Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Reparación Automática de Software: Ciclos Infinitos

Tesis presentada para optar al título de
Licenciado en Ciencias de la Computación

Lamelas Marcote, Sebastián Rodrigo

Director: Martin Monperrus

Codirector: Diego Garbervetsky

Buenos Aires, 2015

REPARACIÓN AUTOMÁTICA DE SOFTWARE: CICLOS INFINITOS

La reparación automática de software (ASR) es un tópico de investigación muy reciente. Consiste en la implementación de herramientas que puedan automáticamente detectar *bugs* y corregirlos. Se considera cualquier tipo de bug: un defecto en el código o una falla durante la ejecución del código. En los últimos años, han surgido enfoques novedosos y con resultados favorables para este campo.

En general, el proceso de desarrollo de software es costoso en términos de tiempo y también económicos. En particular, la fase de mantenimiento de software suele ser considerada como la más costosa. Un aspecto clave del mantenimiento es la reparación del código fuente. Cuando la cantidad de reportes de bugs supera ampliamente la cantidad de programadores, esta etapa de mantenimiento puede convertirse en un cuello de botella dañino para la evolución de un proyecto de software. Aquí reside el objetivo principal de ASR: reparar automáticamente el software para minimizar sus costos de mantenimiento.

Hay quienes creen, dentro del campo de ASR, que la mejor manera de reparar bugs es definiendo una taxonomía claramente definida sobre los distintos tipos de bugs y luego implementar métodos de reparación especializados para cada tipo. Es decir, definir una determinada "clase de defectos" y construir un programa que repare esa clase de defectos, teniendo en cuenta las propiedades específicas de esa clase.

Una clase de defectos perteneciente a cualquier lenguaje de programación es la de ciclos infinitos. Este bug es el culpable de que "se cuelgue un programa". Se trata de un ciclo que itera infinitamente, de forma no delibarada, sin devolver un resultado o lanzar una excepción. El objetivo de esta tesis es implementar un programa que intente reparar automáticamente esta clase de defectos.

En esta tesis se consideran los ciclos del tipo "`while (condition) { /* block */ }`", y la estrategia para reparar ciclos infinitos es en base a la búsqueda de una nueva condición del `while`. Se implementa un programa que intenta reparar automáticamente código Java. Para ello, se desarrollan técnicas de análisis y síntesis automática de código, con el propósito de encontrar automáticamente una condición booleana que prevenga el ciclado infinito.

Este trabajo fue principalmente desarrollado durante una pasantía de 6 meses en el año 2014 en INRIA-Lille, bajo la supevisión de Martin Monperrus, miembro del equipo de investigación INRIA SPIRALS (ver `http://www.monperrus.net/martin/`).

Palabras claves: Fallas, Reparación Automática de Código, Ciclos Infinitos, Análisis de Código, Síntesis de Código.

TOWARDS AUTOMATIC REPAIR OF INFINITE LOOPS

Automatic Software Repair is a recent Computer Science research area. It is concerned with the development of tools which automatically detect and repair *bugs*. Any kind of bug is considered: a defect in the source code or a fault in the execution of it. In recent years, innovative ideas have emerged with favorable results.

Generally, software development processes are costly both in terms of time and money. Particularly, software maintenance is often considered the most expensive one (the "legacy crisis": new software outpaces the ability to maintain it). A key task during the maintenance phase is the source code correction. Unfortunately, for most projects, the number of reported bugs is greater than the number of programmers. This way, the maintenance phase can become a severe bottleneck for the evolution of a project. This is the essential purpose of Automatic Software Repair: it aims to minimize the maintenance costs by automatically repairing software.

Within Automatic Software Repair community, there are some who believe that the most appropriate way to automatically repair bugs is drawing a clear taxonomy of common coding faults and then focusing in developing a specific repair method for each type. That is, each bug is assigned to its corresponding *defect class* and, to repair it, a specific repair method which exploits the defect class' intrinsic properties is used.

One defect class present in every programming language is the "infinite loop" defect class. In our experience, every programmer or user has, at least once, experienced this type of bug. It is one of the coding faults responsible for hanging programs. It consists of a loop which unintentionally iterates nonstop without returning an expected result or throwing an exception. In this work, we aim to automatically repair this defect class.

Particularly, we focus our attention on `while` loops. The repair endeavor includes the infinite loop detection and the synthesis of a patch for the infinite loop. We develop a technique to statically analyze source code but we also use a code synthesis technique based on logic SMT problems.

This work was mainly executed during a 6-month internship in the summer of 2014 at INRIA-Lille under the supervision of Martin Monperrus (from INRIA SPIRALS team; please refer to his website `http://www.monperrus.net/martin/`).

Keywords: Bugs, Automatic Software Repair, Infinite Loops, Code Analysis, Code Synthesis.

Gracias a la educación pública, por permitir que complete estudios secundario y unversitario.
Y gracias a todos los maestros –con y sin guardapolvo– de los cuales aprendí
siquiera pequeñas lecciones.

CONTENTS

1. Towards Automatic Repair of Infinite Loops

1.1 Introduction

Research on automatic software repair is concerned with the development of systems that automatically detect and repair bugs. We consider as bug a behavior observed during program execution that does not correspond to the expected one. Automatic software repair is close to other research areas such as automatic debugging, software testing, program synthesis and machine learning for software engineering. There have been a number of results in this field [1], [6], [2], [12], since seminal work at the end of the 2000ies [1], [6], [3].

The ultimate goal of automatic software repair is to minimize the software maintenance costs. Software maintenance is often considered the most expensive development phase; a phenomenon called the "legacy crisis" [17], where new software outpaces the ability to maintain it. A key task during maintenance is the correction of bugs (colloquially "bug fixing"). The automatic repair of even a fraction of software bugs would translate to huge savings in development time and costs.

Hamill and Goseva-Popstojanova [16] showed that one of the most common types of software faults are *coding faults*. That is, faults directly associated with the source code, independent of the requirements. For instance, incorrectly assigned values, uninitialized values, missing or incorrect data validation, incorrect loop or conditional statements, and so on. In accordance with this, within Automatic Software Repair community, there are some who believe that the most appropriate way to automatically repair bugs is drawing a clear taxonomy of common coding faults and then focusing in developing a specific repair method for each type [7]. That is, each bug is assigned to its corresponding *defect class* and, to repair it, a specific repair method which exploits the defect class' intrinsic properties is used.

One defect class present in every programming language is the "infinite loop" defect class. Every computer programmer or user has, at least once, experienced this type of bug. It is so much part of the programming folklore that Apple Inc. has renamed the street encircling its head quarters "Infinite Loop". It is one of the coding faults responsible for hanging processes. It consists of a loop which unintentionally iterates nonstop without returning an expected result or throwing an exception. In this work, we aim to automatically repair this defect class. To our knowledge, there is no published work on this topic.

1.2 Infinite Loops

The defect class we address is "infinite loop". An infinite loop is the infinite repetitive execution of the block of statements conforming the loop body. An infinite loop occurs when the execution of the loop body does not change anymore the part of the execution state that affects the truth value of the predicate used as the looping condition.

An infinite loop is critical because: 1) the program is not responsive anymore; 2) the infinite loop consumes 100.00 % of the CPU time making no progress.

We believe the infinite loop defect class is fairly common. Take, for instance, one of the historically most popular UNIX commands: grep. Anyone can search in its Git repository [28] for commits fixing infinite loops. One of them is shown in Demo 1.2.1. As indicated by the the comment introduced in the commit, the changes of the commit are made to fix an infinite loop. In order to do so, the boolean condition of the loop is corrected and a break statement is introduced.

```
- while ((match_offset = (*execute) (beg, lim - beg, &match_size, 1)) != (size_t) -1) {
+ while (lim-beg && (match_offset = (*execute) (beg, lim - beg, &match_size, 1)) != (size_t) -1) {
  char const *b = beg + match_offset;
  /* Avoid matching the empty line at the end of the buffer. */
  if (b == lim)
    break;
  + /* Avoid hanging on grep --color "" foo */
  + if (match_size == 0)
    + break;
  fwrite (beg, sizeof (char), match_offset, stdout);
```

Demonstration 1.2.1. Infinite loop patch in grep.c.

Although the defect class definition is simple to understand, the automatic repair of an infinite loop is more troublesome. To begin with, let us consider the problem of detecting an infinite loop. That is, given the source code implementation of a loop, we have to output whether it is infinite or not. Suppose this problem was decidable, and we had the implementation of the solving algorithm isInfinite. We now analyze what would happen with an invocation to the method in Demo 1.2.2, when the argument is that same loop on the rec method, recLoop. If recLoop is infinite, then the guard of the if statement evaluates to true, allowing the loop to terminate with the following break statement; then, the loop is not infinite. If recLoop is not infinite, then the guard of the if statement always evaluates to false; then, the loop could never terminate; then, the loop is infinite. Obviously, this is a contradiction.

```
void rec(Loop loop) {
    while (true) {
        if (isInfinite(loop))
            break;
    }
}
```

Demonstration 1.2.2

As a result, the problem of detecting an infinite loop is undecidable (this is the well-known Halting problem). Therefore, there cannot exist an algorithm which decides whether a loop is infinite or not. If we cannot solve the problem of detecting an infinite loop, let alone automatically repairing it. To overcome this limitation, we delve into more details about Automatic Software Repair.

A widely accepted problem definition [7] for Automatic Software Repair is the following: "given a test suite with at least one failing test, generate a patch that makes all test cases pass". Typically, an automatic repair tool implementation would require the user to provide input source code (single file or complete project) and a test suite, and it would be able to output a patch. In turn, when the patch is applied to the original source code, the execution of the test suite would be successful. The steps of a repair tool generally are:

1. Create a *patch*:
    (a) Failure detection (unexpected behavior is observed).
    (b) Bug diagnosis (reason about why it happened).
    (c) Fault localization (find the possible root cause).
    (d) Repair inference (logical representation of the patch).
    (e) Code synthesis (compilable representation of the patch).
2. Apply the *patch* to the source code and run the test suite.
3. If the execution is successful, output the found *patch*.
4. Otherwise, start again from (1) –or terminate without a solution.

In the context of test-suite based software repair, repairing an infinite loop means modifying the behavior of the infinite loop so that every test case executing the infinite loop both halts and passes. In our case, the patch we aim to synthesize consists of a new boolean expression for the loop condition of the infinite loop. In other words, for the repair to be successful, the new predicate must correct every infinite execution happening in the non-halting test cases.

Let us narrow our problem a little bit further. We begin by saying that the most popular loop constructs present in imperative languages are: `for`, `while` and `do-while`. For sake of simplicity, we'll mainly focus on `while` loops. The ideal repair tool would repair *any* infinite `while` loop. But that is a gargantuan undertaking, and it is our concern to implement a repair technique versatile enough to be practical. Then, we intend to commence solving this problem with a more sensible and reasonable objective.

Historically, as programming languages evolved, researchers spent more and more time deliberating about programming practices. We consider that, through history, conflicting viewpoints and discussions were eventually reconciled in a general consensus favoring program modularity and orthodox source code organization. In [19] Dijkstra shares his stance against the GOTO statement, for it makes the code incredibly hard to follow. In [20], Guy Steele tries to debunk the myth about the alleged performance overhead inherent to procedure calls –which conform a powerful means of expressiveness,– and argues that "spaghetti code" could become much more clear if programmers adopted the usage of procedure calls. Moreover, in [18] Presser succinctly defines structured programming languages. For instance, he defines the loop construct as the: "transfer of control so as to repeat an operation as long as some specified condition, which may be placed anywhere in the loop, is true; when the condition is no longer true control is transferred to next point following the loop in program sequence."

We therefore take a stance on how we think most `while` loops are implemented in practice. Based on our intuition, programming experience and trying to comply with generally accepted best programming practices, our stance is that in most `while` loops the predicate of the looping guard alone determines the flow of the program (i.e., loops do not have `break` or `return` statements) and that loop executions generally have low iteration records.

With this perspective in mind, we propose an approach to automatically repair infinite loops for those cases. The implementation of our repair tool is called *Infinitel*. It is based on the development of different loop manipulation techniques to: dynamically analyze the number of iterations of loops, decide to interrupt the loop execution, and dynamically examine the state of the loop on a per-iteration basis. In order to synthesize a new loop condition, we encode this set of program states as a code synthesis problem using a technique based on Satisfiability Modulo Theory (SMT).

We evaluate *Infinitel* on seven seeded-bugs and on seven real-bugs. Our technique is able to repair all of them, within seconds up to one hour on a standard laptop configuration. We deeply discuss those cases to understand the strength and weaknesses of our automatic repair technique.

1.3 Thesis Contributions

The contributions of this thesis are:

- The definition of the infinite loop defect class.
- The definition of the automatic repair problem for infinite loops.
- The introduction of a loop theory to describe loops beyond their syntactic structure.
- An empirical study on loops, based on three real-world projects, to understand how loops are used in practice.
- Two techniques to dynamically analyze the behavior of loops. One is used in the empirical loop study and the other in the implementation of our infinite loop repair method.
- A technique to statically analyze the scope of loops, which, in turn, enables a dynamic analysis of the state of the loop in each iteration.
- A solution for the automatic repair problem for infinite loops, based on the following techniques: source code manipulation, runtime loop state analaysis and code synthesis.
- The evaluation of the proposed solution with 7 seeded bugs and 7 real bugs.

1.4 Structure of the Thesis

The rest of the thesis is organized as follows. In Chapter 2 we define a loop theory which is referred to throughout this document. In Chapter 3 we present a preliminary case study of `while` loops, which brings support for the assumptions we make in our approach. In this section we develop a technique, based on source code manipulation, which lets us analyze the usage of loops in real projects.

In Chapter 4 we describe our approach to automatically repair the looping guard of *while* loops in detail. This endeavor includes a strategy fot infinite loop detection and the synthesis of a patch for the infinite loop. We develop a technique to manipulate loops, also based on source code manipulation. Specifically, the goal is to control how the loop decides to iterate or break before every iteration. This technique lets us dynamically analyze the number of iterations of the loop and decide to interrupt the loop execution, in the case of an infinite loop execution. This way, we do not hang our repair method when we run the non-halting test cases. We also develop another technique to let us statically analyze the scope of the loop. This technique enables us to dynamically examine the state of the loop on a per-iteration basis. Through the inspection of the loop state throughout different iterations, we are able to produce the patch, with a code synthesis technique based on logic SMT problems.

In Chapter 5 we show the evaluation of our approach. We evaluate our implementation, *Infinitel*, on seven seeded-bugs and on seven real-bugs. We show favorable results in both cases, being able to find a patch for every bug, with a reasonable performance.

In Chapter 6 we review the repair method and discuss its limitations. In Chapter 7, we compare our approach to other related work. In Chapter 8 we express our conclusions and we describe new ideas, triggered by the research of this thesis, for future work.

Finally, we append three explanatory chapters to help the careful reader thoroughly understand the work presented in this thesis. In Appendix A, we show how to repair an infinite loop with a full example. In Appendix B, we describe the code synthesis methodology in detail. In Appendix C, we discuss implementation details concerning the runtime compilation of modified source code.

2. LOOP THEORY

In this section we introduce loop-specific concepts which are referred to throughout the thesis, we present a general testing framework for loops and we briefly introduce two kinds of infinite loop bugs and give a hint on how to repair them.

2.1 Concepts

The syntactic definition of the loop is a code block which repeats itself as long as some conditions hold. One could also envision the loop construct as a tool made to continuously transform objects until they become "ready" to carry on with subsequent execution steps. These definitions give place to different concepts we embodied in a Loop Theory.

```
void clear(int[] array) {
    int n = array.length;
    for (int i = 0; i < n; i++) {
        array[i] = 0;
    }
}
```

(*a*)

```
Node root(Node node) {
    while (true) {
        if (node.parent == null)
            return node;
        node = node.parent;
    }
}
```

(*b*)

```
int index(int[] sorted, int e) {
    int low = 0;
    int high = sorted.length - 1;
    do {
        int mid = (low+high+1)/2;
        if (sorted[mid] <= e) {
            low = mid;
        } else {
            high = mid;
        }
    } while (sorted[low] != e);
    return low;
}
```

(*c*)

```
int method(int a) {
    int b = a;
    while (b > 0) {
        if (b == 18) {
            return a;
        }
        if (b == 9) {
            break;
        }
        b -=1;
    }
    return b;
}
```

(*d*)

Demonstration 2.1.1. (*a*): `for` loop. (*b*): *unbreakable* `while` loop. (*c*) *idempotent* `do-while` loop. (*d*) `while` loop with a *break* and *return* statements.

Loop: control flow statement which permits to repeatedly execute a block of statements until a boolean condition evaluates to false, or until an instruction to terminate the loop is executed (see break and return statements).

Loop body: the block of statements inside the loop to be repeatedly executed. For instance, in Demo 2.1.1(*b*), the loop body consists of an `if` statement followed by an assignment.

Break statement: instruction to break the loop from within the loop body (e.g., statement of second `if` in Demo 2.1.1(*d*)).

Return statement: instruction to both break the loop and exit the method containing it, from wihin the loop body (e.g., statement of first `if` in Demo 2.1.1(*d*)).

Looping guard: boolean condition used in a loop as a way to determine termination. For instance, the looping guard in Demo 2.1.1(a) is "`i < n`". The role of the looping guard is twofold and it changes at runtime. On the one hand, before executing the first iteration of the loop, the looping guard acts as an *entrance precondition*. If the precondition is not met (the first evaluation of the looping guard returns false), the flow of the program continues without entering the loop altogether. However, if the precondition is met, the first iteration begins and, from then on, the looping guard will be acting as an *exit condition*. If any subsequent evaluation of the looping guard evalutes to false, then the exit condition is met, so the flow can continue outsite of the loop. In `do-while` loops, as there is no precondition role, the looping guard always acts as an exit condition.

Completion point: a specific state of the variables in scope of the loop which determine the termination of the loop. We can think of the looping guard as a predicate $\Psi(I)$ which takes as input a set I of variables and outputs a boolean value –*true* to iterate or $false$ to exit the loop. The completion point is achieved when the predicate $\Psi(I)$ evalutes to $false$.

In the previous paragraph, our reference to the completion point does not conflict with the twofold role of the looping guard. When the looping guard acts as the exit condition, the association betwen the exit condition and the completion point is direct. When the looping guard acts as a precondition, the precondition is describing the completion point because it indicates whether the completion point has already been reached before the loop or not.

Loop execution: a loop execution starts the first time the looping guard is evaluated and ends when the flow of the program continues outisde the loop. That is, even if a loop performs zero iterations, we count the single evaluation of the looping guard to false as one loop execution.

Iteration record: number of times the looping guard is evaluated to true during the loop execution.

Infinite execution: non-halting executions of a loop. Abusing language, we refer to "infinite loop" and "infinite execution" interchangeably. E.g., in Demo 2.1.1(c), there is an infinite execution if the element `e` is not present in the input array `sorted`.

Unbreakable loop: a "`while (true)`" loop with return statements, inside a non-void method where no ulterior instruction exists after the loop. The characteristics of such a loop make it "unbreakable" because it does not support the insertion of a break statement. If we wanted to insert a break statement, we would also have to add a return statement following the loop. Otherwise, the method containing the loop would not compile, for it is a non-void method and it must have a return statement in every branch of execution. An example is shown in Demo 2.1.1(b). Note that to identify an unbreakable loop a syntactic analysis is sufficient.

Idempotent loop: a loop whose looping guard could be evaluated arbitrarily more times than needed and the result of the algorithm would not change. That is, in this type of loops, for the loop to be correct, only a lower bound on the number of iterations exists; it does not exist an upper bound. The loop in Demo 2.1.1(c) illustrates this concept. The algorithm finds the index of an element –present in a sorted array– with binary search.

The looping guard makes the algorithm have $O(\log n)$ complexity (where n is the length of the input array). However, we can make this algorithm run in $O(n)$ with the following looping guard conversion (leaving the loop body unmodified):

```
- while (sorted[low] != e)
+ for (int iterations = 0; iterations <= sorted.length; iterations += 1)
```

The result of the algorithm in both cases returns the correct index, but the second implementation would allow the loop to iterate more times than needed.

Door-door execution: a loop execution which ends because the evaluation of the looping guard returns false. The term is coined from the analogy of entering and exiting a room through a door, usually the single conventional entrance to a room. For instance, in Demo 2.1.1(a), the loop will always have door-door executions, because there is not any other kind of way it can terminate. Because the termination of a door-door execution is determined by the evaluation of the boolean condition of the looping guard, we say that door-door executions have "conditional exits".

Door-window execution: a loop execution which does not end like a door-door execution: instead of ending after the evaluation of the looping guard, it ends after executing a statement from within the loop body. Following the same analogy, the sudden exit from inside the loop body parallels an unanticipated evacuation from a room throughout a window. In this case, the termination of the loop can have three causes: the execution of a break statement, the execution of a return statement, or the raise of an uncaught exception. According to its cause, we subclass the termination type in three kinds: "break exit", "return exit" or "throw exit". For instance, in Demo 2.1.1(b), whenever the loop is executed, it can only have door-window executions with return exits.

Exit nature: the way a loop execution ends: conditional, break, return or throw exit. Leaving throw exits aside, sometimes the exit nature can be inferred statically: if there are no break or return statements (such as in Demo 2.1.1(a) and Demo 2.1.1(c)) it will always be a conditional exit; in the case of unbreakable loops (such as in Demo 2.1.1(b)), it will always be a return exit. On other loops with break and/or return statement(s) (Demo 2.1.1(d)), the exit nature can only be determined during runtime.

2.2 Testing Framework for Looping Guards

For completion of this loop theory, in this subsection we present one standard way to describe the behavior of looping guards using test cases. As said in Section 2.1, a looping guard has two roles: entrance precondition and exit condition. The idea is to have enough test cases as to evaluate to true and false in each role. It would require 3 tests to specify `for` and `while` loops, and 2 for `do-while` loops.

Let us use the loop of Demo 2.1.1(a) as an example for the explanation. Each row of Table 2.2.1 represents a test case for the method of that loop, the first column of each row being the test input for that method.

Method Input	Precondition	First Exit Condition	Last Exit Condition	Iteration Record
`[]`	false	-	-	0
`[7]`	true	false	-	1
`[7, 0, 1]`	true	true	false	3

Table 2.2.1. Tests for loop in Demo 2.1.1(a).

The first test case evaluates the looping guard to false the first time it is evaluated. That is, the test case performs a loop execution of 0 iterations. This can be achieved with an empty array as input in the example. The idea is to evaluate the precondition role of the looping guard to false.

The second test case evaluates the looping guard to false the second time it is evaluated. That is, the test case performs a loop execution of 1 iteration. Any array of size 1 would be useful in the example. The idea is to evaluate the precondition role of the looping guard to true and the exit condition role to false.

The third test evaluates the looping guard to false only after it has already been evaluated to true twice before. That is, the test case performs a loop execution of more than 1 iteration. Any array of size greater than 1 would be useful in the example. The idea is to evaluate the exit condition role to true at least once.

In the case of a `do-while` loop, 2 test cases suffice because the looping guard always acts as an exit condition. Therefore, only the test cases of 1 and more-than-1 iterations are required.

2.3 Loop and Bugs

In this section we introduce the kind of bugs related to the infinite loop problem. Based on the two aforementioned roles of a loop condition (see Section 2.1), we can find two kinds of infinite loop bugs: wrong precondition bug or wrong exit condition bug. In the first case, the bug occurs because the program does not skip the loop when it should. In the second case, the bug occurs because the loop does not terminate at the appropriate moment.

To fix a wrong precondition bug, there are two possible repairs. First, one can wrap the loop within an `if/then` statement encoding the precondition. Second, one can modify the loop condition so that the precondition becomes correct while the exit predicate is still valid.

For a wrong exit condition bug, there are four possible repairs: 1) changing the loop condition; 2) adding a window exit such as "`if(X) break`" or "`if(X) return`"; 3) changing the loop body such that the body correctly modifies the execution state that is analyzed in the loop condition; 4) a combination of the previous alternatives. The automatic repair technique we present in this thesis targets a change in the loop condition, which is able to both fix incorrect preconditions and incorrect exit predicates.

