1-1-2020

Invariant-based online software anomaly detection and selective regression testing

Yizhen Chen
University at Albany, State University of New York, jonkiky@gmail.com

The University at Albany community has made this article openly available. Please share how this access benefits you.

Follow this and additional works at: https://scholarsarchive.library.albany.edu/legacy-etd

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
https://scholarsarchive.library.albany.edu/legacy-etd/2456

This Dissertation is brought to you for free and open access by the The Graduate School at Scholars Archive. It has been accepted for inclusion in Legacy Theses & Dissertations (2009 - 2024) by an authorized administrator of Scholars Archive.
Please see Terms of Use. For more information, please contact scholarsarchive@albany.edu.
IN Variant-based Online Software Anomaly Detection and Selective Regression Testing

by

Yizhen Chen

A Dissertation
Submitted to the University at Albany, State University of New York
in Partial Fulfillment of
the Requirements for the Degree of
Doctor of Philosophy

College of Engineering and Applied Sciences
Department of Computer Science
August 2020
ABSTRACT

Software has been extensively used in various domains to provide online services. With the growing popularity of these types of applications, the quality of the software has a great impact on many of our daily activities [1]. Reliable software executions that deliver expected outcomes are essential for quality services. Software is considered abnormal when its behavior deviates from what is expected at any point during its execution. When anomalous behavior propagates to an exit point of the software and produces an incorrect output or an unexpected termination of the execution, it is considered a software failure. An anomaly may or may not induce a failure, but when a failure occurs there must be at least one point in the program execution that is abnormal. Software anomalous behaviors can be caused by residual code defects or anomalies propagated from the execution context.

Online software anomaly detection investigates the abnormal behavior of a production software at any point of the runtime execution. Determination of online software anomaly requires a run-time validation oracle that should not affect the overall performance, which is generally difficult to achieve [2].

This dissertation presents a new approach that enables monitoring software online behavior with minimal execution overhead. We use program invariants as means for capturing persistent conditions of program properties. A program invariant holds the program’s property at a certain point of the program during its execution. Several studies have suggested using program invariants to investigate program behavior [3 4 5]. However, the number of program invariants for a large scale software is usually large, making it infeasible to monitor all invariants online. Thus, an effective selection of the invariants is essential. In this dissertation, we apply the algorithms derived from the sensor placement models to select a close to a minimum set of anomaly-revealing invariants to monitor online software behavior. Our empirical results show that we were able to detect an average of 76.5% of anomalies by using at most 5.5% of execution overhead.

The second part of this dissertation focuses on selective regression testing by
using program invariants. Most software systems are continuously evolving to better serve user needs. Consequently, maintenance needs to be frequently applied to a multi-version production software. The ongoing changes may take place in the core of the systems or in their supporting execution context. When a change is made to the software, to ensure that the changes do not adversely introduce regression faults causing previously working features to fail, regression testing, which re-runs the previous successfully executed test cases on the modified software, should be performed. Furthermore, when the execution context is changed regardless of whether the code is changed or not, regression testing also needs to be performed to check the compatibility within the execution context.

Re-executing all the test cases is a quick solution but it can be time consuming when new tests are continually added to the test suite. To minimize the maintenance cost and to prevent service disruption of the production software, an efficient and effective selection from the regression test suite to perform regression testing is essential. Meanwhile, most selective regression testing approaches [6, 7, 8, 9] focus on the changes made to the code only. When the execution context is changed or if there exists a dependency in the code via an external entity is changed, then these approaches may not be safe.

This dissertation presents a new regression test selection approach, that accounts for both changes in the code and in the execution context. We use program invariants before and after an invocation of a function to determine if the function is affected by the change and can potentially have regression faults. The preconditions and post-conditions preserve the properties required for, or resulting from, a correct execution of the function. After a change, these properties may no longer persist. Our approach selects the tests that execute the functions whose pre- and/or post-conditions are affected; in addition, all the tests that execute the modified code will be selected.

The results of our controlled experiments show that with an average of 16.9% of the test cases, our approach selected all the fault-revealing test cases. The results of three industrial case studies show that all the modification-revealing tests were selected with an average of 25.3% of the test cases. These results suggest that our
approach is effective for selecting fault-revealing test cases on both code and execution context changes.

Furthermore, when the time for regression testing is very limited and only few regression tests can be executed, a prioritization technique will help in the selection of most critical tests. Our approach applies a multi-objective algorithm that uses the coverage of affected and changed program invariants to prioritize the modification-revealing tests, so that the tests that are most likely to have regression faults will be executed first.

The results of our case studies show that with an average of 10% of the test cases, our approach detected all the regression faults and with an average of 50% of the test cases, all the modification-revealing test cases were selected. These results suggest that our approach is effective for early detection of regression faults.

Detecting online software anomalies and selecting fault-revealing regression tests are challenging. The approaches presented in this dissertation are promising. For the online software anomaly detection, our approach is effective and efficient; it selects fault-revealing invariants with high precision, high coverage of anomaly detection, and low execution overhead. For selective regression testing, our approach considers impact from changes in the program as well as in the execution context, and our multi-objective function selects an optimal subset of test cases that are likely fault-revealing within the given time constraint.
ACKNOWLEDGMENT

Foremost, I would like to thank my thesis advisor Prof. Mei-Hwa Chen for the continuous support of my Ph.D study and research. The door to Prof. Chen’s office was always open whenever I ran into a trouble or had a question about my research or writing. Thank for her patience, motivation, enthusiasm, and immense knowledge. She steered me in the right direction. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to express my very profound gratitude to rest of my thesis committee: Prof. Feng Chen and Prof. Jeong-Hyon Hwang for their encouragement, insightful comments, and challenging questions.

I would like to thank my fellow lab mates in software Engineer Research Group and Machine Learning Group: Ming Ying, Daren Liu, Karan Sheth, Chaudhari Ninad, Adil Alim for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last five years.

I would like to thank Professor Jianwei Zhang from Education Development, Vijaya Kokkili, Gavin Hogan, Santosh Iyer, David Cross, and Jonathan Lazzara at CommerceHub for their support on this project, and Jeanette Sperhac for helping us in using the CCR cluster and VIDIA platform for the experiments.

Last but not the least, I would like to thank my family: my parents for giving birth to me at the first place and supporting me spiritually throughout my life.
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td></td>
<td>ii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENT</td>
<td></td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td></td>
<td>ix</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 Anomaly Detection</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.1.2 Regression Test Selection</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>1.2 Program Invariants</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Overview</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1.2.2 The use of program invariants</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>1.2.3 Inferring program invariant</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>1.2.4 The types of program invariants</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>1.3 Online Software Anomaly Detection</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1.3.1 Background</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1.3.2 Motivation</td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1.3.3 Challenges</td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>1.4 Regression Test Selection</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>1.4.1 Background</td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>1.4.2 Motivation</td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>1.4.3 Challenges</td>
<td></td>
<td>16</td>
</tr>
<tr>
<td>2. ONLINE SOFTWARE ANOMALY DETECTION</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>2.1 Overview</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>2.2 Existing approaches and limitations</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>
3.5.1 Case Study I ......................................................... 97
3.5.2 Case Study II ......................................................... 100

4. THREAT TO VALIDITY ................................................ 105

5. CONCLUSIONS ......................................................... 106

APPENDICES

A. Appendix 1: List of the types of program invariants .............. 109
# LIST OF TABLES

1. **Program invariants of withdraw function** ........................................ 5

2. **Examples of the types of the program invariants** .......................... 9

3. **Types of abnormal behaviours** ......................................................... 25

4. **List of program invariant of domain error example** .......................... 26

5. **Summary of proposed solutions** ....................................................... 35

6. **Software used for empirical study I** ................................................. 42

7. **Results of empirical study I - TIME** .............................................. 44

8. **Results of empirical study I - Closure** ............................................ 46

9. **Results of empirical study I - Collections** ...................................... 48

10. **Results of empirical study I - Math** .............................................. 49

11. **Results of empirical study II** ......................................................... 54

12. **Changes coverage matrix** ............................................................... 60

13. **The program invariants of removeProduct** ...................................... 68

14. **Pre- and post-conditions of the modified removeProduct** ............... 70

15. **The program invariants after the change** ........................................ 71

16. **The program invariants after the library change** ............................. 73

17. **The program invariants after the database change** ........................... 75

18. **The details of the subject programs** ................................................. 91

19. **Results of Experiment I** ............................................................... 91

20. **The results of the changed libraries beanutil and digest** ................... 92
3.10 The results of the database changes
3.11 The results of Algorithm 3 on the case study I- ITM
3.12 The results of multi-objective selection algorithms on the case study I-ITM
3.13 The results of Algorithm 3 on the case study II- Microarray
3.14 The results of multi-objective selection algorithms on the case study II- Microarray
3.15 The results of Algorithm 3 on the case study II- Microarray
3.16 The results of multi-objective selection algorithms on the case study II- CEDCD
CHAPTER 1
INTRODUCTION

1.1 Overview

Software is a crucial component in many devices and systems used in our daily life. Thus it is important to ensure that software important correctly. There are various ways to monitor software execution behavior and to maintain software quality. In this dissertation, we focus on the techniques for online anomaly detection and regression test selection.

1.1.1 Anomaly Detection

Most online anomaly detection techniques use monitors in observing the behavior of production software. These monitors contain predefined rules as assertions to check if the software behaves as expected [5,10]. A monitor will raise an alarm when a software anomaly is detected; thus, it can give an warning of potential software failure.

Online anomaly detection needs a real-time reporting, which usually requires some performance overhead. For a large scale software, the overhead can be overwhelming and can significantly degrade the performance of the software.

This dissertation presents a new approach for online anomaly detection, which uses program invariants as oracles to monitor the program’s behavior and applies machine-learning algorithms to minimize the overhead. A program invariant is a program property that holds a particular value at a specific program point. It depicts software behavior at run-time. We collect program invariants from the software’s normal execution which can be used to depict normal software behavior. Then we use these program invariants as monitors to check if the program runs expected. However, using all program invariants may cause performance issues. We use machine learning algorithms (sensor placement algorithm) to select representative program invariants
for monitoring. The selection aims at maximizing the anomaly detection rate while minimizing the overhead by selecting a small subset of fault-revealing invariants.

1.1.2 Regression Test Selection

Regression testing is an important process that reruns the test suite to reassure previously working functions behave the same after changes. However, most software applications are continuously evolving, and test cases are accumulating after each change. Retesting all test cases may not be feasible if the test suite is large and the time to redeploy the software is limited. Selective regression testing and prioritizing test cases to reduce the testing time are essential.

This dissertation presents a new approach for regression testing that covers code changes and execution context changes, such as database changes and dependencies/library changes. We use program invariants as a means to represent the relationship between programs and test cases and select test cases that contain program invariants that are affected by the changes.

A program invariant reveals the software behavior by providing information learned from the executions, such as how a ‘variable’ is used as an instance of a library class. Thus, program invariants can be used to reflect changes in the code as well as changes in the execution context.

Each test case is augmented with a set of program invariants. When a program or the execution context is changed, at least one program invariant will be affected. Any part of the program that uses affected program invariants will be considered as affected and will need to be retested. We use these changed program invariants to select regression test cases for regression testing.

Moreover, the number of test cases that include affected/changed program invariants can be large, so it may not be feasible to retest them when there is only a limited amount of time for regression testing. We adopted a multi-objective algorithm that uses the coverage of affected/changed program invariants to prioritize the modification-revealing tests. The test cases that contain the highest number of the affected/changed invariants are tested first.
1.2 Program Invariants

1.2.1 Overview

A program invariant is a program property that holds a certain value at a specific program point during the execution of the program [Π]. Each program invariant may denote a constraint over a variable, represent a relationship between multiple variable values, or define a mathematical predicate.

A program invariant usually contains three parts: a program point, a set of variables and the relationship or constraint among the variables. A program point is a specific place in the program. A variable set is a set of variables that contain some values and are used to find the invariant. A constraint or relationship over the variable set, e.g. \(a > 0, a < b, a + b = c\), which keeps constant during the execution and this denotes the behavior of the software.

We use example 1.1 to demonstrate what program invariants are. It is a simple withdraw function that takes a parameter “money” and compares its value with “balance”. If “balance” is greater than or equal to “money”, then it subtracts “money” from “balance”. Otherwise, it throws an exception. By analyzing the program, we can identify some properties of the variables used in the program.

**Example 1.1: Program: withdraw function**

```java
public void withdraw(int money)
{
    if (balance < money) {
        throw Exception;
    } else {
        balance = balance - money
    }
}
```

We can infer that the parameter “money” has the constraint that its value must be greater than or equal to zero, which indicates the amount of “money” withdrawn from an account can not be a negative number. The variable “balance” is defined outside the function and must be a positive number. There is a constant relationship
imposed on the variable “balance” and the parameter “money”. We can denote
the constraint as $balance = original(balance) - money$ which means that “balance”
(after executing the program) is equal to the “money” subtracted from the original
“balance” (before executing the program).

The list below summarizes the properties of each variable set.

- Variable set $[money]$. Property [Constraint]: must be a positive number.

- Variable set $[balance]$. Property [Constraint]: must be a positive number.

- Variable set $[balance, money]$. Property [Relationship]: $balance = original(balance) - money$.

Definition: We define a program invariant as a tuple $I = <p; v; c>$, where
$p$ is the program point, $v$ is a set of variables and $c$ is a relationship or condition
imposed on $v$.

A program point is the place where we collect the program invariants. In the
example [1.1] the program point is the function “withdraw”. The terms “enter” or
“exit” following the function name indicate whether the program invariant is collected
before or after the function execution. The term “enter” suggests that the program
property is collected before the function execution and “exit” means that the program
property is collected after the execution. Therefore, example [1.1] has two program
points “withdraw:enter” and “withdraw: exit”.

Each program point contains sets of variables that are defined outside the func-
tion. In this example, the variables are the global variable “balance” and parameter
“money”. They keep certain properties before and after the function’s execution.

The program invariants before the execution of the withdraw function are
$I1=<p: withdraw:enter, v: money, c: must be a positive number.>$ and $I2=<p: with-
draw: enter, v: balance, c: must be a positive number.>$. After the execution, there
are three program invariants $I3$, $I4$ and $I5$. We will have $I3=<p: withdraw:exit, v: money, c: must greater or equal than zero>$ and $I4=<p: withdraw: exit, v: balance, c: must be a positive number>$. Besides these four properties, we also have the
relationship between “money” and “balance” from which we get $I5=<p: withdraw:
exit, v: balance and money, c: balance = org(balance)-money>. Table 1.1 lists the program invariants of example 1.1.

<table>
<thead>
<tr>
<th>Index</th>
<th>Program Point</th>
<th>Variables</th>
<th>Value/Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>withdraw :enter</td>
<td>money</td>
<td>money must be a positive number.</td>
</tr>
<tr>
<td>2</td>
<td>withdraw :enter</td>
<td>balance</td>
<td>balance must be a positive number.</td>
</tr>
<tr>
<td>3</td>
<td>withdraw :exit</td>
<td>money</td>
<td>money must be a positive number.</td>
</tr>
<tr>
<td>4</td>
<td>withdraw :exit</td>
<td>balance</td>
<td>balance must be a positive number.</td>
</tr>
<tr>
<td>5</td>
<td>withdraw :exit</td>
<td>money/balance</td>
<td>balance = org(balance)-money</td>
</tr>
</tbody>
</table>

1.2.2 The use of program invariants

Program invariants have been used in many research areas of software engineering, such as fault localization, reasoning about the correctness of the program, and program evolution [12, 13, 14, 5, 10, 15].

Fault localization

Many researches use program invariants to represent the data flow or execution trace. By comparing the program invariants of a normal execution with the abnormal one, we can get a set of program invariants that are different in the two executions. We can use these different program invariants to locate the first different program invariant, which suggests where the source of the faulty software behavior starts. Brun and Ernst [12, 3] use machine learning models of program properties resulting from program errors to classify and rank properties that may lead to failure.

Reasoning

This approach uses program invariants as an assumption/assertion to verify the status of execution [13, 10]. For example, we can use the program invariant I=< withdraw :enter,[money], the money must be a positive number> as an assertion that money ≥ 0 at the beginning of the function to check the value of the “money” at runtime.

Evolution
A program invariant can prevent programmers from introducing a regression fault, which is against the assumption of the program’s correct behavior [16]. This approach uses program invariants as an assumption/assertion to verify the status of execution after changes.

1.2.3 Inferring program invariant

A program invariant can be inferred through a static analysis or a dynamic analysis. Static analysis analyzes the code or the specifications of the program for all possible run-time behavior or analyzes the specifications of the program. Its strength is that it considers all possible paths of the execution and can perform analysis offline, which does not require execution overhead during the run-time. However, it requires a significant amount of time and domain knowledge to analyze the source code or specification. Besides, many programming languages provide abstractions that help to model the problem. This abstraction is realized during execution. Thus, it is difficult to predict the run-time behavior by using static analysis [11, 17].

Dynamic analysis is performed in a run-time environment. It usually requires instrumenting the program and collecting program properties (execution trace) during the execution of the instrumented program, then inferring the program invariant. Dynamic analysis is a black-box analysis, which does not require an understanding of the code. However, it depends on the execution of the program, so it may not cover all the execution paths and only collect likely program invariants [11]. The program invariants inferred by dynamic analysis are referred to as likely dynamic program invariant. The results obtained from a dynamic analysis may be different from the outcome produced by a static analysis. In example 1.2, the program sums up “number1” and “number2”. The variables “number1” and variable “number2” will not be considered as program invariants in a static analysis. The static analysis will predict that the “number1” and “number2” will contain any integer value. It will not be able to infer any pattern. However, when using dynamic analysis, after analyzing the execution trace, if “number1” and “number2” are always assigned within a specific range, it will infer that “number1” and “number2” have a value in that range.
Example 1.2: Program: add function

```java
public int add(int number1, int number2){
    return number1+number2 ;
}
```

**Tools for inferring program invariants.**

Daikon[18]: Dynamic invariant detection (Daikon) runs a program, observes the values that the program computes, and then reports properties that are true over the observed executions.

DIDUCE[19]: The DIDUCE tool (DIDUCE stands for "Dynamic Invariant Detection Union Checking Engine") instruments the subject program and collects its invariants incrementally during its execution, but it uses the values of variables only.

C-DIDUCE[20]: C-DIDUCE is build on Stanford’s DIDUCE tool (which only works on Java programs) and makes it compatible for C programming language.

Wisconsin and Microsoft[21]: Glenn Ammons, Rastislav Bodk, and James R. Larus a tool developed by which describes a system for inferring temporal specifications.

Virginia and Microsoft[22]: It is a dynamic analysis tool for automatically inferring temporal properties proposed by David Evans and Jinlin Yang. They analyze a program’s execution traces to infer a set of likely temporal properties.

In this dissertation we use dynamic invariant detection (Daikon) to collect program invariants. The dynamic invariant detector collects program invariants by learning over observed executions. It is a machine learning tool that can infer likely invariants in many programming languages such as Java,C/C++,C,F,Visual Basic, Perl, and Eiffel programs. Daikon is well maintained and widely used in software engineering research.

Figure [1.1] shows an architecture for dynamic invariant inference.

The following are the steps for the program invariant inference.

Step 1: Instrument the input program to get instrumented program.
Figure 1.1: dynamic invariant inference

Step 2: Execute test suite on instrumented program to get program execution profiles stored in a data trace file or a database.

Step 3: Analyze the execution profiles to infer the program invariants.

The dynamic invariant inference tool uses pattern matching to infer program invariants.

First, identify the variable set at each program point. Second, instantiate feasible patterns over all combinations of the variables. For example, If a function has integer variables x, y, then instantiate the pattern \( x = y, x \leq y, x \geq y, x \neq y \). Third, execute the program and collect the values assigned to x and y. For each execution, check the values against the pattern. For example, the program has been executed twice and we collect values of x and y: \((x:1,y:1),(x:3,y:5)\). We apply each value in the patterns and dismiss the pattern, if the values do not satisfy the pattern.

- case1, \((x:1,y:1)\), dismiss pattern \(x \neq y\).
- case2, \((x:3,y:5)\) dismiss pattern \(x \geq y, x = y\).

After evaluating all the cases, the relationship \(x \leq y\) remains, others are dismissed because the value set violates the pattern. Then we have a program invariant \(I = < \text{program point, } x, y, x \leq y >\).
1.2.4 The types of program invariants

Table 1.2: Examples of the types of the program invariants

<table>
<thead>
<tr>
<th>Type of variable</th>
<th>Constrains or relationships</th>
<th>example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single variable</td>
<td>Constant Value</td>
<td>(x = a)</td>
</tr>
<tr>
<td>Single variable</td>
<td>Uninitialized Value</td>
<td>(x = \text{uninit})</td>
</tr>
<tr>
<td>Single variable</td>
<td>Small value set</td>
<td>(x \in {a, b, c})</td>
</tr>
<tr>
<td>Single variable</td>
<td>Variable type</td>
<td>(x) is a ”String“</td>
</tr>
<tr>
<td>Single Numeric Variables</td>
<td>Range Limits</td>
<td>(x \geq a, x \leq b)</td>
</tr>
<tr>
<td>Single Numeric Variables</td>
<td>Non-zero</td>
<td>(x \neq 0)</td>
</tr>
<tr>
<td>Single Numeric Variables</td>
<td>Modulus</td>
<td>(x = a \pmod b)</td>
</tr>
<tr>
<td>Single Numeric Variables</td>
<td>Non-Modulus</td>
<td>(x \neq a \pmod b)</td>
</tr>
<tr>
<td>Two Numeric Variables</td>
<td>Linear Relationship</td>
<td>(y = ax + b)</td>
</tr>
<tr>
<td>Two Numeric Variables</td>
<td>Functional Relationship</td>
<td>(y = f(x))</td>
</tr>
<tr>
<td>Two Numeric Variables</td>
<td>Comparison</td>
<td>(x &gt; y, x = y)</td>
</tr>
<tr>
<td>Three Numeric Variables</td>
<td>Combinations of Single Numeric Values</td>
<td>(x+y = a \pmod b)</td>
</tr>
<tr>
<td>Three Numeric Variables</td>
<td>Polynomial Relationship</td>
<td>(z = ax + by + c)</td>
</tr>
</tbody>
</table>

The type of program invariants defines the constraints and relationships over the variables. It also refers to the pattern in the previous discussion. Table 1.2 lists some types of program invariants. For a single variable, the constraint will be “the variable has a constant value”, “whether or not the variable has been initialized”, “value range”, or “the variable’s type”. The variable’s type can be float, int, string, class etc. It is very useful, especially for a Java application where a Java function can take input with various class types. It is good to know exactly what class types can be accepted by the function. The relationship over two or more variables can be linear, functional, comparison, or combinations of single numeric values. For three numeric variables, they can have a polynomial relationship; for example, the fifth program invariant in the program invariant Table 1.1.
1.3 Online Software Anomaly Detection

1.3.1 Background

Abnormal software behavior is any software behavior that deviates from what is expected. It can be caused by residual code defects, malicious activity or the anomalies stemming from the execution context [23, 24].

Software anomaly may eventually lead to a software failure. When an abnormal behavior propagates to an exit point of the software and produces an incorrect output or an unexpected termination of the execution, it is considered a software failure. An anomaly may or may not induce failure, but when a failure occurs, there must be at least one point in the program execution that has an anomalous behavior. Software failure caused by a code defect will not re-occur if the fault is corrected without any regression fault. However, an anomaly or failure caused by the execution context may be recurrently encountered during its production mode.

Anomaly detection refers to the problem of finding patterns in user behavior data that do not conform to expected behavior [25].

