Finding Patterns in Static Analysis Alerts

Improving Actionable Alert Ranking

by

Quinn Hanam

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2014

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Static analysis (SA) tools that find bugs by inferring programmer beliefs (e.g., FindBugs) are commonplace in today's software industry. While they find a large number of actual defects, they are often plagued by high rates of alerts that a developer would not act on (unactionable alerts) because they are incorrect, do not significantly affect program execution, etc. High rates of unactionable alerts decrease the utility of static analysis tools in practice.

We present a method for differentiating actionable and unactionable alerts by finding alerts with similar code patterns. To do so, we create a feature vector based on code characteristics at the site of each SA alert. With these feature vectors, we use machine learning techniques to build an actionable alert prediction model that is able to classify new SA alerts.

We evaluate our technique on three subject programs using the FindBugs static analysis tool and the Faultbench benchmark methodology. For a developer inspecting the top 5% of all alerts for three sample projects, our approach is able to identify 57 of 211 actionable alerts, which is 38 more than the FindBugs priority measure. Combined with previous actionable alert identification techniques, our method finds 75 actionable alerts in the top 5%, which is four more actionable alerts (a 6% improvement) than previous actionable alert identification techniques.

Acknowledgements

A big thank you to my amazing advisors Lin Tan and Reid Holmes for many things, especially all the useful, entertaining and motivating discussions. It has been a pleasure to work with you!

Thank you to Patrick Lam and Vijay Ganesh for challenging me and putting so much time and effort into helping me improve.

Finally, thank you to Molly Kravalis, my research group and my friends for making life fun the past two years.

Dedication

To my parents for always being there.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Static analysis (SA) tools are widely used to find bugs in software before they have a chance to manifest as run time faults. The most popular static analysis tools look through static code and infer a wide variety of bugs, security holes and bad programming practice [13]. Unlike more formal methods, this type of static analysis provides no guarantee that it has found all bugs that it checks for or that the bugs it does find are real [7].

The widespread adoption of SA for entire codebases by commercial software companies [10, 44] is evidence that SA is economically beneficial. However, SA suffers from high false positive rates [6]. Many of the alerts generated by SA suggest errors are present when in fact no errors exist. So despite its widespread use, high rates of false positives in SA still prevent many users from adopting these tools [24].

The terms 'true positive' and 'false positive' can be ambiguous in static analysis. If we define a true positive as any code that contains errors, the definition fails to cover warnings that identify bad practice or places where a bug could be easily be introduced. Alternatively, we could define a true positive as any warning identified by SA, because at the very least the warning has flagged a source of bad practice. This fails for situations where a programmer writes code that performs as intended and does not wish to modify. We therefore use the term *actionable alert* (AA) to define a SA alert that the programmer would act on to resolve and *unactionable alert* (UA) to define a SA alert that the programmer would not act on to resolve [18].

Because of the prevalence of unactionable alerts in SA warnings, considerable work has attempted to predict whether alerts are actionable or not; Heckman and Williams survey this literature [18]. Of particular importance in this research are the set of characteristics used to predict what SA alerts are actionable or unactionable. Heckman and Williams

refer to these characteristics as *alert characteristics* (AC) [18], a term which we adopt in this thesis.

Through our research investigating actionable and unactionable alert rates in SA tools, we make the observation that for any program, a number of actionable and unactionable alerts emerge that follow similar patterns. That is, developers frequently employ source code patterns that are unactionable but are repeatedly flagged by SA tools. Similarly, developers may also use a code pattern that is always actionable. We call these patterns *alert patterns*. These findings are supported by discussions with commercial SA tool developers. In their case, where a low false positive rate is important to get clients to adopt a tool, it is common to manually find patterns that result in many unactionable alerts and use these patterns to modify the SA tool to ignore the pattern. This is time consuming because the SA developer must manually re-program their tools for every unactionable alert.

In this thesis we propose a novel approach to predicting actionable and unactionable alerts by finding alert patterns. We use features of the source code at and near the source of the SA alert to extract ACs and find code patterns. We discuss our AC set in detail in section 3. Using these ACs along with a priori knowledge about which code patterns are actionable, we rank alerts according to the likelihood that they are actionable. This reflects a situation where a developer has limited time before the next release and needs to decide which alerts she should fix first.

We evaluate our technique by answering the following:

Research Question 1 *Do SA alert patterns exist?*

Research Question 2 *Can we use SA alert patterns to improve actionable alert ranking over previous techniques?*

We implement our technique as an Eclipse [41] plugin and evaluate it on FindBugs [43, 22] alerts generated for Tomcat6 [3], Apache Log4j [2] and Apache Commons [1]. We show that SA alert patterns do exist (RQ1). Our approach also effectively reorders the alerts provided to the developer. By examining only the top 5% of alerts (113/2,249) the developer is able to identify 57 of the actionable alerts while FindBugs only ranks 19 actionable alerts in the top 5%.

We show that SA alert patterns can improve actionable alert ranking (RQ2) by combining with previous actionable alert prediction techniques. Without using version histories, we find 75 actionable alerts, or four more (a 6% improvement) than previous actionable alert prediction techniques (which use version histories).

This thesis makes the following contributions:

1. It defines the notion of SA alert patterns.
2. It describes a technique for discovering SA alert patterns using code pattern ACs and provides an Eclipse plugin implementation of the technique. The code pattern ACs are not used in previous work.
3. It shows that our technique alone predicts 38 more AAs than the standard Findbugs ranking in the top 5% of alerts and four (6%) more than previous techniques when combined with ACs from previous techniques.

Chapter 2

Motivating Examples

To demonstrate the concept of alert patterns, we provide three concrete examples from running the FindBugs SA tool on revision 1497967 (June 2013) of the Apache Tomcat 6 webserver.

Motivating Example 1 Consider Figure 2.1, which shows code that defines the member variable `cDateFormat`. Running FindBugs with this code results in the following alert: *"STCAL: Sharing a single instance across thread boundaries without proper synchronization will result in erratic behaviour of the application"*. In this case, the alert is correct and this statement could potentially result in a concurrency error. However, in practice this `SimpleDateFormat` object is never written to beyond its construction. As long as this is the case, there is no need to provide synchronized access to the object.

The work it would take to provide synchronized access to the object, as well as the increased complexity of the resulting program outweigh the possibility of a concurrency error being introduced in the future. By our definition of unactionable alert, since the Tomcat6 developers have not taken action to resolve this potential issue (FindBugs is used by Tomcat6 developers [4]), the alert is likely unactionable.

This type of declaration of SimpleDateFormat is flagged as an alert seven times in

```
1 static final SimpleDateFormat cDateFormat
2   = new SimpleDateFormat ("yyyy-MM-dd");
```

Figure 2.1: Motivating Example 1 - Harmless shared instance variable, immutable? or unmodified after construction?

```
1 public int read(byte[]b,int offset,int len)
2 {
3   if (log.isTraceEnabled()) {
4     log.trace("read() " + b +" "
5       + (b==null ? 0: b.length)
6       + " " + offset + " " + len);
7   }
8   ...
```

Figure 2.2: Motivating Example 2 - Harmless implicit toString() call on an array: occurs in logging code.

```
1 try { socket.close (); }
2 catch (Exception ignore) {}
3 try { reader.close (); }
4 catch (Exception ignore) {}
```

Figure 2.3: Motivating Example 3 - Intentional ignoring of exceptions on calls to close().

the FindBugs anslysis of revision 1497967 of Tomcat 6. We can automatically identify these alerts as unactionable by looking for member variables of type `static final SimpleDateFormat` in statements that are flagged by FindBugs with alert type STCAL.

Motivating Example 2 Figure 2.2 shows (abbreviated) code from Tomcat 6 which reads a message in the form of a byte array. Running FindBugs with this code results in the following alert: *"USELESS_STRING: This code invokes toString on an array, which will generate a fairly useless result such as [C@16f0472."*. Indeed, the `toString` method is being called on byte array `b`, which emits a memory address. However, we believe that this behaviour is intentional, especially since the output of `b.toString()` appears in logging code, where it might actually be useful to disambiguate different byte arrays.

In fact, any call to `toString` on an array within `log.trace()` is likely to be an unactionable alert. We can automatically identify this unactionable alert pattern by looking for calls to `toString` on an array inside the method `log.trace()`. There are 29 occurrences of this unactionable alert pattern in Tomcat6 r1497967.

Motivating Example 3 Figure 2.3 shows code from Tomcat6 which closes `Socket` and `ObjectReader` objects. Running FindBugs on this code results in the following alert on lines 2 and 4: *"This method might ignore an exception."*

This is a case of an unactionable alert, as the program is performing exactly as the developer intends. Since both resources `socket` and `reader` are being closed, the program is clearly done using them. One can easily see that if either are `null` or there is an error while closing the resources, the program can ignore the exception and assume the connection is closed with only minor consequences if the connection fails to close (i.e. trying to determine what went wrong is not worth the developer's effort in this situation). We can automatically identify this unactionable alert pattern by finding calls to `Socket.close()` or `ObjectReader.close()` within the preceding `try` statement of the offending `catch` block.

Chapter 3

Detecting Alert Patterns

Automatically detecting alert patterns presents a number of challenges. Alerts may be tightly bound to their context. That is, two actionable or unactionable alerts may have major semantic and syntactic differences that inhibit their being linked.

Observe the example from Tomcat6 r418016 in Figure 3.1. FindBugs generates the same alert (NP_NONNULL_PARAM_VIOLATION) at line 6 of both pieces of code. This alert indicates that one of the parameters of the method call might be null, but it will be dereferenced inside the method being called. Both code segments perform a similar function: they call the method of an object in the `digester` stack. However, there are semantic and syntactic differences. The class is specified in the third parameter of `callMethod1` on line 7. Syntactically, in the code on the top from SetTopRule.java, a variable named `child` is passed while in the code on the bottom, the variable `parent` is given. Semantically the code on the top calls a method from the object at the top of the `digester` stack, while in the code on the bottom the method is called on the root object.

Furthermore, the defining characteristic for each alert pattern may be different for each alert type and there may even be multiple alert patterns within an alert type.

