EVALUATING LEARNING GEOMETRIC CONCEPTS TO GENERATE PREDICATE ABSTRACT DOMAINS IN STATIC PROGRAM ANALYSIS

by

Patrick Chadbourne

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Computer Science

Boise State University

August 2023

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Patrick Chadbourne

Thesis Title: Evaluating Learning Geometric Concepts to Generate Predicate Abstract Domains in Static Program Analysis

Date of Final Oral Examination: 12th June 2023

The following individuals read and discussed the thesis submitted by student Patrick Chadbourne, and they evaluated the presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Elena Sherman, Ph.D.	Chair, Supervisory Committee
Jim Buffenbarger, Ph.D.	Member, Supervisory Committee
Steven Cutchin, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Elena Sherman, Ph.D., Chair of the Supervisory Committee. The thesis was approved by the Graduate College.

ACKNOWLEDGMENTS

I am grateful to many people without whom this thesis would not have been completed. Foremost among these are my advisor, Dr. Elena Sherman, for her endless determination to keep me on track and moving forward, and my mother Laurie St. Amand who played an equally important role in thoughtfully listening while I poorly explained my work over many phone calls.

I am also thankful for the members of my committee, Dr. Jim Buffenbarger and Dr. Steven Cutchin, for providing valuable insights and feedback during the formative stages of this thesis.

This thesis was made possible with the financial support of the Boise State CS department and the U.S. National Science Foundation under award CCF-19-42044.

ABSTRACT

Accuracy of static analysis over predicate abstract domains depends on the partitions of predicates. More precise predicates approximate concrete values of program variables resulting in more accurate analysis. Manual reasoning about these partitions must be done on a case-by-case basis and is time consuming and difficult.

This work explores learning geometric concepts to automate discovery of predicate domain candidates.

The proposed framework uses run-time data from program executions to gather training data for a PAC-learner to generate separating hyperplanes that can be projected onto predicate domains.

The thesis implements the framework and performs evaluations of it effectiveness on a set of benchmark programs using various test-case generation tools.

This exploratory work discusses several deficiencies in the current state of the art in test case generation, intermediate program representation, and availability of suitable program benchmarks.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The objective of this thesis is to contribute to the development of predicate domains in static analysis (SA), a method of examining code without executing it. SA employing disjoint predicate domains has the advantage of speed but necessitates a predefined domain to carry out the analysis. A randomly generated domain is an option that can be effortlessly implemented, though it often results in an imprecise analysis due to its broad and non-specific nature. Conversely, the employment of a manually created domain can yield a higher degree of precision in the analysis, as it can be tailored to the specific requirements of the program. However, this approach demands a thorough understanding of the program by the domain creator, thus adding complexity to the process. The manual creation of the domain, while beneficial for accuracy, is a time-consuming and subjective step that could potentially delay the analysis process. This work explores an approach that automatically generates predicate abstract domains using data from program executions.

1.1 Problem Background

Static analysis allows for a software engineer to reason about all possible executions of a program without executing it on every possible input. SA reasons about the semantics of the program by analyzing the structure of the code. In comparison

```
1 :func(int n){ //n = {−, 0, +}
2 :if(n < 2){ //n = {−, 0, +}
3 :n = 1/(n-2); //Warns of potential Divide-by-Zero
4 :print(n);
5 :} else { //n = {+}
6 :print(n);}
```

Figure 1.1: Code example annotated with Sign domain.

to traditional testing, SA guarantees the soundness of its results. Whereas runtime testing confirms the existence of faulty behavior after detecting it, SA guarantees the absence of undesired behavior for all program executions when it detects no faults.

This power arises from the ability of SA to reason over elements of abstract domains as opposed to concrete values. An element of an abstract domain represents a finite or infinite set of concrete values. By replacing the potentially infinite possible states of program execution with a finite number of states over an abstract domain a problem is simplified. SA uses these abstract values in place of concrete values, performing analysis on all possible execution paths by examining the abstract values that represent them. SA can then determine whether certain properties exist or assertions hold for the abstract elements. These properties can then exist within the possible concrete values represented by the abstract elements.

Although many different abstract domains exist, this work specifically makes use of predicate domains due to their efficiency [13]. To create elements of a predicate domain, concrete values are partitioned into subsets defined by constraints on concrete values. For example, partitioning the set of all integers into the subsets of integers less than, equal to, and greater than zero creates the Sign abstract domain: $\{-, 0, +\}$. Although the predicate domain provides much greater efficiency by limiting the number of elements that must be analyzed, the accuracy of the analysis is limited by the precision of the partitions used.

```
1 :func(int n){ //n = {< 2, 2, > 2}
2 :if(n < 2){ //n = {< 2}
3 :n = 1/(n-2); //No over-approximated warning
4 :print(n);
5 :} else { //n = {2, > 2}
6 :print(n);}
```

Figure 1.2: Code example annotated with Predicate domain with hand-picked partition at 2.

To demonstrate the effects on SA accuracy that differing partitions provide, let us examine two identical code snippets with a naive partition and a manual partition. In Figure 1.1 we see simple code with comments stating the possible abstract values of n. At the start of `func` SA knows nothing about the incoming value of n and annotates it with negative, positive, and zero abstract values. On line 2 SA analyzes the true branch and determines that n must be less than 2. However since the partition cannot express it exactly, SA soundly approximates it with positive, negative, and zero values again. At line 3 SA issues a potential warning that the code has a division-by-zero failure. This is because SA includes $n = 2$ as a possible value since the concrete value 2 at which the failure occurs is represented by the positive abstract value $\{+\}$.

This over-approximation causing inaccuracy in SA can be mitigated by using a more precise partition for this code. For example, in Figure 1.2 abstract values are predicated on their relation to 2. By changing the partitions used in the analysis, the abstract value of the true branch on line 2 now accurately represents all possible values of n as less than 2. Since values less than two avoid the division-by-zero failure, SA removes the false-positive warning.

In this code example, it is trivial to determine that the second partition choice is better than the first, especially when one of the relational operands is a constant. However for more complex programs identifying best partitions manually is chal-

lenging, especially when both operands are variables. This work seeks a reliable and efficient method to produce precise partitions automatically without requiring a priori knowledge of a program or manual code inspections.

Although there are many ways to naively determine predicate elements such as searching for code constants, they are often limited in their accuracy when handling relationships between multiple variables. This work attempts to infer such partitions by analyzing concrete values that are assigned to variables during program execution. Using them as geometric coordinates in a space where each dimension corresponds to a variable, the proposed technique attempts to generate partitions by learning geometric concepts: abstract shapes or regions that encompass points.

1.2 Geometric Concepts

Geometric concepts can represent an abstract state, e.g., $n \mapsto \{+\}$ by mapping its set of concrete (e.g., positive) values to a region on the n axis for that variable. For example, analysis of two variables results in regions in a two-dimensional space. Increasing the number of variables being analyzed likewise increases the dimensionality of the space holding the regions. Previous work [3] developed algorithms that learn the shape of data points. Following the same route, we can determine suitable partitions that SA can use to compute abstract values.

To demonstrate this concept, consider Figure 1.3 that shows a two dimensional space where concrete values of variables u and w are plotted on cartesian coordinates. These data points are labeled as either "true" or "false" depending on the corresponding branch taken at a conditional statement like $u < w$. Therefore all data points in the shaded region that are labeled as true result in the program executing the true

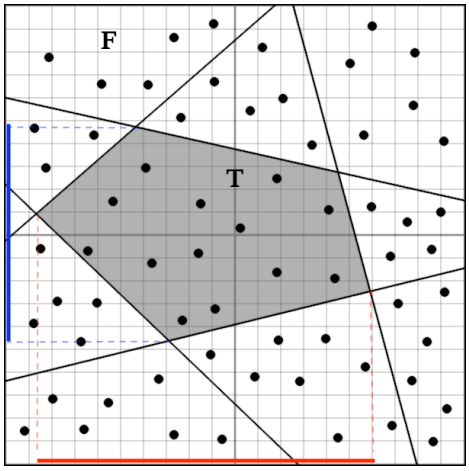

Figure 1.3: Example of a Geometric Concept Used to Partition Concrete Values in Two Dimensional Space.

branch of the conditional statement. The shape of the shaded region represents the geometric concept that is learned by the algorithm. The lines forming the shapes and their intersections correspond to the inferred relational abstract partitions. To find predicates, the shaded region is projected onto u and w axis as show in the figure in red and blue lines, respectively. Then a predicate domain consists of three predicates, "before" the start of the line, the line itself and "after" the end of the line.

This approach gathers concrete variable values as data points through instrumented program executions and passes them into an algorithm that learns a geometric concept representative of that data. The algorithm uses a Probably-Approximately-Correct (PAC) learner [15], which requires a sufficiently large sample of data to accurately compute the representative geometry. Once generated, the resulting region is then projected onto the axis to create a predicate domain with the partitions that precisely separate values in conditional statements.

1.3 Proposed Solution

In order to improve the efficiency of creating domains for use within SA, we examine a novel technique for using PAC learning algorithms and dual space projections to learn geometric concepts that can be interpretted to generate a predicate domain using input data sampled from program executions. Through developing and evaluating this approach, we seek to validate the following thesis statement:

1.3.1 Thesis Statement

How suitably can an algorithm using PAC learning techniques provide precise predicate abstract domains for static analysis by projecting 2D geometric concepts onto 1D abstract domains.

As a means to support the thesis statement, the following research questions will be investigated and answered:

1.3.2 Research Questions

RQ1 How can data be efficiently gathered from programs to provide samples for the PAC learner?

RQ2 How does sample size affect the output of the PAC learning algorithm?

RQ3 Can the PAC learning algorithm produce predicate abstract domains on a set of benchmark programs?

The thesis is organized as follows: Chapter 2 includes a relevant background on the techniques being used in this work and how they are used in existing research, including current SA techniques and PAC learning algorithms. Chapter 3 provides a

high-level overview of the domain generating framework from the initial user provided program to the resulting generated domain. Chapter 4 examines this framework in greater detail, covering how program instrumentation and data gathering occur, as well as the novel dual-space PAC learning algorithm and the resulting domain synthesis. Finally Chapter 5 examines the completed framework and its results when used to generate domains for our input set of test programs. The thesis' findings are compiled in Chapter 6 where the answers to our research questions are provided and potential extensions of this work are proposed as future research topics.

CHAPTER 2

BACKGROUND AND RELATED WORK

The subsequent chapter reviews the necessary background that supports our thesis in topics covering static analysis, PAC learning, and program verification. Prior work in these areas provides methodologies for affirming program behaviors through data flow analysis, employs statistically supported machine learning techniques to learn the boundaries of regions separating sets of data, and extending PAC learning concepts to facilitate direct reasoning about program verification.

2.1 Static Analysis with Predicate Abstract Domains

The goal of a data-flow program analysis A is to compute program invariants that are expressed as elements of its abstract domain $D = \mathsf{po}\langle\mathcal{D}; \sqsubseteq, \sqcup\rangle$, where $\sqcup$ is the least upper bound operator. An instance of an analysis A is defined by the following [7] :

- A complete lattice D (satisfying ascending chain condition), that describes the abstract domain of A.
- A flow F of program P's statements, for example P's control flow graph.
- A set of monotone transfer functions $\mathcal{F}$ for each statement $l \in F$ that maps an element of D to itself, i.e., $f_l \in \mathcal{F} : \mathcal{D} \mapsto \mathcal{D}$.

- A set of external statements E in P, which can be either the entry node $entry(Program)$ or exit nodes $exit(Program)$.
- An initial value $\iota \in \mathcal{D}$ for statements in E.

Then the set of equations for A is defined as follows on entry and exit of each statement $l \in F$:

$$A_{in}(l) = \bigsqcup\{A_{out}(l') \mid (l', l) \in F\} \sqcup \iota^l_E \tag{2.1}$$

$$\text{where } \iota^l_E = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases} \tag{2.2}$$

$$A_{out}(l) = f_l(A_{in}(l)), l \in F \tag{2.3}$$

Where $\sqcup$ is the least upper bound operator, $\bot$ is the bottom element of $\mathcal{D}$ with the following properties $\forall d \in \mathcal{D} : \bot \sqcup d = d$ and $\forall (l) \in F : f_l(\bot) = \bot$. The above set of equations is solved using a fixed-point computation such as a work-list algorithm that iteratively recomputes incoming and outgoing flows for each statement, i.e., $A_{in}(l)$ and $A_{out}(l)$ until it detects no changes in flows of data.

Depending on the type of analysis, $\mathcal{D}$ can have different representation. For example, in the abstract domain D_{LV} of Live Variable analysis, which reasons about program variables, $\mathcal{D}$ is the power set of P's variables Var with $\sqcup$ being the set union operation, $\bot$ being $\emptyset$ and $\top$ being Var. For *Sign* analysis, $\mathcal{D}$ in D_{SA} is a set of maps $\mathsf{Var} \to \mathcal{P}(\mathbf{Sign})$ where as before $\mathbf{Sign} = \{-, 0, +\}$. Usually, an element of $\mathcal{D}$ that A computes for a given program location is referred to as an abstract state $\sigma \in \mathcal{D}$.

