Selecting Predicates For Conditional Invariant Detection Using Cluster Analysis

by

Nii Lartey Dodoo

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

August 2002

Author . . . .
Department of Electrical Engineering and Computer Science
August 22, 2002

Certified by . . . .
Michael D. Ernst
Assistant Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Selecting Predicates For Conditional Invariant Detection Using Cluster Analysis

by

Nii Lartey Dodoo

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis proposes a new technique for selecting predicates for conditional program invariants — that is, implications of the form $\mathsf{p} \Rightarrow \mathsf{q}$ whose consequent is true only when the predicate is true. Conditional invariants are prevalent in recursive data structures, which behave differently in the base and recursive cases, and in many other situations.

We argue that inferring conditional invariants in programs can be reduced to the task of selecting suitable predicates from which to infer these conditional invariants. It is computationally infeasible to try every possible predicate for conditional properties, and therefore it is necessary to limit possible predicates to only those likely to give good results. This thesis describes the use of cluster analysis to select suitable predicates for conditional invariant detection from program execution traces. The primary thesis is: *if "natural" clusters exist in the values of program execution traces, then invariants over these clusters should serve as good predicates for conditional invariant detection.*

The conditional invariants inferred by cluster analysis can be useful to programmers for a variety of tasks. In two experiments, we show that they are useful in a static verification task and can help programmers identify bugs in software programs.

Thesis Supervisor: Michael D. Ernst
Title: Assistant Professor

Acknowledgments

I would like to extend my deepest gratitude to the numerous people who have supported me throughout this thesis:

To my thesis advisor, Prof. Michael Ernst, for his consistent support throughout every stage of this thesis. His active involvement in the work of all his students is uncommon and deeply appreciated. I have gained tremendously from working with him — academically, professionally and personally.

To Kate Baty and Carol Fredericks for making MIT like home to me. For their constant support, advice and encouragement and for being always there for me.

To my friends Selam Daniel, Cabin Kim, Paul Njoroge, Tawanda Sibanda, Kristin Wright and countless others for their friendship, prayers and support.

To other members of the Program Analysis Group in MIT's Laboratory for Computer Science for their help with experiments, bugs and advice.

To my parents for being the reason I've come this far. Thanks for your generosity, encouragement, prayers and for your immense and undying faith in me.

Above all my thanks to God, the fountain of life and the spring of wisdom and understanding.

Contents

List of Figures

Chapter 1

Introduction

A program invariant is a property that is true at a point or at different points in a program. Examples include $x > y$; $y = 2 * x + 3$; and the array A is always empty. Invariants can appear in the form of `assert` statements or formal specifications in programs. Invariants play an important role in program development and evolution. In program development, they can help in statically verifying properties such as type declarations, in runtime checking of invariants encoded as `assert` statements, and in refining a specification into a correct program. In program evolution, they can keep programmers from inadvertently violating assumptions upon which the program's correct behaviour depends. Despite their importance, programmers do not usually write invariants explicitly because of the time and tedium involved in the process of formalizing and writing the invariants.

Daikon [ECGN01] is a tool that infers program invariants. Daikon uses dynamic techniques to infer invariants of a program from execution traces of the program. It first instruments the program to record the values of all variables in scope at various points in the program. Next, it runs the program with a set of test cases supplied by the user. It then postulates and checks invariants against the variable traces. Previous work has shown that the invariants reported by Daikon are mostly sound, relevant and helpful to programmers in performing programming tasks [NE02a, NE02b, ECGN01].

A conditional invariant consists of two parts — a *consequent* and a *predicate*. The consequent is true only when the predicate is also true. For instance, the local invariant over a node of a sorted binary tree, ($\mathsf{n.left.value} \leq \mathsf{n.right.value}$), is only true if `n`, `n.left` and `n.right` are non-null. Conditional invariants are particularly important for the analysis of recursive data structures, where different properties typically hold in the base case and the recursive case, but are also important whenever a program exhibits different behaviors over different ranges of inputs.

Given a predicate p, Daikon can find conditional invariants of the form $\mathsf{p} \Rightarrow \mathsf{q}$ for invariants `q` in Daikon's vocabulary (a description of the technique for conditional invariant detection is given in Section 2.1). However, it is infeasible to check for conditional invariants for every possible predicate `p` in Daikon's vocabulary, because there is a prohibitively large number of them. It is therefore important to judiciously choose the predicates from which to infer conditional program properties for two reasons — to reduce computational cost and to report only those conditional invariants that are relevant and useful (see Section 2.2, page 11).

This thesis presents a new technique for dynamically inferring predicates for conditional invariant detection. Conditional invariants separate the input (and output) spaces of programs (and functions) into groups. For each conditional invariant, there is a group of inputs (or outputs) that satisfies the predicate of the invariant, and another group that does not. We use *cluster analysis* to detect clusters in traced values of program executions, and use invariants over these clusters as predicates for conditional invariant detection. This is done in a two-stage process. The first stage clusters program executions and infers invariants over these clusters. In the second stage, the invariants which were found only in some clusters but not in others are used as predicates to detect conditional invariants.

The remainder of this thesis is organized as follows: Chapter 2 gives background on dynamic invariant detection using Daikon and motivates the problem. Chapter 3 gives a rationale for the clustering technique, describes it in detail and gives background on cluster analysis. Chapter 4 describes other techniques for choosing predicates for conditional invariant detection besides clustering. In Chapter 5, we evaluate the performance of our technique in a static checking experiment (Section 5.1) and a bug detection experiment (Section 5.2). Chapter 6 describes the applications and contributions of our work, and related work. Chapter 7 concludes the thesis.

Chapter 2

Background: Invariant Detection Using Daikon

Dynamic invariant detection [ECGN01, EGKN99] discovers likely invariants from program executions by instrumenting the target program to trace variables of interest, running the instrumented program over a test suite, and inferring invariants over the instrumented values (Figure 2-1). During the inference step, a set of postulated invariants are tested against the values captured from the instrumented variables; those invariants that are tested to a sufficient degree without falsification are reported.

Daikon detects invariants at specific program points, such as procedure entries and exits, and invariants over the program as a whole; each program point is treated independently. The invariant detector is provided with a variable trace that contains, for each execution of a program point, the values of all variables in scope at that point. It then checks the set of postulated invariants at that program point against various combinations of the variables, maintaining those that are verified by the variables and discarding the falsified invariants.

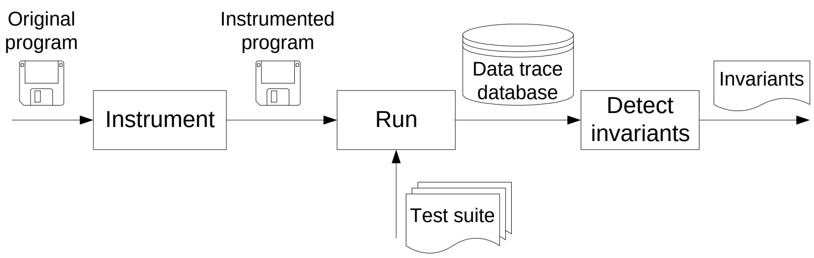

Figure 2-1: Architecture of the Daikon tool for dynamic invariant detection

Given variables `x`, `y` and `z`, and computed constants `a`, `b` and `c` at a program point, Daikon postulates invariants over single variables such as equality with a constant ($\mathsf{x} = \mathsf{a}$), membership in a small set of constants (x in $\{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$) or in a range ($\mathsf{a} \leq \mathsf{x} \leq \mathsf{b}$), non-zero, modulus ($\mathsf{x} \equiv \mathsf{a}(\mathsf{mod}\ \mathsf{b})$), etc. For binary invariants, it checks linear relationships ($\mathsf{z} = \mathsf{ax} + \mathsf{by} + \mathsf{c}$), ordering ($\mathsf{x} \leq \mathsf{y}$), functions $\mathsf{x} = \mathsf{f}(\mathsf{y})$ for built-in unary functions, etc. It also checks ternary invariants and invariants over collections such as lexicographic ordering, element ordering, invariants holding for all elements of a sequence or membership (`x in y`). Given two sequences, some examples of checked invariants include element-wise linear relationships, lexicographic comparison and subsequence relationships.

To enable reporting of invariants involving components or properties of aggregates, Daikon represents such entities as additional derived variables. For instance, if an array `A` and an integer `i` are both in scope at a program point, then properties over `A[i]` and `size(A)` may be of interest, even though these quantities may not appear in the program text. Derived variables are treated just like other variables by the invariant detector, permitting the engine to infer invariants that are not hardcoded into the list.

Daikon also detects invariants over more complex data structures such trees and other recursive structures. It does this by linearizing graph-like data structures, enabling it to report such invariants as mytree is sorted by the relation $<$. It is thus able to report global invariants over entire data structures, as well as local invariants, which relate only a small number of objects.

For each variable or tuple of variables in scope at a given program point, each potential invariant is tested. All invariants are checked against all applicable combinations of variables. An invariant that is invalidated by a sample is discarded immediately. Daikon maintains acceptable performance because false invariants are usually falsified quickly, so the cost of computing invariants tends to be proportional to the number of unfalsified invariants. All the invariants are inexpensive to test and do not require full-fledged theorem proving.

An invariant is only reported if there is adequate evidence of its plausibility. In particular, if there is an inadequate number of samples of a particular variable, invariants detected might be mere coincidence. Consequently, for each detected invariant, Daikon computes the probability that such a property would appear by chance in a random input. The invariant is only reported of this computed probability is less than a user-defined confidence parameter.

2.1 Conditional Invariant Detection

Detecting conditional invariants is nothing more than detecting invariants on subsets of the data trace file. Daikon infers conditional invariant by splitting the data trace into parts based on selected predicates. Users can supply these predicates to Daikon on a per-program basis in a file called the *splitter info file*. This file contains the predicates from which conditional invariants will be inferred. For each predicate, Daikon creates a *splitter* — a concrete instantiation of this predicate — that divides the data trace at a program point into the executions that satisfy the splitting condition, and those that do not. It then infers invariants over these groups separately. These predicates are also known as *splitting conditions* because they split the program executions into two groups. Finally, the separately detected invariants are combined into implications, where possible. If the splitting condition is poorly chosen, then the invariants on both sides of the split are identical and no implications are formed. The problem of conditional invariant detection can therefore be reduced to the task of choosing the right predicates for the detection of these invariants. Figures 2-3 and 2-2 illustrate the process of conditional invariant detection.