To account for semantic and syntactic differences, we use a machine learning approach to detect alert patterns. Given a set of items that need to be classified, machine learning algorithms take a set of features (ACs in our case) for each item (known as a feature vector) and predicts a class for each item. In our case, the machine learning algorithm predicts whether a SA alert is actionable or unactionable. The classification of the given items (called the test set) is based on prior knowledge of the class of items in another set of feature vectors (called the training set).

```
1 public void end() throws Exception {
2   Object child = digester.peek(0);
3   Object parent = digester.peek(1);
4   ...
5   // Call the specified method
6   IntrospectionUtils.callMethod1(child,
7     methodName, parent, paramType,
8     digester.getClassLoader());
```

```
1 public void end() throws Exception {
2   Object child = digester.peek(0);
3   Object parent = digester.root;
4   ...
5   // Call the specified method
6   IntrospectionUtils.callMethod1(parent,
7     methodName, child, paramType,
8     digester.getClassLoader());
```

Figure 3.1: An alert (NP_NONNULL_PARAM_VIOLATION) pattern with semantic and syntactic differences. The code on the top is from SetTopRule.java in Tomcat6 while the code on the bottom is from SetRootRule.java.

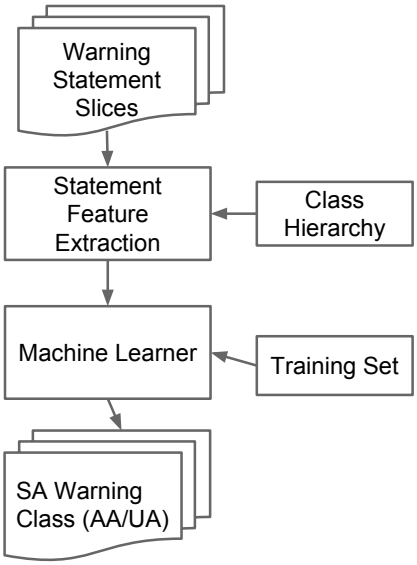

Figure 3.2: Activity diagram showing the method we use to classify SA alerts given slices for each alert.

Given a set of SA alerts, Figure 3.2 shows an overview of our method to detect actionable and unactionable alert patterns in those alerts:

1. We calculate a set of related statements by slicing the program at the site of the alert (discussed further in Section 3.1).
2. Using the statements from step 1 and the class hierarchy for the subject program, we extract a set of ACs.
3. A machine learning algorithm pre-computes a model with which to classify new alerts as actionable or unactionable. The model is trained using previously classified alerts. Alerts are classified by the developer or inferred by the version history.
4. Using the model from step 3, the machine learning algorithm ranks each alert, with those more likely to be actionable at the top.

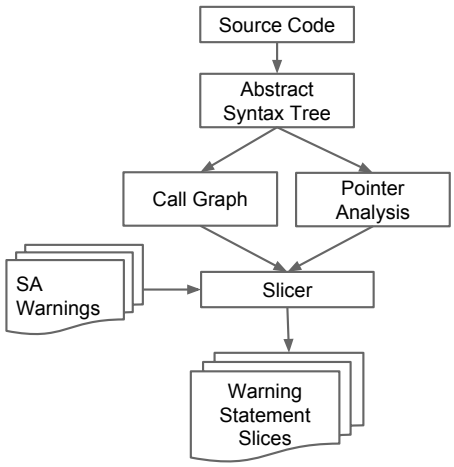

Figure 3.3: Activity diagram showing the method we use to generate SA alert slices.

3.1 Generating Alert Characteristics

Our alert pattern detection technique extracts ACs from the source code regions flagged by SA alerts. We begin by extracting statements that are potentially relevant to the alert. For each alert we reduce the number of statements to inspect by generating a backwards program slice [46] for each alert. A backwards program slice takes a statement in source code (called a seed statement) and determines which statements could have affected the outcome of the seed statement. We use the statements flagged with alerts by SA as seeds for program slice construction.

Figure 3.3 shows an overview of generating program slices using SA alerts as seed statements. The program's source code is parsed into an abstract syntax tree (AST), which is then used to build a call graph and pointer analysis. The call graph and pointer analysis, along with the SA alerts as seed statements are used to construct backwards slices for each alert.

Next, we extract characteristics from the statements selected by the slicer. The first column of Table 3.1 shows the list of statement types that we handle from the set of statements produced by program slicing (except for "Non-Seed Statements", which we explain later). We recognize five different statement types:

- **Call statements**
  (e.g., `System.out.println( "Hello World!" );`)

- **New object heap allocations**
  (e.g., `new String( "Hello World!" )`),
- **Binary operations** (e.g., `i + 1;`)
- **Field access** (e.g., `array.length`)
- **Catch statements** (e.g., `catch( Exception e )`).

The second column of Table 3.1 shows the list of ACs that we extract from each statement type. Below are descriptions of each characteristic.

- **Call Name** — The name of the method being called.
- **Call Class** — The name of the class containing the method being called.
- **Call Parameter Signature** - The signature for the method parameters.
- **Return Type** — The signature for the method's return type.
- **New Type** — The class of the object being created.
- **Concrete Type** — The class of the concrete type of the object being created.
- **Operator** — The operator for the binary operation.
- **Field Access Class** — The class containing the field being accessed.
- **Field Access Field** — The name of the field being accessed.
- **Catch** — Indicates that a catch statement is present.

For example, again consider the code in Figure 3.1 where a FindBugs warning exists on line 6 in both pieces of code. Line 6 is used as the seed statement for calculating a backwards program slice, which will produce the set of statements at lines $\{2, 3, 6\}$. In the code on the left, lines 2 and 3 both contain a method call (statement type "Call" in Table 3.1). The following ACs are extracted:

- Call Name: peek
- Call Class: org.apache.commons.digester.Digester
- Call Parameter Signature: (int)
- Return Type: java.lang.Object

In the code on the right, the ACs above are extracted at line 2, but at line 3 a "Field Access" is encountered with the following ACs extracted:

- Field Access Class: java.lang.Object
- Field Access Field: root

Line 6 also contains a call statement in both pieces of code from which we extract:

- Call Name: callMethod1
- Call Class: org.apache.tomcat.util.IntrospectionUtils
- Call Parameter Signature: (java.lang.Object; java.lang.String; java.lang.Object; java.lang.String; java.lang.ClassLoader)
- Return Type: java.lang.Object

Since the AC sets are similar (the ACs from lines 2 and 6 match) in both pieces of code, the machine learning algorithm can recognize a similar code pattern and use the classification of one alert to classify the other.

Some alerts flag fields, methods and classes, for which we cannot produce a program slice because we do not have a seed statement. In these cases, we extract ACs from the field, method or class definitions. These ACs are show in the "Non-Seed Statement" section at the bottom of Table 3.1.

3.2 Speeding Up Analysis

Generating full call graphs, points-to analysis and program slices can take quite some time and may be impractical for larger programs given limited computing resources. To speed up our analysis and limit memory consumption, we make the following optimizations:

Limiting Call Graph and Slice Size. We make the assumption that most patterns that define code clones can be found within or nearby the method that the alert occurs in. Using this assumption, we exclude all external classes (i.e. classes from libraries that are included in the project).

We create smaller slices by using context-sensitive thin slicing as described by Sridharan, Fink and Bodik [40]. Thin slices track only statements that have a direct effect on the seed statement. For example, if we use the method `println(data)` as our seed, whatever operations produce or modify the variable `data` are included in the thin slice, but not operations that produce or modify their containing objects. This significantly limits the size of the slice.

Table 3.1: Statement ACs

Seed Statements	
Statement Type	**Alert Characteristic**
Call	Call Name
	Call Class
	Call Parameter Signature
	Return Type
New	New Type
	New Concrete Type
Binary Operation	Operator
Field Access	Field Access Class
	Field Access Field
Catch	Catch
Non-Seed Statements	
Statement Type	**Alert Characteristic**
Field	Name
	Type
	Visibility
	Is Static/Final
Method	Visibility
	Return Type
	Is Static/Final/Abstract/Protected
Class	Visibility
	Is Abstract/Interface/Array Class

Alert ID	[Statement 1 Features]	[Statement 2 Features]	...	[Statement D Features]

Figure 3.4: Sample feature vector.

Limiting Alert Characteristic Vector Size. Because program slices can be very large, we limit the size of the feature vector by only using features from the five nearest statements prior to the seed in each slice (where five is the *distance from the alert*). This means we only look at the five statements in the slice that are adjacent to to the seed statement. The distance from the alert is set to five based on manual inspection of a number of alerts that we consider to have similar patterns.

We also only include statements that we believe are relevant to detecting code clones. Currently, this includes the statements in Table 3.1.

While these assumptions may not be correct in some cases, we feel they are necessary for the AC generation to run in a reasonable amount of time and limit the size of the feature vector for better machine learning performance. Our assumptions are supported by positive results for RQ1 discussed in Section 6. In the future, we would like to study the impact of these assumptions.

3.3 Classification

To classify new SA alerts, we need to keep track of actionable and unactionable alerts that have been classified, along with the AC vectors for each alert. This requires a training phase where the developer looks through a number of SA warnings and classifies them as actionable or unactionable, or the classes of prior alerts are inferred from version histories (as we do in section 5.1).

Over time, the developer will build a training set for the tool to differentiate actionable and unactionable alerts. Classifying warnings can be automated by detecting which alerts are closed from one SA to the next without any input from the developer. If there are already a number of alert patterns in the program when the tool is first run, the developer may classify a subset of the alerts and allow the tool to automatically classify the rest, thus reducing the developer's workload.

We use machine learning techniques to classify SA alerts. We test our AC set using multiple machine learning algorithms in Section 6.

Chapter 4

Method

We implement our AC extraction technique as an Eclipse plugin. For static analysis, we use the T.J. Watson Libraries for Analysis (WALA) [45] and the Eclipse JDT [12] library for AST generation.

To answer RQ1 and RQ2, we look at the following metrics: percent of actionable alerts found, precision, recall and F-measure.

The first metric we use measures how many actionable alerts a developer would see if she inspected the top N% of alerts in a ranked list. We express this as a percentage of the total number of AAs so that we have a fair comparison across test subjects (which may contain different numbers of AAs).

Given a set of ranked alerts R, a set of actionable alerts A (where $A \subseteq R$) and integer N where $0 \leq N \leq 100$, let $\%AA_N$ be the percent of actionable alerts found if we inspect the top $N\%$ of alerts in R . To get $\%AA_N$, we select the top N% of alerts in R and call this set R_N. We then extract all actionable alerts from R_N into a new set called R_{NA}. $\%AA_N$ is then $|R_{NA}|/|A| * 100$. For example, consider a situation where A contains 10 actionable alerts ($|A| = 10$) and R contains 200 alerts ($|R| = 200$). If N=10 then we inspect 20 alerts ($|R_{10}| = 20$). If there are five actionable alerts within R_{10} ($|R_{10A}| = 5$), then $\%AA_N = 5/10 * 100 = 50\%$. This formula is shown below.

$$\%AA_N = \frac{|R_{NA}|}{|A|} * 100$$

We also measure precision, recall and F-measure for both classes AA and UA. Precision is a measure of how accurate a classifier is. Recall measures the number of alerts of a given

class that a classifier is able to correctly classify. The F-measure is a weighted average of precision and recall.