Online software anomaly detection is an approach to investigate if the behavior of the production software deviates from what is expected at any point during its execution. Early detection of online software anomalies can reduce negative consequences and further prevent subsequent anomalies.

1.3.2 Motivation

Effective online software anomaly detection is crucial for ensuring the quality of the deployed software, but it is incredibly challenging to perform in practice. Detecting software anomaly at an early stage can potentially reduce the cost of the software anomaly [5].

In practice, many software systems use monitors or loggers to track software behavior, which can be useful to analyze software bugs, such as null pointer exceptions. However, the log files can not be used to analyze business logic errors, which usually require engineers to manually set up a set of rules to track the execution of
the program and require domain knowledge to find the places to put monitors in the program for validation while keeping the overhead at minimum.

Log files can record not only the warning or error messages but also some other information. After an application is running for a while, the log files usually contain a massive amount of data, so searching for a particular record may not be easy. Moreover, most log reorders are manually set up, so some important information may be overlooked.

Online anomalies are usually reported by users after they receive and review the results (except for software crashes or exceptions that can be detected from the server-side). Consequently, the log files are usually used in debugging after the anomalies have been reported, but without an effective method to determine what information to log or how to evaluate this information online, they are not suitable for online anomaly detection.

1.3.3 Challenges

There are two major challenges in anomaly detection.

Validating without a pre-defined oracle

The detection of online software anomalies require a validation oracle, which is generally difficult to create [2]. For online software, inputs are provided by the user at the runtime, and the correctness of the outputs cannot be automatically determined without an oracle [26]. Moreover, some anomalous behavior may be caused by the execution context, not by code defect, which is even more challenging to detect.

Monitoring slows down the performance

The number of program invariants of a large-scale software is usually large, making it infeasible to monitor all invariants online. Using all of them as assertions will slow down the performance. It is crucial to minimize the number of program invariants that are not necessary for monitoring. Thus, an effective selection of invariants is essential for improving the effectiveness of monitoring and minimizing the overhead.
Some research studies proposed solutions to overcome these two challenges.

Baah and Harrold [26] proposed a statistical machine learning approach for online anomaly detection. Their approach creates an observable Markov model at each conditional statement to determine if the execution is abnormal. But their empirical results show that it only works well on certain types of fault and may require a large amount of time in creating a model. Also, the amount of runtime overhead is not clear.

Hangal and Lam [5] proposed using program invariants to find program anomalies. This approach can help to detect the root causes of errors. However, tracking all the invariants may require significant overhead to the program executions. Their empirical results show that it requires 6-12 times of standard performance by using ten machines.

Brun and Ernsts [3] used machine learning techniques to classify the program properties. This approach identifies program properties that are likely to be fault-revealing. They created a model to analyze the properties of any given program and rank the properties based on their likelihood of being fault-revealing. However, their empirical results show that were not very accurate.

We use program invariants as oracles to address the first challenge. Several studies have suggested using program invariants to investigate program behavior [3, 4, 5]. We use program invariants as means for capturing persistent conditions of program properties that should hold for every normal execution.

Using the Java program 1.1 as an example, we can use the program invariant in Table 1.1 \( I_1 = \langle \text{“withdraw :enter”},[\text{“money”}], \text{“the money must be a positive number.”} \rangle \) as an oracle, by putting a monitor at the entrance of the function to check if the input of withdraw function, is valid or not. If the input is a negative number, a warning will be sent to the administrator.

We developed algorithms which adopt the power of machine learning to address the second challenge. Our approach is two phases. The first phase aims to filter out the superfluous program invariants and those that are not sensitive to software anomalies. We used the correlation between program invariants and software anom-
lies and also account for the correlation between program invariants for each execution. In the second phase, we selected an a close to optimal set of program invariants with minimum overhead to cover the maximum number of software anomalies. We clustered program invariants that are frequently violated in the same executions and selected only the representative program invariants (those appear in most clusters) from each cluster for monitoring. For the algorithm, we adopted the commonly used sensor placement models and developed algorithms for invariant selection to maximize anomaly detection rate while minimizing the overhead. We conducted two empirical studies on four open-source software with real faults and on an industrial system to evaluate the effectiveness of our approach. The studies show promising results and suggest the feasibility of applying this approach in practice.

1.4 Regression Test Selection

1.4.1 Background

Most software systems are continuously evolving to serve users’ needs better. Consequently, maintenance needs to be frequently applied to multi-version production software systems. The ongoing changes may take place in the core of the systems or in their supporting execution context. When a change is made to the software or its execution context. Regression testing is performed on the software to validate the changes. The purpose of regression testing is to provide confidence that the newly introduced changes do not obstruct the behaviors of the existing, unchanged part of the software.

To ensure that the changes do not adversely introduce regression faults, causing previously working features to fail, regression testing, which reruns the previous successfully executed test cases on the modified software, should be performed. Furthermore, when the execution context is changed, regardless of whether the code is changed or not, regression testing also needs to be performed to check the compatibility when the execution context such as run-time libraries, external APIs, configuration files or databases are modified.

Re-executing all previous test cases is the most straightforward approach. How-
ever, since software is continuously evolving, the test suite tends to grow. Executing the entire test suite may not be feasible. Many research studies have been performed to accelerate the regression testing process. There are three major approaches: test suite minimization, test case selection, and test case prioritisation \[27\].

Test minimization aims to remove redundant test cases to reduce the testing time. It selects a minimum subset from the test cases to cover the modified parts of the program that need to be retested.

The test suite minimization problem can be defined as follows:

Given a test suite, \( T \), a set of requirements \( \{r_1, \ldots, r_n\} \). The subsets of \( T \), \( \{t_1, \ldots, t_n\} \), each of them is associated with a requirement \( r_i \), such that any of the test cases \( t \) belonging to \( T \) can be used to test requirement \( r_i \).

Problem: Find a minimal set, \( T' \in T \) that covers all the requirements \( r_i \) \[27\].

The criterion is that every requirement \( \{r_1, \ldots, r_n\} \) must be tested. A test requirement \( r_i \) is satisfied when a test case \( t \), that belongs to the \( T_i \) is selected for retesting, and it is associated with \( r_i \). To maximize the effect of minimization, \( T' \) should be a minimal set that covers all requirements. The test suite minimization problem can be a problem of the minimal set cover problem \[27\].

Similar to the test suite minimization problem, test case selection also aims to reduce the size of a test suite. Test case selection aims to find modification-traversing test cases. Thus, test case selection will focus on identifying the modification part of the program or system, then selecting test cases relevant to the changed part.

The test case selection problem can be defined as follows: Given a program \( P \); let \( P' \) be a modified version of \( P \); and let \( T \) be a test suite for \( P \). A typical selective regression testing proceeds as follows:

(1) Select \( T' \in T \), a subset of test cases \( T \).

(2) Test \( T' \) on \( P' \), verify \( P' \)'s correctness.

(3) If necessary, create \( T'' \), a set of new test cases for code change of \( P' \).

(4) Test \( P' \) with \( T'' \), verify \( P' \)'s correctness.
Create $T''$, a new test suite and test execution context for $P'$.

Test $P'$ with $T''$, verify $P'$'s correctness.

Test case prioritization ranks the test cases based on a particular criterion, such as code coverage, and executes the test cases according to the order of their ranks. It is suggested that with the prioritized order, test adequacy can be achieved early and faults can likely be detected early, which can yield certain confidence on the quality of the regression testing process that needs to be terminated early unexpectedly.

The test case prioritisation problem can be defined as follows: Given a test suite, $T$, the set of permutations of $T$, $PT$, and a function from $PT$ to real numbers, $f : PT \rightarrow R$. Problem: to find $T' \in PT$, such that $(T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

Because test case prioritisation and test suite minimisation may not be safe, the criterion for these two approaches is using minimal effort to cover maximum requirements. These approaches may miss some regression faults.

1.4.2 Motivation

Regression testing is crucial as it gives us confidence that the software meets the requirement and changes made to the software will not affect the behavior of the unchanged part. Re-executing all the test cases can be time-consuming when new tests are continuously added to the test suite, making the test suite very large. To minimize the maintenance cost and prevent service disruption of the production software, an efficient and effective selection of test cases from the regression test suite to be used to perform regression testing is essential. Safe and selective regression testing that aims at minimizing the cost while maintaining the effect of retesting all tests has been well studied.

The minimization technique aims to identify and then eliminate the obsolete or redundant test cases from the test suite. It usually applies a criterion, such as code coverage, to select a subset of the test suite that satisfies the criterion with no redundant or obsolete test cases. However, some redundant test cases maybe modification-traversing and eliminate them will make the technique non-safe.
Test case prioritization depends on a particular criterion. It helps in revealing the regression fault early and works well when the budget for regression testing is tight. However, it may overlook some regression faults that have a limited impact on the criterion. Thus prioritization techniques may not be a safe technique.

Many researches focus on regression test selection (RTS). Regression test selection requires identifying how changes in the code impact the software. Modification-revealing tests have been suggested for detecting regression faults [8]. The key assumption for safe, controlled regression testing is that when a modified program $P'$ is tested with a test case $t$, all factors that might influence the output of $P'$, except for the code in $P'$, are kept constant with respect to their states when the original program $P$ is tested with $t$ [8]. These approaches can be safe if the changes are made to the code only. However, when the execution context is changed, or if there exists a dependency in the system via an external entity, these approaches may not be safe.

Some studies show the ability to cover the non-code changes. Haraty et al. [30] presented regression test selection techniques for SQL-based applications. Nanda et al. [31] presented an approach to address regression test selection when there is no code change but property files or databases are changed. These approaches handle configuration/property file and database changes but do not account for code and library/API changes. The regression faults caused by the dependency can be detected by using a dataflow analysis [32, 33, 34, 12]. However, some indirect dependencies via an external entity, such as a database entity or an element in a configuration file, may not be identified. Each of these approaches only tackles a single aspect in the regression testing.

A safe regression test selection technique is needed to identify all fault-revealing test cases and account for the changes in both execution context and program. In the meantime, it should be safe and as effective as existing approaches for program changes.

1.4.3 Challenges

The majority of modern software relies heavily on the support of the execution context, such as runtime libraries, external APIs, configuration files, and databases.
When a software library or API is upgraded to a new version that is not backward compatible, the application may fail [35]. These library compatibility issues are often encountered in multi-team software developments, where each team uses its own execution environment that may differ from the one in the production or the one being used by other teams, and backward or forward compatibility problems may occur. Such compatibility issues have been addressed [35], but an effective method of regression testing that resolves issues in execution context has yet to be explored.

Changes to databases and configuration/property files can also affect a program’s behavior. Haraty et al. [30] and Nanda et al. [31] discussed the impact of database and configuration changes and presented regression testing techniques to handle these types of changes. However, these techniques do not account for code or library/API changes.

The impact of the changes in the execution context on the program behavior is challenging to identify and resolve in software maintenance. To our best knowledge, the existing techniques for regression testing have barely addressed these issues. Most of the safe selection approaches assume that the external factors of the subject program remain constant, which is often impractical for modern multi-team developed evolving software.

Our approach, Context-Aware Regression Testing (CART), accounts for both changes in the code and in the execution context. While existing approaches tackle a single type of change, CART can effectively handle different types of changes at the same time, and select not only modification-traversing but also modification-revealing tests. Thus, it is not subject to the constraint of constant factors imposed on the existing safe approaches.

We use the program invariants before and after an invocation of a function to determine if the function is affected by the change and can potentially have regression faults. A program state before a function call denoted in the precondition of a function, including the properties required for the successful execution of the function and the postcondition of a function, indicates the state of the program after the execution of the function. These properties can be assertions on a class/object attribute, global variables, elements defined in a property/configuration file, input
parameters, return values, or entity objects corresponding to database entities. Any changes among those will be captured in the program invariants. To annotate pre- and postconditions automatically, we use program invariants as a means for capturing persistent properties of functions.

The selection is simple as it identifies tests that execute program invariants affected by the changes in the code, upgrades/downgrades of the library, and changes in databases, API, or configuration files. This approach supports a safe or prioritized selection technique. For a safe selection, CART selects all the tests that execute the affected program state. However, to maintain evolving production software, regression testing must be efficient and effective to avoid service disruption. Testing all selected test cases may be too expensive, especially in the emerging DevOps software development environment. The time for regression testing is often limited and the regression test selection has a significant impact on maintainability. Thus, when time is limited, the test cases selected by the safe approach cannot be executed all. Therefore, we should first select the tests that execute most affected functions, which aims to exploit regression faults early by selecting the subset of these test cases that is most likely to detect regression faults.

Several multi-objective regression testing approaches, which apply multiple criteria to select a close to optimal subset of test cases that account for the selected criteria to obtain a maximal efficacy, have been proposed. To maximize the fault detectability within a constrained time period, we model the test selection as a multi-objective, multi-dimensional Knapsack problem that determines the number of test cases to select so that the total execution time does not exceed the time limit and the total number of faults that can be detected as many as possible. We formulate an objective function that includes the coverage of the affected functions, the coverage of the modified functions, and the test execution time. In addition, we use a function-call network graph to rank the degree of the impact of each function by using the number of callers of each function. It is assumed that if a function is affected by a change, then the callers will have more execution contexts and will be more likely to reveal faults.
CHAPTER 2
ONLINE SOFTWARE ANOMALY DETECTION

2.1 Overview

Online anomaly detection monitors the software and prompts an alert when it detects abnormal behaviors before a program fails [26]. The root cause of abnormal software behavior can be varying and can be caused by software defects or improper data or execution context.

Anomaly detection mechanisms usually require setting monitors at suspicious fault points of the program and checking the program properties at runtime. Once a monitor detects an abnormal behavior, an alert is sent out. The monitors are usually a set of assertions or a set of conditions that should be satisfied during the execution. An assertion can be used to check the type of an object, value range, relationships, etc. For example, when a program is running and one assertion finds that the variable range is out of the pre-defined range, then an alert will be sent. Tracking the values of the variables can prevent unexpected data from being delivered to the client. Assertions can be used to track the execution flow, which can be traced in the control flow. A control flow has a predicate that controls the direction of the execution, such as an if-else statement. When condition C is satisfied, the execution goes to branch A, otherwise branch B. By monitoring the predicate (condition C), we are able to track the program execution flow.

Efficiency and effectiveness are the keys to online anomaly detection. The approach that monitors online software should not cause too much overhead while being able to detect all abnormal behavior. This means that the monitors should have a high-quality assertion that precisely describes what normal behavior is and should be well-placed. This can be formulated as a coverage problem so that the detection uses the minimum number of monitors to cover the entire execution trace.
2.2 Existing approaches and limitations

Baah, Gray and Harrold [26] propose a statistical machine learning approach for online anomaly detection. Their approach first instruments the subject program to collect predicate trace information during the executions in the training phase. The predicate information is used to form predicate states, which are clustered using a learning algorithm to create an observable Markov model. For anomaly detection, the model is deployed with the instrumented software and analyzes the information produced by the instrumented software. The results from the analysis can be used to investigate the causes of the anomaly. This approach has two limitations: Their empirical results show that it only works well on certain types of faults, such as domain errors; besides, it may require a significant amount of time for model creation, and the execution overhead may affect the performance.

Hangal and Lam [5] present a dynamic online program invariant detection and checking engine (DIDUCE) for finding program anomalies. DIDUCE instruments the subject program and collects its invariants incrementally during its executions. At each instrumented program point, if the invariant does not conform to its hypothesis, then a violation is reported and the invariant is relaxed when the program resumes. Invariant violations in the tracked expressions are used to check program anomalies and to help find root causes of program errors. Their empirical studies show that DIDUCE identified the root causes of the known faults in the four real-life programs. It also found a few undetected defects. This approach can help find the root causes of errors. However, the effectiveness of this approach to find program failures or anomalies is not addressed and tracking the invariants may require significant overhead to the program executions. Their results show that the performance of the instrumented programs was slowed by a factor of six to twenty; such costs limit the applicability of this approach at runtime.

Brun and Ernst [3] use machine learning models of program properties resulting from program errors to classify and rank properties that may lead to failure. This approach identifies program properties that are likely to be fault-revealing. It first creates a model for fault-revealing properties by using machine learning to train properties on erroneous programs and their fixed versions. Properties that appear
in the erroneous version but not in the fixed version are considered fault-revealing. This model then can be used to analyze properties of any given program and to rank the properties based on their likelihood of being fault revealing. Their experimental results show that by selecting the top 80 properties, the relevance of choosing fault-revealing properties increases by a factor of 50 on average for the C programs and 4.8 for the Java programs.

Dickinson et al. [36] empirically evaluate a family of observation-based techniques that use automatic cluster analysis to partition the executions and apply sampling methods to select the executions from clusters. In these experiments, function-call counts were used for execution profiles, and an agglomerated hierarchical clustering algorithm was used for clustering the executions. Their results suggest that cluster analysis can detect failures that have unusual profiles, and that failures tend to cluster together. Furthermore, they propose a failure pursuit sampling method to reveal failures from clusters that having unusual profiles more effectively.

The results of their experiments show that the failure-pursuit sampling method is more efficient than the other methods used in the experiments. This approach requires the completion of the executions to investigate if the execution is likely unusual. It may not work on programs that have no known faults, such as most deployed software and no unusual profiles for comparisons.

To reduce the runtime execution overhead, the GAMMA system, developed by Orso et al. [37] applies software tomography technique to multiple instances of software. It distributes monitors in different instances; each of them has a small set of monitors to reduce the execution overhead. The information collected by the monitors from each instance is integrated for analyzing software behavior in different instances. This system collects structural coverage (statement, method and class coverage) information from each instance and does not monitor dynamic behavior, or provide an oracle.

Loyola et al. [38] present a system, Dodona, to support engineers for generating test oracles for Java unit testing. Dodona selects a set of variables to monitor and uses them for creating assertion-based oracles. Their selection strategy is based on the ranks of the program variables. Dodona monitors the dataflow relationships between
variables in each execution to build a network, then applies a network centrality analysis to rank the relevance of each variable. It then maps the variables to the monitoring points and recommends an oracle data set for the given test input. Using this ranking, Dodona proposes a set of variables to monitor that can be used to construct assertion-based oracles. However, the cost of creating the variables network will be very high.

Perkins et al. [39] present a system, ClearView, for automatically patching errors in deployed software with high availability requirements. ClearView focuses on monitoring the values of the properties of registers and memory locations and is designed to detect a specific class of errors (heap buffer overflows, and illegal control flow transfers). ClearView reduces the overhead by analyzing the control flow graphs to identify sets of distinct variables, denoting values in registers or memory locations, in the same procedure that always have the same values and selecting only the one in the earliest instruction of the execution. Our approach is similar to ClearView but not subject to detecting the errors of heap buffer overflows and illegal control flow transfers. Our optimization strategy is different from ClearView, which eliminates variables referring to the same registers or memory locations and is not suitable in our model where each property is at a distinct point of the program.

The existing approach cannot perfectly address the issue we have right now. Some of them may introduce too much overhead or only work well in some fields, missing other types of abnormal behavior.
2.3 Methodology

2.3.1 Introduction

Our online anomaly detection model uses program invariants as assertions to monitor the software at runtime to detect anomalous software behavior. Our method focuses at the function level. For each function, our method produces at least two program invariants: a program invariant at pre-condition and a program invariant at post-condition. A complex application may have thousands of functions and monitoring all the functions will introduce enormous overhead. We use an algorithm which selects an approximately optimal set of representative anomaly-revealing properties (program invariants) for monitoring. The selected anomaly-revealing program invariants should be representative and anomaly sensitive so that when the software runs abnormal, the program invariant will detect it.

The selection criteria are subject to three constraints:

(1) Effectiveness: False positives and false negatives are minimal.

(2) Efficiency: The execution overhead required for monitoring at runtime does not exceed the user-defined performance threshold.

(3) Coverage: The distribution of the monitors satisfies user requirements if it is specified. For example, a user may require a monitor to be placed for each critical or frequently executed class or method.

For effectiveness, we evaluate selected program invariants by collecting the number of faults that are not detected by the program invariants and a number of times that a program invariant sends an alert without a fault. We use an algorithm to select program invariants that can cover most of the anomalies and with minimal false alerts.

For efficiency, we set up a budget (number of monitors that can be used for monitoring) to reduce the risk of overhead.

For the coverage, we use a network graph, such as a function call graph and a class diagram to represent the distribution of the program invariants. Selecting program invariants based on the function call graph or the class diagram will help
the selection to have an even distribution.

Figure 2.1 shows the workflow of our approach. Our approach first needs to instrument the program and learn what properties of the program are preserved in all the normal executions. We run the instrumented program with test cases to get data-trace files, then apply Daikon on it to infer program invariants. When a program fails and faults are corrected, we collect its program invariants from its abnormal execution and compare them with the collection from the normal execution and then analyze this data to infer the program invariants.

This approach helps to find the anomaly-revealing program invariants by comparing the program invariant of normal executions (after bug fix) and abnormal executions (before bug fix). If the same program invariant shows different properties in two executions, then we consider this is an anomaly-revealing program invariant which has a higher value for anomaly detection as it is more sensitive to the fault
than those that are not anomaly-revealing program invariants.

Next, we apply machine learning algorithms to select a small set of anomaly-revealing properties that satisfies the three constraints mentioned above. For the monitoring, we instrument the subject program to monitor these program invariants at runtime. If an abnormal behavior is detected by a monitor, there will be an alert and an engineer can analyze it and fix the issue if possible. When a false alert is encountered, which means the monitor detected a violation of an invariant during a normal execution, then this invariant will be checked and updated.

Our methodology focuses on software anomalies caused by the code defects, external APIs change, libraries change and database changes.

Table 2.1 lists types of abnormal can be detected by our methodology.

<table>
<thead>
<tr>
<th>Type of anomalies</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain Errors / Arithmetic error</td>
<td>Domain errors occur when dealing with mathematical functions, such as numeric expressions that are mathematically undefined or cannot be represented numerically on the computer for reasons other than missing data. A concrete example is the square root of a negative number.</td>
</tr>
<tr>
<td>Point to empty/Null Resource</td>
<td>For example: Null pointer dereference or using an uninitialized variable.</td>
</tr>
<tr>
<td>Business logic</td>
<td>This is a high level software anomaly driven from the requirement. For instance the balance in a bank account cannot be a negative number.</td>
</tr>
</tbody>
</table>

A domain error is usually related to a mathematical function. When an input is outside the scope of where the mathematical function is defined, a domain error will occur.

**Example 2.1: A Java example of domain errors**

```java
private float div ( int a , int b){
    return a/b;
}
```
We use the Java program above to demonstrate how our methodology detects a domain error.

The division function, which takes inputs “a” and “b”, returns the value after executing “a” divided by “b”. The program fails to check the value of “b”, ignoring the risk that “b” can be 0, which would cause an arithmetic error.

Our approach learns what properties of the program should preserved in all the normal executions. By analyzing all the normal execution data, we can infer the program invariants as the list in Table 2.2. There are six program invariants. The first, third and fifth invariants indicate the type of the variable or the return value. The second, fourth and sixth show the property of the variable’s value. The fourth one denotes that the “b” cannot be zero. This property is inferred by an extensive analysis on b’s value in the normal executions. We use this property as a monitor to check b’s value before function div is executed. This way, we can detect an arithmetic error.