Since the focus of this work is generating partitions for predicate domains, we give detailed description of the domain $D_{Pred} = \mathsf{po}\langle\mathcal{D}; \sqsubseteq, \sqcup\rangle$. $\mathcal{D}$ is the power set of

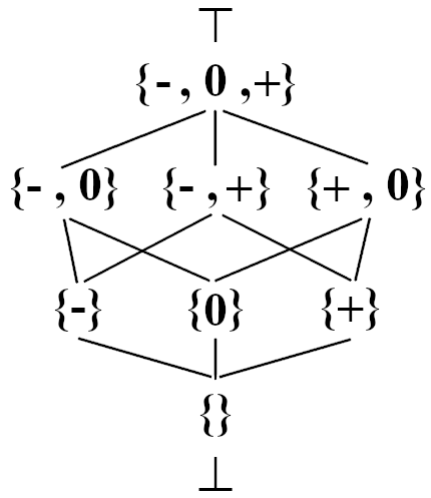

Figure 2.1: Lattice of the Sign domain.

all predicates with elements p_1 through p_k. $\sqsubseteq$, the subsumption operator, is $\supseteq$ which defines the ordering of the lattice. If an element p_3 is a superset of the element p_2 then p_3 will be a higher order on the lattice. $\sqcup$, our least upper bound operator is the set union operation. It ensures that all predicates in differing data flows are preserved when merged, i.e., if values in a certain range are valid during program execution, then data flow merges keep those valid values after merges.

The initial values, ι, is the set of all available predicates at $entry(Program)$ as this is unconditional analysis where the incoming values are not constrained. At nodes that are not the entry point the initial value of $\perp$ will be the empty set. The set of values in $A_{in}(l)$ has the transfer function f_l applied to it which results in $A_{out}(l)$. These transfer functions are defined for each type of statement and reflect the changes of values against the predicates representing them [13].

2.2 PAC Learning of Geometric Concepts

PAC learning algorithms work on the principal that with a sufficiently large sample size the algorithm is able to return a result that is accurate within a statistically insignificant margin of error [3]. PAC learners are capable of learning various geometric concepts by categorizing data as either inside or outside of the geometric shape from which the data is gathered. Baum [1] and Blumer et al. [2] both presented similar methods for PAC learning unions of half-spaces. Blumer et al. [2] also provided an algorithm for PAC learning unions of axis-aligned rectangles.

Algorithm 1 Pseudocode for PAC learning algorithm

function $Learn - C_s^d(s, \epsilon, \delta)$
    Draw labeled sample S of size
        $m = O(\frac{1}{e}\lg\frac{1}{\delta} + \frac{ds}{\epsilon}(\lg\frac{ds}{\epsilon})^3)$
    Create U by adding edge for every +/- pair in S
    $\mathcal{F} \leftarrow \emptyset$
    Create a geometric dual space containing a hyperplane
        for each point in S
    Calculate centroid γ of each feature in the dual space
    $\mathcal{F} \leftarrow \mathcal{F} \cup g'(\gamma)$
    $F \leftarrow$greedy covering of $\mathcal{U}$ using $\mathcal{F}$
    $h \leftarrow$regions formed by F
    for each region $\rho \in h$ **do**
        if $\rho \cap S \neq \emptyset$ **then**
            label ρ according to S
        else
            label ρ arbitrarily
        end if
    end for
    Return h
end function

Bshouty et al. [3] expanded upon these methods by providing an algorithm for PAC learning arbitrary boolean combinations of half-spaces. These general geometric concepts defined by arbitrary boolean combinations of half-spaces provide the

necessary flexibility to describe relative concrete values of variables during program execution and can therefore be applied to problems relating to program verification.

Algorithm 1 presents pseudocode from Bshouty et al. [3] for finding region information from point data, on which our implementation is based. Lines 2 and 3 draw sample data points from a source until m points are gathered. The value of m is calculated so that it is of sufficient size to return results that are accurate within a statistically insignificant margin. Line 4 creates a set of edge data, U, between positive and negative points that later is used to determine when those points are properly separated.

Lines 5 through 9 populate the set of separating edges. This process begins with line 6 converting the point data from our sample S into hyperplanes in a dual space (this process is explained in further detail in Chapter 3). On the dual plane the intersections of hyperplanes create features from which centroids are calculated. Line 9 converts the centroids, which are points in the dual space, to a hyperplane on the primal space and adds them to the set $\mathcal{F}$ of these calculated hyperplanes.

After all centroids have been calculated and converted into primal space hyperplanes, the set U is greedily covered by $\mathcal{F}$. Each hyperplane in $\mathcal{F}$ covering an edge in U via intersection is added to F. On line 11, the remaining hyperplanes from the greedy set covering are then used to divide the primal space and label the resulting regions as positive or negative. As all edges between positive and negative points have hyperplanes from $\mathcal{F}$ separating them, the remaining regions only have points with either a positive or negative label. Then in lines 12-18 these regions are labeled with the same label as the points contained within, and regions that are empty are labeled arbitrarily. The algorithm returns these regions as its result.

2.3 Related Work on PAC learning in Verification

Sharma et al. [11] pioneered using PAC learning in program verification. In their work, researchers use Algorithm 1 to distinguish between reachable states of a program and states that violate specific properties. This approach generates inductive assertions that describe those "bad" states that cause property violation. However, this technique requires additional steps to insure that generated invariants are valid. Generation of invalid invariants restarts execution with additional data to improve the precision of the PAC learning algorithm. The same authors extended their framework to make qualitative comparisons of the precision of different abstract domains [12].

Another study [5, 10] applies the PAC learner algorithm alongside other algorithms to generate assertions at specified points in a target program. These are found by matching program features to set templates using either a PAC learner or a SVM-based learner. While the approach uses the PAC learner to determine boolean combinations of half-spaces, it applies the SVM-based learning technique to find conjunctions of linear equalities. Both learners are then used to produce the desired assertions. These assertions can be generated at specific points in a program by annotating the code at the desired assertion location.

The most recent work [9] uses the same PAC learning algorithm to find program invariants but in a context of different program features. Using extracted data from the target program before and after method calls, the technique creates a representation of a memory graph to learn pre and post conditions of those methods. The study specifically focuses on methods in heap-manipulating programs due to their complexity. By representing the data stored in the heap as a graph and examining how that graph is transformed between the pre and post conditions, the PAC learning

algorithm is able to classify method behaviors. Similarly to the previously described work this recent paper directly learns program invariants which can then be used for program verification as an oracle to the expected behavior. Direct invariant generation has the same drawbacks as in the above research where soundness is not guaranteed by dynamic data.

Our proposed work differs from previous related work in the way these approaches learn different features of target programs or by directly learning program invariants. Although still using the same PAC learning algorithm, the data collection and resulting learned features differ. Our work focuses on obtaining invariants at conditional statements, i.e., predicates using Algorithm 1, taking the concrete values of variables at those conditional statements as well as labels based off of the result of the conditional expression evaluation. This data is then used to find regions that can be projected onto potential predicates that then can generate invariants after the fact through static analysis techniques that use those predicates as their abstract domains. Chapter 3 describes the methodology and implementation details.

CHAPTER 3

APPROACH

Figure 3.1 visualizes a high-level overview of generating invariants from an input program. The framework follows a sequential path with the output from prior components being shown as following the arrows to the succeeding components. The process requires two separate pieces of information to begin: the program under analysis, shown in the boxes labeled 'Program', and a suite of JUnitTests, in the box labeled 'T'. The end result of the process is the performed static analysis using the generated domain, shown in the box labeled 'Analysis'. In a real-world use-case this analysis would reflect the needs of the user performing it, however for the purposes of this work's evaluation, the analysis was limited to a comparison of precision between the new domain and a pre-existing random domain.

At Step 1 in the figure, the input program is instrumented so that labeled data is gathered during executions of branch statements. The program's Java source code is provided as input to an instrumentation program that uses the SOOT Java Framework [16]. This instrumentation program iterates across all methods within the input program and across all conditional statements within the methods. The instrumentation alters the conditional statements so that the values stored in variables referenced by the conditional are saved in an external reporting class. The variable data is paired with positive and negative labels that denote whether the conditional

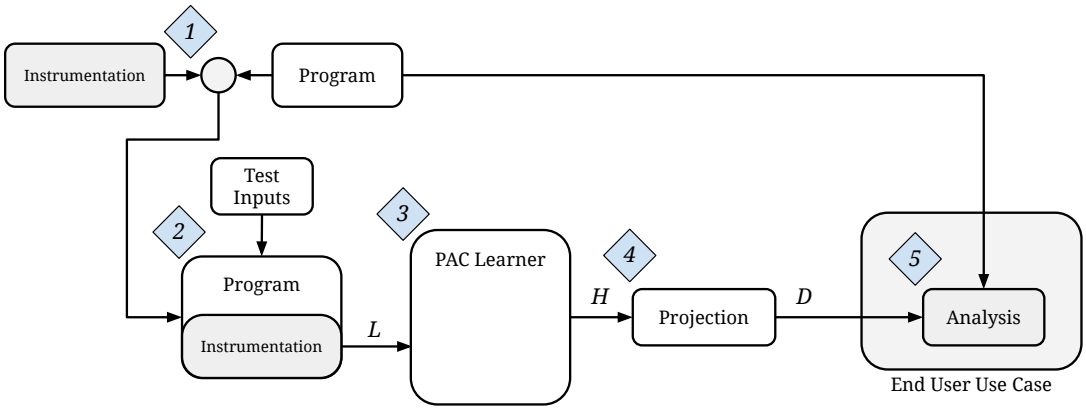

Figure 3.1: High level overview of invariant creation approach.

evaluated is true or false. The instrumentation program also finds the points where the code exits the main method and inserts an additional instruction to write the collected labeled data to an output file where it will be available for later use.

With the input program compiled with the instrumented code it continues to Step 2 where the provided testing suite executes the instrumented code. Although any executions of the instrumented code provide output data, JUnit tests are used for their convenience. A parameterized test that executes the instrumented methods across a wide range of random input values allows a large amount of data to be gathered quickly without requiring the user to manually execute the code. Each successive run of the testing suite provides more sample data that is appended to the instrumentation's output file. This labeled sample data, 'L', is then ready to be used in Step 3.

The PAC learning algorithm takes the labeled sample data as an input and performs a dual-space to form geometric concepts that attempt to capture the behavior of the instrumented code. This process is covered in much more detail in Section 4.3. The geometric concepts are sets of hyperplanes that form labeled regions separating

the input data. As these hyperplanes are much more information-dense than the partitions used in predicate domains they must be projected into predicates.

Step 4 takes the set of hyperplanes, 'H', and computes corresponding predicate partitions with which the final output of the domain, D, can be constructed. As a hyperplane represents a two dimensional line and partitions act as corresponding one dimensional points, they must be sufficiently simplified to be used as output. Hyperplanes with high-magnitude slopes, lines that are nearly vertical, are used to find partitions in the variable corresponding to the geometric concept's Y axis. Inversely, hyperplanes with near-zero slopes, lines that are nearly horizontal, are then used to determine partitions on the X axis' corresponding variable. Hyperplanes that are very near to a slope of 1 or -1, those that represent diagonal lines, are not used as partitions as there is no feasible way to adequately capture the information they represent. These diagonal lines are instead used identify the usefulness of the predicate domain for performing analysis on the input program. Diagonal hyperplanes that successfully separate a majority of the labeled sample data suggest that the relationship between the two variables is purely relational, which is beyond the abilities of analysis using predicate domains. For these programs a different approach is necessary to achieve a more precise analysis.

If Step 4 is completed with a successfully generated predicate domain then the user is able to utilize that domain for their analysis in Step 5. For the purposes of this thesis, the generated domains are used to find program invariants which are then compared against the invariants found by a randomly selected pre-existing domain. If the generated domain results in more precise invariants for the program under analysis than the randomly selected domain then the process is considered a success. Conversely if the invariants are less precise it is a failure. The invariants can also

be equally precise or incomparable, and for both of these situations the result is ambiguous, with the presented framework offering no clear benefit.

CHAPTER 4

IMPLEMENTATION

In order to realize the above framework each individual component is implemented separately as a standalone Java program. Program instrumentation is handled using the SOOT Java framework, jUnit tests are generated and ran to gather the sample data for the PAC learner, and the PAC learning algorithm itself is a Java program that takes that sample data as input and outputs the learned geometric concepts. As no additional outside input is necessary for projecting the geometric concepts into predicate domains that step occurs simultaneously, using the data available to the PAC learning algorithm to assist with the projection. With all steps implemented and executed the final result is either a message explaining that the learned space is likely relational and cannot have a predicate domain generated for it, or the generated predicate domain itself. The following sections provide implementation details of each component.

4.1 Program Instrumentation

Prior to generating a domain using our PAC learning algorithm, it is necessary to gather sample data to learn from. The data is gathered through instrumentation, which inserts additional statements to the compiled code. By modifying the program under analysis in this way, we create a compiled program that behaves as originally

intended while also collecting data by executing the instrumented instructions. This ensures that the outcome of the execution is not modified and therefore the data gathered accurately reflects the information that is required to train the PAC learning algorithm.