We also calculate a weighted average of precision, recall and F-measure across both classes (AA and UA). For this metric, precision, recall and F-measure are weighed according to the number of alerts in each class and averaged. Given the precision, recall or F-measure for actionable ([P,R,F]A) or unactionable ([P,R,F]U) classes, the number of actionable alerts (AA) and the number of unactionable alerts (UA), the weighted average is:

$$\text{Weighted Avg} = \frac{\text{[P,R,F]A} * \text{AA} + \text{[P,R,F]U} * \text{UA}}{\text{AA} + \text{UA}}$$

4.1 Feature Vector Construction

We place our statement ACs in a feature vector according to which statement they occur in. Because of this, the order in which statements appear in the code matters. Figure 3.4 demonstrates how we construct our feature vector, where D is the distance from the alert as defined in section 3.2.

4.2 Classification

To classify SA alerts, we use the machine learning utility Weka [42]. We use three different classification algorithms to classify the feature vectors: decision tree (ADTree), naive Bayes and Bayesian network (BayesNet). The selection of these three classifiers is based on our experience classifying alerts from the FaultBench v0.1 [20, 16] benchmark. For all classifiers we use the default parameters.

Chapter 5

Evaluation

We evaluate our technique on the three subject programs listed in Table 5.1: Apache Tomcat6, Apache Commons Collections and Apache Logging for Java (Log4j). These subjects are selected because of their size (they are large enough to have many static analysis warnings), age (they have source code histories spanning multiple years before FindBugs came into widespread use) and the fact that they have Subversion (a version control system) repositories (using one single version control system for all subjects makes implementing our evaluation easier).

5.1 Ground Truth

We first need a method to accurately classify alerts as actionable or unactionable. To do this we use the FaultBench v0.3 [19] method proposed by Heckman and Williams [16]. This technique uses the source code history of a project to determine if alerts are actionable or unactionable. This process is described below:

Table 5.1: Subject Programs

Subject	**Start Revision**	**End Revision**	**Revisions**	**Revision Interval**	**Size (KLOC)**
Tomcat6	June 2006	February 2008	11	2 months	110–122
Commons	April 2001	July 2008	14	6 months	4–43
Log4j	December 2001	November 2007	11	6 months	12–19

1. Select a number of revisions across a subject project's history.
2. Run a static analysis tool (FindBugs) on each revision to generate a list of alerts for each revision.
3. Find alerts that are closed over the course of the project history:
   - An alert is opened in the first revision it appears.
   - An alert is closed in the first revision after the open revision where the alert is not present (except in the case where it is not present because the file containing it is deleted).
4. Alerts that are closed are classified as actionable, while alerts that are open following the last revision analysed are classified as unactionable.

We chose this methodology because by definition, an actionable alert is an alert that a developer resolves by modifying the program (at some point it will disappear from static analysis). If it is unactionable, the alert will never disappear. If an alert is removed because the file containing the alert is deleted, we consider the alert status as unknown and remove it from the list.

5.2 A Baseline for Comparison

To evaluate our technique, we need a baseline to compare to. We use two baselines in our evaluation: FindBugs priority ranking and machine learning based actionable alert ranking.

Baseline 1 For our first baseline we use the default FindBugs ranking. FindBugs assigns a priority measure (high, medium or low) to each alert [14]. High priority alerts should be more likely to manifest as failures than low priority alerts, so we assume that high priority alerts are more actionable than low priority alerts. Using this assumption, we sort alerts according to the priority measure (higher priority alerts get ranked higher) and randomize the order of alerts with the same priority. To eliminate any bias from randomization, we take the average of $\%AA_N$ across 100 runs.

Baseline 2 For our second baseline, we use our machine learning approach to alert ranking (Section 4.2), but exclusively use ACs from prior research. There are many papers that are dedicated to finding or ranking actionable alerts (e.g., [18]). We use the set of ACs used by Heckman and Williams [17] for predicting actionable alerts. We chose this

Table 5.2: Alert Characteristics

Group	**Alert Characteristic**
SA Tool	Warning Class
	Warning Type
	Project
	Package
	File
	Class
	Method
	Method Signature
	Field
	Field Signature
	Priority
	Total Alerts for Revision
Java NCSS	Classes in Package
	Functions in Package
	Package NCSS
	Functions in Class
	Class NCSS
	Function NCSS
	Function CCN

Table 5.3: Alert Characteristics (continued)

Group	**Alert Characteristic**
Subversion	Open Revision
	Prior Revision
	Highest Contributing Developer
	File Creation Revision
	File Last Modified Revision
	File Age
	Project Added Lines
	Project Deleted Lines
	Project Growth
	Project Total Modified Lines
	Package Added Lines
	Package Deleted Lines
	Package Growth
	Package Total Modified Lines
	Package Percent Modified Lines
	File Added Lines
	File Deleted Lines
	File Growth
	File Total Modified Lines
	File Percent Modified Lines

AC set because we feel it is the most comprehensive to date for techniques using machine learning.

The ACs we use for this baseline are listed in Tables 5.2 and 5.3. We omit some ACs used in [17] from this baseline. These ACs and the logic behind their omission are described below (with AC names as defined by Heckman and Williams):

1. **Number of alert modifications** - This AC requires information not available at the alert opening and therefore not available in a practical application.
2. **Total open alerts for revision** - In our evaluation programs, relatively few alerts are closed and this AC essentially encodes the revision number.
3. **Alert lifetime** - This AC indirectly encodes whether or not an alert is closed in future revisions and gives a classifier a trivial way to classify alerts as actionabe or unactionable.
4. **Staleness** - This AC requires information not available at the alert opening and therefore not available in a practical application.

We retrieve these ACs from the sources listed in the *Group* column of Tables 5.2 and 5.3. The *SA Tool* ACs are retrieved from our FindBugs analysis of each revision from Section 5.1. JavaNCSS [29] is run on each revision to retrieve source code metrics. Finally, we analyse the log files of the Subversion repositories for each subject and calculate source code history metrics.

Chapter 6

Results

Table 6.1 shows the FindBugs alerts produced from our FaultBench implementation. TA represents the total number of unique alerts generated across all revisions of the subject. The *AA* column lists the number of those alerts that are classified as actionable, the *UA* column shows the number of alerts classified as unactionable and *DA* are alerts that were closed because the file they were contained in was removed. We classify 2,249 alerts in total and use these alerts in our experiments. 252 of these are AA and 1997 are UA. 288 alerts are not classified (DA) and are not used in our experiments.

There is no guarantee that the FaultBench method has perfect accuracy. An alert may disappear from the alert history for reasons other than its root being fixed by the developer. For example, the method containing the alert triggering code may be deleted, or the developer may introduce a feature that causes the alert to disappear. To get an idea of how accurate our FaultBench implementation is, we manually inspect all actionable alerts from Apache Commons. We choose Apache Commons because it has a reasonable number of alerts (52) and since it is a general purpose library, the code may be easier to understand and reason about.

Table 6.1: FaultBench classified alerts showing actionable alerts (AA), unactionable alerts (UA) and unclassified alerts (DA)

Subject	**Total Alerts**	**AA**	**UA**	**DA**
Tomcat6	1971	178	1733	60
Commons	329	50	102	177
Log4j	237	24	162	51

Table 6.2: Oracle (Faultbench) Accuracy

Number of Alerts	Classification	Percent of Total
37	Verified	74%
9	Method Deleted	18%
2	File Moved	4 %
2	Fixed Other	4%

Table 6.3: Alerts after extracting statement ACs

Subject	Total Alerts	Unknown	Statement Level	Field Level	Method Level	Class Level	Error
Tomcat6	1971	60	494	712	233	73	399
Commons	329	177	51	32	35	13	21
Log4j	237	51	66	50	17	5	48

Table 6.2 shows the accuracy of the oracle for Apache Commons. 74% of alerts labelled as actionable by FaultBench disappeared from the alert history because they had their root cause fixed (i.e. they are actionable), 11% because the method containing the alert triggering code was deleted, 4% because the file containing the alert triggering code was moved and 4% because unrelated code was changed.

Table 6.3 shows a breakdown of the alerts after generating the alert slices and extracting ACs. Columns one and two (*Total Alerts* and *DA*) are taken from Table 6.1. *Statement Level* alerts are alerts which flag a line containing a statement which can be used as a seed statement for generating a program slice. Field, method and class level alerts flag fields, methods and classes and therefore cannot be used as seed statements for generating a program slice. For these we use characteristics of the field, method or class (e.g., name, visibility, type) shown in the bottom half of Table 3.1. The *Error* column indicates statement flagging alerts for which we were unable to produce a program slice (because of cases we do not yet handle or errors during static analysis). For example, alerts flagging statements in classes created at runtime are in this category because they do not have a statically-known name and cannot be identified through the class hierarchy.

To answer RQ1, we look at the the ability of statement ACs by themselves to predict AAs and UAs. Table 6.4 shows the classification results using only statement ACs. Columns two to four show the percent of actionable alerts in the top 10, 20 and 30 percent of all warnings. Columns five through seven show the precision (AP), recall (AR) and F-measure (AF) for actionable alerts. Columns eight through 10 show the precision (UP),

Table 6.4: Metrics from 10-fold cross validation using only statement ACs. $\%AA_N$, Precision (P), Recall (R) and F-Measure (F) are shown using three classifiers for each subject program.

	$\%AA_N$, $N =$			**Unactionable**			**Actionable**			**Weighted**		
	10	20	30	P	R	F	P	R	F	P	R	F
Tomcat6												
ADTree	0.36	0.42	0.52	0.96	1.00	0.96	1.00	0.31	0.47	0.93	0.93	0.91
Naive Bayes	0.40	0.49	0.61	0.93	0.93	0.93	0.38	0.41	0.39	0.88	0.87	0.87
BayesNet	0.33	0.51	0.60	0.93	0.92	0.93	0.38	0.43	0.41	0.88	0.87	0.88
Commons												
ADTree	0.13	0.29	0.49	0.75	0.73	0.75	0.53	0.58	0.55	0.69	0.68	0.68
Naive Bayes	0.24	0.49	0.64	0.60	0.47	0.60	0.45	0.84	0.59	0.71	0.60	0.60
BayesNet	0.18	0.42	0.58	0.80	0.81	0.80	0.62	0.58	0.60	0.73	0.73	0.73
Logging												
ADTree	0.31	0.46	0.46	0.95	1.00	0.95	0.00	0.00	0.00	0.82	0.91	0.86
Naive Bayes	0.23	0.46	0.54	0.59	0.43	0.59	0.10	0.62	0.17	0.84	0.45	0.55
BayesNet	0.15	0.38	0.54	0.89	0.87	0.89	0.11	0.15	0.13	0.83	0.80	0.82

recall (UR) and F-measure (UF) for unactionable alerts. Finally, columns 11-13 show the weighted precision (WP), recall (WR) and F-measure (WF).

