ABSTRACT

HECKMAN, SARAH SMITH. A Systematic Model Building Process for Predicting Actionable Static Analysis Alerts. (Under the direction of Laurie Williams).

Automated static analysis tools can identify potential source code anomalies, like null pointers, buffer overflows, and unclosed streams that could lead to field failures. These anomalies, which we call alerts, require inspection by a developer to determine if the alert is important enough to fix. Actionable alert identification techniques can supplement automated static analysis tools by classifying or prioritizing the alerts generated by automated static analysis such that the likelihood of a developer inspecting actionable alerts first is increased. By classifying and prioritizing actionable static analysis alerts, the developer will focus his or her time on inspecting and fixing actionable alerts rather than inspecting and suppressing unactionable alerts.

The goal of my research is to reduce inspection time by accurately predicting actionable and unactionable alerts when using static analysis by creating and validating a systematic actionable alert identification model. The Systematic Actionable Alert Identification (SAAI) process uses machine learning to identify actionable alerts. Investigation of the following three hypotheses will inform the goal of my research:

• Hypothesis 1: The artifact characteristics of an alert and the surrounding source code are predictive of the actionability of an alert.

• Hypothesis 2: A systematic actionable alert identification technique using machine learning can accurately identify actionable alerts.

• Hypothesis 3: A systematic actionable alert identification technique using machine learning is project specific.
A benchmark, FAULTBENCH, provides the evaluation framework for the proposed SAAI model building process and comparison with other actionable alert identification techniques. The dissertation presents a feasibility study and three empirical studies evaluating the hypotheses above. The feasibility study evaluates an adaptive actionable alert identification technique that utilizes the alert’s type and code location in addition to developer feedback to prioritize actionable alerts. The first empirical study investigates hypotheses 1-3 using FAULTBENCH on 15 SAAI models generated on five treatments for each of three subject programs. The treatments considered different grouping of alerts within revisions to train and test SAAI. The second empirical study is a comparative evaluation of the generated SAAI models with other actionable alert identification techniques in further evaluation of Hypothesis 2. Additionally, an empirical user study was conducted where students in the senior capstone project course used a custom SAAI model during development of their software project.

Selection of predictive artifact characteristics as part of the SAAI process suggests the acceptance of hypothesis 1. All but four of the 58 artifact characteristics used to build SAAI models were in one or more of the artifact characteristics subsets. The SAAI model identified actionable and unactionable alerts with greater than 90% accuracy for eight of the 15 FAULTBENCH subject treatments. Comparing SAAI models with other actionable alert identification techniques from literature found that SAAI models had the highest accuracy for 11 of the 15 treatments when classifying the full alert sets. Both of the above results support hypothesis 2. Due to accuracies greater than 90% when applying artifact characteristic subsets and machine learning algorithms for one subject program to another subject program, hypothesis 3 is not supported on the evaluated subject programs.
The contributions of this work are as follows:

- A systematic actionable alert identification model building process to predict actionable and unactionable automated static analysis alerts;
- A benchmark, FAULTBENCH, for evaluating and comparing actionable alert identification techniques; and
- A comparative evaluation of systematic actionable alert identification models with other actionable alert identification techniques from literature.
A Systematic Model Building Process for Predicting Actionable Static Analysis Alerts.

by
Sarah Smith Heckman

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2009

APPROVED BY:

Laurie Williams
Committee Chair

Steffen Heber

Robert St. Amant

Tao Xie
DEDICATION

To my husband, Walt.
BIOGRAPHY

Sarah Heckman is a member of the Realsearch Group in the Department of Computer Science at North Carolina State University. She received her MCS and BS in Computer Science from North Carolina State University in 2005 and 2004, respectively. Her research is currently in the area of Software Engineering, with a principal focus in static analysis alert prioritization. Sarah taught Introduction to Computing – Java and has 10 semesters of teaching assistant and lab instructor experience. She is the Managing Editor of the OpenSeminar in Software Engineering and OpenSeminar in Empirical Software Engineering. Sarah is the creator and first General Chair of the 2008 Symposium for Graduate Research in Computer Science held at North Carolina State University and led the first Graduate Student Teaching Assistant Training workshop. She was president and vice-president of the Women in Computer Science student organization, an officer-at-large in the Computer Science Graduate Student Association, and the graduate student representative on the Computer Steering Committee. Sarah served on the organizing committee for Geek-a-Thon and won the 2008 Joyce Hatch Service Award. She is a three-time recipient of the IBM PhD fellowship. In the fall of 2009, Sarah will become the 1st Teaching Assistant Professor for the Department of Computer Science at North Carolina State University.
ACKNOWLEDGMENTS

Many people have supported and encouraged me throughout this journey, and I would not have accomplished all that I have without them.

To Walt. Thank you for keeping me sane and motivated. You have inspired me to be a better person, comforted me during periods of stress, and ensured that I had fun along the way. I could not have done this without you

To Mom and Dad. Thank you both for always encouraging me towards my best. I knew that I could accomplish anything I set my mind to because you both believed in me. Thank you, Mom, for always listening when I needed to talk. Thank you, Dad, for being an example of following your dreams and doing something that you love every day. I hope to look forward to each day of work as you do.

To Patrick. Thank you for always believing in me. Your creativity and passion for your music and art is so inspiring.

To the Heckmans. Thank you for supporting and believing in me during this journey and accepting me so readily into your family.
To my uncle, Andy. Without you, I would never have considered Computer Science as a major, and found a career that I love. You always pushed me academically, and I hope that I have made you proud.

To Laurie. Thank you so much for being my advisor. Without you, I would not have pursued a Ph.D. You have provided me with guidance and motivation, while allowing me to grow and pursue other activities outside of my research. You have been and continue to be an inspiration of all that is possible in an academic career and life.

To the Realsearch Group (past and present): Nachi, Hema, Jiang, Mark, Dright, Lucas, Michael, Yonghee, Stephen, Andy, Lauren, Ben, Patrick, Raza, and Sophia. I could not have asked for a better group of people to work with during my graduate career. You all have taught me so much, making me a better person and researcher. Reading group, while painful, has been invaluable in ensuring that my presentation, theory, and results are as clear and correct as possible. Extra thanks to Ben who helped with last minute statistical advice in preparation of this document and for naming FAULTBENCH.

To Jim and Mike. Thank you for making my time at IBM enjoyable. I learned so much that cannot be taught in the classroom.
To my friends: Alex, Bill, Brandon, Diana, Donna, Kate, Kathryn, Khedron, Larz, Liz, Michael, Phyl, Rebecca, and Sarah. Thank you for reminding me that there is more to life than research: like cookouts, shopping, and fun!

To my committee. Thank you for your feedback and encouragement.

Funding. I have been funded by a CACC grant and an IBM Ph.D. Fellowship from 2006-2009.

Some of the work in this dissertation is revised and expanded from the following conference publications, which are copyright of the following organizations:


TABLE OF CONTENTS

LIST OF TABLES .................................................................................................................. x
LIST OF FIGURES ................................................................................................................ xi
LIST OF ABBREVIATIONS ..................................................................................................... xii

1 Introduction ....................................................................................................................... 1

2 Related Work ................................................................................................................... 5
  2.1 AAIT Overview ............................................................................................................. 6
    2.1.1 Alert Type Selection .............................................................................................. 7
    2.1.2 Contextual Information ....................................................................................... 7
    2.1.3 Graph Theory ...................................................................................................... 8
    2.1.4 Test Case Failures from Hybrid Analysis ........................................................... 8
    2.1.5 Mathematical and Statistical Models .................................................................. 9
    2.1.6 Machine Learning ............................................................................................... 9
    2.1.7 Data Fusion ......................................................................................................... 9
  2.2 Classification AAITs ..................................................................................................... 9
    2.2.1 Ogasawara, et al., 1998 ..................................................................................... 10
    2.2.2 Xiao and Pham, 2004 ......................................................................................... 10
    2.2.3 Williams and Hollingsworth, 2005 .................................................................... 11
    2.2.4 Aggarwal and Jalote, 2006 ................................................................................ 12
    2.2.5 Boogerd and Moonen, 2008b ............................................................................ 13
    2.2.6 Csallner, et al., 2008 ......................................................................................... 14
  2.3 Prioritization AAITs ..................................................................................................... 15
    2.3.1 Kremenek and Engler, 2003 .............................................................................. 15
    2.3.2 Kremenek, et al., 2004 ..................................................................................... 17
    2.3.3 Jung, et al., 2005 ............................................................................................... 18
    2.3.4 Kim and Ernst, 2007a ...................................................................................... 19
    2.3.5 Kim and Ernst, 2007b ...................................................................................... 20
    2.3.6 Kong, et al., 2007 ............................................................................................. 21
    2.3.7 Yi, et al., 2007 .................................................................................................. 22
    2.3.8 Boogerd and Moonen, 2008a .......................................................................... 23
    2.3.9 Meng, et al., 2008 .......................................................................................... 25
    2.3.10 Ruthruff, et al., 2008 ..................................................................................... 25

3 FAULTBENCH ................................................................................................................. 27
  3.1 Related Work .............................................................................................................. 28
  3.2 FAULTBENCH Process ............................................................................................. 30
    3.2.1 Generate Subject Revision History ................................................................. 30
    3.2.2 Subject Build Process ....................................................................................... 31
    3.2.3 Alert Oracle Creation ....................................................................................... 31
    3.2.4 Artifact Characteristic Generation .................................................................. 33
LIST OF TABLES

Table 2.1 Summarization of related work................................................................. 6
Table 3.1 FAULTBENCH V0.3 subject programs .................................................. 35
Table 4.1 Experimental treatments for benchmark evaluation .............................. 50
Table 4.2 FAULTBENCH V0.1 subject programs .................................................. 52
Table 4.3 Area under the anomaly detection curve for prioritization techniques ... 56
Table 4.4 Spearman rank correlation ....................................................................... 57
Table 4.5 Average precision, recall, and accuracy metrics of
un-inspected alerts at each inspection...................................................................... 58
Table 6.1 Train and test alert counts ....................................................................... 77
Table 6.2 Artifact characteristics selection counts for FAULTBENCH subject
programs .................................................................................................................. 78
Table 6.3 SAAI best models for jdom treatments .................................................... 84
Table 6.4 SAAI best models for runtime treatments .............................................. 85
Table 6.5 SAAI best models for logging treatments ............................................... 85
Table 6.6 Accuracy when switching models for treatment 100 SAAI .................... 88
Table 7.1 jdom summary data ................................................................................ 94
Table 7.2 runtime summary data ........................................................................... 102
Table 7.3 logging summary data ........................................................................... 109
Table 8.1 Subject survey responses to AWARE questionnaire ............................ 122
LIST OF FIGURES

Figure 3.1 An alert’s lifecycle .................................................................................. 32
Figure 3.2 Classification table (adapted from Zimmermann et al. [52]) ...................... 39
Figure 4.1 AWARE screenshot .................................................................................. 46
Figure 4.2 Anomaly detection curves for jdom ....................................................... 55
Figure 6.1 Histogram of artifact characteristic selection counts across subjects ..... 81
Figure 7.1 Accuracy across treatments for each AAIT on jdom ............................... 96
Figure 7.2 Precision across treatments for each AAIT on jdom ............................. 98
Figure 7.3 Recall across treatments for each AAIT on jdom ................................. 100
Figure 7.4 Accuracy across treatments for each AAIT on runtime ....................... 104
Figure 7.5 Precision across treatments for each AAIT on runtime ...................... 105
Figure 7.6 Recall across treatments for each AAIT on runtime ............................ 106
Figure 7.7 Accuracy across treatments for each AAIT on logging ....................... 111
Figure 7.8 Precision across treatments for each AAIT on logging ...................... 112
Figure 7.9 Recall across treatments for each AAIT on logging ............................ 113
LIST OF ABREVIATIONS

AAIT ................................................................. Actionable Alert Identification Technique

APM ............................................................................ Adaptive Prioritization Model

ASA ........................................................................... Automated Static Analysis

ATL-D ........................................................................ Alert Type Lifetime in Days

ATL-R ........................................................................ Alert Type Lifetime in Revisions

CNC ............................................................................. Check ‘n’ Crash

HWP ................................................................. History-based Warning Prioritization

LRM ........................................................................ Logistic Regression Model

SAAI ................................................................. Systematic actionable alert identifier
1 Introduction

Automated static analysis (ASA) tools can identify potential source code anomalies, like null pointers, buffer overflows, and unclosed streams, that could lead to field failures [21]. IEEE defines an anomaly “as a condition that deviates from expectations based on requirements specifications, design documents, user documents, or standards, or from someone’s perceptions or experiences” [24]. These potential anomalies, which we call alerts, require inspection by a developer to decide if the alert is important enough to fix [3]. If a developer decides the alert is an important, fixable anomaly, then we call the alert an actionable alert [18, 19, 41]. When an alert is not an indication of an actual code anomaly or the alert is deemed unimportant to the developer (e.g. the alert indicates a source code anomaly inconsequential to the program’s functionality), we call the alert an unactionable alert [18, 19].