For running the PAC learning algorithm we require sampled data of the values of variables in the expression of a conditional statement, as well as whether the expression evaluation has resulted in a "branch-out" or a "fall-through". Branches jump to a label in the following block when the conditional statement of the branching operation evaluates to true, and fall through into the block when the conditional statement evaluates to false. For example if during a program execution a conditional statement compares a variable `x` to a variable `y`, both of which are integers with the values of `3` and `5` respectively, and the code branches out, it is necessary to for the instrumented code to store the values of `3` and `5` as well as a label to denote that the conditional statement resulted in a branch-out.

Sampling the variable values is a fairly straightforward task with Soot, as the three-address code for an `If Statement` in Jimple refers to both of the operands, from which their values can be easily attained. Determining the fall-through status of the condition is more complicated as that is not information that is easily accessible through Soot. In Figures 4.1, 4.2, and 4.3 a code snippet is examined in its original Java, the intermediate Jimple representation, and the Jimple after the instrumentation occurs. Figure 4.1 is straightforward with a branch beginning on line 3. We will want to instrument this code so that when executed the values of x and y are saved, along with whether the branch was taken or not. The Jimple code in Figure 4.2 is similarly simple, however there are some important differences to note. Line 3 of Figure 4.2 is still the conditional statement, however now the condition has been

```
1 : int x = arg0;
2 : int y = arg1;
3 : if(x < y){
4 :    System.out.println(x + y);
5 : }
6 : return;
```

Figure 4.1: Code snippet prior to instrumentation

```
1 :    i0 := @parameter0: int;
2 :    i1 := @parameter1: int;
3 :    if i0 >= i1 goto label1;
4 :    virtualinvoke r1.<java.io.PrintStream: void println
5 :    goto label2;
6 : label1:
7 : label2:
8 :    return;
```

Figure 4.2: Intermediate representation of code snippet

flipped so that instead of $x < y$ it is $x(i0) >= y(i1)$. This is so that when the original Java code in Figure 4.1 would fall-through on a true evaluation, Jimple instead will branch out of the block on a false evaluation. Lines 6 and 7 of Figure 4.2 also hold an important detail in that a label is created for both stepping over the conditional block, and for completion of the conditional block. This is equivalent to giving all if statements in Java an empty else block when one is not already specified.

Figure 4.3 shows the completed Jimple code with the additional instructions added by the instrumentation. Line 3 now has a static invocation of `setData`, line 5 the `setFlag` invocation, and 10 the `storeArgs` method. The `setData` method is always called before the if statement so that the data is unaffected by the result of the statement. `setData` takes `i0` and `i1`, our x and y values, and stores them in an external object's variables. The `setFlag` method is called inside of the conditional block and tells the instrumented code to report the stored variable values as having resulted in a 'true' evaluation. After this both branches converge at `label2` on line

```
1 :    i0 := @parameter0: int;
2 :    i1 := @parameter1: int;
3 :    staticinvoke <void setData(int,int)>(i0, i1);
4 :    if i0 >= i1 goto label1;
5 :    staticinvoke <void setFlag()>();
6 :    virtualinvoke r1.<java.io.PrintStream: void println
7 :    goto label2;
8 : label1:
9 : label2:
10 :    staticinvoke <void storeArgs()>();
11 :    return;
```

Figure 4.3: Jimple representation after instrumentation

9, with only 'true' evaluations having called `setFlag`. The values in the conditional statement as well as the value of the flag are then both added to the final set of values on line 10.

Algorithm 2 Pseudocode for Branch Instrumentation algorithm

```
 1: class BranchInstrumenter extends BodyTransformer
 2: Initialize reporterClass, setData, setFlag, storeArgs, report
 3: function INTERNALTRANSFORM(body, phase, options)
 4:     for stmt in body.units() do
 5:         if stmt is an instance of IfStmt then
 6:             condition ← stmt.getCondition()
 7:             insertBefore(setData, condition.LHS, condition.RHS, stmt)
 8:             insertAfter(setFlag, stmt)
 9:             insertAfter(storeArgs, stmt.getTarget())
10:         end if
11:     end for
12:     if body.isMain() then
13:         for each stmt in body's units do
14:             if stmt is an instance of ReturnStmt then
15:                 insertBefore(report, stmt)
16:             end if
17:         end for
18:     end if
19: end function
```

In order to perform the instrumentation and achieve the desired results an algorithm is used that iterates over the Jimple representation of the code, inserting

additional statements where necessary. Psuedocode for the algorithm created for this purpose is shown in Algorithm 2. The code itself extends Soot's `BodyTransformer` class, which is the Soot framework's tool for making changes to a `Body`, Soot's internal representation of a method's attributes, including the statements that the method is comprised of. Line 2 initializes a `reporterClass` object as well as four methods that the `reporterClass` has: `setData`, `setFlag`, `storeArgs`, and `report`. The `reporterClass` is a helper class that contains a data structure for keeping track of the data that is being gathered while the instrumented code is executed. It also contains the aforementioned methods that Soot treats as objects to pass as parameters to its `insertBefore` and `insertAfter` methods.

As we are using a `BodyTransformer`, we already know on Line 3 that we have access to the `Body` being transformed, and on Line 4 can begin iterating over all of the statements within the method. Line 5 checks each statement in the method's body to see if it is an `IfStmt`, which is Soot's internal representation for all conditional statements. Once we know the current statement is a conditional statement, we can use the `getCondition` function to get all of the relevant data to the condition itself on Line 6. Lines 7 through 9 then call Soot's aforementioned `insertBefore` and `insertAfter` methods. As we know that the instrumentation is currently on the conditional statement, usage of `insertBefore` results in the inserted method being called immediately before the condition occurs. This is used on Line 7 to collect the two values being compared in the conditional statement's `condition`. This takes care of gathering the variable values involved in the conditional statement, but it is still necessary to determine whether the result is a "branch-out" or a "fall-through".

By using insertAfter on Line 8, we add the `setFlag` method to the program's execution at the code point immediately following the conditional statement. As

a conditional statement jumps to a different code point when execution branches out this results in the `setFlag` method only ever being called when a conditional statement results in a fall-through. The method itself then simply sets a flag in the `reporterClass`.

The final method being inserted into the program execution at this point is the `storeArgs` method on Line 9. Although it also uses Soot's `insertAfter` method, it does not take the current statement as a parameter, but the statement's target `stmt.getTarget()`. An added benefit of knowing that the current statement is an `IfStmt` is that it has a target, which is the code point that execution jumps to when the conditional statement results in a "branch-out". In the case where the target is earlier in the code execution than the current statement (such as during loops), it is worth noting that a "fall-through" also reaches the target's code point, albeit after the "fall-through" has been executed. Since both a "branch-out" and a "fall-through" result in execution reaching the conditional's target, this is a safe place to call the `storeArgs` method which saves the stored variable values along with whether the flag was set or not.

Finally the algorithm checks on Lines 12 and 14 if the current statement is the `return` of the `Main` method, and if so it inserts the `reporterClass`'s `report` method that outputs all of the gathered data to a text file for later use.

4.2 Data Gathering

Once the program under analysis is properly instrumented, it is run on test inputs to create our sample data L. The quality and quantity of test input is important to the final process as it must cover a sufficiently wide range of inputs. Without sufficient

data the PAC learning algorithm is unable to accurately and precisely determine where to place separations, both due to lack of dual-projected lines with which to separate, and due to the sparseness of the data being separated. To acquire the desired data, test cases must execute the instrumented code segments and gather varied sample data. Although traditional static analysis does not require execution of the code during the analysis, the machine learning aspect of this thesis makes it essential to gather training data at runtime.

As mentioned this thesis uses test cases as a means of executing the instrumented programs and gather the necessary sample data, however as traditional Java testing is focused on validating or invalidating assertions, our needs differ from those of traditional testing. As we generate tests to execute Java code the majority of our test generation methods also generate assertions, however this is a byproduct of the process. Ideally, tests should run as much of the code as possible while using the same test inputs to increase the likelihood of executing internal methods with valid inputs. While more specific tests that execute a single method are still useful, they may contribute data that is less likely to be encountered during full program executions.

Due to the complexity of the task and the need to find effective techniques for various programs, multiple test generation approaches are employed. Two automatic test generators, Evosuite [4] and Randoop [8], are used for this purpose. Both tools analyze provided compiled Java classes and output suites of JUnit [14] test cases that execute the methods in those classes. To gather more varied data, additional test cases can be generated, or existing test cases can be modified with different input values. In many cases, it is more efficient to reuse existing test cases and provide new input values for testing. JUnit-QuickCheck [6], a library that implements property-based testing for JUnit, is used to supply these new input values to our previously created

tests.

4.2.1 Evosuite

Evosuite is an automated test case generation tool for Java programs, primarily aimed at discovering faults through satisfying coverage requirements [4]. It employs a genetic algorithm to generate test cases, mutating test suites to maximize coverage of the target code. Evosuite has a number of varying fitness functions that guide the search process toward covering as many execution branches as possible. As it is necessary for the PAC learner to have data gathered from executions of both branches of a conditional statement, this maximization strategy directly supports our use-case for the generated tests.

Despite its powerful test case generation capabilities, Evosuite also provides challenges in its use and incorporation into our pipeline. One of the difficulties is the generation of invalid test cases. Since Evosuite's primary focus is on achieving high code coverage, it may sometimes produce test cases that do not adhere to the constraints or requirements of the system under test. One such test that is common for Evosuite to generate provides varied input data but does not execute the method

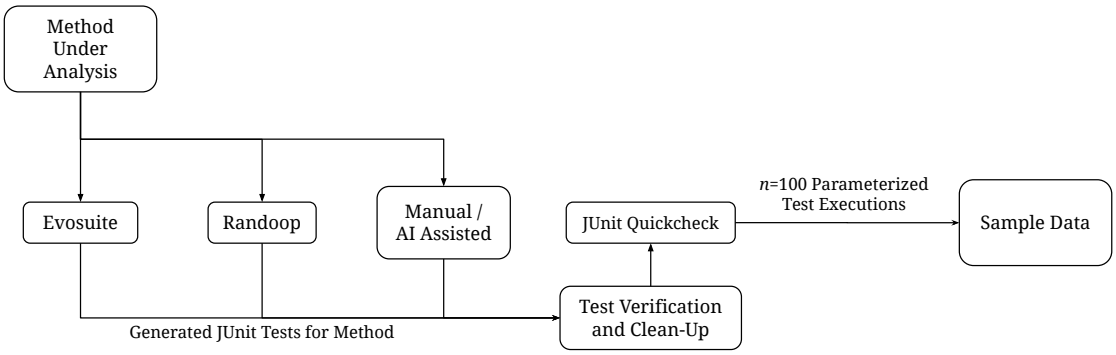

Figure 4.4: High level overview of sample data gathering process.

using it, instead asserting that the supplied value remains unchanged. Although a valid test to uncover and report methods that have unintended side effects, the result of running it provides duplicate sample data for executions across all inputs.

This necessitates manual inspection and pruning of the generated test cases to ensure their validity, with a high number of programs not receiving any worthwhile test cases after the generation process completes. This is partially offset by performing manual code cleaning before providing it to Evosuite, as it may struggle when analyzing code with complex data structures or code that relies heavily on external libraries. It is occasionally necessary to strip out some of these aspects before Evosuite would run successfully. Doing so, however, runs the risk of altering the behavior of the method being analyzed and may lessen the reliability of the gathered data.

Ultimately Evosuite is a research tool and does not have the same reliability as an industry tool, and as such suffered from numerous crashes or other failures to generate tests depending on the complexity of the input code. This limitation made the effective use of Evosuite difficult as a general solution for test input generation across a wide range of programs.

Despite these challenges, Evosuite remains valuable for automated test case generation. Its ability to generate a diverse set of test cases that cover a wide range of code branches makes it an essential component of the test generation process. However, due to these aforementioned issues, additional tools are also employed for cases when Evosuite was insufficient.

4.2.2 Randoop

Due to the high number of programs where Evosuite is either unable to generate any tests or the generated tests are unusable for gathering training data, Randoop [8] is

considered as an alternative.

Programs where Evosuite fails are then provided to Randoop [8] to generate tests. Randoop is another automated test case generation tool for Java programs, which focuses on generating unit tests by employing a feedback-directed random test generation approach. Randoop's methodology consists of iteratively building test cases, executing them, and using the results to guide the generation of further test cases. It employs heuristics to increase the likelihood of generating test cases that reveal faults, aiming to improve software quality.

Similarly to Evosuite, Randoop introduces its own challenges. One of the main issues faced while using Randoop is the generation of trivial test cases. When used across our collection of sample programs, Randoop frequently produces test cases that do not meaningfully execute the conditional statements being analyzed, similarly to Evosuites generation of side-effect tracking tests. This results in the majority of runtime data collected from execution of Randoop-generated test cases either being duplicate, such as sampled data consisting of hundreds of entries of the exact same value pair and label, or being entirely random, where the method itself has no influence over the gathered data and further evaluation shows more about Randoop's number generation than the program being analysed.