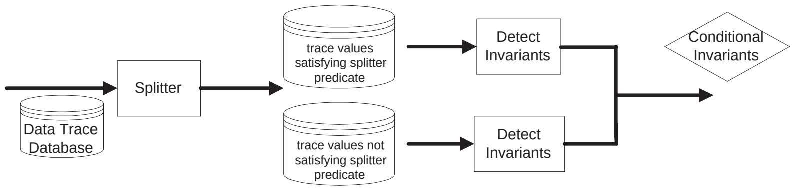

Figure 2-2: The splitter mechanism for detecting conditional invariants. Invariant detection is performed independently on the two groups produced by the splitter. Implications are then created from these sets of invariants (Figure 2-4).

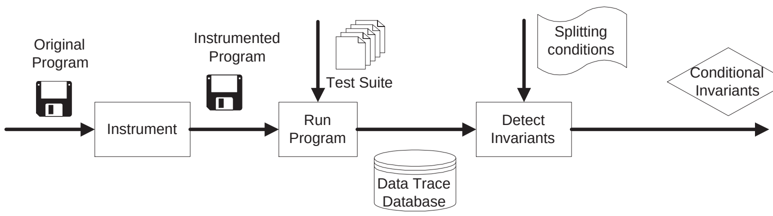

Figure 2-3: Conditional invariant detection using the splitter info file

In the current implementation of Daikon, implications are produced on a per-predicate basis. Each predicate potentially produces two subsets — T_1 and T_2 — of the data, one on either side of the split. Pairs of mutually exclusive invariants are then identified in the two subsets. A mutually exclusive invariant implies every other invariant in its set. It is not necessary to use any universally true property (a property appearing in both subsets) as the consequent of an implication. In other words, if c is universally true, there is no sense forming the implication $\mathsf{a} \Rightarrow \mathsf{c}$ for some predicate a.

Figure 2-4 gives pseudocode for the implication inference stage. The Create-Implications routine is run twice, swapping the arguments during the second run. The resulting implications are then simplified according to the standard rules of boolean logic. A concrete example of the operation of Create-Implications is given in Figure 2-5.

The algorithm currently does not find implications across more than two splits of the data. That is, if two or more predicates partition the same subspace of the data (for example $\mathsf{x} < 5$, $5 < \mathsf{x} < 10$ and $\mathsf{x} > 10$), it does not consider implications that hold over more than one subset, but not all the subsets of the data. An example of such an implication is $(\mathsf{x} < 5 \texttt{ or } \mathsf{x} > 10) \Rightarrow \mathcal{I}$, produced when $\mathsf{x} < 5 \Rightarrow \mathcal{I}$, $\mathsf{x} > 10 \Rightarrow \mathcal{I}$ and $5 < \mathsf{x} < 10 \Rightarrow \neg\mathcal{I}$.

The predicates appearing in implications need not be the splitting conditions, but they can be any invariants detected in a subset of the data. Many of the resulting implications would not be created if predicates were limited only to pre-specified splitting conditions. Moreover, in many cases, Daikon is able to refine the splitting

{Create implications from elements of S_1 to elements of S_2. S_1 and S_2 are sets of invariants.}
CREATE-IMPLICATIONS(S_1, S_2)
for all $s_1 \in S_1$ **do**
  if $\exists s_2 \in S_2$ such that $s_1 \Rightarrow \neg s_2$ and $s_2 \Rightarrow \neg s_1$ **then**
    {s_1 and s_2 are mutually exclusive}
    for all $s' \in (S_1 - S_2)$ **do**
      output "$s_1 \Rightarrow s'$"
    end for
  end if
end for

Figure 2-4: Pseudocode for creation of implications from invariants over mutually exclusive subsets of the data.

Invariants T_1	Invariants T_2	Implications	Implications	Simplified
a	$\neg a$	$a \Rightarrow b$	$\neg a \Rightarrow \neg b$	$a \Leftrightarrow b$
b	$\neg b$	$a \Rightarrow d$	$\neg a \Rightarrow f$	$a \Rightarrow d$
c	c	$a \Rightarrow e$	$\neg b \Rightarrow \neg a$	$a \Rightarrow e$
d	f	$b \Rightarrow a$	$\neg b \Rightarrow f$	$\neg a \Rightarrow f$
e		$b \Rightarrow d$		
		$b \Rightarrow e$		

Figure 2-5: Creation of implications from invariants over subsets of the data. The left portion of the figure shows invariants over two subsets T_1 and T_2. The middle portion shows all implications that are output by two calls to the CREATE-IMPLICATIONS routine of Figure 2-4; `c` appears unconditionally, so does not appear in any implication. The right portion shows the implications after logical simplification.

condition into a simpler or more exact predicate (see Figure 3-2, page 16). In practice, the splitting condition appears as an invariant in one subset of the data, and its negation appears over the other subset. However, the splitting condition might not be produced by Daikon in the subset if it is inexpressible in Daikon's grammar, if a stronger condition is detected by Daikon or if it is not statistically justified because the subset is too small. Suppressing redundant or unjustified invariants greatly reduces the clutter in Daikon's output and typically does not reduce the usefulness of the result.

2.2 The Problem

As mentioned in Section 2.1, the principal difficulty in inferring conditional invariants is determining the right predicates with which to split the data. There are two reasons why careful selection of predicates for conditional invariant detection is important. These are:

1. Computational Feasibility
   Every program has an infinite number of candidate predicates and conditional invariants. It is computationally infeasible to try all invariants in Daikon's grammar as possible predicates. The work done in invariant detection increases linearly with the number of splitting conditions used.

2. Relevance
   Our aim in inferring conditional invariants, is to increase Daikon's grammar without burdening the user with more incorrect invariants. Too many splits will likely increase the number of incorrect invariants (see Section 5.1.1, page 28), clutter the output and make it harder to read Daikon's output. It is therefore important to select only those splits that are likely to produce useful invariants, in this way reducing clutter in Daikon's output.

In this thesis, we investigate how we can automatically predict which predicates are best using cluster analysis.

Chapter 3

Using Cluster Analysis to Infer Predicates for Splitting

3.1 Rationale

The main contribution of this thesis is the application of cluster analysis to the extraction of useful splitting conditions for conditional invariant detection. Cluster analysis [KR90, And73] is a multivariate analysis technique that is used to find groups or clusters of relatively homogeneous objects within a data set. The aim of cluster analysis is to find clusters that are internally highly homogeneous (members of the same group are similar to one another) and externally highly heterogeneous (members of different groups are different from one another.) It has been extensively applied to such diverse areas of research as biology [SS73], chemistry [MK83], psychiatry [EGK71], economics [Fis69], marketing research [GFR67], financial analysis [Kin83], sociology [Hud82], and cybernetics [TNH99]. In all these areas it has been used to analyse, simplify, and interpret large amounts of multidimensional data.

A "natural" cluster in a data set is a point or set of similar points, whose features are significantly different from all other points in the data set, and therefore easily distinguishable in the data set. The idea of using cluster analysis to find predicates for conditional invariant detection arises from the observation that any conditional property divides the applicable program executions into groups — those that satisfy the predicate and those that do not. For example, for a tree data structure, each node has a non-null parent node, except the root, whose parent node is null. In this case, the two groups are the root, and all other nodes. The hypothesis, then, is that if natural clusters or groups exist in the program executions at a program point, they could be the result of conditional properties inherent in the program. The aim of the cluster analysis approach is to infer the splitting conditions for these conditional properties by a reverse approach: find clusters in the data, infer invariants over these clusters and use the inferred invariants as splitting conditions.

3.2 Implementation

Figure 3-1 gives a general overview of the steps needed to find conditional invariants using cluster analysis.

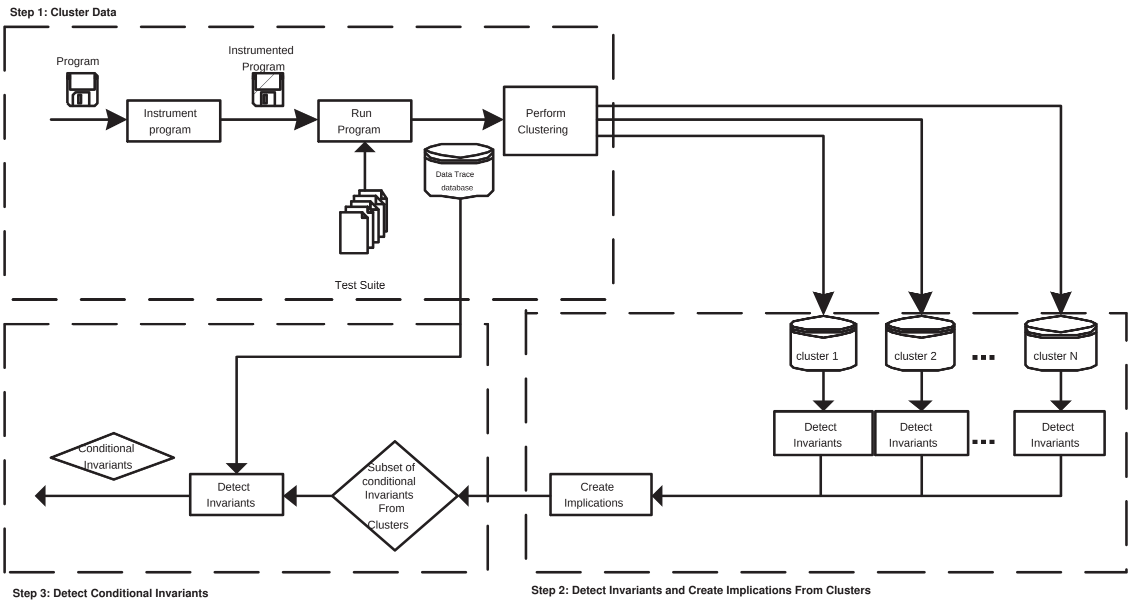

Figure 3-1: An overview of the application of cluster analysis to invariant detection. This is an extension of the mechanism described in Figure 2-1. In step 1, the data trace file is separated into clusters. In step 2, invariants are inferred in the different clusters separately. Invariants found only in a subset of the clusters, but not in the whole data set, are extracted. In step 3, the extracted invariants are translated into Java statements and supplied to Daikon as splitting conditions.