Static analysis tools generate many alerts; an alert density of 40 alerts per thousand lines of code (KLOC) has been empirically observed [18]. Developers and researchers found that 35% to 91% of reported alerts are unactionable [2, 6, 18, 19, 26, 27, 30, 31]. A large number of unactionable alerts may lead developers and managers to reject the use of ASA as part of the development process due to inspection overhead [6, 27, 30, 31]. Suppose, a tool reports 1000 alerts and each alert requires five minutes for inspection. The time to inspect the alerts would take 10.4 uninterrupted eight-hour workdays. Identifying the 35%-91% unactionable alerts could lead to time savings of 3.6-9.5 days of developer time. Wagner, et al. [46] reported that identification of three or four actionable alerts that could be field failures
justifies the cost of using ASA. Identification of actionable alerts generated by ASA could reduce the inspection overhead associated with ASA.

Improving ASA’s ability to generate predominantly actionable alerts through development of tools that are both sound\(^1\) and complete\(^2\) is an intractable problem [9, 10]. Additionally, the development of algorithms underlying ASA requires a trade-off between the level of analysis and execution time [9]. Applying a very detailed analysis can report more alerts likely to be actionable [6], but may increase the time for running ASA [51]. Annotations can improve ASA by supplying conditions and invariants about the code under analysis, but the annotations may be specified incorrectly, and their creation requires developer overhead [14]. Customizing an ASA tool by selecting properties, like the type of alerts to search for, may also lead to a reduction of reported alerts and unactionable alerts [51], but may miss some actionable alerts.

Another way to reduce inspection overhead when using static analysis is to use the alerts generated by ASA with other information about the software under analysis, called artifact characteristics, to prioritize or classify alerts. We call these techniques actionable alert identification techniques\(^3\) (AAIT). AAITs supplement ASA tools by classifying or prioritizing the alerts generated by ASA such that the likelihood of a developer inspecting

---

\(^1\) For this research, we consider the generation of an alert as the indication of a potential anomaly. Therefore, sound static analysis ensures that all reported alerts are anomalies (actionable) [11].

\(^2\) For this research, we consider the generation of an alert as the indication of a potential anomaly. Therefore, complete static analysis ensures that if there is a place where an anomaly could occur in the source code, the tool reports an alert [11].

\(^3\) AAITs are techniques that identify actionable alerts. Some AAITs have been referred to as false positive or unactionable alert mitigation [18, 19], warning prioritization [26, 27], and actionable alert prediction [41].
actionable alerts earlier is increased. *Classification* AAITs divide alerts into two groups: alerts likely to be actionable and alerts likely to be unactionable [18]. *Prioritization* AAITs order alerts by the likelihood an alert is an indication of an actionable alert [18]. By classifying and prioritizing ASA alerts, the developer will focus his or her time on inspecting and fixing actionable alerts rather than inspecting and suppressing unactionable alerts.

The goal of my research is to *reduce inspection time by accurately predicting actionable and unactionable alerts when using static analysis by creating and validating a systematic actionable alert identification model*. Investigation of the following three hypotheses will inform the goal of my research:

- **Hypothesis 1**: The artifact characteristics of an alert and the surrounding source code are predictive of the actionability of an alert.
- **Hypothesis 2**: A systematic actionable alert identification technique using machine learning can accurately identify actionable alerts.
- **Hypothesis 3**: A systematic actionable alert identification technique using machine learning is project specific.

These hypotheses guided the creation and evaluation of the Systematic Actionable Alert Identifier (*SAAI*) process and were used to evaluate the efficacy of using an AAIT synergistically with ASA.

A benchmark, *FAULTBENCH* [18], was created as part of this research to provide the evaluation framework for the proposed *SAAI* process and the comparison of *SAAI* with other AAITs. The dissertation presents a feasibility study and three empirical studies, which
evaluate the above hypotheses. The feasibility study uses FAULTBENCH to evaluate three initial AAITs, which utilize the alert’s type and location to predict if the alert is actionable. The first empirical study investigates hypotheses 1-3 using FAULTBENCH on 15 SAAI models generated on five treatments for each of three subject programs. The treatments considered different grouping of alerts within revisions to train and test SAAI. The second empirical study is a comparative evaluation of the generated SAAI models with other AAIT from literature in further evaluation of Hypothesis 2. Additionally, an empirical user study was conducted where students in the senior capstone project course at North Carolina State University used AWARE\textsuperscript{4} containing a custom SAAI model during development of their software project. Data collected by the AAIT tooling and the student’s responses to a survey are reported.

The remainder of this dissertation is organized as follows: Chapter 2 discusses related work in the form of other AAITs, Chapter 3 presents FAULTBENCH, a benchmark for evaluating and comparing AAITs, Chapter 4 presents a feasibility study that provides the foundational work to the SAAI proposed in Chapter 5. Chapter 6 describes how SAAI performs on FAULBENCH and Chapter 7 compares SAAI with other AAIT. Chapter 8 presents a user study and Chapter 9 concludes, lists the contributions, and future work.

\textsuperscript{4} AWARE is an Eclipse plug-in that gathers ASA alerts, classifies them or prioritizes them with the applied AAIT, and presents the classified or prioritized alerts to the developer: http://agile.csc.ncsu.edu/aware.
2 Related Work

Many AAIT for supplementing ASA have been proposed in the literature. The following subsections describe 18 AAIT. The AAIT are divided into two groups: classification and prioritization. Table 2.1 provides an overview of the related work with a forward reference to the section that contains the details of each paper. The limitations of each AAIT are described in the subsection for that study. A general limitation to all AAITs is that there is not yet sufficient empirical evidence for any of the models that allow for generalization beyond application to their subject programs or for a comparison between AAIT based upon information available in the literature. First, the high-level approaches for AAIT are described in Section 2.1. Next, the AAIT are described in chronological order by publication date within the classification (Section 2.2) and prioritization (Section 2.3) groupings.
Table 2.1. Summarization of related work. A listing of the ASA supplemented by the proposed AAIT, the programming language, the AAIT name, the high-level AAIT type (as discussed in Section 2.1), and the research methodology used to evaluate the AAIT proposed in each reference.

<table>
<thead>
<tr>
<th>Sec.</th>
<th>Paper</th>
<th>ASA</th>
<th>Lang.</th>
<th>AAIT Name</th>
<th>AAIT Type</th>
<th>Research Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.4</td>
<td>Aggarwal and Jalote, 2006 [2]</td>
<td>BOON</td>
<td>C</td>
<td>AJ06</td>
<td>contextual information</td>
<td>other</td>
</tr>
<tr>
<td>2.3.8</td>
<td>Boogerder and Moonen, 2008a [8]</td>
<td>UNKNOWN</td>
<td>C</td>
<td>ELAN, EFANv, EFANy</td>
<td>graph theory</td>
<td>other baseline comparison</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Boogerder and Moonen, 2008b [7]</td>
<td>QA C, QMORE</td>
<td>C</td>
<td>BM08B</td>
<td>graph theory</td>
<td>other</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Csallner, et al., 2008 [12]</td>
<td>ESC/JAVA</td>
<td>Java</td>
<td>CHECK ‘N’ CRASH, DSD-CRASHER</td>
<td>test case failures</td>
<td>other model comparison</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Jung, et al., 2005 [25]</td>
<td>AIRIC</td>
<td>C</td>
<td>JKS05</td>
<td>machine learning</td>
<td>train and test</td>
</tr>
<tr>
<td>2.3.4</td>
<td>Kim and Ernst, 2007a [26]</td>
<td>FINDBUGS, PMD, JLINT</td>
<td>Java</td>
<td>ALERTLIFETIME</td>
<td>math/stat models</td>
<td>other baseline comparison</td>
</tr>
<tr>
<td>2.3.5</td>
<td>Kim and Ernst, 2007b [27]</td>
<td>FINDBUGS, PMD, JLINT</td>
<td>Java</td>
<td>HWP</td>
<td>math/stat models</td>
<td>train and test</td>
</tr>
<tr>
<td>2.3.6</td>
<td>Kong, et al., 2007 [29]</td>
<td>RATS, ITS4, FLAWFINDER</td>
<td>C</td>
<td>ISA</td>
<td>data fusion</td>
<td>other model comparison</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Kremenek and Engler, 2003 [31]</td>
<td>MC</td>
<td>C</td>
<td>Z-RANKING</td>
<td>math/stat models</td>
<td>random and optimal comp.</td>
</tr>
<tr>
<td>2.3.9</td>
<td>Meng, et al., 2008 [33]</td>
<td>FINDBUGS, PMD, JLINT</td>
<td>Java</td>
<td>MMW08</td>
<td>data fusion</td>
<td>other</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Ogasawara, et al., 1998 [36]</td>
<td>QA C</td>
<td>C</td>
<td>OAY98</td>
<td>alert type selection</td>
<td>other</td>
</tr>
<tr>
<td>2.3.10</td>
<td>Ruthruff, et al., 2008 [41]</td>
<td>FINDBUGS</td>
<td>Java</td>
<td>RPM08</td>
<td>math/stat models</td>
<td>other model comparison</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Williams and Hollingsworth [48]</td>
<td>RETURN VALUE CHECKER</td>
<td>C</td>
<td>HISTORYAWARE</td>
<td>math/stat models</td>
<td>other model comparison</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Xiao and Pham, 2004 [50]</td>
<td>UNKNOWN</td>
<td>C</td>
<td>SCAS</td>
<td>contextual information</td>
<td>other</td>
</tr>
<tr>
<td>2.3.7</td>
<td>Yi, et al., 2007 [51]</td>
<td>AIRIC</td>
<td>C</td>
<td>YCK07</td>
<td>machine learning</td>
<td>train and test</td>
</tr>
</tbody>
</table>

2.1 AAIT Overview

The following subsections introduce the high-level trends in the creation of AAITs. Each of the AAITs in the related work use one of the high-level approaches for predicting actionable
alerts listed below. The specific AAIT used in each study are described in Section 2.2 and 2.3, below.

2.1.1 Alert Type Selection

ASA tools list the types of problems that can be detected (e.g. a potential null pointer access or an unclosed stream) via a detector or bug pattern, which we call alert type. ASA tools may allow for selection of individual alert types by the user. Selecting alert types that are more relevant for a code base leads to the reduction of reported unactionable alerts, but may also lead to the suppression of actionable alerts in the types that were not selected. Alert type selection works best for alert types that tend to be homogeneous by type (e.g. where all alerts of a type are either actionable or unactionable, but not both) [26, 27, 30]. The alert types that are most relevant may vary by code base. Therefore, a study of the actionability of alert types for a particular code base is required to select the appropriate types. If alerts sharing the same type have been fixed in the past, then the other alerts of that type are likely actionable. AAITs that use alert type selection either use the alert history for a project that can be found through mining the source code repository or bug database or additional knowledge about the specific alert types to determine which alert types to select for a project.

2.1.2 Contextual Information

Due to imprecision in the analysis, ASA may miss anomalies [2]. ASA may also not understand code constructs like pointers, which may lead to a large number of unactionable alerts [2, 50]. By understanding the precision of ASA and selecting areas of code that an ASA tool can analyze well (e.g. code with no pointers), the number of generated and
unactionable alerts can be reduced. The selection of code areas to analyze by ASA can be a manual or automated process created by knowledge of the ASA tool’s limitations. Contextual information techniques identify areas of code where ASA will identify the most actionable alerts and the fewest unactionable alerts [2, 50].