Unfortunately despite promising results with initial experimentation, the challenges and failed attempts at test generation outweighed the sporadic successes. As such Randoop is not used for any test generation that resulted in training data that was passed to the PAC learning algorithm. With automatic solutions proving infeasible for large-scale test generation of random programs, it was necessary to rely on test generation methods more tailored to the source code.

4.2.3 Manual and Semi-Automated Test Generation

In situations where prior methods fall short in generating adequate test cases, manual test case generation becomes necessary to allow the collection of meaningful runtime data for the PAC learning algorithm. Writing test cases manually, while time-consuming, ensures the creation of test cases that are better suited to gathering training data for the PAC learning algorithm. In this thesis, this process is accelerated by leveraging the capabilities of Chat-GPT, an AI language model trained to understand and assist with a wide range of tasks, including code analysis and test case generation. Chat-GPT can offer suggestions for test cases or specific input values that are more likely to exercise different execution paths and uncover potential faults, thus improving the quality and diversity of the generated test cases. Although the results when using a Large Language Model such as Chat-GPT are unpredictable, it is often possible to achieve valuable results through prompt engineering, an interative process of tailoring requests to better address the short-comings of previous responses.

The following example is a generic prompt that can be given to Chat-GPT alongside the source code of the method in question:

```
Write a JUnit test for the following method:
```

The jUnit tests that resulted from the above prompt included all necessary boilerplate code and would compile and run without issue. Chat-GPT also provides seemingly random input values for the methods being tested, satisfying the need of test generation process by providing varied inputs for the instrumented programs so that they generate training data.

It is worth noting that all methods of test case generation, including those produced by Evosuite, Randoop, and Chat-GPT, require some level of manual cleaning

and inspection to ensure the validity and relevance of the generated test cases. This manual effort is crucial for maintaining the overall quality of the test suite and ensuring that the collected runtime data is reliable and useful for the PAC learning algorithm.

4.2.4 JUnit-QuickCheck

In addition to manually cleaning generated and written test-cases, it is necessary to run those test cases across a wide variety of valid values so that the generated data is not directly limited to the number of available test cases. JUnit-QuickCheck is a library that integrates with JUnit and implements property-based testing, allowing tests to be executed on a range of parameterized inputs [6]. Running the generated jUnit tests across these parameterized inputs allows a handful of tests to provide hundreds of inputs for the methods being analyzed.

By specifying properties that input values should satisfy, such as constraints on the range or type of values, JUnit-QuickCheck quickly generates a variety of inputs that meet these conditions. Once the test cases are executed they are run repeatedly with different values for the parameterized inputs until the set conditions are satisfied. As the data gathered by this process depends on the structure of the software itself, the parameterized tests can then be run again until a sufficient amount of data for all branch executions is generated with each iteration providing new randomized inputs within the parameters. This process of gathering a wide range of parameterized data makes it possible to observe the behavior of conditional statements using the PAC learning algorithm, and facilitates the generation of a wide array of test inputs upon which to run our previously generated testing suite.

The parameterization process is fairly trivial to do manually, however as ChatGPT is already being utilized to generate the tests themselves, a simple change to the above prompt results in the generated test cases already having the necessary boilerplate code and changes to provide parameterized inputs:

```
Write a JUnit test for the following method that uses jUnitQuickcheck
to constrain the methods parameters between -1000 and 1000:
```

This reliably results in a jUnit test that can immediately be run that uses jUnitQuickcheck to select random input variables for the methods being tested. One execution of the jUnit test results in 100 executions of the parameterized test, which then results in a varied amount of training data from the instrumented program depending upon how often the instrumented instructions are run during a single method call.

4.3 PAC Learning Algorithm

With the sampled data L from our test executions, we can provide the PAC learning algorithm with the input for generating regions. The PAC learning algorithm accomplishes its task by utilizing dual space projection to create encapsulating regions out of sampled data. There are two spaces, the original (x, y) plane is referred as a primal V' , and the new space (u, v) is the dual space V^*. When a point on a two dimensional plane is projected onto a dual space it becomes a line on that dual plane. That is, this projection takes a point with $(2, 3)$ coordinates and places them in the slope-intercept formula for a line, with the value of x coordinate becoming the slope and the value of y coordinate being negated and used as the intercept, i.e., $v = 2u - 3$.

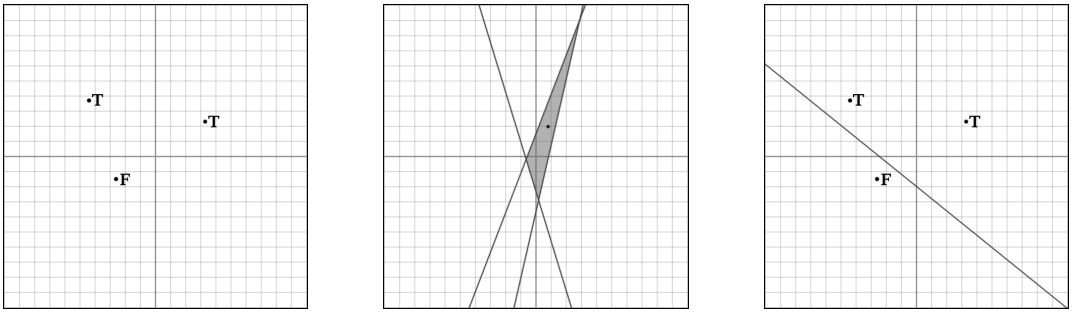

Figure 4.5: Visual example of the dual space projections used in the PAC Learning Algorithm

Using this approach, the sampled data L is first projected from the primal space V' to the dual space to form a set of lines which separate the dual space V^* into regions R^*. A region $r \in R^*$ is defined as a conjunction of polyhedra, which we form them by points where lines in V^* intersect. In order to "separate" points in V' we need to find centroids of each region in R^*. Given a region $r \in R^*$ that is defined by set of points (u_i, v_i), with $i = 1, \ldots n$, the coordinates of the centroid (u_r, v_r) are the arithmetic mean all points forming the region, i.e., $u_r = (\sum_{i=1}^{n} u_i)/n$ and $v_r = (\sum_{i=1}^{n} v_i)/n$. The centroids in V^* are projected back to V' which once again results in lines, thus (u_r, v_r) in V^* becomes $y = u_r x - v_r$ in V'. These lines define the final set of regions R' in the primal space, which is used in the rest of the thesis to identify a set of predicates P.

To illustrate the process of dual space projections consider Figure 4.5, where the left image contains three sample labeled points. The middle image shows the projection of these onto the dual plane where the three points become three overlapping lines. Their intersection points form a region, which is shaded and a point inside it is the computed centroid. This centroid is then projected back onto the primal plane as depicted on the right image, where it becomes a line. In this example this centroid line separates the initial three data points.

Algorithm 3 provides pseudocode for the region-finding process that we imple-

mented. The algorithm takes the set of hyperplanes $\mathcal{H}$ as input and returns a set of regions $\mathcal{S}'$, where each region is a set of intersection points. An intersection κ is represented by coordinates and the set of hyperplanes that create this intersection. In our implementation, h will be a line represented by the slope intercept formula, and contains references to all other lines that intersect ordered by the x value of the intersection's coordinates (in an x-y coordinate system).

Algorithm 3 Pseudocode for Region Finding algorithm.

```
1:  function FIND-REGIONS(H)
2:      S' ← ∅
3:      for all hyperplane h ∈ H do
4:          for all intersection κ ∈ h do
5:              h' ← hyperplane intersecting h at κ
6:              κ_n ← next intersection along h'
7:              S ← {κ}
8:              repeat
9:                  S ← S ∪ {κ_n}
10:                 h' ← hyperplane intersecting h' at κ_n
11:                 κ_n ← next intersection along h'
12:             until κ = κ_n
13:             S' ← S' ∪ {S}
14:         end for
15:     end for
16:     Return S'
17: end function
```

Once all intersections are calculated, lines 4 through 14, the algorithm iterates over these intersections with each intersection serving as a starting point for finding a potential region. Lines 8 through 12 take the starting point κ and increments κ_n to the next closest intersection along the current line h'. On line 10 h' is updated with the closest intersecting hyperplane.

The algorithm continues updating κ_n and h', until κ_n is back to the initial intersection for the region. Intuitively, the algorithm 'walks' around the perimeter of the region's edges visiting and storing each vertex as the algorithm progresses. Upon

terminating the loop on line 12, the algorithm has traversed the entire region, which it stores as the set of these intersection points, and adds these points to the return set on line 13. Once all possible starting intersections are traversed, the algorithm has obtained the full set of regions and returns it.

Once our final regions in the primal space are found we can examine the labels on the data and begin labeling regions as positive and negative. The sorting method uses a greedy set covering algorithm to find the region that contains the greatest number of positively labeled data points and remove that region and those points from consideration. The set covering algorithm then continues until all of the positive data points have been removed. The final set of regions encapsulating those positive points form a boolean combination of halfspaces H that is output into the next step of the process.

4.4 Domain Projection

Once the algorithm has generated the final set of halfspaces H, it can then generate partitions that form our predicate domain. Given a hyperplane h in H is defined by a slope m and a Y intercept b, we need to develop a method that finds an associated integer value to act as a partition in the domain. The algorithm determines what value is appropriate to create the partition as well as which variable the partition is defined for. As h represents a potential position for separating values of a variable we use horizontal hyperplanes as partitions for values along the vertical axis, and vertical hyperplanes for values along the horizontal axis.

There are a number of considerations that must be taken into account to achieve this projection. As a hyperplane is defined by slope, the corresponding line can

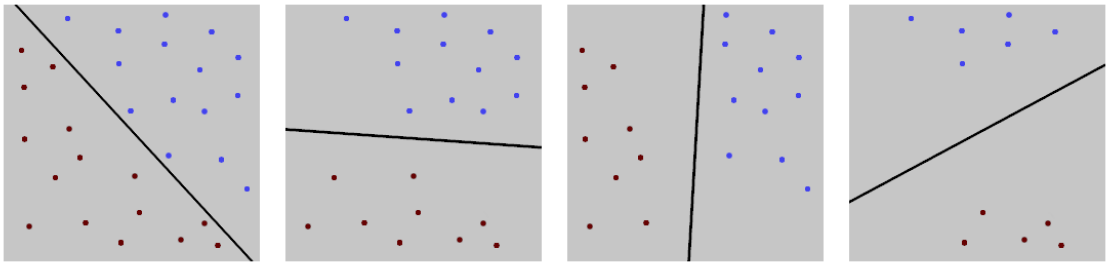

Figure 4.6: Diagonal, horizontal, vertical, and unclear hyperplane lines.

never be vertical as $m < \infty$. Furthermore a hyperplane with $m = 0$, corresponding to a perfectly horizontal line, is unlikely to occur. This is due to the nature of the dual-space projection, where m in the primal-space corresponds to the u value of a region's centroid in the dual-space. Although it is possible for this centroid to be at exactly $u = 0$ it is more likely to be slightly offset, resulting in a non-horizontal hyperplane.

Both of these issues are resolved by introducing a threshold $K°$, the number of degrees that the corresponding line of h can diverge from each axis while still being considered as axially aligned. In total there are four separate cases that the algorithm considers when projecting the PAC learning algorithm's hyperplanes onto a predicate domain; these cases are shown in Figure 4.6 as examples. An abstract representation of determining the cases is shown in Figure 4.7.

4.4.1 Vertical Separations

For the third case shown in Figure 4.6 when the slope of h is within $K°$ of vertical it is suggesting a potential separation of values at the X intercept of h. This value is calculated by solving for X when $Y = 0$ and is then added to a list of potential partitions. The Y intercept is used as the data being separated in this work is always

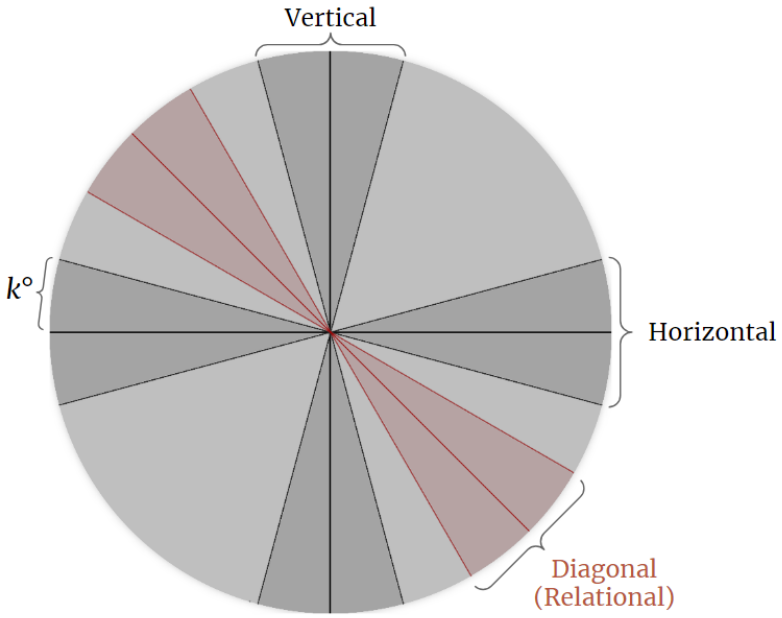

Figure 4.7: Unit circle visualizing ranges of line angles resulting in differing hyperplane projections.

centered around $(0, 0)$. However, for data that is known to be offset the intercept should be offset as well to account for the variation from the intercept caused by $K°$.