3. PRELIMINARY STUDY

In this section, we present an empirical study on the usage of loops. We focus –here, and in the rest of the thesis– on `while` loops only, leaving the other two (`for` and `do-while`) for future work.

3.1 Hypothesis

In Section 1.2 we stated our stance on how we think `while` loops are generally implemented in practice. What follows is an empirical study in order to contrast our stance with real life projects. We do not intend to achieve large scale statistical validity of the results presented. Instead, we want to test our stance in correlation with reality taking a few arbitrary real life projects. The evidence provided on this preliminary case study could allow us to imagine how a prototypical loop would be implemented, and to contrast our stance with this prototypical loop.

3.2 Study Protocol

3.2.1 Dataset

Three projects were selected for the study (Table 3.2.1). These projects were selected because they are well-known among *Java* programmers, and because they are large projects (number of files and lines of code shown in the table were calculated by *CLOC* [25]). In addition, another good reason to select these projects is that they represent three libraries with different domains, which is good for the generalizability of our results.

Project Name	Git Commit	Java Files (src/test)	LOC (src/test)
Apache Commons Lang	dc27be2	132 / 143	23 504 / 40 481
Apache Commons Collections	b5ffdaf	300 / 190	25 335 / 29 746
Apache Commons Math	32ef444	893 / 554	91 878 / 96 046

Table 3.2.1. Description of the three projects used for the preliminary study.

3.2.2 Analysis

To carry out this study we have the source code of each project (they are open source) and their corresponding test suite. The idea is to run each test suite, record information of every loop execution and analyze the recorded execution data according to different criteria. For each test case we collect the following information: number of executions of each loop and, for each execution, the iteration record and exit nature. To collect those numbers, a source code instrumentation is carried out on each loop (Subsection 3.2.3).

We perform two types of analysis: static and dynamic. Static analysis tells how many loops a project contains, how many loops with break or return statements there are, whether a given loop is unbreakable, etc. Dynamic analysis is only possible with loop executions, obtained from running the test suites. It tells the iteration records observed in practice, the exit nature of the loop in each execution, etc.

3.2.3 Project Instrumentation

Automatic Instrumentation and Recompilation

We want to run each project's test suite and collect information of every loop execution. One way to do this is by rewriting the source code (appending, for instance, procedure calls to a static method to register the desired information,) recompiling it and running the test suite on the modified code. We refer to this technique as *project instrumentation.*[1]

The methodology is sketched in Fig. 3.2.1. Suppose we originally have the source code of one loop implementation (step 1 in the figure). We proceed by modifying (and recompiling) the source code, "implanting" a monitor to the loop which will generate callbacks to our program, so we can keep track of the behavior of the loop (step 2). Then, during the test suite execution, we make sure the modified source code is loaded (step 3). This way, when that loop is executed in a thread running the test suite, the attached loop monitor can generate callbacks to another thread collecting loop information.

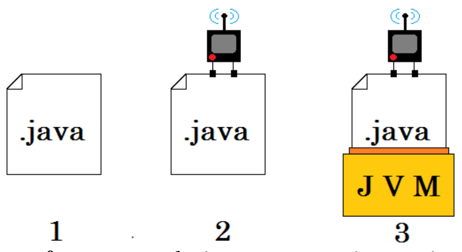

Figure 3.2.1. Diagram of source code instrumentation to implant a loop monitor.

Project Instrumentation for the Preliminary Study

We now explain the specific instrumentation used in this case. The whole process is completely automatic and this particular instrumentation completely preserves the semantics of the original source code, so the impact on the test executions is none. The instrumentation is carried out on every loop of each project before running the test suite.

To better follow the detail of the instrumentation, we show in Demo 3.2.1 the modifications performed on the loop shown in Demo 2.1.1(d) according to this instrumentation. Firstly, we fetch the loop monitor. Assuming that we assign the id 83 to this loop, we obtain the loop monitor from a global static list (line 3). Secondly, we declare a local variable (line 4), to tally the number of iterations of each loop execution, and increment its value (line 6). Finally, we write where the callbacks should be realized by the loop monitor. To acknowledge return exits, we insert a callback before the return statement (line 9). Similarly, we insert a callback before the break statement for break exits (line 13). To acknowledge throw exits, we wrap the original loop body in a `try-catch` (lines 7 and 17-20). In the case an exception is thrown, we catch it in the appended `catch` block (line 17), we write the callback (line 18), and then we throw the exception (line 19), to preserve the semantics of the loop. In each of these three callbacks, the loop monitor can both determine the iteration record of the loop execution and the exit nature. To handle conditional exits, we simply introduce a callback to the loop monitor after the loop body (line 22). In turn, the loop monitor discerns a conditional exit from a break exit because, in the latter case, this callback would be preceded by a previous callback handling the break exit.

[1] We use *Spoon* [5] to implement source code instrumentation.

```
int method(int a) {
  int b = a;
+ LoopMonitor LM_83 = Global.getMonitor(83);
+ int ITERS_83 = 0;
  while (b > 0) {
+   ITERS_83 ++;
+   try {
      if (b == 18) {
+       LM_83.returnExit(ITERS_83);
        return a;
      }
      if (b == 9) {
+       LM_83.breakExit(ITERS_83);
        break;
      }
      b -=1;
+   } catch (Throwable THW_83) {
+     LM_83.throwExit(ITERS_83);
+     throw THW_83;
+   }
  }
+ LM_83.end(ITERS_83);
  return b;
}
```

Demonstration 3.2.1. Instrumentation on loop from Demo 2.1.1(d)

3.3 Empirical Analysis

Here, we present the empirical analysis on the information collected from the execution of the test suite for the selected projects: *Lang*, *Collections* and *Math*. We divide the analysis in different sections.[2]

3.3.1 General Outline

In table Table 3.3.1 we summarize useful properties for the three projects to be analyzed: *Lang*, *Collections* and *Math*. These are values which will be referred to further in the following sections.

The first part of the table shows properties obtained by statically analyzing the source code of each project, whereas the second part of the table –as well as the following sections– shows results derived from dynamic analysis (information gathered from test executions.)

Property	Lang	Collection	Math
Loops	90	155	278
Loops with `break` statement	11	8	23
Loops with `return` statement	5	54	62
Loops with `break` and `return`	0	1	3
Unbreakable loops	0	4	40
Tests	2 556	14 792	6 077
Tests executing at least one loop	525	3 850	2 620
Tests executing at least one loop (%)	20.54 %	26.03 %	43.11 %

Table 3.3.1. Properties of the projects to analyze.

[2] Boxplots shown below have whiskers for 10% and 90% percentiles. The bottom and top of each box are always the first and third quartiles (Q1 and Q3), and the band inside each box is always the second quartile (the median.)

3.3.2 Tests per Loop

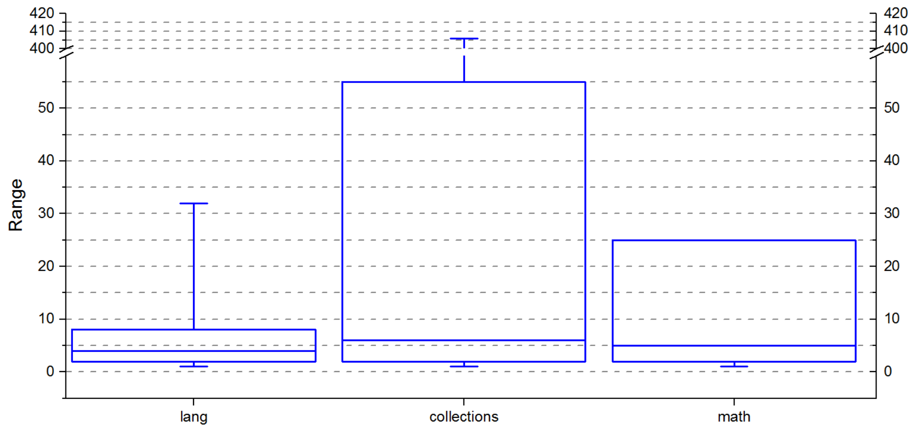

Figure 3.3.1. Tests per loop across projects.

Concern: the first thing we would like to know is whether the test suite is a good resource to analyze loops. Considering what was said in Section 2.2, we would need at least 3 tests per loop to specify every looping guard.

Results: from figures exhibited above, a direct calculation would give us that *Lang* has 5.83 (525/90) tests per loop, *Collections* has 24.84 (3 850/155) and *Math*, 9.42 (2 620/278). However, let us be more analytical and provide more evidence about the number of tests per loop.

In Fig. 3.3.1 the plot shows, for each project, the boxplot of the number of tests per loop. The median for *Lang* is 4, which means that 50.00 % of loops have at least 4 tests. The median for *Collections* is 6, and for *Math* is 5. In *Lang* 61.11 % (55/90) of loops have at least 3 tests, in *Collections* 67.10 % (104/155) and in *Math* 66.91 % (186/278).

Insight: in the worst case (*Lang*), 61.11 % of loops have at least 3 tests. Being 3 the bare minimum, this suggests that we can indeed exploit the test suite to study loops.

3.3.3 Loops per Test

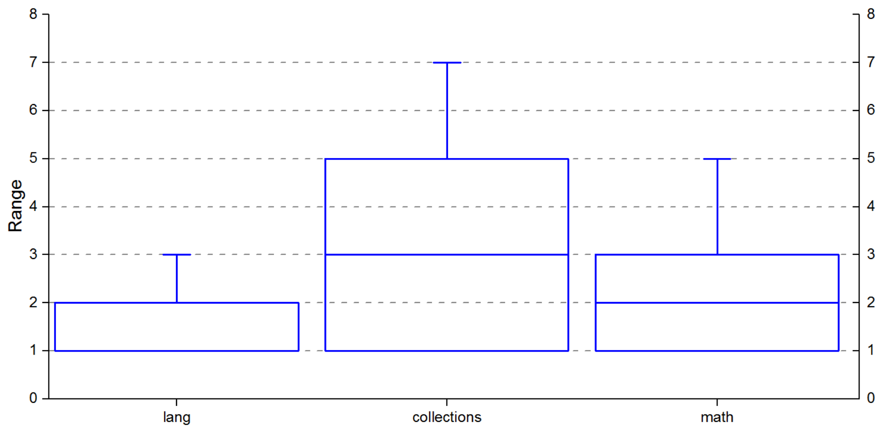

Figure 3.3.2. Loops per test across projects.

Concern: if we are trying to fix a failing test of a project, but we do not know which is the failing loop, we would like to know how many different loops each test executes to predict how many loops should be analyzed during the fault localization phase.

Results: in Fig. 3.3.2 the plot shows, for each project, the boxplot of the number of different loops executed per test. The highest Q3 value is 5 for *Collections*. This means that 75.00 % of tests execute 5 or less loops. The Q3 is 3 for *Math*, and 2 for *Lang*.

Insight: when repairing a failing test, in the worst case, we would have to analyze at most 5 loops to find the failing loop.

3.3.4 Loop Executions per Test

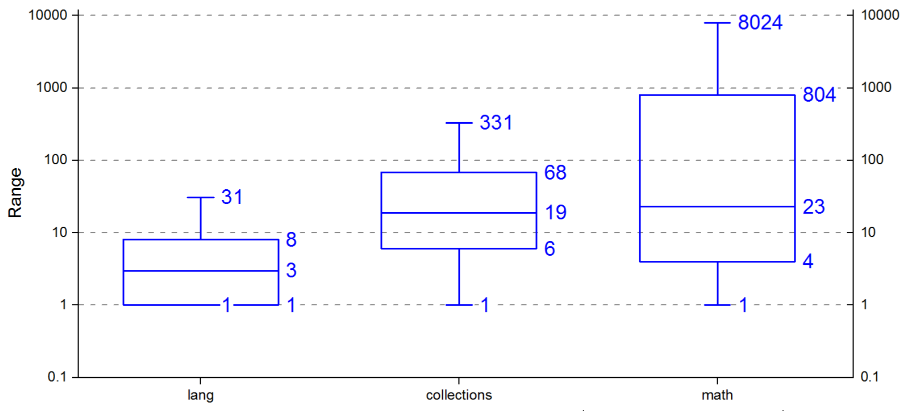

Figure 3.3.3. Loop executions per test (logarithmic scale).

Concern: we have analyzed the relation between number of tests and number of loops. We can further study this relation with a finer-grained study by counting the number of loop executions in each test.

Results: in Fig. 3.3.3 the plot shows the boxplot of the number of loop executions in the test cases of each project (with logarithmic scale in vertical axis.) The median for *Lang* is 3, which means that 50.00 % of the test cases in the project have at most 3 loop executions. The median for *Collections* is 19 and for *Math* is 23. However, there is a high standard deviation and the Q3 values are 8 for *Lang*, 68 for *Collections* and 804 for *Math*.

It should be said that counting the number of loop executions involves dealing with rather large figures. The total number of loop executions is 3.08×10^6 in *Lang*, it is 5.62×10^6 in *Collections* and it is 1.37×10^9 in *Math*.

Insight: the number of loop executions in each test largely varies among and within projects. A reason that could possibly explain this is that loop constructs are more critical in *Math* (it is both the project with most loops and highest proportion of tests executing loops –with 278 and 43.11 %, respectively), then in *Collections* (those magnitudes are 155 and 26.03 %) and finally in *Lang* (90 and 20.54 %).

3.3.5 Exit Nature

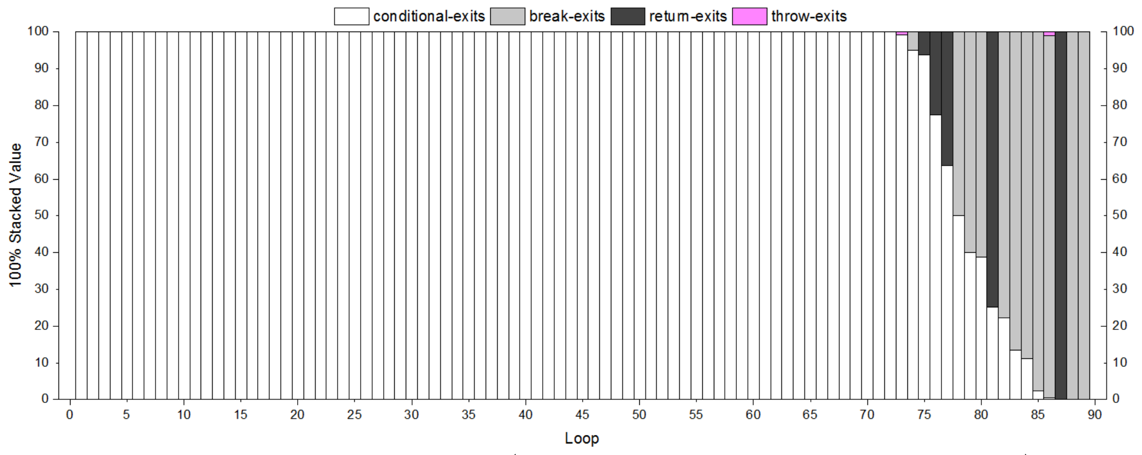

Figure 3.3.4. Exit nature in Lang (ordered by decreasing conditional exit ratio).

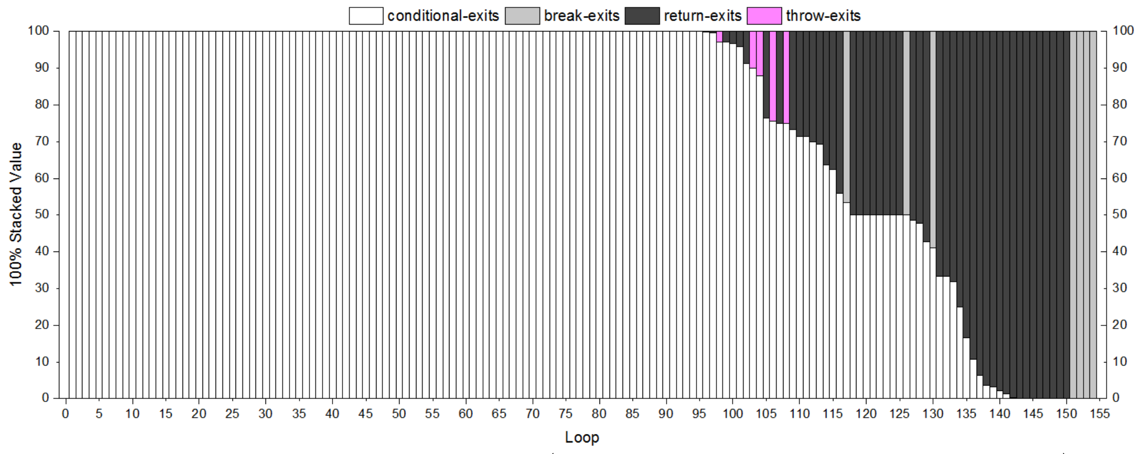

Figure 3.3.5. Exit nature in Collections (ordered by decreasing conditional exit ratio).

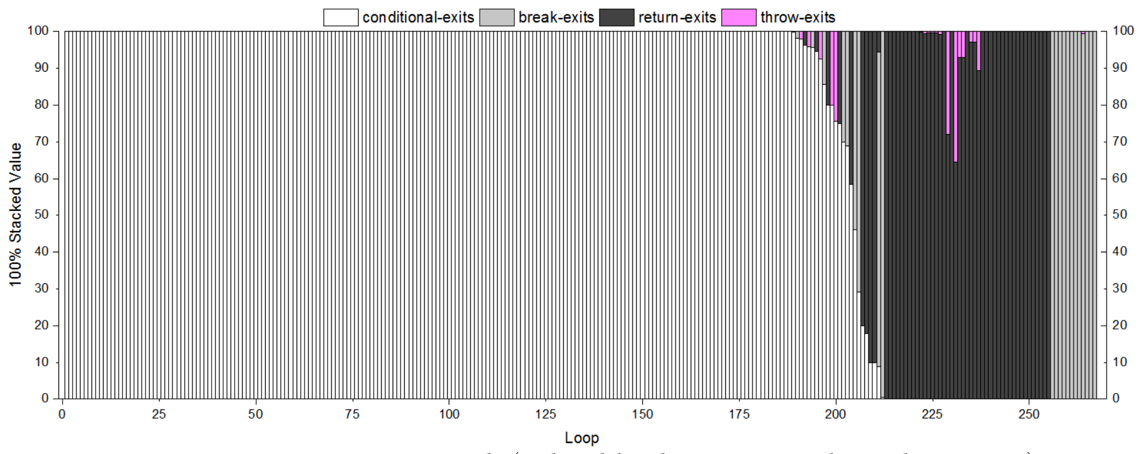

Figure 3.3.6. Exit nature in Math (ordered by decreasing conditional exit ratio).

Concern: in Section 1.2 we said that we expect most loops to have door-door executions and conditional exits. Then, we need to analyze the exit nature of the loops in real projects.

Results: we first analyze the loops statically. In *Lang* 74/90 loops (82.22 %) do not have break or return statements. In *Collections*, 94/155 (60.65 %). In *Math*, the number is 196/278 (70.50 %). On the other hand, *Lang* has 0 unbreakable loops, *Collections* has 4/155 (2.58 %) and *Math*, 40/278 (14.39 %).

We now analyze the exit nature dynamically. In Fig. 3.3.4, Fig. 3.3.5 and Fig. 3.3.6 the plots show, for each loop in the project, the proportion of each observed exit nature (i.e., out of 100 loop executions, how many executions exited through a conditional, break, return or throw exit).

It is interesting to look what happens with loops which have break or return statements. As we can see in the figures, there are loops which exhibit both door-door and door-window executions. Excluding unbreakable loops (because they could never have door-door executions), this happens in 13/16 (81.25 %) in *Lang*, 51/57 (89.47 %) in *Collections* and 20/42 (47.62 %) in *Math*.

Finally, we can add that 99.77 % of all executions in *Lang* are door-door executions. That value is 85.21 % for *Collections*. It is 85.35 % for *Math*.

Insight: the static analysis exposes an absence of break and return statements in the majority of the loops, so most loops are constructed only with door-door executions in mind. The dynamic analysis confirms this bias for door-door executions, showing that, in all projects, most loop executions end with conditional exits. Even when the loop has door-window executions, most loops with door-window executions have a mixed exit nature.

3.3.6 Iteration Record

Concern: if we want a fix for a door-door execution, we would have to correct the looping guard of the loop. To verify that the fixing looping guard is correct, we would have to evaluate it throughout all iterations of each passing test. The complexity of this verification phase would depend on the iteration record of each execution, so we would like to know what to expect in this regard for the prototypical loop.

Results: in Fig. 3.3.7, Fig. 3.3.8 and Fig. 3.3.9 the plots show, for each exit nature, the boxplot of the iteration record of each loop execution. For conditional exits, the median is 0 in all projects. This means that 50.00 % of loop executions ending with a conditional exit perform 0 loop iterations in all projects.

Regarding other exit natures, the median for break exits is 2 for *Lang*, 4 for *Collections* and 1 for *Math*. The median for return exits is 2 for *Lang*, 1 for *Collections* and 1 for *Math*. Finally, the median values for throw exits are 1 in *Lang* and *Collections* and 26 in *Math*.

Insight: the door-door execution is the fastest one to finish. Return exits are slightly faster than break exits. In any case, the expected iteration record for the prototypical loop is fairly low.

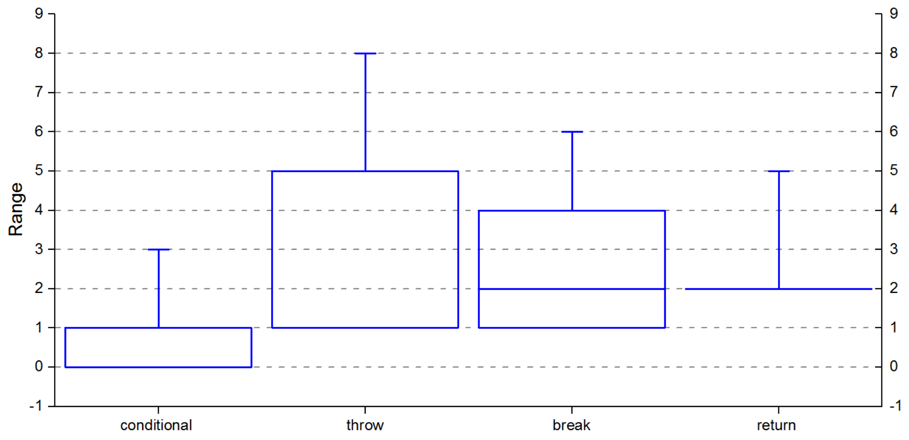

Figure 3.3.7. Exit nature record comparison in Lang.

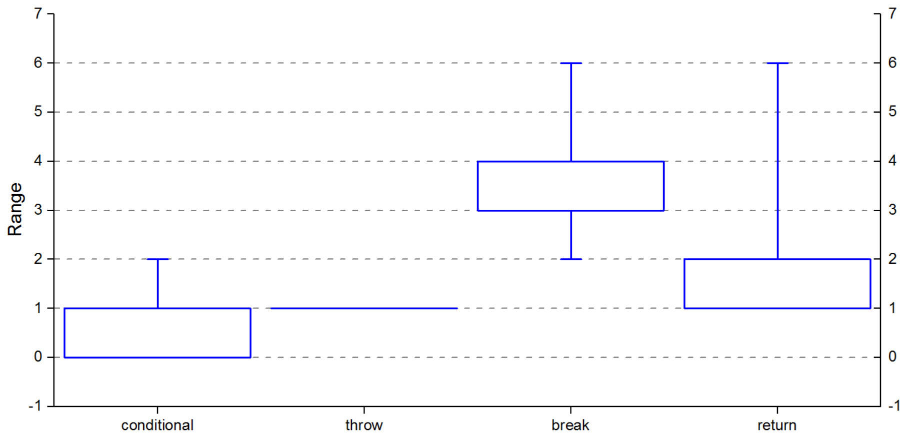

Figure 3.3.8. Exit nature record comparison in Collections.