2.1.3 Graph Theory

AAITs that use graph theory to identify actionable alerts take advantage of the source code’s structure to provide additional insight to static analysis. System dependence graphs provide insight into both the control and data flow for a program and are used to calculate the execution likelihood of a particular location of code that contains a static analysis alert [6, 8]. Other graphs of artifact characteristics, like the source code repository history, can also show the relationship between source code changes that may be associated with openings and closures of static analysis alerts [7].

2.1.4 Test Case Failures from Hybrid Analysis

Unlike static analysis, the results of dynamic analyses do not require inspection because a failing condition is identified through program execution [12]. Dynamic analyses can improve the results generated by static analysis through the generation of test cases that may cause the location identified by the alert to demonstrate faulty behavior [12]. Additionally, by using static analysis to focus automated test case generation, some of the limitations to automated test case generation, like a large number of generated tests, may be reduced [12].
2.1.5 Mathematical and Statistical Models

AAITs may use mathematical or statistical models to determine if an alert is actionable or unactionable. In some cases, these AAITs may exploit knowledge about the specific ASA tool and where mistakes may be made in the analysis to determine if other alerts are actionable or not [31]. Other AAITs may use the history of the code to build a linear model that may predict actionable alerts [26, 27, 41, 48]. Additionally, knowledge about the ASA tools and the observed relationships between the alerts can be used to create linear and other mathematical models [18].

2.1.6 Machine Learning

Machine learning “is the extraction of implicit, previously unknown, and potentially useful information about data” [49]. Machine learning techniques find patterns within sets of data and may then use those patterns to predict if new instances of the data are similar to other instances. AAITs can use machine learning to predict or prioritize alerts as being actionable or unactionable by using information about the alerts and surrounding code [19, 25, 30, 51].

2.1.7 Data Fusion

Data fusion combines data from multiple ASA tools and merges redundant alerts. Similar alerts from multiple tools increase the confidence that an alert is actionable [29].

2.2 Classification AAITs

Classification AAITs divide alerts into two groups: alerts likely to be actionable and alerts likely to be unactionable [18]. These AAITs use one of the high-level approaches above as
the foundation for predicting actionable alerts. The subsections below describe the seven classification AAITs.

2.2.1 Ogasawara, et al., 1998

Ogasawara, et al. [36] use alert type selection to identify the most actionable alert types. The static analysis team identified the 41 alert types from over 500 alert types available in QA C from their experiences using ASA. Out of initial 1,459 alerts generated by QA C, 250 alerts were of the selected alert types and 88 of those alerts were found to be actionable. Thirty percent of the 88 actionable alerts were found to be serious problems in the system during inspection.

One limitation not discussed by Ogasawara, et al., is that not all of the alerts of the types removed from the analysis may be unactionable (i.e. actionable alerts were suppressed resulting in a false negative). For the alert types that remain in the analysis, there may still be many unactionable alerts that are generated.

2.2.2 Xiao and Pham, 2004

Xiao and Pham [50] use contextual information about the code under analysis to extend a static analysis tool reducing. Unactionable alert reduction was added to three different detectors: (1) memory leak, (2) missing break, and (3) unreachable code. The memory leak detector keeps track of pointers, especially to global variables, at the function level, and searches for memory leaked specifically from local pointers. If a local pointer does not have a memory leak, then the alert is considered unactionable and is not reported to the developer. The missing break detector uses belief analysis. Belief analysis uses the source code to infer
the developer’s beliefs about software requirements. The beliefs inferred from the context of
the source code are combined with a lexical analysis of the comments to determine if missing
break alerts are actionable or unactionable. The unreachable code detector maintains a
database of patterns that suggest unreachable code. Alerts reported about unreachable code
may be compared with the patterns in the database. Additionally, any unreachable code alert
suppressed by the developer in the user interface of their tool is transformed into the
constituent pattern and recorded in the database. The additional techniques suggested by
Xiao and Pham may generate overhead making them too costly to use in-process. Xiao and
Pham [50] find that their unactionable alert suppression technique suppresses 32% of the
generated alerts.

2.2.3 Williams and Hollingsworth, 2005

Williams and Hollingsworth [48] use a mathematical and statistical model in addition to
source code repository mining to drive the creation of an ASA tool and to improve the
prioritized listing of alerts. The static analysis tool Williams and Hollingsworth created
identifies locations where the return value of a function call is not checked before use. Alerts
associated with the called function are grouped together, and the called functions are ranked
using the HISTORYAWARE prioritization technique. HISTORYAWARE first groups functions by
mining the software repository for instances in the project’s history where the function call’s
return value had a check added, leading to the alert’s closure in later revisions. Next, the
latest revision of the source code is considered and the number of times the return value of a
function is checked are counted. The functions are prioritized by the count of checked
functions. If the return value of a function is checked most of the time, then the prioritization of that function is high, indicating that instances where the return value is not checked are likely actionable. However, if the return value of a function is almost never checked, then the alerts are likely unactionable. When the return value of a called function is always or never checked, the tool does not alert the developer because there are no inconsistencies.

Williams and Hollingsworth [48] perform a case study to compare a naïve ranking of the alerts based on the current version of code and the HISTORYAWARE prioritization. They measure the precision and recall of the classification. The precision of the top 50 alerts generated by static analysis is 62.0% for HISTORYAWARE and 53.0% for NAÏVERANKING for Wine\(^5\) and 42.0% for HISTORYAWARE and 32.0% for NAÏVERANKING for Apache httpd\(^6\). The HISTORYAWARE ranking has a false positive rate between approximately 0 and 70% across the contemporary context of the alerts. The NAÏVERANKING false positive rate between 50% and 100% on the same contemporary context.

2.2.4 Aggarwal and Jalote, 2006

Aggarwal and Jalote [2] use contextual information to identify potential buffer overflow vulnerabilities, as represented by the strcpy library function in C source code, quickly and effectively through a combination of static and dynamic analysis. The ASA tool, BOON [45], has difficulty in understanding aliased pointers, which may lead to BOON missing buffer overflow vulnerabilities. Dynamic tools, like STOBO [16], can find vulnerabilities where

\(^5\) Wine is an open source program for running Windows applications on Linux, Unix, and other similar operating systems. Wine may be found at: http://www.winehq.org/.

\(^6\) Apache httpd is open source server software that may be found at: http://httpd.apache.org/.
static analysis fails. Dynamic analysis requires the generation of test cases, which can increase the time required to use the tool. Aggarwal and Jalote [2] created a tool that identifies areas of code that require buffer overflow analysis and marks the code where pointers are aliased and where they are not. The latter are better for static analysis while the former necessitate dynamic analysis.

Aggarwal and Jalote [2] ran their analysis on rdesktop and wzdftpd and found an increase in static analyzer accuracy and a reduction of test cases required to find buffer overflow vulnerabilities. The analysis of wzdftpd showed that only 37.5% of the dangerous strcpy functions in the code required dynamic analysis. The tool generated by Aggarwal and Jalote [2] for identifying aliased pointers, is limited when structured elements are aliased. Additionally, the buffer overflow vulnerability is not restricted to strcpy functions only. Therefore, the technique can only identify a subset of buffer overflow vulnerabilities.

2.2.5 Boogerd and Moonen, 2008b

Boogerd and Moonen [7] present a technique for evaluating the actionable alert rate for ASA alert types that deal with style issues generated by the ASA tool QA C. I think the technique they propose can be extended to assess the actionable alert rate for all ASA alert types and can be used to prioritize alert types. The actionable alert rate for an alert type is the number of actionable alerts for the alert type divided by all alerts generated for the alert type. They evaluate two prioritization techniques: temporal coincidence and spatial coincidence. Temporal coincidence associates alerts with code changes. However, just because an alert is removed due to a code change does not mean that the alert was associated with the
underlying anomaly. Spatial coincidence reduces the noise from temporal coincidence by assessing the type of change made that removed an alert. Alerts are considered actionable if they are associated with changes due to fixing faults rather than other source code changes. The requirement for generating spatial coincidence is that changes in the source code that are checked into the repository should be associated with bugs listed in the bug database.

Boogerd and Moonen [7] use graph theory to prioritize alert types. A version history graph is created for each file and is annotated with code changes on each edge. Alerts closed due to fixing a fault increment the alert count. After all versions of a file are evaluated, the remaining open alerts contribute to the overall count of generated alerts. The actionable alert rate values can be used to prioritize alert types in future versions of software.

Boogerd and Moonen [7] used the MISRA-C style rules reported by QAC and compared the alert closures with faults fixed in the history of the development of an embedded mobile TV software (TVoM). The experiment considers 214 revisions of the TVoM project developed between August 2006 until June 2007. Information about the bug reports were mined from the bug database, and had to meet the following requirements: 1) reports were a bug and not a functional change request; 2) associated with the C portions of the project; and 3) reports that were closed on or before June 2007.

2.2.6 Csallner, et al., 2008

Csallner, et al. [12] use augment ASA by generating and running automated unit test cases for each reported alert. Test case failures identify alerts that have a concrete error path that
are likely actionable. The CHECK ‘N’ CRASH and DSD-CRASHER tooling, respectively, are evaluated in Csallner, et al. [12].

Csallner et al. [12] used JBoss JMS and Groovy as subjects to compare CHECK ‘N’ CRASH with DSD-CRASHER. Csallner et al. found that they could eliminate one unactionable alert when using DSD-CRASHER for JBoss JMS and seven unactionable alerts when analyzing Groovy. Additionally, Csallner, et al. compared DSD-CRASHER with eCLAT [38], a dynamic tool. DSD-CRASHER found three class cast exceptions that eCLAT did not find for JBoss JMS and two additional alerts that eCLAT missed when analyzing Groovy. Finally, Csallner, et al. investigated how well the ASA underlying CHECK ‘N’ CRASH and DSD-CRASHER, ESC/JAVA 2, finds bugs seeded in open source projects. They evaluated three versions of Apache XML Security containing 13-20 seeded bugs. Approximately half of the seeded bugs are unable to be found by ESC/JAVA 2, and the remainder that could be associated with alerts generated by ESC/JAVA 2 had no associated failing test cases generated by DSD-CRASHER.

2.3 Prioritization AAITs

Prioritization AAITs order alerts by the likelihood an alert is an indication of an actionable alert [18]. The subsections below describe the 10 prioritization AAITs.

2.3.1 Kremenek and Engler, 2003

Kremenek and Engler [31] proposed a statistical model for prioritizing static analysis alerts. Unlike most other ASA tools, the MC [13] ASA tool reports “(1) the locations in the program that satisfied a checked property [successful checks] and (2) locations that violated the checked property [failed checks]” [31]. The Z-RANKING statistical technique is built on
the premise that alerts, identified by the same detector, and associated with many successful checks are likely actionable. Additionally, Kremenek and Engler hypothesize that unactionable alerts are associated with many other unactionable alerts. A special case is that if there are no successful checks then the alerts are likely unactionable. A hypothesis test is run on the proportion of successful checks out of all reports for a given grouping of checks. Alerts are grouped by an artifact characteristic, called a grouping operator, the alerts have in common (e.g. call site, number of calls to free memory, function). The hypothesis testing allows for consideration of the size of each possible grouping of checks, and the final number is called a z-score. Alerts generated by static analysis are prioritized by their z-score. A limitation of the z-ranking technique is that the prioritization’s success depends on the grouping operator.

Kremenek and Engler [31] compare alerts ranked with the z-ranking technique against an optimal and random ranking of the same alerts. The ranking of the alerts are evaluated for each of alert types reported by MC. A hypergeometric distribution is used to generate the random ordering of alerts.

Kremenek and Engler [31] evaluate the z-ranking of alerts for three different detectors in the MC system: lock error, free errors, and string format errors. The cumulative number of bugs discovered is plotted on a graph for each inspected alert. For the lock errors, the first 10% of the ranked alerts found 3 time to 6.9 times more bugs than the first 10% of randomly ordered alerts. For the free errors and string format errors, the first 10% of the ranked alerts found 3.3 times and 2.8 times the bugs than the first 10% of randomly ordered alerts.
Additionally, Kremenek and Engler randomly generate $1.0 \times 10^5$ ordering of alerts, and determine the number of orderings that are better than the ordering generated by the z-ranking technique. At most, 1.5% of the random orderings were better than alerts ordered by z-ranking.

