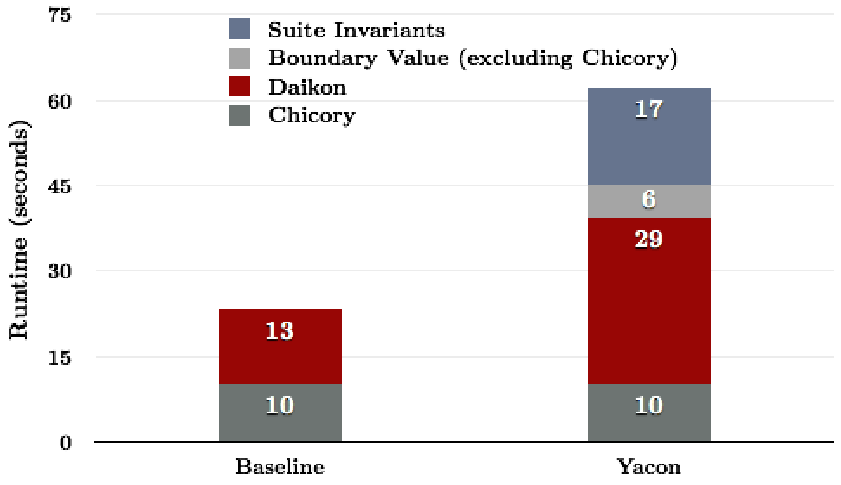

Figure 5.12: Runtime distribution

It can be concluded that our attempt to improve Daikon, by giving it more splitting conditions to process, resulted in 1.5-5 times the slower runtime. This emphasises the importance of generating good partitioning, since a useless splitting condition would slow down the process unnecessarily.

## 5.3 Threats to Validity

This section identifies the threats to the validity of our evaluation.

**Small sample size.** The evaluation was done on 7 textbook programs and 11 test suites. These small sizes of samples reduce the confidence level of our result from statistical point of view.

**Small and unrealistic sample programs.** All our sample programs are taken from textbooks. They are considered small and unrealistic, compared with the industrial standard. This makes it difficult to generalise our result to larger, more realistic programs usually found in the industry.

**Subjectivity of invariants assessment.** The assessment of correctness, usefulness, and relevance of generated invariants were done manually using our own judgement. Despite our best efforts to make the assessment as objective and consistent as possible, this process is inherently subjective, and different assessor could produce different results.

**Selection bias.** 3 of 7 sample programs are selected because their characteristics fit our expectation of good candidates for our solution. This possible over-representation of good candidates could make our evaluation more biased toward good results.

**Test suite construction bias.** We created the full coverage test suites ourselves. Since at the time we wrote these tests we already knew exactly how our solution operates. This could make the test suites we developed becoming unnatural, specially tuned to work for our solution.

## 5.4 Summary

We have presented the evaluation of Yacon, for its ability to answer the research question. We found that the idea of using equivalence partitioning to assist discovery of invariants has potential to be successful, as shown by the increasing invariants detection effectiveness (section 5.1.4). However, the two recovery strategies we devised, namely test suite invariants strategy and boundary value strategy, can only raise the effectiveness in 3 out of 11 test cases, leaving much room for improvement (section 5.1.3). We also found that Yacon has an effect on the overall quality of detected invariants, reducing the level of correctness, usefulness, and precision, as well as increasing the runtime by a factor of 1.5 to 5 (section 5.2). Lastly, we identified a number of threats to the validity of our study.

# 6 Conclusion

In this last chapter we draw our conclusions, identifying contributions being made and suggesting possible future work related to this project.

## 6.1 Results and Contribution

The motivation of this work is to combine together the two approaches to ensure software dependability – testing and formal verification, using one to assist the other. We ended up with a research question that focuses on the area of *dynamic invariant detection*. It asks how a software testing technique of equivalence partitioning could assist detecting a special kind of invariants – the conditional ones.

We answered the research question by designing and developing Yacon, a prototype solution that works with a mature dynamic invariant detection tool Daikon. In the process we also created a file format called 'partitioning file' that can represent partitions information of Java programs. Yacon works by first creating these partitioning files from a given test suite, then translating them to a format Daikon can process.

We introduced two strategies to create partitioning files from test suites. One is to look for the existence of boundary values, the other is to look for reuse of function call in the test code. We argued how these are possible indicators of domain partitioning.