4.4.2 Horizontal Separations

The second case in Figure 4.6 follows a similar process when the slope of h is within $K°$ of horizontal. This implies a potential separation at the Y intercept h, where the Y intercept is used in place of the X intercept as the point of separation. This value is already calculated as part of h and can likewise be added to the list of potential partitions.

4.4.3 Diagonal Separations

Handling nearly diagonal lines as shown in the first case in Figure 4.6, when the slope of h is within K° of $(horizontal \pm 45^\circ)$, requires additional consideration. As a partition derived from h would not directly relate to either variable individually it is unfitting to use h to provide a partition. Furthermore, data that can be cleanly separated by a purely relational line, one defined by a relationship of $X = Y$ or $X = -Y$ suggest program behavior that is itself relational. For programs such as these the predicate domain is insufficient for performing analysis. As such the algorithm provides the user with a warning when such a hyperplane is encountered during the partition generating process, suggesting that a more expressive abstract domain may be necessary.

4.4.4 Unclear Orientations

For the remaining cases when h does not fit into previous categories, the hyperplane is discarded as any partition generated from it would be unlikely to provide any value to the analysis. Such a line is shown as the fourth case in Figure 4.6 where the separating hyperplane is not suitable for projection.

4.5 Synthesizing Complete Domains

The partitions generated for each variable form the basis of the complete domain. The process of merging these partitions is influenced by the order in which the hyperplanes are selected by the PAC learning algorithm and the potentially relational nature of the program's behavior. Furthermore as the goal of this thesis is to specifically

create disjoint domains, the individual partition values must be made into disjoint predicates.

4.5.1 Partition Prioritization

As the partitions generated earlier in the PAC learning algorithm process separate more points they are assigned a higher priority. This ensures that these early partitions, which tend to carry more information, have a larger influence on the shape of the final domain.

4.5.2 Relational Program Behavior

The possibility of the program behavior being relational is taken into account. If the most weighted partition suggests a relational behavior, i.e., the partition is diagonal and separates both variables equally, then it is highly unlikely that lower priority partitions are beneficial to the analysis.

In such a case, the algorithm halts the partition process and alerts the user, as a single variable predicate domain would be insufficient for capturing variable relations and a more comprehensive abstract domain is required.

4.5.3 Domain Construction

Once all partitions have been prioritised and any relational behavior has been addressed, the complete predicate domain is constructed. The partitions are sorted in decreasing order of the number of pairs separated, and each is added to the domain. A singular integer value that was generated during the projection process is split into three separate predicates for values less than, greater than, and exactly equal to the generated value.

The addition of these partitions creates a new segment of the domain. For instance, adding a partition at value j splits the range of that variable into three segments: $(-\infty, j)$, $[j]$, and $(j, +\infty)$. Each added partition further subdivides the range of its variable.

In some cases, not enough partitions are generated for a worthwhile domain. For these cases, partitions generated for one variable can be utilized for both variables. This strategy ensures that at least some partitioning is accomplished, though it might not be optimal.

CHAPTER 5

EVALUATION

To determine the effectiveness of our approach for static analysis domain generation we perform the following evaluation: we gather sample data from test executions of instrumented programs, provide that sample data as training material for our PAC learning algoithm to learn potential partitions in our domain, and test the newly generated domain against randomly selected common domains by evaluating their comparative precision across all execution paths. These results are then examined to determine if the generated domain is more precise than the randomly selected domains on average, and if so how much sample data is necessary for training the PAC learning algorithm to receive such a domain.

The following sections present the results for each evaluiation step as well as notes on attempted domain generation. The results are presented to assist in answering the folowing research questions:

RQ1 How can data be efficiently gathered from programs to provide samples for the PAC learner?

RQ2 How does sample size affect the output of the PAC learning algorithm?

RQ3 Can the PAC learning algorithm produce predicate abstract domains on a set of benchmark programs?

5.1 Initial Filtering of Test Programs

Evaluation is performed over a set of 33 Java program files contatining 209 methods [13]. This benchmark set is obtained from previous work related to this thesis, including SA research using disjoint predicate domains. Despite being used in similar work there are many programs and methods that do not have learnable features to train the dual-space projection PAC learning algorithm.

The PAC learning algorithm is applicable to problems comparing two non-constant variables with $\leq$ or $\geq$ relations. When one of the variables is constant it results in a one-dimensional space within the dual-space projection, and furthermore can be analysed using simpler methodologies such as value extraction where the constant value is used as the partition. Equality or inequality result in labeled data where the majority of the results share the same labels. Such unbalanced data cannot be used to learn accurate separations.

An additional difficulty in the provided Java programs arose from transformations applied to them to allow the programs to compile and run without additional external code. A common transformation assigns random values to unresolvable expressions. For SA evaluation this poses no problems as the analysis is performed without executing the code and therefore the random values are suitable placeholders. However, this work leverages the actual values of variables are necessary for the training data, thus such a transformation creates many situations where the sampled training data would be direct comparisons of two random integers. Although this data would be technically usable within the PAC learning algorithm, it would not provide any insight into the behavior of the program under analysis as it would be equivalent to training the PAC learning algorithm entirely on random values.

To eliminate such situations, the initial programs are manually filtered at the source code to only those methods that contain if statements comparing two or more non-constant variables. Furthermore those non-constant variables must not have been set to random values without any calculations performed altering them prior to the comparison. If the variables are initially set to random values but then are changed by the program behavior the methods are kept.

Table 5.1: Class Filtering Results

Category	**Classes**	**Methods**
Total	33	209
Only trivial comparisons	16	158
No valid comparisons	8	16
Unable to create test suite	1	9
Valid benchmark programs	8	11

Table 5.1 shows the number of programs that are filtered out from the benchmark set along with the reason for filtering. Programs listed as 'Only trivial comparisons' are those that had comparisons of two or more non-constant variables, but the values of those variables would be entirely randomly generated independant of the program's execution. These programs are able to be instrumented and can have the PAC learning algorithm attempt to generate domains for them. However, such results are not able to accurately capture the relational behavior of the variables being analysed. 'No valid comparisons' are the programs that have no if statements comparing two or more non-constant variables. These programs provide no data for the PAC learning algorithm to train on. One program is removed because executing it requires external map data for the program to perform calculations on, without which it behaves as the 'Only trivial comparisons' programs. Since it is not feasible to obtain this map data, it is removed from the set.

With filtering complete the remaining eight Java programs are set to be instrumented and used within the domain generating framework and have jUnit tests generated for them so that sample data can be gathered.

5.1.1 Instrumentation

The instrumentation process is straightforward with the procedure in Chapter 4 adding the value reporting statements to each of the valid programs. The majority of the programs have a single conditional statement that is instrumented resulting in a single set of sample data each. The `GeoEngine` and `Sort` classes are exceptions, with `GeoEngine` containing three separate methods with valid comparisons that are instrumented, of which `GeoEngine.moveCheck()` contains two separate valid conditional statements that are instrumented. `Sort` contains both `Sort.selectSort()` and `Sort.bubbleSort()` which both contain valid conditional statements that are instrumented.

5.2 Test Generation

As the test generation step of the framework requires significant manual oversight, the results are highly specific to the individual target programs. As such different programs have different steps performed on them. The following results demonstrate this aspect with various classes reporting no data for some steps, as those classes are handled in a different manner.

Table 5.2 provides the results of Evosuite generating jUnit tests for the valid benchmark programs. The table shows the number of tests that Evosuite generated as well as the reported statement coverage of those tests. The column labeled 'Useful'

Table 5.2: Evosuite Test Generation Results

Class Name	Tests Generated	Coverage (%)	Useful
AmyFastSimplex	n/a	n/a	n/a
BinarySearch	8	39	6
Example1M	8	34	8
GeoEngine	n/a	n/a	n/a
InfBlocks	n/a	n/a	n/a
MultidementionalArray	3	44	0
QRCodeDataBlockReader	1	0	0
Sort	n/a	n/a	n/a

denotes how many of the generated jUnit tests can be used to run the methods containing the instrumented comparisons. As Evosuite generates tests for an entire class file, not all of the instrumented comparisons execute during a jUnit test run. Thus it is possible for the jUnit tests to diverge from the relevant portions of the program.

For the programs `MultidementionalArray`, `QRCodeDataBlockReader`, and the four for which Evosuite fails to produce tests, the tests are instead created either manually or with assistance from ChatGPT. For `BinarySearch` and `Example1M` one of the useful jUnit tests is selected and augmented using jUnitQuickCheck to be able to run repeatedly with parameterized inputs. Once all of the programs have jUnit tests that have been augmented by jUnitQuickCheck the tests are able to be run to gather the sample data for training the PAC learning algorithm.

Table 5.3 shows the results of 5 sets of 100 parameterized executions of the method being tested, for a total of 500 method executions across varied input values. The table shows that `binarySearchUsingDivision()` has far more gathered samples than the 500 parameterized executions, since each execution results in the instrumented code being called multiple times due to the method recursively calling itself. The

Table 5.3: Method Execution Results

Method Name	**Samples**
AmyFastSimplex.noise()	0
binarySearchUsingDivision()	2611
example1M()	500
GeoEngine.moveCheck() 1	789
GeoEngine.moveCheck() 2	500
GeoEngine.nLOS	0
GeoEngine.nGetNSWE	487
InfBlocks.proc	0
MultidementionalArray.maxProduct()	0
QRCodeDataBlockReader.getNextBits()	0
Sort.selectSort()	0
Sort.bubbleSort	0

majority of the tested methods show no samples being gathered. This is caused by the instrumented code not being executed during the test case run. Although these methods do contain conditional statements that match the requirements of the filtering process, the actual outcome of the instrumentation does not match expectations due to how the instrumentation process handles three-address-code. More information about this difference is discussed at the end of this chapter in Section 5.4. In the cases where the instrumented code did provide usable training data it is then passed on to the PAC Learning algorithm to generate a predicate domain.

These results provide insight into "*RQ1: How can data be efficiently gathered from programs to provide samples for the PAC learner?*". With a strict filtering set and multiple tools utilized to gather data, the process is fallible and inconsistent. Manual user intervention is required for nearly all steps, most notably the filtering process and the augmentation of jUnit tests for parameterized executions. With less than half of the filtered benchmark programs providing usable tests it does not seem flexible

enough to be incorporated into a program-agnostic analysis pipeline. Furthermore with tests written for the full set of filtered benchmark programs, instrumentation gathered training data for less than half of all valid conditionals.

5.3 PAC Learning Algorithm

5.3.1 Synthetic Data

Initial experiments with the PAC learning algorithm are performed on a trivial Java method `MinOfThree` that takes three input values and compares them, returning the minimal value. Experiments using `MinOfThree` are directly invoked with random values of varying ranges. Whereas tests on the benchmark set of programs were all performed using a standardized range of input values, the tests on `MinOfThree` used synthetic data with specific variables set to values within specified ranges. Figures 5.1, 5.2, and 5.3 show sample data gathered across handpicked ranges for the random values of the input variables. Although `MinOfThree` takes three input variables, each comparison only deals with two at a time, with the left hand side (LHS) of the comparison on the horizontal axis and the right hand side (RHS) on the vertical axis.

Figure 5.1 shows the results of running the PAC learning algorithm on sample data from two separate tests of the `MinOfThree` program, once with a sample size of 50 in the left column and once with a sample size of 200 in the right column. As each execution of `MinOfThree` has three comparisons, it results in three different sets of output. Each row displays data sampled from one of the three comparisons, with the first row having a sampled data point for each execution. As the second comparison only happens when the first evaluates to 'True' and the third when 'False', each of

those rows has roughly half the number of samples. Samples are colored based on their label from the instrumentation which relates to whether the conditional statement resulted in a 'fall-through' or a 'branch-out', and lines are the hyperplanes generated by the PAC learning algorithm.

Figure 5.2 applies a different setup from Figure 5.1 with input values clamped between -1000 and -500 when negative or between 500 and 1000 when positive. This provides clear values for ideal partition points within the values of the sample data, although as shown by the lines representing selected hyperplanes, no such separations are found.

Figure 5.3 repeats the experiment with clamped input values on an alternative implementation of the `MinOfThree` program. This implementation takes the same inputs and provides the same output, however uses a different internal configuration of conditional statements, resulting in a different distribution of sampled data points. This series of tests provided the results closest to the desired outcome of horizontal and vertical lines at key input values.

Examination of these figures provides initial insight into "*RQ2: How does sample size affect the PAC learning algorithm?*". With smaller sample sizes there is a greater variance in the sampled data, balanced by a lower number of pairs that must be separated. Larger sample sizes provide data that more clearly demonstrates program behavior, with a higher likelihood of pairs that are close together and more difficult to separate. These hard to separate points occur more frequently when both LHS and RHS variable are nearly equal. Data with a large number of hard-to-separate points is more likely to require a greater number of hyperplanes to form a complete separation.