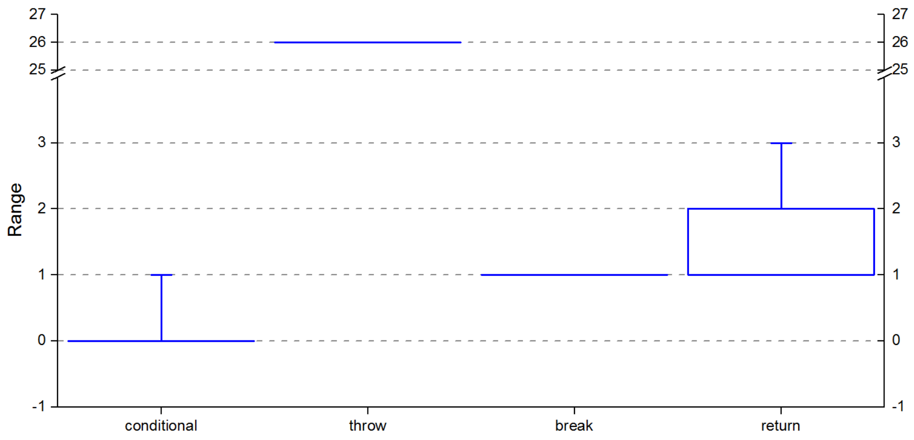

Figure 3.3.9. Exit nature record comparison in Math.

3.3.7 Loop by Loop

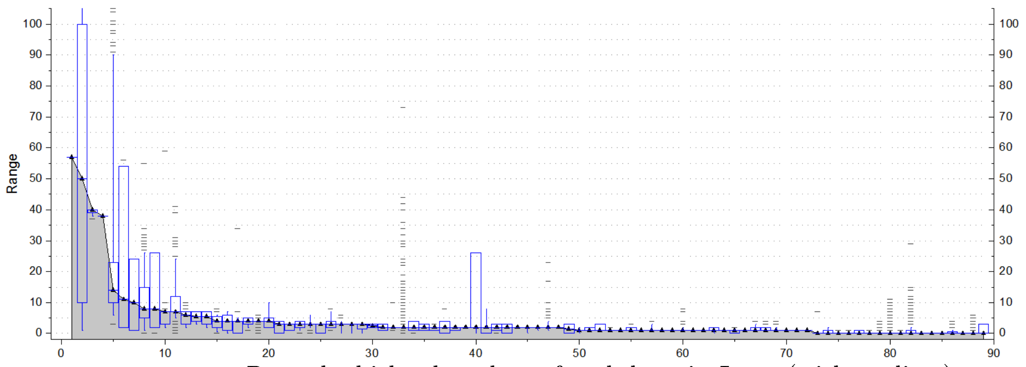

Figure 3.3.10. Record whisker boxplots of each loop in Lang (with outliers).

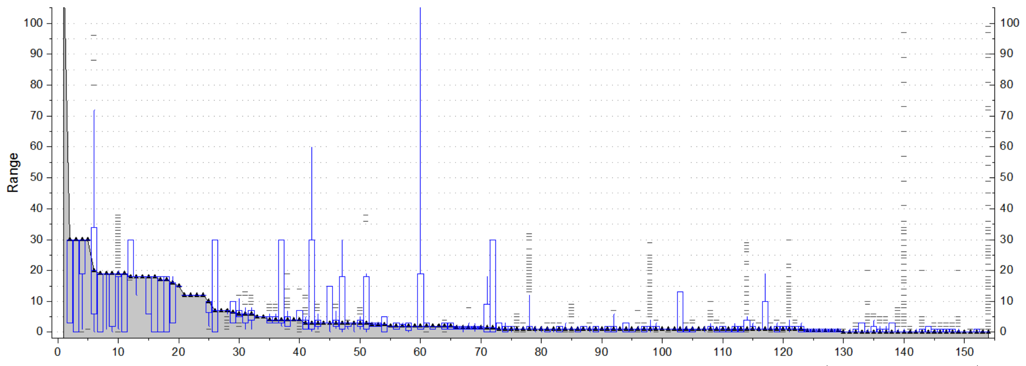

Figure 3.3.11. Record whisker boxplots of each loop in Collections (with outliers).

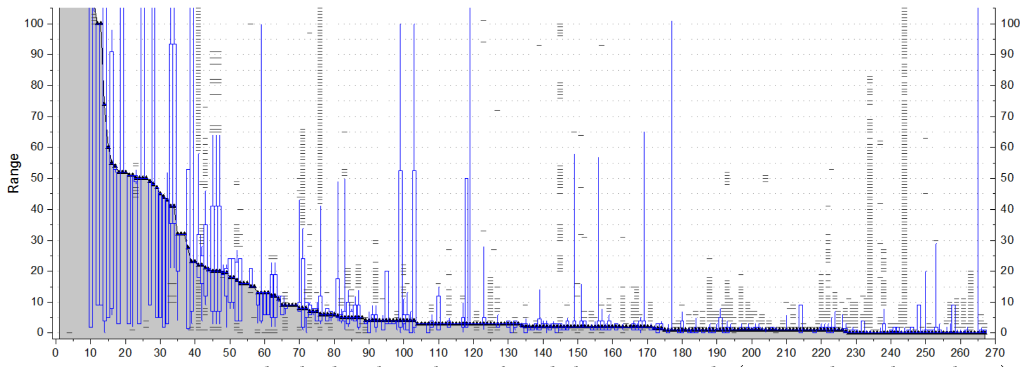

Figure 3.3.12. Record whisker boxplots of each loop in Math (cropped, with outliers).

Concern: our previous analysis only takes into consideration aggregated figures, so further analysis is desirable to verify that the expected iteration record of a loop execution is fairly low.

Results: in Fig. 3.3.10, Fig. 3.3.11 and Fig. 3.3.12, the plots show the boxplots of iteration records individually for each loop (with outliers). A shadowed area is shown from the median value of each boxplot to the horizontal axis. Loops are ordered descendingly by the value of the median record.

First, we can see that in all projects there is a long-tailed distribution of median values, where higher median values are present only in the minority of the loops (the left side of each figure). Second, we can see that the IQR (interquartile range) of the boxplots is generally low in the tail of the diagram, where most loops reside. This means that most iteration records of a loop are close to the median. These two remarks suggest that most loops have a similar median value and a narrow IQR.

To illustrate the previous remark with numbers, we can say that in *Lang* nearly 92.00 % of the loops have a median iteration record equal or lower than 8; nearly 93.00 % of loops in *Collections* have a median iteration record equal or lower than 18; and, nearly 90.00 % of loops in *Math* have a median iteration record equal or lower than 49.

In the left side of Fig. 3.3.11 and Fig. 3.3.12 we can spot boxplots where the median overlaps the Q3 value. This means that the median is close to the top record of each loop. Then, although the median is higher than in most other loops, the worst iteration record would not go much higher than the median.

Finally, if we compare the plots of the projects, we see that there are more outliers in *Math*, and then in *Collections*. This is because in those projects the number of loop executions is higher, so it is more likely to find outliers.

Insight: as a long-tail distribution suggests, most loops throughout a project exhibit similar iteration records. In the worst case (*Math*) nearly 90.00 % of loops have a median iteration record equal or lower than 49. This means that most loop executions have an iteration record lower than 50.

3.3.8 Iteration Records of 0 or 1

Concern: the previous analysis suggest that that the number of iterations in each loop execution is fairly low. So far, we are overlooking border cases. We now analyze one extreme case by observing iteration records of 0 or 1. We refer to "n-executions" to loop executions with an iteration record of exactly n iterations. Then, here we analyze 0- and 1-executions.

Results: in Fig. 3.3.13, Fig. 3.3.14, Fig. 3.3.15 the plots show, for each loop in the project, the proportion of 0-executions, 1-executions and n-executions for $n > 1$. We can see that some loops have the ideal loop specification explained in Section 2.2. Namely, 20/90 (22.22 %) in *Lang*, 66/155 (42.58 %) in *Collections* and 68/278 (24.46 %) in *Math*. We can also see, in all projects, as said in Section 2.1, that the looping guard is effectively taking the role of a precondition in many executions.

In *Lang*, 59.97 % of all executions are 0-executions, in *Collections* 49.12 % and 72.21 % in *Math*. On the other hand, 17.61 % of all executions are 1-executions in *Lang*, 32.39 % in *Collections* and 22.46 % in *Math*. This means that 77.58 % of all executions have less than 2 iterations in *Lang*, 81.51 % in *Collections* and 94.67 % in *Math*.

Insight: around 50.00 % of the overall loop executions of a project are 0-executions, and around 80.00 % of the overall loop executions of a project perform less than 2 iterations. This supports the fact that the prototypical loops would have low iteration records.

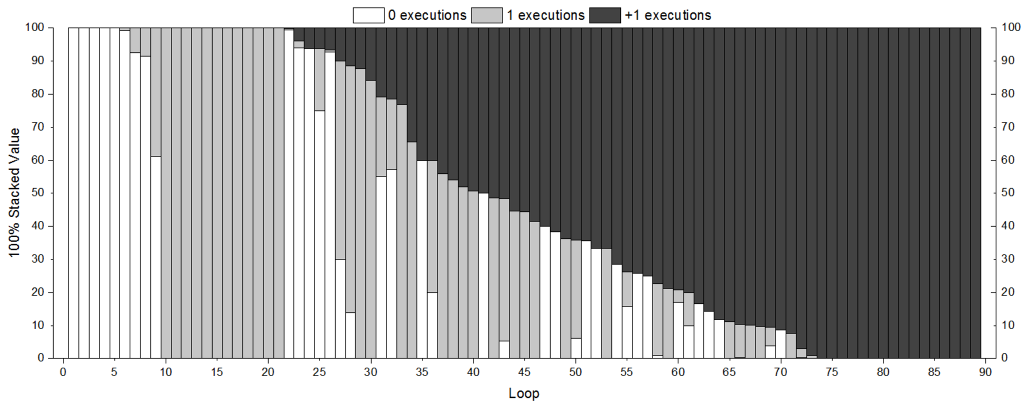

Figure 3.3.13. 0-1 Executions in Lang.

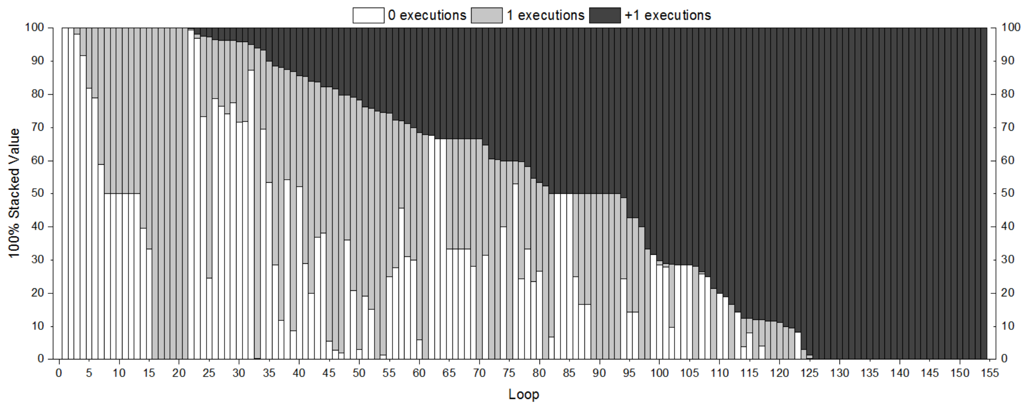

Figure 3.3.14. 0-1 Executions in Collections.

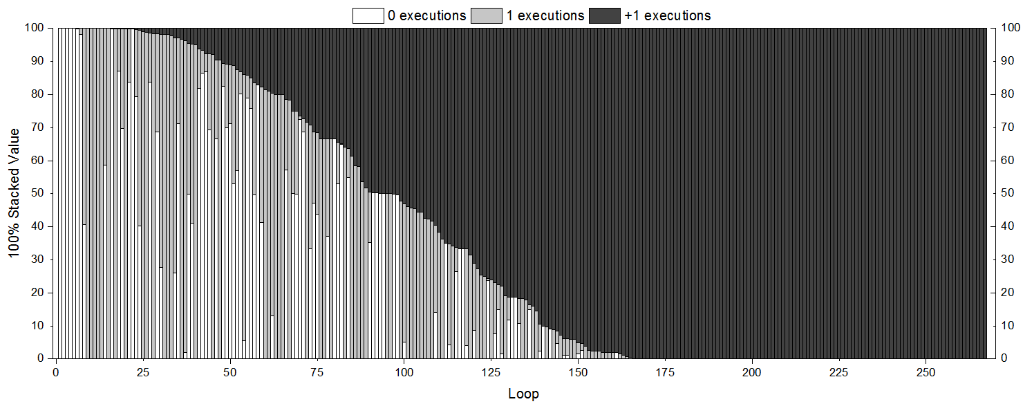

Figure 3.3.15. 0-1 Executions in Math.

3.3.9 Top record

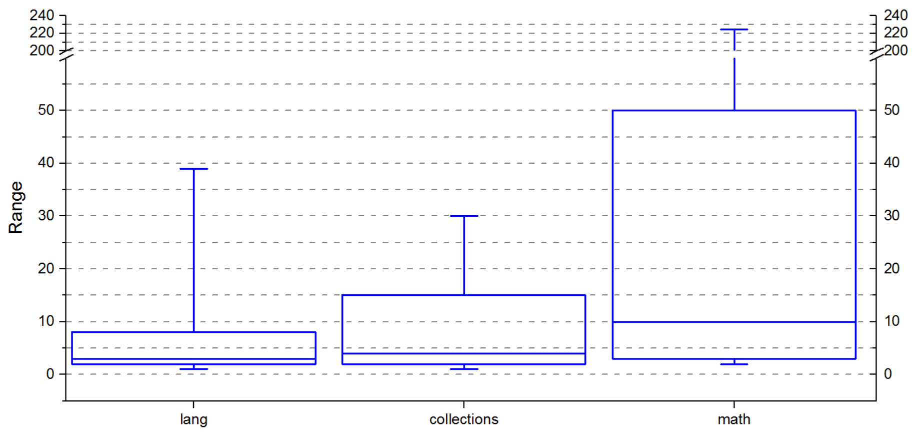

Figure 3.3.16. Top records across projects.

Concern: we now shift our attention to another extreme case: what happens when we look at the top record (maximum iteration record observed throughout all executions) of each loop.

Results: in Fig. 3.3.16 the plot shows, for each project, the boxplot of the top record for each loop. The highest top records observed for each project are 5 279 in *Lang*, 10 000 in *Collections* and 5 000 000 in *Math*. These values are all outliers. The median top record is 3 for *Lang*, 4 for *Collections* and 10 for *Math*; and the Q3 values are 8, 15 and 50, respectively.

Taking 50 as reference, in Lang 83/90 (92.22 %) of loops have a lower top record, 148/155 (95.48 %) in *Collections* and 207/278 (74.46 %) in *Math*.

Insight: considering the worst case only (*Math*), around 75.00 % of loops have a top record lower than 50. This supports the fact that most loop executions have an iteration record lower than 50.

3.3.10 Idempotent Loops

Concern: we introduced in Section 2.1, the concept of idempotent loop. This kind of loop is interesting to analyze because it only has a lower bound for iteration records. This feature could be exploited, when fixing a failing loop, to generate a new looping guard. For this reason, we also analyze the amount of idempotent loops in the selected projects.

Results: finding idempotent loops requires an individual analysis of each loop, so, for practicality, we look for a weaker characterization of loops: loops that behave like idempotent loops.

To find these loops, we do the following. We manipulate each loop and force them to perform 1 000 extra iterations every time they are executed. That is, when the looping guard evaluates to false for the first time, we override the looping guard, and perform another 1 000 iterations. If, with these changes, all tests using these loops still pass, and if, in every extra iteration, the looping guard keeps evaluating to false, then the altered loop behaves like an idempotent loop. For simplicity, loops with break or return statements are ignored.

Altering each loop with the above methodology, and rerunning all its tests, we obtain that 3/90 loops in *Lang* behave like idempotent loops, 5/155 in *Collections* and 18/278 in *Math*.

Insight: most loops are not idempotent.

3.4 Revisiting the Hypothesis

Based on the discussions above, in this particular preliminary study a prototypical loop has the following characteristics:

- It is not idempotent.
- There are at least 3 test cases executing the loop.
- It only has door-door executions with conditional exits.
- The number of iterations in each execution is generally lower than 50.

Notwithstanding the exact number of tests and iterations, nor the universal validity of these claims, these results are consistent with the stance we mentioned in Section 1.2. We can now fully introduce our repair technique based on this stance.

4. PROPOSED APPROACH

In this section we present our approach to fixing infinite loops, called *Infinitel*. We focus on `while` loops where the bug lies in the loop condition. According to the lexicon presented in Chapter 2, our approach repairs wrong exit condition bugs of door-door loop executions. Our technique is based on test cases, the infinite door-door executions to be fixed are those manifested while running the test suite.

In this context, "repairing" the infinite loop means finding a looping guard for the infinite loop such that each test case using that loop both halts and passes all the assertions. We first introduce the overview of our repair approach, and then we proceed by describing each step individually.

4.1 Overview

Algorithm 4.1.1. Top level algorithm to repair an infinite loop.

```
1: procedure infiniteLoopRepair($src$, $tests$)
2:     $src2 \leftarrow$ instrumentLoops($src$)
3:     $loop \leftarrow$ detectInfiniteLoop($src2, tests$)
4:     $thresholds \leftarrow$ findThresholds($loop, src2, tests$)
5:     $patch \leftarrow$ findPatch($loop, src2, tests, thresholds$)
6:     return $patch$
7: end procedure
```

In Algorithm 4.1.1 we illustrate the top level algorithm of our repair method. The input for our algorithm is the source code containing an infinite loop (parameter *src*) and the test suite of the source code (parameter *tests*). The test suite should be composed of passing tests and at least one hanging test, the one that triggers the infinite loop.

The first step is to instrument the source code of the input project *src*. The instrumentation enables us to remotely control loop executions (for instance, to stop tests from hanging). Once the instrumentation is performed, the second step is to find an infinite loop during the execution of the test suite. Actually, because infinite loop detection is not decidable, we implement a straightforward strategy to find the infinite loop. We do this by running the test suite and detecting hanging tests.

In the third step, the goal is to find the exact number of iterations needed by the detected infinite loop to pass the assertions of each hanging test. We call this number a "threshold" for that loop in that test. When breaking the infinite loop right after it has performed a number of loop iterations equal to the threshold, the test case passes.

In the last step, a new looping guard is synthesised, this is the final patch. The detailed explanation of each step is given in the following sections (Subsections 4.2, 4.3, 4.4 and 4.5).

4.2 Project Instrumentation

```
 1 int method(int a) {
 2   int b = a;
 3 + LoopMonitor LM_83 = Global.getMonitor(83);
 4 + int ITERS_83 = 0;
 5 - while (b > 0) {
 6 + while (true) {
 7 +   boolean stay = LM_83.decide(b > 0, ITERS_83);
 8 +   LM_83.collect(stay, b, a, ...);
 9 +   if (stay) {
10 +     ITERS_83 ++;
11       if (b == 18) {
12         return a;
13       }
14       if (b == 9) {
15         break;
16       }
17       b -=1;
18 +   } else break;
19   }
20   return b;
21 }
```

Demonstration 4.2.1. Illustration of our loop instrumentation on Demo 2.1.1(d). The code prefixed by "+", in green, is automatically injected with source code transformation.

We show here how to modify the implementation of a loop in order to control its executions remotely. With this instrumentation, we can implant a loop monitor (a technique also shown in Subsection 3.2.3). By doing so, we implant a hook between the thread running test cases and the main thread of our repair program. The loop monitor can perform callbacks to the main thread, so we can control loop executions in the thread running test cases from the thread of our repair program. Specifically, we want to control the looping guard and preserve the behavior of the loop body intact. The loop monitor is a reification of the looping guard: it decides to iterate or break the loop in each iteration.

We illustrate the source code instrumentation in Demo 4.2.1. Firstly, we fetch the loop monitor who controls the looping guard of the loop (line 3). Secondly, a local variable is created to store the iteration record of each loop execution (line 4). Then, we disassemble the original loop. To do this, we wrap the original loop body (lines 11-17) within an if statement (lines 9-18). The original looping guard is deleted (line 5), and we postpone the decision to iterate for later, so we force the flow of the program to enter the loop with a trivial looping condition (line 6). The decision to proceed with a new iteration or break the loop is delegated to the loop monitor (continuing with the door analogy, the loop monitor is the new doorkeeper). How the loop monitor takes a decision is explained in Section 4.3. The decision of whether to stay in the loop or break it is stored in another local variable (line 7). Then, according to this decision, either a new iteration is carried out (`then` branch of the new wrapping `if`) or the loop breaks (`else` branch). In the former case, the local variable is incremented (line 10). Finally, we add one more statement on this instrumentation to have the loop monitor collect information of each iteration (line 8). The meaning of this statement is explained in Section 4.5.

4.3 Infinite Loop Detection

Our strategy to detect infinite loops is straightforward. We keep track of the number of iterations throughout a loop execution and, if a maximum number of iterations is exceeded, we assume it is an infinite loop. We implement this detection strategy with the non-trivial instrumentation explained in Section 4.2.

During the loop execution, the loop monitor is responsible for deciding whether to iterate or break before starting a new iteration. To do this, it receives the evaluation of the original looping guard and the number of already completed iterations. If this number exceeds a maximum number of iterations, the loop monitor labels the loop as “infinite” and breaks it. Otherwise, the loop monitor simply returns the evaluation of the original looping guard. That is, this decision mechanism resembles that of a conventional looping guard, with the addition of a restriction on the iteration record. This threshold is fully parameterizable, and we use a reasonable value of 1 million.

The infinite loop detection is detailed in Algorithm 4.3.1. At this stage, the parameter *src* is the instrumented source code and the parameter *tests* is the test suite. We simply run the whole test suite on *src*. Every loop execution is monitored by a loop monitor. In the event of an infinite execution of a hanging test, the loop monitor will detect the infinite execution and it will break the loop when the threshold is exceeded. Also, because this infinite loop is detected during the infinite execution, the loop monitor stores the execution rank of the infinite execution (for instance, “the fourth loop execution of hanging test `testABC`”). The output of this algorithm is a specific data structure that contains the list of hanging tests, the infinite loop where each one hangs, and the execution rank of the infinite execution in each case.

Algorithm 4.3.1. Detecting infinite loops after instrumentation.

```
procedure detectInfiniteLoops(src, tests)
    hangingTests ← {}
    monitors ← implantedMonitors()
    setLimitInAll(monitors, 1 000 000)
    for test ∈ tests do
        run(src, test)
        for monitor ∈ monitors do
            if monitor.hasInfiniteExecution() then
                loop ← monitor.getLoop()
                rank ← monitor.getInfiniteExecutionRank()
                hangingTests.put(test, loop, rank)
            end if
            monitor.reset()
        end for
    end for
    return hangingTests
end procedure
```

4.4 Finding Thresholds in Hanging Tests

A hanging test, when executed, gets trapped in an infinite loop execution because the looping guard never evaluates to false. That is, the looping guard does not break the loop when it should. To rectify this, we have to amend the looping guard so that it breaks the loop during the infinite execution at the *appropriate* moment. Hence, we first have to determine the appropriate moment to break the loop in each infinite execution.

We estimate the appropriate moment to break the loop in an infinite execution by controlling the iteration record of the infinite loop in that execution. As seen in Section 4.3, we can use the loop monitor to break any loop by simply setting a maximum value of permitted iterations. If we set the maximum value equal to χ during the infinite execution of an infinite loop, and we observe that the hanging test both halts and passes, then we have found this appropriate moment, it is when χ iterations have been executed. We refer to the target χ value as an "angelic record". We use this terminology based on the literature terminology [13], [11].

Our method to find the angelic record of a hanging test is the following: we explore values from 0 to the predefined threshold in order, run the hanging test each time and assess whether it passes. If it does, the probed value is the angelic record χ. This strategy is attractive for us because of its simplicity, but also because across our experiments with real projects we have often found that the number of loop iterations is likely a low value (less than 50).