The three steps in our clustering technique are all performed automatically without any human input in intermediate steps. Step 1 is the clustering stage, where the program executions at each program point are separated into clusters. We cluster the executions at the *exit* of each procedure and assign each execution (both entry and exit) to a single cluster. (We chose the exit program point as the reference for cluster groupings because the variable values at the exit of the program contain more information about the behaviour of the procedure than the values at the entry into the program. The `return` value of the program, for instance, is only available at the exit of the program.)

The feature vector representing a single execution of a procedure includes the numeric variables (derived and otherwise) and boolean variables in scope at that program point. For simplicity, we make no distinction between categorical variables (variables that take on a small discrete set of values, for example, boolean variables) and real-valued variables. (More sophisticated methods for dealing with mixtures of categorical and numerical data have been proposed. These include forming indicator variables for the categories of a categorical variable and performing a principal components analysis of all the variables [DHS00].) Before clustering, we normalize the data (see Section 3.3.4) so that each dimension (variable) has a standard deviation of 1 and a mean of 0. This gives all the variables equal weights in the clustering algorithm.

In step 2, the original data trace file is augmented to add a new variable, `cluster`, to the data trace file. The value of this variable indicates the cluster number associated with that particular invocation. We run Daikon with the augmented data trace file, split the data into the clusters found by the clustering tool, and infer conditional invariants over these clusters. This produces conditional invariants of the form $(\mathsf{cluster} = \mathsf{k}) \implies \mathcal{I}$, where $\mathcal{I}$ is some invariant unique to cluster k. The invariants on the *right hand side* (consequents) of all such implications are extracted from the Daikon output, translated into Java statements and put into a splitter info file. These splitting conditions are invariants that were true in at least one, but not all clusters at a program point.

In step 3, we run Daikon on the original data trace file with the splitter info file obtained from step 2 as input. We obtain conditional invariants of the form $\mathcal{I} \implies \mathcal{J}$, where $\mathcal{J}$ is an invariant that depends on some condition or predicate obtained in step 2.

3.2.1 Refining Splitting Conditions

The procedure above employs invariant detection (in step 2) to produce a set of splitting conditions that can be employed in a second invocation of invariant detection (in step 3). An alternative one-pass technique might add the *cluster* variable to a trace file, perform invariant detection over that trace file, and filter references to the *cluster* variable out of the resulting invariants. Our two-pass technique, which performs invariant detection twice, is preferable to the one-pass technique for two reasons.

First, the two-pass technique produces a set of splitting conditions that can be used on the original data trace. They are human-readable, easily inspected and edited, and can be reused during other invariant detection steps. Second, performing invariant detection helps to refine the cluster information. Clustering is inherently statistical and inexact; it may not partition the data exactly as desired. However, as long as one cluster induces one of the desired properties and some other cluster does not, the extra invariant detection step can leverage this into the desired splitting condition. If the original clustering produces the desired grouping exactly, then the additional step does no harm.

As an example, consider Figure 3-2. The clusters nearly, but do not exactly, match the true separation between behaviors, so no non-cluster-related invariants can be reported. However, the implication $(\mathsf{cluster} = 1) \implies (\mathsf{x} < 0)$ is produced. Using $\mathsf{x} < 0$ as a splitting condition in a subsequent invariant detection pass produces the desired properties $(\mathsf{x} < 0) \iff \triangle$ and $(\mathsf{x} > 0) \iff \square$.

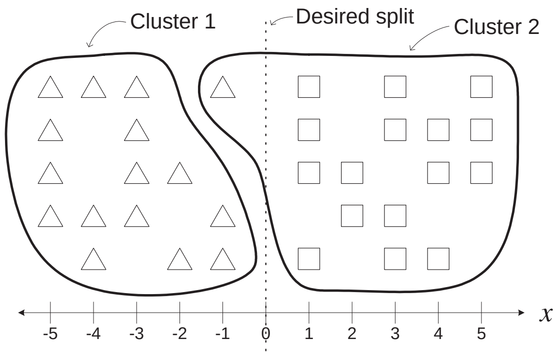

Figure 3-2: Refinement of cluster information via an extra step of invariant detection. The first clustering process approximates, but does not exactly detect, the natural division in the data set (between triangles and squares, at $x = 0$.) An extra invariant detection step produces the desired property ($x < 0$) as a splitting condition.

3.3 Cluster Analysis

This section discusses implementation choices required to perform cluster analysis. These include:

- selecting a suitable distance metric (Section 3.3.1),
- selecting a suitable clustering algorithm (Section 3.3.2),
- dividing the data into the number of clusters that correctly represents the data set (Section 3.3.3), and
- choosing suitable weights for the differently scaled features representing the objects (Section 3.3.4).

3.3.1 Choice of a Distance Metric

In cluster analysis, each data point is represented by a (usually numerical) set of features or properties, called a *feature vector*. Clustering algorithms employ the *similarity* or *dissimilarity* between objects in assigning them to groups. The similarity between two objects is measured by a distance metric applied to their feature vectors. Two commonly used distance metrics are the Euclidean distance and the Manhattan distance. Given two n-dimensional feature vectors **x** and **y**, the Euclidean distance is given by

$$d_E(x, y) = \left[\sum_{i=1}^{n} (x_i - y_i)^2 \right]^{1/2}$$

The Manhattan distance is given by

$$d_M(x,y) = \sum_{i=1}^{n} |x_i - y_i|$$

The choice of distance metric has an important effect on the clustering. The Euclidean and Manhattan distances, for example, produces clusters that are invariant to rotation or translation, but not invariant to linear transformations or other transformations that distort the distance relationship. Therefore a simple scaling of the coordinate axes might result in an entirely different grouping [DHS00]. In our k-means and hierarchical clustering algorithms, we use the Euclidean distance metric, which is one of the most commonly used in clustering algorithms.

3.3.2 Clustering Algorithms

Choosing a clustering algorithm for a particular problem can be a daunting task. There are two main approaches to cluster analysis: partitioning and hierarchical clustering. This thesis evaluates two different implementations of the k-means partitioning algorithm (a simple k-means algorithm and an extension of the k-means algorithm to efficiently detect the number of clusters), and an implementation of a hierarchical clustering algorithm.

Partition Clustering

Partition clustering algorithms define a cost function

$$c \ : X : X \subseteq S \rightarrow \Re^{+}$$

that associates a cost with each cluster of points. The goal of the algorithm is to minimize $\sum_{i=1}^{k} c(S_i)$, the sum of the costs of the clusters. The most well-known criterion is the *sum-of-squares* criterion, which assigns a cost

$$C(S_i) = \sum_{r=1}^{|S_i|} \sum_{s=1}^{|S_i|} (d(x_r^i, x_s^i))^2$$

to each cluster, with $d(x_r^i, x_s^i)$ being the distance between the points x_r^i and x_s^i the r^{th} and s^{th} elements of the set S_i.

Partition clustering algorithms take as input, a set S of N objects and an integer k, divide S into k clusters or subsets $S_1, S_2, ..., S_k$ (arbitrarily), and iteratively improve the solution by reassigning points to different clusters until convergence is reached (i.e., until reassigning the points does not reduce the total cost of the clusters).

These methods are also called optimization techniques because they seek a partition of the S that optimizes a predetermined numerical measure. The most common partition clustering algorithm is the k-means algorithm. It is described below:

1. Select k initial cluster centroids. The initial centroids determine the final cluster groupings. There is a significant amount of research on heuristics for selecting the k initial centroids for best results [PLL99].
2. Choose a distance metric.
3. Assign each instance x in S to the nearest cluster centroid.
4. For each cluster, recalculate the cluster centroid based on the new instances in the cluster.
5. Repeat items 3 to 5 until convergence.

The iterative approach is appealing because it allows the possibility for a poor initial clustering to be corrected in later stages. It has several drawbacks, however. In many cases, the results from a k-clustering are not unique. The result depends on the initial selection of k centers, and the algorithm is known to converge to a local optimum in many cases. To minimize this problem, the algorithm is usually run several times and the best clustering is chosen from the different runs. Another problem is that these algorithms require the user to input the value of k. This can be a strength of this algorithm, if the user knows *a priori* the number of clusters in the data set. However, choosing a wrong value of k may impose structure on the data set and hide the true structure. A further problem with optimization methods is that they heavily favor spherical clusters in the data, even when the clusters in the data are not spherically shaped.

Hierarchical Clustering

Hierarchical clustering algorithms take a data set S and produce, as output, a tree $T(S)$ in which the nodes represent subsets of S. There are two general types of hierarchical clustering algorithms. *Divisive* algorithms start by assigning all the points to one cluster. At each step, a cluster is divided into two. This recursive partitioning continues until only singleton subsets remain. *Agglomerative* algorithms, which are more common, begin with singleton subsets and recursively merge the most similar subsets at each stage until there is only one set. Hierarchical clustering methods are faster than partitioning methods, but produce lower quality clustering because a bad merge or division carried out in an early stage cannot be undone. They are also appealing because partitions with varying numbers of clusters can be retrieved from the tree in a single run. A traversal of a well-constructed tree visits increasingly tightly-related elements.

3.3.3 The Number of Clusters in a Data Set

Often, the most difficult problem in the clustering process is determining the optimal number of clusters. Clustering the data set into the wrong number of clusters may lead to failure to detect the desired conditional invariants because a single point in the wrong cluster may falsify all the cluster-specific invariants, or may lead to the detection of conditional invariants formed over subsets of the data with no real meaning to the user.

Finding the right number of clusters in a data set is a hard problem to solve, even when the data is organized in clusters. Dividing n objects, each represented by a point in p-dimensional space, into clusters that reflect the underlying structure of the data is an NP-hard problem, even when the number of clusters is known [GJ78]. Hierarchical clustering enables one to traverse the hierarchical tree structure and therefore the user can view the different clusters produced by a single run of the algorithm, as the number of clusters increases from 1 cluster to N singleton clusters. Partitioning methods, on the other hand, require multiple runs in order to see the clusterings produced for different values of k.