2.3.2 Kremenek, et al., 2004

Based on the intuition that alerts sharing an artifact characteristic tend to be either all actionable or all unactionable, Kremenek et al. [30] developed an adaptive prioritization algorithm, FEEDBACK-RANK that prioritizes alerts generated by the MC ASA tool. Each inspection of an alert by a developer adjusts the ranking of uninspected alerts. After each inspection, the set of inspected alerts are used to build a Bayesian Network (a machine learning algorithm), which models the probabilities that groups of alerts sharing a characteristic (e.g. location in the course code) are actionable or unactionable. Additionally, a value representing how much additional information inspecting the report will provide to the model is generated for each alert. The information gain value is used to break ties between alerts with the same probability of being an anomaly.

Kremenek et al. [30] compare alerts ordered by their FEEDBACK-RANK to the optimal and random ordering of the same alerts. For the FEEDBACK-RANK algorithm, they consider two alert treatments. In one treatment, there is no information about already-inspected alerts to build the model. The model is updated as alerts are inspected, which represents a project just starting to use ASA. The other treatment considers a set of alerts as already inspected, and uses the classifications from those alerts to build the initial model, which could potentially...
lead to a better initial prioritization of alerts. Additionally, Kremenek et al. use three different alert sets to generate different models, in particular the conditional probability distribution of the Bayesian Network: the entire code base, self-trained, and a 90% reserved model. For the Bayesian Network trained on the entire code base, all of the generated alerts and their classifications are used to build the conditional probability distribution of the actionable and unactionable alerts. For the self-trained set, the conditional probability distribution values are trained on the set of alerts that are also ranked by the Bayesian Network. Finally, in the 90% reserved model, 90% of the alerts are used to train the conditional probability distributions for the Bayesian Network and the model is tested on the remaining 10% of alerts.

Kremenek et al. [30] use a custom metric, performance ratio, to compare the rankings generated via their technique and the random ordering of alerts. Performance ratio is the ratio between random and the ranking techniques’ “average inspection ‘delay’ or ‘shift’ per bug from optimal.” The results show that all detectors in the MC system show a 2-8x improvement of performance ratio over random when using feedback-rank. The self-trained Alock detector model showed a 6-8x improvement of performance ratio over random when seeded with partial knowledge some alert classifications.

2.3.3 Jung, et al., 2005

Jung et al. [25] use a Bayesian network (a machine learning algorithm) with a Monte Carlo technique to generate the probability of an actionable alert given a set of 22 code characteristics (e.g. syntactic and semantic code information like nested loops, joins, and
array information). The model is generated via inspected alerts generated on the Linux kernel code and several textbook C programs. A user-specified threshold limits the number of alerts reported to developers, which reduces the set of alerts for a developer to inspect.

The train and test technique was used to evaluate the proposed prioritization technique. The alerts were randomly divided into two equal sets. One set was used to train a Bayesian model using the artifact characteristics (which Jung, et al. [25] call symptoms) generated for each alert, and the model was tested on the second set of alerts. The training and testing of models was repeated 15 times.

Jung, et al. [25] evaluated their prioritization technique on “… some parts of the Linux kernel and programs that demonstrate classical algorithms.” The precision, recall, and accuracy are 38.7%, 68.6%, and 73.7%, respectively. Additionally, 15.17% of the unactionable alerts were inspected before 50% of the actionable alerts were inspected. Jung et al. observe that if the threshold for “trueness” is lowered, then all actionable alerts will be provided to the user at the cost of an unknown additional amount of unactionable alerts.

2.3.4 Kim and Ernst, 2007a

Kim and Ernst [26] prioritize alert types by the average lifetime (a mathematical model) of the alerts. The premise is that alerts fixed quickly are more important to developers; therefore, the developer will want to inspect alert of that type. The lifetime of an alert is measured at the file level from the time the first instance of an alert type appeared until closure of the last instance of an alert type. Alerts that remain in the file at the last studied revision are given a penalty of 365 days added to their lifetime. Lack of alert tracing when
line and name changes occur leads to error in the alert lifetime measurement and the variance of lifetimes for an alert type is unknown. The technique assumes that important problems are fixed quickly; however, alerts that are fixed quickly may be the easiest bugs to fix and not the most important alerts [26].

Kim and Ernst [26] validate their alert lifetime prioritization model by comparing the alerts ordered by lifetime with alerts ordered by the tool specified severity, and found that the alert lifetime prioritization did not correspond to the tool specified severity. Additionally, the alert lifetime prioritizations between two projects are compared. The correlation coefficient between the alert type prioritization between the subject programs was 0.218, which demonstrates that the alert type ordering for one program may not be applicable for another program.

2.3.5 Kim and Ernst, 2007b

Kim and Ernst [27] use the commit messages and code changes in the source code repository to prioritize alert types using a mathematical model. The history-based warning prioritization (HWP) weights alert types by the number of alerts closed by fault- and non-fault fixes. A fault-fix is a source code change where a fault or problem is fixed (as identified by a commit message) while a non-fault fix is a source code change where a fault or problem is not fixed, like a feature addition. The initial weight for an alert type is zero. At each fault-fix the weight increases by an amount, $\alpha$. For each non-fault-fix, the weight increases by $1-\alpha$. The final step normalizes each alert type’s weight by the number of alerts sharing the type. A higher weight implies that alerts with a given type are more likely to be actionable. The
prioritization technique considers all alert sharing the same type in aggregate, which assumes that all alerts sharing the same type are homogeneous in their classification. The prioritization fails for alerts generated in later runs of ASA if the alert type never appears in earlier versions of the code.

The fix-change prioritization suggested by Kim and Ernst [27] was evaluated by training the model using the first \((n/2)-1\) revisions and then testing the model on the latter half of the revisions. The precision of the tool’s alert prioritization with the prioritization of alerts based on the project’s history were compared. The best precision for the three subject programs (Columba, Lucene, and Scarab) is 17\%, 25\%, and 67\%, respectively when using HWP as compared to 3\%, 12\%, and 8\%, respectively when prioritizing the alerts by the tool’s severity or priority measure. Additionally, when only considering the top 30 alerts, the precision of the fix-based prioritization is almost doubled, and in some cases tripled, from the tool’s ordering of alerts.

2.3.6 Kong, et al., 2007

Kong, et al. [29] use data fusion to identify vulnerable code using alerts generated by ASAs focused on finding security vulnerabilities. The ISA tool, created by Kong, et al., reports a score for each aggregated alert type, which represents the likelihood the alert is a vulnerability. The score is the combination of the tool’s alert severity and the contribution of each tool summed across all tools. The feedback from the user when inspecting alerts contribute to the weights associated with a specific ASA. The technique is limited by the mapping of alerts between tools.
Kong, et al. [29] compare the prioritization of alerts when using data fusion of three tools with each of the tools individually. All of the subject programs, wuftp7, Net-tools8, and Pure-ftpd9, have known vulnerabilities. The results show that ISA has a lower rate of false positives and false negatives than the individual ASA for two of the three subject programs. Additionally, ISA is found to be more efficient (defined as the likelihood of finding a vulnerability when inspecting the alerts) than the individual static analysis tools.

2.3.7 Yi, et al., 2007

Yi et al. [51] compare the prioritization of actionable and unactionable alerts for several machine learning algorithms. The static analysis tool AIRAC, finds potential buffer overrun vulnerabilities in C code. Three types of symptoms or artifact characteristics may be predictive of actionable alerts: syntactic, semantic, and information about the buffer uncovered by static analysis. Yi et al. did not consider attribute subset selection techniques to identify uncorrelated symptoms except when building linear regression.

Yi et al. [51] utilize eight machine learning techniques to prioritize static analysis alerts into actionable and unactionable groups. The symptoms about the alerts are the independent variables and the classification of the alert is the dependent variable. The alerts are divided into a training and test set using an approximately two-thirds one-third split. The training-

7 wu-ftpd is ftp software for Unix systems.
9 Pure-ftpd is ftp server software for Unix systems: http://www.pureftpd.org/project/pure-ftpd.
The test cycle is repeated 100 times. The results are summed over all 100 models. The open-source statistical program R\textsuperscript{10} was used to train and test the models.

Yi et al. evaluated their alert classification models on 36 files and 22 programs, the details of which are not provided. Overall, there were 332 alerts generated for all of the sample programs. The different machine learning techniques were evaluated by comparing the ACU of ROC curves were compared. The closer the area is to 1, the better the performance of the model. The AUC for the ROC curves varied from 0.8745-0.9290. Additionally, only 0.32\% of the unactionable alerts were identified before the first 50\% of the actionable alerts. Also, 22.58\% of the actionable alerts were inspected before the first unactionable alert was inspected.

2.3.8 Boogerd and Moonen, 2008a

Boogerd and Moonen [8] prioritize alerts by execution likelihood [6] and by execution frequency [8] using graph theory. Execution likelihood is defined as “the probability that a given program point will be executed at least once in an arbitrary program run” [8]. Execution frequency is defined as “the average frequency of [program point] v over all possible distinct runs of [program] p” [8]. Alerts with the same execution likelihood are prioritized the same, but may actually have varying importance in the program. Execution frequency solves the limitation of execution likelihood by providing a value of how often the code will be executed [8].

\textsuperscript{10}R is an open source statistical program: http://www.r-project.org/.
Prediction of the branches taken when calculating the execution likelihood and frequency are important to the Execution Likelihood ANalysis (ELAN) and Execution Frequency ANalysis (EFAN) techniques introduced by Boogerd and Moonen [8]. The ELAN AAIT (introduced in [6]) traverses the system dependence graph of the program under analysis and generates the execution likelihood of an alert’s location and heuristics are used for branch prediction. There are two variations of the EFAN AAIT: one uses heuristics for branch prediction based on literature in branch prediction (EFAN$_H$) and the other uses value range propagation (EFAN$_V$). Value range propagation estimates the values of variables from information in the source code.

Five open source programs (Antiword, Chktex, Lame, Link, and Uni2Ascii) were used in the case study to compare ELAN and EFAN. Boogerd and Moonen [6, 8] compare the effectiveness of their prioritization technique with actual execution data, gathered by automated regression test runs, and not with the actual actionability of ASA alerts.

Boogerd and Moonen [8] use Wall’s unweighted matching method [47] to compare the prioritized list of alerts with the list of alerts ordered by the actual execution data. The ELAN AAIT had an average correlation of 0.39 with the actual execution values for the top 10% of alerts, which outperformed the EFAN$_H$ and EFAN$_V$ with correlations of 0.28 and 0.17, respectively. One limitation of the work is that the created system dependence graph may miss dependencies, which could lead to missing potential problems. Additionally, dependencies that are actually impossible to traverse introduce unactionable alerts.
2.3.9 Meng, et al., 2008

Meng et al. [33] propose a data fusion approach that merges alerts that are common across multiple static analysis tools run on the same source code. The combined alerts are first prioritized by the severity of the alert and are then prioritized by the number of tools that identify the alerts. A map associates alerts for a specific tool to a general alert type. Meng et al.’s [33] technique has the same limitations as Kong et al. [29].

Meng et al. [33] evaluate their data fusion technique on a small, unnamed, sample program. They run FindBugs, PMD, and JLint on the subject program. Meng et al. [33] report four of the alerts generated for the small subject program, one of which was reported by two tools.

2.3.10 Ruthruff, et al., 2008

Ruthruff et al. [41] use a logistic regression model to predict actionable alerts. Thirty-three artifact characteristics are considered for the logistic regression model. Reducing the number of characteristics for inclusion in the logistic regression model is done via a screening process, whereby logistic regression models are built with increasingly larger portions of the alert set. Characteristics with a contribution lower than a specified threshold are thrown out until some minimum number of characteristics remains. The minimum number of artifact characteristics is from models in related literature [4, 34, 37]. The final logistic regression model is created on a sample of the alerts. The screening process could identify collinear artifact characteristics [41].
Ruthruff et al. [41] considered two types of models: one for predicting unactionable alerts and the other for predicting actionable alerts. For the actionable alerts model, two specific models were built: one considered only alerts identified as actionable and the second considered all alerts.