Upon evaluating the prototype solution, we found that there is potential in the concept of using partitioning information to uncover more conditional invariants. We also conceded that the two strategies we created have not fulfil that potential yet, as they only work in limited condition at best. Also learned is the effect of attempting to discover more relevant invariants – the overall quality of generated invariants worsens and Daikon takes longer time to process.

In conclusion, our contribution consists of:

- A new approach to help detecting conditional invariants, by using information from equivalence partitioning in test suites.
- Two partitions recovery strategies - the boundary value strategy and the test suite invariants strategy.
- Yacon, a prototype implementation of the approach.

While our implementation admittedly left much to be desired, we believe this work has addressed the research question and motivation, by showing that it is possible to use information from test suites to assist conditional invariant detection.

## 6.2 Future Work

In retrospect, this project was much more complicated than we initially expected. We imagined converting form partitioning information into splitting conditions a straightforward process. In reality, the complexity of the solution and the evaluation has led us to explore a number of concepts unfamiliar to us, such as source code manipulation. So it was a good learning experience.

On the other hand, we also recognised a number of areas that could have been done better. Here we suggest some future work that could improve this work.

### 6.2.1 Automatic/Objective Assessment of Yacon

A couple of the threats to validity of this work are related to the assessment of each invariant – it was done manually using our own judgement. One drawback of this approach is that it introduces subjectivity to the evaluation; another is that it limits the size of the sample programs, since using manual approach we could not assess too many invariants. A more objective and/or automatic assessment of generated invariants would help improve the validity of this work immensely (or invalidate it).

### 6.2.2 New Recovery Strategies

As stated previously, our two recovery strategies only perform well in limited circumstances. This leaves room for other possible recovery strategies to fill. A mechanism to add more recovery strategies to Yacon is already implemented (see section 4.3.3) to support future attempt to explore this direction.

### 6.2.3 Comparative Assessment of Yacon against Other Splitting Policies

As described in section 2.3, there are a number of existing approaches to help Daikon detect conditional invariants, collectively known as *splitting policies*. Yacon falls into this same category. It is therefore possible to compare the effectiveness of Yacon against these other policies to see which would perform better in which circumstances.

### 6.2.4 Adapting Yacon to Different Tools

Daikon is not the only tool in the field of conditional invariants detection. It is possible to adapt the idea of Yacon to other tools, such as those introduced in section 2.2. One might attempt to create a bridge to adapt Yacon result, either from the intermediate partitioning file or the splitter files, to work with these other tools.

# Appendices

# A Mathematics Notions

These definitions and notations are used throughout this document.

## A.1 Equivalence Relation, Equivalence Class, and Partition

For the following definitions, $S$ is an arbitrary set and $EQ$ is a binary relation on $S$.

**Definition A.1.** $EQ$ is an **equivalence relation** on $S$, if it has all the following properties:

1. *Reflexivity*: $\forall a \in S, (a, a) \in EQ$.
2. *Symmetry*: $\forall a, b \in S, (a, b) \in EQ \Rightarrow (b, a) \in EQ$, and
3. *Transitivity*: $\forall a, b, c \in S, (a, b) \in EQ \wedge (b, c) \in EQ \Rightarrow (a, c) \in EQ$.

**Definition A.2.** The **equivalence class** of $a$, where $a$ is a member of $S$, denoted by $[a]_{EQ}$, is the subclass of $S$ that contains all members that are equivalent to $a$. Formally, $[a]_{EQ} = \{x \in S : (a, x) \in EQ\}$.

**Definition A.3.** A **partition** of $S$ is a set of *disjoint* nonempty subsets of $S$, where the union of all these subsets is $S$ itself.

**Theorem 1.** *Given an equivalence relation $EQ$ on $S$, then the set of equivalence classes of $EQ$ is a partition of $S$. Conversely, given a partition of $S$, there exists an equivalence relation that has each member of the partition as its equivalence classes.*

See also, [53].

## A.2 Intervals

An *interval* is a set of numbers between two fixed numbers called endpoints. The endpoints might or might not be included in the interval. In this document, the following terms an notations are used to classify different types of intervals between $a$ and $b$:

- Open interval, denoted $(a, b)$ - both endpoints are not in the interval.
- Closed interval, denoted $[a, b]$ - both endpoints are in the interval.
- Left-open, right-closed interval, denoted $(a, b]$ - the maximum endpoint is in the interval, but the minimum is not.
- Left-closed, right-open interval, denoted $[a, b)$ - the minimum endpoint is in the interval, but the maximum is not.