We find that statement ACs perform significantly better than a random ordering and conclude that alert patterns do exist. If alert patterns exist, then for our first metric ($\%AA_N$) the classifier performance using statement ACs should perform better than a random ordering. Using a random ordering, we would expect our $\%AA_N$ metric to evaluate to 0.1 for N=10%, 0.2 for N=20% and 0.3 for N=30%. Table 6.4 shows a significant improvement over random for all classifiers. Take Tomcat6 and the ADTree classifier. For N=10%, ADTree discovers three times as many actionable alerts as would be expected for a random sorting while for N=20% that number is still over twice as many. Since alert patterns exist, we can use them to enhance alert ranking and provide developers with a better indication of which alerts they should investigate first.

The results in Table 6.4 also show the precision and recall for the classifiers using only statement ACs. The average weighted precision (across all projects and classifiers) is 0.81, the average weighted recall is 0.76 and the average weighted F-measure is 0.77. We find

that while precision and recall for predicting actionable alerts is low (possibly because of the low ratio of actionable to unactionable alerts), we can effectively use the classifier's probability distribution to rank the alerts.

For ADTree, the recall for Tomcat6 and Logging is 100%. Because there is a high number of UAs compared to AAs, it is effective for the classifier to classify most alerts as UA (resulting in a low recall for AAs). We might tune the precision and recall by setting a lower threshold on the probability distribution (e.g., instead of using a 50% confidence threshold, we say the classifier only needs to be 40% confident to classify a warning as actionable and 60% confident to classify a warning as unactionable). For RQ2, we rank alerts by the probability distribution.

We answer RQ2 in two parts: First we compare our method to Baseline 1 (FindBugs priority ranking). Second, we compare our method to Baseline 2 (machine learning based alert ranking).

Using only statement ACs (those from Table 3.1), our method discovers 38 more AAs than Baseline 1 in the top 5% of all alerts across our three subject programs. Figures A.10, A.8 and A.11 shows the ranking results for our three subject programs using the ADTree classifier. The y-axis shows the $\%AA_N$ described in section 4, while the x-axis shows the value of N (% of warnings inspected). Our technique using only statement ACs is labelled *Statement* while FindBugs priority ranking is labelled *Baseline 1*. As an example, observe the graph for Tomcat6. When x=5, statement ACs discover 33% (or 51/153) of all AAs while FindBugs priority ranking discovers 10% (or 15/153) of all AAs. Across all three subject programs, statement ACs discover 57 AAs in the top 5% of alerts and Baseline 1 discovers 19. This result shows that our technique by itself out-performs FindBugs priority ranking and that it could be an effective tool to enhance alert ranking.

Using statement ACs combined with SATool (FindBugs) ACs and JavaNCSS ACs, our technique discovers four more AAs than Baseline 2 in the top 5% of all alerts across our three subject programs. Figures A.10, A.8 and A.11 again show the results of our evaluation. Our combined method is labelled *SATool + JavaNCSS + Statement*. The machine learning results using SATool, JavaNCSS and Subversion ACs are labelled *Baseline 2*. In all cases, our combined method performs better than or equal to Baseline 2 for the top 5% and 10% of alerts. Across all three subject programs, our method discovers 36% (75/211) of AAs in the top 5% of alerts and Baseline 2 discovers 34% (71/211), which is a 6% improvement. This result shows that our technique out-performs prior methods. Adding statement ACs to alert prioritization methods may be a useful tool to help developers decide which alerts need to be resolved first.

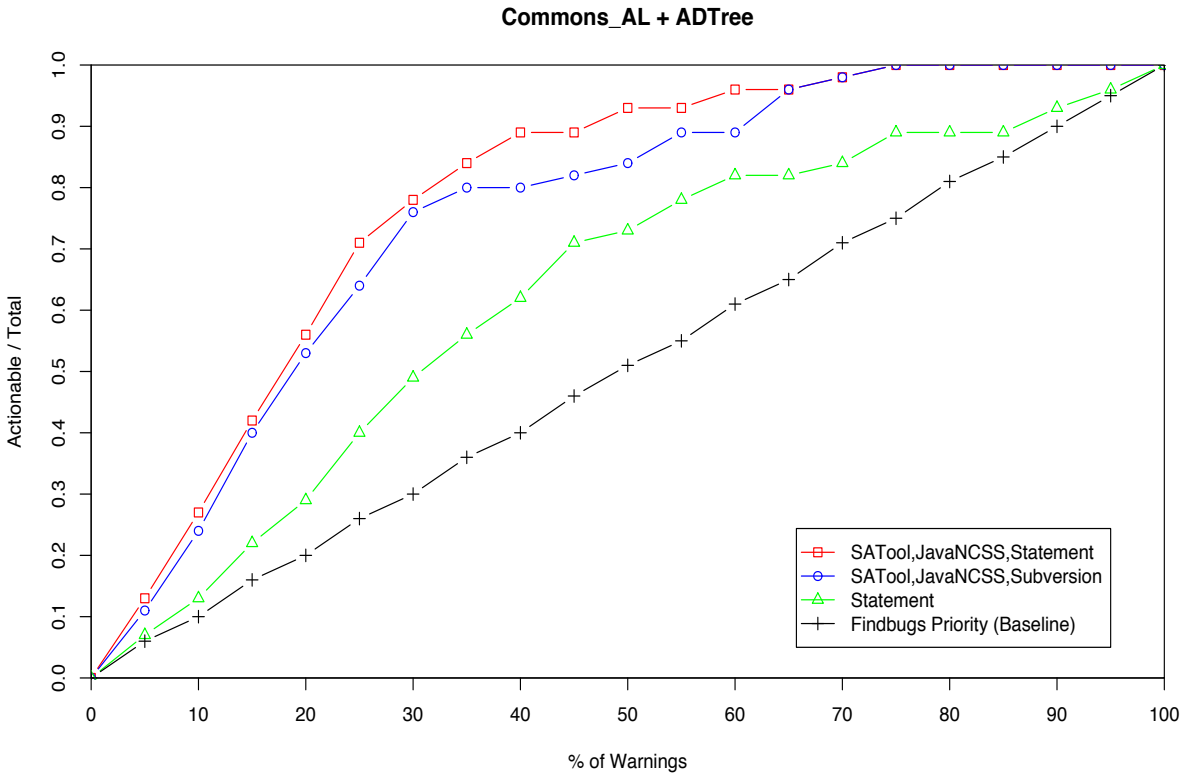

Figure 6.1: Commons decision tree results showing the percent of actionable alerts found within the first n% of warnings.

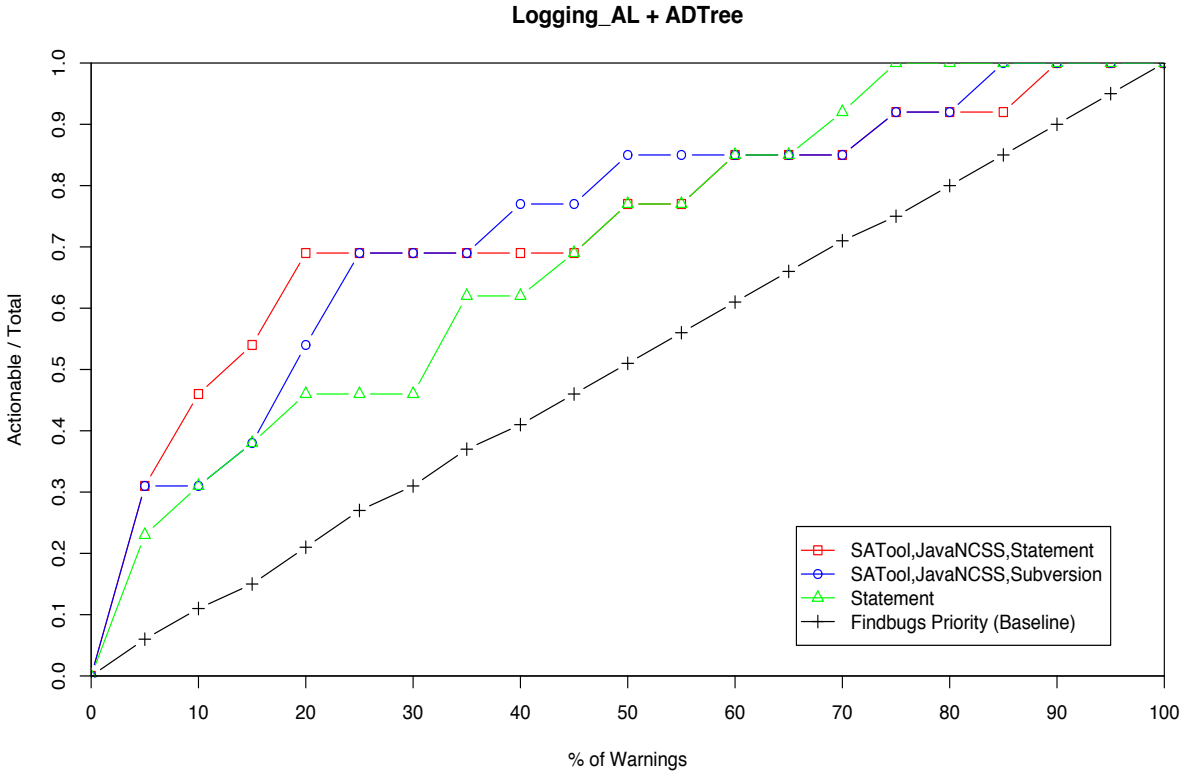

Figure 6.2: Logging decision tree results showing the percent of actionable alerts found within the first n% of warnings.