Table 2.2: List of program invariant of domain error example

<table>
<thead>
<tr>
<th>Program point</th>
<th>variables</th>
<th>condition/ relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>div:enter</td>
<td>a</td>
<td>a is integer</td>
</tr>
<tr>
<td>div:enter</td>
<td>a</td>
<td>a has a value</td>
</tr>
<tr>
<td>div:enter</td>
<td>b</td>
<td>b is integer</td>
</tr>
<tr>
<td>div:enter</td>
<td>b</td>
<td>b! = 0</td>
</tr>
<tr>
<td>div:exist</td>
<td>return value</td>
<td>return value type is float</td>
</tr>
<tr>
<td>div:exist</td>
<td>return value</td>
<td>return value !=-1</td>
</tr>
</tbody>
</table>

For the null pointer exception, we can use the same example like the one above. The second program invariant in Table 2.2 indicates that the variable must have a value and should not be null. We use this property as a monitor to check a’s status.
before the function div is executed. This way, we can detect the null pointer exception.

For the business logic error, we reuse the example in the introduction section (the program Java Code 1.1 Withdraw function). The function withdraw takes parameter, money, which is an integer and compares its value with the balance. If the balance is greater than or equal to the money, then it is fine to subtract money from the balance. Otherwise, throw an exception. Ideally, by a static analysis, the parameter money can be any integer value. However, business logic places the restriction that the money should be greater than or equal to zero. This can be inferred through an analysis of the actual value of money in all the normal executions. Through that, we get a program invariant in Table 1.1 to show this business logic constraint. By monitoring the value of money, we can add a business logic constraint to the program.

Our methodology includes three phases.

The training phase aims to find anomaly-revealing program invariants.

The probing phase seeks to use a machine learning technique to select a close-to-optimal subset of the invariant.

The monitoring phase utilizes the program invariant as an assertion to monitor software behaviors and validate the performance of abnormal software detection.

2.3.2 The Training Phase

The training phase automatically identifies software anomaly-revealing properties through a learning process to investigate the correlation between software property and anomalous behavior, and the correlation between properties that were violated in the same anomaly execution. In the training process, the subject software is executed by using a set of test cases, and the values of the properties are evaluated and compared between normal and abnormal executions to determine anomaly-revealing properties. We use program invariants to denote properties. A program invariant is a condition that holds the program's property at a given point, where the invariant should hold a true value if the program's execution behaves as expected; otherwise, a false value should be observed. Under this assumption, if software fails or behaves un-
expectedly, then at least one or more invariants should have a false value. Thus, these violated invariants can be used as an indicator of software anomaly. Our approach is to run a large set of test cases on the subject software. After each successful execution (normal behavior), the invariants are recorded or updated, and for each failure (abnormal) execution all the violated invariants are clustered, which are considered as candidates of anomaly-revealing invariants.

Program invariant collection requires instrumenting the program, collecting the execution data then by analyzing the execution data to infer the likely program invariant. We use Daikon in the training phase to identify likely dynamic program invariants. The Daikon invariant detector, developed by Ernst et al. [11], has been used in many research areas of software engineering, such as for monitoring runtime heap and memory for security protection [39], inferring program contracts [40], automated software fault localization [41, 42, 43, 14], identifying non-equivalent mutants in mutation testing [13], verifying component-based software systems [44], detecting dynamic invariants of relational databases and their applications [45] and software reliability prediction [46, 47]. Daikon detects dynamic invariants by instrumenting subject programs at selected points to trace the values of the variables at these program points during the executions on a test suite and infers invariants over both instrumented and derived variables.

The training process first instruments the subject program by using Chicory and then runs Daikon on the instrumented program with a set of test cases. If a test passes, then the values of the invariants are updated, otherwise the values of the invariants are compared with those in the program in which the corresponding faults have been corrected. The invariants that have different values are considered as anomaly-revealing. Each abnormal/failure execution should have a set of anomaly-revealing invariants, which are clustered into the same group. The need for identifying anomaly-revealing invariants is because the number of invariants inferred by Daikon is very large, usually about the same size of the programs LOC. By running a set of abnormal test cases, we can filter out invariants that are not sensitive to program anomalous behaviors. The invariants that belong to the same cluster are considered correlated, and are likely to be violated at the same time when an anomaly occurs,
and we can further select one representative from each cluster to reduce the number of invariants to monitor.

### 2.3.3 The Probing Phase

The aim of the probing phase is to select a close-to-optimal subset of the invariants satisfying three requirements:

1. If any of the invariants in this set is violated, then the software is most likely to fail;
2. The distribution of these invariants satisfies the user’s requirement;
3. The overhead required for monitoring these invariants at runtime does not exceed the user-defined performance threshold.

To model this problem, we adopted the sensor placement approach to develop efficient and effective algorithms for selecting the invariants. The sensor placement problem has been mostly explored in environmental areas, such as placing sensors in a large water distribution network to detect the malicious introduction of contaminants [48] and in an indoor environment to monitor temperature [49]. Existing solutions to this problem fall into two main categories: geometric and model-based. The former assumes that each sensor has a fixed region (e.g., circle) within which everything can be perfectly observed and then aims to place sensors such that the overlap of their regions fully covers the sensing area [50]. The latter builds a probabilistic model of the underlying process and then places sensors to optimize an objective function of that model.

Since the model-based approaches fit our problem better; therefore, we applied them in our model. Many different objective functions have been proposed for model-based sensor placement, such as the quality of parameter estimates in statistical community [51] and information-theoretic measures, notably entropy [52]. All of the criteria discussed thus far yield challenging combinatorial optimization problems that are often NP-complete [49] and the proposed optimization techniques in the literature can be categorized into two groups: combinatorial search and continuous relaxation. For this first group, since an exhaustive search is usually unfeaa-

29
sible, local heuristic searches without theoretical guarantees have commonly been applied \[53\], \[54\]. Recently, by exploring sub-modularity of information-theoretic measures (e.g., mutual information), optimization algorithms have been proposed which guarantee both the runtime and the quality of the achieved solutions \[48\], \[55\], \[56\], \[49\]. For the second group, the combinatorial optimization problem is approximated as a relaxed numerical optimization problem and then convex optimization techniques are applied \[57\], \[58\]. This sub-modularity-based approach has been well applied in practice and is available in public tool boxes for water distribution systems, such as Sensor Placement (S-PLACE) Toolkit \[59\] and Threat Ensemble Vulnerability Assessment Sensor Placement Optimization Tool (TEVA-SPOT) \[60\]. This approach has also been applied to many other relevant applications, such as wind studies around buildings \[61\], indoor climate monitoring \[62\], underwater surface inspection \[63\] and traffic dynamics reconstruction \[64\].

Assuming that \( S \) is the set of selected invariants that fulfills the three requirements mentioned above, let \( F(S) \) be the overall detection risk that accounts for possible false alarms and false positives and \( C(S) \) be the performance cost, which is a function of the performance cost for monitoring the invariants in \( S \). The goal is to determine an optimal set \( S \) that minimizes \( F(S) \) and \( C(S) \), subject to a budget constraint on the total number of invariants that can be monitored (\( C(S) \leq K \)). This is a multi-criterion optimization problem and the scalarization approach \[58\] is commonly used to find such Pareto-optimal solutions. In particular, we optimize the objective function \( O(S) = F(S) + \lambda C(S) \) by choosing the appropriate weight \( \lambda > 0 \). All possible Pareto-optimal solutions to the minimization of \( O(S) \) subject to the budget constraint \( C(S) \leq K \) can be obtained by varying the weight \( \lambda \). The overall detection risk \( F(S) \) is also a sub-modular set function. If the performance cost function \( C(S) \) is a modular set function (e.g., the sum of the cost for monitoring invariants), then each of the preceding sub-problems corresponding to the specific weight \( \lambda \) can be formulated as a budgeted sub-modular minimization problem and be efficiently approximated in nearly-linear time \[65\]. If \( C(S) \) is not modular (e.g., the product of the cost for monitoring invariants), then techniques for optimizing difference between sub-modular functions can be applied \[36\]. Details are given in the following.
The software is considered as a network of components (classes) and is represented as \( G = (V, E, c, x) \), where \( V = \{v_1, \cdots, v_N\} \) refers to the set of classes (vertices), \( N \) refers to the total number of classes, \( E \subseteq V \times V \) refers to the set of edges (relations) between classes, each vertex \( v_i \) has \( P_i \) invariants, \( c_{i,j} \) refers to the cost of the \( j \)-th invariant of \( v_i \), \( x = (x^{(1)}, \cdots, x^{(T)}) \) represents invariant violations reported at \( T \) time intervals and \( x^{(t)}_{i,j} \) refers to the number of the violations of the \( j \)-th invariant of \( v_i \) at time \( t \), \( i = 1, \cdots, N \) and \( j = 1, \cdots, P_i \). Without loss of generality, we assume that \( x \) and \( y = (y^{(1)}, \cdots, y^{(T)}) \) represent the training data, in which each time interval involves the execution of a single testing case and \( y^{(i)} \in \{0, 1\} \) indicates the success (0) or failure (1) of this test.

We define \( U = \{(v_1, 1), \cdots, (v_1, P_1), \cdots, (v_N, 1), \cdots, (v_N, P_N)\} \) as the ground set of all invariants, in which the tuple \((v_i, j)\) denotes the \( j \)-th invariant of the vertex \( v_i \), \( \mathcal{P}(U) \) as the power set of \( U \) and \( Q \subseteq \mathcal{P}(U) \) as the set of all feasible sets of invariants that satisfy network topology constraints as required by the specific application (e.g., a software system may require at least one class in each connected subgraph and one or more invariants in the chosen class be included in the set.) The goal in the probing phase is to determine a close-to-optimal set of invariants \( S \in Q \) that takes into consideration two main factors: \( F(S) \), the overall detection risk of failures and \( C(S) \), the overall cost of the invariants in \( S \).

1) \( Q \): The set of feasible sets of invariants.

To select an appropriate set \( S \) of invariants for monitoring, we often expect that the selected invariants satisfy certain network topology constraints. Several typical topology constraints are described as follows: (a) **Vertex cover constraint.** The classes (vertices) corresponding to invariants in \( S \), \( \mathbb{V}_S = \{v_i| (v_i, j) \in S\} \), should be a vertex cover of the network, which means each edge of the network is incident to at least one vertex of \( \mathbb{V}_S \), where an edge is a relationship between two classes, such as association, composition, or dependency relationship. (b) **Group cover constraint.** The classes (vertices) are partitioned disjoint groups (or categories) and \( \mathbb{V}_S \) should contain at least one vertex from every group. (c) **Must-link and can-not-link constraints.** We often have prior knowledge that several vertices are redundant on the function (e.g., the vertices have the same functions in a fault-tolerant architecture.
style) and we can add a can-not-link constraint to ensure that at most one of them is selected. Similarly, several vertices are highly dependent on each other and we can add a must-link constraint that if one of them is selected, the others should also be selected. (d) **Connectivity constraint.** The subgraph induced by \( V_S \) should be connected. (e) **Density constraint.** The density (number of edges) of the subgraph inducted by \( V_S \) should be above a predefined threshold.

2) \( F(S) : \text{The overall detection risk of invariants in } S. \)

The overall detection risk, \( F(S) \), which combines type I errors (false alarm, software does not fail but at least one invariant violated) and type II errors (false negative, software fails but no invariant violation) has the general form \( R(S) \):

\[
E_{[X_S|y=0]} I\left( \sum_{(v_i,j) \in S} X_{i,j} > 1 \right) + E_{[X_S|y=1]} I\left( \sum_{(v_i,j) \in S} X_{i,j} < 1 \right),
\]

where \( E_{[X_S|y=0]}(\cdot) \) is the expectation of the input, taken over all variables in \( X_S \) with the condition that \( y = 0 \) and \( I(\text{True}) = 1 \) and \( I(\text{False}) = 0 \). The first term refers to type I error and the second term refers to type II error. Using the training data, we can measure the empirical risk \( F(S) \) as:

\[
F(S) = \frac{1}{\sum_{t=1}^{T} I(y(t) = 0)} \sum_{t=1}^{T} I( \sum_{(v_i,j) \in S} x_{i,j}^{(t)} > 1) I(y(t) = 0) + \frac{1}{\sum_{t=1}^{T} I(y(t) = 1)} \sum_{t=1}^{T} I( \sum_{(v_i,j) \in S} x_{i,j}^{(t)} < 1) I(y(t) = 1)
\]

where the first term refers to the type I empirical error and the second term refers to type II empirical error.

3) \( C(S) : \text{The overall cost of invariants in } S. \)

The selection of the cost function is dependent on the specific application. The cost function can be a modular function (e.g., \( C(S) = \sum_{(v_i,j) \in S} c_{i,j} \)), a sub-modular function (e.g., \( C(S) = 0.9^{|S|} \sum_{(v_i,j) \in S} c_{i,j} \)) that represents a 10% discount when deploying sensors in bulk due to economies of scale), a super-modular function (e.g.,
Given the input domain \( Q \) and the two objectives defined as above, the goal is to find a set of invariants \( S \) from \( Q \) that have minimal overall detection risk \( F(S) \) and minimal overall cost \( C(S) \), subject to a budget constraint on \( C(S) \): \( C(S) \leq K \). This is a multi-criterion optimization problem and the scalarization approach \(^{58}\) is commonly used to find such Pareto-optimal solutions. In particular, we optimize the following problem

\[
\min_{S \in Q} \ O(S) = F(S) + \lambda C(S) \quad \text{s.t.} \quad C(S) \leq K 
\]

(2.2)

by choosing the appropriate weight \( \lambda > 0 \). All possible Pareto-optimal solutions to the minimization of \( O(S) \) subject to the budget constraint \( |C(S)| \leq K \) can be obtained by varying the weight \( \lambda \).

Problem (3.1) is a hard combinatorial optimization problem and exhaustive search of the optimal set is usually unfeasible. We note that there is no unique form of the objective function \( O(S) \), which varies depending on its application domain. Hence, we design customized algorithms by exploring the specific structure of the problem under different situations. In particular, \( O(S) \) has three major components: the detection risk function \( F(S) \), the performance cost function \( C(S) \) and the graph topology constraints for defining the set of feasible sets of invariants \( Q \). These three components are all application dependent. We first categorize the possible forms of these components into two main groups as shown in Table 1 and then propose customized solutions for each group.

We first introduce several basic concepts related to sub-modular optimization in the context of invariant selection. A function \( g(\cdot) \) is said to be sub-modular if for any invariant \( s \in U \setminus B \) and sets \( A \subseteq B \subseteq U \), where \( U \) represents the ground set of invariants, \( g(A \cup \{s\}) - g(A) \geq g(B \cup \{s\}) - g(B) \). This is called the diminishing return property and it can be interpreted informally as: the addition of an invariant to a smaller set increases the function value more than the addition of the same element to a larger set. Sub-modular functions provide a natural modeling of coverage and

\[ C(S) = 1.1 |S| \sum_{(v_i,j) \in S} c_{i,j} \text{ that represents 10% overhead}, \text{ or a function that is neither sub-modular nor super-modular (e.g., } C(S) = 1.1 |S| \prod_{(v_i,j) \in S} c_{i,j} \). \]
diversity in real-world applications. A function \( g(\cdot) \) is said to be super-modular if it satisfies the increasing return property:

\[
g(A \cup \{s\}) - g(A) \leq g(B \cup \{s\}) - g(B), \forall s \in U \setminus B, \forall A \subseteq B \subseteq U.
\]

A function \( g(\cdot) \) is said to be modular if it satisfies the equal return property:

\[
g(A \cup \{s\}) - g(A) = g(B \cup \{s\}) - g(B), \forall s \in U \setminus B, \forall A \subseteq B \subseteq U.
\]

It can be readily devised that a modular function is both sub-modular and super-modular.

**Combinatorial search based on sub-modular optimization:** We first observe that the overall detection risk function \( F(S) \) includes two sub-components corresponding to the type I (false positive) and type II (false negative) empirical error rates. The first sub-component is a sub-modular function and the second component is a super-modular function. In a special case when all invariants do not have false alarms, the type I error rate is always zero and then \( F(S) \) becomes a super-modular function. Recall that there are three good properties about super-modularity and sub-modularity: 1) a modular function is both sub-modular and super-modular; 2) the minus of a super-modular function is sub-modular and 3) the summation of multiple sub-modular functions is still sub-modular. We can then conclude that when there are no invariants that have false alarms and the cost function \( C(S) \) is modular or sub-modular, then the whole objective function \( O(S) \) is a sub-modular function. If no network topology constraints are considered, problem (3.1) is called a sub-modular minimization problem subject to knapsack constraints and can be solved in nearly linear time with the approximation factor “\( (1 - 1/e) \approx 0.632 \)” using the greedy algorithm [66], as shown in algorithm 1. This algorithm starts from an empty set \( S = \emptyset \) and adds the element maximizing the discrete derivative \( \Delta(s|S) = O(S \cup s) - O(S) \) (Line 3 and Line 4). The approximation factor is defined as \( O(S)/O(S^*) \) where \( S \) refers to the approximated set of invariants and \( S^* \) refers to the optimal set. In other situations when some invariants have false alarms, or the cost function \( C(S) \) is sub-modular function, or network topology constraints (vertex or group cover constr-
Table 2.3: Summary of proposed solutions

<table>
<thead>
<tr>
<th>Proposed Solutions</th>
<th>Properties of the performance cost function $C(S)$</th>
<th>Network topology constraints for defining $Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algorithm 1: Combinatorial search based on sub-modular optimization.</td>
<td>Modular function, super-modular function, sub-modular function.</td>
<td>Vertex cover, group cover.</td>
</tr>
<tr>
<td>Algorithm 2: Structured convex optimization based on continuous relaxation.</td>
<td>Modular function sub-modular function non-sub-modular function</td>
<td>Vertex cover, group cover, connectivity, density must-link, can-not-link.</td>
</tr>
</tbody>
</table>

Graph-structured convex optimization based on continuous relaxation: In the above, we consider the cases where the overall risk function $F(S)$, the overall cost function $C(S)$ and the constraints are sub-modular. However, it is more often that both the type I and type II error rates are non-zero and the topological constraints are not submodular (e.g., connectivity, must-link and can-not-link). In this case, the resulting problem is neither super-modular nor sub-modular and existing sub-modular optimization techniques cannot be applied. We can instead apply convex optimization techniques to solve problems (3.1) based on continuous relaxation. In particular, we first define a vector format of $S$ as $z \in \{0, 1\}^{|U|}$ and $S = \text{supp}(z)$ as the set of nonzero entries in $z$. The two objective functions $F(S)$ and $C(S)$ can be reformulated as the functions based on $z$ as $f(z)$ and $c(z)$, respectively. The constraint $S \in Q$ can be reformulated as $\text{supp}(z) \in Q$. Problem (3.1) can then be reformulated as:

$$\min_{z \in \{0, 1\}^{|U|}, \text{supp}(z) \in Q} o(z) = f(z) + \lambda c(z), \quad s.t. \quad c(z) \leq K,$$

(2.3)

which can be approximated using graph-structured convex optimization techniques [70, 71], where "graph-structured" means the constraint $\text{supp}(z) \in Q$ is defined based on...
graph topology. In recent years, a number of advanced optimization techniques have been proposed in the machine learning community to address problems similar to problem (3.2), such as projected gradient descent [72, 73] and approximation algorithms for graph-structured compressive sensing [71, 74]. Algorithm 2 shows the basic steps of the projected gradient descent approach for solving problem (3.2). In particular, \( \nabla f(z) \) refers to the gradient of \( f(z) \) with respect to \( z \) and \( \nabla c(z) \) is defined similarly.

The vector \( \nabla f(z^{(i)}) + \lambda \cdot \nabla c(z^{(i)}) \) calculated in Line 4 is the gradient of the overall objective function \( o(z) \). The vector \( b^{(i)} \) is the gradient descent update, which is the same as the update of the standard gradient descent algorithm and \( \lambda \) refers to the step size. As \( b^{(i)} \) is often not within the graph-structured domain \( \mathcal{Q} \), line 5 is to project \( b^{(i)} \) to the input domain \( \mathcal{Q} \) based on the following projection operator:

\[
T(b^{(i)}) = \min_{z \in \{0,1\}^{\lvert \text{supp}(z) \rvert}, \text{supp}(z) \in \mathcal{Q}} \| z - b^{(i)} \|_2^2 \quad \text{s.t.} \quad c(z) \leq K, \tag{2.4}
\]

where \( \text{supp}(z) \) refers to the set of non-zero entries in \( z \).

---

**Algorithm 1** A submodular algorithm for invariant selection

**Input:** Network \( G \) and the ground set of invariants \( U \).

**Output:** \( S \)

1. \( S = \emptyset \)
2. **while** \( C(S) < K \) **do**
3. \( s = \arg \max_{s \in U} O(S) - O(S \cup s) \quad \text{s.t.} \quad S \cup s \in \mathcal{Q} \)
4. \( S = S \cup \{s\} \)
5. **end while**

---

**Algorithm 2** A projected gradient descent algorithm for invariant selection

**Input:** Network \( G \) and the ground set of invariants \( U \).

**Output:** \( S \)

1. \( i = 0 \)
2. \( z^{(i)} = 0 \)
3. **repeat**
4. \( b^{(i)} = z^{(i)} - \left( \nabla f(z^{(i)}) + \lambda \cdot \nabla c(z^{(i)}) \right) \)
5. \( z^{(i+1)} = T(b^{(i)}) \)
6. \( i = i + 1 \)
7. **until** Convergence
8. \( S = \text{supp}(z^{(i)}) \)
If the above projection operator is solvable, algorithm 2 is guaranteed to converge after a finite number of steps and return a local optimal solution of problem (3.2). However, for network topology constraints, such as connectivity, density, must-link and can-not-link constraints, the projection problem (3.3) is NP-hard. In this case, algorithm 2 does not provide good quality guarantees about the returned solution, if we conduct approximations for the projection problem (3.3). More advanced optimization algorithms such as approximation algorithms for graph-structured compressive sensing [71, 74] can be adapted to address this case.

### 2.3.4 The Monitoring Phase

The online monitoring phase is to observe the software runtime behavior by comparing the values of the invariants selected during the execution. This phase focuses on how to instrument the monitors to minimize the execution overhead and how to update these monitors when a false positive or false negative is encountered. To minimize the execution overhead, it is important to compare the values of the monitored invariants at runtime efficiently. Existing approaches such as DIDUCE [5] and Javalanche [75] check the violations of the invariants at runtime by mapping each checked expression to an integer and insert monitors into bytecode to check for invariant violations before and after a method call. In our approach, we instrument each monitored invariant with a guard that reviews the value of the invariant when the execution reaches the corresponding program point.

Nguyen et al. [76] suggest that if the number of false positives is too high (30% or more), then the cost of analyzing false positives may be more than that for online monitoring. To reduce false positives and false negatives, we use a monitor controller to dynamically update the monitors (the expected values of the invariants) after occurrence of false positives or negatives.
2.4 Empirical Studies

To evaluate the validity of the proposed approach, we have developed a prototype software that implements the proposed algorithms and integrates with Daikon to collect and select anomaly-revealing invariants. We conducted two empirical studies. The first study was conducted in a controlled environment where two subject programs were selected from Defects4J [77], which is a database of real faults for real-world programs and two open source programs were chosen from the Apache Software Foundation. The second study was conducted on an industrial application that has been used for many years and there is no any known code defect in the system. However, during its executions, anomalies caused by the execution context, such as delayed or loss of messages, may be encountered.

For both empirical studies, we took a two-step approach. The first part was the modeling process, which used Daikon to collect a set of anomaly-revealing invariants, applied two selection strategies to select a subset from this set of invariants, then instrumented monitors to these selected invariants. The second part was the validation phase, which evaluated the effectiveness of the two selection strategies, where we executed the instrumented programs on a set of test cases. In this phase, when an anomaly or failure was encountered, this anomaly/failure was considered to be detected if there was at least one violation of the monitored invariant. Also this invariant must not have been violated in the fixed version of the program, thus making the violation relevant to the anomaly. Otherwise, if there was no reported violation, it would result in a false negative. On the other hand, if a violation was reported while the program was successfully (normally) executed, then it was a false positive.