B

# Partitioning File Format Syntax

The syntax of the partitioning file format can be described by a formal grammar. These conventions are used:

- `[x]` denotes zero or one occurrences of `x`
- `x*` denotes zero or more occurrences of `x`
- Terminals (strings that are not changed by the grammar rules) are enclosed with a pair of single quotes.

**IMPORTS Section**

⟨*partitioning*⟩ ::= ⟨*newline*⟩* [⟨*imp-section*⟩] (⟨*newline*⟩* ⟨*par-section*⟩)* (⟨*newline*⟩* ⟨*class-section*⟩)* ⟨*newline*⟩*

**IMPORTS Section**

⟨*imp-section*⟩ ::= ⟨*imp-declaration*⟩ ⟨*assignment*⟩* ⟨*newline*⟩

⟨*imp-declaration*⟩ ::= 'IMPORTS' ⟨*newline*⟩

⟨*assignment*⟩ ::= ⟨*alias*⟩ '=' ⟨*fully qualified class name*⟩ ⟨*newline*⟩

**PARTITION Section**

⟨*par-section*⟩ ::= ⟨*par-declaration*⟩ ⟨*eq-class*⟩* ⟨*newline*⟩

⟨*par-declaration*⟩ ::= 'PARTITION' ⟨*name*⟩ ':' ⟨*domain type*⟩ ⟨*newline*⟩

⟨*eq-class*⟩ ::= ( ⟨*predicate-class*⟩ | ⟨*interval-classes*⟩ | ⟨*complement-class*⟩ ) ⟨*newline*⟩

⟨*predicate-class*⟩ ::= 'EQClass' ⟨*predicate*⟩

⟨*interval-classes*⟩ ::= 'IntervalClasses' [ 'Func' '[' ⟨*transform*⟩ ']' ':' ⟨*domain type*⟩] ⟨*interval-mode*⟩ '(' ⟨*interval*⟩ ( ',' ⟨*interval*⟩)* ')'

⟨*interval-mode*⟩ ::= 'Minima' | 'Maxima' | 'Mixed'

⟨*complement-class*⟩ = 'ComplementClass'

**CLASS Section**

⟨*class-section*⟩ ::= ⟨*class-declaration*⟩ ⟨*class-ppt*⟩* ⟨*method-ppt*⟩* ⟨*newline*⟩

⟨*class-declaration*⟩ ::= 'CLASS' ⟨*domain type*⟩ ⟨*newline*⟩

⟨*class-ppt*⟩ ::= ⟨*field-declaration*⟩

⟨*method-ppt*⟩ ::= ⟨*method-declaration*⟩ [⟨*dependencies*⟩]

⟨*method-declaration*⟩ ::= 'METHOD' ⟨*method name*⟩ '(' ⟨*argument-list*⟩ ')' ':' ⟨*return-type*⟩ ⟨*newline*⟩

⟨*return type*⟩ ::= 'void' | ⟨*partition-type*⟩

⟨*dependencies*⟩ ::= 'DEPENDS' ⟨*newline*⟩ ⟨*field-declaration*⟩*

⟨*argument-list*⟩ ::= $\epsilon$ | ⟨*argument*⟩ ( ',' ⟨*argument*⟩)*

⟨*argument*⟩ ::= ⟨*arg name*⟩ ':' ⟨*partition-type*⟩

⟨*field-declaration*⟩ ::= ( 'STATIC' | 'INSTANCE' ) ⟨*field name*⟩ ':' ⟨*partition-type*⟩ ⟨*newline*⟩

⟨*partition-type*⟩ ::= ⟨*domain type*⟩ [ '{' ⟨*partition name*⟩ ( ',' ⟨*partition name*⟩)* '}']

# C Building and Running Yacon

## C.1 Building Yacon form Sources

First, ensure that the build machine meets the following criteria:

- It has *Apache Maven* installed. We use version 3.2.5 in our tests, but older versions should work too.
- Daikon (version $\geq$ 5.2.0) must be set up properly, with the variable `DAIKONDIR` pointing to Daikon's root directory.