It is necessary to clarify that we look for the angelic record only during the infinite execution. For instance, if test `testABC` hangs on the fourth execution of the infinite loop, we look for an angelic record only in the fourth execution, assuming that the loop will not hang in any subsequent execution.

The angelic record search is detailed in Algorithm 4.4.1. We receive a detected infinite loop (parameter *loop*), the instrumented source code (parameter *src*) and the test suite of the project (parameter *test*). From the previous step (Section 4.3), we already know the hanging tests of an infinite loop. Then, for each hanging test, we probe different values until we find the angelic record. We do this for all hanging tests. We store this information in an associative array where the key is an infinite loop under repair.

Algorithm 4.4.1. Detail of threshold search phase.

```
1: procedure findThresholds(loop, src, tests)
2:     thresholds ← {}
3:     monitor ← loop.getMonitor()
4:     hangingTests ← hangingTestsOf(loop)
5:     for test ∈ hangingTests do
6:         for (i = 0; i ≤ 1 000 000; i++) do
7:             monitor.setLimit(i)
8:             result ← run(src, test)
9:             if result.isSuccesful() then
10:                thresholds.put(test, i)
11:                break
12:            end if
13:        end for
14:    end for
15:    return thresholds
16: end procedure
```

4.5 Patch Synthesis

To synthesize a new looping guard, we use a program synthesis technique. The idea is to synthesize a new looping guard that would make all tests pass. In particular, we use a similar approach as [4], [14] and [11]. The idea is to formulate the synthesis problem as a logical SMT problem. To do this, we encode runtime information into a first-order logic formula. Then, we use the logical solver *Z*3 [22] to find a solution. If a solution for the problem is found, we translate the solution into a boolean expression –the new looping guard.

To describe this phase of our repair method, we briefly introduce the general code synthesis based on SMT problems (we present the full explanation in Appendix B) and, then, we explain how it is used particularly in our repair method.

4.5.1 Synthesis as an SMT Problem

Introduction

The goal of code synthesis is to synthesise a program Ψ which satisfies the following condition: for any acceptable input $\tilde{I}$, the synthesised program Ψ should output an acceptable output $\tilde{O}$. To synthesise Ψ, the code synthesis algorithm receives a program specification as parameter, in the form of an "input-output" pair set $\mathcal{V}$. Then, for any given pair $(I, O) \in \mathcal{V}$, whenever I is the argument of Ψ, then the program should output the value O.

One technique in code synthesis, the one used in our method, is the *component-based synthesis* [14]. In this case, the code synthesis algorithm receives a set of "base components" $\mathcal{C}$, in addition to the set $\mathcal{V}$. The specification of the to-be synthesised algorithm is the following: for any given pair $(I, O) \in \mathcal{V}$, whenever I is the argument of Ψ, then the program has to return the value O; and, in order to compute this value, it can only use components from the $\mathcal{C}$ set.

In component-based synthesis, the synthesis algorithm can be viewed as a higher-order function. It receives functions as arguments (the component set $\mathcal{C}$ includes different functions –unary, binary and ternary operators), and it returns another function (target program Ψ). In our case, we use comparison operators ($C_>$, $C_\geq$, $C_=$, $C_\neq$), logic operators (C_{not}, C_{or}, C_{and}), linear arithmetic operators (C_+, C_-), multiplication ($C_\times$) and `if-then-else` (C_{ite}).

Example

We now illustrate component-based synthesis with an example. Suppose we want to build and algorithm to answer whether the sum of two values p and q equals a number n. Suppose we select the component and input-output pair sets as in Demo 4.5.1(a).

One valid solution is presented in Demo 4.5.1(b). There are three inputs: p, q and n. And there are 3 components: C_+, $C_=$ and $C_\times$. Then, the target algorithm can compute up to 6 different values (one for each input, and one value for the result of operating with each component). However, the return statement can only return one of these values; and it should return that value which enables the algorithm to comply with the specification. In the presented solution, `o6` is the returned value. Note that this statement is equivalent to

```
𝒞 = {C+, C=, C×}

𝒱 = {
    ({p=1,q=2,n=3}, true),
    ({p=2,q=-1,n=1}, true),
    ({p=11,q=2,n=3}, false)
    }
```

```
Ψ(I):
    o1 := I.p
    o2 := I.q
    o3 := I.n
    o4 := o2 + o1
    o5 := o3 × o4
    o6 := o4 == o3
    return o6
```

(*a*) (*b*)

Demonstration 4.5.1. (*a*) Components and specification. (*b*) Valid code synthesis solution.

"`o4 == o3`", in turn equal to "`(o2 + o1) == o3`", equal to "`(q + p)==n`". To be valid, the program in Demo 4.5.1(*b*) should comply with the three (I, O) pairs of $\mathcal{V}$. We verify this for the first (I, O) pair. The output of this program for input $I = \{p = 1, q = 2, n = 3\}$ is "`(2+1)==3`" $\equiv$ *true*, which is equal to the specified output. Finally, note that `o5` is not used to compute the returned value. However, we allow this case because the algorithm is semantically equivalent with our without it, so it is not a concern for code synthesis.

Reference

One way to solve the code synthesis problem is to encode the description of the target program Ψ with first-order logic constraints. The constraints describe both the syntax of the algorithm (such as number of lines, declaration of local variables, etc) and the semantics (to make the program compliant with the specification). Then, an SMT solver is used to decide whether there exists a solution satisfying every constraint. If there is, the solution is decoded back and translated into an algorithm.

The technique using the SMT-solver originates in [4] under the name *component-based synthesis*. It is refined in [14] with the use of input-output pairs as the algorithm specification, and, finally, it is also used in [11] with an explicit aim towards object-oriented programming. Also in [11], code synthesis is used to generate a predicate, derived from the synthesised algorithm with backwards traversal of it. That is, we can condense the algorithm into an expression; and, because the output of the algorithm is a boolean value, the condensed expression is a predicate. For instance, the generated predicate of Demo 4.5.1 would be: "`(q + p)==n`". In this thesis we also use this technique.

We postpone the last part of the explanation of the component-based synthesis until Appendix B, so we can continue with the main task of this thesis: infinite loop repair.

4.5.2 Synthesis of a New Looping Guard

We now use the synthesis method described in Subsection 4.5.1 to generate a new looping guard for the infinite loop. The specification of the new looping guard can be informally expressed as follows: the looping guard predicate should allow every test executing the infinite loop to both halt and pass. To synthesize the new looping guard, we need two arguments: the component set and the input-output pair set.

Component Set

The selection of components is done accumulatively. We start off with an empty set of components. In this case, there is only one possibility to find a new looping guard: an input $i \in I$ of the specification has to be equal to the corresponding output for every (I, O) pair of $\mathcal{V}$. If this is the case, then the new looping guard would simply be "`while (i)`". If not, we formulate a new SMT problem with the same specification $\mathcal{V}$ and a new non-empty component set. We try to find a new looping guard. If we succeed, the synthesis phase is finished. If not, we keep adding components until either a new looping guard is found, or until we exhaust all of the available components and finish the synthesis phase unsuccessfully.

In our implementation, the component set is fixed and the first components we add are comparison operators ($C_>$, $C_\geq$, $C_=$, $C_\neq$), then logic operators (C_{not}, C_{or}, C_{and}), then linear arithmetic operators (C_+, C_-), then `if-then-else` (C_{ite}), and, finally, multiplication ($C_\times$). We choose this order based on previous experience with code synthesis in [11].

Input-Output Pair Set

Whereas we directly pick the components to formulate an SMT problem, yielding the input-output pair set is a whole different story. This set represents the specification of the new looping guard. Normally, it is the programmer who gets to write the looping guard. In doing so, she usually writes the looping guard already knowing which variables are involved in the predicate, predicting the predicate will evaluate to $false$ accordingly. She can do this beacuse of her *human* comprehension of the overall loop's specific purpose. However, this methodology is not appropriate when we plan to generate a looping guard automatically. So we do the opposite. Based on the evaluations of the looping guard (which take place when running test cases), we register how the looping guard should evaluate in each iteration. Then, through code synthesis, we generate a looping guard that evaluates exactly to those same values.

The input-output specification $\mathcal{V}$ is assembled as follows. The collection of input-output pairs is done by the loop monitor. At each iteration it creates an (I, O) pair associating the decision of the loop monitor to O and the context information –whose collection is described in Subsection 4.5.3– to I.

4.5.3 Runtime Value Collection

The context information is collected by the loop monitor within the callback in line 8 of Demo 4.2.1. The context information reflects the local state of the program at each iteration. It is composed of variables collected with a source code analysis technique which we refer to as "runtime value collection". It allows to collect values in five different ways:

Reachable variables: we scan the scope of the loop to gather every *reachable variable*. A reachable variable is a variable with two qualities: it is accessible from the loop scope (it is declared within the lexical scope of the loop) and it is initialized (to prevent compilation errors). It could either be a local variable, method parameters or instance fields.

Visible field access: for each reachable variable of a user-defined type, we also gather its visible fields. That is, if the class (owner of the method containing the loop being instrumented) has visibility of some of the variable's fields, we use them. We only refer to user-defined types, so "`anArray.length`" would not be a visible field access.

Getters: for a reachable variable of a user-defined type, we also include, when possible, procedure calls to "getter methods". To do this, we must review the source code declaration of the variable class in search of such methods. We consider that a method is a getter if it has the following characteristics: it has no parameters, it is implemented in one line, the line is a `return` statement, and the returned element is an instance field. If we find a getter, and the class has visibility access to it, then we use it.

Recycling of the original looping guard: although the original looping guard is not used as the real looping guard after the loop instrumentation, it is highly likely that it still provides precise information about the iteration context. For this reason, we also include the value of its evaluation.

Subvalues of the original looping guard: whenever possible, we also inspect the values of subcomponents of the original looping guard. For instance, if the original looping guard is a conjunction, we also include the evaluation of each subpredicate of the conjunction.

We have now shown different alternatives to gather context information within the lexical scope of the loop. However, SMT solvers only support boolean or numeric values. This means we have to extract information of the supported types from the amassed variables:

Extraction by value: for a variable of primitive type (`boolean`, `char`, `int`, `double`, etc.) we are able to copy by value.

Extraction by queries: for each gathered variable of a non-primitive type, we perform different queries. First, we check nullness; and, if the variable is not `null`, we may also extract information by using a hardcoded list of typical queries (such as the length of a `String`, or the size of a `List`). We can see the hardcoded queries in Table 4.5.1. The way to interpret the table is: if the variable's class subclasses a given superclass, then we perform the corresponding queries on the variable.

Superclass	**Queries**
Object	`variable != null`
Array	`variable.length`
Iterator	`variable.hasNext()`
Enumeration	`variable.hasMoreElements()`
Collection	`variable.size()`
	`variable.isEmpty()`
CharSequence	`variable.length()`
	`variable.length()==0`
Dictionary	`variable.size()`
	`variable.isEmpty()`
Map	`variable.size()`
	`variable.isEmpty()`

Table 4.5.1. Queries for each type.

Finally, there is one last refinement phase of the input-output pair set. Firstly, we want each set I of every pair $(I, O) \in \mathcal{V}$ to have the same elements. That is, $\forall (I_1, O_1), (I_2, O_2) \in \mathcal{V}$ we have that $x \in I_1 \Leftrightarrow x \in I_2$. Secondly, we enrich each set I of every (I, O) pair with three constants $\{-1, 0, 1\}$. These are values commonly used in predicates, so we make sure they are available for code synthesis. In third place, we remove redundant information. Note that, as the definition of set implies, there cannot be two different pairs (I_1, O_1), $(I_2, O_2) \in \mathcal{V}$ such that $(I_1 = I_2 \wedge O_1 = O_2)$. To go further, we remove input elements with the same value in every (I, O) pair if that value is one of the constants we used to enrich set $\mathcal{V}$. That is, if $x \in I_1$ (where $(I_1, O_1) \in \mathcal{V}$) and $I_1.x = c$ with $c \in \{-1, 0, 1\}$, and there isn't a set I_2 where $I_2.x \neq c$, we remove x from every I_i set.

4.5.4 Synthesis algorithm

The algorithm of the new looping guard synthesis can be seen in Algorithm 4.5.2. The first step is to collect the input-output pair set (detailed in Algorithm 4.5.1). To do this we execute every test using the loop and fetch the collected input-output pairs of every test run. Once we obtain this specification, we start the search of a new looping guard. We begin with an empty component set. We formulate an SMT problem and use a solver to find a solution. If we succeed, we transform the SMT solution back into a boolean code expression, the patch. If not, we add operators to the component set and formulate a new SMT problem. We do this until we exhaust all of the components or a correct looping guard has been synthesized.

Algorithm 4.5.1. Obtaining the input-output pair set.

```
1: procedure SPECIFICATION(loop, thresholds, src)
2:     $V \leftarrow \{\}$
3:     monitor ← loop.GETMONITOR()
4:     tests ← TESTSOF(loop)
5:     for test ∈ tests do
6:         if ISHANGINGTESTOF(loop, test) then
7:             threshold ← thresholds.KEYFOR(test)
8:             monitor.SETLIMIT(threshold)
9:         end if
10:        RUN(src, test)
11:        pairs ← monitor.GETPAIRS()
12:        V.ADDALL(pairs)
13:    end for
14:    return V
15: end procedure
```

Algorithm 4.5.2. Patch synthesis.

```
1: procedure FINDPATCH(loop, thresholds, src, tests)
2:     spec ← SPECIFICATION(src, loop, thresholds)
3:     components ← {}
4:     while not EXHAUSTEDALL(components) do
5:         smtProblem ← ENCODETOSMT(spec, components)
6:         if smtProblem.ISFEASIBLE() then
7:             solution ← smtProblem.SOLUTION()
8:             patch ← DECODETOPATCH(solution)
9:             return patch
10:        end if
11:        operators ← NEXTOPERATORBUNDLE()
12:        components.ADDALL(operators)
13:    end while
14: end procedure
```

This concludes the explanation of our method. To explain the details of this synthesis technique, it requires a large amount of space. We encourage the attentive reader to refer to Appendix B for an in-detail explanation of the component-based synthesis and to Appendix A for a full end-to-end example of the repair process.

5. EVALUATION

In this section we present the evaluation on our implementation, *Infinitel*. Our evaluation is based on the repair of 7 seeded bugs and 7 real bugs. We aim to answer the following research questions:

RQ1 *Competence:* does *Infinitel* solve the bugs? How does it compare with a human-written one?

RQ2 *Performance:* does *Infinitel* solve the bugs in a reasonable amount of time? What is the bottleneck of the repair method?

RQ3 *Adequacy:* how hard is the code synthesis of each patch? Does the synthesis based on SMT problems scale?

RQ4 *Limit:* does the 1 million iteration limit (Section 4.3) have type I errors (false positives)? Does it affect the performance of *Infinitel*?

RQ5 *Technique:* how does our Runtime Value Collection technique (Subsection 4.5.3) impact on each patch? Is it sufficient to describe the loop state in each bug?

RQ6 *Records:* what are the angelic record values (Section 4.4) for the real bugs? Is our strategy –probing values– appropriate for this step?

RQ7 *Idempotence:* is there any idempotent loop among the real bugs? Would a repair method taking advantage of this improve performance?

5.1 Evaluation Setup

5.1.1 Environment

Infinitel is implemented on *Java*, running on a JRE version 7, with a maximum heap of 2 GB. The SMT solver used is *Z*3 [22] version 4.3.2. The operating system where the evaluation is performed is `OS X Mavericks`. The full specifications are shown in Table 5.1.1.

Operating System	`OS X 10.9`
RAM	`4GB RAM 1.067MHz DDR3`
CPU	`2.8 GHz Intel Core 2 Duo`
Heap Size	2048 MB
SMT Solver	`Z3-SMTLib v.4.3.2`
Java Platform	`Java SE Runtime Environment build 1.7.0_55-b13`

Table 5.1.1. Running environment of the evaluation.

5.1.2 Methodology

Our evaluation is based on the repair of 7 seeded bugs and 7 real bugs. In both cases, each bug consists of one infinite loop with at least one hanging test. We now describe each type of bug.

Seeded Bugs

A seeded bug is a project deliberately "infected" with a manually created infinite loop. We create four projects which only have one class, one test class and one infinite loop. We also have three projects for large scale evaluation. These three projects are real projects which are manually infected with an infinite loop. To achieve this, we select a loop in the project and we perform two small transformations on it. Firstly, we substitute the looping guard for "`while (true)`"[1]. Secondly, the loop body is wrapped with a `try/catch` with an empty `catch` block. This is done to prevent an exception from breaking the infinite loop.

For the large scale evaluation of seeded bugs we use *Collections* and *Math* projects (same commits as shown in Table 3.2.1). The first two bugs come from two different infected loops on *Collections* (`AbstractMapBag.java` on line 590 and `AbstractDualBidiMap.java` on line 352), whereas the third one comes from an infected loop on *Math* (`FastMath.java` line 3 120).

Real Bugs

Name	Repository	Commit	Subproject	Test
csv	`git://git.apache.org/commons-csv.git`	4dfc8ed	–	`Y`
fop	`git://git.apache.org/fop.git`	13984cc	–	`N`
pdfbox A	`git://git.apache.org/pdfbox.git`	b10cf48	–	`N`
pdfbox B	`git://git.apache.org/pdfbox.git`	a2ab77f	fontbox	`N`
pig	`git://git.apache.org/pig.git`	5abfbd0	piggybank	`Y`
tika	`git://git.apache.org/tika.git`	1b694e7	tika-parser	`N`
uima	`git://git.apache.org/uima-uimaj.git`	155596a	jVinci	`N`

Table 5.1.2. Real bugs.

The seven real bugs come from existing projects of the Apache Git repositories [27]. To find real bugs, we individually analyze some projects looking for commits reporting and fixing an infinite loop bug. Specifically, we perform a keyword[2] based search on the Git [26] log of each project repository.

We depict each real bug in Table 5.1.2[3]. In the case of `csv` and `pig` the commit includes both code changes to fix the infinite loop and a test case validating those changes. For the rest of the commits, the commit does not include a validating test case. Then, because our method requires the manifestation of the infinite loop in at least one test case, manually created tests were added for the 5 remaining commits. The policy followed to manually create tests is the following: *a*) at least one of these tests has an infinite execution of the loop attempted to be fixed by the commit changes; *b*) the added hanging tests halt without the changes introduced in the commit and pass with them; and, *c*) the added and not hanging tests, if any, pass.

[1] Actually, changing to "`while (true)`" would raise compilation errors for unreachable code. We use an equivalent form: "`while ("".isEmpty())`".

[2] Used keywords: infinite, loop, iteration, hang, endless, ending, terminating.

[3] The `pdfbox B` bug is detected and incorrectly reported as fixed in commit `e41cbd1`, but it is only fixed in later commit `a2ab77f`. We use the buggy loop in the first commit and use the second commit to compare the fix.

To evaluate *Infinitel* on these bugs we revert the committed human fix (preserving validating tests, in `csv` and `pig`). This way, we can compare the found patch against the human fix.

5.1.3 Metrics

We present here the different aspects to be measured from the repair of each bug. We group them in two categories: "basic metrics" and "time metrics". Basic metrics are concerned with the description of the bug. Time metrics are concerned with execution time of different actions. Unless indicated otherwise, each time metric is rounded to units, which is seconds. Time metrics are used to answer *RQ2*.

Basic Metrics

Tests: the total number of tests in the test suite of the project.

Added Tests (only in real bugs): the number of tests added to reproduce the infinite loop bug.

Added LOC (only in real bugs): the total number of lines of the added tests.

Loop Tests: the number of tests executing the detected infinite loop. Recall that not every test is a hanging test; the infinite loop can be executed –with finite executions– in passing test cases as well. The number of tests executing the infinite loop is related to *RQ3* and *RQ5*, because we extract loop state information from the execution of each test (Algorithm 4.5.1).

Hanging Tests: the number of tests which do not halt due to the infinite loop. This number is related to *RQ6*, because for each hanging test we must find an angelic record (Section 4.4).

Idempotence (only in real bugs): whether the hanging tests pass or not when they are run for the first time (Algorithm 4.3.1), forcing the infinite loop to break during the infinite execution. If they do, we can say that the infinite loop behaves like an idempotent loop. That is, during the interrupted infinite execution, the loop completed more iterations than needed, but this did not ultimately affect the result of the test. Cleary, this metric is related to *RQ7*.

Angelic Record (only in real bugs): the value of the highest angelic record for the infinite loop out of all hanging tests. For each hanging test we search for an angelic record (Section 4.4), and we use the highest value to analyze if probing values is a good strategy for this step (*RQ6*).

Total Traces: the size of the input-output pair set described in Subsection 4.5.2. This number impacts on the number of constraints of the SMT problems created during code synthesis.

Context Size: the size of each trace. That is, the number of inputs inside each input-output pair, plus 1 (for the output value). This number is related to *RQ5*, because it represents the number of extracted values (Subsection 4.5.3) being used to describe the state of each loop iteration.

SMT Formulations: the number of total SMT problems needed to synthesise a patch. As indicated in Algorithm 4.5.2, we successively create SMT problems until the synthesis suceeds. The number of SMT problems needed to find a solution is a good metric to classify the difficulty of the found patch (*RQ3*).

SMT Components: the number of total components used in the synthesised patch (Subsection 4.5.1).

SMT Component Types: the number of different component types used in the synthesised patch (there are 5 different types: comparison, logic, linear arithmetic, multiplication and `if-then-else`).

Application Classes: the total number of declared classes in the project source code, excluding test classes. This metric is also equal to the total number of classes which are instrumented and recompiled during the project instrumentation (Section 4.2).

LOC: lines of code in the project source code, excluding test code. Figures are obtained with *CLOC* [25].

Time Metrics

Instrumentation: time to implant the loop monitors in every `while` of the project source code (Section 4.2).

Compilation: time to compile the instrumented source code.

Test Suite: time to run the test suite of the project (Algorithm 4.3.1). This metric includes the time of running –and inducing loop termination of– hanging tests.

Hanging Tests: time to run –and induce loop termination of– the hanging tests of the infinite loop. Every infinite execution is interrupted after a maximum iteration number is reached (Section 4.3).

Angelic Value Mining: time to find the angelic records of each hanging test (Section 4.4).

Value Collection: time to collect contexts for tests executing the infinite loop (Algorithm 4.5.1).

SMT Solving: overall time solving SMT problems until a solution is found.

Total Time: the sum of the previous 7 metrics, which accounts for approximately the total execution time to fix the bug.

Total Time (timestamp): the human readable equivalent of Total Time with the format of hours, minutes and seconds.

5.2 Results

5.2.1 Metrics Results

We present here the values of the different evaluation metrics for the seeded bugs (Table 5.2.1) and real bugs (Table 5.2.2).

Seeded Bugs

	Collections						
	A	B	Math	Ex.1	Ex.2	Ex.3	Ex.4
	Basic Metrics						
Tests	14 792	14 792	6 077	5	2	3	3
Loop Tests	11	57	1	5	2	3	3
Hanging Tests	11	57	1	1	1	1	1
Total Traces	3	3	53	15	4	3	61
Context Size	15	9	67	4	8	4	9
SMT Formulations	1	1	2	4	1	2	2
SMT Components	0	0	1	9	0	1	1
SMT Component Types	0	0	1	3	0	1	1
Application Classes	463	463	1 188	1	1	1	1
LOC	25 338	25 338	91 878	11	10	15	43
	Time Metrics (in seconds)						
Instrumentation	18	17	34	2	1	1	1
Compilation	13	12	17	1	1	1	1
Test Suite	245	795	589	0	0	0	0
Hanging Tests	47	407	0	0	0	0	0
Angelic Value Mining	14	2	0	0	0	0	0
Value Collection	1	1	0	0	0	0	0
SMT Solving	11	6	6	6	0	0	1
Total Time	302	833	646	9	2	2	3
Total Time (timestamp)	0:05:02	0:13:53	0:10:46	0:00:09	0:00:02	0:00:02	0:00:03

Table 5.2.1. Evaluation of *Infinitel* on 7 seeded bugs.