In these studies, we considered the empirical detection risk function \( F(S) \) and the overhead cost function \( C(S) \) and developed two efficient algorithms, including Algorithm 1 (a submodular algorithm for invariant selection) and Algorithm 2 (a projected gradient descent algorithm for invariant selection), to approximately solve the following problem:

\[
\min_{S \in Q} O(S) := F(S) + \lambda C(S) \quad s.t. \quad C(S) \leq K,
\]
where $K$ is a budget on the overhead cost and $\lambda$ is a tradeoff parameter. Algorithm 1 is a linear-time approximation algorithm that has been shown to be successful in sub-modular optimization problems and provides rigorous guarantees as discussed in Section 2.3.3. This algorithm starts from an empty set and adds invariants one by one in a greedy manner based on the marginal gain on the objective function, until the budget constraint is violated. Algorithm 2 solves a relaxed numerical optimization problem:

$$\min_{z \in [0,1]^{|\mathcal{U}|}} f(z) + \lambda c(z) \quad s.t. \quad c(z) \leq K,$$

where $f(z)$ and $c(z)$ are vector forms of the functions $F(S)$ and $C(S)$, respectively, as discussed in Section 2.3.3. Algorithm 1 has limitations in that it cannot support popular topological constraints, such as vertex cover and path cover. In this study, we only consider vertex cover as proof of concept about the usefulness of topological constraints. Algorithm 2 is based on the framework of projected gradient descent and aims to solve the above problem subject to the additional constraint that the selected invariants should cover all the leaf vertices of the network. In Line 4, Algorithm 2 conducts a standard gradient descent update $b^{(i)} = z_i - (\nabla r(z_i) + \lambda \cdot \nabla c(z_i))$ as used in standard gradient descent optimization. As the updated solution $b^{(i)}$ may not be a feasible solution of the above problem, Algorithm 2 conducts a projection process to ensure that the resulting solution is feasible.

$$T(b^{(i)}) = \min_{z \in \{0,1\}^{|\mathcal{U}|}, \text{supp}(z) \in \mathcal{Q}} \|z - b^{(i)}\|_2^2 \quad s.t. \quad c(z) \leq K$$

where supp$(z)$ refers to the set of non-zero entries (indices of selected invariants) in $z$ and $\mathcal{Q}$ refers to all feasible subsets of invariants that cover the leaf vertices of the network. Algorithm 2 iterates the previous steps until convergence. Two types of network graphs were used in Algorithm 2: Algorithm 2a used class interaction in the network graph, where a class is a vertex and a class relation (including generalization, realization, association, nesting and dependency) is an edge. Algorithm 2b used function interaction in the network graph, where a function (method) is a vertex and a function call relation is an edge.
2.4.1 Empirical Study I

We selected Joda-Time and Closure Compiler from Defects4J and chose Commons-Math and Commons-Collections from the Apache Software Foundation. Defects4J provides multiple versions of each program through its development cycle and each version contains one reported fault and may have unknown faults and it also provides the fixed version for the fault. The only difference between both fixed and non-fixed programs is this reported fault; there might be some unknown faults in both programs. It also provides a test suite associated with each version, which contains one fault-triggering test case that produces anomaly execution. We selected a set of earlier reported faults for the modeling phase and some later faults for the validation phase. These two sets of faults are disjoint. The versions of the program used in the two phases were slightly different. The one used in the validation phase is newer than the one used in the modeling phase and may contain new features or remove some features from the previous one. The Apache Software Foundation does not provide a fixed version for each fault, so we selected the faults in the repository and fixed them one by one. The criterion for selecting the faults is to keep the changes to the subject software as small as possible.

Joda-Time is the a standard date and time library for Java; it replaces Java Date and Time classes. In the Joda-Time experiment, we used 18 faults and 4,062 test cases in the modeling phase, 9 faults and 3,872 test cases in the validation phase. The Closure Compiler is a tool for making JavaScript download and run faster. Instead of compiling from a source language to machine code, it compiles from a JavaScript to a better one. We used 35 faults, 7,593 test cases in the modeling phase and 15 faults and 7,435 test cases for the validation phase. Commons-Collections is a Java-based framework that provides many powerful data structures that accelerate development of most significant Java applications and seek to build upon the JDK classes by providing new interfaces, implementations and utilities. We used Collections 4.0 and selected 17 faults from its fault repository. In the modeling phase, 12 faults and 13,849 test cases were used and in the validation phase, 5 faults and 14,397 test cases were used. Commons-Math is a library of lightweight, self-contained mathematics and statistics components addressing the most common problems not available in the
Java programming language or Commons Lang. We used Math 3.0 and selected 23 faults from its repository. In the modeling phase 16 faults and 3,519 test cases were used, and in the validation phase, 7 faults and 3,565 test cases were used. Table 2.5 shows the information of these programs.

The steps of this study were conducted as the following:

Step 1: Obtaining anomaly-revealing invariants: Daikon’s Chicory front end was used to instrument the programs on both versions (faulty and fixed). For each selected fault in the modeling phase we used JUnit to execute test cases provided by the two repositories on the instrumented fixed program (no known faults) and obtained the invariants. Next we ran the test suite on the faulty program to obtain another set of invariants. The two sets of invariants for each program were compared and the invariants that have different values in the two sets were considered anomaly-revealing. This process was repeated for all the selected faults.

Step 2: Creating network graphs: We used ObjectAid [78] to get class interaction graph and Java-callgraph [79] to create function interaction graph. The priority for each invariant was set to 1.

Step 3: Selecting anomaly-revealing invariants: The anomaly-revealing invariants identified at Step 1 were used as the input to execute Algorithm 1 and Algorithm 2a and 2b. The algorithms select a subset of essential anomaly-revealing invariants that maximize the overall detection risk and minimize the overall cost of these invariants, subject to the budget size approximately $K = 5\%$ of the average execution time. Algorithm 2a and 2b are subject to the additional class/function (vertex) coverage constraint and the budget constraint $K$. Both type I and type II errors were considered to define the overall detection risk function. The overall cost of these invariants is defined as the summation of the cost of each invariant that is set to 1. The vertex coverage constraint is defined as the constraint that there should be at least one invariant selected for each vertex. If a vertex does not have any anomaly-revealing invariant, the invariants are selected in a greedy manner based on their frequency of appearance in the training cases. This process was repeated ten times and the result was obtained from the average of the ten results.

Step 4: Instrumenting the monitors in the selected invariants and executing
the instrumented programs in the validation phase. For each failure execution, we investigated the monitored invariants and checked their values.

Step 5: Measuring overhead: We used a java function to record the timestamps of the starting and ending times of each execution. We executed both monitored and non-monitored versions ten times and recorded the average execution time.

Step 6: Computing Precision, Recall and F-measure, where Precision is computed as the ratio of the reported anomalies that are true anomalies over the total number of reported anomalies. Recall is computed as the ratio of the reported anomalies that are true anomalies over the total number of true anomalies in the validation phase. F-measure = \(2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}\). The results are summarized in Table 2.4.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Time</th>
<th>Closure</th>
<th>Collections</th>
<th>Math</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Classes</td>
<td>195</td>
<td>553</td>
<td>271</td>
<td>564</td>
</tr>
<tr>
<td>No. of Functions</td>
<td>2,820</td>
<td>5,981</td>
<td>1,470</td>
<td>3,882</td>
</tr>
<tr>
<td>No. of Statements</td>
<td>139,898</td>
<td>251,813</td>
<td>35,075</td>
<td>66,561</td>
</tr>
<tr>
<td>No. of Test Cases</td>
<td>3,872</td>
<td>7,435</td>
<td>14,397</td>
<td>3,565</td>
</tr>
<tr>
<td>Class Coverage</td>
<td>97%</td>
<td>90%</td>
<td>79%</td>
<td>94%</td>
</tr>
<tr>
<td>Function Coverage</td>
<td>91%</td>
<td>90%</td>
<td>74%</td>
<td>88%</td>
</tr>
<tr>
<td>Statement Coverage</td>
<td>90%</td>
<td>90%</td>
<td>89%</td>
<td>89%</td>
</tr>
<tr>
<td>No. of Invariants</td>
<td>245,991</td>
<td>1,039,786</td>
<td>22,394</td>
<td>226,614</td>
</tr>
</tbody>
</table>

**Table 2.4: Software used for empirical study I.**

Data Analysis: [Joda-Time:] In the modeling phase, 18 faults were used and 1,992 anomaly-revealing invariants were identified and among them 490 invariants were selected from Algorithm 1, Algorithm 2a and Algorithm 2b. In the validation phase, nine different faults were used and Algorithm 1 detected an average of five faults, Algorithm 2a detected an average of 4.1 faults and Algorithm 2b detected an average of 2.0 faults. The invariants selected by Algorithm 2b were unable to
detect seven of the nine faults because the missing faults’ the maximum percentage of affected classes (the classes that have at least one violated invariant) is only 1.3% and the maximum percentage of affected functions (the functions that have at least one violated invariant) is only .15%. Thus, these faults had very little impact on the program. The two faults that were detected have at least 3% of affected classes and 0.3% of affected functions. The invariants selected by Algorithm 2b only cover 17% of the functions. Figure 2.2 and figure 2.3 show details. Because both the number of functions with monitored invariants and the number of affected functions are very small, it is unlikely that the violated invariants in these affected functions would be detected by Algorithm 2b and thus, the fault detection rate is small. The results are summarized in Table 2.5.

Figure 2.2: Time project’s class impact coverage  
Figure 2.3: Time project’s method impact coverage
Table 2.5: Results of empirical study I -TIME

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Algo 1</th>
<th>Algo 2a</th>
<th>Algo 2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anomalies in modeling phase</td>
<td>18</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>Anomaly-revealing invariants</td>
<td>1,992</td>
<td>1,992</td>
<td>1,992</td>
</tr>
<tr>
<td>Invariants selected</td>
<td>490</td>
<td>490</td>
<td>490</td>
</tr>
<tr>
<td>Anomalies in validation phase</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>No. of test cases</td>
<td>3,872</td>
<td>3,872</td>
<td>3,872</td>
</tr>
<tr>
<td>Anomalies detected</td>
<td>5</td>
<td>4.1</td>
<td>2</td>
</tr>
<tr>
<td>False negative</td>
<td>4</td>
<td>4.9</td>
<td>7</td>
</tr>
<tr>
<td>False positive</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Recall</td>
<td>0.56</td>
<td>0.46</td>
<td>0.22</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1.1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.72</td>
<td>0.63</td>
<td>0.7</td>
</tr>
<tr>
<td>Avg. exec time</td>
<td>1277ms</td>
<td>1277ms</td>
<td>1277ms</td>
</tr>
<tr>
<td>Avg. exec time (monitored)</td>
<td>1340.8ms</td>
<td>1340.8ms</td>
<td>1340.8ms</td>
</tr>
<tr>
<td>Overhead</td>
<td>4.99%</td>
<td>4.99%</td>
<td>4.99%</td>
</tr>
</tbody>
</table>

[Closure:] In the modeling phase, 35 faults were used and 319 anomaly-revealing invariants were reported. 460 invariants were selected by the two algorithms. In the validation phase, 15 different faults were used; Algorithm 1 and Algorithm 2a detected all the faults and Algorithm 2b detected an average of 14.8 faults. The faults, used in the modeling phase, contain sufficient coverage of affected classes (4.37%) and there was an even distribution of anomaly-revealing invariants among the classes and functions. Figure 2.4 and figure 2.5 show details. The faults in the modeling phase have good coverage of affected classes and functions and the anomaly-revealing invariants were distributed in the classes and functions evenly. Therefore,
in the validation phase, faults can be easily detected by at least one of the selected invariants. The results are summarized in Table 2.6.

Figure 2.4: Closure project’s class impact coverage

Figure 2.5: Closure project’s method impact coverage
Table 2.6: Results of empirical study I-Closure.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Algo 1</th>
<th>Algo 2a</th>
<th>Algo 2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anomalies in modeling phase</td>
<td>35</td>
<td>35</td>
<td>35</td>
</tr>
<tr>
<td>Anomaly-revealing invariants</td>
<td>319</td>
<td>319</td>
<td>319</td>
</tr>
<tr>
<td>Invariants selected</td>
<td>460</td>
<td>460</td>
<td>460</td>
</tr>
<tr>
<td>Anomalies in validation phase</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>No. of test cases</td>
<td>7,435</td>
<td>7,435</td>
<td>7,435</td>
</tr>
<tr>
<td>Anomalies detected</td>
<td>15</td>
<td>15</td>
<td>14.8</td>
</tr>
<tr>
<td>False negative</td>
<td>0</td>
<td>0</td>
<td>0.2</td>
</tr>
<tr>
<td>False positive</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Recall</td>
<td>1</td>
<td>1</td>
<td>0.99</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>1</td>
<td>1</td>
<td>0.99</td>
</tr>
<tr>
<td>Avg. exec time</td>
<td>13,236ms</td>
<td>13,236ms</td>
<td>13,236ms</td>
</tr>
<tr>
<td>Avg. exec time (monitored)</td>
<td>13,947ms</td>
<td>13,947ms</td>
<td>13,947ms</td>
</tr>
<tr>
<td>Overhead</td>
<td>5.37%</td>
<td>5.37%</td>
<td>5.21%</td>
</tr>
</tbody>
</table>

Collections: In the modeling phase, 12 faults were used and 7,544 anomaly-revealing invariants were identified and 600 invariants were selected. In the validation phase, five faults were used, with Algorithm 1 detecting an average of 3.0 faults, Algorithm 2a detecting an average of 3.25 faults and Algorithm 2b detecting an average of 3.57 faults. The reasons that some faults were not detected are: (1) Insufficient test coverage: we used 14,396 test cases that covered 79% of classes, 74% of functions and 89% of statements. There were 57 classes, 382 functions and 3,858 statements that were not executed during the training phase; thus, no invariant was selected from them. (2) The undetected faults have very low impact on the
program, as they only affected 1% of the classes on average. (3) The faults used in the modeling phase were not influential: not only was the number of faults too small, but also the impact of these faults on the program was not significant. This caused the set of anomaly-revealing invariants to be inadequate and the selected invariants to be ineffective. Figure 2.6 and figure 2.7 show details. The results are summarized in Table 2.7.

Figure 2.6: collection project’s class impact coverage
Figure 2.7: collection project’s method impact coverage
Table 2.7: Results of empirical study I - Collections.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Algo 1</th>
<th>Algo 2a</th>
<th>Algo 2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anomalies in modeling phase</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Anomaly-revealing invariants</td>
<td>7,544</td>
<td>7,544</td>
<td>7,544</td>
</tr>
<tr>
<td>Invariants selected</td>
<td>600</td>
<td>600</td>
<td>600</td>
</tr>
<tr>
<td>Anomalies in validation phase</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>No. of test cases</td>
<td>1,397</td>
<td>14,397</td>
<td>14,397</td>
</tr>
<tr>
<td>Anomalies detected</td>
<td>3</td>
<td>3.25</td>
<td>3.57</td>
</tr>
<tr>
<td>False negative</td>
<td>2</td>
<td>1.75</td>
<td>1.43</td>
</tr>
<tr>
<td>False positive</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Recall</td>
<td>0.6</td>
<td>0.65</td>
<td>0.71</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.75</td>
<td>0.79</td>
<td>0.833</td>
</tr>
<tr>
<td>Avg. exec time</td>
<td>18,668</td>
<td>18,668</td>
<td>18,668</td>
</tr>
<tr>
<td>Avg. exec time (monitored)</td>
<td>18,564ms</td>
<td>18,569ms</td>
<td>18,576ms</td>
</tr>
<tr>
<td>Overhead</td>
<td>5.21%</td>
<td>5.24%</td>
<td>5.28%</td>
</tr>
</tbody>
</table>

[Math:] In the modeling phase, 16 faults were used and 11,259 anomaly-revealing invariants were identified and 1,770 invariants were selected. In the validation phase, seven faults were used. Both Algorithm 1 and Algorithm 2a detected all seven faults and Algorithm 2b detected six faults on average. Although the number of faults used in the modeling phase is not large, each fault carries significant influence, generating a large number of anomaly-revealing invariants evenly distributed in the program. The faults in the validation phase also have a large number of affected classes and functions, resulting in the high fault detection rate. Figure 2.8 and figure 2.9 show details. The results are summarized in Table 2.8.
Figure 2.8: Math project’s class impact coverage

Figure 2.9: Math project’s method impact coverage

Table 2.8: Results of empirical study I - Math.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Algo 1</th>
<th>Algo 2a</th>
<th>Algo 2b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anomalies in modeling phase</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Anomaly-revealing invariants</td>
<td>11,259</td>
<td>11,259</td>
<td>11,259</td>
</tr>
<tr>
<td>Invariants selected</td>
<td>1770</td>
<td>1770</td>
<td>1770</td>
</tr>
<tr>
<td>Anomalies in validation phase</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>No. of test cases</td>
<td>3,565</td>
<td>3,565</td>
<td>3,565</td>
</tr>
<tr>
<td>Anomalies detected</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>False negative</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>False positive</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Recall</td>
<td>1</td>
<td>1</td>
<td>0.8671</td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F-measure</td>
<td>1</td>
<td>1</td>
<td>0.9231</td>
</tr>
<tr>
<td>Avg. exec time</td>
<td>136,983</td>
<td>136,983</td>
<td>136,983</td>
</tr>
<tr>
<td>Avg. exec time (monitored)</td>
<td>143,174ms</td>
<td>142,914ms</td>
<td>143,038ms</td>
</tr>
<tr>
<td>Overhead</td>
<td>4.52%</td>
<td>4.33%</td>
<td>4.42%</td>
</tr>
</tbody>
</table>
2.4.2 Discussion

In summary, the effectiveness of the proposed approach strongly depends on the quality of the training data. In order to obtain strong results, the training data should provide an adequate test suite that contains sufficient faulty test cases such that each execution trace at runtime will contain at least one monitored invariant. The adequacy of the training data can be determined by the number of anomaly-revealing invariants and the distribution of these invariants in the program. The faults in the validation phase can also affect the outcome. If (1) the types of the faults in the modeling and the validation phases are similar, (2) the numbers of affected classes and functions are sufficient (at least 5%) and (3) there exists some overlapping of the affected classes/functions between the two phases, then there will potentially be a high detection rate.
2.4.3 Empirical Study II

The second study was conducted on one of the core components of Commerce-Hub, Batch Agent, that provides cloud-based technologies and services enabling retailers to radically expand their product offering without inventory risk. Batch Agent controls the scheduling, batching and distribution of workloads in the batch processing system. It reads work that needs to be done (requests, scheduled requests, etc.) and places it into containers (batches) to group them for distribution to workers as a unit via a batch channel. Batch Agent is a stable system which has been used for more than 10 years and is the core component of the batch processing system. The system is an event-driven real time system which is a multi-threaded Java application consisting of 141 classes, 964 methods and 18,161 LOC. It adopts AKKA framework that supports concurrent, fault tolerant and scalable implementation using an Actor model. Batch Agent implements nine actors, each of which is initiated by a thread at startup; it uses HornetQ as the message bus to communicate with the worker reporting system and stores persistent data in a Microsoft SQL server. Most of the concurrency controls used to avoid race and deadlock conditions are done by a relationship locking mechanism to coordinate the work assignments. The runtime anomalies that have been encountered are mainly caused by delay or loss of messages, invalid inputs, anomalies in the execution context such as database update error, lost connection to message queues and unexpected work disruptions such as workers failing to complete the assigned request for various reasons.

In the training phase, we used state machines and Petri Nets as the test models. The state machines for the actors depict the events and actions of the actor and the Petri Nets describe the concurrency nature of the system. Based on these models. The figure 2.10 is the state machine graph of the Batch Agent system. The red line shows the abnormal transitions. For example BatchRouter recieve a message and form a mutablebatch to status5, if the request is null then put this request into blacklist. Blacklist contains all the invalid request. The figure 2.11 is the Petri Nets graph of the Batch Agent system. Similar to the state machine graph the red line shows shows the abnormal transitions. Created test cases and used state and transition as the coverage criteria for the state machines and state-transition-trace coverage for Petri
Nets [80], where a scenario begins at an initial place and ends at a terminal place. To cover all the normal states and transitions and traces, 24 test cases were created. To cover abnormal states and scenarios, 26 test cases were created; for example, a test case for a scenario where a worker fails to acknowledge the status of the work, which triggers a timeout event and sends a “failed to dispatch” message. We used Daikon to
augment the execution of these test cases to collect a set of likely dynamic invariants and their values under these test runs and 10,114 invariants were reported, among which, 3,853 invariants were violated in the abnormal tests. In the probing phase, we created two network graphs, class interaction graph and function interaction graph and used these network graphs to select a set of anomaly-revealing invariants that have the most coverage of the runtime executions. Leveraging these network graphs maximizes the coverage of the runtime executions under the budget constraint. We set the budget constraint to be 1% of the number of the invariants in the algorithms. In addition, some classes perform more complex or critical business logic which is likely to be fault prone; we specified these classes on the network graphs such that at least one invariant will be selected from each of these classes. For example, in many cases, a software anomaly is due to a lost of connection between the software and its execution context. Thus, the classes coupled to the external entities will need to be monitored. The decision of these classes was made based on the domain knowledge and by applying design metrics to select fault-prone classes [28]. Algorithm 1 selected 35 invariants, Algorithm 2a used the class interaction graph and selected 41 invariants and Algorithm 2b used the function interaction graph and selected 46 invariants. In the monitoring phase, we used the system’s existing logging system to place the selected invariants, where the values of the invariants of each run were compared with their expected values and an alarm was triggered if the values are different. The overheads of monitoring the selected invariants are around the 1% range, which is almost not noticeable.

In the validation phase, we checked the submitted production incidence tickets. There were 11 tickets and two of them were a result of an sql logic issue, which are situations where open and closed batches are not taken by the system’s recoverer from the database and are resolved by a modified sql statement. In these cases, invariants were unable to detect the anomalies as they occurred outside of the program logic, which is beyond the scope of this paper, so we only considered the other nine tickets. We were able to reproduce eight out of the nine tickets. The ninth ticket is a complex situation caused by multiple issues and we were not able to reproduce the scenario. The invariants selected by Algorithm 1 and Algorithm 2a were able to detect seven anomalies; with the help of the function interaction graph, the selected invariants in
Algorithm 2b detected all eight anomalies. The anomaly-revealing invariants selected by the three algorithms are 7, 7 and 8, respectively. The results are summarized in Table 2.4.

### 2.4.4 Discussion

In summary, the results of these empirical studies suggest that (1) program invariants can be used to detect program anomalous behaviors. We observed that in every failure or abnormal execution, there exists at least one invariant violation. (2) The value of the precision was 1 for all the experiments, which means that all the reported violations are true violations. (3) An average of 76.5% of the anomalies in both studies were detected by the monitored invariants with no more than 5.5% of execution overhead. These results are promising and suggest the effectiveness of the proposed approach.
CHAPTER 3
CONTEXT-AWARE REGRESSION TESTING

3.1 Overview

Regression Testing focuses on the impact of software modification. The tests used during the regression testing phase ensure that changes do not introduce regression faults, which may cause previously working features to fail.