The figures also provide examples of hyperplanes along the $X = Y$ line. Due

to the limitations of three-address-code any hyperplane along this line will perfectly separate all points. Although no definitive pattern emerges, it appears likely that with a greater number of sample data points the dual-space projection will be more likely to produce a hyperplane along $X = Y$ resulting in this trivial separation.

5.3.2 Input Variable Symmetry

Since the PAC learning algorithm is not symmetric in terms of the LHS and RHS variables, this thesis examined the results from switching the effective side of the variable when providing the sample data to the learning algorithm. Whereas by default the algorithm takes the LHS and RHS variables and assign them to the horizontal and vertical axii respectively, it is equally valid to instead assign the LHS variable to the vertical axis and vice versa. Despite using the same sample data values and performing the same technique, this provides different results as the dual-space projection behaves different for each axis.

Figure 5.4 shows four different sets of results from applying the PAC learning algorithm to the `ExampleM1` program. The left column displays the default algorithm results, the central column displays the flipped algorithm results, and the right column provides an overlay of both, showing that the blue points on the left column correspond to the yellow points in the center column. This process can be seen to result in a different set of hyperplanes depending on whether it is the default or flipped behavior. Most notably the third row demonstrates that even with identical input values the ending result can be highly variable.

The results of flipping the input variable sides are gathered to attempt to address the inconsistancy of generated hyperplanes shown in prior figures. However, as Figure 5.4 shows in the third column, the overlayed hyperplanes show no apparent

convergence toward a unified result. Although the existence of hyperplanes along the $X = Y$ is still noticeable and reinforced by flipping the input variables, no clear vertical or horizontal separations show themselves in both the original and flipped result.

The algorithm is run across multiple sample programs with the same results demonstrating no clear benefit in attempting to unify results from original and flipped input variables. However as later data shows the PAC learning algorithm does perform differently depending on the input program being evaluated when evaluating flipped variables. This is further examined in the following section.

5.3.3 Benchmark Results

Figure 5.5 provides the results of four separate runs of the PAC learning algorithm across different sample data sets from the `binarySearchUsingDivision` method. For each run of the algorithm a clear diagonal line was found separating the points with differing labels, resulting in an evaluation of the program being purely relational. Although vertical lines were found they have lower priority in the greedy set covering and as such were not considered as significant. Furthermore these vertical lines consistently appeared above the values of the variable corresponding to the X axis. With external knowledge of the behavior of the binary search algorithm, such a value would not provide a meaningful partition for program analysis. Instead a vertical line corresponding to a partition at $X = 0$ would have been desirable.

Tables 5.4, 5.5, 5.6 contain excerpts of data from using the PAC learning algorithm with data sampled from the `BinarySearch` program. The first four columns of each table provide values for the number of hyperplanes evaluated as a line orientation as described in Section 4.4. The **Primary** column lists which orientation is selected by

Table 5.4: BinarySearch PAC Algorithm Results (Sample Size 50, K=15)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
0	1	1	2	Unclear	4
0	0	1	3	Diagonal	4
0	0	1	2	Diagonal	3
0	0	2	2	Diagonal	4
0	0	2	2	Unclear	4
0	0	3	2	Diagonal	5
0	0	2	0	Diagonal	2
1	1	0	3	Unclear	5
0	1	2	1	Diagonal	4
0	0	1	3	Diagonal	4

Table 5.5: BinarySearch PAC Algorithm Results (Sample Size 50, K=15, Flipped Input Variables)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
0	1	1	0	Diagonal	2
0	0	2	0	Diagonal	2
0	0	2	1	Diagonal	3
0	0	1	0	Diagonal	1
8	2	0	1	Horizontal	11
0	0	1	1	Diagonal	2
1	0	2	1	Diagonal	4
0	0	1	0	Diagonal	1
0	0	2	0	Diagonal	2
0	0	1	1	Diagonal	2

the greedy set covering algorithm first, and corresponds to the orientation of the line that separates the greatest number of points in the dual-space projection.

When comparing Tables 5.4 and 5.5 to determine the efficacy of evaluating original variable assignments to flipped input variables, there is a clear difference between the two sets of data. The original set of hyperplanes in Table 5.4 has a far greater number of 'Unclear' lines that are neither axially aligned nor diagonal. The corresponding flipped data has fewer hyperplanes generated on average with a significantly reduced

Table 5.6: BinarySearch PAC Algorithm Results (Sample Size 200, K=15)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
1	2	0	3	Unclear	6
0	0	1	2	Diagonal	3
0	0	3	0	Diagonal	3
1	0	1	0	Diagonal	2
0	0	1	0	Diagonal	1
1	0	1	0	Diagonal	2
0	0	2	0	Diagonal	2
0	0	2	0	Diagonal	2
0	0	1	1	Diagonal	2
0	2	1	2	Unclear	5

Table 5.7: Example1M PAC Algorithm Results (Sample Size 50, K=15)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
1	0	1	0	Diagonal	2
0	1	1	0	Diagonal	2
3	2	1	1	Diagonal	7
2	1	1	1	Diagonal	5
0	2	2	1	Diagonal	5
1	0	1	0	Diagonal	2
1	1	2	1	Diagonal	4
0	0	3	0	Diagonal	3
0	0	2	0	Diagonal	2
0	0	2	0	Diagonal	2

number of 'Unclear' lines.

This difference suggests that while considering *RQ1* it is advisable to provide the PAC learning algorithm with data in both the original configuration as well as the flipped configuration as behavior may only be learnable in one of the two. However, using both configurations together to determine any unifying characteristics does not appear possible as the behavior of the two results seems unconnected.

Tables 5.7, 5.8, 5.9 contain the same data gathered from the `Example1M` program. The behavior of this program is very likely to be purely relational with nearly every

Table 5.8: Example1M PAC Algorithm Results (Sample Size 50, K=15, Flipped Input Variables)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
0	0	1	1	Diagonal	2
0	0	2	0	Diagonal	2
0	0	2	1	Diagonal	3
0	0	1	0	Diagonal	1
0	0	2	0	Diagonal	2
0	0	1	0	Diagonal	1
1	0	3	1	Diagonal	5
0	0	1	1	Diagonal	2
0	0	3	0	Diagonal	3
3	0	0	4	Unclear	7

Table 5.9: Example1M PAC Algorithm Results (Sample Size 200, K=15)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
0	0	3	0	Diagonal	3
0	0	1	0	Diagonal	1
1	0	3	1	Diagonal	5
1	0	2	2	Diagonal	5
0	0	1	0	Diagonal	1
0	0	3	0	Diagonal	3
0	0	3	0	Diagonal	3
0	0	2	0	Diagonal	2
0	3	2	0	Diagonal	5
13	4	0	2	Vertical	19

run of the PAC learning algorithm resulting in a hyperplane corresponding to a diagonal line. The difference between original and flipped variables seen in `BinarySearch` is not present here as the sample data was uniformly distributed across both LHS and RHS variables.

Comparison of Tables 5.7 and 5.9 provides additional confirmation of earlier conclusions regarding *RQ2*. Both the number of hyperplanes selected for pair separation and the orientation of associated lines appear equally dsitributed regardless

Table 5.10: GeoEngine PAC Algorithm Results (Sample Size 50, K=15)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
0	0	0	1	Unclear	1
1	0	0	0	Vertical	1
0	0	0	1	Unclear	1
2	0	0	0	Vertical	2
1	1	0	0	Horizontal	2
0	2	0	1	Unclear	3
1	0	0	0	Vertical	1
0	0	0	1	Unclear	1
0	1	0	1	Unclear	2
0	0	0	2	Unclear	2

Table 5.11: GeoEngine PAC Algorithm Results (Sample Size 50, K=15, Flipped Input Variables)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
0	0	0	1	Unclear	1
0	0	0	1	Unclear	1
1	0	0	1	Unclear	2
0	1	0	1	Unclear	2
0	0	1	0	Diagonal	1
0	0	0	1	Unclear	1
0	0	0	1	Unclear	1
0	0	0	1	Unclear	1
0	0	0	1	Unclear	1
1	0	0	0	Vertical	1

of sample size. Although not shown here, additional tests are performed throughout the evaluation process using a wide range of sample sizes, displaying similar results. Sample sizes over 500 seemingly provided no benefit, but did increase execution time of the learning algorithm significantly due to the program's complexity. As such sample size does not appear to have any measurable effect on the output of the PAC learning algorithm or its ability to produce domains.

Finally, tables 5.10, 5.11, 5.12 contain the data gathered from the `GeoEngine`

Table 5.12: GeoEngine PAC Algorithm Results (Sample Size 200, K=15)

Vertical	**Horizontal**	**Diagonal**	**Unclear**	**Primary**	**Total**
0	0	0	1	Unclear	1
0	0	1	0	Diagonal	1
1	0	0	1	Unclear	2
0	0	0	1	Unclear	1
0	1	0	1	Unclear	2
1	0	0	0	Vertical	1
1	0	0	0	Vertical	1
0	0	0	1	Unclear	1
1	0	0	1	Vertical	2
1	0	0	1	Unclear	2

program. A notable difference in this program is a much lower number of hyperplanes are needed to separate sample points. The tables do show a number of runs that resulted in singular vertical lines separating the data points which appears promising, however the reasoning behind this is more indicitve of the PAC learning algorithm's behavior as opposed to anything learned from the program being analysed.

Despite being run on data gathered from the same range of input values as the other benchmark programs (-1000 to 1000), the sampled data points themselves cover a much larger range (roughly 0 to 1,500,000,000). When given such a wide range of potential values the dual-space projection produces hyperplanes with slopes that vary more significantly. Correspondingly, the intercept values that the domain projection uses for selecting partitions have a much higher variance as well. With slopes varying across a wider range, a greater number of lines are marked as vertical, as any value above or below a certain threshold is considered vertical. This differs from the behavior of hyperplanes corresponding to horizontal lines which are always constrained within the same range of slopes centered around a value of zero.

5.4 Challenges

Once data has been processed using the dual-space projection it is necessary to interpret the resulting lines and regions in order to derive a suitable method for projection onto the predicate domain. This presents numerous challenges that must be uniquely addressed, as prior work utilized techniques with additional complexity that provide invariant generation methods that are unavailable to us. In the work by Sharma et al. [11] the resulting regions from the PAC learner are directly used as invariants as the domain in use for their static analyses allows complex predicates that can accurately represent the generated region. As the goal of this work is to generate partitions in the predicate domain, it is also necessary to find a meaningful way to project the generated region onto this domain.

An additional challenge arises from the nature of the tools being used to gather data from the programs under analysis. In Sharma et al.'s work each branch being evaluated can have a wide range of possible conditional statements. Due to the use of SOOT [16] in this work and the three-address code nature of Jimple, SOOT's internal representation of code, the conditional statements that are available to evaluate are far simpler. In the original source code a conditional may include any number of variables and constant values being compared to each other, for example in a statement `If` ($X \geq 50$ `AND` $X \leq 100$) where the value is bounded between 50 and 100 for all true executions. Once represented in Jimple, however, this would result in two separate conditional statements being evaluated, one that checks for X to be equal or greater than 50, and another that checks if it is equal to or under 100. By evaluating these two conditions separately the information that they are related is not immediately available to the PAC learning algorithm.

When examining comparisons of two variable values, the case with the most complexity available to us, the data is also limited by the number of available operations. With greater-than and less-than comparisons both resulting datasets have regions that are perfectly separated by the linear representation of the two values being equal. Equality and inequality comparisons similarly have separations around that line, separating the values on the line from those not on the line.

Having resulting data sets that are all separable along the line of equality results in regions that are largely unbounded as well as a large number of relational separating lines being generated. These relational lines often efficiently separate the two data sets but are difficult to turn into useful predicate partitions, as each partition can only provide an invariant for a single variable and not a relation between two variables. Sharma et al. [11] address this in their work by specifically selecting lines with slopes that were able to be used as lines along a single variable and then translating those lines to be in an appropriate position. For example finding a horizontal line with a $slope = 0$ at $Y = 3$ and translating that line to $Y = 10$ where all data points are cleanly separated at this point. However this technique could be similarly applied by originally generating lines at those positions prior to doing the dual-space projection, and as such the technique is not beneficial to this work.

5.4.1 Domain Creation

Regarding *RQ3: "Can the PAC learning algorithm produce predicate abstract domains on a set of benchmark programs"*, these challenges suggest that the framework using the PAC learning algorithm is not suitable for this purpose within the limitations of this work. The limited set of programs that both remained in the benchmark set after filtering and produced training data after instrumentation is limited to three methods.

Of these three methods that are viable for using the PAC learning algorithm, each returns consistent results that state a relational analyses is necessary for improving precision of SA. In cases where the PAC learning algorithm generated hyperplanes that could be used for partitioning, the projected values were neither meaningful nor beneficial.