Ruthruff et al. [41] evaluate their prioritization technique by comparing their models with a modified model from related work in fault identification and a model built using all of the suggested artifact characteristics. The related work models use complexity metrics to predict faults and come from the work by Bell, Ostrand, and Weyuker [4, 37]. Ruthruff et al. [41] adapt the models, which they call BOW and BOW+, for alert prioritization. The BOW model is directly from the work by Bell, et al. The BOW+ models include two static analysis metrics, the bug type and the alert priority, in addition to the complexity metrics. The model generated by identifying the most important artifacts characteristics is also compared to a model built using all of the artifact characteristics.

Ruthruff et al. [41] evaluated the models by comparing the cost in terms of time to generate the data and build the model and the precision of the prediction. The proposed model building technique took slightly less than seven hours to build and run, which is reasonable compared to the five days required to build a model using all of the data. However, the proposed model takes a much longer time than the BOW and BOW+ models. The precision of the screening models ranged from 73.2%-86.6%, which was typically higher than the BOW and BOW+ models with precision between 60.9%-83.4%, especially when predicting actionable warnings.
Several open questions in software engineering involve evaluating processes and techniques that potentially improve aspects of the software development lifecycle. Empirical analysis of research theories are a component for acceptance of the theory within a research community [43]. Benchmarks provide an experimental basis for evaluating software engineering theories, represented by software engineering techniques, in an objective and repeatable manner [43]. A *benchmark* is defined as “a procedure, problem, or test that can be used to compare systems or components to each other or to a standard” [23]. Benchmarks represent the research problems of interest and solutions of importance in a research area through definition of the motivating comparison, task sample, and evaluation measures [42]. The task sample can contain programs, tests, and other artifacts dependent on the benchmark’s motivating comparison. A benchmark controls the task sample reducing result variability, increasing repeatability, and providing a basis for comparison [42]. Additionally, successful benchmarks promote collaboration within a research community [42].

Several benchmarks in the realm of software anomaly detection have emerged in recent years [32, 35, 40]. These benchmarks often contain subject programs of various sizes, in multiple languages, and with real or seeded faults. Current benchmarks provide meaningful points of comparison; however, they lack a detailed, repeatable process. Supplementing prior benchmarks by gathering a set of small, real, and anomalous Java programs from a variety of domains and providing a process allow for the evaluation of the following software anomaly
detection problem: how to identify which alerts generated by static analysis tools are program anomalies.

FAULTBENCH provides a basis for comparison of AAITs and contributes subject programs; an analysis procedure; and evaluation metrics. The current version of FAULTBENCH, v0.3, contains three, open-source subject programs written in Java. We describe how we created FAULTBENCH, and present the process of evaluating and comparing FP mitigation techniques.

### 3.1 Related Work

There are several benchmarks in the realm of software anomaly detection. The SIEMENS [22] benchmark was created by researchers at Siemens Corporate Research and contains multiple versions of small C programs each containing a single anomaly and a suite of test cases. The benchmarks were created to evaluate control- and data-flow test adequacy criteria and were later used by Rothermel et al. [40] to evaluate regression test case prioritization.

BUGBENCH [32] is a benchmark containing seventeen buggy, open source, C/C++ applications ranging from seven thousand lines of code (KLOC) to 1028 KLOC in various domains. A Java benchmark for evaluation of the CHORD race condition detection static analysis tool [35] contains twelve concurrent programs ranging from 2.5 KLOC to 650 KLOC. PROMISE [5] is a repository for data sets from empirical research in predictive modeling, and half of the 60 data sets are for anomaly prediction. However, most of the PROMISE data sets provide metrics without the project source, and some data sets refer to large, open source projects while the remainders refer to commercial products. Other static
analysis researchers [26, 27, 30, 48] have used large open source projects (e.g. Apache’s httpd\textsuperscript{11}, Wine\textsuperscript{12}, Sun’s JDK 1.6.0\textsuperscript{13}, Columba\textsuperscript{14}) or commercial programs to evaluate AAITs. While large open-source programs provide confidence and scale, the size of the sample evaluated (one to three programs) is a threat to external validity (e.g. the generalization of the results). Additional studies and subjects increase the generalization of experimental results [40]. Commercial examples show scalability of the technique in an industrial setting at the cost of repeatability and comparison.

These current benchmarks were insufficient for evaluating AAITs for several reasons. First, current benchmarks were lacking a detailed, repeatable process for use and evaluation of AAITs. Additionally, the current benchmarks were mostly for the C/C++ programming languages. Finally, alert classification and prioritization research, especially adaptive AAITs, requires the removal of anomalies by a researcher unfamiliar with the program, which is costly for large projects with a large number of alerts. Therefore, we want to create a benchmark of relatively small, real, and anomalous Java programs from a variety of domains.

\textsuperscript{11} http://httpd.apache.org/

\textsuperscript{12} http://www.winehq.org/

\textsuperscript{13} http://java.sun.com/javase/

\textsuperscript{14} http://columba.sourceforge.net/
3.2 FAULTBENCH Process

The goal of FAULTBENCH is to create a (1) suite of subject programs and alert oracles and (2) repeatable procedures for evaluation of actionable alert identification techniques. The FAULTBENCH process consists of two steps: 1) data collection for each subject and 2) evaluation of AAITs. There are four steps of data collection required to gather all of the defined artifact characteristics: 1) generate the subject revision history; 2) the subject build process; 3) alert classification; and 4) artifact characteristic generation. The evaluation metrics (Section 3.3.3) are used in model evaluation (Section 3.2.5). The FAULTBENCH v0.3 process is automated via batch scripts and Java programs. These programs are available at http://agile.csc.ncsu.edu/faultbench.

3.2.1 Generate Subject Revision History

Large software projects typically record software changes in a source code repository, like CVS\textsuperscript{15} or SVN\textsuperscript{16}. Mining a subject’s history through a source code repository or previous releases provides data for predicting actionable and unactionable alerts [19, 26, 27, 41, 48]. The data of interest are the sets of changes made to the source code and the reasons for those changes. A group of source code files that changed together is called a revision, and each revision in the history of the project should be evaluated to understand how alerts have changed over time. For projects with a large revision history, using a subset of revisions can reduce the analysis time.

\textsuperscript{15}CVS stands for concurrent versions system: http://www.nongnu.org/cvs/.

\textsuperscript{16}SVN, or Subversion, is a source code repository: http://subversion.tigris.org/.
3.2.2. Subject Build Process

For each revision in the full or subset revision history, we checked out and built the subject program. Building a subject program may require additional projects and libraries. If the subject program does not build, we move on to the next revision. Subjects that do not build provide inconsistent static analysis data. After building the project(s), we gather size and complexity metrics and static analysis alerts using tools appropriate to the programming language and environment.

3.2.3. Alert Oracle Creation

An alert’s actual classification from the alert history provides an oracle for AAIT. The alert history is the set of all alerts reported by ASA over the entire history of the project. The alert history for a project is built by comparing alerts between two subsequent revisions [19]. An alert is identified by the project name, package name, file name, method signature, alert type, and one of either an ASA generated unique identifier or the line number of the alert. An alert is closed and a new alert is opened if both the unique identifier and the line number change. If an alert does not have a unique identifier generated by the ASA tool, then the alert is tracked by the line number. If the line number changes, then the alert is closed and a new alert is opened. Alerts sharing the same identifying details within the same revision are considered as the same alert within the revision.

The alert history identifies closed or suppressed alerts. These closed or suppressed alerts serve as oracles for applying an AAIT. Figure 3.1 presents a state chart of an alert’s lifecycle. An alert is opened if the alert is not in any of the prior revisions [18, 19]. An alert
closure occurs when the alert was in a prior revision, but is not reported in a later revision [18, 19]. An alert is reopened if the alert was closed in a prior revision and is reported in a later revision. An alert is suppressed by identifying the alert in a filter file or through a code annotation. An alert is in the deleted state when the alert was in a file that was deleted. We do not want to consider alerts in deleted files because the alert closures are not associated with fixing the alerts.

![Figure 3.1. An alert’s lifecycle.](image)

Figure 3.1. An alert’s lifecycle. An alert is in the open state when first created. The alert is closed when the alert is no longer reported by ASA. If the alert is later reported by ASA, then the alert is reopened. An alert associated with a filter file or a suppression annotation in the source code is filtered. An alert in a file deleted during a revision is in the deleted state.

The classifications of alerts that remain open at the last revision of the source code are unknown. There are two possibilities of classification for these alerts. The first is to have a
developer inspect some or all of the open alerts and determine if the alert is actionable or unactionable. By inspecting all of the alerts there is a full oracle. The other option is to classify all of the open alerts as unactionable. The reasoning is that if developers have not fixed the anomaly associated with the alert during the history of the project, the alert may not be important.

### 3.2.4 Artifact Characteristic Generation

*Artifact characteristics* are information about the alerts and the surrounding source code that may be predictive of actionable and unactionable alerts [19]. Because we consider each distinct alert individually, the artifact characteristics are specific to that alert. However, similar alerts may have the same values for an artifact characteristic (e.g. alerts opened during the same revision will have the same *alerts for revision* value). The details about the artifact characteristics included in FAULTBENCH are presented in Chapter 5.

### 3.2.5 AAIT Evaluation Process

We considered five treatments when predicting actionable alerts for each of the AAITs. Four of the five treatments simulate how an AAIT would be used in practice, where the state of alerts at a specific revision are used to train a model and the model is applied to alerts after the revision. These training sets use a percentage of the revisions as training revisions and the remaining revision as test. The training set contains alerts opened before and not closed due to file deletions on or before the cutoff revision. The test set contains alerts closed after the cutoff revision or open alerts. The train and test sets do contain overlapping alerts: those alerts open at the cutoff revision. The fifth treatment uses all of the alerts to build a model
and evaluates the model on all of the alerts. For AAIT that incorporate randomness into the model building process, cross-fold validation or multiple runs of the AAIT can increase confidence in the generated AAIT.

3.3 Definition of FAULTBENCH

We define FAULTBENCH in terms of the three components presented by Sim et al. [42]: motivating comparison, task sample, and evaluation measures.

3.3.1 Motivating Comparison

The motivating comparison advocated by Sim et al. [42] describes why the results of comparing two tools or techniques are important for furthering the research surrounding the comparison. The motivating comparison of FAULTBENCH is to evaluate and compare AAITs. Specifically, we can use FAULTBENCH to answer the following research questions:

- RQ1: How accurately does an AAIT predict actionable and unactionable alerts?
- RQ2: What is the rate of anomaly detection for an AAIT?
- RQ3: How does an AAIT compare with other AAITs?

3.3.2 Task Sample

The task sample is a representative sample of tests that AAITs should solve [42]. For FAULTBENCH v0.3, the task sample consists of (1) three real Java subject programs ranging from 355 – 15,516 LOC; and (2) the set of FINDBUGS [21] and CHECK ‘N’ CRASH [12] alerts and associated artifact characteristics identified as actionable and unactionable in the context of the subject program’s history (alert oracle). Programs and scripts to automate the creation
of the FAULTBENCH v0.3 task sample for the three subject programs are available on the
FAULTBENCH website\textsuperscript{17}.

The subject programs in the benchmark must meet the following criteria: open source;
small (less than 20 KLOC), of various domains, written in Java; and compliable with Java
1.4.2 and Java 1.5. Information about the subject programs used in the comparative study are
in Table 3.1. Every 25\textsuperscript{th} revision including the first revision and the last revision were
sampled to reduce the time required for the revision-by-revision analysis.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
 & jdome & runtime & logging \\
\hline
Domain & data format & software dev. & logging library \\
\hline
Size (LOC) & 9035-13146 & 2066-15516 & 355-1785 \\
\hline
Time Frame & 05/27/2000 & 05/05/2001 & 08/02/2001 \\
(mm/dd/yy) & 12/17/2008 & 08/07/2008 & 09/20/2008 \\
\hline
# of Revisions & 1168 & 1324 & 710 \\
\hline
# of Sampled Revisions & 48 & 44 & 48 \\
\hline
# Built Revisions & 30 & 48 & 42 \\
\hline
Total Alerts & 489 & 632 & 91 \\
\hline
Actionable Alerts & 200 & 549 & 31 \\
\hline
Unactionable Alerts & 254 & 41 & 36 \\
\hline
Deleted Alerts & 35 & 42 & 24 \\
\hline
\end{tabular}
\caption{FAULTBENCH v0.3 subject programs. The FAULTBENCH subject programs and
the descriptive information. The size range for each subject program is provided across the
revision history from the sampled revisions. The provided sizes do not correspond to the
earliest and latest revisions, but typically, the size of the program increased over time.}
\end{table}

\textsuperscript{17}FAULTBENCH may be found at: http://agile.csc.ncsu.edu/faultbench/.
First, the subject programs were compiled with Java 1.5 revision 17. If the programs built correctly, an open source metrics tool, JavaNCSS\(^{18}\), and FINDBUGS [21] were run on the subject program. Next, the subject was recompiled with Java 1.4.2 revision 18 because CHECK ‘N’ CRASH [12] requires that the subject programs are compiled using Java 1.4.2. CHECK ‘N’ CRASH was then run on the subject program.