To build Yacon form sources:

1. Open the command line interface and navigate to the project root folder – the folder containing `pom.xml`.
2. Since Yacon is using a custom, non-released version of the Spoon library, it needs to be installed locally. To do this, type in the following command:

   ```
   mvn install:install-file -Dfile=lib/spoon-core-4.0-
   SNAPSHOT-jar-with-dependencies.jar
   -DpomFile=lib/spoon-pom.xml
   ```

   Note that subsequent build attempts on the same build machine can skip this step.
3. Build the package using the following command:

```
mvn package -Ddaikon.dir=$DAIKONDIR
```

This compiles the source files, runs TestNG test suite, and packages the solution into 2 jar files, one with all the dependencies and one without. Note that Daikon itself, despite being a dependency, is assumed to be installed separately and not included in any of the jar files.

## C.2 Running Yacon

Yacon is intended to be used from the command line interface, similar to all tools shipped with Daikon. The command to execute the solution takes the following form:

```
java daikon.Yacon <action> <configuration> [overrides...]
```

Also specify Java options, such as classpaths, as necessary.

<action> refers to the mode of execution, which can be 'extract' for the extraction process, or 'translate' for the translation process.

<configuration> is a reference to the main configuration file.

Users can also override values provided by the configuration file, by specifying any number of `<configuration key>=<value>` as subsequent arguments.

## C.3 Configuration

For the configuration files for Yacon, we use the HOCON format and library, developed by Typesafe Inc [57]. HOCON is a superset of JSON that is more flexible, and is claimed to be more convenient for human users to edit, making it more suitable for configuration file.

HOCON configurations are basically a number of nested key-value pairs. All the configuration values for Yacon are listed as follows:

1. General configuration:

    - **`yacon.extraction.output_file`** (String) Output partitioning file of the extraction phase.
    - **`yacon.extraction.strategy.name`** (String) Name of the extraction strategy. This value is used as the top-level name for

the strategy-specific configurations – so it must be 'boundary_-values' for boundary value recovery Strategy, and 'suite_invariants' for test suite invariants recovery strategy.

- **`yacon.extraction.strategy.class`** (String) Fully qualified name of a custom extraction strategy. It is ignored if any of the two built-in strategies are used.
- **`yacon.translation.input_file`** (String) Input partitioning file of the translation phase.
- **`yacon.translation.output_file`** (String) Output splitter info file of the translation phase.

2. Configuration for boundary value recovery strategy:

    - **`boundary_values.chicory_args`** (String) Arguments to be passed to Chicory.
    - **`boundary_values.trace_name`** (String) Name (without extension) of the trace file produced by Chicory.
    - **`boundary_values.adj_predicates`** (Object array) List of predicate objects. A predicate object has two subkeys: `predicate_-class` – the fully qualified name of the `AdjacencyPredicate` subclass, and `target_type` – name of the type on which this predicate is applied.
    - **`boundary_values.patterns`** (String array) Target classes to be analysed by the strategy, in glob-like *wildcard* format.
    - **`boundary_values.second_order_trigger`** (integer) If the number of possible boundaries detected is not lesser than this configuration value, then the *second-order boundaries* mechanism is applied. Use negative value to turn this off completely.
    - **`boundary_values.floating_point_tolerance.equality`** (double) the equality tolerance $\epsilon_0$. This is a constant rather than the originally designed function.
    - **`boundary_values.floating_point_tolerance.adjacent`** (double) the adjacency tolerance $\epsilon_1$. This is a constant rather than the originally designed function.

3. Configuration for test suite invariants recovery strategy:

    - **`suite_invariants.source_folders`** (String array) List of source folders containing Java source files to be analysed.

- **`suite_invariants.source_patterns`** (String array) Java source files to be analysed, in glob-like *wildcard* format.
- **`suite_invariants.target_folder`** (String) Folder to store the proxified source files.
- **`suite_invariants.target_patterns`** (String array) Target classes to be analysed by the strategy, in glob-like *wildcard* format.
- **`suite_invariants.binary_folder`** (String) Folder to store the compiled proxified classes.
- **`suite_invariants.daikon_args`** (String) Arguments to be passed to Chicory/Daikon.
- **`suite_invariants.daikon_output_filename`** (String) Name of the output Splitter info file.

# D
# Evaluation Details

## D.1 Environment

All the evaluation in chapter 5 was done on the following environment:

| | |
|---|---|
| Machine Type | MacBook Pro |
| Operating System | OS X Yosemite 10.10.3 |
| Processors | Intel Core i5 2.5 GHz |
| Memory | 8 GB 1600 MHz DDR3 |
| Java Runtime version | OpenJDK 1.7.0-internal |
| Daikon version | 5.2.2 (April 2015) |
| Simplify version | 1.5.4 |

## D.2 Configuration Values for Yacon and Daikon

Aside from the ones necessary to control Yacon and Daikon to execute the right programs, the following configurations were used during evaluation.