Many different approaches have been used to infer the number of clusters in a data set. A common approach is to run a clustering algorithm on the data set with different values of k and then pick the best clustering. We evaluate our k-means algorithm on 10 programs for $k = 2, 3, 4$ and 5 (see Section 5). We also evaluate the x-means algorithm [PM01], an algorithm that estimates the number of cluster in a data set by iteratively running k-means and in each iteration, adding or removing centroids where they are needed, while optimizing some numerical measures calculated over the clusters (the Bayesian Information Criterion (BIC) or the Akaike Information Criterion (AIC)). The x-means algorithm does a good job at estimating the number of clusters in a data set. Experiments performed over synthetic gaussian data sets [PM01, PM00] showed that it estimated the correct number of clusters to within 15% accuracy and produced higher quality clusters with lower distortion values than the k-means algorithm, even when k-means was given the true number of clusters.

3.3.4 Data Standardization

In real world problems, we do not expect all the variables to have the same type or the same dimensions. The variables we encounter could be interval-based (numerical), binary, categorical (nominal), ordinal, or mixtures of these data types. The units chosen for measuring the different variables can arbitrarily affect the similarities between objects. If different units are used for different variables, the variable with the largest dispersion will have the largest impact on the clustering; merely changing the units of one variable can drastically alter the resulting groupings. For example, if the variable `x` ranges between 0 and 100, and the variable `y` ranges between 0 and 1, then the values of `x` will carry more weight in determining the similarities between objects unless we standardize the data. Standardizing the data values and casting them into dimensionless quantities allows all the variables to contribute equitably to the similarities among data points. For lack of a better prior information about the relative importance of the variables, we standardize each variable to have zero mean and unit variance.

For mixtures of variables with nominal, ordinal, interval and/or ratio scales there are rarely clear methodological answers to choosing either a distance metric or a standardizing function [DHS00].

Chapter 4

Other Strategies for Selecting Splitting Conditions

The value of our new technique cannot be fully measured unless we compare it to other techniques for conditional invariant detection. In this chapter we discuss other techniques and ideas for conditional invariant detection as a basis for Chapter 5, where we empirically compare our technique with some of them. The current techniques used to select predicates for conditional invariant detection include built-in methods (procedure return analysis), static analysis, random sampling and context sensitive analysis.

4.1 Built-in Methods: Procedure Return Analysis

There are two effective splitting strategies built into Daikon for conditional invariant detection. The first splits the data at the exit of a function if the function returns a boolean value. It separates the cases for which the function returns `true` from those for which it return `false`. The second built-in mechanism splits data based on the return site. If a procedure has multiple `return` statements, then it is likely that they represent different behaviours of the program: a normal case, an exceptional case, a base case, etc. The executions are therefore split based on their exit point from the procedure.

4.2 Static Analysis

In static analysis, splitting conditions are chosen by analysing the program's source code. Boolean condition used in the program (specifically predicates in `if` and `while` loops), and boolean member variables, are extracted from the source code and used as splitting conditions. The assumption is that conditional statements that are explicitly tested in the program's source are likely to play an important role in the program's behavior and may lead to useful conditional invariants.

Static analysis is scalable, efficient and inexpensive because obtaining splitting conditions from this method requires only textual analysis of the program source.

However its drawback is its inability to infer conditional properties whose predicates are not explicitly stated in the source[1]. For example in `QueueAr` (an array-based Java implementation of a Queue [Wei99]), the invariant
$(\mathsf{currentSize} > 0 \ \&\& \ \mathsf{back} < \mathsf{front}) \Longrightarrow (\mathsf{theArray}[\mathsf{back}+1 \ldots \mathsf{front}-1] = \mathsf{null})$
describes the relationship between the array indices `front` and `back` (the front and back of the queue) and the parts of the array containing useful data. This invariant cannot be detected by static analysis because the condition $\mathsf{back} < \mathsf{front}$ is never explicitly tested in the program.

4.3 Random Sampling

In the random selection approach [DDLE02, RKS02], we first select r different subsets of the data trace, each of size s, and perform invariant detection over each subset separately. We then use any invariant detected in one of the subsets, but not in the full data trace, as a splitting condition. The intuition behind this approach is as follows: suppose that some property holds in a fraction of the data. It will never be detected by ordinary (unconditional) invariant detection. However, if one of the randomly-selected subsets of the data happens to contain only executions over which that property holds, then the condition will be detected in the random sample and redetected when the splitting conditions are used in invariant detection. For a property that holds in a fraction $f < 1$ of all samples, the probability that all the elements of a randomly selected subset have that property is given by $p = f^s$. Thus, the property holds in at least one of the subsets, and is detected by the random selection technique with probability $1-(1-p)^r$. This quantity is graphed in Figure 4-1 for different values of r and s.

An alternative approach would compare invariants detected over the subsets with one another rather than with invariants detected over the full data. This avoids the need for another potentially costly run of Daikon. The analysis changes as follows: the property holds in either all or none of the s elements with probability $p = f^s+(1-f)^s$. The property (or its negation) holds in at least one of the subsets with probability $1-(1-p)^r$. However, if it holds on all subsets, it is still not detected, so we adjust the value down to $(1-(1-p)^r)(1-f^{rs}-(1-f)^{rs})$. Figure 4-2 graphs this quantity against f, for several values of s and r, and shows that this approach is superior for relatively uncommon properties ($.02 < f < .4$) but worse for rarely violated properties ($f > .98$). If both the property and its negation are expressible in Daikon's grammar, as is usually the case, then the original approach always dominates.

Figures 4-1 and 4-2 show that the random selection approach is most effective for unbalanced data; when f is near .5, it is likely that both an example and a counterexample appear in each subset of size s.

The larger the value of s, the more likely it is to include sample executions that do not have the desired property. Therefore it is desirable to choose small values of s and large values for r. The danger of reducing s is that the smaller the subset, the

[1] This drawback holds for the built-in methods also.

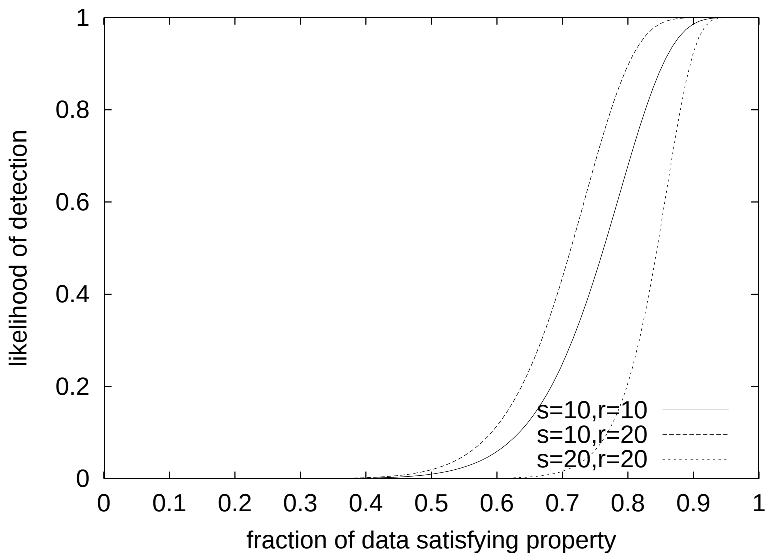

Figure 4-1: Likelihood of finding an arbitrary split via random selection, comparing against invariants over the full data set. s is the size of each randomly-chosen subset, and r is the number of subsets.

more likely that the invariants are not statistically justified and the more likely that any resulting invariants overfit the small sample. The danger of increasing r is that the amount of work linearly increases with r.

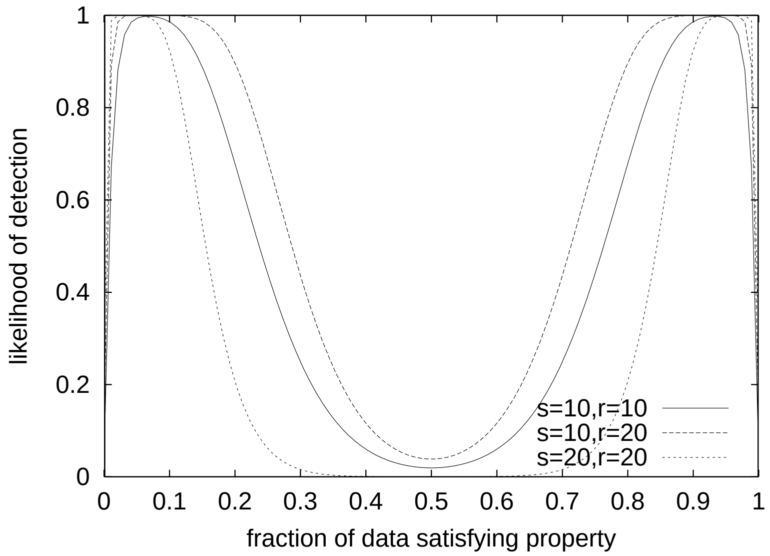

Figure 4-2: Likelihood of finding an arbitrary split via random selection, without comparing against invariants over the full data. s is the size of each randomly-chosen subset, and r is the number of subsets.

This randomized approach has the same strength as cluster analysis, in that it can find splitting conditions that do not depend on the control flow of the program.

We expect to get better splitting conditions in the cluster analysis approach than in random sampling because cluster analysis is effectively a random division of the data into groups, with the groupings subsequently refined to introduce some structure into them.

4.4 Other Methods

In addition to the methods mentioned above, a programmer may select splitting conditions by hand or according to any other strategy, and manually write them into a splitter info file. Writing sensible and useful splitting conditions requires a fair amount of knowledge and analysis of the program and its properties, hence the need for automatic ways to derive these splitting conditions.

Other promising candidates for splitter selection that have not yet been incorporated into Daikon include:

- A *special values* policy, that compares a variable to preselected values chosen statically (such as null, zero or literals in the program), or dynamically (such as commonly-occuring values, minima, or maxima.)

- A policy based on *exceptions to detected invariants*, that tracks variable values that violate potential invariants, rather than immediately discarding the falsified invariants. If the number of falsifying samples is moderate, those samples can be separately processed, resulting in a nearly-true invariant plus an invariant (or invariants) over the exceptions.