Real Bugs

	csv	fop	pdfbox		pig	tika	uima
			A	B			
	Basic Metrics						
Tests	83	2 693	24	11	243	476	2
Added Tests	0	3	1	4	0	1	2
Added LOC	0	36	25	32	0	22	11
Loop Tests	33	19	4	4	22	6	2
Hanging Tests	1	3	2	1	1	1	1
Idempotence	Y	Y	Y	Y	Y	N	N
Angelic Record	1	45	0	0	3	0	10
Total Traces	6 631	209	1 474	6	23	703	12
Context Size	41	15	15	11	14	37	13
SMT Formulations	2	4	2	3	3	3	2
SMT Components	1	4	1	2	2	4	1
SMT Component Types	1	3	1	2	2	2	1
Application Classes	11	2 340	429	93	211	263	83
LOC	1 218	157 445	39 551	9 339	11 380	19 767	7 135
	Time Metrics (in seconds)						
Instrumentation	2	45	11	5	8	14	6
Compilation	1	33	7	3	7	13	4
Test Suite	2	330	57	0	442	102	2 898
Hanging Tests	1	0	1	0	69	1	2 897
Angelic Value Mining	0	1	0	0	0	0	0
Value Collection	11	5	27	0	61	11	0
SMT Solving	1 781	3 205	21	1	7	369	3
Total Time	1 797	3 619	123	9	525	509	2 911
Total Time (timestamp)	0:29:57	1:00:19	0:02:03	0:00:09	0:08:45	0:08:29	0:48:31

Table 5.2.2. Evaluation of *Infinitel* on 7 real bugs.

5.2.2 Patches

We present here the comparison between the found patch and human-written code. In Table 5.2.3 we compare the original looping guard –before the manual "infection"– against the found patch (the faulty looping condition is the infected one). In Table 5.2.4 we compare the faulty looping condition against the committed human fix and the found patch.

Seeded Bugs

Collections A	
faulty	`"".isEmpty()`
original	`it.hasNext()`
infinitel	`buf.length()==0`

Collections B	
faulty	`"".isEmpty()`
original	`it.hasNext()`
infinitel	`it.hasNext()`

Math	
faulty	`"".isEmpty()`
original	`(mantissa >>> 52) != 1`
infinitel	`(mantissa)<(FastMath.TWO_POWER_52)`

Ex. 1	
faulty	`b != a`
infinitel	`!((((a)+(-1))<(b))&&((((1)-((a)+(-1)))<=(-1))||((a)==(b))))`

Ex. 2	
faulty	`oneIteration || !  oneIteration && a == 0`
infinitel	`(a == 0)`

Ex. 3	
faulty	`true`
infinitel	`(-1)<(aCopy)`

Ex. 4	
faulty	`canKeepConsuming(index, word)`
infinitel	`(this.consumer.getSize())!=(this.consumer.getConsumed())`

Table 5.2.3. Patches for seeded bugs.

Real Bugs

	csv
faulty	`!tkn.isReady`
manual	`!tkn.isReady && tkn.type != TT_EOF`
infinitel	`(tkn.type)<(0)`

	pig
faulty	`(!((fileStatusArr = fs.listStatus(path)) == null || fs.isFile(path)))`
manual	`(!((fileStatusArr = fs.listStatus(path)) == null || fs.isFile(path) || fileStatusArr.length == 0))`
infinitel	`(!(((fileStatusArr = fs.listStatus(path)) == null) || (fs.isFile(path))))` `&&((0)<(fileStatusArr.length))`

	tika
faulty	`getContentLength() < getBlockLength()`
manual	adding variable `continueLoop`: `continueLoop && getContentLength() < getBlockLength()`
infinitel	`((getContentLength()) < (getBlockLength()))` `&&((!((this.chmSection.getData().length)==(this.state.getWindowSize())))` `||(this.state.getMainTreeTable()!=null))`

	fop
faulty	`(scale < 1 && nextStepFontSize > baseFontSize || scale > 1 && nextStepFontSize < baseFontSize)`
manual	adding a `break` statement
infinitel	`(((scale < 1) && (nextStepFontSize > baseFontSize)) || ((scale > 1) && (nextStepFontSize < baseFontSize)))` `&&(((FontSizePropertyMaker.FONT_SIZE_GROWTH_FACTOR)+` `((FontSizePropertyMaker.FONT_SIZE_GROWTH_FACTOR)-(nextStepFontSize)))<(-1))`

	pdfbox A
faulty	`(amountRead = rawData.read(buffer, 0, Math.min(mayRead,BUFFER_SIZE))) != -1`
manual	adding wrapping `if`
infinitel	`((amountRead = rawData.read(buffer, 0, Math.min(mayRead,BUFFER_SIZE))) != -1)` `&&(filterIndex)<(amountRead))`

	pdfbox B
faulty	`(amountRead = read(data, totalAmountRead, numberOfBytes-totalAmountRead)) != -1 && totalAmountRead < numberOfBytes`
manual	modifying `read()` method
infinitel	`((amountRead = read(data, totalAmountRead, (numberOfBytes - totalAmountRead))) != (-1))` `&&(totalAmountRead < numberOfBytes))` `&&((amountRead)==((numberOfBytes - totalAmountRead))`

	uima
faulty	`offset > 0`
manual	modifying loop body
infinitel	`(indent.length())!=(offset)`

Table 5.2.4. Patches for real bugs.

5.3 Discussion

Here, we answer the research questions presented at the beginning of Chapter 5.

5.3.1 *RQ1* Competence

Seeded Bugs

We can compare the patch found for each bug with the original looping guard of bugs `Collections A`, `Collections B` and `Math`, shown in Table 5.2.3. For `Collections A`, the found patch is different from the original looping guard. However, for `Collections B` the original looping guard was restored. And, for `Math`, the original looping guard is restored semantically, but not syntactically: whereas the original looping guard checks that `mantissa` is lower than 2^{52} using bitwise right shift operator, the found looping guard does so by comparing with the value of constant `TWO_POWER_52`. For the rest of the seeded bugs, there is no reference to compare the found patch with.

Real Bugs

```
- while (condition) {
+ while (newCondition) {
```

```
+ boolean flag = true;
- while (...) {
+ while (flag && ...) {
    ...
+    flag = ...;
}
```

```
while (...) {
...
+  if (...) {
+    break;
+  }
}
```

```
+  if (...) {
    while (...) {
      ...
    }
+  }
```

(*a*) (*b*) (*c*) (*d*)

Demonstration 5.3.1. Human fixes of: (*a*) `csv` and `pig`; (*b*) `tika`; (*c*) `fop`; (*d*) `pdfbox A`

Here we compare the patch found for each bug with the committed human fix. Now, bare in mind that the only possible fix we consider is replacing the looping guard of the infinite loop by another looping guard. In practice, we find that the real bugs manifest different repair strategies.

Let us first consider `csv` and `pig` bugs. The human fix for these bugs is similar to Demo 5.3.1(*a*), which is the same repair strategy as our repair method. We can compare the found patch with the human fix, shown in Table 5.2.4. For `csv`, despite that the fixes are different, both the human fix and the found patch base the looping guard on the value of `tkn.type`. In the case of `pig`, the human fix and the found patch are equivalent (`fileStatusArr` is an array, so the length is either 0 or positive).

We now consider `tika` bug. Whereas our found patch simply restricts the original looping guard with a conjunction operation (Table 5.2.4), the human fix is more complex. In this case, it is similar to Demo 5.3.1(*b*): a boolean variable `continueLoop` is introduced in the fix, the variable is updated at the end of every iteration, and the value of this variable is used in the new looping guard.

If we analyze `fop` bug, we will again find different repair strategies. This time, the human fix is similar to Demo 5.3.1(*c*): a break statement is added at the end of the loop body.

The same happens when we compare the found patch and the human fix in `pdfbox A` bug. The human fix has a repair strategy similar to Demo 5.3.1(d): the introduction of a wrapping `if` around the `while`. This repair strategy is the same as the one mentioned in Section 2.3 for wrong precondition bugs. That is, the infinite loop is due to a wrong precondition to enter the loop. Correspondingly, we find that the Angelic Record for the hanging tests in `pdfbox A` (Table 5.2.2) is 0: this indicates that the precondition to enter the loop should evaluate to false.

Finally, another two repair strategies are used in `pdfbox B` (human fix modifies a method invoked in the looping guard) and `uima` (human fix adds a statement to the loop body).

In the light of the previous paragraphs, we observe a great versatility of our repair method. On the examples of our evaluation, repairing the looping guard of the infinite lop is equivalent to other repair strategies exposed by the human fixes. This versatility makes our repair method reasonably competent for repairing infinite loops.

5.3.2 *RQ2* Performance

Seeded Bugs

Looking at the time metrics in Table 5.2.1, we notice that the total execution time for the Apache projects (almost 10 minutes on average) is significantly higher than the other seeded bugs (4 seconds on average).

In all three cases, the bottleneck is the running time of the Test Suite, and this is independent of the running time of Hanging Tests in the case of `Collections A` and `Math`. In the case of `Collections B`, the Hanging Tests account for approximately 50.00 % of the time of running the Test Suite. Moreover, `Collections B` has the highest Hanging Tests (57).

Regarding the other 4 seeded loops, `Ex.1` takes the longest to fix, essentially due to the SMT Solving. This seems logical if we look at the value of SMT Formulations (4) for `Ex.1`, compared to the other projects (all lower than 4).

Real Bugs

Looking at the time metrics in Table 5.2.2, we can see that `fop` has the longest repair time with approximately 1 hour. It follows `uima` with almost 49 minutes and `csv` with roughly 30 minutes. For the other 4 real bugs, 10 minutes is enough to repair them. On the whole, based on our evaluation, the longest it can take to repair an infinite loop on a real project is 1 hour. We now analyze the bottleneck of each bug indivudally.

In `csv` the clear bottleneck is the time of SMT Solving: 99.00 % of the total repair time is focalized in that task. Something similar happens in `fop`, with 88.00 %. This may be due to the size of the SMT problems in `csv` (where each of the 6 631 Total Traces accounts for a constraint in the SMT problem) and due to the complexity of the found patch in `fop` (with the highest SMT Formulations of 4).

In `uima`, running the Hanging Tests is the evident bottlneck. This is due to a performance overhead caused by the string concatenation operation. The infinite loop in `uima` only has one statement which is a concatenation of strings with the sum operator; then, in an infinite execution of this loop, one million (the maximum iteration number) different strings are created and copied.

In `pdfbox A` and `pig`, the bottleneck seems again to be related to the test cases. In `pdfbox A` almost 70.00 % of the time to repair the infinite loop (84 seconds) is directed at executing the Test Suite (57 seconds) or executing Loop Tests for Value Collection (27 seconds). This figure escaltes to 95.00 % of the time (503 seconds) in `pig` (442 seconds running the Test Suite and 61 seconds for Vale Collection).

In `tika` the time for the Test Suite and the SMT Solving combined amounts to more than 90.00 % of the total repair time. Finally, in `pdfbox B` the repair is largely delayed by the project instrumentation (the Instrumentation and Compilation amount almost to 90.00 % of the total repair time). However, this is a clear outlier (the total repair time is only 9 seconds).

In sum, on this evaluation, *Infinitel* is practicable in terms of performance. Generally, the bottleneck is located on running test cases or solving SMT problems.

5.3.3 ***RQ3*** Adequacy

Seeded Bugs

We now analyze the code synthesis method for seeded bugs. Here, the SMT Solving time is not crucial even for large projects (`Collections A`, `Collections B` and `Math`).

The SMT Formulations of `Collections A`, `Collections B` and `Ex.2` is 1 (Table 5.2.1). That is, no components were necessary to produce the patch (Subsection 4.5.2). Then, we have `Math`, `Ex.3` and `Ex.4` with an SMT Formulations of 2. In table Table 5.2.3, we can see that they use just one component: `Math` and `Ex.4` use distinct (`!=`), whereas `Ex.3` uses lower than (`<`). Finally, `Ex.1` has an SMT Formulations of 4. That is, four different SMT problems (each one with more components than the previous one) are solved to find the patch (Algorithm 4.5.2). Particularly, 9 operators are used: sum (`+`, twice); substraction (`-`); lower or equal than (`<=`); lower than (`<`); equality (`==`); distinct (`!=`); disyunction (`||`); and, conjunction (`&&`).

Real Bugs

We now analyze the code synthesis method for real bugs. The SMT Solving time varies from seconds, to minutes, to almost an hour. The longest SMT Solving time (almost one hour) is for `fop` bug, which also has the highest SMT Formulations. It is followed by `csv` bug (almost half hour), which has the highest Total Traces (each of them is later translated to one constraint for each SMT problem). In third place we have `tika` (almost 6 minutes), which has 3 SMT Formulations and a high number of Total Traces as well. We can correlate the complexity to find a patch with the complexity of the patch. That is, the more constraints (Total Traces) and the more components the patch has, the "more expensive" the synthesis is (the longer it takes to find the patch).

The SMT Formulations of `fop` is 4 (it uses 4 components); for `tika` (4 components), `pig` (2 components) and `pdfbox B` (2 components) is 3; and, for `csv` (1 component), `pdfbox A` (1 component) and `uima` (1 component) is 2. In all cases at least a component is required to find the patch. This means that to be able to describe the completion point (Section 2.1) we must operate with more than one loop state extracted value (Subsection 4.5.3); there is not a value which can be used to describe the completion point alone (such as "`iter.hasNext()`"). This fact supports the adequacy of a component based code synthesis.

On the whole, the ability to combine and compose components on SMT problems gives the code synthesis a high versatility to generate predicates. In terms of efficiency, we exhibit reasons to favour that this strategy does scale; after all, the evaluation on real bugs is based on large projects.

5.3.4 *RQ4* Limit

Seeded Bugs

Regarding the one million iteration restriction, we mention that in the case of `Math`, there was a false positive. There is a non-infinite loop that needs over one million iterations in one test so it can pass (as said on Subsection 3.3.9). For this reason, we use 5 million as the new limit. However, this did not affect the performance in this particular case: the time to run the hanging test in `Math` is less than one second.

Real Bugs

Among the real bugs, there was no false positives (loops uncorrectly detected as infinite).

In sum, although there will be exceptions (as said in Section 4.3, 1 of 523 loops, counting loops of the projects used in Chapter 3), the one million iteration limit is a safe threshold to detect infinite loops.

5.3.5 *RQ5* Technique

Seeded Bugs

We now analyze the impact of our runtime value collection technique (Subsection 4.5.3) on seeded bugs. We highlight different ways of gathering context information by looking at the found patches in Table 5.2.3. We use extraction by queries in `Collections A` ("`buf.length()==0`") and `Collections B` ("`it.hasNext()`"). We use extraction by value from reachable variables in `Math` (the local variable `mantissa` and the static field `TWO_POWER_52`) and `Ex.1` (the values of local variables `a` and `b`). We use subvalues of the original looping guard in `Ex.2` ("`a == 0`" comes from the subclause of the original looping guard, and not from the equality components –as evidenced by an SMT Formulations of 1 for `Ex.2`). Finally, we use getters in `Ex.4` (for instance, "`this.consumer.getSize()`").

Real Bugs

We now analyze the impact of our runtime value collection technique on real bugs. We highlight our strategies by looking at the found patches in Table 5.2.4. We use visible field access in `csv` ("`tkn.type`"). We use recycling of the original looping guard in `pig`, `tika`, `fop` and `pdfbox B` (coloured in blue in the table). We use extraction by queries in `pig` ("`fileStatusArr.length`"). We use subvalues of the original looping guard in `pdfbox B` ("`numberOfBytes-totalAmountRead`" in the right hand side of the equality component). We use extraction by value of reachable variables (such as the static field `FONT_SIZE_GROWTH_FACTOR` in `fop`). We also use getters in `tika` (e.g., "`this.state.getWindowSize()`").

In sum, our runtime value collection technique is able to gather context information in different ways. All of them seem to be useful, given that they are all present in the found patches.

5.3.6 *RQ6* Records

Real Bugs

We now discuss the angelic records for real bugs. As shown in Table 5.2.2, the highest angelic record is 45 for `fop`, then 10 for `uima`, 3 for `pig`, 1 for `csv` and 0 for the rest (`pdfbox A`, `pdfbox B` and `tika`). This is consistent to what was said in Section 3.4: the iteration record of each execution of a prototypical loop is lower than 50. Besides, in every real bug, the time of Angelic Value Mining is meaningless next to the Total Time. Then, our strategy proves to tackle the challenge.

5.3.7 *RQ7* Idempotence

Real Bugs

To conclude, we remark that the infinite loops of `csv`, `fop`, `pdfbox A`, `pdfbox B` and `pig` all behave like idempotent loops: when we run the corresponding hanging tests of these loops and force them to stop after the maximum number of iterations is reached, all hanging tests still pass. This aspect could be exploited during the code synthesis phase: it may occur that the completion point (Section 2.1) becomes "easier" to describe if more iterations are performed in each loop execution. If this was the case, the easiness would be reflected on a lower SMT Formulations value for these bugs (which are 2, 4, 2, 3 and 3, respectively). In turn, this would improve the SMT Solving time, which is particularly high for `csv` and `fop`. However, we do not focus on idempotent loops on this thesis (see Section 3.4), so we leave this question for future work.

6. APPROACH DISCUSSION AND LIMITATIONS

Our approach focuses only on `while` loops, but we believe it is extensible to other constructs (`for`, `do-while`) because no assumptions are made on this regard.

We also mention that our approach lets us find a fix in an object-oriented fashion. This is expressly visible during runtime value collection. We first support the usage of getters. We also use reflection to know what queries can be handled by a variable (Table 4.5.1). And then, we associate each value to a code snippet, so that the new looping guard can be translated to a readymade predicate.

In other respects, our approach does require three assumptions.

The first one is that in each test case there is at most one infinite execution. That is, if the infinite execution is interrupted, any subsequent execution –during the same test run– will be finite. Consider the run of a hanging test case. There can only be one loop iterating endlessly. We assume that this loop is executed zero or more times with finite executions, but at most once with an infinite execution. We use this assumption in Algorithm 4.4.1 because we only probe the angelic record for that single infinite execution.

The second one is that the hanging tests have a deterministic execution. If during the execution of a hanging test the n^{th} execution of an infinite loop is an infinite execution of the loop, then the n^{th} execution of that loop is the infinite execution every time that test is run. We use this assumption in Algorithm 4.4.1, because we probe the angelic record in a specific execution number.

The third one is that all tests are run in a single thread. Concurrency is not supported with our loop instrumentation.

At the same time, we identify four limitations of our approach:

Intermittent failure: non-determinism in passing tests may also impact on the patching guard, if one is found. We only run each test once to collect runtime values; and the patching guard guarantees to be correct only for the given input-output pair set. However, there may be an unfortunate case for our synthesis method. Suppose a passing test produces two different set of input-output pairs during the n^{th} loop execution of two different runs. Namely, (I_a, O) and (I_b, O). That is, the original looping guard evaluates to the same boolean value, in spite of having different input sets. Because we synthesise the patch using only one of these inputs, say I_a, there are no guarantees that the new looping guard would also evalute to O for input I_b. Hence, the patch could make the once passing tests fail intermittently. One way to eliminate this problem is to assume that every test has a deterministic execution, although this restriction is unnecessary for our synthesis method.

Hampering missing values: another threat to our approach is present in the runtime value collection method. It may occur that the collected values are not comprehensive enough to describe the completion point (Section 2.1); that is, the completion point could only be described by the value of an undetected variable, or by the return value of a method which is not a getter. Under these circumstances, the synthesis will fail to produce a patching guard.

One way to mitigate this problem is to use more method calls when amassing variables for runtime value collection (see Section 8.1). Another way is to improve our runtime value collection technique. For instance, when it comes down to analyzing a loop in a method of an *anonymous class* (in Java), we don't include instance fields, if any. The same measure is taken if the analyzed loop is located inside a constructor. Also, in our implementation, we only use getters on instance fields, and we only access visible fields of method parameters. Then, we could include local variables' visible fields/getters, or even static visible fields.

Theory over practice: in two particular steps of our approach we find theory hindering implementation. Firstly, in Section 4.3, where we describe a straightforward strategy for infinite loop detection. The theoretical demarcation of the Halting problem clearly restrains the implementation of this step. Secondly, in Subsection 4.5.3, when we formulate the code synthesis problem as an SMT problem. Current SMT solver implementations either only support primitive types or have difficulties in working with user-defined types, not to mention exponential time complexity of their algorithms. For this reason we perform the already mentioned "extraction by queries" phase in order to extract primitive-typed data. Nonetheless, we still believe there is still more work to be done to improve our approach in both cases.

Global looping guards: the fourth limitation of our approach is related to looping guard representability. There is a high coupling between our code synthesis phase and the local state of a program. That is, we expect a new looping guard to be inferred based on the state of the program in each iteration. However, a looping guard could as well be defined in terms of a global theoretical boundary instead of the program state itself. Such is the case for fixed-point iteration algorithms like the Newton-Raphson method, where the convergence of the method –under some assumptions– should be mathematically proved. It is unclear what the notion of state comparison may be in those cases, and whether code synthesis based on execution traces is applicable for such algorithms.

7. RELATED WORK

One of the most successful approaches on Automatic Software Repairs is *GenProg* [6], based on *genetic programming* (GP). GP uses computational analogs of biological mutation and crossover to generate new program variations, called “variants”. A user-defined *fitness-function* evaluates each variant, and variants with high fitness are selected for continued evolution. Therefore, the goal is to search for a program variant that retains required functionality and passes all test cases. To reduce the search space, program modifications are favoured in source statements executed during failing test cases. Drawing a comparison, our repair method addresses a specific defect class (infinite loops), whereas in *GenProg*, in principle, any kind of bug is fixable. However, *GenProg* can only find a patch if the repair code already exists in the program, whereas we are able to genuinely synthesise a new expression. Moreover, we do not know how *GenProg* would handle the case of intinite loops and hanging test cases, because it works with *failing* test cases. It expects the test to either pass or fail, but it does not expect the test not to halt.

Another reacent approach is *SemFix* [12]. The methodology consists of finding a faulty statement, and then executing the test cases with symbolic execution on that statement. Symbolic execution is useful to gather constraints on the input-output pairs of the faulty statement. Next, the constraints are encoded into a formulation of a SMT problem, which is fed to an SMT-solver. If a solution exists, it is decoded back to a logical formulation which can eventually be written in the correct language syntax. Drawing a comparision, *SemFix* handles many kind of bugs and, using SMT code synthesis, it does not depend on the repair code to exist already in the program. Our repair method uses a similar code synthesis technique. However, whereas *SemFix* uses symbolic execution to gather input-output pairs, our method uses a runtime value collection technique with focus on finding a patch in an object-oriened fashion. Similar to *GenProg*, we do not know how *SemFix* would handle hanging test cases, because they expect the test to halt and retrieve information from failures.

NoPol [11] is another approach on Automatic Software Repair. The main novelty is the combination of angelic execution (instead of symbolic execution) along with the constraint based synthesis. In angelic execution, a selected statement in the program is interpreted as a query to an angel, who yields a value that would make the current test execution to pass. Therefore, if an angelic value is found for all tests, the constraint based synthesis phase would produce a patch.

As in our repair method, *NoPol* also addresses a specific defect class: wrong conditionals or missing preconditions. As M. Monperrus stated in [7], although there is not a consensus in the Automatic Software Repair community, narrowing the search space for repair to specific defect classes could be the next direction for research. It would enable the community to answer the questions: what are the automatically “repairable” defect classes, why is a specific defect class easy or hard to repair, and what are the techniques to repair each one of them? Nevertheless, a standard list of defect classes has not been defined yet.

This thesis follows a similar approach to the one used in *NoPol*, with two differences.