### Yacon configuration

- **`boundary_values.second_order_trigger`** = 8
- **`boundary_values.floating_point_tolerance.equality`** = 0.0001
- **`boundary_values.floating_point_tolerance.adjacent`** = 0.01

## Daikon command-line options

- `--no_text_output` – to suppress printing invariants output to the screen.
- `--suppress_redundant` – to use the automated solver *Simplify* to suppress redundant invariants. This is use in conjunction with Java property `-Dsimplify.path=wine` <path to Simplify executable>. [1]

## Daikon configuration

- **`daikon.split.PptSplitter.disable_splitting`** = false
- **`daikon.split.PptSplitter.dummy_invariant_level`** = 2

[1]There is no Simplify executable for the architecture of our evaluation machine (Intel MacBook). We had to use *Wine*, an emulation system that makes it possible to run Windows executables on OS X system, to run the Simplify build for Windows.

# Bibliography

[1] ISO/IEC/IEEE 29119 Software Testing Standard. `http://www.softwaretestingstandard.org/`, September 2014. [Online; accessed 13-February-2015].

[2] *The Daikon Invariant Detector Developer Manual*, 5.2.0 edition, March 2015.

[3] *The Daikon Invariant Detector User Manual*, 5.2.0 edition, March 2015.

[4] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[5] Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and software verification: model-checking techniques and tools.* Springer Publishing Company, Incorporated, 2010.

[6] Tim Berners-Lee, Larry Masinter, Mark McCahill, et al. Uniform resource locators (URL). 1994.

[7] Cédric Beust and Hani Suleiman. *Next Generation Java Testing: TestNG and Advanced Concepts.* Pearson Education, 2007.

[8] Marat Boshernitsan, Roongko Doong, and Alberto Savoia. From daikon to agitator: lessons and challenges in building a commercial tool for developer testing. In *Proceedings of the 2006 international symposium on Software testing and analysis*, pages 169–180. ACM, 2006.

[9] Jonathan P Bowen, Kirill Bogdanov, John A Clark, Mark Harman, Robert M Hierons, and Paul Krause. Fortest: Formal methods and testing. In *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, pages 91–101. IEEE, 2002.

[10] Stephen Brown, Joe Timoney, Tom Lysaght, and Deshi Ye. Software testing principles and practice, 2012.

[11] Michael K. Buckland and Fredric C. Gey. The relationship between recall and precision. *JASIS*, 45(1):12–19, 1994.

[12] John N Buxton and Brian Randell. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee.* NATO Science Committee; available from Scientific Affairs Division, NATO, 1970.

[13] Capgemini, Sogeti, and HP. World quality report 2014-2015. `http://www.capgemini.com/resources/world-quality-report-2014-15`, October 2014. [Online; accessed 10-February-2015].

[14] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[15] Rick D Craig and Stefan P Jaskiel. *Systematic software testing.* Artech House, 2002.

[16] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th international conference on Software engineering*, pages 281–290. ACM, 2008.

[17] Bruce Dawson. Comparing floating point numbers, 2008. [Online; accessed 10-April-2015].

[18] Nii Dodoo. Selecting predicates for conditional invariant detection using cluster analysis. Master's thesis, Massachusetts Institute of Technology, 2002.

[19] Nii Dodoo, Lee Lin, and Michael D Ernst. Selecting, refining, and evaluating predicates for program analysis. Technical Report MIT-LCS-TR-914, Massachusetts Institute of Technology, Laboratory for Computer Science, Cambridge, MA, 2003.

[20] Michael D Ernst. *Dynamically discovering likely program invariants.* PhD thesis, University of Washington, 2000.

[21] Michael D Ernst, William G Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering program invariants involving collections. Technical Report UW-CSE-99-11-02, University of Washington, 2000.

[22] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, 2007.

[23] Hani Fouladgar, Behrouz Minaei-Bidgoli, and Hamid Parvin. On possibility of conditional invariant detection. In *Knowlege-Based and Intelligent Information and Engineering Systems*, pages 214–224. Springer, 2011.

[24] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Pearson Education, 1994.