Regression testing is performed after some code changes or execution context changes (library update, database change), which will re-run previous test cases on the modified program. Re-running all test cases is the easiest way to conduct a regression testing, but it can be time-consuming when there are too many test cases or changes are committed frequently. Many research studies have proposed selecting a small number of test cases for the regression testing. These studies can be categorized into three groups: test suite minimization, test case prioritization, test case selection.

Test minimization aims to get a minimum set of test cases that covers all the test requirements by focusing on removing redundant tests. For example, if both test A and test B execute the test requirement C, only one of them needs to be retested. In this case, the regression test can improve its performance, but this method may overlook some regression faults.

Test case prioritization aims to rank the test cases based on a specific rule and then based on the program requirements to select the top N test cases for retesting. Prioritization may not be safe as it may also miss some test cases that are low in the rank but have a regression fault.

Test case selection aims to select test cases to cover the required criterion. A typical criterion is selecting modification-traversing test cases which execute the modified part of the program. The test case selection technique is safe as it will select all the test cases that traverse the modified program.

We propose to use test case selection technique for safety purposes and adopt
test case prioritization as a means to identify the regression fault quickly. The root cause of a regression fault can be a code defect or development environment changes. The regression fault that is caused by the modified part of the program can be detected easily by selecting test cases that execute the modified program. However, detecting those caused by the environment changes can be challenging. Many studies have proposed techniques that focus on the selection of modification-traversing test cases. Some regression faults are not directly related to the modified program but they are introduced through dependencies on the modification. For instance, if a modified Program P changes the data stored in a database that is used by function F, resulting in a different output produced by F. Selecting the modification-traversing tests only miss the test regarding the function F, because function F does not have any changes.

Execution context change is difficult to find and to test the correctness after the change, as the relationship between the execution context and the program is not explicit. It requires an extra layer to reveal the relationship between the test cases and the execution context. Using the database as an example, in a Java application, we usually have an entity object that represents a table in the database, and the objects attributes denote columns in a database table. These mappings are usually established within the configuration setting or annotations. By analyzing the configuration settings and annotations, we can map the Java entity object into a database table. Thus, we can figure out the impact of changing a database table. The Java entity object can be one kind of program invariant. When the database changes which will cause java entity object change, Any test case that contains a program invariant related to the Java entity object should be selected for regression testing.

It is a challenge to have a single solution that can cover the regression fault induced by modification of the code and execution context, such as database changes and library changes. We propose to use program invariants as a means to associate relationships between the program, the execution context, and the test cases.

3.2 Existing approaches and limitations

Techniques for regression testing have been well studied. Existing approaches can be classified into test case minimization, test case prioritization, and test case
selection. The minimization \[6, 81, 82, 83\] and the prioritization approaches \[84, 29, 85\] aim at selecting a minimum or high priority subset from the test suite to cover the critical parts of the program that need to be retested. They first identify portions of a program that should be retested and then select some tests that will exercise these parts or satisfy certain criteria. This method can reduce the number of test cases to be retested but they may overlook some fault-revealing test cases \[8\]. The selection approaches \[86, 87, 88, 6, 89, 8\] on the other hand, will select all the fault-revealing test cases, thus guaranteeing the quality of the software. However, it requires a longer processing time and a larger size of the retest test suite in general.

Execution, dynamic and relevance slicing \[87, 90\] are typical slicing methods, among which execution and relevance slicing are safe while dynamic slicing is not. Execution slicing regression testing will only retest the test cases which execute the modified codes. It has been adopted in many regression testing tools due to its simplicity and efficiency. The execution slicing technique has many variations; it can be based on the statement, block, or function level. Regardless of which, precision is a major problem that is difficult to overcome. Wong et al. \[91\] proposed a methodology that takes the coverage information into account or assigns priorities to different test cases, so that a smaller subset of the test cases, based on these criteria, can be selected.

Dynamic and relevance slicing only retest the test cases in which the behavior of the system is affected by a modification. Dynamic slicing will only consider the non-predicate statements which affect the output, whereas relevance slicing considers all the predicate statements which will affect the output. Dynamic slicing is more precise than the others, but in most cases, it is not as safe.

Rothermel and Harrold \[92\], and Ball \[88\] provide a regression approach for both procedure programs. They use the control flow graph (CFG) to represent a program. After modification, the CFGs of the program before and after the modification is compared. It traverses the two CFGs and compares pairs of nodes in the two graphs. Starting with the two root nodes, it compares the successors of the pair of the nodes with identically labeled edges. If the two successor nodes are not equivalent (have different labels), then the tests that exercise this edge are considered
modification-traversing. They propose a group of methods that can be classified into basic algorithms and algorithms with added precision. The basic algorithm is similar to execution slicing, and the algorithms with added precision also consider the effects of the modifications on the output of the system.

The firewall approaches [93, 89, 86, 94, 95], which identify the set of modules that may be affected by the modification, perform unit testing for the modified modules and integration testing on the set of modules that have been selected. Rosenblum [96] developed a tool, TestTube, which identifies modified/affected entities (functions, types, variables and macros). Then it retests all the test cases that include at least one affected entity.

Orso et al. [97] proposed a partition-based approach for Java programs. Their approach constructs Inter-class Relation Graphs (IRG) for the program before and after a modification. The IRG is similar to a class diagram that contains hierarchy, aggregation, and the relation between classes and interfaces. The data-flow-based approaches [33, 34, 12] use dataflow information to augment Control Flow Graph by adding global variables, function parameters, and a return value of functions. The configuration-based approach [98, 99, 100] aims to select test cases to execute the code in the new configuration that is not tested in the old one. A configurable system uses different parts of the system based on a predefined set of configurations. When a configuration is changed, it compares the differences between the two configurations and uses program slicing to determine which functions are affected (not covered by the old one) by the change of the configuration. The test cases that exercise an affected function are selected for regression testing.

For non-code based changes, Haraty et al. [30] presented an approach for regression testing SQL-based database applications. They built control flow graphs for database modules consisting of SQL compound statements and performed dataflow analysis on database columns. Based on this information, they conducted a two-phase methodology. Phase 1 identified modifications and performed a change impact analysis. Phase 2 used two algorithms, Graph walk algorithm and Call graph firewall algorithm, for regression test selection. The Graph Walk algorithm traverses the control flow graph of database modules and selects a safe set of test cases to
retest. The Call Graph Firewall algorithm uses dataflow analysis to build a firewall at the inter-procedural level, which includes the modified database components in the firewall.

Nanda et al. [31] presented an approach to address regression test selection when there is no code change but property files or databases are changed. They first built abstract models for configuration files and databases and compared the models before and after the changes to identify modified entities. Then, they monitored each execution to build a traceability matrix that links the test cases to the accessed property and the issued database commands. Based on the modified entities and the traceability matrix, they selected tests that traverse the modified entities to retest.

Gligoric et al. [101] proposed a lightweight regression test selection approach and a tool, EKSTAZI, to reduce the end-to-end regression testing time. It tracks dynamic dependencies of tests on files and uses checksum to determine the changed elements and their impact on the tests at the file level. Zhang [102] further improved the effectiveness of EKSTAZI and proposed a hybrid approach. The basic HyRTS analyzes the change impact at the method level and performs selection at the file level. Two variants of HyRTS were introduced, one used the basic block level of analysis, and the other transformed the instance method additions and deletions into file-level changes.

Table 3.1 summarizes the limitations of each approach. Execution, dynamic, relevance slicing CFGs, FireWall, UML, and Partition(IRG) are mainly based on the code changes. These approaches are not able to perform the changes in the library, configuration files, database, or API. Meanwhile, the regression testing on SQL-based database applications and approach proposed by Nanda seek to solve non-code changes. Both of them can work well on configuration file change and database change. However, they are not able to perform on the code changes. It may be tedious to use one technique from code change based and one from non-code based changes to cover both changes for the safety. EKSTAZi and HyRTS are based on file-level changes. Their approaches can perform on code change, library change, configuration files changes, but their approach will select a large number of test cases, which may not be feasible in a large project with a large volume of test cases. Besides,
these two techniques do not cover a database change and API change. Most of those approaches work very well on current types of changes, but none of them can cover it all. The goal of our CART approach is to be a safe regression test selection technique that covers all of those changes.

<table>
<thead>
<tr>
<th></th>
<th>Code Change</th>
<th>Library Change</th>
<th>Configuration File change</th>
<th>Database Change</th>
<th>API Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slicing</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>CFGs</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>FireWall</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>UML</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Partition(IRG)</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SQL-Based</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>Nanda</td>
<td>NO</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>EKSTAZI</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>HyRTS</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>CART</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>
3.3 Methodology

3.3.1 Objective

The goal for a safe selective regression test is to minimize the cost while maintaining the effect of retesting all test cases [6, 7, 8, 9]. Many researches focus on finding the modification-traversing test cases. These approaches work well when the changes are made to the code. When there are changes to the execution context, such as the library or database, these approaches may not be safe. Testing execution context changes has been well studied over the years [30, 31, 101, 102], but these approaches can only cover a small part of execution context changes.

Execution contexts support program executions which include libraries, external APIs, configuration files and databases. A correct execution of a program depends on the communication with its execution context. When a library is updated, the response from the library changes. Then the program will get an unexpected input from the library and will produce an incorrect output. The majority of modern software relies heavily on the support of the execution context. A regression testing technique that covers all of these types of changes has not been explored by the existing approaches.

To address these issues, our solution should be able to:

1. Safely identify all fault-revealing test cases.
2. Account for changes in both the execution context and the program.
3. Be safe and effective as the existing approaches for testing program changes.

A fault-revealing test case is the test case that executes a faulty program and fails in the testing. Different from modification-traversing tests in which test case that executes modified programs may or may not fail.

We use program invariants to determine if a function is affected by the change and if the affect function will cause regression faults. A program invariant includes a class/object attribute, global variables, elements defined in a property/configuration...
file, input parameters, return values, or entity objects corresponding to database entities. Any changes among these will be caught by the program invariant.

Our approach is a regression test selection approach, named Context-Aware Regression Testing (CART), that accounts for changes in both the code and the execution context. CART selects tests that execute functions affected by the changes to the program, upgrade/downgrade of the library, database changes, or configuration file changes.

Our approach can support prioritized selection techniques. CART selects all the tests that execute the modified or affected functions. When the time for regression testing is limited and not all of the test cases selected by the safe approach can be executed, then we formulate a multi-objective function striving to obtain an optimal solution to maximize the fault detection rate under a time constraint. This approach focuses on selecting the tests impacted most by the changes in the program, upgrade/downgrade of the library, the changes in databases or configuration files.

3.3.2 Using program invariants to detect changes

The basic unit of the investigation in our approach is the function. We use a program invariant before and after an invocation of a function to determine if the function is affected by the change and can potentially have regression faults. The affected functions include modified functions and functions that are not modified but potentially have indirect dependencies on the modified function.

A program invariant before a function call is denoted in the precondition of the function and includes the properties required for the successful execution of the function. A program invariant in postcondition of a function indicates the state of the program after the execution of the function. These properties can be assertions on a class/object attribute, global variables, elements defined in a property/configuration file, input parameters, return values, or entity objects corresponding to database entities.

When a change is made to a function, it will either change its program invariant before the execution or its logic to produce a different result, which will affect the
Examples 3.1 and 3.2 show how a program change affects its program invariant at the pre-condition and the post-condition of the function.

Example 3.1: Program div before change

```c
void div(int number1, int number 2){
    return number1/number2 ;
}
```

Example 3.2: Program div after change

```c
void div(int number1){
    int number2 = 10;
    return number1/number2 ;
}
```

Program 3.1 is a div function which returns the result of “number1” divided by “number2”. The function div's program invariants at pre-condition are:

<div : enter, number1 , number1 is integer. >

<div : enter, number1, number1 has a value. >

<div : enter, number2, number2 has a value. >

<div : enter, number2, number2 != 0. >

Program 3.2 is the div function after the change. The function div changes its input. It takes one input instead of two inputs. After the code change, the variable “number2” is no longer included in the program invariants of the function at pre-condition. The change of the program invariant indicates that the function has been changed.

The example below shows how a modification affects the program invariant’s property.
Example 3.3: Program after change

```java
Class Calculator {
    private int number2 = 10;
    void div(int number1) {
        return number1 / number2;
    }
}
```

Program 3.3 has a function div defined in class Calculator, and it uses the class's attribute “number2” as a denominator in the computation. The program invariants of the function div at pre-condition are:

- `<div : enter, number1, number1 is integer.>`
- `<div : enter, number1, number1 has a value.>`
- `<div : enter, number2, number2 has a value.>`
- `<div : enter, number2, number2 == 10>`

When the attribute number2’s value is set to 20, the property of “number2” in the program invariant of the pre-condition of function div will change from number2 = 10 to number2 = 20. This change indicates that the class attribute “number2” has been changed and function div is affected. The above examples show that the global variables are defined outside the function and they are used in the function. Once this type of variables are changed, the function uses the changed variables is considered affected, and its program invariants at pre-condition will change.

Example 3.4: Program after change

```java
void div(int number1, int number2) {
    return number1 / number2 + 10;
}
```

Program 3.4 is related to a change in the program logic. It changes its logic from number1 / number2 to number1 / number2 + 10. After the change, whatever input it takes, the result will scale up by 10, which result in the function producing a different return value and changing the post-condition.
The above examples demonstrate that program changes will affect the program invariants. By locating the changed program invariant, we can find which part of the program has been changed or affected.

**Dependency detection**

The dependency of the program invariants before and after a function implicitly shows the data flow across the function. For example, function A depends on function B; when function A’s program invariant in the pre-condition is changed by function B, then function A has an invariant dependency on function B.

![Diagram of function A calling function B](image)

**Figure 3.1**: Function A has a direct dependency on function B

Figure 3.1 shows that function A calls function B and uses function B’s output. If a change occurs in function B and its output becomes different, then its program invariant in the post-condition will be changed. In this case, function A is an affected function because it uses function B’s changed output, which will affect function A’s program invariant in the pre-condition. The program invariant changes in the pre-condition, showing that function A is affected.

Figure 3.2 shows an example where function A calls function B and function B produces variable C that is used by function A. The variable C will be included in the program invariant in the post-condition of function B and in the pre-condition of function A. If a change in function B results in a change in C, then function A is affected since its program invariant(variable C) in the pre-condition is changed.

The example shown in Figure 3.3 shows function A uses variable C, which is being affected by function B. In this case, function A does not call function B. Then a possible test case T1(B) tests function B, which would change variable C first. Then, test case T2(A) tests function A and function A takes the changed variable C, which can potentially cause function A to fail. The variable C is the program invariant.
Figure 3.2: Function A has an indirect dependency on function B

Figure 3.3: Function A has an indirect dependency on function B

in the post-condition of function B and pre-condition of function A. If variable C changes then function A is affected.

In this example variable C can be a global variable, a constant object for configuration, or a piece of data in the database.

The difference between this case and the two earlier examples is that the first two examples can be covered by a modification-traversing approach, since function A will call modified function B. Selecting test cases that exercise function B will be sufficient. However in the case shown in Figure 3.3 function A does not directly call function B, so there is a no direct relationship between function A and function B.
The variable C indirectly binds the two functions together. A test case that does not test the modified function B, and only tests function A can still fail. This is out of the scope of the modification-traversing approach, but by using program invariants, we are able to cover this missing part.

The above examples and explanations show how program invariant can be used in the modification — affected program detection. In the following, we will have a discussion regarding what type of changes can be detected by our approach. Our approach can support the code changes and some of execution context changes. Here we list the types of changes supported by our approach.

1. Delete a class’s attribute
2. Change a class’s attribute (type/value/scope)
3. Delete a global variable
4. Change a global variable (type/value/scope)
5. Delete a function
6. Change a function (parameter/logic/scope)
7. Delete a data entity object
8. Change the data entity object (type/value/scope)
9. Add a function
10. Add a class
11. Add an global variable
12. Add the data entity object
13. Add a class’s attribute
14. Upgrade/downgrade library
15. Delete a table in the database
16. Change a database table’s column (type/value)
17. Add a database table’s column
18. Change a configuration setting (value)

We use ShoppingCart as an example to demonstrate our methodology for some code changes.

**Example 3.5: ShoppingCart.**

```java
public class ShoppingCart{
    private float totalCost = 0;
    private List<Product> products;
    public void removeProduct(Product p){
        if (p in products){
            products . remove(p)
            . . .
            totalCost = totalCost - price
        }
    }
}
```

Example 3.5 shows the ShoppingCart class, its attributes ("totalCost" and "products"), and its method removeProduct. The removeProduct function removes a product from the shopping cart and updates the value of the class attribute "totalCost". The attribute "totalCost" is involved in the program invariants in the pre-condition and post-condition of the function removeProduct. It has a constant property that the value of "totalCost" is greater than or equal to zero in the pre-condition, and this value is less than or equal to the one prior to the execution after the execution of the function. Table 3.2 shows the invariants in the pre- and post-conditions of the method.

**Table 3.2: The program invariants of removeProduct.**

<table>
<thead>
<tr>
<th>Program Point</th>
<th>variables</th>
<th>Condition/Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShoppingCart.removeProduct: Enter</td>
<td>totalCost</td>
<td>totalCost ≥ 0</td>
</tr>
<tr>
<td>ShoppingCart.removeProduct: Exit</td>
<td>totalCost</td>
<td>totalCost ≤ orig(totalCost)</td>
</tr>
</tbody>
</table>

Delete a class’s attribute, Change a class’s attribute
If an attribute (a variable) in program \( P \) is changed in the modified program \( P' \), then all the functions that use this changed attribute (variable) are affected. The test cases executing these functions will need to be re-tested.

In the ShoppingCart example, when the name of the attribute “totalCost” is changed to “Cost”, then these invariants will be replaced by the invariants inferred by “Cost”. Any function that has “totalCost” in the program invariant at pre-condition will be affected and will need to be re-tested.

Variable change can be changes in type, value or scope (private,public). In the example above the change is the deletion of a variable. In this example, the attribute “totalCost” is changed from 0 to 10, causing invariant properties to change from “totalCost” = 0 into “totalCost” =10. Any function that uses “totalCost” will be affected and needs to be re-tested.

Change in a function

If a function \( f \) is changed and the modified version is \( f' \); then a new test set \( T' \) will be created to test \( f' \) until all tests in \( T' \) pass. Note that \( T' \) may overlap with \( T \). Every test in \( T \) that executes \( f' \) must be tested. If the program invariant in the post-condition of \( f' \) is different from the program invariant in the post-condition of \( f \), then \( f' \)’s post-condition will be updated.

Every function \( g \) in \( P \) that is not changed in \( P' \) but has a program invariant in the precondition that is affected by the program invariant in the postcondition of \( f' \), then \( g \) is affected. All the tests that exercise \( g \) will be selected for re-run.

**Example 3.6: modified removeProduct**

```java
public class ShoppingCart {
    Private float totalCost;
    Private List<Product> products;
    ... other attributes and functions
    Public void removeProduct(Product p) {
        products.remove(p)
        totalCost -= p.price
    }
```
Public void checkout () {
    return totalCost;
}

Example 3.6 shows the modified function removeProduct, where a statement that checks if the product to be removed is in the shopping cart is deleted. Table 3.3 lists all the program invariants. The variable “totalCost” is involved in the program invariants in the pre-condition and post-condition of all functions. By comparing the program invariant before and after change, we find that the invariant \( totalCost \leq orig(totalCost) \) is changed to \( totalCost < orig(totalCost) \). Because of this we consider “totalCost” to be an affected variable, which in turn affects the pre-condition of the function checkout. Therefore, every test that exercises the checkout function should be selected for re-run.

<table>
<thead>
<tr>
<th>Program Point</th>
<th>variables</th>
<th>Condition/Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShoppingCart.removeProduct: Enter</td>
<td>totalCost</td>
<td>totalCost ( \geq 0 )</td>
</tr>
<tr>
<td>ShoppingCart.removeProduct: Exit</td>
<td>totalCost</td>
<td>totalCost &lt; orig(totalCost)</td>
</tr>
<tr>
<td>ShoppingCart.removeProduct: Exit</td>
<td>totalCost</td>
<td>totalCost ( \geq 0 )</td>
</tr>
<tr>
<td>ShoppingCart.checkout: Enter</td>
<td>totalCost</td>
<td>totalCost has a value</td>
</tr>
<tr>
<td>ShoppingCart.checkout: Exit</td>
<td>totalCost</td>
<td>totalCost has a value</td>
</tr>
</tbody>
</table>

Example 3.7: A new function promotion is added to ShoppingCart class

Promotion.cate_1=1
Promotion_1=10%
Desc_1= 10% discount
Code_1= happy Friday
Promotion.cate_2=2
Promotion_2=20%

public class ShoppingCart{
    Private float totalCost ;
    Private List<Product> products;
other attributes and functions

```java
Public int promotion(){
    PropertiesConfiguration pcfg =
    new PropertiesConfiguration("config/promotion.properties");
    Return pcfg.int("promotion_1");
}
```

```java
Public void checkout(){
    return totalCost * promotion();
}
```

Example 3.7 shows an example where ShoppingCart has a new function promotion which reads in the promotion profile and returns the first promotion that has 10% discount for each sale. When it reads the profile, it calls the propertiesConfigurations configuration function and sets the address of the profile. Table 3.5 lists all the program invariants. The parameter URL is captured in the invariant as a constant value “config/promotion.properties”. When this property file has some content changes, every function using this property file is affected and needs to be retested. Furthermore, the program changes the checkout function to call promotion, which changes the invariant in the post-condition of checkout from “totalCost” to \( totalCost = orig(totalCost \ast (1 - 10\%)) \). Every test that exercised checkout will need to be retested.