The program `Example1M` was run repeatedly through the PAC learning algorithm with large sample sizes until a number of hyperplanes was generated that could be used to populate a single domain. The result of this was the domain $(\infty, -110](-110, 0)01(1, 35)[35, \infty)$ with two generated partitions, at -110 and 35, along with partitions at 0 and 1 which are always assumed to be beneficial.

Analysis was then performed on the `Example1M` program using both this generated domain and several other pre-existing domains that were randomly selected. For every comparison between the generated domain and the randomly selected domain, the two performed the analysis with equal accuracy. As such the problem posed by *RQ3* does not seem to be answered by this thesis, and if attempted with the results that are provided accuracy is not improved.

5.5 Threats to Validity

5.5.1 Internal

Due to the low success rate of using Evosuite for test generation it is likely that the tool is not being operated in ideal circumstances and may have resulted in inoptimal jUnit tests for data gathering. This is mitigated by the manual review process of the jUnit tests along with the manual augmentation that occurs afterwards. Likewise the parameterized input values generated by jUnitQuickCheck may not be indicative of

real user inputs and could negatively affect the validity of the learning data sampled from them.

5.5.2 External

The programs used in the evaluation benchmark are likely not representative of the problems that would benefit from this form of analysis, evident due to the large number that were manually filtered out. The filtering process mitigates this issue although additional programs that pass the filtering process would undoubtedly improve the reliability of the results. Finding such programs proves to be a significant challenge and was not able to be done during this work.

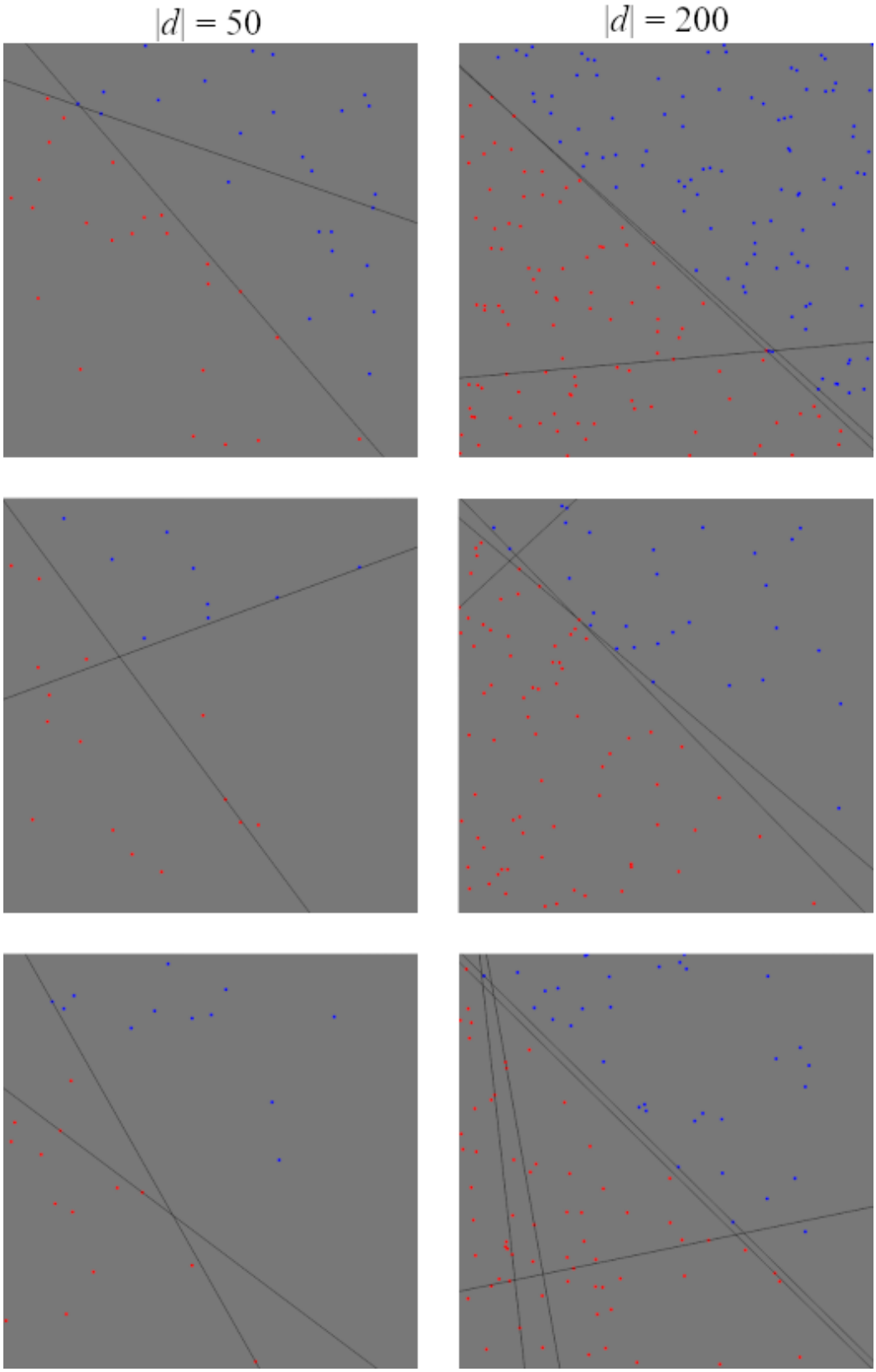

Figure 5.1: Results of PAC Learning the MinOfThree program with random input values between -1000 and 1000

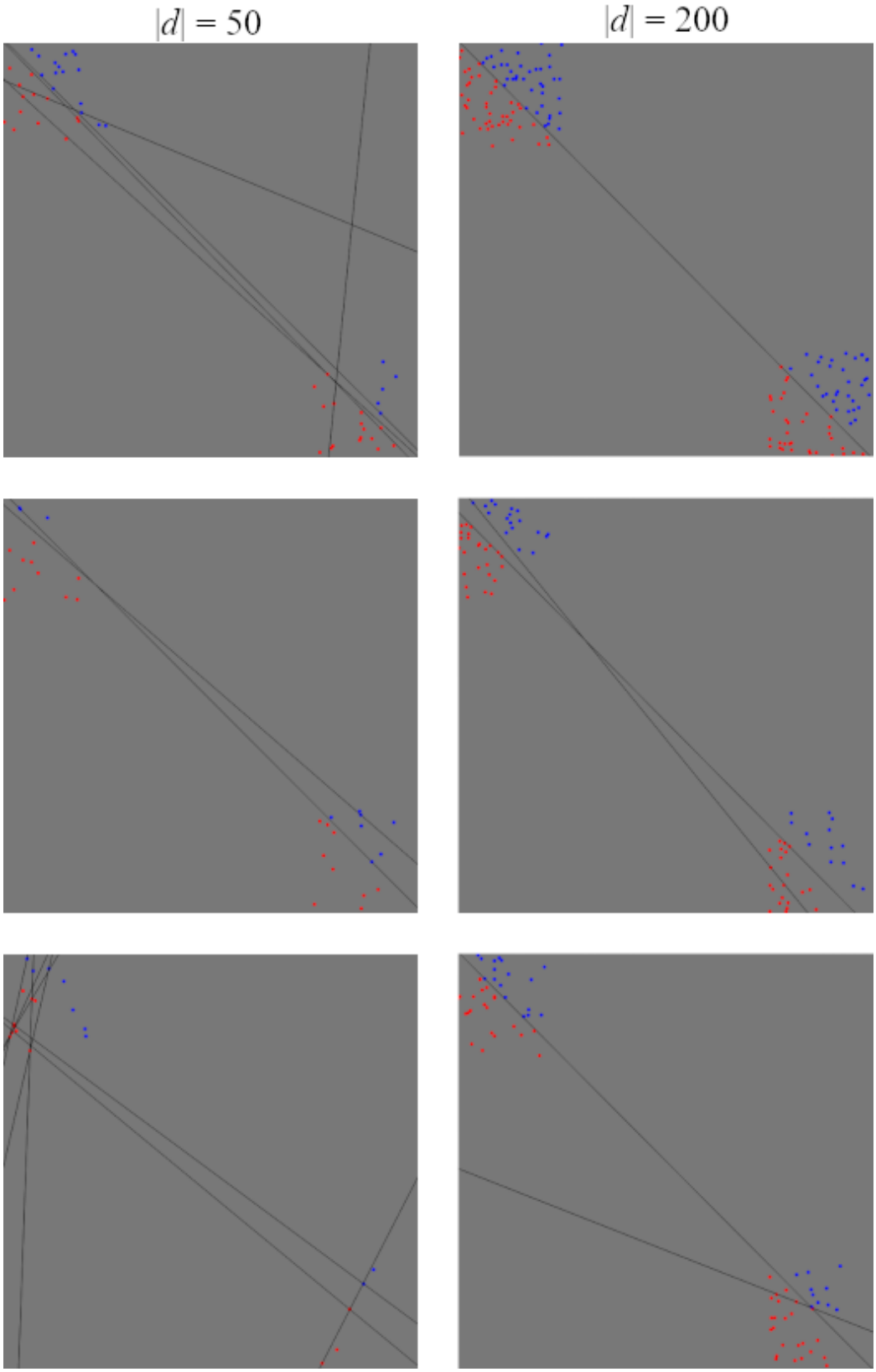

Figure 5.2: Results of PAC Learning the MinOfThree program with random input values between -1000 and -500 or between 500 and 1000

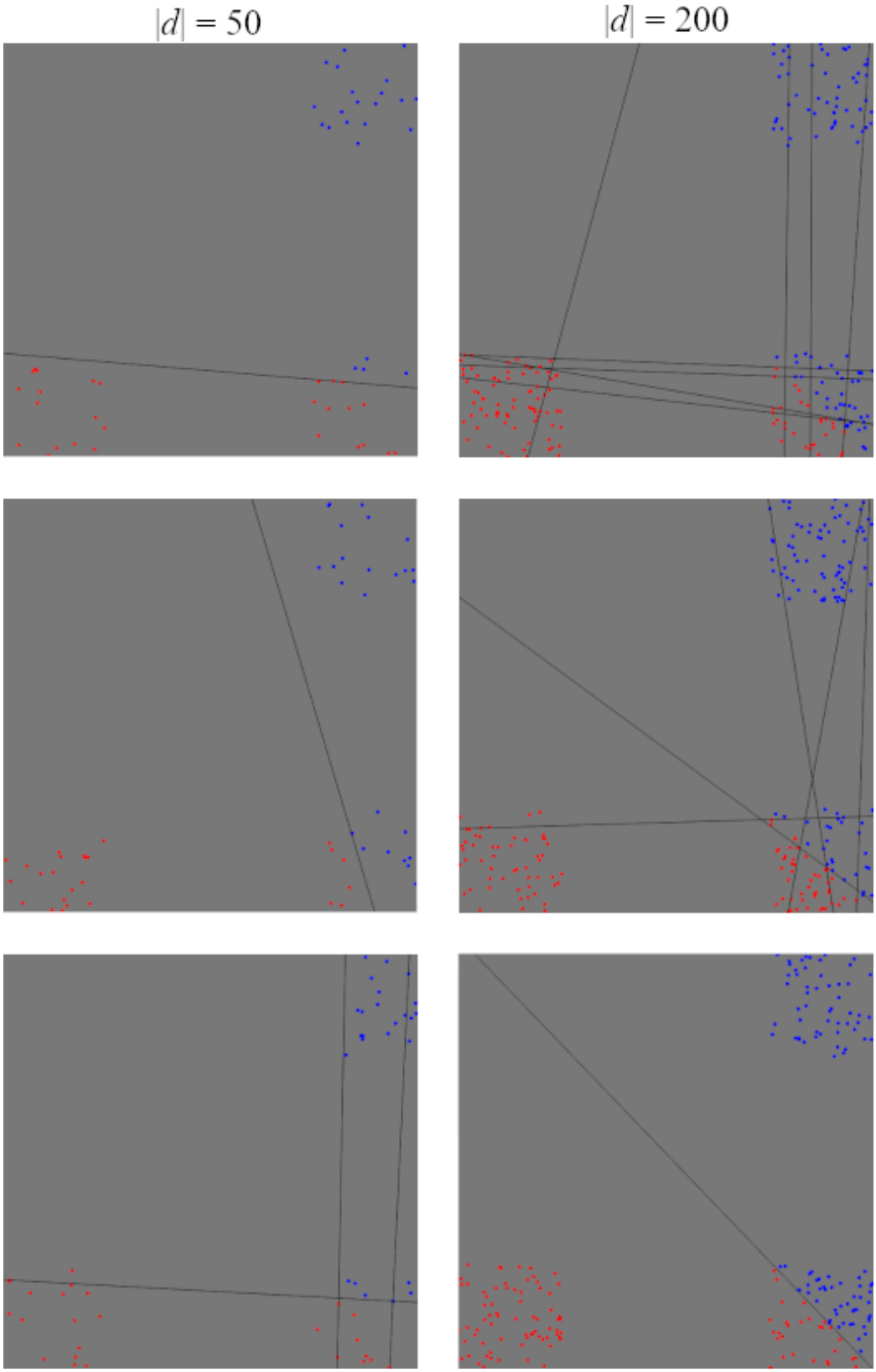

Figure 5.3: Results of PAC Learning an alternate MinOfThree program with random input values between -1000 and -500 or between 500 and 1000