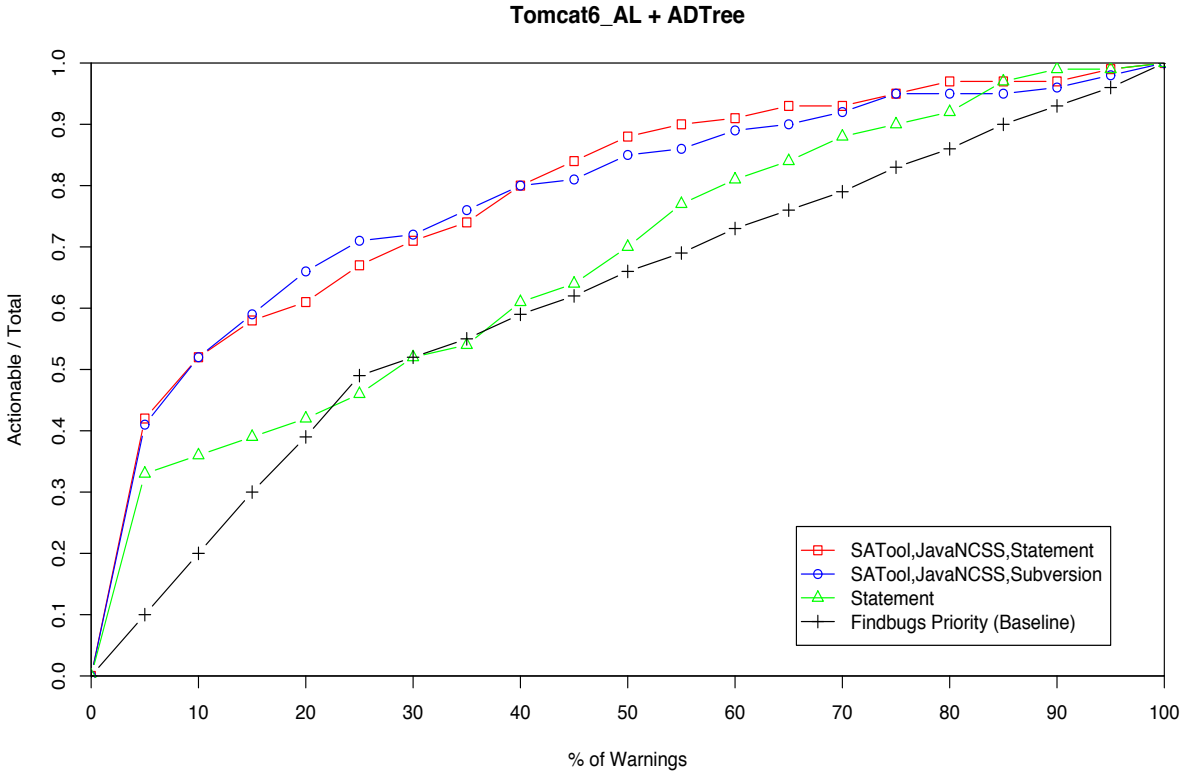

Figure 6.3: Tomcat6 decision tree results showing the percent of actionable alerts found within the first n% of warnings.

Chapter 7

Related Work

7.1 Actionable Alert Prediction.

An abbreviated version of this work was originally published in MSR 2014 [15].

Heckman and Williams conduct a comprehensive literature review on actionable alert identification techniques [18]. They identify 18 prior papers that provide methods of predicting actionable alerts and group what attributes were used to classify warnings as actionable or unactionable. These methods include using alert characteristics, code characteristics, source code repository metrics, bug database metrics and dynamic analysis metrics to identify actionable alerts. In this thesis, we use attributes that would be classified as code characteristics in the literature review. However, none of the research identified in the literature review identifies similar code patterns and none use the code characteristics we use in this thesis. Two closely related papers from the literature review are discussed below.

Ruthruff, Penix and Morgenthaler use metrics and machine learning to predict actionable FindBugs alerts in Google's code base [39]. They use 33 metrics including information from the warnings themselves (warning category and warning bug patterns). The bug patterns reported by FindBugs contain attributes most similar to those discussed in this thesis. However, the bug pattern information is limited (especially for more trivial checkers) [22] and it is unclear how these attributes are used in their study. In our study, we could theoretically modify the FindBugs checkers themselves to gather information, as FindBugs checkers can make use of control flow and data flow graphs [22]. However, not all checkers use this depth of analysis and program slicing is not directly supported.

Heckman and Williams also use alert characteristics and machine learning to predict actionable FindBugs alerts [17]. They evaluate 51 alert characteristics including those discussed in Section 5.2. Using the FaultBench benchmark [16], they evaluate 51 alert characteristics and 15 machine learning algorithms. This is one of the most comprehensive actionable alert prediction studies to date and achieves very good precision and recall (83-99%) using the FaultBench benchmark.

Our technique differs from the two above in that we include alert characteristics derived from static analysis. Since our technique uses only metrics found within the source code, our technique might be more practical and require less tooling. Our technique might also be better at predicting more problematic unactionable alerts that occur frequently because of a certain developer's style, while still leaving important actionable alerts.

Bodden, Lam and Hendren use static analysis to deduce run-time properties of programs [8]. They use decision trees with code characteristics as features to eliminate false positives. The alert characteristics used in this thesis are different from ours and are used to filter results from a much more specific type of static analysis.

7.2 Code Clone Detection.

Detecting code clones is a well studied field that involves detecting source code that has been copied from one part of a program to another, with possible minor modifications. It relates to this thesis because we are detecting a special kind of code clone: alert patterns.

Roy and Cordy [38] provide a summary of code clone detection techniques as well as a study of the situations in which these techniques would work. We chose a unique detection method that is similar to feature-based code clone detection methods discussed in the summary.

7.3 Suppressing Known False Positives.

In a study by Chimdyalwar and Kumar [9], SA alerts that are marked as false positives by users are suppressed from future SA runs by removing warnings from code regions that were not affected by the latest set of changes. Our technique is complementary to this. We automatically label alerts as unactionable or actionable in order to develop a model to rank alerts.

Chapter 8

Future Work

8.1 Method Improvements

In this paper we use WALA to perform an inter-procedural backwards data dependency analysis and extract the five nearest statements from the seed statement. We found WALA's inter-procedural analysis to be time and resource expensive. WALA requires a class hierarchy, data bindings, points-to analysis and call graph in order to run its dependency analysis. Although WALA does allow the exclusion of some components, we found this difficult to work with. Because of this, WALA might not be suitable for larger evaluations or mining applications.

We are currently developing a lightweight intra-procedural analysis that only inspects the method containing the alert. We believe this analysis will scale to large evaluations and mining applications. Using this analysis, we plan on evaluating our technique on more subject programs and revisions, comparing the run times of retrieving statement ACs to that of other (e.g., version history) ACs, investigating what statement ACs rank alerts the best and mining alert patterns on a large scale (e.g., to evaluate our technique's inter-project potential).

Another improvement we might consider is to adopt a bag of words approach. In this case we collect words from our current feature set (e.g., statement types, method names and field names) and use them as features. A feature's value is then the number of occurrences of that word in the statements we inspect. By doing this, we get two potential benefits:

1. The order of the statements does not matter (as it does with our current method). This is true because the bag of words approach counts the number of occurrences

of words in the statement set rather than using those words as values of ordered statement features.

2. We can use natural language processing to extract words or tokens from method and field names. For example, if we have a call to a method named `closeUserFile( )`, it might be useful to extract the words `close`, `user` and `file` from the method name. The fact that the method contains the word `close` might be more meaningful than the fact that the method contains the string `closeUserFile` (e.g.,Figure 2.3).

We have done some preliminary evaluations using a bag of words approach by pivoting the statement feature vector from Section 6. These results are displayed in Appendix A.4.

8.2 Evaluation Improvements

The method we use to evaluate the effectiveness of actionable alert ranking techniques can be improved. In our current evaluation we randomly order all alerts (generated across multiple revisions) and use cross validation to evaluate the classifiers. In a real world application, the developer performs changes in a temporal order and alerts appear and disappear following that order. To most accurately mimic a real world application, we should therefore:

1. Extract alerts at a revision level granularity. Ideally, the static analysis tool is run on each source control revision over some time period. This ensures that no actionable alerts are missed (i.e. alerts that are opened and closed in the set of uninspected revisions between two revisions that are inspected).

2. Train the classifiers on alerts that occur on or before the revision we wish to classify alerts for. For example, take a developer performing a static analysis task on revision 3 of some software. He has classified the results for revisions 1 and 2, which make up the training set. Because revisions > 3 do not exist yet, they cannot be used to classify revision 3. Since we can not use alerts that will occur in the future in practice, we should not in our evaluation.

These changes to the evaluation would demonstrate how effective our technique is as a developer aid. Ideally, when a developer classifies a new alert, our technique can then correctly classify all future alerts with a similar pattern. We can then measure the number of unactionable alerts which the developer does not have to view.

8.3 Inspecting All Alerts

An often overlooked part of static analysis is the benefit of fixing bad practice alerts as well as the time spent reviewing code while investigating alerts. A case might be made that developers should fix or investigate all static analysis alerts, since this may lead to improved code (fewer post-release defects). In fact, Nagappan and Ball demonstrate how Microsoft uses the number of static analysis alerts to predict how many defects will be found through testing [33]. This might suggest that if developers fix all static analysis alerts, the code will have fewer defects that aren't detected directly by static analysis.

Performing this research would be a difficult task. Two ways we might do this are through a user study or by mining software repositories.

A user study would have a number of challenges. Real world software systems are large and evolve over years. The effects of repairing all static analysis alerts might not manifest until a few months or even years after the practice starts. A user study in a lab environment therefore might not be suitable because subjects are generally available for a very short period of time. Real world software system are also complex. Having users modify a small program might not reflect a real world setting. Still, a user study operating on a small program might give us insights into whether or not repairing all alerts improves code quality.

Mining software repositories might give us insights into a potential relationship between the number of alerts and the number of defects. However, external factors are difficult to control. For example, every software system is built and maintained by different developers for different purposes. The results for a web browser might not generalize to a web server. Other factors that would need to be controlled include commercial vs. open source software, software age and position in the software lifecycle.

8.4 Tool Improvements

Reducing the number of unactionable alerts is only one part of a bigger challenge of getting the software community to adopt static analysis tools. Johnson et. al. [24] conduct a user study to investigate why developers do not use static analysis tools. One of their findings is that developers want information about potential bugs inside the editor with the code it is flagging. Commercial tools address this problem somewhat by providing web based code browsers where a user can trace through the events that led to the alert with in-code annotations. Modern IDEs are not yet equipped to support this functionality. It may be

worth studying this more to build a case for (or direct the implementation of) better IDE support for bug understanding.

Chapter 9

Conclusions

In this thesis we present alert patterns: similar patterns of code that are frequently flagged by SA alerts that as a group may or may not result in remedial action by a developer. We introduce a technique for finding alert patterns using statement ACs.

We show that SA alert patterns do exist by using our technique, which is able to rank 57/211 actionable alerts in the top 5% (113) of all warnings, while the standard FindBugs ranking only finds 19.

We show that SA alert patterns can improve actionable alert ranking. Our technique combined with previous work finds 75/211 actionable alerts, or four more (a 6% improvement) than previous actionable alert prediction techniques. By tuning our method and incorporating cross-project data, we believe we can significantly improve our method for practical use.