Chapter 5

Evaluation

We evaluate the invariants produced by cluster analysis in two different experiments. The first experiment measures the accuracy of the invariants in two program verification tasks (Section 5.1). The second experimental evaluation measures our method's performance in an error detection experiment (Section 5.2). In both experiments, we compare the results of cluster analysis with the base case (no conditional invariants from splitting conditions), static analysis and random sampling. In the static analysis experiment, we also compare various clustering approaches to one another.

For both evaluation experiments, we report our results through measurements of *precision* and *recall*, standard measures from information retrieval [Sal68, vR79]. Assuming we have a goal set of invariants needed to perform a certain task (for example reveal bugs in code), the precision is defined as

$$\text{precision} = \frac{\text{number of reported invariants that are correct}}{\text{total number of reported invariants}}$$

and it gives a measure of correctness of the reported invariants. Differences in precision indicate the relative degree to which new techniques worsen the overfitting problem and increase the number of incorrect invariants

The recall of our invariant set is defined as

$$\text{recall} = \frac{\text{number of reported invariants that are correct}}{\text{total number of invariants in goal set}}$$

and it gives a measure of how complete the reported set of invariants is. Differences in recall indicate the relative degree to which new techniques improve retrieval of the goal set.

Sections 5.1 and 5.2 describe the two experiments and how these two measures are calculated.

5.1 Static Checking

The static checking experiment uses the ESC/Java static checker. ESC/Java [DLNS98, Det96] is a static analyzer that detects common errors in programs, such as null dereference errors, array bounds errors and type cast errors, that are usually not detected until runtime. In addition, it can be used to verify annotations written into the program, indicating those that are guaranteed to be true and those that are unverifiable. Programmers must write program annotations, many of which are similar to *assert* statements, but they need not interact with the checker as it processes the annotated program.

Daikon can automatically insert its output into a Java source file as ESC/Java annotations. These annotations may not be completely verifiable by ESC/Java. Some annotations may require removal either because they are not universally true or because their verification is beyond the verifier's capabilities. Verification may also require addition of missing annotations when those annotations are necessary for the correctness proof or for the verification of other annotations in the Java source. There are potentially many different sets of ESC/Java-verifiable annotations for a given program. One set might, for instance, ensure that a representation invariant is maintained, while another set might ensure the absence of run-time errors in the program.

We performed two experiments. The first used as goal, the set of invariants that establish the absence of run-time errors in the program, and that required the smallest number of changes to Daikon's output. Some correct invariants from Daikon's output had to be removed because they required the addition of several more annotations in order to verify. However, invariants from Daikon's output that were not required to establish the absence of run-time errors but were still verifiable were left intact. In Section 5.1, we give the results of this experiment. The second experiment used as goal, the set of invariants that included all the correct and verifiable invariants from Daikon's output. In a few cases, we had to add several more annotations to enable verification of some of the correct invariants in Daikon's output. Section 5.1.3 (page 31) gives the results of this experiment.

For both experiments, we performed the addition and removal of annotations by hand. Given a set of invariants reported by Daikon and the changes necessary for verification, we counted the number of reported and verifiable invariants (`ver`), reported but unverifiable (`unver`) and unreported but necessary invariants (`miss`). We computed precision as

$$\text{precision} = \frac{\texttt{ver}}{\texttt{ver} + \texttt{unver}}$$

and recall as

$$\text{recall} = \frac{\texttt{ver}}{\texttt{ver} + \texttt{miss}}$$

The experiments were performed for 10 small programs described in Figure 5-1. `DisjSets`, `StackAr` and `QueueAr` come from a data structures textbook [Wei99]. The remaining programs are solutions to assignments in a programming course at MIT.

This method of statically verifying Daikon's output using ESC/Java has been used in previous research [NE01, NE02b, NE02a]. In [NE02a], the experiment was performed for the programs described in Figure 5-1, using the base version of Daikon without splitting conditions. The results are shown in Figure 5-2.

Program	Program size LOC	NCNB	Meth.	Description
QueueAr	116	56	7	queue represented by an array
StackAr	114	50	8	stack represented by an array
RatNum	276	139	19	rational number
Vector	536	202	28	`java.util.Vector`
StreetNumberSet	303	201	13	collection of numeric ranges
DisjSets	75	29	4	disjoint sets. Support union, find
RatPoly	853	498	42	polynomial over rational numbers.
GeoSegment	269	116	16	pair of points on the earth
FixedSizeSet	76	28	6	set represented by a bitvector
Graph	180	99	17	generic graph data structure
Total	4886	2449	273	

Figure 5-1: Summary statistics of programs used in static verification evaluation. "LOC" is the total lines of code. "NCNB" is the non-comment, non-blank lines of code. "Meth" is the number of methods.

Program	Number of invariants Ver.	Unver.	Miss.	Accuracy Prec.	Rec.
QueueAr	42	0	13	1.00	0.76
StackAr	25	0	0	1.00	1.00
RatNum	25	2	1	0.93	0.96
Vector	100	2	2	0.98	0.98
StreetNumberSet	22	7	1	0.76	0.96
DisjSets	32	0	0	1.00	1.00
RatPoly	70	10	1	0.88	0.99
GeoSegment	38	0	0	1.00	1.00
FixedSizeSet	16	0	0	1.00	1.00
Graph	15	0	2	1.00	0.88

Figure 5-2: ESC/Java verification results for test programs with Daikon base version. The results are taken from [NE02a].
k is the number of clusters used for k-means and hierarchical clustering.
"Ver" is the number of reported invariants that ESC/Java verified.
"Unver" is the number of reported invariants that ESC/Java failed to verify.
"Miss" is the number of invariants not reported by Daikon but required by ESC/Java for verification.
"Prec" is the precision of the reported invariants, the ratio of verifiable to verifiable plus unverifiable invariants.
"Rec" is the recall of the reported invariants, the ratio of verifiable to verifiable plus missing.

The use of splitting conditions adds implications to Daikon's output. Adding such invariants may increase recall if these invariants were previously needed to verify the annotations but were missing. We predict that precision will generally decline (compared to the case without splitting conditions) because conditional invariants are inferred from subsets of the test cases used to infer unconditional invariants (because of the split.) They are therefore more likely to be untrue or unverifiable because there is more overfitting. Another reason why precision is likely to decrease is that we are increasing Daikon's grammar and reporting more invariants from the same amount of data. Recall may also be reduced if the new conditional invariants added by clustering require the addition of new invariants in order to be verifiable by ESC/Java.

Both anecdotal evidence and controlled user experiments [NE02b] have demonstrated that recall is more important than precision for users. Users can easily filter out incorrect invariants, but have more trouble producing annotations from scratch — particularly implications, which tend to be the most difficult invariants for users to write. Therefore adding implications may be worthwhile even if precision decreases, because it might eliminate the need for users to formulate these conditional invariants.

Figures 5-3 to 5-8 show our results from performing the same experiments for six of the programs: `QueueAr`, `StackAr`, `RatNum`, `Vector`, `StreetNumberSet` and `DisjSets`. Clustering did not yield any new splitting conditions for `Graph`. The programs have not changed between the two experiments, and the output of Daikon has not changed in any significant way to affect comparability of the results. We perform our experiments for the following cases:

1. The simple k-means algorithm, for $k = 2,\ 3,\ 4$ and 5.
2. A hierarchical clustering algorithm, for $k = 2,\ 3,\ 4$ and 5.
3. The x-means algorithm.

The differences in the number of invariants between the base case (no splitting conditions from clustering) and the other cases are due to the addition of implications.

`QueueAr`										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	42	0	13	1.00	0.76	—	—	—	—	—
2	75	16	7	0.82	0.91	62	7	7	0.90	0.90
3	65	22	8	0.75	0.89	61	10	6	0.86	0.91
4	82	18	7	0.82	0.92	62	10	6	0.86	0.91
5	96	22	7	0.81	0.93	96	22	7	0.81	0.93

X-means						
`max(k)`	`avg(k)`	Ver.	Unver.	Miss.	Prec.	Rec.
17	10.5	78	35	9	0.69	0.90

Figure 5-3: Results for `QueueAr` with clustering. The first row ($k = 1$) shows the results from [NE02a] (i.e., without implications from clustering). For the x-means algorithm, `max(k)` refers to the maximum number of clusters that was observed across all program points. `avg(k)` refers to the average number of clusters that x-means found. For each row other than the base case (row 1), the difference in the number of invariants between that row and the base case is caused by the addition of implications.

We evaluated the remaining programs with the x-means algorithm only. This is because of the observation that the results for the x-means algorithm, k-means and hierarchical clustering did not differ very much. The results are shown in Figure 5-9.

Stackar										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	25	0	0	1.00	1.00	—	—	—	—	—
2	35	4	0	0.90	1.00	39	8	0	0.83	1.00
3	39	8	0	0.83	1.00	40	8	0	0.83	1.00
4	36	4	0	0.90	1.00	40	8	0	0.83	1.00
5	47	8	0	0.83	1.00	40	8	0	0.83	1.00

X-means						
`max(k)`	`avg(k)`	Ver.	Unver.	Miss.	Prec.	Rec.
10	5.9	44	7	0	0.86	1.00

Figure 5-4: Results for `Stackar` with clustering.

Ratnum										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	25	2	1	0.93	0.96	—	—	—	—	—
2	55	30	3	0.65	0.95	40	22	2	0.65	0.95
3	50	28	3	0.64	0.94	38	45	3	0.46	0.93
4	50	28	3	0.64	0.94	47	41	3	0.53	0.94
5	58	50	4	0.54	0.94	45	44	4	0.51	0.92

X-means						
`max(k)`	`avg(k)`	Ver.	Unver.	Miss.	Prec.	Rec.
17	10.6	55	32	3	0.63	0.95

Figure 5-5: Results for `Ratnum` with clustering.

Vector										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	100	2	2	0.98	0.98	—	—	—	—	—
2	132	50	2	0.73	0.99	139	84	2	0.62	0.99
3	133	75	2	0.64	0.99	147	85	2	0.63	0.99
4	131	60	2	0.69	0.98	143	79	2	0.64	0.99
5	133	58	2	0.70	0.99	149	75	2	0.67	0.99

X-means						
max(k)	avg(k)	Ver.	Unver.	Miss.	Prec.	Rec.
16	7.3	132	78	2	0.63	0.99

Figure 5-6: Results for `Vector` with clustering.