<table>
<thead>
<tr>
<th>Program Point</th>
<th>variables</th>
<th>Condition/Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShoppingCart.promotion(): Enter</td>
<td>totalCost</td>
<td>totalCost is not null</td>
</tr>
<tr>
<td>ShoppingCart.checkout: Enter</td>
<td>totalCost</td>
<td>totalCost has a value</td>
</tr>
<tr>
<td>ShoppingCart.checkout: Exit</td>
<td>totalCost</td>
<td>totalCost=orig(totalCost*(1-10%))</td>
</tr>
<tr>
<td>PropertiesConfiguration. Configuration(): Enter</td>
<td>url</td>
<td>url = config/promotion.properties</td>
</tr>
<tr>
<td>PropertiesConfiguration. Configuration(): Exit</td>
<td>url</td>
<td>URL has a value</td>
</tr>
</tbody>
</table>

**Table 3.4: The program invariants after the change.**

Example 3.8: Java bean as a configuration

```java
@Configuration
@PropertySource("classpath:application.properties")
public class ConfigurationDAO {
    ```
Sometimes the configuration will map to a Java Object. In the example 3.8 the Configuration DAO class is a configuration java bean. It reads configuration settings from application properties and maps the settings into class attributes. When the configuration settings change, the related class attributes value will change. Test case selection for this cases is the same as class attributes changes.

**Change in a library**

If a function l in a library L in EC is changed, then all the functions in P that call l are affected, and all the tests exercising these functions need to be retested. Note that if the changes in L are unknown, then we take a conservative approach that considers all the functions affected by these library calls and select all the tests exercising these functions to retest.

**Example 3.9: An example of library change**

```java
public boolean isAfterPayDay(DateTime datetime) {
    if (datetime.getMonthOfYear() == 2) {
        return datetime.getDayOfMonth() > 26;
    }
    return datetime.getDayOfMonth() > 28;
}
```
Example 3.9 shows an example of a library change, where the function “isAfterPayDay” takes a date time object as an input, which is an object from the Joda time library and calls its method getDayOfMonth(). When this library function is changed, we select test cases that execute the functions where the type of variable in the pre-condition is related to a library or the program point that is associated with the changed library. Table 3.5 shows the invariants in the pre- and post-conditions of this library call.

<table>
<thead>
<tr>
<th>Program Point</th>
<th>variables</th>
<th>Condition/Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>isAfterPayDay:Enter</td>
<td>datetime</td>
<td>datetime.getClass().getName() = org.joda.time.DateTime</td>
</tr>
<tr>
<td>isAfterPayDay:Exit</td>
<td>return</td>
<td>Return in true,false</td>
</tr>
<tr>
<td>org.joda.time.DateTime.getDayOfMonth():Enter</td>
<td>todayOfMonth</td>
<td>todayOfMonth is not null</td>
</tr>
<tr>
<td>org.joda.time.DateTime.getDayOfMonth():Exit</td>
<td>dayOfMonth</td>
<td>dayOfMonth =orig(dayOfMonth )</td>
</tr>
</tbody>
</table>

Change in a database

If a database schema is changed, then all the entity objects associated with the changed table are affected and all the tests exercising these objects need to be retested. Our approach uses entity objects to build relationships between database and program. Any function that uses an entity object will include the entity object in the pre-condition program invariant of the function. Most modern software applications use Object Relational Mapping to create relationships between database entities and program entities through configuration file or annotation. By analysis of the configuration file and annotation, we can find the relationship between a database table and an entity object.

Example 3.10: Hibernate configuration file. label