APPENDICES

Appendix A

Full Classification Results

A.1 All Feature Combinations

To find the best combination of features, we produce graphs showing all feature-group combinations (using the SATool, JavaNCSS, Subversion and Statement feature groups from Tables 5.2 and 5.3). These graphs show the results for all feature-group combinations for values up to N=20%.

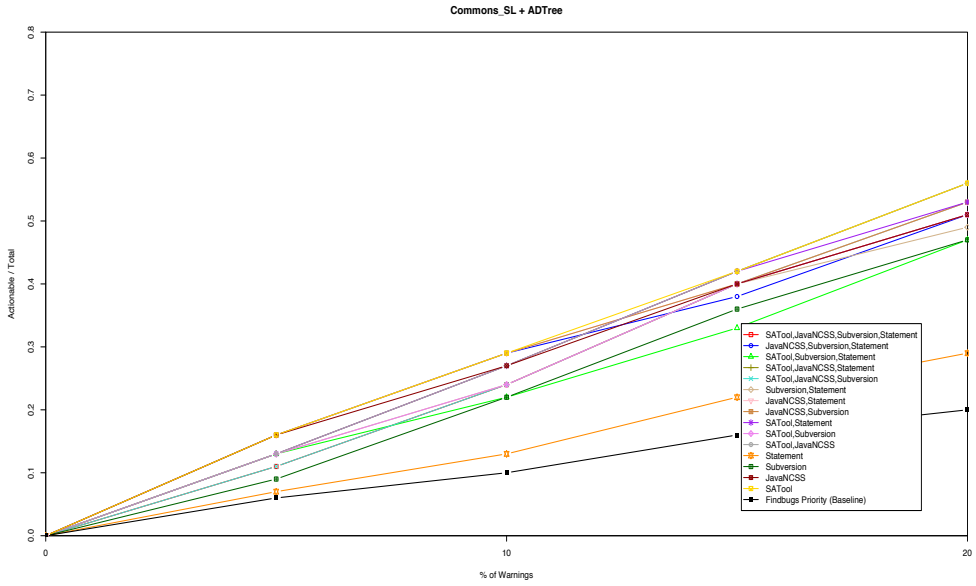

Figure A.1: Commons decision tree results showing the percent of actionable alerts found within the first n% of warnings.

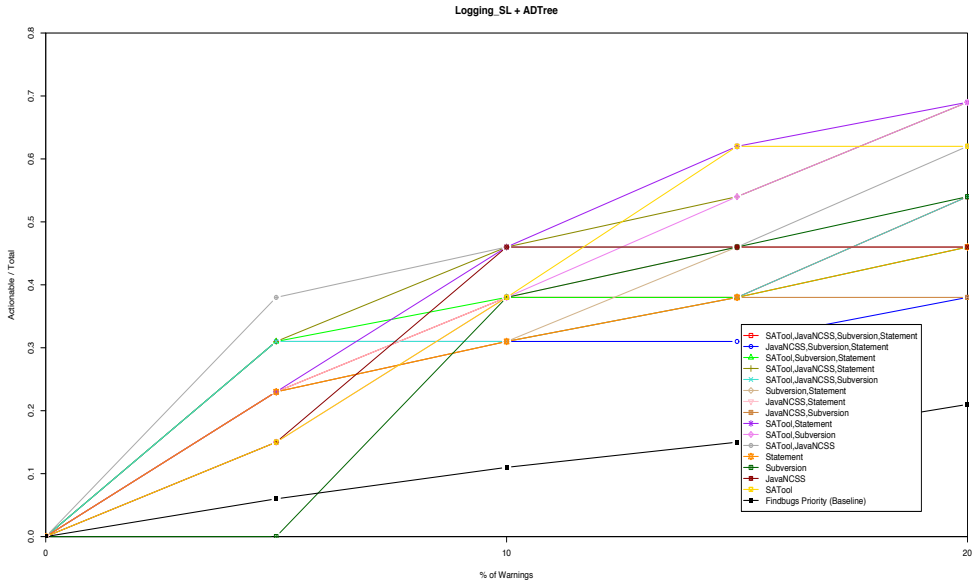

Figure A.2: Logging decision tree results showing the percent of actionable alerts found within the first n% of warnings.

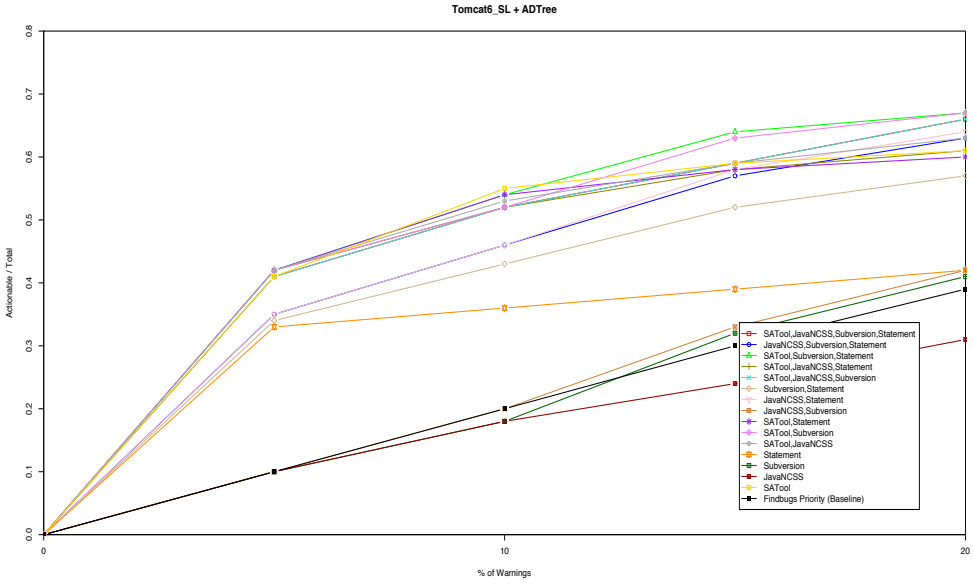

Figure A.3: Tomcat6 decision tree results showing the percent of actionable alerts found within the first n% of warnings.

A.2 Seed Statement Results

Graphs of classification results for alert types flagging seed statement (i.e. the alerts do not flag fields, methods or classes).

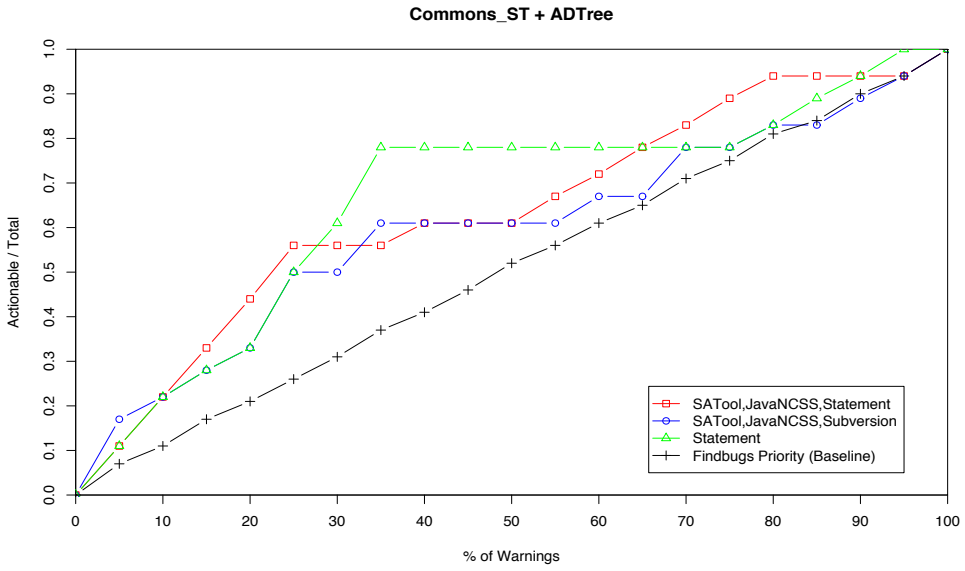

Figure A.4: Commons decision tree results showing the percent of actionable alerts found within the first n% of warnings.

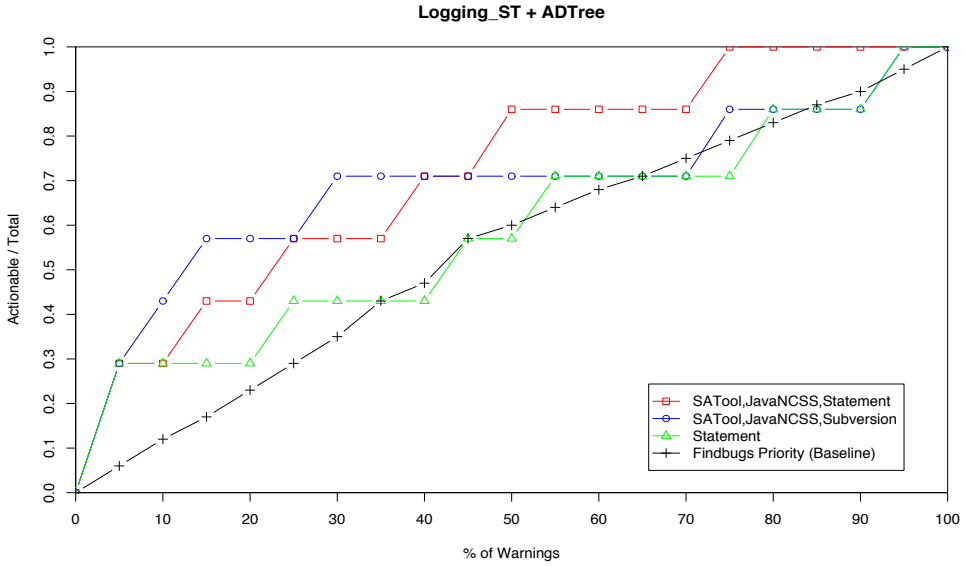

Figure A.5: Logging decision tree results showing the percent of actionable alerts found within the first n% of warnings.