JavaNCSS is an open source tool for generating source code metrics, specifically the KLOC count in non-commented source statements (NCSS), method and class counts, and cyclomatic complexity. FINDBUGS [21] is an open source ASA tool that generates alerts that match bug patterns. CHECK ‘N’ CRASH [12] generates JUnit test cases from the alerts generated by the ASA tool ESC/JAVA [14] and reports those test cases that fail to the developer. A later implementation of CHECK ‘N’ CRASH, DSD-CRASHER, which considers an invariant generation step before running ASA was not considered for the comparative evaluation. The metrics and FINDBUGS alerts were recorded in .xml files. The alerts generated by ESC/JAVA and the failing test cases for CHECK ‘N’ CRASH were recorded in text files. These files are parsed, the data are stored in a local database, and an alert history is created for each ASA. Generation of artifact characteristics for each alert finishes the FAULTBENCH data collection step. AAiT may be applied to the alert data and evaluation follows using the evaluation metrics listed in Section 3.3.3. The remainder of this section describes specific information about each of the FAULTBENCH subjects.

\(^{18}\) JavaNCSS may be found at: http://www.kclee.de/clemens/java/javancss/
3.3.2.1 jdom

The subject program jdom\textsuperscript{19} is an open source library for XML. The jdom code is kept in a CVS source code repository. The jdom project consists of three sub-projects, all of which were considered for the comparative evaluation: jdom, jdom-contrib, and jdom-test. For the remainder of this dissertation, the three projects will be referred to collectively as, jdom. All projects are built using ant\textsuperscript{20}.

3.3.2.2 runtime

The subject program runtime is a core plug-in of the Eclipse\textsuperscript{21} integrated development environment. Runtime has a CVS repository and up to 12 other plug-ins are required to build runtime. Only alerts generated for the runtime plug-in were considered in the analysis. The build step consists of a headless Eclipse build. CHECK ‘N’ CRASH was not run on runtime due to inconsistent library usage throughout the revision history.

3.3.2.3 logging

The subject program logging is a Java logging library that is part of the Apache commons\textsuperscript{22}. The logging program is maintained in a SVN source code repository. All

\textsuperscript{19} jdom may be found at: http://www.jdom.org/.

\textsuperscript{20} Ant is a build library for Java programs: http://ant.apache.org/.

\textsuperscript{21} Eclipse is an open source integrated development environment that may be found at: http://www.eclipse.org/

\textsuperscript{22} Apache Commons is a top-level Apache project containing common libraries. Logging may be found at: http://commons.apache.org/logging/.
libraries required to build the logging program were obtained through Maven\textsuperscript{23} for each build and the project was built using ant.

\subsection*{3.3.3 Evaluation Measures}

FAULTBENCH evaluates AAITs. The key metrics associated with alert classification are below:

- **True positive classification** (TP): classifying an alert as actionable when the alert is actionable.
- **True negative classification** (TN): classifying an alert as unactionable when the alert is unactionable.
- **False positive classification** (FP): classifying an alert as actionable when the alert is actually unactionable.
- **False negative classification** (FN): classifying an alert as unactionable when the alert is actually actionable.

We are focusing on the classification of alerts identified by the static analysis tool; therefore, we are not considering software anomalies not found by static analysis tools. Figure 3.2 is a classification table.

\footnote{Maven is a build management tool: http://maven.apache.org/}
Anomalies are observed

<table>
<thead>
<tr>
<th>Model predicts alerts</th>
<th>Actionable</th>
<th>Unactionable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actionable</td>
<td>True positive (TP)</td>
<td>False positive (FP)</td>
</tr>
<tr>
<td>Unactionable</td>
<td>False negative (FN)</td>
<td>True negative (TN)</td>
</tr>
</tbody>
</table>

Figure 3.2. Classification table (adapted from Zimmerman et al. [52]). The rows contain the predicted classification and the columns contain the observed classification. The cells shaded in grey are the TP and TN classifications that we want to obtain.

The following metrics [27, 48, 49, 52] are used to evaluate the classification of static analysis alerts:

- **Precision** [49]: the proportion of correctly classified anomalies (TP) out of all alerts predicted as anomalies (TP + FP). The precision calculation is presented in Equation 1.

\[
precision = \frac{TP}{TP + FP}
\]  

(1)

- **Recall (also called True Positive Rate or Sensitivity)** [49]: the proportion of correctly classified anomalies (TP) out of all possible anomalies (TP + FN). The recall calculation is presented in Equation 2.

\[
recall = \frac{TP}{TP + FN}
\]  

(2)

- **Accuracy** [49]: the proportion of correct classifications (TP + TN) out of all classifications (TP + TN + FP + FN). The accuracy calculation is presented in Equation 3. Additionally, the confidence interval, the interval containing the actual population accuracy with some level of confidence [44], may be obtained when the train and test
evaluation is used. The test set of alerts is considered to have a Bernoulli distribution [49].

\[
\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}
\]  

(3)

Alert prioritization AAITs can be evaluated using classification metrics, discussed above, if a threshold is specified that divides the alerts into actionable and unactionable sets. The following performance measures are used to evaluate AAITs in FAULTBENCH.

- **Area Under the Curve (AUC):** a measure of the area under the graph of the number or percentage of actionable alerts identified over time. The AUC may be measured for many graphs such as the receiver operator characteristic (ROC) curve [49], which plots the percentage of true positives against the percentage of false positives at each alert inspection. The confidence interval is an interval containing the sample AUC that is expected to contain the actual AUC with some level of confidence [44]. The confidence interval for the AUC when measured by the ROC curve is obtained by following the procedure outlined in Hanley and McNeil [15]. The ROC curve measures the probability that an alert selected from the set of actionable alerts a ranked higher than an alert selected from the set of unactionable alerts [15]. Hanley and McNeil use the relationship between the AUC for a ROC curve and the Wilcoxon (Mann-Whitney) statistic to calculate a conservative standard error for data classified categorically. I have utilized their technique to generate the ACU for ROC curves in the area of actionable alert prediction. There are two exceptions for using this technique. The first is when the number of misclassified pairs is less than five [15]. Similarly, to the binomial
distribution, when there are very few failures, the exact, left 1-sided, confidence interval is obtained [1, 15]. The second exception is due to the AUC value used. Hanley and McNeil [15] use the Wilcoxon statistic to calculate the AUC because the Wilcoxon statistic is equivalent to the AUC as calculated by the trapezoidal rule. The AUC calculated by the Wilcoxon statistic is less than the AUC calculated using a continuous method, like the AUC value calculated by Weka [49]. When calculating the standard error via the method of Hanley and McNeil [15], the AUC generated by Weka [49] was used rather than the AUC as calculated by the Wilcoxon statistic. If the AUC squared was greater than two intermediate values in the standard error calculation, the standard error was unable to be calculated. In those instances, even if the number of misclassified pairs was greater than five, the exact, left 1-sided, binomial confidence interval was used.

- **Number of Unactionable Alerts Identified before 50% Percentage of Actionable Alerts (UA50) [30, 31]:** the number of unactionable alerts the prioritization technique identifies before 50% percent of actionable alerts are identified by the technique. Ideally, an AAIT should identify all actionable alerts before all unactionable alerts.

- **Inspections before First Actionable Alert (IFAA) [30, 31]:** the number of unactionable alerts that are inspected before the first actionable alert is identified by a prioritization technique.

- **Number of Alerts Inspected before All Actionable Alerts Identified (IAAA) [30, 31]:** the number of alert inspections required to identify all actionable alerts.
3.4 Desiderata for Benchmarks

Sim et al. [42] describe seven properties of successful benchmarks: accessibility, affordability, clarity, relevance, solvability, portability, and scalability. Lu et al. [32] also provide five benchmark selection criteria: representative, diverse, portability, accessibility, and fairness. The following subsections describe how FAULTBENCH meets these desiderata:

- **Accessibility:** A benchmark should be easy to obtain and use. Each of the FAULTBENCH subjects is available online through various open source licenses. The subject programs, generated alerts, and evaluation materials related to FAULTBENCH are publicly available at http://agile.csc.ncsu.edu/faultbench.

- **Affordability:** A benchmark’s cost (e.g. human, software, and hardware resources) should be comparable to the value of the results. Data collection for FAULTBENCH depends on the number of revisions analyzed and the time required to build and run ASA on the subject program. The data collection process could be parallelized. Additional time is required to generate the alert history and artifact characteristics. Once the data is collected, AAIT evaluation can occur, and the time required is dependent on the AAIT.

- **Clarity:** A benchmark’s documentation should be clear and concise. The FAULTBENCH documentation is provided at http://agile.csc.ncsu.edu/faultbench for evaluation and comparison of other FP mitigation techniques to ensure repeatability and disclosure.

- **Relevance/Representative:** A benchmark must contain representative subjects and performance measures related to the motivating comparison. FAULTBENCH contains Java programs from various domains created by developers of varying levels of experience.
The performance measures are standard in the area of data mining [49], software anomaly detection [52], and AAIT evaluation [48].

- **Solvability**: Completing the task sample and obtaining correct metrics is not difficult. The task samples vary in size. Additionally, an analysis program is provided as part of the benchmark materials.

- **Portability**: A benchmark should be useable by different AAITs without bias. The task sample consists of stand-alone Java projects containing required libraries. Use of the Java language assumes platform portability.

- **Scalability/Fairness**: A benchmark should be scalable to varying AAITs and not have bias towards a specific technique. Currently, FAULTBENCH contains Java subject programs and can only evaluate AAITs on alerts generated by Java static analysis tools. FAULTBENCH supplements other benchmarks in C and C++ and expansion of the benchmark is encouraged.

### 3.5 Threats to Validity

We consider three threats to validity for our work: construct validity, internal validity, and external validity. The threat to construct validity is in the measurement and calculations for the artifact characteristics. The calculations are explained in Chapter 5, and a more detailed explanation of most of the artifact characteristics may be found in [17].

Internal validity concerns potential sources of bias. One concern is the tools used to automate the generation of the alert history, artifact characteristics, and evaluation metrics for AAIT. The alerts and metrics that serve as inputs to the artifact characteristics were
generated via batch scripts\textsuperscript{24} that downloaded the source code for each revision, built the project, and then ran FINDBUGS, CHECK ‘N’ CRASH, and JavaNCSS on the code. A Java program parsed the alerts and metrics, generated the alert history, and calculated the artifact characteristics for each alert in a project. The build scripts and evaluation programs were manually tested, while the alert history and artifact generation program was tested with automated unit tests. All of the programs may be found on the FAULTBENCH website. To reduce threats to internal validity when comparing AAITs, the same train and test sets were used for all treatments. Additionally, the artifact characteristics for all AAITs were generated before any of the AAITs were applied, except for the ATL AAIT which aggregates the alert lifetime of an alert type on a per file rather than per alert basis. Finally, threats to internal validity was reduced by generating the alert oracle via the alert history. Those alerts closed throughout the alert history were actionable alerts and unclosed alerts were unactionable alerts.

One of the goals of FAULTBENCH is to reduce threats to external validity, and increase the generalizability of our results, by providing a breadth of sample programs to evaluate AAITs. We used three of the subject programs to evaluate SAAI in Chapter 6 and for a comparative evaluation of SAAI with other AAIT from the literature in Chapter 7.