[25] Marie-Claude Gaudel. Testing can be formal, too. *TAPSOFT'95: Theory and Practice of Software Development*, pages 82–96, 1995.

[26] Marie-Claude Gaudel. Formal methods and testing: Hypotheses, and correctness approximations. In *FM 2005: Formal Methods*, pages 2–8. Springer, 2005.

[27] Mark K Goldberg, Mykola Hayvanovych, and Malik Magdon-Ismail. Measuring similarity between sets of overlapping clusters. In *Socialcom/passat*, pages 303–308, 2010.

[28] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *Software Engineering, IEEE Transactions on*, (2):156–173, 1975.

[29] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition.* Addison-Wesley Professional, 1st edition, 2013.

[30] Dorothy Graham, Erik Van Veenendaal, Isabel Evans, and Rex Black. *Foundations of software testing: ISTQB certification.* Course Technology Cengage Learning, 2008.

[31] Peter Haberl, Andreas Spillner, Karin Vosseberg, and Mario Winter. Survey 2011:» software test in practice «. *Translation of Umfrage*, 2011.

[32] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. In *Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on*, pages 206–215. IEEE, 1988.

[33] Sudheendra Hangal, Naveen Chandra, Sridhar Narayanan, and Sandeep Chakravorty. Iodine: a tool to automatically infer dynamic invariants for hardware designs. In *Proceedings of the 42nd annual Design Automation Conference*, pages 775–778. ACM, 2005.

[34] Sudheendra Hangal and Monica S Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. ACM, 2002.

[35] Anne Mette Jonassen Hass. *Guide to advanced software testing.* Artech House, 2008.

[36] Robert M Hierons, Kirill Bogdanov, Jonathan P Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, et al. Using formal specifications to support testing. *ACM Computing Surveys (CSUR)*, 41(2):9, 2009.

[37] Cay S Horstmann. *Java Concepts: Compatible with Java 5, 6 and 7.* John Wiley &amp; Sons, 6th edition, 2009.

[38] Paul C Jorgensen. *Software testing: a craftsman's approach.* CRC press, 2nd edition, 2002.

[39] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing: A Context-Driven Approach.* John Wiley &amp; Sons, 2002.

[40] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[41] Joseph R Kiniry and Daniel M Zimmerman. Secret ninja formal methods. In *FM 2008: Formal Methods*, pages 214–228. Springer, 2008.

[42] Nancy G Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7), 1993.

[43] Y Daniel Liang. *Introduction to Java Programming - Comprehensive Version*. Pearson, 6th edition, 2007.

[44] Marina Meilă. Comparing clusterings—an information based distance. *Journal of Multivariate Analysis*, 98(5):873–895, 2007.

[45] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[46] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 3rd edition, 2011.

[47] Oracle. Setting the class path. `http://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html`, 2014. [Online; accessed 21-April-2015].

[48] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon v2: Large scale source code analysis and transformation for java. Technical Report hal-01078532, 2014.

[49] Matthew Phillips. Knight shows how to lose $440 million in 30 minutes. `http://www.bloomberg.com/bw/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minutes`, August 2012. [Online; accessed 10-February-2015].

[50] ISWB Prasetya, Jurriaan Hage, and Alexander Elyasov. Using subcases to improve log-based oracles inference. Technical Report UU-CS-2012-012, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, September 2012.

[51] Mark Ravina. Computing in japanese. 1992.

[52] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Software Metrics Symposium, 1997. Proceedings., Fourth International*, pages 64–73. IEEE, 1997.

[53] Kenneth Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 7th edition, 2011.

[54] David Shay. Disabling System.exit() (in Java). `http://jroller.com/ethdsy/entry/disabling_system_exit`, November 2006. [Online; accessed 8-April-2015].

[55] BCS SIGIST. Standard for software component testing. 3, 2001.

[56] Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in action.* Manning Publications Co., 2nd edition, 2010.

[57] Typesafe. HOCON (Human-Optimized Config Object Notation). `https://github.com/typesafehub/config/blob/master/HOCON.md`, 2015. [Online; accessed 23-April-2015].

[58] Mark Allen Weiss. *Data structures and algorithm analysis in Java.* Addison-Wesley Longman Publishing Co., Inc., 1998.

[59] Mark Allen Weiss. *Data structures and problem solving using Java.* Addison-Wesley Longman Publishing Co., Inc., 4th edition, 2009.

[60] Dan Zuras, Mike Cowlishaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.