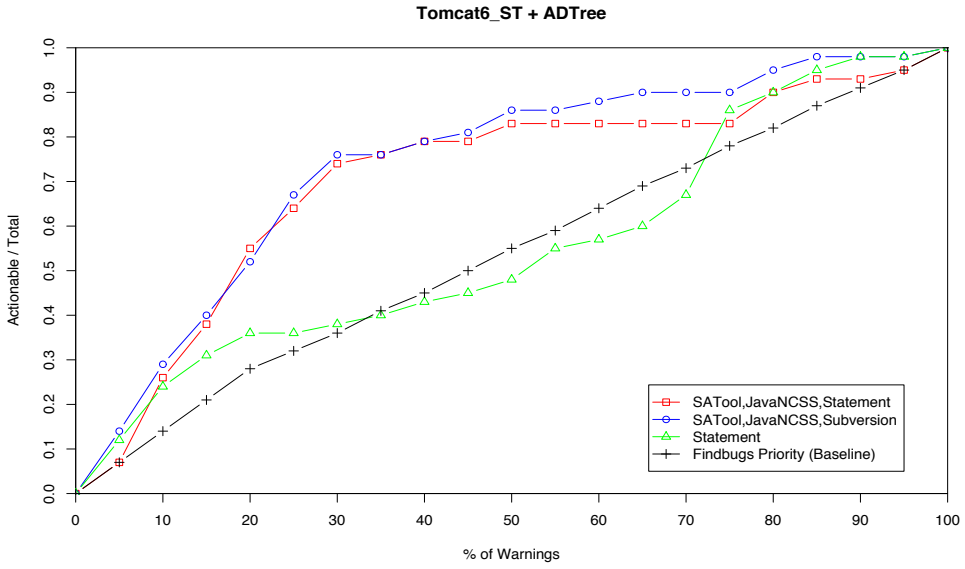

Figure A.6: Tomcat6 decision tree results showing the percent of actionable alerts found within the first n% of warnings.

A.3 Non-seed Statement Feature Graphs

Graphs of classification results for alert types flagging non-seed statements (i.e. the alerts flag fields, methods or classes).

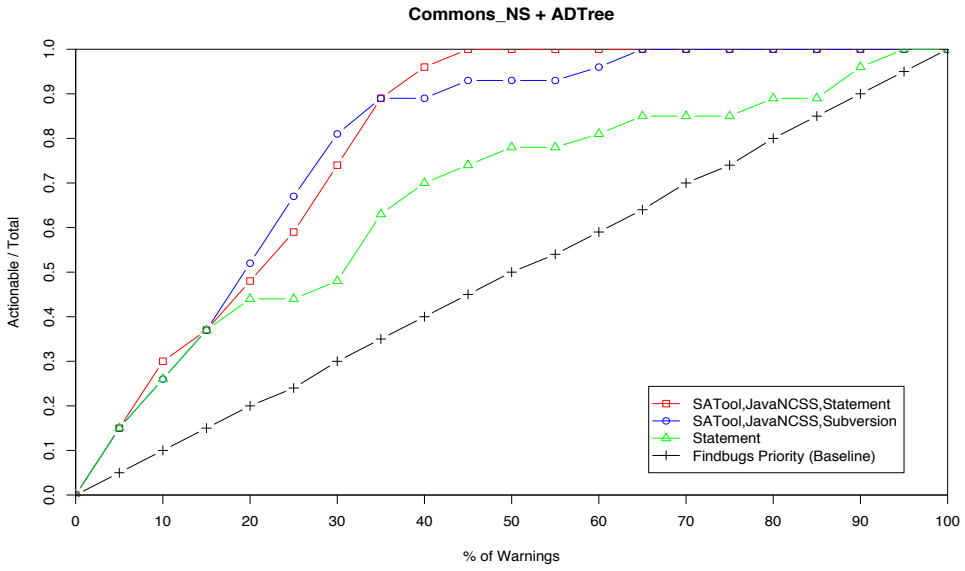

Figure A.7: Commons decision tree results showing the percent of actionable alerts found within the first n% of warnings.

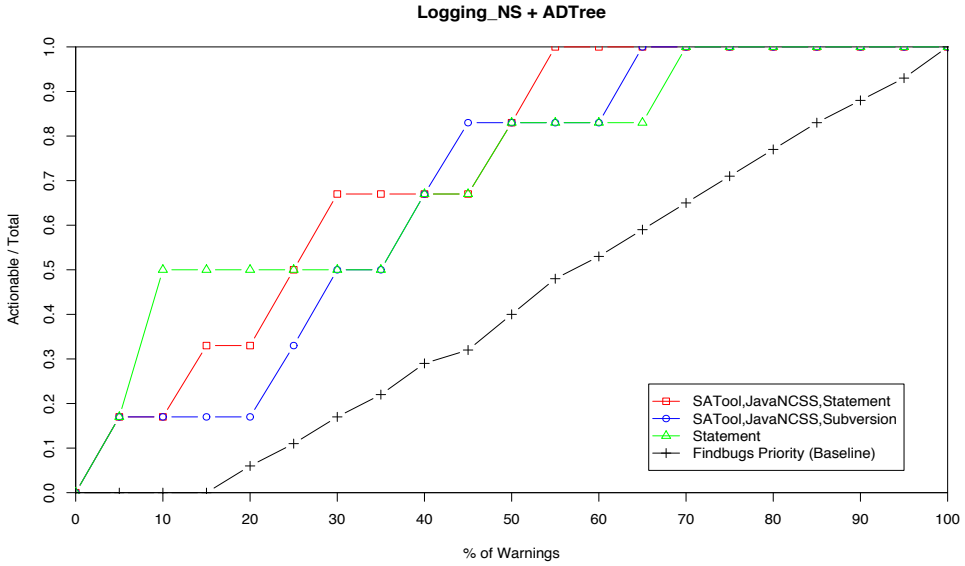

Figure A.8: Logging decision tree results showing the percent of actionable alerts found within the first n% of warnings.

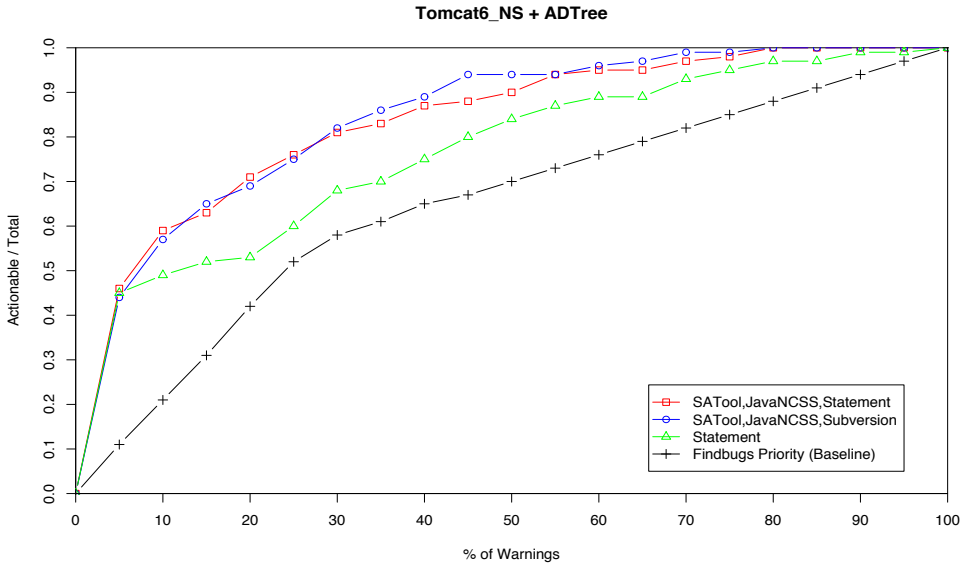

Figure A.9: Tomcat6 decision tree results showing the percent of actionable alerts found within the first n% of warnings.

A.4 Bag of Words Approach

Graphs of classification results using a bag of words approach.

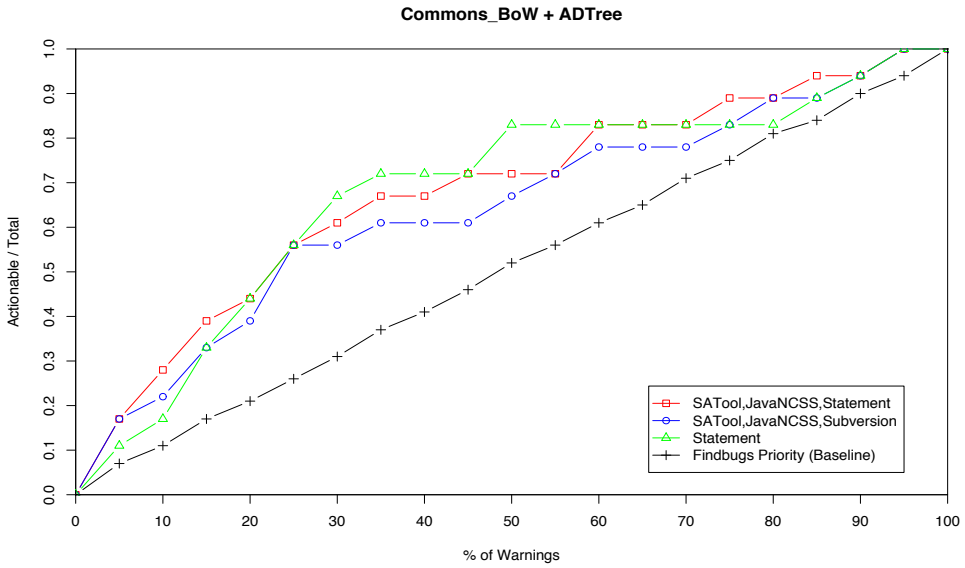

Figure A.10: Commons decision tree results showing the percent of actionable alerts found within the first n% of warnings.

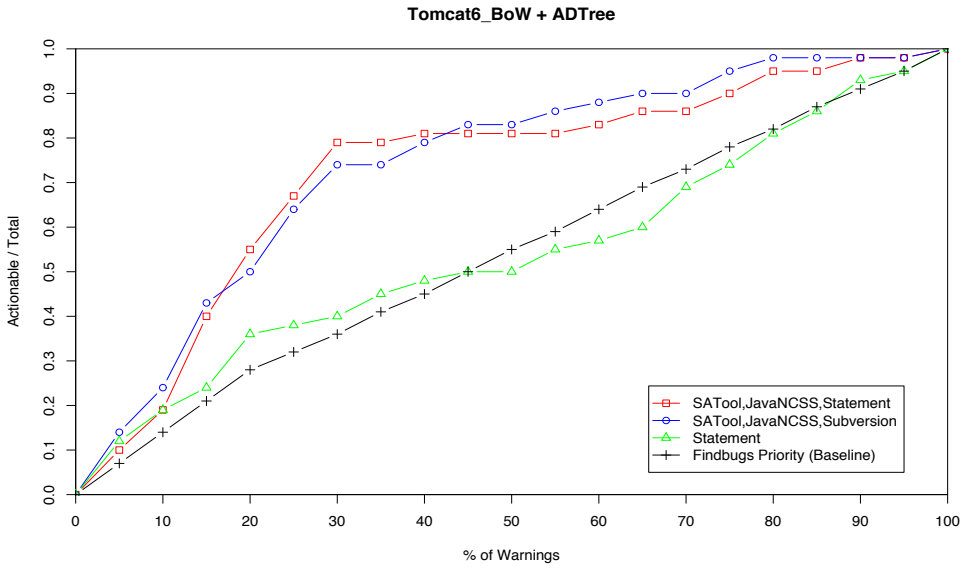

Figure A.11: Tomcat6 decision tree results showing the percent of actionable alerts found within the first n% of warnings.