```java
@Entity
@Table(name= "T_PRODUCT")
public class Product {
    @Column(name="ID")
```
private int id;
@Column(name="PRICE")
private float price;
......
}
<hibernate mapping>
<class name="Product" table="T_PRODUCT">
<id name="id" column="ID">
<generator class="increment" />
</id>
<property name="price" column="price" />
...
</class>
</hibernate mapping>

Example 3.11: An example of database change

public class ShoppingCart{
Private float totalCost ;
Private List<Product> products;
.... other attributes and functions
Public void removeProduct(Product p){
if (p in products ){
    products.remove(p)
    totalCost =p.price
}
}
Public void checkout (){return totalCost ;
}
Table 3.6: The program invariants after the database change.

<table>
<thead>
<tr>
<th>Program Point</th>
<th>variables</th>
<th>Condition/Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>ShoppingCart.removeProduct:Enter</td>
<td>p</td>
<td>( p \neq \text{null} )</td>
</tr>
<tr>
<td>ShoppingCart.removeProduct:Enter</td>
<td>p</td>
<td>( p . \text{price} \geq 0 )</td>
</tr>
<tr>
<td>ShoppingCart.removeProduct:Exit</td>
<td>totalCost</td>
<td>( \text{totalCost} \geq 0 )</td>
</tr>
<tr>
<td>ShoppingCart.removeProduct:Exit</td>
<td>Products.size</td>
<td>( \text{Products.size} \leq \text{orig(Products.size)} )</td>
</tr>
</tbody>
</table>

In the example shown in Example 3.11, the class Product maps to Table T.PRODUCT and its attribute’s “id” and “price” map to Table T.PRODUCTs column “id” and “price”, respectively. Hibernate uses a configuration file to create this mapping; when the database schema changes, we can identify the affected program entities from the mapping in the configuration file. In this example, when the column “price” in the PRODUCT table is changed, the attribute “price” of Product is affected. Every test that has “price” in the pre-condition of its function call will need to be retested.

3.3.3 CART Regression Test Selection Process

Figure 3.4 shows the work flow of CART regression test selection. Given a program \( P \) and a regression test suite \( T \), we first run Daikon with \( P \) on \( T \) to obtain a set of invariants \( PI = < p, v, c > \). This process only needs to be performed once. It can be done during the testing process, or any time before the first maintenance activity. It will be re-performed only if

1. There is a major change to the program and most of the test cases have become obsolete.
2. New test cases are added to \( T \), Daikon will be run with the program on the new test cases to update the invariants.
3. Test case \( t \ ( t \in T ) \) is modified, Daikon will be run with modified test cases to update the invariants.

To keep track of the relationship between the invariants and the test cases, we create an invariant traceability matrix (ITM) to render the relationship.
Figure 3.5 shows the relationship between test case and program invariant. A test case contains multiple program invariants and each program invariant can be included in different test cases, forming a many-to-many relationship. Figure 3.6 shown an example ITM that depicts this relationship. Each row in ITM is associated with an invariant that shows the occurrence of the invariant in each test in T, and each column is associated with a test case that shows which invariants appear in the test case (0 means the test case does not traverse this program invariant and 1 vice versa). Additionally, we add the names of the callers in the pre-condition to record the caller-callee relationship. ITM can be created during the testing phase when the test cases are created and executed and be incrementally updated when there are changes in the invariant after each new test is executed.

When a change is made to P and we have a modified program $P'$, we select all the modification-traversing test cases that execute the modified code (functions/vari-
If the change is about fixing a bug, there is usually a failed test case that needs to be fixed. After the program is changed, then the previously failing test case should now pass.

If the change adds a new feature, which will introduce a new test case, We can consider newly added test cases as modification-traversing test cases.

If the change does not have any failure or new test cases associated with it, we assume that the modified code is known. Otherwise, a simple diff operation can find

<table>
<thead>
<tr>
<th>Program Points</th>
<th>Invariant</th>
<th>buggy</th>
<th>buggy</th>
<th>buggy</th>
<th>buggy</th>
<th>buggy</th>
<th>buggy</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):ENTER this has only one value</td>
<td>1 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):ENTER this.instant has only one value</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):ENTER this.iField has only one value</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):ENTER this.iField.getClass().getName() == org</td>
<td>0 0 1 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):ENTER arg0 == 8</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):EXIT this.instant == org(this.instant)</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):EXIT this.iField == org(this.iField)</td>
<td>0 1 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):EXIT this.instant has only one value</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):EXIT this.iField.getClass().getName() == org</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):EXIT return has only one value</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(int):EXIT this.iField.getClass().getName() == org</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(long):ENTER this has only one value</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(long):ENTER this.instant has only one value</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(long):ENTER this.iField has only one value</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.joda.time.DateMidnight$Property.addCopy(long):ENTER this.iField.getClass().getName() == org</td>
<td>0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
the modified functions, so that we are able to find all the modification-traversing test cases by using ITM.

ITM keeps track of all the functions executed by every test case and program invariants record how the functions interact with variables, so these modification-traversing test cases can be identified by a simple parsing of ITM.

We refer to this modification-traversing test suite as $T'$, and $T' \subset T$. We run Daikon with $P'$ on $T'$ to detect regression faults and to check if any invariants are affected in the executions of these modification-traversing test cases. The set of invariants obtained from these executions is denoted as $PI'$ and the affected program invariants are denoted as as $PI'af$. An invariant $i$ is affected if one of the following conditions holds:

- $i \in I$ and $i \notin I'$
- $i \in I'$ and $i \notin I$
- $i \in I$ and $i \in I'$, where $i.p = i'.p$, and $i.v = i'.v$, but either $i.c \neq i'.c$ or $i.c \not\subseteq i'.c$ if $c$ is a set of values.

If an invariant is affected, then we update this invariant in ITM and include it in the set of affected invariants $I_{af}$.

Next, we identify all the functions that are affected by the affected program invariant and select the test cases that execute these affected functions A function is affected if one of the invariants in its pre- or post-conditions is affected.

### 3.3.4 Regression Test Selection

The details of our approach for selecting regression tests are as follows:

**Code change:** Given a program $P$, a regression test suite $T$, the modified program $P'$, and ITM.

1. If a function $f$ in $P$ is changed to $f'$ in $P'$, and there is at least one statement $s$ such that either $s \in f \& s \notin f'$ or $s \in f' \& s \notin f$, then every test $t$ in $T$ that executes $f$ is modification-traversing and will be selected.
Note that this will select all the modification-traversing tests for $P$ and $P'$ because the execution trace $ET(P(t)) \neq ET(P'(t))$ if either $s \in ET(P(t))$ or $s \in ET(P'(t))$, but not both. It is also possible that a non-modification-traversing test case is selected when only unmodified code in the modified function is executed by the test case. This can be further refined by keeping track of the execution path of the modified function, which will be a trade-off between the precision and the complexity.

To determine if any unmodified function is affected by the change, we execute the modification-traversing test cases with the instrumented $P'$ to detect regression faults and to check the values of the invariants in $P'$. If the values are different from the ones recorded in ITM then the invariants are marked as affected and their values in ITM are updated.

An unmodified function $g$ in $P$ and $P'$ is affected, if there is at least one invariant $i$ in the precondition of $g$ such that $i.v \cap j.v \neq \emptyset$, $\forall j$ in the postcondition of $f$. This implies that at least one variable used in $g$ is manipulated in $f'$; thus $g$ is affected. Also, if there is at least one invariant used in the postcondition of $g$ is affected then $g$ is affected. Every test that executes $g$ will be selected.

(2) When a new function $g$ is added to $P'$, some functions in $P$ will be modified in $P'$ to call $g$, and the test selection with respect to these changed functions is the same as (1).

(3) If a function $f$ in $P$ is deleted, then every function $g$ in $P$ that calls $f$ is affected and needs to be modified. The test selection with respect to $g$ is the same as (1).

(4) If an attribute $a$ (a variable) in $P$ is changed (type or name change) in $P'$, then every function $f$ that uses $a$ is affected, i.e., there is at least one invariant $i$ in the precondition of $f$ such that $a \in i.v$. The test cases that execute $f$ will be selected.

Execution context change: Given a program $P$, a test suite $T$ for $P$, and the execution context $EC$ for $P$.

(5) If $l$ is a function in a library $L$, or is an external API in $EC$ is changed, then every function $f$ in $P$ that calls $l$ is affected, and all the tests that execute $f$ will be selected.
Algorithm 3 regression test selection algorithm

**Input:** $T = t_0, t_1, \ldots, t_n$: a test suite of $P$

1. $ITM[][]$: Invariant Traceability Matrix of $P$ w.r.t. $T$
2. $I_{af}$: a set of affected invariant(s)
3. $P_M$: a set of modified program point(s)
4. $V_M$: a set of modified variable(s)
5. $L$: a set of modified library functions or APIs
6. $D$: a set of modified database tables

**Output:** $T'$

7. $T' = \emptyset$
8. $i = 0$
9. for each $inv \in ITM$ do
   10.   for each $p_m \in P_M$ do
       11.     if $inv.p == p_m$ then
           12.       $I_{af} = I_{af} \cup \{inv\}$
           13.     end if
       14.   end for
   15. for each $v_m \in V_M$ do
       16.     if $inv.v == v_m$ then
           17.       $I_{af} = I_{af} \cup \{inv\}$
           18.     end if
       19. end for
10. end for
11. while $i < |T|$ do
12.   for each $inv \in I_{af}$ and $t_i \in T$ do
13.     if $ITM[inv.index][i] == true$ then $T' = T' \cup \{t_i\}$
14.   end if
15. end for
16. for each $d \in D; obj$ is associated with $d$, and $inv$ is associated with $obj$ do
17.   if $ITM[inv.index][i] == true$ then $T' = T' \cup \{t_i\}$
18. end if
19. end for
20. for each $l \in L$ and $inv$ is associated with $l$ do
21.   if $ITM[inv.index][i] == true$ then
22.       $T' = T' \cup \{t_i\}$
23.   end if
24. end for
25. end while
26. return $T'$
Most modern software applications use Object Relational Mapping to create relationships between database entities and program entities. ORM can be defined by using annotation or using the setting in configuration files. A developer can easily find the correlation between class object and database table. When a program using pure SQL. SQL script will contain table information, attribute information and operation sign (insert, update, delete, select), it is not complex to find the relationship between SQL script with the database. Usually, the result of executing an SQL script will refer to an object; we also call them entity objects.

If a database schema is changed, then all the entity objects associated with the changed table are affected. If a database entity is changed (type, name, or constraint change), then every function \( f \) that references the object \( o \) associated with the changed database entity is affected, i.e., there is at least one invariant \( i \) in the precondition of \( f \) such that \( o \in i.v \). Every test that executes \( f \) will be selected. If ORM is not used by the program, then additional effort will be required to capture the variables used in the sql statements.

If a setting in a configuration file is changed (name, or value if the variable denotes a path or url), the variable \( v \) in the program referred to this setting is affected. The program invariant \( i \) referring to this \( v \) is affected. Then every function that uses this affected program invariant \( i \) is affected. The selection is the same as (4).

Algorithm 3 shows the regression test selection algorithm.

### 3.3.5 Complexity Analysis

The complexity of Algorithm 3 includes the execution time required for inferring program invariants (run Daikon to analyze the subject program), comparing the program invariants to identify the affected invariants, and selecting test cases.

1. The execution time required for running Daikon.

Invariant inferring is based on pattern matching. Daikon predefines many program invariant patterns and checks if variable(s) match one of those or not.

The steps for invariant inferring are as follows: at each program point:
a. Instantiate program invariant patterns over all combinations of variables. For example, if there are two variables x, y and a pattern indicating one variable’s value is greater than the other, then algorithm instantiates \( x > y \), \( x < y \).

b. Validate patterns by consuming samples. A sample is the set of values associated with the key variables x and y. Check each sample against the pattern. Remove the pattern is contradicted by the sample; otherwise, keep the pattern. For example, \( x = 4 \) and \( y = 5 \). The pattern \( x > y \) will be removed and \( x < y \) will remain.

c. Report the pattern that remains after all samples have been processed. This pattern will be summarized as the condition/constraint of the program invariant.

The execution time required by CART contains three parts the time for program invariant inferring, identifying the affected program invariants and test selection.

Let \(|V|\) denote the number of variables at the program points, \(|P|\) denote the number of the instrumented program points and \(|T|\) denote the number of test cases, assume each \(|T|\) has a set of samples \(|S|\). The memory required for Daikon is \( O(|V|^3 \times |P|) \), which is the number of patterns. The algorithm will apply all the samples to the patterns, then the execution time required for running Daikon is \( O(|V|^3 \times |P| \times |S|) \).

2. The maximum number of program invariants we can get is \( |V|^3 \times |P| \), then the execution time required for comparing \( I \) and \( I' \) (program invariants after changes) and identifying the affected invariants is \( O(|V|^3 \times |P|) \). The comparison is based on program point level, for each program point we compare its program invariants before and after code change.

3. The execution time required for the test selection is \( O((|P| \times |V|^3)^2) \), which is required for the algorithm to parse the invariant traceability matrix. \(|I_{af}|\) denotes the the number of affected program invariant, \(|I|\) denotes the number of program invariants, then the selection requires \( O(|I_{af}| \times |I|) \) and the worst case is \( O(|V|^3 \times |P|^2) \).

The total execution time required by CART is \( O(|V|^3 \times |P| \times |S|) + O(|V|^3 \times |P| \times |S|) + O((|P| \times |V|^3)^2) \).
3.3.6 Multi-Objective Regression Test Selection Overview

A safe selective regression testing re-tests all the modification-traversing tests, which ensures that if there are regressions faults in the program that can be detected by the given regression test suite then the selected regression tests will reveal them. However, in a fast-paced software development and deployment environment such as DevOps, regression testing has to be performed quickly and may not have sufficient time to execute all the selected tests.

Techniques for improving the efficacy of regression testing have been well studied. The test suite minimization approaches \[6, 81, 82, 83\] aim at removing obsolete test cases that are no longer valid for the changed software and redundant test cases to form a minimum test suite that satisfies the test requirements. The prioritization approaches \[84, 29, 85\] intend to determine an order of test executions that can achieve the desired goal early, such that if the testing process terminates early then the test cases that have potentially higher values such as higher code coverage or estimated fault detection rate would be executed before the termination. The minimization and the prioritization methods can reduce the number of test cases to be retested, but they may overlook some fault-revealing test cases \[8\].

When executing all or selective regression tests cannot be done within the given time constraint, it is strongly desired that if the changed software is to fail then it should fail in very few test executions, so if there are any regression faults they can be corrected immediately. Taking the time constraint into account, several multi-objective regression testing approaches have been proposed \[103, 104, 105, 106\], which apply multiple criteria to select a closed to an optimum subset of test cases that account for the selected criteria to obtain a maximal efficacy. These approaches apply various optimization algorithms to find the best solution to balance the trade-offs between the cost and code coverage. The existing approaches estimate the fault detectability of the regression test case by code coverage, such as block or function coverage, or fault detectability based on history. However, regression faults are introduced by the changes, which do not exist before the changes and are relevant to the modified and affected parts of the software only. It has been suggested that a test case is fault-revealing only if it is modification-traversing \[8\]. Therefore, a test case
that has a high code coverage but does not execute any modified or affected code, i.e., not modification-revealing, will not be fault-revealing. Therefore, it is not clear that the techniques based on code coverage or history can be effective for detecting regression faults.

We present a new approach that formulates a multi-objective function striving to obtain an optimal solution to maximize the fault detection rate under a time constraint. Our approach focuses on selecting the tests impacted most by the changes in the program, upgrade/downgrade of the library, the changes in databases or configuration files. We use program states before and after an invocation of a function to determine if the function is affected by the change and can potentially have regression faults. A program state before a function call denoted in the precondition of a function, including the properties required for the successful execution of the function and the postcondition of a function, indicates the state of the program after the execution of the function.

To maximize the fault detectability within a constrained time, we model the test selection as a multi-objective Knapsack problem, which determines the number of each test case to select so that the total execution time does not exceed the time limit while maximizing the number of fault-revealing tests (failed tests). We apply two optimization algorithms commonly used to solve the sensor placement problems to obtain an optimal set of test cases.

The goal of our test selection technique is to identify test cases executed successfully before the change of the program or the execution context, but they potentially can fail because of the changes. Our approach differs from the existing techniques in (1) it considers the change impact from both program changes and execution context changes, and (2) it focuses on the change impact coverage, not on the code coverage.

To evaluate the effectiveness of the proposed approach, we conducted three case studies on three industrial systems and real regression faults. The results suggest that our approach is much more effective than the existing code coverage-based approaches.
3.3.7 Multi-Objective Regression Test Selection Methodology

Our goal in the multi-objective regression test selection is to maximize the number of regression faults that can be detected with the selected test cases and to ensure that the cost of the test execution time will not exceed the budgeted time constraint. To model this problem, we adopt the model-based sensor placement approach to develop efficient and effective algorithms for selecting the test cases. The criteria we use include the coverage of the affected invariants, modified functions, and execution time. The aim is to select the test cases that have a high number of affected invariants, cover more modified functions, and have low execution time.

Given the input domain $T$ and the objectives defined as above, the goal is to find a set of test cases $S$ from $T$ that have minimal overall detection risk $F(S)$ that accounts for the coverage, and minimal overall cost $C(S)$, which is the cost of the test execution time, subject to a budget constraint on $C(S)$: $C(S) \leq K$, where $K$ is the time allocated for the regression testing. This is a multi-criterion optimization problem and the scalarization approach [58] is commonly used to find such Pareto-optimal solutions. In particular, we optimize the following problem

$$
\min_{S \in T} \quad O(S) = F(S) + \lambda C(S) \quad s.t. \quad C(S) \leq K
$$

(3.1)

by choosing the appropriate weight $\lambda > 0$. All possible Pareto-optimal solutions to the minimization of $O(S)$ subject to the budget constraint $C(S) \leq K$ can be obtained by varying the weight $\lambda$. Problem (3.1) is a hard-combinatory optimization problem, and an exhaustive search of the optimal set is usually unfeasible. We note that there is no unique form of the objective function $O(S)$, which varies depending on its application domain. Hence, we design customized algorithms by exploring the specific structure of the problem under different situations.

We applied two algorithms to solve the problem: Algorithm 1: a sub-modular for test selection that is a combinatorial search based on sub-modular optimization. Algorithm 2: a projected gradient descent algorithm for test selection which is a graph-structured convex optimization based on continuous relaxation. Problem (3.1)
is called a sub-modular minimization problem subject to knapsack constraints if there is no network topology constraint and can be solved in nearly linear time with the approximation factor \((1 - 1/e) \approx 0.632\) using the greedy algorithm [66], as shown in Algorithm 1. This algorithm starts from an empty set \(S = \emptyset\), and adds the element maximizing the discrete derivative \(\Delta(s|S) = O(S \cup s) - O(S)\) (line 3 and line 4). The approximation factor is defined as \(O(S)/O(S^*)\) where \(S\) refers to the approximated set of invariants, and \(S^*\) refers to the optimal set.

Furthermore, we want to consider the function interaction relationship to identify which affected function or modified function has the highest coupling with the other functions. For example, a function has the highest number of callers. If the function is affected, then the more calling contexts the function has, the more likely the regression faults, if they exist, in this function can be detected. Under this assumption, we utilize the function call graph in each test as a network graph, where a node represents a function and an edge denotes a call relationship and select the nodes that have higher incoming edges. By considering the network graph constraint, the problem becomes neither super-modular nor sub-modular. We can instead apply convex optimization techniques to solve problems (3.1) based on continuous relaxation. We first define a vector format of \(S\) as \(z \in \{0,1\}^{|U|}\), and \(S = \text{supp}(z)\) as the set of nonzero entries in \(z\). The two objective functions \(F(S)\) and \(C(S)\) can be reformulated as the functions based on \(z\) as \(f(z)\) and \(c(z)\), respectively. The constraint \(S \in \mathbb{T}\) can be reformulated as \(\text{supp}(z) \in \mathbb{T}\). Problem (3.1) can then be reformulated as:

\[
\min_{z \in \{0,1\}^{|U|}, \text{supp}(z) \in \mathbb{T}} o(z) = f(z) + \lambda c(z), \quad \text{s.t. } c(z) \leq K, \tag{3.2}
\]

which can be approximated using graph-structured convex optimization techniques [70][71], where “graph-structured” means that the constraint \(\text{supp}(z) \in \mathbb{T}\) is defined based on graph topology. Algorithm 2 shows the basic steps of the projected gradient descent approach for solving problem (3.2), where \(\nabla f(z)\) refers to the gradient of \(f(z)\) with respect to \(z\), and \(\nabla c(z)\) is defined similarly.

The vector \(\nabla f(z^{(i)}) + \lambda \cdot \nabla c(z^{(i)})\) calculated in line 4 is the gradient of the overall objective function \(o(z)\). The vector \(b^{(i)}\) is the gradient descent update, which
is the same as the update of the standard gradient descent algorithm, and \( \lambda \) refers to the step size. As \( b^{(i)} \) is often not within the graph-structured domain \( \mathcal{T} \), line 5 is to project \( b^{(i)} \) to the input domain \( \mathcal{T} \) based on the following projection operator:

\[
T(b^{(i)}) = \min_{z \in \{0,1\}^{[u], \text{supp}(z) \in \mathcal{T}}} \| z - b^{(i)} \|_2^2 \quad s.t. \quad c(z) \leq K, \quad (3.3)
\]

where \( \text{supp}(z) \) refers to the set of non-zero entries in \( z \).

**Algorithm 4** A submodular algorithm for test selection

**Input:** the ground set of test cases \( \mathcal{T} \).

**Output:** \( S \)

1: \( S = \emptyset \)
2: \textbf{while} \( C(S) < K \) \textbf{do}
3: \( s = \text{argmax}_{s \in \mathcal{T}} O(S) - O(S \cup s) \quad s.t. \quad S \cup s \in \mathcal{T} \)
4: \( S = S \cup \{s\} \)
5: \textbf{end while}

**Algorithm 5** A projected gradient descent algorithm for invariant selection

**Input:** Network \( \mathcal{G} \) and the ground set of invariants \( \mathcal{U} \).

**Output:** \( S \)

1: \( i = 0 \)
2: \( z^{(i)} = 0 \)
3: \textbf{repeat}
4: \( b^{(i)} = z^{(i)} - (\nabla f(z^{(i)}) + \lambda \cdot \nabla c(z^{(i)})) \)
5: \( z^{(i+1)} = T(b^{(i)}) \)
6: \( i = i + 1 \)
7: \textbf{until} Convergence
8: \( S = \text{supp}(z^i) \)
3.4 Controlled experiments

We conducted three controlled experiments to evaluate the efficacy of our approach. The research questions we asked in these studies are:

(1) For code changes, is Algorithm 3 as effective as the existing approaches?

(2) Is Algorithm 3 a safe approach, i.e., can it select all the fault-revealing regression tests from the regression test suite?

(3) For changes in the execution context, can Algorithm 3 select all the affected test cases?

(4) How effective is our multi-objective selection to detect all the regression faults within the time constraint?

To answer the first question, we compared Algorithm 3 with three prevailing approaches that identify modification-traversing test cases by comparing the source code before and after a change. For the second question, we conducted experiments on three opensource programs that have reported regression faults and regression test suites. We compared Algorithm 3 with the retest-all approach to evaluate if Algorithm 3 can select all the fault-revealing test cases selected by the retest-all approach. For the third question, we upgraded and downgraded libraries that the application depends on and checked if Algorithm 3 selected all the test cases affected by the change of the library and potentially have backward/forward compatibility issues. Furthermore, we modified the database of an application and investigated if Algorithm 3 identified all the test cases that referenced the modified database entities. We then used our multi-objective function to select test cases to answer the last question.

To conduct these experiments, we developed a prototype of Algorithm 3 for the proposed approach. Three existing approaches, execution trace-based (ET), File dependence RTS (FRTS) [101], and the Hybrid RTS [102] (HyRTS), were used to compare the effectiveness of the regression test selection. To implement the execution trace based approach, we used JTracor [107], to collect the execution information. JTracor is a framework that uses the Java Debugging Interface (JDI) to launch the program and to trace information in a debug mode of JVM and notify an interface.
when events occur in this interface. For FRTS and HyRTS, we used the plugin systems provided by the authors of the papers [108, 101, 102].

We conducted the experiments on four open source software: Commons-Lang3, Commons-IO and Commons-Validator obtained from Apache Software Foundation and BookStore from Github. Apache Software Foundation provides the release history and a source repository in which we can get source code of each release and the test suite for each project. Between two versions of the program there are some modifications for bug fixing or for improving software performance. GitHub provides online project hosting services, from which we selected BookStore that provides issue tracking and test cases.

These studies were conducted in two phases. The first phase was to obtain the relationships between the likely invariants and the test cases. Daikon was used to instrument the base version of the subject software and to run the test suite to construct the invariant traceability matrix (ITM). The second phase performs regression testing on the modified software. We used five different approaches, retest-all, Algorithm 3, ET, FRTS, and HyRTS to evaluate the effectiveness.

3.4.1 Experiment I

The first experiment was to investigate the effectiveness of the selective regression testing for the code changes. In this study, Commons-Lang3, Commons-IO, and Commons-Validator were used. The source code and the test cases were obtained from the Apache Software Foundation that provides the release history and a source repository in which we can get the source code of each release and the test suite for each project. Table 3.7 shows the details of these systems. We first ran all the test cases to find all the regression faults (w.r.t. the test suite) for each experiment, then we used three existing approaches, execution trace-based (ET), File dependence RTS (FRTS) [101], and the Hybrid RTS (HyRTS) [102], to compare the effectiveness of the regression test selection, based on the number of the test cases selected by the techniques and the number of the regression faults detected by their selected test cases. Additionally, we applied our multi-objective algorithm on the test suite selected by Algorithm 3 and used 10% of the total testing time as the time constraint.
to select test cases that are most likely fault-revealing.

The experiments were conducted in two phases. The first phase instrumented the base version of the subject software and ran the test suite to construct the invariant traceability matrix (ITM). The second phase performed the regression testing on the modified software. To implement the execution trace-based approach, we used JTracor \cite{107}, to collect the execution information. This approach detected the modification at the statement level, which is the finest level used to select the modification-traversing test cases. For FRTS and HyRTS, we used the plugin systems \cite{108} provided by the authors of the papers \cite{101,102}.

The results are shown in Table 3.8, which shows the total number of the test cases, the total number of the test cases selected by each technique, the total number of the regression faults (Total_RF) found in the given test suite, the total number of the regression faults detected by the selected test cases (Detected_RF), the total number of the modified classes (M_Classes), the total number of the modified functions (M_Functions), the total number of the modified statements (M_Statements), the execution time used to build ITM (PhaseI_ITM), the execution time used to compute the initial data for HyRTS (PhaseI_HyRTS), the execution time used to select the regression tests by Algorithm 3 (PhaseII_CART), and the execution time used to select the regression tests by HyRTS (PhaseII_HyRTS). In this experiment, all the four techniques detected all the regression faults. In the Commons-Lang3 project, ET selected the least number of test cases in all three modifications. Algorithm 3 selected 1% to 10% more test cases than ET. In version 3.2, both FRTS and HyRTS selected 100% of the test cases. In version 3.3, HyRTS selected 5% more test cases than ET, and 4% less than Algorithm 3. In the Commons-IO project, Algorithm 3 selected 1% to 3% more test cases than ET, and FRTS selected 100% test cases, and HyRTS selected 98% test cases in version 2.3. In the Commons-Validator project, for version 1.5 and 1.6, both FRTS and HyRTS selected 100% test cases; Algorithm 3 selected 6% more test cases than ET in version 1.5, and 5% more than ET in version 1.6.

In summary, all approaches selected all the fault-revealing tests: Algorithm 3 performed slightly worse than ET and HyRTS in two cases. Although FRTS and
HyRTS selected many test cases, the time required for the analysis was very short (a few sections). The total time required for Algorithm 3 is $O(n \times T)$ for phase 1 and $O(T)$ for phase 2, where $n$ is the size of the program and $T$ is the number of the test cases. Phase 1 only needs to be performed once, the time is required to annotate the program and run the test cases to construct ITM. Phase 2 is performed after each change, the time is used to parse the ITM to select the affected test cases and update ITM. We used a bit-comparison of the invariant, so each test case used a constant time to determine if it executed a modified or an affected function. At the beginning of the experiments, Algorithm 3 annotated the target program and ran the test cases to construct ITM, which took 10 to 15 minutes. HyRTS used 8 to 87 seconds to run the test cases and compute the metadata. After each change, both Algorithm 3 and HyRTS can efficiently select the test cases. Algorithm 3 used less than two seconds for the annotation of the modified program and the test case selection. HyRTC used less than one second for each selection process. HyRTC is more efficient than Algorithm 3 because it compared the checksum of the byte code, while Algorithm 3 needed to annotate the modified program and compare the invariants. However, the difference is not much. Both FRTC and HyRTC only perform well for small changes in Java programs, which limits their applicability.

Table 3.7: The details of the subject programs.

<table>
<thead>
<tr>
<th>Program</th>
<th>Commons-Lang3</th>
<th>Commons-IO</th>
<th>Commons-Validator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>3.0</td>
<td>3.1</td>
<td>3.2</td>
</tr>
<tr>
<td>Class</td>
<td>86</td>
<td>99</td>
<td>107</td>
</tr>
<tr>
<td>Function</td>
<td>4306</td>
<td>4344</td>
<td>4625</td>
</tr>
<tr>
<td>LOC</td>
<td>50766</td>
<td>58967</td>
<td>50766</td>
</tr>
<tr>
<td># of Test cases</td>
<td>1902</td>
<td>2051</td>
<td>2392</td>
</tr>
</tbody>
</table>

Table 3.8: Results of Experiment I

<table>
<thead>
<tr>
<th>Program</th>
<th>Commons-Lang3</th>
<th>Commons-IO</th>
<th>Commons-Validator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total test cases</td>
<td>1902</td>
<td>2051</td>
<td>2392</td>
</tr>
<tr>
<td>Algorithm 3</td>
<td>301 (28.97%)</td>
<td>1480 (72.19%)</td>
<td>1458 (66.95%)</td>
</tr>
<tr>
<td>ET</td>
<td>57 (27.71%)</td>
<td>1399 (68.24%)</td>
<td>1225 (54.21%)</td>
</tr>
<tr>
<td>TRTS</td>
<td>1031 (85.79%)</td>
<td>2050 (100%)</td>
<td>2035 (85.98%)</td>
</tr>
<tr>
<td>HyRTC</td>
<td>1347 (93.82%)</td>
<td>2050 (100%)</td>
<td>2320 (90.46%)</td>
</tr>
<tr>
<td>Total RF</td>
<td>4</td>
<td>7</td>
<td>17</td>
</tr>
<tr>
<td>Detected RF</td>
<td>4</td>
<td>7</td>
<td>17</td>
</tr>
<tr>
<td>M, Classes</td>
<td>23</td>
<td>36</td>
<td>39</td>
</tr>
<tr>
<td>M, Functions</td>
<td>34</td>
<td>139</td>
<td>200</td>
</tr>
<tr>
<td>M, Statements</td>
<td>296</td>
<td>2073</td>
<td>2324</td>
</tr>
<tr>
<td>PhaseI HANDLE</td>
<td>1.250</td>
<td>1.504</td>
<td>1.299</td>
</tr>
<tr>
<td>PhaseII HANDLE</td>
<td>0.55</td>
<td>0.58</td>
<td>0.53</td>
</tr>
</tbody>
</table>
3.4.2 Experiment II

The second study focused on the changes in the program dependent libraries. When a library is changed, in most cases, it is difficult to identify where the changes are and how the program will be affected. Our approach selects all the tests that use the changed library. In our experiment, Daikon generated program invariants based on every statement executed, including calls to library functions, which provides the information of which library calls are executed by the test. We used Commons-Validator and changed the dependent libraries by modifying the pom file. We downgraded and upgraded two versions of its dependent libraries.

The results of regression testing on this study are summarized in Table 3.9, which shows that all the fault-revealing tests in the test suites were selected by retest-all and Algorithm 3. The other three approaches did not select any test case because there was no change in the code. In this experiment, the upgrade and downgrade versions of the library beanutils were very different, so both upgrading and downgrading had 121 affected tests. However, for the library digester, upgrading did not have any issue but downgrading had 90 affected tests.

<table>
<thead>
<tr>
<th>Program</th>
<th>Commons-Validator</th>
</tr>
</thead>
<tbody>
<tr>
<td>library</td>
<td>beanutils</td>
</tr>
<tr>
<td>Modification</td>
<td>1.8-1.9.3</td>
</tr>
<tr>
<td>Failed tests</td>
<td>121</td>
</tr>
<tr>
<td>Reg. faults</td>
<td>1</td>
</tr>
<tr>
<td>Selected tests</td>
<td>121(23%)</td>
</tr>
<tr>
<td>Det. reg. faults</td>
<td>1</td>
</tr>
</tbody>
</table>

3.4.3 Experiment III

The third experiment focused on the database change; we used a web application BookStore from GitHub that provides the source code and the test cases. For each modification, we ran all the test cases to find the regression faults in this test suite as we did in the other two studies. The bookStore is an online shopping platform for
searching and buying books. This web application stores data in MySQL database, and uses com.mchange.v2.c3p0 package and javax.sql to transform data from MySQL database into entity objects. There were three modifications to the database. The first one changed the name of the "book" table from "name" to "title". When the program initialized the book entity object, this change caused the program to fail. The "book" table is mapped to the "book" entity object, which can be located from the program invariants. Any test case referring to the "book" entity will be selected. The second change deleted the "category" table, which is used for categorizing books. After deleting this table, all the books were categorized into one group. Because the "category" table is associated with the "category" entity object, all the test cases referring to this object were selected. The third modification added a column into the "book" table. Since the new column was not used, it did not affect the program.

The results of these three database modifications are shown in Table 3.10. For the first modification, Algorithm 3 selected 30 test cases and all three failed test cases were selected. For the second modification, Algorithm 3 selected 62 test cases and detected all 12 failed test cases. For the last one, there was no regression fault found in this change, and our method selected 31 test cases. This experiment demonstrates that Algorithm 3 can select all the test cases affected by the database changes and detect all the regression faults, which cannot be detected by the techniques based on the changes in the code [8, 109, 102].

Table 3.10: The results of the database changes.

<table>
<thead>
<tr>
<th>Database Change</th>
<th>Modify column</th>
<th>Delete table</th>
<th>Add column</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failed tests</td>
<td>3</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>Regression faults</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Selected tests</td>