Firstly, whereas *NoPol* uses a suspicion-ordering criteria for fault localization, we face a different problem: detecting an infinite loop. Secondly, whereas *NoPol* needs to find one boolean angelic value (for each failing test case, the angelic value tells whether some condition must evaluate to true or false), in this thesis we search for an integer angelic record (test threshold) which accounts for the number of iterations a hanging test case needs to complete to halt and pass.

Regarding one of our aforementioned loop properties, the phenomenon of "idempotent loops", has also been observed in [15]: "a specific instance of the loop can iterate for fewer or greater number of iterations without affecting program output". However, their goal is to characterize outcome-tolerant branch instances to discover new ways to enhance the processor's performance. Performance can be limited by the ability to predict branches, but they claim that this prediction does not need to be correct all the time.

We can find an approach for infinite loop detection in *Jolt* [9]. *Jolt* attaches to an application to audit its progress. It records the program state at the start of each loop iteration. If two consecutive loop iterations produce the same state, *Jolt* reports that the application is looping endlessly. Contrastingly, our approach to detect the infinite loop is simply based on a maximum iteration restriction (within a loop execution), supported by our preliminary study on loops (Section 3.3). On the one hand, the criteria used in *Jolt* may detect an infinite loop sooner (because it does not wait until the maximum iteration number is exceeded); and, it would not have false positives (if state Γ makes a transition –an iteration– to state Γ, then this state is going to be repeated indefinitely). On the other hand, our criteria could detect the infinite loop when the same state is repeated within 3 or more iterations, or even when the state is never repeated. Hence, our criteria has a greater recall and lower precision.

Finally, it is also worth mentioning the case found in *Lisp 1.5* manual [10]. Being recursion one of the fundamental features of LISP language, John McCarthy describes a strategy to detect infinite recursion, which is very similar to our infinite loop detection strategy. Whereas we implant loop monitors and count loop iterations, he counts the number of invocations to the function `cons`: "The cons counter is a useful device for breaking out of program loops. It automatically causes a trap when a certain number of conses have been performed. The counter is turned on by executing `count[n]`, where `n` is an integer. If `n` conses are performed before the counter is turned off, a trap will occur and an error diagnostic will be given."

8. CONCLUSIONS

In this thesis, we define the infinite loop defect class and narrow down the problem to propose a practical method to automatically repair infinite loops. Particularly, we focus on the repair of infinite *while* loops with a wrong looping condition. For this goal, we develop both static and dynamic source code analysis techniques, along with a code synthesis technique based on SMT problems.

Although the problem of repairing an infinite loop is related to the undecidable Halting Problem, in the context of test-driven repair the problem becomes manageable. The reason for this is that correctness is defined in relation to the project's test suite, and infinite loop detection is implemented in terms of a maximum iteration threshold.

8.1 Future Work

In future work, we plan to extend our method to support fixing infinite loops with more than one infinite execution during the same test case, and to improve the runtime value collection with other method calls (besides getters). We also think that our method could be applied to fix a more general defect class: wrong loop conditions, contrary to wrong conditions which result in infinite loops. In this case, the only difference with the current approach is the detection of the faulty loop, which in this case would not necessarily have infinite executions. In addition, we plan to extend the scope of our method to repair other forms of loops: `do-while`, `for` and recursion. Also, in this work we have focused on door-door executions. In the future, we should aim at fixing door-window executions as well. Finally, among the ones who favor the research of specific defect classes, it remains for future work to combine different tools, each of them specialized in a particular defect class, into a more complex general automatic bug fixer.

To conclude, we now address a final task for the future of Automatic Software Repair. Through the experience of searching real bugs for the evaluation of our method, we stress the importance of having a universal bug dataset. It is important to rely on a bug dataset for two reasons. Firstly, because finding real bugs is time consuming and hard to achieve: we search for a target commit which fixes a specific defect class, but in practice committers do not follow a standard procedure to commit changes and/or to write log messages. This way, to complete a successful commit search, it is inevitable for the researcher to manually check a bundle of commits until the desired commit is found. Secondly, because Automatic Software Repair research should focus on real problems. And what would be more real than inspecting real bugs and fixes from a universal bug dataset? With a universal bug dataset already at hand, the researcher could inspect, theorize and learn from actual fixes, and, with the knowledge obtained, aim to build more sophisticated repair tools.

9. REFERENCES

[1] A. Arcuri and X. Yao. “A novel co-evolutionary approach to automatic software bug fixing.” In Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2008

[2] D. Kim, J. Nam, J. Song, and S. Kim. “Automatic patch generation learned from human-written patches.” In Proceedings of the 2013 International Conference on Software Engineering, pages 802-811, 2013.

[3] V. Dallmeier, A. Zeller, and B. Meyer. “Generating fixes from object behavior anomalies.” In Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering, pages 550-554. IEEE Computer Society, 2009.

[4] Gulwani, Sumit, et al. “Synthesis of loop-free programs.” ACM SIGPLAN Notices. Vol. 46. No. 6. ACM, 2011.

[5] Pawlak, Renaud, et al. “Spoon: Java Source Code Analysis and Transformation for the Masses.”

[6] Le Goues, Claire, et al. “GenProg: A generic method for automatic software repair.” Software Engineering, IEEE Transactions on 38.1 (2012): 54-72.

[7] Monperrus, Martin. “A critical review of ‘automatic patch generation learned from human-written patches’: essay on the problem statement and the evaluation of automatic software repair.” ICSE. 2014.

[8] Kim, Dongsun, et al. “Automatic patch generation learned from human-written patches.” Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013.

[9] Carbin, Michael, et al. “Detecting and escaping infinite loops with Jolt.” ECOOP 2011–Object-Oriented Programming. Springer Berlin Heidelberg, 2011. 609-633.

[10] McCarthy, John.“LISP 1.5 programmer’s manual”. MIT press, 1965.

[11] F. Demarco, J. Xuan, D. L. Berre, and M. Monperrus. “Automatic repair of buggy if conditions and missing preconditions with SMT.” In Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, pages 30-39. ACM, 2014.

[12] Nguyen, Hoang Duong Thien, et al. “SemFix: Program repair via semantic analysis.” Proceedings of the 2013 International Conference on Software Engineering. IEEE Press, 2013.

[13] Chandra, Satish, et al. “Angelic debugging.” Software Engineering (ICSE), 2011 33rd International Conference on. IEEE, 2011.

[14] Jha, Susmit, et al. “Oracle-guided component-based program synthesis.” Software Engineering, 2010 ACM/IEEE 32nd International Conference on. Vol. 1. IEEE, 2010.

[15] Wang, Nicholas, Michael Fertig, and Sanjay Patel. "Y-branches: When you come to a fork in the road, take it." Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on. IEEE, 2003.

[16] Hamill, Margaret, and Katerina Goseva-Popstojanova. "Common trends in software fault and failure data." Software Engineering, IEEE Transactions on 35.4 (2009): 484-496.

[17] Seacord, Robert C., Daniel Plakosh, and Grace A. Lewis. "Modernizing legacy systems: software technologies, engineering processes, and business practices." Addison-Wesley Professional, 2003.

[18] Presser, L. "Structured Languages." In Proceedings of the May 19-22 1975 National Computer Conference and Exposition. ACM, 1975.

[19] Dijkstra, E. "A Case Against the GOTO Statement." In Communications of the ACM 11, 1968.

[20] Steele Jr, G. "Lambda: The Ultimate GOTO." Vol. 443, AI Lab Memo, 1977.

[21] SMT-LIB: The Satisfiability Modulo Theories Library. `http://smt-lib.org/`

[22] Z3 Theorem Prover. `http://z3.codeplex.com/`

[23] CVC4 Theorem Prover. `http://cvc4.cs.nyu.edu/web/`

[24] Java Language and Virtual Machine Specifications. `http://docs.oracle.com/javase/specs/`

[25] Count Lines of Code. `http://cloc.sourceforge.net/`

[26] Git. `http://git-scm.com/`

[27] Apache Git Repositories. `http://git.apache.org/`

[28] Grep Git Repository. `http://git.savannah.gnu.org/cgit/grep.git/`

References for Appendix C:

[29] *The Java Virtual Machine Specification*, by Tim Lindholm and Frank Yellin.

[30] *Lesson: Packages.*
`http://docs.oracle.com/javase/tutorial/java/package/index.html`

[31] *Java Virtual Machine's Internal Architecture.*
`http://www.artima.com/insidejvm/ed2/jvm.html`

[32] *Understanding Network Class Loaders (Oracle Tutorial)*
http://www.oracle.com/technetwork/articles/javase/classloaders-140370.html

[33] *Classloader : Java Glossary*
`http://www.mindprod.com/jgloss/classloader.html`

[34] *Threads and Class loading*
`http://book.javanb.com/java-threads-3rd/jthreads3-CHP-13-SECT-4.html`

[35] *Object request broker*
http://en.wikipedia.org/wiki/Object_request_broker

[36] *Understanding the Java Classloading Mechanism*
www2.sys-con.com/itsg/virtualcd/java/archives/0808/chaudhri/index.html

[37] *The Lifetime of a Type*
http://www.artima.com/insidejvm/ed2/lifetype.html

[38] *The Linking model*
http://www.artima.com/insidejvm/ed2/linkmod.html

[39] *Compiling from Memory*
http://www.java2s.com/Code/Java/JDK-6/CompilingfromMemory.htm

[40] *Using built-in JavaCompiler*
http://atamur.blogspot.fr/2009/10/using-built-in-javacompiler-with-custom.html

[41] *Create Dynamic Applications with `javax.tools`*
http://www.ibm.com/developerworks/java/library/j-jcomp/index.html

[42] *Java File Manager*
http://docs.oracle.com/javase/6/docs/api/javax/tools/JavaFileManager.html

[43] *Dynamic in-memory Compilation*
http://www.javablogging.com/dynamic-in-memory-compilation/

APPENDIX

A. FULL REPAIR EXAMPLE OF AN INFINITE LOOP

A.1 Example Description

In this section we illustrate the procedure of repairing an infinite loop with a full example. We have three *Java* classes: `Partitioner`, `SumList` and `PartitionerTest`.

Partitioner.java

```java
package uba;

import static java.lang.Math.abs;
import java.util.Iterator;

public class Partitioner {

    public Partitioner(int[] array) {
        partA = new SumList(array);
        partB = new SumList();
    }

    public int partition(int tolerance) {
        Integer a = 0;
        Integer b = 0;
        Integer gain = 0;
        Iterator<Integer> iterA = null;
        Iterator<Integer> iterB = null;
        int diff = Integer.MAX_VALUE;
        while (abs(diff) > tolerance) {
            diff = getPartA().getSum() - getPartB().getSum();
            for (a = 0, iterA = getPartA().iterator(); ; a = iterA.next()) {
                for (b = 0, iterB = getPartB().iterator(); ; b = iterB.next()) {
                    gain = abs(diff) - abs(diff - 2*a + 2*b);
                    if (gain > 0 || !iterB.hasNext())
                        break;
                }
                if (gain > 0 || !iterA.hasNext())
                    break;
            }
            if (gain > 0) {
                getPartA().migrateTo(getPartB(), a);
                getPartB().migrateTo(getPartA(), b);
            }
        }
        return abs(diff);
    }

    public SumList getPartA() {
        return partA;
    }

    public SumList getPartB() {
        return partB;
    }

    private SumList partA;
    private SumList partB;
}
```

SumList.java

```
package uba;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class SumList {

    public SumList() {
        sum = 0;
        list = new LinkedList<Integer>();
    }

    public SumList(int [] array) {
        this();
        addAll(array);
    }

    public void add(Integer e) {
        if (e != 0) {
            sum += e;
            getList().add(e);
        }
    }

    public void addAll(int[] array) {
        for (int e : array) {
            add(e);
        }
    }

    public void migrateTo(SumList sumList, Integer e) {
        if (getList().remove(e)) {
            sum -= e;
            sumList.add(e);
        }
    }

    public Integer getSum() {
        return sum;
    }

    public List<Integer> getList() {
        return list;
    }

    public Iterator<Integer> iterator() {
        return getList().iterator();
    }

    @Override
    public String toString() {
        return getList().toString() + ": " + getSum() ;
    }

    private Integer sum;
    private List<Integer> list;
}
```

PartitionerTest.java

```java
package uba;

import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class PartitionerTest {

    @Test
    public void toleranceOverSumSimple() {
        int[] array = new int[] {1,3};
        Partitioner p = new Partitioner(array);
        assertEquals(2, p.partition(2));
        assertEquals("[3]: 3", p.getPartA().toString());
        assertEquals("[1]: 1", p.getPartB().toString());
    }

    @Test
    public void equalParititonSimple() {
        int[] array = new int[] {1,1};
        Partitioner p = new Partitioner(array);
        assertEquals(0, p.partition(0));
        assertEquals("[1]: 1", p.getPartA().toString());
        assertEquals("[1]: 1", p.getPartB().toString());
    }

    @Test
    public void equalParititonComplex() {
        int[] array = new int[] {2,10,3,8,5,7,9,5,3,2};
        Partitioner p = new Partitioner(array);
        assertEquals(0, p.partition(0));
        assertEquals("[9, 5, 3, 2, 8]: 27", p.getPartA().toString());
        assertEquals("[2, 10, 3, 5, 7]: 27", p.getPartB().toString());
    }

    @Test
    public void toleranceOverSumComplex() {
        int[] array = new int[] {5,8,13,27,14};
        Partitioner p = new Partitioner(array);
        assertEquals(3, p.partition(3));
        assertEquals("[14, 13, 5]: 32", p.getPartA().toString());
        assertEquals("[8, 27]: 35", p.getPartB().toString());
    }

    @Test
    public void unreachableToleranceSimple(){
        int[] array = new int[] {10,20};
        Partitioner p = new Partitioner(array);
        assertEquals(10, p.partition(5));
        assertEquals("[20]: 20", p.getPartA().toString());
        assertEquals("[10]: 10", p.getPartB().toString());
    }

    @Test
    public void unreachableToleranceComplex(){
        int[] array = new int[] {30,50,10,20,60,80,60,90,40,20,70,50,80,90};
        Partitioner p = new Partitioner(array);
        assertEquals(10, p.partition(0));
        assertEquals("[40, 20, 70, 50, 80, 90, 30]: 380", p.getPartA().toString());
        assertEquals("[50, 10, 20, 60, 80, 60, 90]: 370", p.getPartB().toString());
    }
}
```

This example is based on the BALANCED PARTITION PROBLEM: given an array of n integers, find a partition into two subsets which minimizes $|S_1 - S_2|$ (where S_1 and S_2 denote the sums of the elements in each subset). In the given implementation, a `SumList` represents an object which holds an integer list, it provides an interface to add and remove numbers and it also keeps track of the current sum of the list. A `Partitioner` holds an integer array and provides an interface to partition the array into two subsets ($partA$ and $partB$). Lastly, the class `PartitionerTest` contains 6 tests specifying the `Partitioner` class.

The algorithm used to partition the array is an iterative method. It starts off from a trivial configuration: $partA$ holds all the numbers and $partB$ is empty. Then, in each iteration, the partition is improved in one of three ways: *i*) moving one number from $partA$ to $partB$; *ii*) moving one number from $partB$ to $partA$; *iii*) swapping a number of $partA$ with a number of $partB$. Such exchange only takes place if, by doing so, the value of $|S_1 - S_2|$ decreases.

This algorithm is implemented in `Partitioner.partition()` method. The method receives one integer parameter (*tolerance*). This number is used in the termination condition of the loop: as long as $|S_1 - S_2|$ is bigger than *tolerance*, the algorithm tries to improve the current partition. However, with this termination condition, the algorithm will enter into an infinite loop if the tolerance is unfeasible (e.g., the optimal partition of array $[1, 1, 3]$ is $[1, 1]$ and $[3]$; then, a tolerance of 0 will cause the algorithm to loop indefinitely).

In `PartitionerTest` we present 6 tests on `Partitioner` class:

- `PartitionerTest.toleranceOverSumSimple()`:
  Array: `[1,3]`. The optimal difference is 2, partitioning into: `[3]` and `[1]`. The tolerance is 2, so the loop in `Partitioner.partition()` will halt.
- `PartitionerTest.equalParititonSimple()`:
  Array: `[1,1]`. The optimal difference is 0, partitioning into: `[1]` and `[1]`. The tolerance is 0, so the loop in `Partitioner.partition()` will halt.
- `PartitionerTest.equalParititonComplex()`:
  Array: `[2,10,3,8,5,7,9,5,3,2]`. The optimal difference is 0, partitioning into: `[9, 5, 3, 2, 8]` and `[2, 10, 3, 5, 7]`. The tolerance is 0, so the loop will halt.
- `PartitionerTest.toleranceOverSumComplex()`:
  Array: `[5,8,13,27,14]`. The optimal difference is 3, partitioning into: `[14, 13, 5]` and `[8, 27]`. The tolerance is 3, so the loop will halt.
- `PartitionerTest.unreachableToleranceSimple()`:
  Array: `[10,20]`. The optimal difference is 10, partitioning into: `[20]` and `[10]`. The tolerance is 5, so the loop in `Partitioner.partition()` will not halt.
- `PartitionerTest.unreachableToleranceComplex()`:
  Array: `[30,50,10,20,60,80,60,90,40,20,70,50,80,90]`. The optimal difference is 10, partitioning into: `[40, 20, 70, 50, 80, 90, 30]` and `[50, 10, 20, 60, 80, 60, 90]`. The tolerance is 0, so the loop in `Partitioner.partition()` will not halt.

A.2 Infinite Loop Repair

Now that the example has been presented, we can proceed with the repair of the loop in `Partitioner.partition()` method.

A.2.1 Instrumentation

The first thing to get done is the project instrumentation of every `while` loop (Section 4.2). In the running example, the source code of `Partitioner.partition()` is modified –and recompiled– into this one:

```java
public int partition(int tolerance) {
    Integer a = 0;
    Integer b = 0;
    Integer gain = 0;
    Iterator<Integer> iterA = null;
    Iterator<Integer> iterB = null;
    int diff = Integer.MAX_VALUE;
    LoopMonitor LM_0 = Global.getMonitor(0);
    int ITERS_0 = 0;
    while (true) {
        boolean stay = LM_0.decide(abs(diff) > tolerance, ITERS_0);
        if (stay) {
            ITERS_0 += 1;
            diff = getPartA().getSum() - getPartB().getSum();
            for (a = 0,iterA = getPartA().iterator(); ; a = iterA.next()) {
                for (b = 0,iterB = getPartB().iterator(); ; b = iterB.next()) {
                    gain = abs(diff) - abs(diff - 2*a + 2*b);
                    if (gain > 0 || ! iterB.hasNext())
                        break;
                }
                if (gain > 0 || ! iterA.hasNext())
                    break;
            }
            if (gain > 0) {
                getPartA().migrateTo(getPartB(), a);
                getPartB().migrateTo(getPartA(), b);
            }
        } else break;
    }
    return abs(diff);
}
```

A.2.2 Infinite Loop Detection

After instrumenting the source code, the tests on `PartitionerTest` are executed (Section 4.3). Two tests are detected as hanging tests: `unreachableToleranceSimple` and `unreachableToleranceComplex`.

A.2.3 Finding Thresholds in Hanging Tests

The next task is to find the angelic threshold for each hanging test (Section 4.4). In this step, the repair method finds that the test `unreachableToleranceSimple` passes if the loop in `Partitioner.partition()` iterates twice. The test `unreachableToleranceComplex` passes if the same loop performs only 10 iterations.

A.2.4 Patch Synthesis

At this point, we can run the 6 tests in `PartitionerTest` and make them pass. The non-hanging tests are already successful; the hanging tests pass with guidance from the loop monitor (it breaks the loop after the required number of iterations are completed). The next task is to trace runtime values from the test executions (Subsection 4.5.3) in order to infer a new looping guard for the loop in `Partitioner.partition()`. To enable runtime value collection, another loop instrumentation is carried out:

```
public int partition(int tolerance) {
    Integer a = 0;
    Integer b = 0;
    Integer gain = 0;
    Iterator<Integer> iterA = null;
    Iterator<Integer> iterB = null;
    int diff = Integer.MAX_VALUE;
    LoopMonitor LM_0 = Global.getMonitor(0);
    int ITERS_0 = 0;
    while (true) {
        boolean stay = LM_0.decide(abs(diff) > tolerance, ITERS_0);
        LM_0.collectInput("abs(diff) > tolerance", abs(diff) > tolerance);
        LM_0.collectInput("a", a);
        LM_0.collectInput("b", b);
        LM_0.collectInput("diff", diff);
        LM_0.collectInput("gain", gain);
        LM_0.collectInput("iterA", iterA);
        LM_0.collectInput("iterB", iterB);
        LM_0.collectInput("tolerance", tolerance);
        LM_0.collectInput("this.partA", this.partA);
        LM_0.collectInput("this.partB", this.partB);
        if (this.partB!=null) {
            LM_0.collectInput("this.partB.getList()", this.partB.getList());
            LM_0.collectInput("this.partB.getSum()", this.partB.getSum());
        }
        if (this.partA!=null) {
            LM_0.collectInput("this.partA.getList()", this.partA.getList());
            LM_0.collectInput("this.partA.getSum()", this.partA.getSum());
        }
        LM_0.collectOutput(stay);
        if (stay) {
            ITERS_0 += 1;
            diff = getPartA().getSum() - getPartB().getSum();
            for (a = 0,iterA = getPartA().iterator(); ; a = iterA.next()) {
                for (b = 0,iterB = getPartB().iterator(); ; b = iterB.next()) {
                    gain = abs(diff) - abs(diff - 2*a + 2*b);
                    if (gain > 0 || ! iterB.hasNext())
                        break;
                }
                if (gain > 0 || ! iterA.hasNext())
                    break;
            }
            if (gain > 0) {
                getPartA().migrateTo(getPartB(), a);
                getPartB().migrateTo(getPartA(), b);
            }
        } else break;
    }
    return abs(diff);
}
```

We can observe that different type of variables are being collected: local variables (`a`, `b`, `diff`, `gain`, `iterA` and `iterB`), method parameters (`tolerance`), instance fields (`this.partA`, `this.partB`), non-visible fields from getters (`getList()` and `getSum()` for `this.partA` and `this.partB`), and the evaluation of the original looping guard (the value of the predicate "`abs(diff) > tolerance`"). For every collected variable, we provide a `String` representation. This representation is used during code synthesis, to associate numerical or boolean values to code snippets. The decision of whether to iterate or break the loop is also collected (method call to `collectOutput()` with the added local variable `stay`).

With this new instrumentation, we can rerun the tests in `PartitionerTest` and trace runtime values. Here, we show 3 different traces (35 were collected in total). The input elements are extracted values from the collected variables, and the output value is the same as the previous `stay` variable:

```
output: true
input: {"abs(diff) > tolerance"=true,
        "a"=7, "b"=8, "gain"=2, "diff"=-2, "tolerance"=0,
        "iterA!=null"=true,
        "iterB!=null"=true,
        "iterA.hasNext()"=true,
        "iterB.hasNext()"=true,
        "this.partA!=null"=true,
        "this.partB!=null"=true,
        "this.partA.getSum()"=27,
        "this.partB.getSum()"=27,
        "this.partA.getList().size()"=5,
        "this.partB.getList().size()"=5,
        "this.partA.getList()!=null"=true,
        "this.partB.getList()!=null"=true,
        "this.partA.getList().isEmpty()"=false,
        "this.partB.getList().isEmpty()"=false}.

output: false
input: {"abs(diff) > tolerance"=false,
        "a"=3, "b"=1, "gain"=0, "diff"=2, "tolerance"=2,
        "iterA!=null"=true,
        "iterB!=null"=true,
        "iterA.hasNext()"=false,
        "iterB.hasNext()"=false,
        "this.partA!=null"=true,
        "this.partB!=null"=true,
        "this.partA.getSum()"=3,
        "this.partB.getSum()"=1,
        "this.partA.getList().size()"=1,
        "this.partB.getList().size()"=1,
        "this.partA.getList()!=null"=true,
        "this.partB.getList()!=null"=true,
        "this.partA.getList().isEmpty()"=false,
        "this.partB.getList().isEmpty()"=false}.

output: false
input: {"abs(diff) > tolerance"=false,
        "a"=1, "b"=1, "gain"=0, "diff"=0, "tolerance"=0,
        "iterA!=null"=true,
        "iterB!=null"=true,
        "iterA.hasNext()"=false,
        "iterB.hasNext()"=false,
        "this.partA!=null"=true,
        "this.partB!=null"=true,
        "this.partA.getSum()"=1,
        "this.partB.getSum()"=1,
        "this.partA.getList().size()"=1,
        "this.partB.getList().size()"=1,
        "this.partA.getList()!=null"=true,
        "this.partB.getList()!=null"=true,
        "this.partA.getList().isEmpty()"=false,
        "this.partB.getList().isEmpty()"=false}.
```

These data is used to infer a new looping guard for the loop in method `partition()` of class `Partitioner`. The new looping guard must equal, when evaluated on each of the 35 input sets, the corresponding collected output. To do so, a patch synthesis problem is formulated (Subsection 4.5.2). Once solved, the following looping guard is found:

```
        ((0)<(gain))||(this.partB.getList().isEmpty())
```

B. COMPONENT-BASED SYNTHESIS

In this section, we explain how the component-based synthesis (introduced in Subsection 4.5.1) encodes the synthesis problem into an SMT problem, and how it decodes the solution of the SMT problem into an algorithm.