Appendix B

UML Diagrams

B.1 Process Sequence Diagram

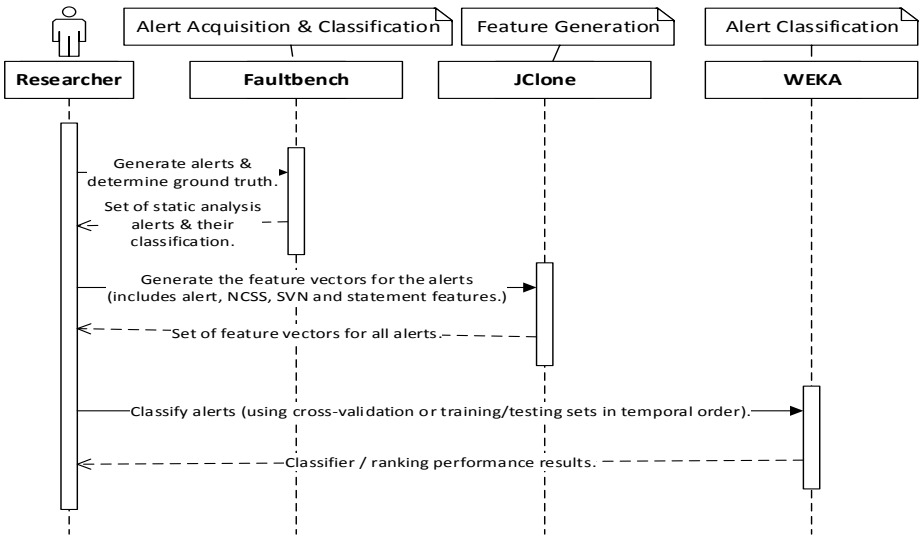

Figure B.1: A sequence diagram showing the interaction between the researcher and software components in order to investigate the performance of the feature set.

B.2 Feature Generation Sequence Diagrams

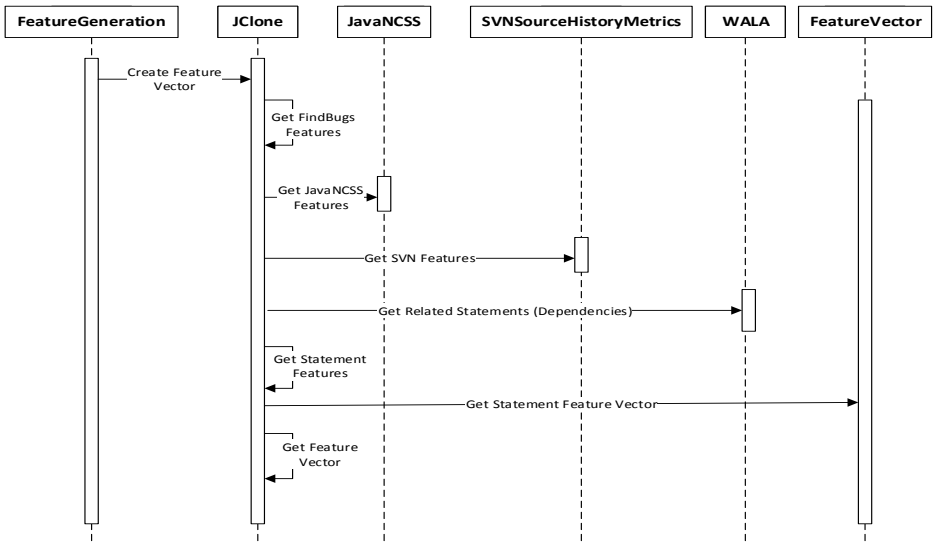

Figure B.2: A sequence diagram showing the interaction between classes in the JClone feature generation software component that is used for this work.

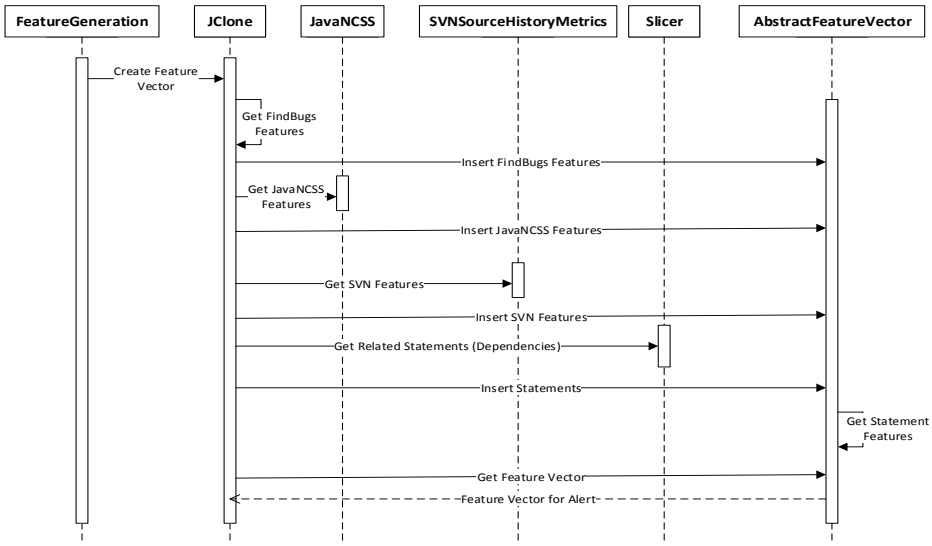

Figure B.3: A sequence diagram showing the interaction between classes in the next-generation JClone feature generation software component (as discussed in Section 8.2).

References

[1] Apache Software Foundation, The. Apache commons. http://commons.apache.org/, 2014.

[2] Apache Software Foundation, The. Apache log4j. http://logging.apache.org/log4j/1.2/, 2014.

[3] Apache Software Foundation, The. Apache tomcat 6.0. http://tomcat.apache.org/tomcat-6.0-doc/index.html, 2014.

[4] Apache Software Foundation, The. Open source tools. http://tomcat.apache.org/tools.html, 2014.

[5] Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, 2008.

[6] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 1–8, 2007.

[7] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010.

[8] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 36–47, 2008.

[9] Bharti Chimdyalwar and Shrawan Kumar. Effective false positive filtering for evolving software. In *Proceedings of the 4th India Software Engineering Conference*, pages 103–106, 2011.

[10] Coverity. Who uses coverity? http://www.coverity.com/customers/, 2013.

[11] Neil Davey, Paul Barson, Simon Field, and Ray J Frank. The development of a software clone detector, 1995.

[12] Eclipse Foundation, The. Eclipse Java development tools (JDT). http://www.eclipse.org/jdt/, 2013.

[13] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. *SIGOPS Oper. Syst. Rev.*, 35(5):57–72, 2001.

[14] FindBugs. Data mining of bugs with FindBugs. http://findbugs.sourceforge.net/manual/datamining.html, 2014.

[15] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. Finding patterns in static analysis alerts: Improving actionable alert ranking. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 152–161, 2014.

[16] Sarah Heckman and Laurie Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 41–50, 2008.

[17] Sarah Heckman and Laurie Williams. A model building process for identifying actionable static analysis alerts. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, pages 161–170, 2009.

[18] Sarah Heckman and Laurie Williams. A systematic literature review of actionable alert identification techniques for automated static code analysis. *Inf. Softw. Technol.*, 53(4):363–387, 2011.

[19] Sarah Heckman and Laurie Williams. Faultbench. http://www.realsearchgroup.org/faultbench/, 2014.

[20] Sarah Heckman and Laurie Williams. Faultbench v0.1. http://www.realsearchgroup.org/faultbench/parts/version_0.1/index_content.html, 2014.

[21] Sarah Smith Heckman. Adaptively ranking alerts generated from automated static analysis. *Crossroads*, 14(1):7:1–7:11, 2007.

[22] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[23] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, pages 96–105, 2007.

[24] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681, 2013.

[25] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware c analyzer by a bayesian statistical post analysis. In *Proceedings of the 12th international conference on Static Analysis*, pages 203–217, 2005.

[26] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 45–54, 2007.

[27] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. Int'l Symp. Static Analysis*, pages 40–56, 2001.

[28] Ted Kremenek, Ken Ashcraft, Junfeng Yang, and Dawson Engler. Correlation exploitation in error ranking. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 83–93, 2004.

[29] Christoph Clemens Lee. JavaNCSS - a source management suite for java. http://www.kclee.de/clemens/java/javancss/, 2014.

[30] Seunghak Lee and Iryoung Jeong. Sdd: high performance code clone detection system for large scale source code. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 140–141, 2005.

[31] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. Cp-miner: a tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 20–20, 2004.

[32] J. Mayrand, C. Leblanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 244–253, 1996.

[33] Nachiappan Nagappan and Thomas Ball. Static analysis tools as early indicators of pre-release defect density. Association for Computing Machinery, Inc., May 2005.

[34] The University of Waikato. Weka. http://www.cs.waikato.ac.nz/ml/weka/, 2013.

[35] Oracle. Operators. http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html, 2014.

[36] Polyglot. Polyglot. http://www.cs.cornell.edu/Projects/polyglot/, 2013.

[37] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using indus and kaveri. *Int. J. Softw. Tools Technol. Transf.*, 9(5):489–504, 2007.

[38] C.K. Roy and J.R. Cordy. Scenario-based comparison of clone detection techniques. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 153–162, 2008.

[39] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proceedings of the 30th international conference on Software engineering*, pages 341–350, 2008.

[40] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–122, 2007.

[41] The Eclipse Foundation. Eclipse. http://www.eclipse.org/, 2014.

[42] The University of Waikato. Weka. http://www.cs.waikato.ac.nz/~ml/weka/, 2013.

[43] University of Maryland. Findbugs. http://findbugs.sourceforge.net, 2012.

[44] University of Maryland. Findbugs users. http://findbugs.sourceforge.net/users.html, 2012.

[45] WALA. Main page. http://wala.sourceforge.net/wiki/index.php/Main_Page, 2013.

[46] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449, 1981.

[47] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. An empirical study on classification methods for alarms from a bug-finding static c analyzer. *Inf. Process. Lett.*, 102(2-3):118–123, 2007.