<td>30(12.6%)</td>
<td>62(26%)</td>
<td>30(12.6%)</td>
</tr>
<tr>
<td>Detected regression faults</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

3.4.4 Discussion

In these controlled experiments, we observed that (1) for code change only, Algorithm 3 selected all the affected test cases that were selected by ET, but it selected 0.67 % to 1.9% more test cases than ET. Both approaches selected all the
fault-revealing test cases in the test suites. (2) For library changes, Algorithm 3 selected all the fault-revealing test cases with 0.5% to 23% of the test cases. (3) For database changes, Algorithm 3 also selected all the fault-revealing test cases with 12.6% to 26% of the test cases. Therefore, in these experiments Algorithm 3 was as effective as ET, was safe in both changes in the code and in the execution context.
3.5 Case Studies

To evaluate the validity of the proposed approach and its significance, we conducted three case studies on industrial systems, for which the developer teams provided the source code, the test cases, and the trouble reports. We first annotated the applications and ran the test cases provided by the developer team to construct ITM, then used the three strategies retest-all, Algorithm 3, and ET to select and run the selected cases. The current versions of FRTS and HyRTS do not support JavaScript, so they were not used in these studies. In addition to the regression test selection, we applied the multi-objective selection algorithm that uses affected invariants as the value objective to select test cases under a time constraint.

The research questions we asked in these studies are:

(1) How efficiently can Algorithm 3 and Algorithm 4 detect all the regression faults as compared to the existing approaches?

(2) How efficiently can Algorithm 5 detect all the regression faults with the help of the network graph?

To answer the those two questions, We applied our algorithms on these programs to compare the effectiveness of selecting fault-revealing tests and fault detectability. In addition, we computed the average of fault age (the number of regression tests executed to detect the regression fault) suggested by Kim and Porter [110] for measuring the effectiveness of regression test selection, by using different objectives.

For the multi-objective selection algorithms, we used the coverage of modified functions, affected invariants, function calls, and execution time as the criteria to form the objective function. We first define:

\[
Risk_{\text{Modified Fn}}(s) = \frac{\text{total number of uncovered modified function}}{\text{total number of modified function}}
\]

\[
Risk_{\text{Affected Invariant}}(s) = \frac{\text{total number of uncovered affected invariants}}{\text{total number of affected invariants}}
\]

\[
Function\_Call\_Coverage(S) = \frac{\text{total number of uncovered function calls}}{\text{total number of function calls}}
\]
We developed three Multi-Objective functions:

**Algorithm 1a** uses Algorithm 4 to get a set of test cases that minimizes the objective function.

\[ F(s) = \text{Risk}_{\text{Modified Fn}}(s) + \text{Risk}_{\text{Affected Invariant}}(s) \]

\[ C(s) : \text{The sum of the execution time for each test case} \]

\[ O(S) = F(s) + \lambda C(s) \]

**Algorithm 1b** uses code coverage and execution time as the objectives and applies Algorithm 4.

**Algorithm 2** uses Algorithm 5 to find an optimal set of test cases that minimizes the objective function.

\[ F(s) = \text{Risk}_{\text{Modified Fn}}(s) + \text{Risk}_{\text{Affected Invariant}}(s)+\text{Function Call Coverage}(s) \]

\[ C(s) = \text{the sum of the execution time for each test cases} \]

\[ O(s) = F(s) + \lambda C(s) \]

The cost function was modeled at 5%, 10%, 20%, 30%, 40%, and 50% of the total execution time of the regression test suite. Algorithm 2 used the projected gradient descent algorithm, which obtained a local optimal point when it converged. After its first convergence, we re-calibrated the objective function to obtain the next optimal point.
3.5.1 Case Study I

The first study was conducted on an education software, Idea Thread Mapper [ITM], used in k-12 schools located in four countries, including the US, Canada, Singapore, and Taiwan. The system is built based on the microservices and is implemented in JavaScript, D3.js, Java, and uses MySQL. The current version contains 39 Java classes, 1,085 functions, and 142,908 lines of code. At the beginning of the study, there were 397 test cases created by the testing group, and in the end there were 412 test cases after removing obsolete test cases and adding new test cases during this study. Each test case takes several minutes to an hour to execute. There is often very limited time for regression testing and retesting all the test cases will delay the restoring of the services and may disrupt the classroom use, which is not viable. There were five corrective activities during this study, including the fixes after a missing database table, a change to a global variable, a change to a session, a change of URL in the configuration, and a change of library. The results of this study are summarized in Table 3.11 and Table 3.12, where \( f_t \) denotes the total number of selected fault-revealing tests and \( r_f \) denotes the total number of detected regression faults.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>All</th>
<th>ET</th>
<th>Algorithm 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( f_t ) 399</td>
<td>10(2.5%)</td>
<td>110(27.5%)</td>
</tr>
<tr>
<td></td>
<td>( r_f ) 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>( f_t ) 399</td>
<td>98(24.5%)</td>
<td>111(27.8%)</td>
</tr>
<tr>
<td></td>
<td>( r_f ) 3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>( f_t ) 399</td>
<td>158(39.6%)</td>
<td>154(38.6%)</td>
</tr>
<tr>
<td></td>
<td>( r_f ) 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>( f_t ) 412</td>
<td>0</td>
<td>46(11.1%)</td>
</tr>
<tr>
<td></td>
<td>( r_f ) 1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>( f_t ) 412</td>
<td>0</td>
<td>15(3.6%)</td>
</tr>
<tr>
<td></td>
<td>( r_f ) 1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

In Case 1, a new feature that requires a new table was added but some of the external servers did not have the new table, causing the system to fail when it tried
Table 3.12: The results of multi-objective selection algorithms on the case study I- ITM.

| Case | Algorithm 1a | | | | Algorithm 1b | | | | Algorithm 2 | |
|------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|-------------|
|      | 5% | 10% | 20% | 30% | 40% | 50% | 5% | 10% | 20% | 30% | 40% | 50% | 1 | 2 |
| 1    | f.  | 5  | 5  | 5  | 9  | 9  | 1  | 1  | 2  | 4  | 4  | 4  | 5(29.8%) | 9(34.5%) |
|      | r.  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| 2    | f.  | 8  | 16 | 22 | 37 | 86 | 97 | 3  | 17 | 47 | 51 | 77 | 79 | 89(41.2%) | 98(56.1%) |
|      | r.  | 2  | 3  | 3  | 3  | 3  | 3  | 1  | 3  | 3  | 3  | 3  | 3  | 3  | 3  |
| 3    | f.  | 19 | 39 | 68 | 89 | 140| 147| 11 | 29 | 93 | 115| 121| 140(43.9%) | 150(58.5%) |
|      | r.  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| 4    | f.  | 5  | 11 | 43 | 58 | 60 | 61 | 0  | 5  | 25 | 42 | 46 | 60(40.3%) | 66(52.4%) |
|      | r.  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| 5    | f.  | 1  | 3  | 3  | 3  | 3  | 3  | 0  | 0  | 0  | 0  | 0  | 0  | 3(8.5%) | N/A |
|      | r.  | 1  | 1  | 1  | 1  | 1  | 1  | 0  | 0  | 0  | 0  | 0  | 0  | 1  | N/A |

to retrieve data from these servers. There were nine fault-revealing tests and one regression fault. Some object maps to the changed table in the database, which has many attributes. When one of the attributes of the entity object was changed, the object was considered as affected, and every function that takes the affected entity object as input was considered as affected. Some of them executed the affected object but not the modified attribute; thus Algorithm 3 selected many non-fault-revealing test cases. ET selected 10 test cases but failed to detect the regression fault.

Algorithm 2 used 29.8% of the cost and detected the regression fault, and used 34.5% of the cost selecting all the fault-revealing tests; Algorithm 1a used 5% of the cost detecting the regression fault and selected all the fault-revealing tests with 40% of the cost. Algorithm 1b also detected the regression fault with 5% of the cost, but it only selected four fault-revealing tests with 50% of the cost. The average fault age for Algorithm 1a and 1b was 12 and 18, respectively.

In Case 2, there were 98 fault-revealing tests and three regression faults. Both Algorithm 3 and ET selected all the fault-revealing test cases; Algorithm 3 selected 27.8% and detected all three regression faults with the 40 test cases selected by the algorithm, and ET selected 24.5% of the test cases. Algorithm 1a and Algorithm 1b detected all three regression faults by using 10% of the cost. Algorithm 2 used 41.2% of the cost to detect the three faults. For the fault-revealing tests, Algorithm 1a selected 97 tests and Algorithm 1b selected 79 tests with 50% of the cost. Algorithm
2 selected all 98 tests with 56.1% of the cost. The average fault age for Algorithm 1a and 1b was 6 and 8, respectively.

In Case 3, there were 150 fault-revealing tests and one regression fault. Algorithm 3 selected 39.6% and detected the regression fault with the first 20 test cases, and ET selected 38.6% of the test case. Algorithm 1a and Algorithm 1b detected the regression fault by using 5% of the cost. Algorithm 2 used 43.9% to detect the fault. For the fault-revealing tests, Algorithm 1a selected 147 tests and Algorithm 1b selected 121 tests with 50% of the cost. Algorithm 2 selected all 150 tests at 58.5% of the cost. The average fault age for Algorithm 1a and 1b was 2 and 3, respectively.

In Case 4, there were 66 fault-revealing tests and one regression fault. Because they did not have any code change, ET did not select any test case, thus did not detect any regression fault. Algorithm 3 selected 11.1% test cases and detected the regression fault by the first 21 test cases selected by the algorithm. Those regression faults was detected by Algorithm 1a at 5% of the cost, by Algorithm 1b at 10% of the cost, and by Algorithm 2 at 40.3% of the cost. For the fault-revealing tests, Algorithm 1a selected 61 tests and Algorithm 1b selected 46 tests at 50% of the cost, and Algorithm 2 selected all 66 test cases at 52.4% of the cost. The average fault age for Algorithm 1a and 1b was 8 and 34, respectively.

In Case 5, there were three fault-revealing tests and one regression fault. Algorithm 3 selected 3.6% of the test cases and detected the regression faults by the first 21 test cases selected by the algorithm. Algorithm 1a detected the regression fault at 5% of the cost, and Algorithm 2 detected the fault at 8.5% of the cost. Algorithm 1b failed to detect the fault with 50% of the cost. All three fault-revealing tests were selected by Algorithm 1a at 10% of the cost and by Algorithm 2 at 8.5% of the cost. Algorithm 1b did not select any of them at 50% of the cost. The average fault age for Algorithm 1a and 1b was 1 and 223, respectively.

Discussion: In this case study, for all the five modifications, Algorithm 3 was able to select all the fault-revealing test cases with 3.6% to 38.6% of the test cases. By applying the multi-objective algorithm, Algorithm 3 detected all the regression faults with 10% of the test cases. ET failed to select any fault-revealing test cases for the three context changes in case 1, case 4, and case 5. It was able to select
all the fault-revealing test cases for case 2 and case 3 with 24.6% and 38.6% of the test cases that are slightly better than Algorithm 3. We observed that Algorithm 1a performed much better than Algorithm 1b for detecting regression faults and selecting fault-revealing tests. Algorithm 2 was able to select all the fault-revealing tests with 8.5% to 58.5% of the cost. It detected all the regression faults but used more test cases than Algorithm 1a; when there were large numbers of the affected functions and invariants, Algorithm 2 required a long time to converge and obtain a local optimum. Algorithm 1b used function coverage and failed to detect the regression fault within the budget in Case 5. It is suggested that if it is desired to find a regression fault early and fix it immediately, then Algorithm 1a will be the better choice. When the regression testing is to be performed automatically to find all the fault-revealing tests within the budget, then Algorithm 2 will be the better choice.

3.5.2 Case Study II

The second case study was conducted on two health-related projects developed for the National Cancer Institute [112]. The first project was Microarray, a service that analyzes biology data and provides visualization of the analyzed results. It is a web application powered by Node.js, React.js, and R programming language. There are 21,409 lines of code, 40 functions, and 84 test cases.

A program issue was reported in JIRA due to a change in one of the APIs that returns data and its schema. The program receives the data and uses the schema to extract the data for visualization. The API changed data schema, which introduced one regression fault in the search function (Case 1) and one in the sorting function (Case 2). Among the 84 test cases. The results are summarized in Table 3.13 and Table 3.14.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>All</th>
<th>ET</th>
<th>Algorithm 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ft</td>
<td>84</td>
<td>48(57.1%)</td>
<td>48(57.1%)</td>
</tr>
<tr>
<td>r-f</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ft</td>
<td>101</td>
<td>9(8.9%)</td>
<td>18(17.8%)</td>
</tr>
<tr>
<td>r-f</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.13: The results of Algorithm 3 on the case study II- Microarray.
Table 3.14: The results of multi-objective selection algorithms on the case study II - Microarray.

<table>
<thead>
<tr>
<th>Case</th>
<th>Algorithm 1a</th>
<th>Algorithm 1b</th>
<th>Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5%</td>
<td>10%</td>
<td>20%</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_f</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>r_f</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_f</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>r_f</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In Case 1, there were nine fault-revealing tests and two regression faults. Both Algorithm 3 and ET selected the same test cases and all the fault-revealing test cases. With the algorithm, Algorithm 3 used four test cases for B1 and five test cases for B2 to detect the regression faults.

Algorithm 2 used 42.3% of the cost to select all the fault-revealing tests. Algorithm 1a selected all the fault-revealing tests within 50% of the cost. Algorithm 1b only selected three tests with 50% of the cost. For the two regression faults. Algorithm 1a detected one regression fault by using 5% of the cost and two regression faults by using 10% of the cost. Algorithm 1b used 30% of the cost to detect one regression fault and failed to detect the second regression fault by 50% of the cost. Algorithm 2 used 38.1% of the cost to detect the first fault and 42.3% of the cost to detect both faults. The average fault age for Algorithm 1a and 1b was 2 and 22.5, respectively.

In Case 2, there were only two fault-revealing tests. Algorithm 3 used 17.8% of test cases and selected all fault-revealing test cases. ET used 8.9% of the test cases but failed to select any fault-revealing test cases. This is because the modified code changed the data in the sessions, and the changed data were used by an unchanged function. A test case that executes this function will fail and will be selected by Algorithm 3 because an invariant related to this piece of data is affected; however, ET does not select this test case, because it did not execute any modified function. Algorithm 1a selected both tests with 10% of the cost and Algorithm 2 used 8.2%. Algorithm 1b only selected one tests within the budget. Algorithm 1a used 5% of
the cost to detect the regression fault, Algorithm 2 used 8.2%, and Algorithm 1b used 30% of the cost. The average fault age for Algorithm 1a and 1b was 1 and 42, respectively.

The second project was the Cancer Epidemiology Descriptive Cohort Database (CEDCD) [113], which contained descriptive information about cohort studies that follow groups of persons over time for cancer incidence, mortality, and other health outcomes. The CEDCD program facilitates collaboration and highlights the opportunities for research within existing cohort studies. It is a web application maintained by the Epidemiology and Genomics Research Program (EGRP), located in the Division of Cancer Control and Population Sciences, National Cancer Institute’s (NCI’s), National Institutes of Health.

The program is powered by Node.js, React.js, and uses MySql database. It has 32,796 lines of code, 155 functions, and 190 test cases. Two modifications that introduced the regression faults were reported in JIRA. The first change was a business logic change, which modified a stored procedure and changed its output. The return value was a data table that has a column to indicate genders. Before the change, the gender was an integer, -1(unknown), 0(both), 1(male), -1(female). After the change, the gender was changed into a String Type as male, female, unknown, and both, which caused one test case to fail. A second change was made to a drop-down selection box, which changed no cancer to No Cancer. This change caused five test cases to execute a function that takes no cancer as input to fail. The results are summarized in Table 3.15 and Table 3.16.

<table>
<thead>
<tr>
<th>Case Study</th>
<th>All</th>
<th>ET</th>
<th>Algorithm 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_f</td>
<td>190</td>
<td>120(63.1%)</td>
<td>120(63.1%)</td>
</tr>
<tr>
<td>r_f</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>f_f</td>
<td>190</td>
<td>1(0.5%)</td>
<td>1(0.5%)</td>
</tr>
<tr>
<td>r_f</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3.15: The results of Algorithm 3 on the case study II- Microarray.
Table 3.16: The results of multi-objective selection algorithms on the case study II - CEDCD.

<table>
<thead>
<tr>
<th>Case</th>
<th>Algorithm 1a</th>
<th>Algorithm 1b</th>
<th>Algorithm 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5% 10% 20% 30% 40% 50% 5% 10% 20% 30% 40% 50%</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>f. t</td>
<td>1 1 3 5 5 5 0 0 0 1 1 3</td>
<td>4(21.3%)</td>
</tr>
<tr>
<td></td>
<td>r. f</td>
<td>1 1 1 1 1 1 0 0 0 1 1 1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>2</td>
<td>f. t</td>
<td>1 1 1 1 1 1 0 0 0 1 1 1</td>
<td>1(2.6%)</td>
</tr>
<tr>
<td></td>
<td>r. f</td>
<td>1 1 1 1 1 1 0 0 0 1 1 1</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

In Case 1, there were five fault-revealing tests and one regression fault. Both Algorithm 3 and ET selected 63.1% of test cases and detected all fault-revealing test cases. The changed attribute, gender, was widely used in many test cases; thus both approaches selected many test cases. The regression fault was detected by the multi-objective algorithm with 10 test cases in all three studies. Algorithm 2 used 26.5% of the cost to select all the fault-revealing tests. Algorithm 1a used 30% and Algorithm 1b only selected three tests within the 50% budget. The regression fault was detected by Algorithm 1a with 5% of the cost, Algorithm 1b with 30%, and Algorithm 2 with 21.3%. The average fault age for Algorithm 1a and 1b was 8 and 65, respectively.

In Case 2, there were only one fault-revealing test and one regression fault. The change impact was small and both approaches used only 1% of the test cases and selected the fault-revealing test case. Algorithm 1a used 5% of the cost to select the test and detect the regression fault. Algorithm 1b used 30% and Algorithm 2 used 2% of the cost to select the test and detect the regression fault. The average fault age for Algorithm 1a and 1b was 1 and 76, respectively.

Discussion: In this case study, the number of the affected functions and invariants was much less than the ones in the Case study I and we observed that when changes were made to the execution context and not to the code, the execution trace-based technique that selected test cases executing modified code will not be safe, because the behavior of the system can be affected by the change even if the code of the system is the same. For the code changes, both ET and Algorithm 3 can select all the fault-revealing test cases but in some cases that data dependence relationship
between the functions was introduced indirectly via the data in the execution context; then ET was not able to detect the fault-revealing test cases. Algorithm 3 used the invariants to keep track of these relationships and was able to detect all the fault-revealing faults. However, in general, Algorithm 3 selected slightly more test cases than ET in these studies. Besides, Algorithm 1a performed much better than Algorithm 1b for selecting fault-revealing tests and detecting regression faults. Algorithm 2 converged faster in this study than in Case study 1. For the regression fault detection, the performance of Algorithm 1a and Algorithm 2 was about the same. Thus, Algorithm 2 will be the better choice when the impact of the change is small.
CHAPTER 4
THREAT TO VALIDITY

To automatically obtain pre- and postconditions of the functions, our approach relies on Daikon, which has several limitations. First, like all machine learning techniques, Daikon requires sufficient data to infer likely invariants. If the test suite is small and/or has very few faults, then the obtained invariants may be incomplete or inaccurate. Besides, for regression test selection, if the regression test suite is small, then there is no need for select optimization. Retesting all test cases should not be too costly. Second, Daikon produces too many invariants that are irrelevant to the program behavior. Using all the invariants produced by Daikon may cause high false-positives in software anomaly detection and result in selecting too many regression tests. Thus, we filtered out some spurious invariants. Third, for languages that are not supported by Daikon, in our empirical studies, we implemented a parser to instrument Javascript that is not supported by the current version of Daikon. Fourth, the same invariant may be inferred by the different instances of the objects, which may cause selecting the wrong program invariant for monitoring and for regression test case selection, it will select test cases that are not modification traversing.

The subject programs used in the controlled experiments were broadly used in the empirical studies of the existing work to make fair comparisons. These experiments were conducted to demonstrate the differences in these approaches, so we only chose a few representative versions and did not include all the versions used in the original studies. We focused more on the empirical studies using industrial systems that are more complex in order to demonstrate the applicability of our approach in practice, crossing different application domains.
CHAPTER 5
CONCLUSIONS

Online software anomaly detection and regression testing are essential activities for ensuring software quality. Online software anomaly detection usually requires monitoring the online software executions and using predefined oracles to validate the software behavior. Automatic online software anomaly detection is challenging. On one hand, it requires domain knowledge to define and place monitors; on the other hand, the overhead introduced from the online monitoring should not degrade the performance of the software.

We use program invariants as a means to observe program behavior and to validate the correctness of program states.

Meanwhile, machine learning algorithms have been adopted to improve monitoring performance by selecting a small set of representative program invariants for monitoring with maximum anomaly detection performance.

The program invariants selected by the algorithms can achieve a high anomaly detection rate, while keep minimum false positives, false negatives and execution overhead. The algorithm utilizes the sensor placement algorithm analysis of the correlation between program invariants to conduct the selection.

The contribution of our software anomaly detection study as follows:

1. We propose a method to select a small subset of anomaly-revealing invariants, which can be used to detect runtime software anomalous behavior with minimal execution overhead.

2. We use different algorithms and network graphs in the selection process to improve the quality of the program invariant selection.

3. For event-driven multi-threaded applications, we use state machines and Petri Nets to create test cases that explore possible anomaly scenarios in the training phase to better identify anomaly-revealing invariants.
4. Our empirical studies demonstrate that the proposed approach is feasible and effective enough to be applied on large scale complex industrial systems.

In addition to software anomaly detection, we applied program invariants on regression test selection. The challenge of regression testing is being able to select all failure-revealing test cases while using a small number of test cases to detect most of the regression faults.

A safe regression test case selection should account for the changes in both the code and the execution context. To our best knowledge, the existing techniques [86, 87, 88, 8, 92, 88] for regression testing barely addressed this issue. Most of the safe selection approaches assume that factors external to the subject program remain constant which is often impractical for modern multi-team developed evolving software.

We propose a new approach that uses the program invariants as means to select test cases for regression tests. The change of a program invariant between the pre-modified program and post-modified program indicates that it is a modification-revealing program invariant. The test cases that traversed modification-revealing program invariants need to be selected for regression tests.

Furthermore, we developed a multi-objective prioritization algorithm for selecting test cases based on a score function that computes the rank of each test case using the number of affected invariants.

The contributions of our software regression are as follows:

1. While existing approaches only tackle a single type of change, CART can effectively handle different types at the same time and select not only modification-traversing but also modification-revealing tests (assuming the program terminates). Thus, it is not subject to the constraint of constant factor imposed on the existing safe approaches.

2. Our multi-objective test selection takes the coverage of affected functions, modified functions and test execution time as the objective, which will select the test cases that are more likely to expose regression faults.

3. Our approach can be fully automated and we have implemented a prototype.
to conduct empirical studies. The results show that CART significantly reduces the size of the regression tests (if there is not a major change of the code) while selecting all the modification-revealing regression tests. Thus, regression testing can be performed efficiently and effectively to maintain the quality of the software after changes.

4. The subject programs and the regression faults used in the studies are actual industrial programs and faults. These regression faults were identified along with the evolution processes, which demonstrates the feasibility of applying the proposed approach to real-life industrial systems.
Appendix 1: List of the types of program invariants

The type of a program invariant defines what kind of information will be learning during the program invariant analysis process. The more types of program invariant used the more program invariants can be inferred, thus they can cover more aspects of the program behavior.

The types of program invariants we used in the study are based on what the program invariant inferring tool can provide. We used Daikon to collect program invariants. We enhanced Daikon to infer the program invariants that are not provided by Daikon and collected program invariants of a program written in JavaScript programming language.

We did not use all the program invariants provided by Daikon. Many of them do not provide useful information and were excluded from our analysis.

The following is a list of type of the invariants we used in the study.

- Sequence of float value. Represents sequences of double values.
- Sequence of long value. Represents sequences of long values.
- Sequence of string. Represents string sequences.
- Lower bound of a sequence of long value. Represents that each element of a sequence of long values is greater than or equal to a constant.
- Represents that each element of a sequence of long values is not equal to zero.
- Represents that each element of a sequence of double values is not equal to zero.
- Represents sequences of long values where the elements of the sequence take on only a few distinct values.
- Subset of a sequence of string. A sequences of String values where the elements of the sequence take on only a few distinct values.
• Upper bound of a sequence of long value. A relationship that each element of a sequence of long values is less than or equal to a constant.

• Equality represents (==) relationship between adjacent elements (x[i], x[i+1]) of a double sequence.

• Inequality represents (≥) relationship between adjacent elements (x[i], x[i+1]) of a double sequence.

• Inequality represents (> ) relationship between adjacent elements (x[i], x[i+1]) of a double sequence.

• Inequality represents (≤) relationship between adjacent elements (x[i], x[i+1]) of a double sequence.

• Inequality represents (<) relationship between adjacent elements (x[i], x[i+1]) of a double sequence.

• Equality between adjacent elements (x[i], x[i+1]) of a long sequence.

• Inequality represents (≥) relationship between adjacent elements (x[i], x[i+1]) of a long sequence.

• Inequality represents (> ) relationship between adjacent elements (x[i], x[i+1]) of a long sequence.

• Inequality represents (≤) relationship between adjacent elements (x[i], x[i+1]) of a long sequence.

• Inequality represents (<) relationship between adjacent elements (x[i], x[i+1]) of a long sequence.

• Equality represents (==) relationship relationship between two double scalars.

• Inequality represents (≥) relationship between two double scalars.

• Inequality represents (>) relationship between two double scalars.

• Inequality represents (≤) relationship between two double scalars.
• Inequality represents ( < ) relationship between two double scalars.

• Equality through bit-wise operation: $x = \text{BitwiseAnd}(y, z)$ over three long scalars.

• Equality through bit-wise operation: $x = \text{BitwiseOr}(y, z)$ over three long scalars.

• Equality through division operation: $x = \text{Division}(y, z)$ over three long scalars.

• Equality through logical and operation: $x = \text{LogicalAnd}(y, z)$ over three long scalars.

• Equality through logical or operation: $x = \text{LogicalOr}(y, z)$ over three long scalars.

• Equality through logical xor operation: $x = \text{LogicalXor}(y, z)$ over three long scalars.

• Equality through Lshift operation: $x = \text{Lshift}(y, z)$ over three long scalars.

• Equality through maximum operation: $x = \text{Maximum}(y, z)$ over three long scalars.

• Equality through minimum operation: $x = \text{Minimum}(y, z)$ over three long scalars.

• Equality through mod operation: $x = \text{Mod}(y, z)$ over three long scalars.

• Equality through multiply operation: $x = \text{Multiply}(y, z)$ over three long scalars.

• Equality through power operation: $x = \text{Power}(y, z)$ over three long scalars.

• Equality through RshiftSigned operation: $x = \text{RshiftSigned}(y, z)$ over three long scalars.

• Equality through RshiftUnsigned operation: $x = \text{RshiftUnsigned}(y, z)$ over three long scalars.

• Equality through division operation: $x = \text{Division}(y, z)$ over three double scalars.
• Equality through maximum operation: \( x = \text{Maximum}(y, z) \) over three double scalars.

• Equality through minimum operation: \( x = \text{Minimum}(y, z) \) over three double scalars.

• Equality through multiply operation: \( x = \text{Multiply}(y, z) \) over three double scalars.

• Equality represents (==) between two long scalars.

• Inequality represents (\( \geq \)) between two long scalars.

• Inequality represents (>) between two long scalars.

• Inequality represents (\( \leq \)) between two long scalars.

• Inequality represents (<) between two long scalars.

• Inequality represents (\( \neq \)) between two long scalars.

• Equality represents (==) between two string scalars.

• Inequality represents (\( \geq \)) between two string scalars.

• Inequality represents (>) between two string scalars.

• Inequality represents (\( \leq \)) between two string scalars.

• Inequality represents (<) between two string scalars.

• Inequality represents (\( \neq \)) between two string scalars.

• A linear relationship between two long scalars \( x \) and \( y \), of the form \( ax + by + c = 0 \).

• A linear relationship between two double scalars \( x \) and \( y \), of the form \( ax + by + c = 0 \).

• A linear relationship over three long scalars \( x \), \( y \), and \( z \), of the form \( ax + by + cz + d = 0 \).
• A linear relationship over three double scalars \( x, y, \) and \( z, \) of the form \( ax + by + cz + d = 0. \)

• Represents the relationship \( x \geq c, \) where \( c \) is a constant and \( x \) is a long scalar.

• Represents the relationship \( x \geq c, \) where \( c \) is a constant and \( x \) is a double scalar.

• Represents the relationship \( x \leq c, \) where \( c \) is a constant and \( x \) is a long scalar.

• Represents the relationship \( x \leq c, \) where \( c \) is a constant and \( x \) is a double scalar.

• Represents long scalars that are always members of a sequence of long values.

• Represents double scalars that are always members of a sequence of double values.

• Represents String scalars that are always members of a sequence of String values.

• Represents the relationship \( x == r \pmod{m} \) where \( x \) is a long scalar variable, \( r \) is the (constant) remainder, and \( m \) is the (constant) modulus.

• A sequences of distinct long values.

• A sequences of distinct double values.

• Long scalars that are non-zero.

• Double scalars that are non-zero.

• The square relationship between two double scalars.

• Represents the zero tracks relationship between two double scalars; that is, when \( x \) is zero, \( y \) is also zero.

• Represents the BitwiseAnd == 0 relationship between two long scalars; that is, \( x \) and \( y \) have no bits in common.

• Represents the bitwise complement relationship between two long scalars.

• Represents the bitwise subset relationship between two long scalars; that is, the bits of \( y \) are a subset of the bits of \( x. \)
• Represents the square relationship between two long scalars.

• Represents the zero tracks relationship between two long scalars.

• Represents double variables that take on only a few distinct values.

• Represents long scalars that take on only a few distinct values.

• Represents long[] variables that take on only a few distinct values.

• Represents String variables that take on only a few distinct values.

• Represents String[] variables that take on only a few distinct values.

• Represents a relationship between corresponding elements of two sequences of double values (x[],y[]). For all x[i] == y[i].

• Represents a relationship between corresponding elements of two sequences of double values (x[],y[]). For all x[i] ≥ y[i].

• Represents a relationship between corresponding elements of two sequences of double values (x[],y[]). For all x[i] > y[i].

• Represents a relationship between corresponding elements of two sequences of double values (x[],y[]). For all x[i] < y[i].

• Represents a relationship between corresponding elements of two sequences of double values (x[],y[]). For all x[i] ≤ y[i].

• Represents a relationship between corresponding elements of two sequences of long values (x[],y[]). For all x[i] == y[i].

• Represents a relationship between corresponding elements of two sequences of long values (x[],y[]). For all x[i] ≥ y[i].

• Represents a relationship between corresponding elements of two sequences of long values (x[],y[]). For all x[i] > y[i].

• Represents a relationship between corresponding elements of two sequences of long values (x[],y[]). For all x[i] < y[i].
• Represents a relationship between corresponding elements of two sequences of long values \((x[], y[])\). For all \(x[i] \leq y[i]\).

• Represents a linear relationship between the corresponding elements of two sequences of long values\((x[], y[])\). Each \((x[i], y[i])\) pair is examined.

• Represents a linear relationship between the corresponding elements of two sequences of double values\((x[], y[])\). Each \((x[i], y[i])\) pair is examined.

• Represents a division relationship between the corresponding elements of two sequences of double values\((x[], y[])\). Each \((x[i], y[i])\) pair is examined.

• Represents a square relationship between the corresponding elements of two sequences of double values\((x[], y[])\). Each \((x[i], y[i])\) pair is examined.

• Represents the zero tracks relationship between corresponding elements of two sequences of double; that is, when \(x[]\) is zero, \(y[]\) is also zero.

• Represents the BitwiseAnd == 0 relationship between corresponding elements of two sequences of long; that is, \(x[]\) and \(y[]\) have no bits in common.

• Represents the bitwise complement relationship between corresponding elements of two sequences of long.

• Represents the bitwise subset relationship between corresponding elements of two sequences of long; that is, the bits of \(y[]\) are a subset of the bits of \(x[]\).

• Represents the ShiftZero relationship between corresponding elements of two sequences of long; that is, \(x[]\) right-shifted by \(y[]\) is always zero.

• Represents the square relationship between corresponding elements of two sequences of long.

• Represents the substring relationship between corresponding elements of two sequences of String.

• Represents a relationship between corresponding elements of two sequences of string values \((x[], y[])\). For all \(x[i] == y[i]\).
• Represents a relationship between corresponding elements of two sequences of string values \((x[],y[])\). For all \(x[i] \geq y[i]\).

• Represents a relationship between corresponding elements of two sequences of string values \((x[],y[])\). For all \(x[i] > y[i]\).

• Represents a relationship between corresponding elements of two sequences of string values \((x[],y[])\). For all \(x[i] < y[i]\).

• Represents a relationship between corresponding elements of two sequences of string values \((x[],y[])\). For all \(x[i] \leq y[i]\).

• Represents two sequences of long where one is in the reverse order of the other.

• Represents two sequences of double where one is in the reverse order of the other.

• Represents a relationship between a double scalar \(X\) and a sequence of double values \(Y[]\) that \(Y[i] == X\).

• Represents a relationship between a double scalar \(X\) and a sequence of double values \(Y[]\) that \(Y[i] \geq X\).

• Represents a relationship between a double scalar \(X\) and a sequence of double values \(Y[]\) that \(Y[i] > X\).

• Represents a relationship between a double scalar \(X\) and a sequence of double values \(Y[]\) that \(Y[i] \leq X\).

• Represents a relationship between a double scalar \(X\) and a sequence of double values \(Y[]\) that \(Y[i] < X\).

• Represents a relationship between a long scalar \(X\) and a sequence of long values \(Y[]\) that \(Y[i] == X\).

• Represents a relationship between a long scalar \(X\) and a sequence of long values \(Y[]\) that \(Y[i] \geq X\).
• Represents a relationship between a long scalar $X$ and a sequence of long values $Y[]$ that $Y[i] > X$.

• Represents a relationship between a long scalar $X$ and a sequence of long values $Y[]$ that $Y[i] \leq X$.

• Represents a relationship between a long scalar $X$ and a sequence of long values $Y[]$ that $Y[i] < X$.

• Represents a type of variable, for example a class object.

• Represents a structure of variable $x$. $X$ can have its own attribute, $X$ has attribute $X.attr$.

• Represents a function caller.

• Represents a API call.