B.1 Overview

The way in which component-based synthesis generates an algorithm is by composing base components. That is, given a set of base components $\mathcal{C}$ and the target algorithm specification $\mathcal{V}$, we generate equations $\psi(I)$ which are a composition of zero or more components from $\mathcal{C}$. We refer to a generated equation $\psi(I)$ as a "component composition". The main difficulty of component-based synthesis resides on finding a component composition suitable for the target algorithm.

Let us revisit Demo 4.5.1. The base components are C_+, $C_\times$ and $C_=$. In addition, we have three inputs: p, q and n. These values are going to be the parameters of the target algorithm. Then, the goal of component-based synthesis is to generate an algorithm using six *ingredients*: C_+, $C_\times$, $C_=$, p, q and n. How we compose these ingredients into a function $\psi(p, q, n)$ determine the return value of the target algorithm. The return value can simply be "$\psi(p, q, n) = q$", or the result of the sum "$\psi(p, q, n) = p+q$", or the result of the equation "$\psi(p, q, n) = (p \times n) + q$", and so on. To determine which of these component compositions are corect for the target algorithm, we use the specification of the target algorithm: the input-ouput pair set $\mathcal{V}$. The equation $\psi(I)$ is correct if and only if $\psi(I) = O$ for every $(I, O) \in \mathcal{V}$.

One way to find the target algorithm is to use *naive exponential search*: generate all possible component compositions and discard those that violate the specification. The problem of this approach is that the number of component compositions that can be constructed using a given set of components is exponential in the number of components.

A more sophisticated search can be attained by delegating exponential reasoning to Satisfiability (SAT) and Satisfiability Modulo Theory (SMT) solvers. We can encode the target algorithm as an SMT problem and try to solve the latter. If a solution exists, the solution can be decoded back to an equivalent algorithm.

B.2 SMT Encoding

The encoding of the target algorithm consists of *describing* the algorithm with a series of first-order logic constraints. Some of the constraints describe the syntaxis of the algorithm, and the rest describe the semantics.

B.2.1 Syntaxis Encoding

Here, we focus on the syntaxis of the algorithm. The main concern is to have a well-formed algorithm which respects the syntactic rules of imperative programming. We explain how to do this with the running example of Subsection 4.5.1.

```
Ψ(I):
    o1 := I.p
    o2 := I.q
    o3 := I.n
    o4 := ?
    o5 := ?
    o6 := ?
    return out
```

(*a*)

```
Ψ(I):
    o1 := I.p
    o2 := I.q
    o3 := I.n
    o4 := o1 × o3
    o5 := o1 == o3
    o6 := o4 + o2
    return o2
```

(*b*)

```
Ψ(I):
    o1 := I.p
    o2 := I.q
    o3 := I.n
    o4 := o2 + o1
    o5 := o3 × o4
    o6 := o4 == o3
    return o6
```

(*c*)

Demonstration B.2.1

We can sketch the skeleton of the target algorithm as shown in Demo B.2.1(a). In Demo B.2.1(b) and Demo B.2.1(c) we can see two possible algorithms following the same outline. The first one represents the component composition "$\psi(p, q, n) = q$" and the second one, "$\psi(p, q, n) = (q + p) == n$". From the skecth, we note that the length of the algorithm is $|\mathcal{I}| + |\mathcal{C}| + 1$ lines of code. In the first $|\mathcal{I}|$ lines we declare local variables for reading the input variables. In the following $|\mathcal{C}|$ lines we declare local variables to store the result of operating with each component. The last statement is the return value *out*.

From the shown examples we see that there are three things missing in the sketch: a) the correspondence between each component and the line where it is used (for instance, C_+ is used in line 6 in Demo B.2.1(b), but it is used in line 4 in Demo B.2.1(c)); b) the correspondence between the operands of each component and the line where it is declared (for instance, the left component of C_+ is declared in line 4 in Demo B.2.1(b), but it is declared in line 2 in Demo B.2.1(c)); c) the correspondence between the returned value and where it is declared (it is declared in line 2 in Demo B.2.1(b), but it is declared in line 6 in Demo B.2.1(c)). These missing correspondences are exactly what we want the SMT problem to find.

We now present the constraints to describe the syntaxis of the algorithm. Let the following definitions hold:

- $\mathcal{I}$: inputs for the target algorithm. For the running example: $\mathcal{I} = \{p, q, n\}$.
- $\mathcal{C}$: base components used in the target algorithm. For the running exmaple: $\mathcal{C} = \{C_+, C_\times, C_=\}$.
- $\mathcal{A}$: operands of each component in $\mathcal{C}$. For the running example we include the operands of C_+ (s_1, s_2), of $C_\times$ (m_1, m_2) and $C_=$ (e_1, e_2): $\mathcal{A} = \{s_1, s_2, m_1, m_2, e_1, e_2\}$.
- *out*: the output of the target algorithm.
- $\mathcal{E}$: union of inputs and components. $\mathcal{E} = \mathcal{I} \cup \mathcal{C}$.
- $l(x) : (\mathcal{E} \cup \mathcal{A} \cup \{out\}) \to \{1, 2, \ldots, |\mathcal{E}|\}$. Function which resolves the missing correspondences. The value of $l(x)$ for $x \in \mathcal{I}$ gives the line number where input x is read and stored in a local variable $o_{l(x)}$. The value of $l(x)$ for $x \in \mathcal{C}$ gives the line number where component x is used, and where the result of the operation is stored in a local variable $o_{l(x)}$. The value of $l(x)$ for $x \in \mathcal{A}$ gives the line number where

the local variable $o_{l(x)}$ (which serves as operand x) is declared. Finally, $l(out)$ gives the line number where the local variable $o_{l(x)}$ (which serves as the return value of the target algorithm) is declared. For instance, based on Demo B.2.1(b), we have $l(q) = 2$, $l(C_+) = 6$, $l(s_1) = 4$ (the left operand of C_+), and $l(out) = 2$.

Here are the syntaxis constraints:

- First of all, we make sure that $l(x)$ is defined in a consistent way. There cannot be two mappings of $l(x)$ to the same line number for the inputs and the components:

$$\Phi_{cons}(\mathcal{E}) = \forall e_1, e_2 \in \mathcal{E} : e_1 \neq e_2 \Rightarrow l(e_1) \neq l(e_2)$$

- As said before, the first $|\mathcal{I}|$ lines of the algorithm are intended to read the inputs:

$$\Phi_{input}(\mathcal{I}) = \forall i \in \mathcal{I} : 1 \leq l(i) \leq |\mathcal{I}|$$

- Next, the following lines are intended to operate with the components:

$$\Phi_{components}(\mathcal{C}) = \forall c \in \mathcal{C} : |\mathcal{I}| + 1 \leq l(c) \leq |\mathcal{E}|$$

- We ignore what is the correspondence of the operands of each component and *out*, so we have:

$$\Phi_{bound}(\mathcal{A}, out) = \forall e \in (\mathcal{A} \cup \{out\}) : 1 \leq l(a) \leq |\mathcal{E}|$$

- What we do know about the operands is that they can only refer to already declared local variables in the target algorithm. This can be expressed in the following *acyclicity constraint*:

$$\Phi_{acyc}(\mathcal{C}, \mathcal{A}) = \forall c \in \mathcal{C}, \forall a \in \mathcal{A} : (a \in operandsOf(c)) \Rightarrow l(a) < l(c)$$

- There is also another restriction for the operands: type checking. For example, if we are using a logic component C_{and} and an arithmetic component C_+, we cannot assign s_1 to the result returned by C_{and} because C_+ expects s_1 to be a number. Then, the idea is to make the operands reference only variables of the same type:

$$\Phi_{types}(\mathcal{C}, \mathcal{A}) : \forall c \in \mathcal{C}, \forall a \in \mathcal{A} : typeOf(c) \neq typeOf(a) \Rightarrow l(c) \neq l(a)$$

  Here, the function *typeOf* will answer: for a component, its return type; for an input, operand or *out*, its own type.

Then, the complete syntaxis encoding can be represented in a single *well-formed constraint*:

$$\Phi_{well-formed}(\mathcal{I}, \mathcal{C}, \mathcal{A}, out) = \exists l : \Phi_{cons}(\mathcal{I} \cup \mathcal{C}) \wedge \Phi_{input}(\mathcal{I}) \wedge$$
$$\Phi_{components}(\mathcal{C}) \wedge \Phi_{bound}(\mathcal{A}, out) \wedge \Phi_{acyc}(\mathcal{C}, \mathcal{A}) \wedge \Phi_{types}(\mathcal{C}, \mathcal{A})$$

B.2.2 Semantics Encoding

Here, we focus on the semantics of the algorithm. The main concern is to have an algorithm compliant with the specification. That is, for every $(I, O) \in \mathcal{V}$, the output of the target algorithm when it is evaluated with input I should be equal to O.

Let the following definition hold:

- $eval(x) : (\mathcal{E} \cup \mathcal{A}) \rightarrow Value$: function which permits to evaluate inputs, components and operands. For instance, in Demo B.2.1(c), if the input-output pair is $(\{p = 1, q = 2, n = 3\}, true)$ we have: $eval(q) = 2$, $eval(s_1) = eval(q) = 2$ (left operand of C_+), $eval(s_2) = eval(p) = 1$ (right operand of C_+) and $eval(C_+) = eval(s_1) + eval(s_2) = 3$.

Here are the semantics constrains:

- How to obtain the value of each input I of an (I, O) pair is straightforward:

$$\Phi_{values}(I) = \forall i \in I : eval(i) = i$$

- For each component, the constraint differs according to its expected behavior. In the running example we have one constraint for C_+, another one for $C_\times$ and another one for $C_=$:

$$\Phi_{lib}(\{C_+, C_\times, C_=\}) = (eval(C_+) = eval(s_1) + eval(s_2)) \wedge$$
$$(eval(C_\times) = eval(m_1) \times eval(m_2)) \wedge (eval(C_=) = eval(e_1) == eval(e_2))$$

- We want to be able to compose different components, so that the result of one component operation can be "tunneled out" as an operand of another component. For instance, in Demo B.2.1(c), we have e_1 (the left operand of $C_=$) linked to the local variable `o4` (that is, $l(e_1) = 4$). In turn, $l(C_+) = 4$ because C_+ is used in line number 4. As a consequence, the operand e_1 is really taking the value of the result of the C_+ operation. We express this relation in a *connectivity constraint*:

$$\Phi_{conn}(\mathcal{C}, \mathcal{A}) = \forall c \in \mathcal{C}, \forall a \in \mathcal{A} : l(a) = l(c) \Rightarrow eval(a) = eval(c)$$

- Finally, we make sure that the value referenced by *out* is equal to O of an (I, O) pair:

$$\Phi_{target}(O, out) = (eval(out) = O)$$

Then, the complete semantics encoding can be represented in a single *specification constraint*:

$$\Phi_{speficiation}(\mathcal{V}, \mathcal{C}, out) = \forall (I, O) \in \mathcal{V} : \exists eval :$$
$$\Phi_{values}(I) \wedge \Phi_{lib}(\mathcal{C}) \wedge \Phi_{conn}(\mathcal{C}, \mathcal{A}) \wedge \Phi_{target}(O, out)$$

B.2.3 Synthesis Constraint

Putting it all together, the complete SMT encoding can be formulated in a single *synthesis constraint*:

$$\Phi_{synth}(\mathcal{V}, \mathcal{I}, \mathcal{C}, \mathcal{A}, out) = \Phi_{well-formed}(\mathcal{I}, \mathcal{C}, \mathcal{A}, out) \wedge \Phi_{speficiation}(\mathcal{V}, \mathcal{C}, out)$$

The synthesis constraint says that there exists a function $l(x)$ that lets us build a well-formed algorithm which receives the parameters in $\mathcal{I}$ and operates with the components in $\mathcal{C}$ to output a value out; and, for every input-output pair $(I, O) \in \mathcal{V}$ the evaluation of out for input I is equal to O.

B.3 Decoding an Algorithm

The solution of the SMT problem gives the assignments of $l(x)$ for each input, component, component operands and output. With these values, the procedure to decode the solution of the SMT problem into an algorithm is straightforward. As shown in Demo B.2.1(a), in the first $|\mathcal{I}|$ lines we read input, and in the following $|\mathcal{C}|$ lines we operate with the components. The order of every element is dictated by $l(x)$.

We illustrate the procedure to build the algorithm with the running example of Subsection 4.5.1. Let us suppose the SMT solution is the one shown in Table B.3.1, which yields the same algorithm shown in Demo B.2.1(c). The table shows the $l(x)$ assignments as well as its corresponding local variables. Also, in the last row, we show a string representation of inputs and components. The string representation of each element lets us ultimately translate the line assignment with a source code snippet.

$l(p)$	$l(q)$	$l(n)$	$l(C_+)$	$l(s_1)$	$l(s_2)$	$l(C_\times)$	$l(m_1)$	$l(m_2)$	$l(C_=)$	$l(e_1)$	$l(e_2)$	$l(out)$
1	2	3	4	2	1	5	3	4	6	4	3	6
o1	o2	o3	o4	o2	o1	o5	o3	o4	o6	o4	o3	o6
I.p	I.q	I.n	o2+o1			o3×o4			o4==o3			

Table B.3.1. Assignments for $l(x)$ to arrange the algorithm of Demo B.2.1(c).

```
Ψ(I):
    o1 := _
    o2 := _
    o3 := _
    o4 := _
    o5 := _
    o6 := _
    return out
```

(a)

```
Ψ(I):
    o1 := I.p
    o2 := I.q
    o3 := I.n
    o4 := _
    o5 := _
    o6 := _
    return out
```

(b)

```
Ψ(I):
    o1 := I.p
    o2 := I.q
    o3 := I.n
    o4 := o2 + o1
    o5 := o3 × o4
    o6 := o4 == o3
    return out
```

(c)

```
Ψ(I):
    o1 := I.p
    o2 := I.q
    o3 := I.n
    o4 := o2 + o1
    o5 := o3 × o4
    o6 := o4 == o3
    return o6
```

(d)

Demonstration B.3.1

We begin by writing the outline of the algorithm (Demo B.3.1(a)). We have three inputs (p, q and n) and three components (C_+, $C_\times$ and $C_=$). Therefore, we leave 6 lines, plus another one for the return statement.

Next, we fill the lines for reading the input variables (Demo B.3.1(b)). For each input we look in the table its line assignment. For input p, the assignment $l(p)$ is 1. Then, we should read input p in the first line and assign it to `o1`. We do the same for the rest of the input variables.

Next, we fill the lines for operating with the components (Demo B.3.1(c)). For each component we look in the table its own line assignment, as well as those for its operands. For component C_+ the assignment $l(C_+)$ is 4. Then, we use this component in line 4, storing its value in `o4`. The operation is done between s_1 and s_2. The line assignment $l(s_1)$ is 2. This means that the local variable declared in line 2 (`o2`) acts as the left operand. The right operand (s_2) is `o1`, because the assignment $l(s_2)$ is 1. Then, the sum operation is performed with `o2` and `o1`. We do the same for the rest of the components and its operands.

Finally, we assign the corresponding local variable for the return vale (Demo B.3.1(d)). The table shows the assignment $l(out)$ is 6. Then, we write the return statement with `o6`.

This way, we are able to decode the solution of the SMT problem into an algorithm. Note that with component-based synthesis, the synthesised algorithm has the important characteristic of being correct by construction, because it derives from a solution of a logic problem.

Lastly, we mention that from the same solution of the SMT problem we can synthesise an expression instead of an algorithm of imperative programming fashion. We do it by backwards traversal of the returned value until every intermediate local variable is replaced with a component or an input.

$$\texttt{o6} \rightsquigarrow \texttt{(o4 == o3)} \rightsquigarrow \texttt{((o2 + o1) == o3)} \rightsquigarrow \texttt{((I.q + o1) == o3)} \rightsquigarrow$$

$$\texttt{((I.q + I.p) == o3)} \rightsquigarrow \texttt{((I.q + I.p) == I.n)}$$

C. IMPLEMENTATION ESSENTIALS

Throughout this thesis, we exhibit how Automatic Software Repair employs source code analysis and modification. When modifying source code automatically, avoiding compilation errors is a must because we want the modified code to be executable. Once compiled, we have to make sure we execute the modified source code instead of the original. All these tasks require dealing with program logic complexity, language syntaxis and the programming language metamodel. In this section, we briefly introduce the Java Virtual Machine's (JVM) architecture and we explain how to perform the "source code replacement" to run the modified code in the implementation of the present work.

C.1 Java Terminology

We first introduce some Java specific terminology taken from [30].

Package

It is a grouping of related types providing access protection and name space management. Programmers bundle classes and interfaces in a package for several reasons: *a*) to easily determine that types of same package are related; *b*) to create a new namespace to avoid conflicts with the type names of other packages; *c*) to allow types within the package to have unrestricted access to one another yet still restrict access for types outside the package.

Package Member

The types that comprise a package. To use a `public` package member from outside its package, one can do one of the following: *a*) refer to the member by its fully qualified name; *b*) import the package member; *c*) import the member's entire package.

Qualified Name

The qualified name of a class includes the package name. That is, the qualified name of the `Rectangle` class in the `graphics` package is `graphics.Rectangle`, and the qualified name of the `Rectangle` class in the `java.awt` package is `java.awt.Rectangle`.

Many implementations of the Java platform rely on hierarchical file systems to manage source and class files, although the Java language specification does not require this. The qualified name of the package member and the path to the file are parallel. If the qualified name is `java.awt.Rectangle`, then the path to the source file of the class should be `sourcepath/java/awt/Rectangle.java`. Here, `sourcepath` is a placeholder for the path of the root folder which contains all the `.java` files of the project.

Classfile

When a source file is compiled, the compiler creates a different output file for each type defined in it. The base name of the output file is the name of the type, and the extension `.class` (hence, the classfile). The compiled file for `java.awt.Rectangle` will be located in `classpath/java/awt/Rectangle.class`. Here, `classpath` is a placeholder for the path of the root folder which contains all the `.class` files of the project.

Classpath

Like the `.java` source files, the compiled `.class` files should be in a series of directories that also reflect the package name. However, paths to `.class` files do not have to be the same as the `.java` source files. As a convention, most programmers arrange the `.java` and `.class` root directories in two different locations:

```
sourcepath/java/awt/Rectangle.java    classpath/java/awt/Rectangle.class
```

By doing this, one can give the `classpath` directory to other programmers without revealing the sources. Another reason for maintaining a directory structure is because it helps the compiler and the JVM find all the types a program uses. For example, if `/usr/home/app/bin/` is the classpath, to load the `Rectangle` class, the compiler and JVM look for `.class` files recursively from the classpath following the package name `java.awt`. That is, they expect to find the `Rectangle` classfile in:

```
/usr/home/app/bin/java/awt/Rectangle.class
```

Optionally, a class path may include several paths (separated by a colon in Linux):

```
/usr/home/app/bin:/usr/home/dependency/bin
```

It can also include a `.jar` file (from where the retrieval of a class file or resource is done in a hierarchical way as well):

```
/usr/home/app/bin:/usr/home/app/external/ext.jar
```

Finally, it is important to note that if a class path includes the same package more than once, only the first location of the path is used by the JVM. This could be troublesome if you have the same package referenced twice in the class path with different versions in each reference and the application keeps executing the wrong version because it appears before the desired one in the class path. Moreoever, this could introduce runtime errors as `NoSuchMethodError` (when the target class is found but not the referenced member). In these cases, special attention must be paid to the class path configuration.

C.2 Java Virtual Machine

To understand the JVM one should first be aware that the term "Java Virtual Machine" can reference three different things: the abstract specification, a concrete implementation, or a runtime instance. The abstract specification is a concept, described in detail in [29]. Concrete implementations, which exist on many platforms and come from many vendors, are either all software or a combination of hardware and software. A runtime instance hosts a single running Java application.

A runtime instance of the JVM has a clear mission in life: to run one Java application. When a Java application starts, a runtime instance is born. When the application completes, the instance dies. If we start two Java applications at the same time, on the same computer, using the same concrete implementation, we would get two JVM instances. Each Java application runs inside its own JVM.

JVM Lifespan

Inside the JVM, threads appear in two flavors: daemon and non-daemon. A daemon thread is ordinarily a thread used by the virtual machine itself, such as a thread that performs garbage collection. The application, however, can mark any threads it creates as daemon threads. The initial thread of an application –the one that begins at `main()`– is a non-daemon thread.

A Java application continues to execute ("the virtual machine instance continues to live") as long as any non-daemon thread keeps running. When all non-daemon threads of a Java application terminate, the virtual machine instance exits. Alternatively, if allowed by the security manager, the application can explicitly cause its own demise by invoking the `exit()` method of class `Runtime` or `System`.

When a new thread is created it inherits the daemon status of its parent. The method `setDaemon()` can be used to change the Thread daemon properties, but only before the start of the thread.

C.3 Runtime Data Areas

When a JVM runs a program, it needs memory to store many things, including bytecodes and other information it extracts from loaded class files, objects the program instantiates, parameters to methods, return values, local variables, and intermediate results of computations. The JVM organizes the memory it needs to execute a program into several runtime data areas. In other words, the runtime data areas are a mechanism to organize memory managed by the JVM.

C.3.1 Thread Memory

As each new thread comes into existence, it gets its own *pc register* (program counter) and *Java stack*. If the thread is executing a Java method (not a native method), the value of the pc register indicates the next instruction to execute. A thread's Java stack stores the state of Java (not native) method invocations for the thread. The state of a Java method invocation includes its local variables, the parameters with which it was invoked, its return value (if any), and intermediate calculations. The state of native method invocations is stored in an implementation-dependent way in native method stacks, as well as possibly in registers or other implementation-dependent memory areas.

C.3.2 Shared Memory

Each instance of the JVM has one *method area* and one *heap*. These areas are shared by all threads running inside the virtual machine.

- Method Area: when the virtual machine loads a class file, it parses information about a type from the binary data contained in the class file. It places this type information into the method area. All threads share the same method area, so access to the method area's data structures must be designed to be thread-safe. If two threads are attempting to find a class named `Lava`, for example, and `Lava` has not yet been loaded, only one thread should be allowed to load it while the other one waits.

- Heap: as the program runs, the virtual machine places all objects the program instantiates onto the heap. The virtual machine itself is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application. Usually, a JVM implementation uses a garbage collector to manage the heap.