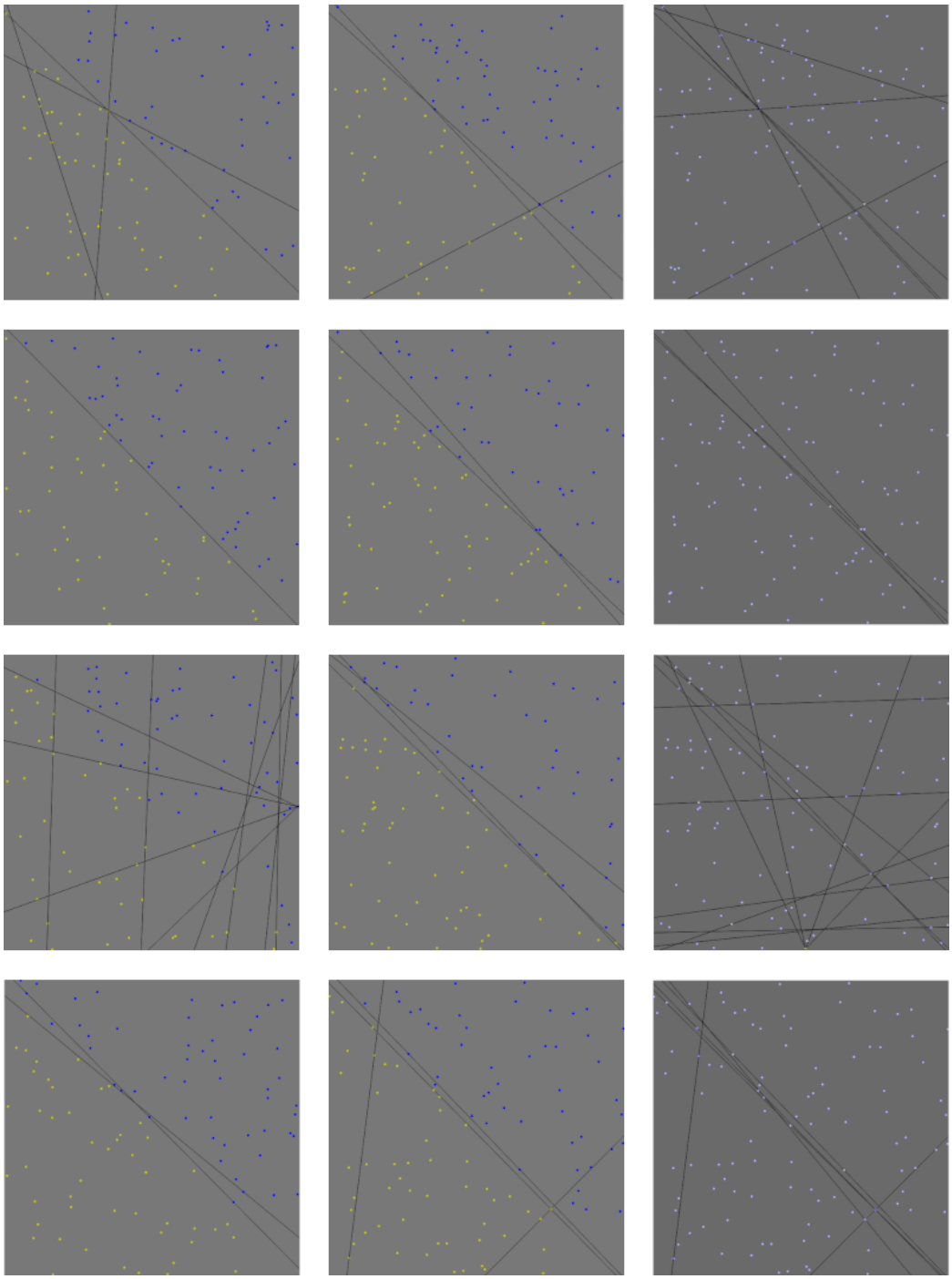

Figure 5.4: Results of PAC Learning the `ExampleM1` program with the original and flipped LHS/RHS assignments.

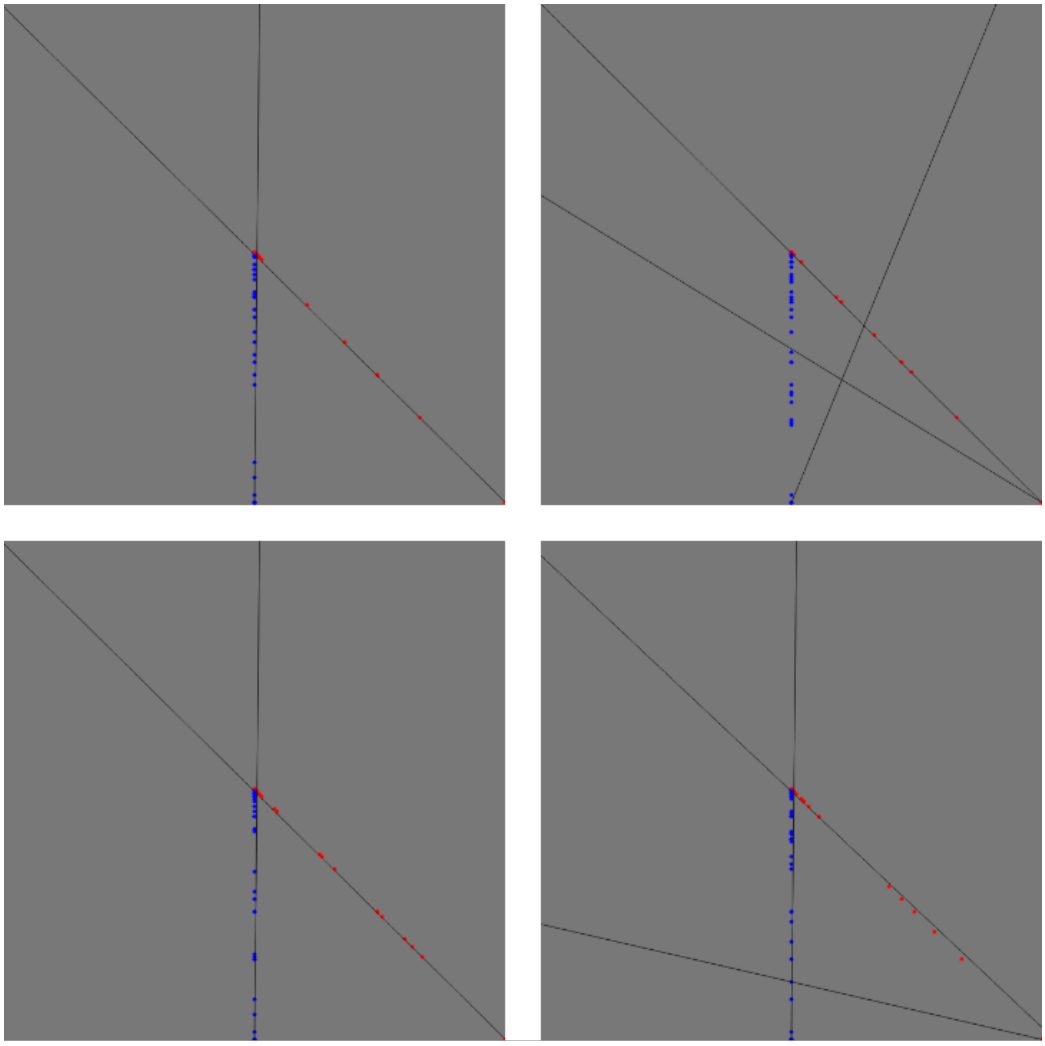

Figure 5.5: Results of PAC Learning the `BinarySearch` program.

CHAPTER 6

CONCLUSION

The research conducted has yielded several important findings that contribute to better understanding of static analysis domain generation and identified new opportunities for future research.

Firstly, the issue of gathering runtime data from arbitrary programs remains an unsolved problem. A separate investigation is required to address this defficiency fully as it currently poses a considerable obstacle to the repeatability of the methodology used in this study. Without a reliable approach to generate tests for an arbitrary program and receive sample data for training the PAC learning algorithm the framework requires significant manual effort, which makes the approach less feasible. However, this work discovered that tools such as jUnitQuickcheck and ChatGPT can aid in required test case generation. This could be a potential avenue for further exploration and improvement in this area, with the potential for fully automating the framework in the future.

Secondly, the PAC learning algorithm demonstrates a high level of flexibility in terms of sample size. Evaluations show that the algorithm can operate effectively with no clear dependency on the sample size, other than the decreased runtime efficiency for large sizes. This suggests that the PAC learning algorithm itself is robust and adaptable, capable of handling a wide range of data inputs at the abstraction level

used in this thesis.

This work also revealed a narrow scope of programs that can benefit from the proposed approach. Due to the constraints imposed by the three-address-code and integer based analysis, the number of programs from the available benchmark with required characteristics is significantly limited. This restricts the applicability of the framework to large sets of programs. Furthermore, this limited program availability precluded any evaluations that determine changes in generated domain precision.

In terms of the projection of hyperplanes onto predicate domains, it was found to be feasible using the proposed categorization, that uses a degree range of hyperplane angles. However, the evaluation of the precision of the generated predicate domains remains to be explored. To address this, the current benchmark set should be augmented with benchmark programs with non-relational behaviors or the PAC learning and projection technique should be changed.

In conclusion, while the research has yielded valuable insights and advanced our understanding of static analysis domain generation, it has also highlighted several areas for future work. The limitations brought about by three-address-code and disjoint predicate domains likely preclude the ability to perform two-dimensional geometric analysis, and therefore learnability of these geometric concepts with a PAC learning algorithm appears infeasible on the given set of programs. Nevertheless this thesis proposes several suggestions to improve the approach in the hopes of finding more promising results.

6.1 Future Work

The findings from this research have opened up several avenues for future explorations and improvements in the field of automated abstract domain generation.

One of the key areas for future work is the improvement of automation for test case generation in order to gather a sufficient data sample more efficiently. Using existing tools makes the current process labor-intensive. This step can benefit from the development of more sophisticated automated tools or techniques. Alternatively, future research could explore different methods for executing arbitrary code to gather sample data, which could potentially overcome some of the limitations encountered in this study.

Another promising area of exploration is the development of a methodology for evaluating the full behavior of variables at a given program location. By capturing richer information relating to variable relations there is potential for a machine learning algorithm to synthesize finer results. This could result in a less restricted selection criterion imposed by the three-address-code, which was found to limit the number of suitable programs. In addition, there is potential to extend the PAC learning algorithm itself to handle more than two variables using higher-dimensionality dual-space projections. This could enhance the versatility and applicability of the PAC learning algorithm.

Finally, future research could also investigate different PAC learning methodologies. For instance, finding rectangular bounds of data samples could provide an alternative approach to domain generation and offer new insights into the capabilities of PAC learning in this context, while also circumventing issues that arose during projecting hyperplanes onto predicate domains.

REFERENCES

[1] Eric B Baum. On learning a union of half spaces. *Journal of Complexity*, 6(1):67–101, 1990.

[2] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred K Warmuth. Learnability and the vapnik-chervonenkis dimension. *Journal of the ACM (JACM)*, 36(4):929–965, 1989.

[3] Nader H Bshouty, Sally A Goldman, H David Mathias, Subhash Suri, and Hisao Tamaki. Noise-tolerant distribution-free learning of general geometric concepts. *Journal of the ACM (JACM)*, 45(5):863–890, 1998.

[4] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[5] Long H. Pham, Ly Ly Tran Thi, and Jun Sun. Assertion generation through active learning. In Zhenhua Duan and Luke Ong, editors, *Formal Methods and Software Engineering*, pages 174–191, Cham, 2017. Springer International Publishing.

[6] Paul Holser. JUnit-Quickcheck github repository.

[7] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer Science & Business Media, 2004.

[8] Carlos Pacheco and Michael D. Ernst. Randoop: feedback-directed random testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM, October 2007.

[9] Long H Pham, Jun Sun, and Quang Loc Le. Compositional verification of heap-manipulating programs through property-guided learning. In *Asian Symposium on Programming Languages and Systems*, pages 405–424. Springer, 2019.

[10] Long H. Pham, Ly Ly Tran Thi, and Jun Sun. Assertion generation through active learning. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 155–157, 2017.

[11] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. Verification as learning geometric concepts. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 388–411, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] Rahul Sharma, Aditya V Nori, and Alex Aiken. Bias-variance tradeoffs in program analysis. *ACM SIGPLAN Notices*, 49(1):127–137, 2014.

[13] Elena Sherman and Matthew B Dwyer. Exploiting domain and program structure to synthesize efficient and precise data flow analyses (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 608–618. IEEE, 2015.

[14] JUnit Team. Junit.

[15] L. G. Valiant. A theory of the learnable. *Commun. ACM*, 27(11):1134–1142, nov 1984.

[16] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.