\textsuperscript{24} The batch scripts and other FAULTBENCH materials are available at http://agile.csc.ncsu.edu/faultbench.
4 Feasibility Study

The goal of my research is to reduce inspection time by accurately predicting actionable and unactionable alerts when using static analysis by creating and validating a systematic actionable alert identification model. As a first step at achieving this goal, the Adaptive Prioritization Model (APM) [18] AAIT that uses alert information and developer feedback for prioritizing ASA alerts was created and tested. APM gathers data from three sources: 1) the alerts generated by ASA; 2) developer feedback in the form of explicitly suppressing false positive alerts and implicitly closing alerts due to a software fix; and 3) historical data about the ranking factors. APM uses the following artifact characteristics to prioritize ASA alerts:

- Alert type accuracy (ATA): a value representing the homogeneity of alerts sharing the same alert type (e.g., null pointer and cast error); and

- Code locality (CL): a value representing the homogeneity of alerts in the same location of code (e.g., method, class, and source folder) [30].

The AWARE tool provides APM as an Eclipse plug-in by gathering ASA alerts, data on the developer's actions to suppress or close alerts, and historical information about the prioritization factors. AWARE presents the listing of alerts, prioritized using APM, to the developer on the continuum [-1, 1] in the AWARE View as seen in Figure 4.1. AWARE v1.7.x was used for the feasibility study, where the x refers to the specific version of APM used in the study.
Figure 4.1. AWARE screenshot. The AWARE view is in the bottom portion of the Eclipse interface. Alerts are also identified by FINDBUGS using the bug icon, which may be seen to the left of line 148 in the source code.

The feasibility study demonstrated that using artifact characteristics like the alert’s type and location are predictive of actionable alerts. However, the results do not provide the level of precision (greater than 70%) that has been observed in literature [41]. While developers may look at the alert’s type and location as part of the inspection process there may be additional characteristics about an alert that are also predictive. Additionally, modeling the developer’s decisions would require a non-linear model like a decision tree, which leads in a new direction of research.
4.1 Adaptive Prioritization Model Process

Adaptive prioritization model (APM) [20] adaptively prioritizes and classifies static analysis alerts by the likelihood an alert is an indication of an important anomaly. Alerts are prioritized on the continuum, [-1,1] where:

- A priority in [-1,0) implies the alert is likely a unactionable,
- A priority in (0,1] implies the alert is likely a actionable, and
- A priority of 0 means there is not enough information to determine if the alert is likely a actionable or unactionable.

APM reprioritizes alerts after each alert inspection by a developer [18, 20, 30]. The developer has two actions that lead to the reprioritization of alerts: closing the alert or suppressing the alert. When ASA no longer identifies an alert, usually after an alert fix, the alert is considered closed. An alert may be fixed by an action on the part of the developer at the alert location or surrounding code. An alert can also be fixed implicitly by related changes or fixes to code unrelated to the alert. Configuration changes of the ASA (e.g., the alert type is no longer selected) may also close alerts.

Alert suppression is an explicit action on the part of the developer to indicate that he or she does not want to see that particular alert in future ASA runs. A developer will suppress an alert he or she deems to be unactionable.

Alerts have artifact characteristics, which may demonstrate some causality with the likelihood an alert is actionable. The alert type [20, 27] and alert location [30] at the package, file, and method, are the artifact characteristics used in the current version of APM. There are
two important measures associated with the artifact characteristics: 1) the relative size of the alerts sharing the characteristic to the overall set of alerts and 2) the developers’ actions when inspecting alerts sharing a characteristic. These measures, *size context* and *developer context*, respectively, are presented in Section 4.1.1 and 4.1.2, below. The size context and developer context comprise the value for each artifact characteristic (Section 4.1.3), which may be combined into the overall APM model (Section 4.1.4). Section 4.1.5 describes APM’s limitations.

### 4.1.1 Size Context

The *size context* (SC) represents information about the size, in number, of the alerts sharing a characteristic relative to the total number of alerts generated for a program. Alerts sharing a characteristic tend to be homogeneous [20, 30], and by increasing the priority of large sets, we can quickly classify many alerts (similar to *information gain* in [30]). The size context is the number of alerts sharing a characteristic divided by the number of alerts for the project. The formula for calculating the size context is presented in Equation 4.

$$SC_c = \frac{\# alerts_c}{total\# alerts}$$  

### 4.1.2 Developer Context

The *developer context* (DC) represents information about actions the developer has taken to close and suppress alerts while using static analysis during development. We take advantage of homogeneous artifact characteristics [20, 30] to utilize the developer’s feedback about the alerts to predict the likelihood that other, similar alerts are anomalies. The development
context is the difference between closed and suppressed alerts divided by the number of inspected alerts as demonstrated in Equation 5.

\[
DC_c = \frac{\#\text{closed}_c - \#\text{suppressed}_c}{\#\text{closed}_c + \#\text{suppressed}_c}
\]  

(5)

4.1.3 Artifact Characteristics

The following describes how we calculate the relationship between alerts sharing the same characteristic like accuracy (ATA) and code locality (CL). The coefficients to the baseline (\(\beta_{BC}\)) and developer (\(\beta_{DC}\)) context have a value of 0.5 implying that the baseline and developer context contribute equally to an artifact characteristic calculation.

- **Alert Type Accuracy (ATA):** ATA is the likelihood an alert (a) is an anomaly based on the type of the alert (e.g. null pointer, unclosed stream, etc.) [27, 28]. ATA is the weighted combination of the baseline and developer context of the alert’s type. The ATA calculation is described in Equation 6.

\[
ATA(a) = (\beta_{SC} \times SC_{type}) + (\beta_{DC} \times DC_{type})
\]

(6)

- **Code Locality (CL):** CL is the likelihood an alert (a) is an anomaly based on the location of the alert (e.g. at the source folder, class, or method level). CL is the weighted combination of the baseline and developer context of the alert’s location. The contribution of each location is calculated by normalizing the counts of non-singleton source folder, methods, and classes from Table 2b of [30]. The coefficients for the contributions of the source folder, classes, and methods are 0.06, 0.25, and 0.69, respectively and are represented by the coefficients \(\gamma_{sf}\), \(\gamma_c\), and \(\gamma_m\). We are only interested
in the non-singleton groups of alerts sharing a characteristic because any action taken on an alert can be used to predict if the other alerts in the group are likely to be anomalies [30]. Singleton alerts do not provide any predictive data. The calculation for $CL$ is described in Equation 7.

$$CL(a) = \left( \beta_{sc} \ast \left( (\gamma_{sf} \ast SC_{sf}) + (\gamma_{sf} \ast SC_{sf}) + (\gamma_{sf} \ast SC_{sf}) \right) \right) + \left( \beta_{dc} \ast ((\gamma_{sf} \ast DC_{sf}) + (\gamma_{sf} \ast DC_{sf}) + (\gamma_{sf} \ast DC_{sf})) \right)$$  (7)

4.1.4 Adaptive Prioritization Models

The overall alert prioritization calculation is the combination of artifact characteristic calculations divided by the number of artifact characteristics. Three versions of APM are presented in Table 4.1. Three models are considered to compare how well each of the artifact characteristics performs individually and together.

<table>
<thead>
<tr>
<th>Treatment</th>
<th>Formula</th>
<th>AWARE Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATA</td>
<td>$R(a) = ATA(a)$</td>
<td>1.7.1.1</td>
</tr>
<tr>
<td>CL</td>
<td>$R(a) = CL(a)$</td>
<td>1.7.2.0</td>
</tr>
<tr>
<td>ATA + CL</td>
<td>$R(a) = \frac{ATA(a) + CL(a)}{2}$</td>
<td>1.7.3.0</td>
</tr>
</tbody>
</table>

4.1.5 APM Limitations

Similarly to [26, 27, 30], our prioritization technique works best when the groups of alerts sharing a characteristic of interest are fine-grained (e.g. many alert types and locations) and homogeneous. Alert type and code location may not be homogeneous, and additional
information about the alert or the surrounding code may provide additional information. Additionally, the relationship between an alert’s classification and the information about the alert may not be linear like the APM calculation. Other techniques for building models, as presented in Chapter 5, could provide a direction of research that would remove the homogeneity limitation from AAIT.

4.2 Case Study

For the feasibility case study, an earlier version of FAULTBENCH, version 0.1, was used. FAULTBENCH v0.1 has six subject programs and prioritized or classified alerts without use of the subject’s history. The following research questions were answered as part of the feasibility case study:

- [Q1]: Can APM improve the rate of anomaly detection when compared to the tool’s output?
- [Q2]: How does the rate of anomaly detection compare between APM versions?
- [Q3]: Can alert categorization correctly predict actionable and unactionable alerts?

The case study considers the three versions of APM proposed in Table 4.1 on FAULTBENCH v0.1, which contains six subject programs, listed in Table 4.2.
Table 4.2. FAULTBENCH V0.1 subject programs. Each of the subject programs is released under an open source license. Additionally, the domains of the subject programs vary to allow for generalization of the results.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Version</th>
<th>License</th>
<th>Domain</th>
<th># Dev</th>
<th># LoC</th>
<th># Alerts</th>
<th>Maturity</th>
</tr>
</thead>
<tbody>
<tr>
<td>csvobjects</td>
<td>0.5beta</td>
<td>GNU GPL</td>
<td>data format</td>
<td>1</td>
<td>1577</td>
<td>7</td>
<td>Production</td>
</tr>
<tr>
<td>importscrubber</td>
<td>1.4.3</td>
<td>Apache Software License</td>
<td>software dev</td>
<td>2</td>
<td>1653</td>
<td>35</td>
<td>Beta</td>
</tr>
<tr>
<td>itrust</td>
<td>Fall 2007</td>
<td>Educational</td>
<td>web</td>
<td>5</td>
<td>14120</td>
<td>110</td>
<td>Alpha</td>
</tr>
<tr>
<td>jbook</td>
<td>1.4</td>
<td>GNU GPL</td>
<td>educational</td>
<td>1</td>
<td>1276</td>
<td>110</td>
<td>Alpha</td>
</tr>
<tr>
<td>jdom</td>
<td>1.1</td>
<td>Apache-style</td>
<td>data format</td>
<td>3</td>
<td>8422</td>
<td>55</td>
<td>Production</td>
</tr>
<tr>
<td>org.eclipse.</td>
<td>3.3.1.1</td>
<td>Eclipse Public License</td>
<td>software dev</td>
<td>100</td>
<td>2791</td>
<td>98</td>
<td>Production</td>
</tr>
<tr>
<td>core.runtime</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.2.1 Study Setup

Alerts were prioritized via the AWARE [20] Eclipse plug-in. AWARE gathers static analysis alerts generated from FINDBUGS and prioritizes the alerts using one of the APM models presented in Table 4.1. AWARE tracks alert closures and suppressions by comparing the prior state of alerts with the set of alerts generated after each static analysis run. The closures and suppressions are used to modify the prioritization of the alerts. We used Eclipse version 3.3.1.1 for all of the benchmark subjects except iTrust. For iTrust, we used the Eclipse IDE for Java EE Developers version 3.3.1.1 because iTrust is a web application and the Java EE developers version of Eclipse provides a web development environment. Each version of AWARE contains one of the three versions of the APM AAIT. Table 4.1 also presents the AWARE version for each of the prioritization techniques.

Adaptive AAIT require an action based on the classification of the alert at the top of the prioritization. If the alert is actionable, then the alert is fixed in the source code, while unactionable alerts are explicitly suppressed. Therefore, all of the actionable alerts had
associated code changes for closure, and the alerts were fixed and suppressed by the researcher while evaluating the model in a post hoc study. The researcher emulates the actions a developer would take to inspect and act upon alerts to obtain data about the alert prioritization at each inspection for evaluation of the model. Additionally, experimental controls in the form OPTIMAL, RANDOM, and TOOL ordering were created.

The OPTIMAL ordering of alerts has all actionable alerts at the top of the alert list; therefore, there are (actionable!)*(unactionable!) optimal permutations. For FAULTBENCH V0.1 the OPTIMAL ordering is generated by a greedy analysis of the actionable alerts. Alerts are initially sorted hierarchically in the context of the subject program (e.g. by project, source folder, class, method, line number, alert type, and description), which provides a repeatable ordering for alerts. To reduce potential bias, prioritization techniques should use the same hierarchical alert ordering to break ties when alerts share the same priority. Alerts are first added to the OPTIMAL ordering by the number of actionable alerts that are closed when making an alert change. When two alerts close the same number of actionable alerts, first the number of unactionable alerts closed is a tiebreaker, followed by the hierarchical ordering of alerts. At a minimum, the optimal curve will fix one actionable alert at each inspection until all actionable alerts are fixed.