When the JVM loads a type, it uses a class loader to locate the appropriate class file. The class loader reads in the class file –a linear stream of binary data– and passes it to the virtual machine. The virtual machine extracts information about the type from the binary data and stores the information in the method area. Memory for class (`static`) variables declared in the class is also taken from the method area. For each type it loads, the JVM must store the following information in the method area:

- The fully qualified name of the type.
- The fully qualified name of the type's direct superclass (unless the type is an interface or class `java.lang.Object`, neither of which have a superclass).
- Whether or not the type is a class or an interface.
- The type's modifiers (some subset of `public`, `abstract`, `final`).
- An ordered list of the fully qualified names of any direct superinterfaces.
- The constant pool for the type (a constant pool is an ordered set of constants used by the type).
- Field information (for each field the following information is stored: name, type and modifier –subset of `public`, `private`, `protected`, `static`, `final`, `volatile`, `transient`).
- Method information (for each method the following information is stored: name, return type, number and type of parameters and modifiers –subset of `public`, `private`, `protected`, `static`, `final`, `synchronized`, `native`, `abstract`). When the method is not abstract nor native, these are also stored: method's byte codes, size of operands, stack and exception table.
- All class (`static`) variables declared in the type, except constants. A constant is a class variable with `final` modifier.
- A reference to the type's `ClassLoader`. For each type it loads, a JVM must keep track of whether or not the type was loaded via the bootstrap class loader or a user-defined class loader. For those types loaded via a user-defined class loader, the virtual machine must store a reference to the user-defined class loader that loaded the type. This information is stored as part of the type's data in the method area. The virtual machine uses this information during dynamic linking. When one type refers to another type, the virtual machine requests the referenced type from the same class loader that loaded the referencing type. Most implementations of the JVM don't wait until all classes used by the application are loaded before it begins executing `main()`; they load classes only as they are needed.
- A reference to an instance of class `Class`. An instance of class `java.lang.Class` is created by the JVM for every type it loads. The virtual machine must in some

way associate a reference to the `Class` instance for a type with the type's data in the method area. The static method `forName()` in class `Class`, allows access to the `Class` instance for any loaded class.

C.4 Class Loader Subsystem

Three main components constitute the JVM: runtime data areas, the execution engine (responsible for executing the instructions contained in the methods of loaded classes) and a class loader subsystem. The class loader subsystem is a mechanism for loading types (classes and interfaces) given fully qualified names. We now proceed by explaining this component in detail.

C.4.1 Introduction

Applications written in statically compiled programming languages, such as C and C++, are compiled into native, machine-specific instructions and saved as an executable file. The process of combining the code into an executable native code is called *linking*: the merging of separately compiled code with shared library code to create an executable application. This is different in dynamically compiled programming languages such as Java. In Java, the `.class` files generated by the Java compiler remain as-is until loaded into the JVM. In other words, the linking process is performed by the JVM at runtime. Class loaders are responsible for loading the classes into the JVM.

The smallest unit of execution that gets loaded by a class loader is the Java `.class` file. A class file contains the binary representation of a Java class, which has the executable bytecodes and references to other classes used by that class, including references to classes in the Java API. Stated simply, a `ClassLoader` locates the bytecodes for a Java class that needs to be loaded, reads the bytecodes, and creates an instance of the class `java.lang.Class`. This makes the class available to the JVM for execution.

Initially, when a JVM starts up, nothing is loaded into it. The first class to run, the entry point into the application, is the one with the static `main()` method. Next, other classes and interfaces are loaded as they get referenced in the bytecode being executed. The JVM thus exhibits *lazy loading*, loading classes only when required. This way, at start-up, the JVM does not need to know the classes that would get loaded during runtime. Lazy loading plays a key role in providing dynamic extensibility to the Java platform.

The class loader subsystem is responsible for more than just locating and importing the binary data for classes. It must also verify the correctness of imported classes, allocate and initialize memory for class variables, and assist in the resolution of symbolic references. These activities are performed in a strict order:

1. Loading: finding and importing the binary data for a type.
2. Linking: performing verification, preparation, and (optionally) resolution.
    (a) Verification: ensuring the correctness of the imported type.
    (b) Preparation: allocating memory for class variables and initializing the memory to default values.

int	long	short	char	byte	boolean	float	double	reference
0	0L	(short) 0	'\u0000'	(byte) 0	false	0.0f	0.0d	null

Table C.4.1. Default values for each type.

(c) Resolution: transforming symbolic references from the type into direct references.

3. Initialization: invoking Java code that initializes class variables to their proper starting values.

Every JVM implementation has a *bootstrap class loader*, which knows how to load trusted classes, including the classes of the Java API. The JVM specification does not define how the bootstrap loader should locate classes. That is another decision the specification leaves to implementation designers. Given a fully qualified type name, the bootstrap class loader must in some way attempt to produce the data that defines the type.

Although user-defined class loaders themselves are part of the Java application, four of the methods in class `java.lang.ClassLoader` are gateways into the JVM:

```
protected final void resolveClass(Class c);
protected final Class findSystemClass(String name);
protected final Class defineClass(String name,byte[] data,int offset,int length);
protected final Class defineClass(String name,byte[] data,int offset,int length,
                                  ProtectionDomain protectionDomain);
```

Any JVM implementation must take care to connect these methods of class `ClassLoader` to the internal class loader subsystem.

C.4.2 Class Loader Terminology

Class loaders in Java terminology can be classified in two kinds. A class loader `L` may create class `C` by defining it directly or by delegating to another class loader:

- If `L` creates `C` directly, we say that `L` *defines* `C` or, equivalently, that `L` is the *defining loader* of `C`.
- When one class loader delegates to another class loader, the loader that initiates the loading is not necessarily the same loader that completes the loading and defines the class. If `L` creates `C`, either by defining it directly or by delegation, we say that `L` *initiates* loading of `C` or, equivalently, that `L` is an *initiating loader* of `C`.

C.4.3 Class Creation

Class or interface creation of class `C` is triggered when another class or interface `D` references `C` through its runtime constant pool. The JVM uses one of three procedures to create class or interface `C`:

- If the name of class `C` denotes an array class, the class is created directly by the JVM, not by a class loader. However, the defining class loader of `D` is used in the process of creating `C`.
- Otherwise, if `D` was defined by the bootstrap class loader, then the bootstrap class loader initiates loading of `C`.

- Otherwise, if `D` was defined by a user-defined class loader, then that same user-defined class loader initiates loading of `C`.

A given `ClassLoader` can load a given class only once. There is no need to unload a class. When the objects using it are no longer referenced, the class object itself, along with the code, are garbage collected.

We now illustrate how a class is created with an example. Suppose a new instance of class `Purchase` is created inside a method of class `Client`. The JVM looks up in `Client`'s constant pool for the `Purchase` entry. That entry is a *symbolic reference* to `Purchase`. With that symbolic reference, the JVM looks up in the method area if `Purchase` was loaded. As it discovers that the class was not loaded, it proceeds to find and read in file `Purchase.class`, extracting the definition of the class and importing the binary data into the method area.

To do this, the virtual machine uses the defining class loader of the referencing type to load the referenced type. Let us assume `Client` was defined by a user-defined class loader. When the virtual machine resolves the reference to `Purchase`, it checks to see if `Purchase` has been loaded into the namespace to which `Client` belongs. That is, it checks to see if the class loader that defined `Client` has previously loaded a type named `Purchase`. If not, the virtual machine requests `Purchase` from the same class loader that defined `Client`. This is true even if a class named `Purchase` had previously been loaded into a different namespace. This is because, at run time, a class or interface is determined not by its name alone, but by a pair: its qualified name and its defining class loader. Each such class or interface belongs to a single *runtime package*. The runtime package of a class or interface is determined both by the package name and defining class loader of the class or interface. As a consequence, from the JVM's point of view, the same classes, loaded by different class loaders, are considered distinct classes.

Once the class is loaded, the JVM replaces the symbolic reference in `Client`'s constant pool with a pointer to the class data for `Purchase`. If the JVM ever needs to use `Purchase`'s constant pool entry once again, it won't have to reload `Purchase`. This process of replacing symbolic references with direct references is called *constant pool resolution*.

Finally, the virtual machine is ready to actually allocate memory for a new `Purchase` object. Once again, the virtual machine consults the information stored in the method area. It uses the pointer (which was just put into `Client`'s constant pool entry) to the `Purchase` data (which was just imported into the method area) to find out how much heap space is required by a `Purchase` object.

(Note: in the previous explanation, the qualified name `some.package.Purchase` is actually used, instead of the simple name `Purchase`).

C.5 User-defined Class Loaders

The Java runtime can be customized by implementing a custom `ClassLoader` in a program. Developing class loaders is an inherently dangerous undertaking as this can cause no end of security troubles. For this reason, the Java 2 platform has added useful classes to the core APIs in order to make developing and using class loaders. For example, the `java.security.SecureClassLoader` class extends the `ClassLoader` with additional support for defining classes with an associated code source and permissions which are retrieved by the system policy by default.

The `java.net.URLClassLoader` class, which is a subclass of `SecureClassloader`, can be used to load classes and resources from a search path of URLs referring to directories and JAR files. The URLs will be searched in the order specified for classes and resources after first searching in the parent class loader.

The `URLClassLoader` can be used to develop an application capable of loading classes and resources from remote servers. The first step is to define the URLs where the loader will search for class files. Any URL that ends with a slash ('/') is assumed to refer to a directory, otherwise the URL is assumed to refer to a JAR file which will be opened as needed. Once an instance of the `URLClassLoader` is constructed, the `loadClass()` method of the `ClassLoader` class is used to load the class with the specified qualified name.

C.5.1 Implementing a User-defined Class Loader with JVM 1.1

Prior to JVM 1.2, the `loadClass()` method of `java.lang.ClassLoader` was abstract. To create a user-defined class loader, one way was to subclass `ClassLoader` and override `loadClass()`. A typical implementation is shown in Demo C.5.1.

```java
public synchronized Class loadClass(String className, boolean resolve)
  throws ClassNotFoundException {
    Class result;
    byte classData[];
    result = findLoadedClass(className);
    if (result != null) {
        return result;
    }
    try {
        result = super.findSystemClass(className);
        return result;
    }
    catch (ClassNotFoundException e) {}
    if (className.startsWith("java.")) {
        throw new ClassNotFoundException();
    }
    classData = getTypeFromClassPath(className);
    if (classData == null) {
        throw new ClassNotFoundException();
    }
    result = defineClass(className, classData, 0, classData.length);
    if (result == null) {
        throw new ClassFormatError();
    }
    if (resolve) {
        resolveClass(result);
    }
    return result;
}
```

Demonstration C.5.1

The `loadClass()` method begins by checking to see if the requested type has already been loaded by this class loader. It does this by invoking `findLoadedClass()`, an instance method in `ClassLoader`, passing in the fully qualified name of the requested type as a parameter. If this class loader has already been marked as an initiating class loader of a type with the requested fully qualified name, `findLoadedClass()` will return the Class instance representing the type.

If the requested type has not been loaded into its name space, it next passes the name of the requested type to `findSystemClass()`. In 1.1, when this method is invoked, the primordial class loader attempts to load the type. In 1.2, the system class loader attempts to load the type. If the load is successful, `findSystemClass()` returns the `Class` instance representing the type, and `loadClass()` returns that same `Class` instance. Otherwise, a `ClassNotFoundException` is thrown.

The `loadClass()` method next checks to make sure the requested class is not part of the java package. This check prevents members of the standard java packages (`java.lang`, `java.io`, etc.) from being loaded by anything but the bootstrap class loader. Without this check, any type that declared itself to be part of the Java API (or any other restricted packages) could be granted access to other package-visible members. In JVM 1.2, to prevent this, the notion of *runtime package* is introduced. This way, only classes from the Java API defined by the boostrap class loader are granted access to package-visible members.

If the type name does not begin with "`java.`", the `loadClass()` method next invokes the method `getTypeFromClassPath()`, which attempts to import the binary data in a custom way defined by the user-defined `ClassLoader`. Typically, the method looks for a file with the type name plus a `.class` extension in the class path directory passed to this custom `ClassLoader`. If the `getTypeFromClassPath()` method is unable to find the file, it returns a null result and the `loadClass()` method ends by throwing a `ClassNotFoundException`. Otherwise, `loadClass()` invokes `defineClass()`, passing the byte array returned by method `getTypeFromClassPath()`.

The `defineClass()` method completes the loading process. It parses the binary data into internal data structures and creates a `Class` instance. It also makes sure all the type's supertypes are loaded. It does this by invoking `loadClass()` on this user-defined class loader for each direct superclass and superinterface, and recursively applies the resolution process on all supertypes in the hierarchy.

If `defineClass()` is successful, the `loadClass()` method checks to see if `resolve` is set to true. If so, it invokes `resolveClass()`, passing the `Class` instance returned by `defineClass()`. The `resolveClass()` method links the class and, finally, `loadClass()` returns the newly created `Class` instance.

C.5.2 Impementing a User-defined Class Loader with JVM 1.2

In JVM 1.2, a bootstrap class loader that is built into the JVM is responsible for loading the classes of the Java runtime. This class loader only loads classes that are found in the boot classpath (`rt.jar`, trusted classes of the Java runtime), and since these are trusted classes, the validation process is not performed as for untrusted classes.

Each `ClassLoader` except the bootstrap `ClassLoader` has a parent `ClassLoader`, conceptually forming a treelike structure of `ClassLoaders` (Fig. C.5.1). Since there is more than one class loader, they are represented in a tree whose root is the bootstrap class loader. Each class loader has a reference to its parent class loader. When a class loader is asked to load a class, it consults its parent class loader before attempting to load it itself. The parent in turn consults its parent, and so on. It is only after all the ancestor class loaders cannot find the class that the current class loader gets involved. In other words, a *delegation model* is used.

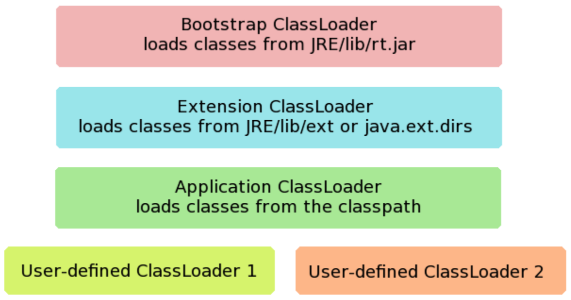
Bootstrap ClassLoader
loads classes from JRE/lib/rt.jar
Extension ClassLoader
loads classes from JRE/lib/ext or java.ext.dirs
Application ClassLoader
loads classes from the classpath
User-defined ClassLoader 1
User-defined ClassLoader 2

Figure C.5.1. Class loaders treelike structure in JVM 1.2.

The immediate child of the bootstrap `ClassLoader` is the extension `ClassLoader` responsible for loading classes from standard extension APIs. Then, it follows the application `ClassLoader`, which loads classes from the classpath. From that point, the class loading tree can become arbitrarily complicated, depending on the class loading hierarchy when the application has started many different class loaders.

The `java.lang.ClassLoader` is an abstract class that can be subclassed by applications that need to extend the way in which the JVM dynamically loads classes. Constructors in `java.lang.ClassLoader` (and its subclasses) allow to specify a parent when instantiating a new class loader. If no parent class loader is specifed, the virtual machine's system class loader is assigned as the default parent. This way, every instance of `ClassLoader` has an associated parent class loader. This is done to preserve the delegation model to search for classes and resources.

The concrete implementation of `loadClass()` included in 1.2 supports the delegation model and makes it easier and less error prone to create a user-defined class loader. To create a user-defined class loader, rather than overriding `loadClass()`, the `findClass()` method should be overriden –a method with a much simpler contract than `loadClass()`.

The `loadClass()` method described in Demo C.5.1, which was originally designed for JVM 1.1, would still work in 1.2. Although a concrete default implementation of `loadClass()` was added to `java.lang.ClassLoader` in 1.2, this concrete method can still be overridden in subclasses. Because the contract of `loadClass()` did not change from 1.1 to 1.2, legacy user-defined class loaders that override `loadClass()` should still work as expected in 1.2.

Contract of `loadClass()`

Given the fully qualified name of the type to find, the `loadClass()` method should in some way attempt to locate or produce an array of bytes, purportedly in the Java class file format, that define the type. If `loadClass()` is unable to locate or produce the bytes, it should throw `ClassNotFoundException.` Otherwise, `loadClass()` should pass the array of bytes to one of the `defineClass()` methods declared in class `ClassLoader`. By passing the byte array to `defineClass()`, `loadClass()` asks the virtual machine to import the type represented by the passed byte array into the namespace of this user-defined `ClassLoader`.

When `loadClass()` calls `defineClass()` in 1.2, it can also specify a protection domain with which the type data should be associated. When the `loadClass()` method of a class loader successfully loads a type, it returns a `java.lang.Class` object to represent the newly loaded type.

The concrete implementation of `loadClass()` from class `java.lang.ClassLoader` fullfills the contract using four steps:

1. Invoke `findLoadedClass()` to see if the requested type has already been loaded into this class loader's namespace. If so, return the `Class` instance for that already-loaded type.
2. Otherwise, delegate to the parent loader. If the parent returns a `Class` instance, return that same `Class` instance.
3. Otherwise, invoke `findClass()`, which should attempt to locate or produce an array of bytes, purportedly in the Java class file format, that define the desired type. If successful, `findClass()` should pass those bytes to `defineClass()`, which will attempt to import the type and return a `Class` instance. If `findClass()` returns a `Class` instance, `loadClass()` returns that same `Class` instance.
4. Otherwise, `findClass()` ends abruptly with some exception, and `loadClass()` ends abruptly with the same exception.

Contract of `findClass()`

The method `findClass()` accepts the fully qualified name of a desired type as its only parameter. First, it attempts to locate or produce an array of bytes, purportedly in the Java class file format, that define the type of the requested name. If `findClass()` is unable to locate or produce the array of bytes, it ends abruptly with `ClassNotFoundException`. Otherwise, `findClass()` invokes `defineClass()`, passing in the requested name, the array of bytes and, optionally, a `ProtectionDomain` object with which the type should be associated. If `defineClass()` returns a `Class` instance for the type, `findClass()` simply returns that same `Class` instance to its caller. Otherwise, `defineClass()` ends abruptly with some exception, and `findClass()` ends abruptly with the same exception.

The same implementation as the one shown in Demo C.5.1 can be achieved with the concrete implementation of `loadClass()` by overriding `findClass()` as shown in Demo C.5.2. Here again, to find the class definitions, the method `getTypeFromClassPath()` consults the directories and jar files in the classpath in search of the requested class.

```java
protected Class findClass(String className) throws ClassNotFoundException {
    byte[] classData = getTypeFromClassPath(className);
    if (classData == null) {
        throw new ClassNotFoundException();
    }
    return defineClass(className, classData, 0, classData.length);
}
```

Demonstration C.5.2

C.5.3 Usage of User-defined Class Loaders

There are many reasons why it is desirable to create customized class loaders. They let us deal with byte code served from a database, local or remote. They let us create a sandbox to control exactly what classes are loadable. They let us dynamically change the classpath. They let us upgrade software (upgrade Java classes) in a running application without restarting it, a feature known as *hot deployment*. This feature is exploited in most Java-based application servers. A `ClassLoader` cannot reload a class that it has already loaded, but using a new instance of a `ClassLoader` will reload a class into a running program.

The way to use a specific `ClassLoader` to load classes is by using threads. Threads interact with the `ClassLoader` in one particular case. Each thread is assigned a specific `ClassLoader` known as the *context class loader*. This class loader is retrieved with the `getContextClassLoader()` method and set (before starting the thread) with the `setContextClassLoader()` method.

The context class loader is not used to load things in a general case, it is only used when it is explicitly called via the `getContextClassLoader()` method. The reason a context `ClassLoader` is needed is to be able to add classes to the application which are outside the application classpath (so they are unknown by the application class loader). If we set a custom `ClassLoader`, which knows how to find classes outside the application classpath, as the context `ClassLoader`, then we would have a way to load external classes in the thread. That is, the context `ClassLoader` is a *hook* to define certain classes outside the application classpath. This hook is unrelated to threading issues: the context `ClassLoader` can be set and reset at will in the application. The `Thread` class simply provides a convenient location to put this hook.

C.6 Source Code Replacement

Now that the inner *modus operandi* of the JVM is explained, we can explain how we manage to replace the original source code with the modified source code in our infinite loop repair program.

The typical framework strategy of Automatic Software Repair of a project is the following. First, we run the test suite. If all tests pass, no repair is needed. If some tests fail (or, in our case, hang), we modify the project source code (but not the test suite code, because it is the repair contract). To know whether the modifications represent a possible repair, we run the test suite again (the test suite should be able to run the modified source code). If all tests now pass, the repair was found. If not, the cycle starts over again until a repair is found.

The way to implement this in Java is as follows. The input of the repair tool is the classpath of the test suite, and the source path of the project source code, excluding the test suite source code. Whenever we modify some class' source code, we need to compile it and store the bytecodes in a classfile –during runtime. We can achieve this with an API provided by java in `javax.tools` package [39], [41]. Basically, the `javax.tools.JavaCompiler` class lets us create a `CompilationTask` which reads a `java.lang.CharSequence` (the string of the modified code) and which deposits its compiled bytecodes in an `java.io.OutputStream`. When the `CompilationTask` is finished, we can directly associate the qualified name of the modified class with the bytecodes generated by the `CompilationTask`.

Once the runtime compilation of the modified source code ends, we create a custom `ClassLoader` which uses a `java.util.Map` of `String` key type (qualified name of a class) and `byte[]` value type (the bytecodes of the compiled source code). Whenever the method `findClass()` is called upon this custom `ClassLoader`, it can look on the `Map` to see whether it has the requested qualified name bytecodes. If it does, it can directly call `defineClass()` with those bytecodes. If not, it calls its inherited implementation of the method ("`super.findClass()`"). In our implementation, this custom `ClassLoader` is called the `BytecodeClassLoader`.

Now, there is one last step needed for the source code replacement to actually work: being able to execute the modified code when running test suite. To do so, we create a `Thread` which, when started, instances a `org.junit.runner.JUnitCore` object. This object receives the collection of test classes and, when the `Thread` starts, it executes the test cases in each of them. The trick here is to set the `BytecodeClassLoader` as the `Thread`'s context class loader. The provided `BytecodeClassLoader` contains the bytecodes of the modified source code and also the classpath of the test suite. Then, if we explicity use this class loader to load the test classes (to be executed by the `JUnitCore` instance), the `BytecodeClassLoader` will be marked as the defining class loader of the test classes. Consequently, whenever a class of the project is referenced by a test class, this will be the class loader used to find that class. Because the `BytecodeClassLoader` only contains the bytecodes of the modified source code, the modified class will be defined in the thread when running the test suite.

REPAIR MANIFESTO

WE HOLD THESE TRUTHS TO BE SELF-EVIDENT

IF YOU CAN'T FIX IT, YOU DON'T OWN IT.

REPAIR IS BETTER THAN RECYCLING

Making our things last longer is both more efficient and more cost-effective than mining them for raw materials.

REPAIR SAVES YOU MONEY

Fixing things is often free, and usually cheaper than replacing them. Doing the repair yourself saves you money.

REPAIR TEACHES ENGINEERING

The best way to find out how something works is to take it apart.

REPAIR SAVES THE PLANET

Earth has limited resources. Eventually we will run out. The best way to be efficient is to reuse what we already have.

REPAIR CONNECTS PEOPLE AND THINGS | REPAIR IS WAR ON ENTROPY | REPAIR IS SUSTAINABLE

WE HAVE THE RIGHT:

- TO DEVICES THAT CAN BE OPENED
- TO REPAIR DOCUMENTATION FOR EVERYTHING
- TO REPAIR THINGS IN THE PRIVACY OF OUR OWN HOMES
- TO ERROR CODES & WIRING DIAGRAMS
- TO CHOOSE OUR OWN REPAIR TECHNICIAN
- TO NON-PROPRIETARY FASTENERS
- TO REMOVE 'DO NOT REMOVE' STICKERS
- TO REPLACE ANY & ALL CONSUMABLES OURSELVES
- TO TROUBLESHOOTING INSTRUCTIONS & FLOWCHARTS
- TO AVAILABLE, REASONABLY-PRICED SERVICE PARTS

BECAUSE REPAIR

- IS INDEPENDENCE SAVES MONEY & RESOURCES
- REQUIRES CREATIVITY
- MAKES CONSUMERS INTO CONTRIBUTORS
- INSPIRES PRIDE IN OWNERSHIP