5.1.1 Discussion

Splitting Conditions From Clustering Generally Reduce Precision

The results confirm our expectations. Precision generally decreased with the addition of splitting conditions from clustering. This is not only true when the splitting conditions come from cluster analysis. Conditional invariants generally have lower precision than unconditional invariants because they are inferred from a smaller set of data points.

StreetNumberSet										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	22	7	1	0.76	0.96	—	—	—	—	—
2	28	8	1	0.78	0.97	30	8	1	0.79	0.97
3	29	8	1	0.78	0.97	29	8	1	0.78	0.97
4	29	8	1	0.78	0.97	28	6	1	0.82	0.97
5	29	8	1	0.79	0.97	29	6	1	0.83	0.97

X-means						
max(k)	avg(k)	Ver.	Unver.	Miss.	Prec.	Rec.
1	2	26	12	1	0.68	0.96

Figure 5-7: Results for `StreetNumberSet` with clustering.

DisjSets										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	32	0	0	1.00	1.00	—	—	—	—	—
2	44	2	2	0.96	0.96	39	2	2	0.95	0.96
3	45	2	2	0.96	0.96	45	2	2	0.96	0.96
4	47	1	2	0.98	0.96	45	2	2	0.96	0.96
5	46	3	2	0.94	0.96	45	2	2	0.96	0.96

X-means						
max(k)	avg(k)	Ver.	Unver.	Miss.	Prec.	Rec.
17	8.2	41	1	2	0.98	0.95

Figure 5-8: Results for `DisjSets` with clustering.

	X-means					Base				
Program	Ver	Unver	Miss	Prec	Recall	Ver	Unver	Miss	Prec	Recall
RatPoly	51	17	1	0.75	0.98	70	10	1	0.88	0.99
GeoSegment	31	7	0	0.82	1.00	38	0	0	1.00	1.00
FixedSizeSet	22	0	0	1.00	1.00	16	0	0	1.00	1.00

Figure 5-9: Results of static verification for remaining programs. The results are only shown for the x-means algorithm.

However this drop in precision is not a big problem for two reasons. In the user study performed by Nimmer and Ernst [NE02b], a general observation was that low precision was not very bothersome. Users could easily identify the wrong invariants and remove them from Daikon's output. This observation was confirmed in our experiments too: the wrong invariants were generally easy to identify and eliminate from Daikon's output. Secondly, it is easy to increase the precision of the results with little additional work (addition of test cases to remove false invariants; see Section 5.1.1).

Splitting Conditions From Clustering Can Significantly Improve Recall

The results from [NE02a] (Figure 5-2) show that most of the programs had adequate recall with the base case of Daikon (meaning that in most cases, there were few missing invariants.) This left little room for cluster analysis to improve recall with the addition of implications. For these cases the precision was generally worse, although the recall was not affected negatively by the addition of splitting conditions from clustering.

There was significant recall improvement in the case of `QueueAr`. In [NE02a], there were 12 missing program invariants that were particularly troublesome for users to state. Of these, 11 were conditional invariants. The results from QueueAr show that the different clustering techniques were able to produce an average of 6 of these missing invariants, leading to a sizeable increase in the recall. This shows that in cases where implications are needed to improve recall, cluster analysis can be a valuable addition.

Effect of Test Suite Size

The drop in precision is due to overfitting, a problem that arises when wrong invariants persist during invariant inference because there are no data samples to invalidate them. Daikon's output makes it easy to improve the test suites because we can identify the wrong invariants and add test cases to falsify the wrong invariants.

For each program, we added a few more general test cases in approximately one hour to increase the precision. Figure 5-10 shows the results for four of the programs after adding new test cases. The results are shown only for the x-means algorithm.

Program	Ver	Unver	Miss	Prec	Recall	Orig Prec	Orig Recall
QueueAr	91	9	7	0.91	0.93	0.69	0.90
StackAr	39	0	1	1.00	1.00	0.86	1.00
RatNum	36	5	1	0.88	0.97	0.63	0.95
DisjSets	42	0	3	1.00	0.95	0.98	0.95

Figure 5-10: Results of static verification with augmented test suites. The results are only shown for the x-means algorithm with `QueueAr`, `StackAr`, `RatNum` and `DisjSets`. "Orig Prec" and "Orig Recall" refer to the precision and recall of the original test suites with x-means clustering.

These significant gains in precision indicate that it is worthwhile to improve the test suite after the first run of Daikon. It is relatively easy to improve the test suite in this way, because the bad invariants in the first run give valuable hints about test cases that need to be added to the test suite, or areas in the program that have not been adequately tested. There may also be additional benefits, such as inferring new invariants due to corner cases that were previously not adequately tested. However it is important to note that adding new test cases can change the output more than expected, because these new samples can lead to the detection or justification of other (sometimes unwanted) invariants.

Effect of the number of clusters

From the results shown, we cannot conclude much about the best number of clusters to use. **Increasing** the value of `k` could have two effects: It increases the ability of the program to isolate clusters of identical elements in the data by allowing it to break the data set into smaller groups. This could lead to better recall. However it also increases the possibility of getting spurious invariants because each cluster contains less data. This leads to lower precision. Increasing the number of clusters could also break up large clusters into smaller ones. This is not a problem though, because cluster specific invariants can be inferred in either of the sub-clusters.

Decreasing the number of clusters could lead to higher precision because we have more data in each cluster. However it could inhibit the ability of the program to isolate clusters of identical elements. For example, if the data is naturally grouped into three clusters and we use $k = 2$, it is likely that two of the clusters will be combined into one, or the third cluster will be split among the other two clusters. In both cases, the imprecise splitting prevents the inference of any cluster-specific invariants in both of the detected clusters. However by increasing `k` to 3 or 4, it is more likely to isolate these clusters. This means that by increasing `k`, we could get better results.

It is not always the case that natural clusters exist in program traces. In cases where they do not exist, cluster analysis has no value and will only lead to incorrect implications, if any are produced[1]. In such cases, increasing the number of clusters can only lead to a worsening of the result. This discussion makes the x-means algorithm (or any other algorithm that makes an attempt at correctly identifying the number of clusters) appealing. In our tests, the results of k-means were mostly comparable to the best values of the k-means and hierarchical algorithms.

5.1.2 Comparison With Static Analysis

We performed the same static verification experiments described above using splitting conditions obtained from static analysis of the program source (Chapter 4). In general, the static analysis procedure produces many fewer splitting conditions than cluster analysis. The recall is generally not as high because the fewer splitting conditions generally produced fewer conditional invariants than in cluster analysis[2]. The results are shown in figure 5-11.

5.1.3 Effect of Verification Goal

We performed another static verification experiment using the same Daikon output but with a different task in mind. The goal was to prove as many invariants as possible correct from Daikon's output. The main difference between this experiment and the previous static verification task is the fact that no invariants were removed from Daikon's output as long as they were correct and verifiable. The implication of this is that some invariants from Daikon's output sometimes required the *addition*

[1]Implications can be produced in this case if the clustering stage produces clusters in the data and invariants are detected in these clusters when, in fact, these are not natural clusters.

[2]In most cases, the splitting conditions from static analysis was usually a subset of the splitting conditions produced by cluster analysis

Program	Ver	Unver	Miss	Static Prec	Static Recall	Cluster Prec	Cluster Recall	Base Prec	Base Recall
QueueAr	57	16	11	0.78	0.84	0.69	0.90	1.00	0.76
StackAr	38	2	0	0.95	1.00	0.86	1.00	1.00	1.00
RatNum	32	8	1	0.80	0.97	0.63	0.95	0.93	0.96
Vector	117	21	2	0.85	0.98	0.63	0.99	0.98	0.98
DisjSets	41	1	2	0.96	0.96	0.98	0.95	1.00	1.00
RatPoly	50	12	3	0.81	0.94	0.88	0.99	0.88	0.99
GeoSegment	31	7	0	0.82	1.00	1.00	1.00	1.00	1.00
FixedSizeSet	22	0	0	1.00	1.00	1.00	1.00	1.00	0.88

Figure 5-11: Results of static verification with splitting conditions from static analysis. The clustering results are only shown for the x-means algorithm.

of one or more invariants to the set in order to be verifiable. Addition of more invariants leads to lower recall. In the first verification experiment, the invariants requiring the addition of extra invariants would have been removed if they were not required to prove the absence of runtime errors, and if removing these invariants led to less changes to Daikon's output than adding extra invariants[3].

Surprisingly, in most cases, the invariants from clustering did not require the addition of many new invariants in order to verify. The results are shown in figures 5-12 to 5-18. Precision is generally higher in the second experiment than the first experiment (because we tried to verify as may correct invariants as possible). However, the recall is worsened because there were more missing invariants.

QueueAr										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	42	0	13	1.00	0.76	—	—	—	—	—
2	79	12	9	0.87	0.90	63	6	10	0.91	0.86
3	69	18	11	0.79	0.86	64	7	10	0.90	0.86
4	87	13	9	0.87	0.96	65	7	10	0.90	0.87
5	97	21	9	0.82	0.92	99	19	8	0.84	0.93

X-means						
`max(k)`	`avg(k)`	Ver.	Unver.	Miss.	Prec.	Rec.
17	10.5	81	32	10	0.71	0.89

Figure 5-12: Results for `QueueAr` with splitting conditions from clustering. The goal of this experiment is to prove as many invariants from Daikon's output as possible correct. The first row ($k = 1$) shows the results from [NE02b] (ie. without implications from clustering.) For the x-means algorithm, `max(k)` refers to the maximum number of clusters that was observed across all program points. `avg(k)` refers to the average number of clusters that x-means found. For each row other than the base case (row 1), the difference in the number of invariants between that row and the base case is caused by the addition of implications.

[3]For example, if a single correct invariant in Daikon's output required the addition of three new invariants in order to verify, it would be removed from the goal set. However, if two or three invariants all required the addition of a single invariant for them to verify, we would add that invariant.

Stackar										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	25	0	0	1.00	1.00	—	—	—	—	—
2	37	2	2	0.95	0.95	41	6	2	0.87	0.95
3	41	6	2	0.87	0.95	42	6	2	0.88	0.95
4	38	2	2	0.95	0.95	42	6	2	0.88	0.95
5	41	6	2	0.87	0.95	42	6	2	0.88	0.95

X-means						
`max(k)`	`avg(k)`	Ver.	Unver.	Miss.	Prec.	Rec.
10	5.9	46	5	1	0.90	0.98

Figure 5-13: Results for `Stackar` with clustering.

Ratnum										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	25	2	1	0.93	0.96	—	—	—	—	—
2	55	30	3	0.65	0.95	40	22	2	0.65	0.95
3	50	28	3	0.64	0.94	38	45	3	0.46	0.93
4	50	28	3	0.64	0.94	47	41	3	0.53	0.94
5	58	50	4	0.54	0.94	45	44	4	0.51	0.92

X-means						
`max(k)`	`avg(k)`	Ver.	Unver.	Miss.	Prec.	Rec.
17	10.6	55	32	3	0.63	0.95

Figure 5-14: Results for `Ratnum` with clustering.

Vector										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	100	2	2	0.98	0.98	—	—	—	—	—
2	134	48	4	0.74	0.97	143	80	5	0.64	0.97
3	135	73	4	0.65	0.97	151	81	5	0.65	0.97
4	135	56	5	0.71	0.96	148	74	5	0.67	0.97
5	136	55	5	0.71	0.96	154	70	5	0.69	0.97

X-means						
max(k)	avg(k)	Ver.	Unver.	Miss.	Prec.	Rec.
16	7.3	135	75	4	0.64	0.97

Figure 5-15: Results for `Vector` with clustering.

5.2 Bug Detection

The bug detection experiment evaluates the ability of our clustering methodology to help programmers find errors in programs. It has been used in a previous study [DDLE02] to evaluate the effectiveness of cluster analysis and other predicate selection techniques at revealing bugs in programs. The rationale for using cluster analysis to find bugs in programs is as follows: Bugs in programs or procedures produce results that do not meet output constraints or expectations. Sometimes these faulty

StreetNumberSet										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	22	7	1	0.76	0.96	—	—	—	—	—
2	28	8	1	0.78	0.97	30	8	1	0.79	0.97
3	29	8	1	0.78	0.97	29	8	1	0.78	0.97
4	29	8	1	0.78	0.97	28	6	1	0.82	0.97
5	29	8	1	0.79	0.97	29	6	1	0.83	0.97

X-means						
max(k)	avg(k)	Ver.	Unver.	Miss.	Prec.	Rec.
1	2	26	12	1	0.68	0.96

Figure 5-16: Results for `StreetNumberSet` with clustering.

DisjSets										
	K-means					Hierarchical				
K	Ver.	Unver.	Miss.	Prec.	Rec.	Ver.	Unver.	Miss.	Prec.	Rec.
1	32	0	0	1.00	1.00	—	—	—	—	—
2	42	1	3	0.98	0.93	40	1	3	0.98	0.93
3	46	1	3	0.98	0.94	46	1	3	0.98	0.94
4	48	0	3	1.00	0.94	46	1	3	0.98	0.94
5	47	2	3	0.96	0.94	46	1	3	0.98	0.94

X-means						
max(k)	avg(k)	Ver.	Unver.	Miss.	Prec.	Rec.
17	8.2	42	0	3	1.00	0.93

Figure 5-17: Results for `DisjSets` with clustering.

	X-means					Base				
Program	Ver	Unver	Miss	Prec	Recall	Ver	Unver	Miss	Prec	Recall
RatPoly	51	17	1	0.75	0.98	70	10	1	0.88	0.99
GeoSegment	31	7	0	0.82	1.00	38	0	0	1.00	1.00
FixedSizeSet	22	0	0	1.00	1.00	16	0	0	1.00	1.00

Figure 5-18: Results of Static Verification for Remaining Programs. The results are only shown for the x-means algorithm.

executions are easily distinguishable from correct output. If the results of these faulty executions are different (i.e., different values for the variables at runtime) from the correct executions, then it may be possible to use cluster analysis to separate the good executions from the bad ones. Our goal is to capture the differences between erroneous and correct runs of the program and present them to a user, with the hope that these differences will lead the programmer to the underlying error. We speculate that if there are bugs in the program, then some of the dynamically detected implications may reveal the bugs. For this approach to succeed, the test suite must sufficiently exercise the program path containing the bug so that invariants over the set of faulty executions will be statistically justified by Daikon.

Cluster analysis cannot always isolate the bad executions from the good ones.

Failures do not necessarily cluster together, and buggy code does not necessarily produce output that is differentiable from correct output. Failures from the same procedure may appear in different clusters if they have different causes or if they differ with respect to the non-defective code they traverse. In addition, successful executions may occur in the same clusters as failures if they are similar with respect to the non-defective code they traverse.

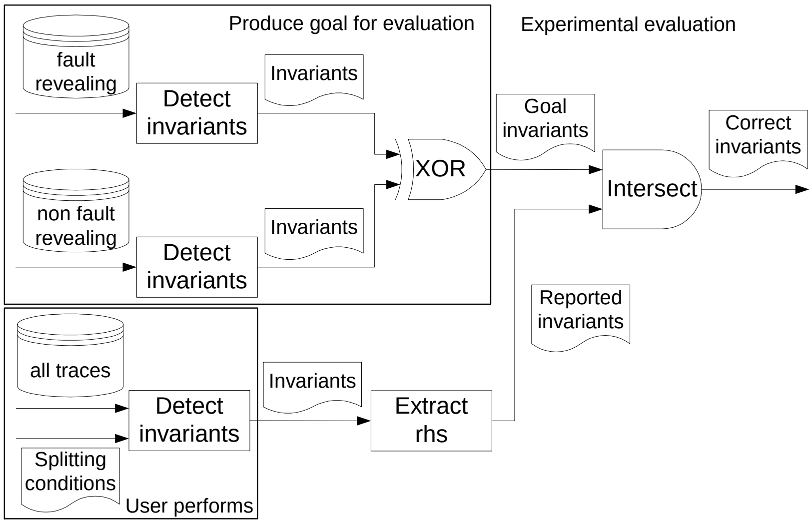

Figure 5-19: Overview of the experimental evaluation procedure for bug detection.

Our evaluation mechanism is described in Figure 5-19. We define a *fault-revealing* test case as a test case that executes a program path containing a bug, and whose output is therefore not correct. For each buggy program, we separate the test-cases into fault-revealing and non-fault-revealing sets, and detect invariants over these two sets separately. The **reported set** is the set of invariants reported by a run of Daikon augmented with splitting conditions from cluster analysis. The **goal set** is the set of invariants detected in only the fault-revealing set or the non-fault-revealing set, but not in both (this is the *xor* of these two invariant sets.) This is the set of invariants that will hopefully lead the programmer to the error. The **correct set** is the intersection of the goal set and the reported set. Given the goal, correct and reported sets, we compute precision and recall by

$$\text{precision} = \frac{\texttt{correct}}{\texttt{reported}}$$

and recall as

$$\text{recall} = \frac{\texttt{correct}}{\texttt{goal}}$$

The mechanism described above need not use cluster analysis. It can take in any splitter info file, that hopefully separates faulty executions from correct ones. We evaluate the effectiveness of cluster analysis for bug detection on four sets of buggy programs by comparing the results with those obtained by using splitter info files from static analysis and random selection and results from the base case (no splitter info file.)

Three of the programs come from the TopCoder programming competition website.[4]. These programs were submitted by contestants; the website publishes actual submissions and test cases after the match is complete. The three programs computed how football scores could be achieved, how elements could be distributed into bins, and how lines could be drawn to pass through certain points. We selected a total of 26 submissions that contained real errors made by contestants. The last set of programs were written by students in an undergraduate class at MIT (6.170 Laboratory in Sofware Engineering.) We selected 20 submissions that implemented a datatype for polynomials with rational coefficients. For all the programs we evaluated, the programmers did not know that their work would be used for our evaluation. The TopCoder test suites were more exhaustive, and better tested boundary cases because the contestants augmented them in an effort to disqualify their opponents.

Figure 5.2 shows preliminary results from the bug detection experiment. A precision of `x`% means that of the invariants reported by Daikon, `x`% of them were bug-revealing. The recall shows the percentage all fault-revealing invariants that were detected by our mechanism. These two measures give a measure of how effective each technique is at revealing faulty program behavior.

		Base		Cluster Analysis		Static Analysis		Random Selection	
Program	Ver.	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall
NFL	10	0	0	0.08	0.10	0.13	0.11	0.22	0.30
MultiSum	6	0	0	0.07	0.06	0	0	0.08	0.21
Contest	10	0	0	0.19	0.24	0.17	0.20	0.21	0.26
RatPoly	20	1.00	0.98	0.68	0.68	0.76	0.76	0.08	0.50

Figure 5-20: Detection of invariants induced by program errors. “Ver” is the number of versions of the program. “Prec.” is the precision of the precision of the reported invariants, the ratio of correct to reported. “Recall” is the recall of the reported invariants, the ratio of correct to goal.

The three splitting techniques had varying degrees of success in the bug detection experiment (as shown by the precision and recall reported in Figure 5.2). This suggests that they can be used as good complements for each other, or for other bug detection techniques. The random technique produced by far the largest set of invariants — an order of magnitude more than the other techniques. The many reported invariants permitted the random method to reveal more of the differences between erroneous and non-erroneous runs: more potential bugs were indicated by the output. Therefore the recall was generally higher than the other methods (except in the case of `RatPoly`.)

[4]TopCoder (`http://www.topcoder.com.`) is an online programming competition featuring programming talent from colleges in the U.S. The contests require competitors to quickly solutions to a set of somewhat simple algorithmic problems.

Chapter 6

Related Work

Dickinson, *et al* [DLP01] use cluster analysis as a filtering procedure in *observation based testing* to find failures in programs. Observation based testing [LPW00] is a software testing procedure that involves the following steps: taking an existing set of program inputs; executing an instrumented version of software under test on those inputs to produce *execution profiles* characterizing the executions; analyzing the resulting profiles, with automated help; and finally selecting and evaluating a subset of the original executions for conformance to requirements. Observation based testing seeks to reduce the manual effort required to evaluate executions for conformance to requirements by filtering out a subset of the original set of executions that is more likely to contain failures than is a randomly chosen subset. The technique is potentially applicable whenever it is desired to filter a large set of potential test cases in order to identify a promising subset to evaluate for conformance to requirements. The assumption is that executions that fail are likely to have unusual properties that may be reflected in execution profiles. An observation-based testing method is defined by specifying the form of execution profiling to be used and a filtering procedure for selecting the set of executions to be evaluated for conformance to requirements. Some profiling techniques are statement/basic-block profiling, branch profiling, path profiling, function-call profiling and various forms of data flow profiling (these are discussed and evaluated in [HRWY98].) The filtering procedure is designed to filter out these unusual profiles.

The focus of the research by Dickinson, *et al*, is the evaluation of filtering mechanisms that use cluster analysis. They cluster the exeuctions based on their profiles and then select one or more executions from each cluster or from certain clusters such as small ones. Their motivation for using cluster analysis to filter execution profiles is based on work by Podgurski, *et al* [PYM93] which suggests that software failures are often isolated in small clusters. Some of their filtering techniques include one-per-cluster sampling (selecting one execution at random per cluster) and adaptive sampling (initially selecting one execution per cluster, and including the remaining executions from any cluster where the initially selected execution was a failure.) They found that filtering procedures based on clustering are more effective than simple random sampling for identifying failures in program executions. Also, dissimilarity metrics that gave extra weight to unusual profile features were most effective.

Their work is similar to our research in a number of ways. Their use of cluster analysis to separate profiles of faulty executions from correct ones is similar to our bug detection approach, where we use cluster analysis to identify clusters of executions that reveal faulty behavior. As our execution profile, we use the set of values of all variables in scope at a program point. Both our techniques rely on the expectation that clusters of failure profiles will be distinguishable from clusters of normal profiles, and our results both report some degree of success in this regard.

Chapter 7

Conclusions

7.1 Conclusions

This thesis has proposed the use of cluster analysis to infer conditional properties in programs. The experimental data confirm the value of this method. There is a trade-off between increased recall (the addition of new useful invariants for a program verification task) and better precision (the absence of wrong invariants due to overfitting.) However, anecdotal evidence, confirmed by our experiments, tells us that the increase in recall by adding implications from clustering is worth the cost of decreased precision — wrong invariants can be easily identified and eliminated from Daikon's output. In addition, the precision can be increased by judiciously adding test cases to eliminate the wrong invariants.

In most cases, the invariants inferred by Daikon in the base case were nearly sufficient for the program verification task; for these the addition of implications did not have a significant effect on the recall. In the cases where the base case invariants were not sufficient, implications improved the recall.

The precision or correctness of Daikon's invariants, was invariably made worse by clustering. However this loss in precision can be mitigated by improving the test suite.

The results also show that increasing (or decreasing) the number of clusters does not have a deterministic effect on the precision and/or recall of our static verification task. More importantly, changing the number of clusters does not have a large effect on the precision or recall. On one hand, increasing the number of clusters increases the chances of isolating small natural clusters in the data, and therefore getting cluster-specific invariants. On the other hand the precision is worse because the overspecification problem is heightened. A good trade-off is to use the x-means algorithm, which produced results that were usually as good as those produced by k-means and hierarchical clustering. The k-means algorith, with `k` equal to 4 or 5 also worked well. In either case, recall is higher and precision is not negatively affected if the test suite is augmented. The output of Daikon gives valuable hints about where to augment the test suite, making the augmentation task easy.

We have shown that the invariants produced by cluster analysis could also help programs find bugs in programs. For a bug detection experiment, cluster analysis was able to detect a few invariants that revealed properties that we claim can lead a programmer to identify bugs in the programs evaluated.

Overall, we have seen that cluster analysis adds some correct and verifiable conditional invariants to Daikon's output. These invariants can be useful to programmers in various tasks.

7.2 Contributions

We have described a novel approach to inferring conditional properties in programs: detect natural clusters in the program and use invariants over those clusters as predicates for implications. To our knowledge, ours is the first system that uses this approach to dynamically detect conditional properties in programs. It is significant because inferring these conditional properties manually is very time consuming, and requires intimate knowledge of the program — a luxury many programmers and developers cannot afford.

Detection of conditional properties is important, as they give programmers a deeper understanding of how a program works. We have shown how the implications obtained from cluster analysis can help us improve the recall in a static verification task. These implications had previously been difficult to write by hand.

We have also described a novel approach to error-detection in programs that has shown promising results in our preliminary experiments. The results show that conditional properties from cluster analysis can reveal properties that can lead programmers to bugs in software programs.

Lastly, implications produced by cluster analysis can be used to find subdomains and ranges of program functions. One specific example is the case of bug detection, where the specific subdomains revealed are those that produce the errors.

7.3 Future Work

The clustering algorithms we used are not very sophisticated. For example, they do not deal adequately with noise (i.e., data that does not cluster naturally with any other element) and heavily favor spherical clusters. Research dealing with these issues include work by Banfield and Raftery [BR93] and Celeux and Govaert [CG95].) *Density based clustering* [EKSX96, AGGR98] is also another clustering method that deals with the issues of noise and cluster shape. We could use these techniques to improve our results. Section 3.

In Section 2.1, we mentioned that the Create-Implications algorithm could be improved. We are working on ideas to improve the algorithm to infer implications that occur in more than one, but not all the splits of the data.

We are also interested in investigating more carefully, the ability of cluster analysis to reveal bugs and find outliers in programs. The inability to visualizing the data limited our ability to see the effects of clustering on our data. Using some visualization technique (e.g., multidimensional scaling) would aid us get a better insight into the results.

Bibliography

[AGGR98] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of the ACM SIGMOD Conference on Management of Data (SIGMOD-98)*, pages 94–105, 1998.

[And73] Michael R. Anderberg. *Cluster analysis for applications*. Academic Press, New York-London, 1973. Probability and Mathematical Statistics, No. 19.

[BR93] J. D. Banfield and A. E. Raftery. Model-based gaussian and non-gaussian clustering. *Biometrics*, 43(3):803–821, September 1993.

[CG95] G. Celeux and G. Govaert. Gaussian parsimonious clustering models. *Pattern Recognition*, 28(5):781–793, 1995.

[DDLE02] Nii Dodoo, Alan Donovan, Lee Lin, and Michael D. Ernst. Selecting predicates for implications in program analysis, March 16, 2002.

[Det96] David L. Detlefs. An overview of the Extended Static Checking system. In *Proceedings of the First Workshop on Formal Methods in Software Practice*, pages 1–9, January 1996.

[DHS00] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley & Sons, 2000.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, December 18, 1998.

[DLP01] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE-01)*, pages 339–348, May 12–19, 2001.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):1–25, February 2001.

[EGK71] B. S. Everitt, A. J. Gourlay, and R. E. Kendall. An attempt at validation of traditional pcychiatric syndromes by cluster analysis. *British Journal of Psychiatry*, 119:399–412, 1971.

[EGKN99] Michael D. Ernst, William G. Griswold, Yoshio Kataoka, and David Notkin. Dynamically discovering pointer-based program invariants. Technical Report UW-CSE-99-11-02, University of Washington, Seattle, WA, November 16, 1999. Revised March 17, 2000.

[EKSX96] M. Ester, H.P. Kriegel, J. Sander, and X. Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*, volume 16, pages 226–231, 1996.

[Fis69] W. D. Fisher. *Clustering and Aggregation in Economics*. Johns Hopkins University Press, Baltimore, MD, 1969.

[GFR67] Paul E. Green, Ronald E. Frank, and Patrick J. Robinson. Cluster analysis in test market selection. *Management Science*, 13(8):387–400, 1967.

[GJ78] Michael R. Garey and David S. Johnson. *Computers and Intractability / A Guide to the Theory of NP-Completeness*. W.H. Freeman & Company, San Francisco, 1978.

[HRWY98] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. An empirical investigation of program spectra. In *Workshop on Program Analysis For Software Tools and Engineering*, pages 83–90, 1998.

[Hud82] H. C. Hudson, editor. *Classifying Social Data*. Jossey–Bass, San Francisco, 1982.

[Kin83] B. F. King. Market and industry factors in stock price behavior. *Journal of Business*, 13(2):72–79, 1982-83.

[KR90] Leonard Kaufman and Peter J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, March 1990.

[LPW00] David Leon, Andy Podgurski, and Lee J. White. Multivariate visualization in observation-based testing. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE-00)*, pages 116–125, 2000.

[MK83] D. L. Massart and L. Kaufman. *The Interpretation of Analytical Chemical Data By The Use of Cluster Analysis*. Wiley, New York, 1983.

[NE01] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, July 23, 2001.

[NE02a] Jeremy W. Nimmer and Michael D. Ernst. Automatic generation and checking of program specifications. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*, Rome, Italy, July 22–24, 2002.

[NE02b] Jeremy W. Nimmer and Michael D. Ernst. Invariant inference for static checking: An empirical evaluation. In *Foundations of Software Engineering*, Paris, France, November 2002.

[PLL99] J. M. Pena, J. A. Lozano, and P. Larranaga. Empirical comparison of four initialization methods for the K-means algorithm. *Pattern Recognition Letters*, 20:1027–1040, 1999.

[PM00] Dan Pelleg and Andrew Moore. Accelerating exact k-means with geometric reasoning. Technical Report CMU-CS-00-105, Carnegie Mellon University, 2000.

[PM01] Dan Pelleg and Andrew Moore. X-means: Extending K-means with efficient estimation of the number of clusters. Unpublished Manuscript, http://www.cs.cmu.edu/ dpelleg/download/xmeans.ps.gz, 2001.

[PYM93] Andy Podgurski, Charles Yang, and Wassim Masri. Partition Testing, Stratified Sampling, and Cluster Analysis. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 169–181, December 1993.

[RKS02] Orna Raz, Philip Koopman, and Mary Shaw. Semantic anomaly detection in online data sources. In *Proceedings of the 24th International Conference on Software Engeneering (ICSE-02)*, Orlando, Florida, May 22–24, 2002.

[Sal68] Gerard Salton. *Automatic Information Organization and Retrieval.* McGraw-Hill, 1968.

[SS73] P. H. A. Sneath and R. R. Sokal. *Numerical Taxonomy.* W. H. Freeman, San Francisco, 1973.

[TNH99] Jon Timmis, Mark Neal, and John Hunt. Data analysis using artificial immune systems, cluster analysis and Kohonen networks: Some comparisons. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, Tokyo, Japan, 1999.

[vR79] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, London, second edition, 1979.

[Wei99] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java.* Addison Wesley Longman, 1999.