The TOOL ordering of alerts is created from the tool’s alert log information. The RANDOM ordering of alerts is generated via a random number generator\textsuperscript{25}. Cases where more than one alert is closed must be considered when creating the OPTIMAL, RANDOM, and TOOL ordering.

\textsuperscript{25} A random sequence generator may be found at http://random.org.
prioritization. The prioritization of an uninspected closed alert is a fraction of the number of alerts closed during an inspection. If there were three alerts (a, b, and c) closed at inspection 3, then the inspected alert (a) would have a rank of 3, the uninspected alert first in the ordered listing (b) would have a rank of 3.33 and other uninspected alert (c) would have a rank of 3.66.

4.2.2 Research Questions

FAULTBENCH V0.1 provides data to answer the following research questions:

• [Q1]: Can APM improve the rate of anomaly detection when compared to the tool’s output?

• [Q2]: How does the rate of anomaly detection compare between APM versions?

• [Q3]: Can alert categorization correctly predict actionable and unactionable alerts?

Question 1 and 2 are answered by using the *Spearman rank correlation* and the *area under the curve* metric for the anomaly detection curve, which measures the percent of anomalies detected at each inspection. Question 3 is answered using *precision*, *recall*, and *accuracy* metrics.

4.2.3 Q1: Improving Anomaly Detection Rate

We plot the cumulative percentage of anomalies detected against the number of inspections and measure the AUC to evaluate Question 1. Figure 4.2 provides an example of these plots for the jdom subject program. When actionable alerts are fixed, the percentage of detected anomalies increases. There are plateaus in the prioritization curve when an unactionable alert is suppressed at an inspection. A large plateau means there were a number of suppressions.
A good prioritization will minimize the large plateaus until most or all of the actionable alerts have been identified.

![Anomaly detection curves for jdom. Alert prioritization close to the optimal line are preferred and have a higher AUC.](image)

Table 4.3 presents the AUC for anomaly detection curves for each of the prioritization techniques and benchmark subjects. The first question compares alert prioritization techniques to the TOOL ordering of alerts. In the absence of prioritization, developers only have the static analysis tool’s output for investigation. If the tool’s ordering performs well, then alert prioritization is not needed. However, all prioritization techniques except on csvobjects and iTrust perform better than the tool ordering. On average, all
prioritization techniques have a larger AUC (53.94% - 72.57%) than the tool ordering (50.42%) of alerts.

Table 4.3. Area under the anomaly detection curve for prioritization techniques. AUCs close to the optimal AUC are preferred.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Optimal</th>
<th>Random</th>
<th>ATA</th>
<th>CL</th>
<th>ATA + CL</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>csvobjects</td>
<td>78.57%</td>
<td>59.52%</td>
<td>50.00%</td>
<td>21.43%</td>
<td>30.39%</td>
<td>54.76%</td>
</tr>
<tr>
<td>importsscrubber</td>
<td>84.29%</td>
<td>71.82%</td>
<td>66.10%</td>
<td>40.91%</td>
<td>66.62%</td>
<td>36.23%</td>
</tr>
<tr>
<td>iTrust</td>
<td>95.5%</td>
<td>48.91%</td>
<td>74.36%</td>
<td>68.09%</td>
<td>67.36%</td>
<td>75.09%</td>
</tr>
<tr>
<td>jbook</td>
<td>78.55%</td>
<td>49.83%</td>
<td>46.26%</td>
<td>62.57%</td>
<td>74.19%</td>
<td>39.87%</td>
</tr>
<tr>
<td>jdom</td>
<td>91.82%</td>
<td>71.66%</td>
<td>86.16%</td>
<td>63.54%</td>
<td>85.35%</td>
<td>46.89%</td>
</tr>
<tr>
<td>org.eclipse.core.runtime</td>
<td>96.81%</td>
<td>68.61%</td>
<td>82.53%</td>
<td>67.09%</td>
<td>82.78%</td>
<td>49.67%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>87.58%</strong></td>
<td><strong>61.73%</strong></td>
<td><strong>72.57%</strong></td>
<td><strong>53.94%</strong></td>
<td><strong>67.88%</strong></td>
<td><strong>50.42%</strong></td>
</tr>
</tbody>
</table>

Table 4.4 presents the Spearman rank correlation values between the APM models and OPTIMAL. A positive correlation implies that the specified prioritization is similar to the OPTIMAL prioritization while a negative correlation implies that the specified prioritization is opposite OPTIMAL. The closer the correlation is to 1 or -1, the stronger the match or opposition of the specified prioritization. Cells containing one star (*) have correlations significant at the 0.05 level, while cells containing two stars (**) have correlations significant at the 0.01 level.
Table 4.4. Spearman rank correlation. Correlations closer to 1 show that the prioritization of alerts generated by an AAIT are close to the optimal ordering of alerts.

<table>
<thead>
<tr>
<th></th>
<th>ATA</th>
<th>CL</th>
<th>ATA+CL</th>
<th>TOOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>csvobjects</td>
<td>0.321</td>
<td>-0.643</td>
<td>-0.393</td>
<td>0.607</td>
</tr>
<tr>
<td>importsrubber</td>
<td>0.512**</td>
<td>-0.026</td>
<td>0.238</td>
<td>0.203</td>
</tr>
<tr>
<td>iTrust</td>
<td>0.418**</td>
<td>0.264**</td>
<td>0.261**</td>
<td>0.772**</td>
</tr>
<tr>
<td>jbook</td>
<td>0.798**</td>
<td>0.389**</td>
<td>0.590**</td>
<td>-0.002</td>
</tr>
<tr>
<td>jdom</td>
<td>0.675**</td>
<td>0.288*</td>
<td>0.457**</td>
<td>0.724**</td>
</tr>
<tr>
<td>org.eclipse.core.runtime</td>
<td>0.395**</td>
<td>0.325**</td>
<td>0.246*</td>
<td>0.691**</td>
</tr>
</tbody>
</table>

The TOOL experimental control prioritization has a moderately strong correlation (e.g. correlation value > 0.600) with OPTIMAL for four of the subject programs. The strong correlation is likely due to a similar ordering of the unactionable alerts, and is not necessarily an indication of the anomaly detection capabilities of the TOOL ordering. For example, the TOOL ordering for jdom has a correlation of 0.724; however, the area under the anomaly detection curve for TOOL is at least 20% less than ATA, CL, and ATA+CL as seen in Table 4.3.

4.2.4 Q2: Comparing Prioritizations

Table 4.3 presents the area under the anomaly detection curve metrics for the APM models and the control prioritizations for the FAULTBENCH V0.1 subjects. The average area under the optimal curve is 87.58%. The ATA prioritization is closer to OPTIMAL than the CL prioritization. Additionally, the average ATA area is 18.63% larger than CL’s average area. ATA+CL splits the difference between ATA’s and CL’s prioritization.

Table 4.4 presents the Spearman rank correlation values between the APM models and OPTIMAL. The correlations between the APM models and OPTIMAL are similar to the patterns observed in the area under the curve measurement in Table 4.3. However, the ATA
correlation with OPTIMAL is typically stronger, indicating that ATA is the better prioritization technique.

4.2.5 Q3: Categorizing Alerts

Table 4.5 presents the average precision, recall, and accuracy metrics before each inspection for each APM model. We only consider the precision, recall, and accuracy metrics for uninspected alerts because we are trying to predict if the uninspected alerts are actionable or unactionable. A priority greater than 0 is a prediction that the alert is actionable while a priority less than 0 is a prediction that an alert is unactionable. We then assess the prioritization’s classification using the alert oracle and the predicted classification.

Table 4.5. Average precision, recall, and accuracy metrics of un-inspected alerts at each inspection. The precision, recall, and accuracy are calculated for each alert inspection for a subject program, and the average values are presented. Precision, recall, and accuracy values close to 1 are preferred.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Average Precision</th>
<th>Average Recall</th>
<th>Average Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ATA</td>
<td>CL</td>
<td>ATA +CL</td>
</tr>
<tr>
<td>csvobjects</td>
<td>0.32</td>
<td>0.50</td>
<td>0.39</td>
</tr>
<tr>
<td>import-scrubber</td>
<td>0.34</td>
<td>0.20</td>
<td>0.18</td>
</tr>
<tr>
<td>iTrust</td>
<td>0.05</td>
<td>0.02</td>
<td>0.05</td>
</tr>
<tr>
<td>jbook</td>
<td>0.22</td>
<td>0.27</td>
<td>0.23</td>
</tr>
<tr>
<td>jdom</td>
<td>0.06</td>
<td>0.09</td>
<td>0.06</td>
</tr>
<tr>
<td>org.eclipse.core.runtime</td>
<td>0.05</td>
<td>0.04</td>
<td>0.03</td>
</tr>
<tr>
<td>Average</td>
<td>0.17</td>
<td>0.19</td>
<td>0.16</td>
</tr>
</tbody>
</table>

If the alert falls in the TP or TN categories, the prioritization correctly classified the alert as actionable or unactionable. As we learn more about the alerts from the developers, we expect the precision, recall, and accuracy to increase; however, the precision and recall
tended to be 0 for many of the inspections, which leads to a low average precision and recall. After all actionable alerts were identified; there was no longer a numerator in the precision and recall equations leading to a value of 0 for some inspections. The average accuracy is a better measure of how the classification techniques performed. ATA had the best average accuracy, and correctly predicted if an alert is actionable or unactionable 76% of the time.

4.3 Discussion

APM provides a prioritization of alerts with an accuracy of at most 76% and precision in the range of 0.16-0.19. Precision of over 0.75 has been observed in literature [41]. Since the goal of this research is to reduce the inspection of unactionable alerts, minimizing the reported FPs (as measured by the precision) is key to a successful model.

The artifact characteristics chosen for APM are from knowledge about ASA. Developers inspect alerts by first understanding what the potential problem is, represented by the alert type, and by then inspecting the code containing the alert, the alert’s location. However, this may not be the only information a developer may utilize, and there may be other predictive information that the developer may not consider. Therefore, a systematic approach to identifying the important artifact characteristics can provide additional insight into the actionable alert identification problem. Using machine learning algorithms can also provide non-linear relationships about the alerts that may be more predictive of actionable alerts than the linear model presented in APM.
5 Systematic Actionable Alert Identification Model

Building Process

APM and other current models [18, 20, 26, 27, 30, 31, 48] do not systematically choose potential independent variables or models when creating AAIT. Additionally, an AAIT that works for one program may be ineffective on other programs. The goal of my research is to reduce inspection time by accurately predicting actionable and unactionable alerts when using static analysis by creating and validating a systematic actionable alert identification model. Machine learning allows for systematically finding patterns in data [49]. Witten and Frank [49] outline a strategy for using machine learning to identify predictive attributes and selecting machine learning models, called classifiers. From Witten and Frank’s strategy, we can use machine learning to create an AAIT and evaluate the research hypotheses by systematically identifying, if there are any, predictive artifact characteristics and the most predictive machine learning models. The proposed technique is called Systematic Actionable Alert Identification (SAAI).

The following hypotheses are under evaluation:

- Hypothesis 1: The artifact characteristics of an alert and the surrounding source code are predictive of the actionability of an alert.
- Hypothesis 2: A systematic actionable alert identification technique using machine learning can accurately identify actionable alerts.
- Hypothesis 3: A systematic actionable alert identification technique using machine learning is project specific.
Machine learning allows for the selection of predictive attributes out of a larger set of attributes about the phenomena under study. For AAIT, the alert’s classification is the phenomena of interest, and the potentially predictive attributes are information about the alerts and surrounding code, called *artifact characteristics*. Related work in AAITs and my knowledge about ASA have lead to the collection of 58 artifact characteristics that may be predictive of actionable alerts. Hypothesis 1 states that artifact characteristics of an alert and the surrounding code are predictive of the actionability of an alert. Using attribute selection, the predictive artifact characteristics for a subject program can be identified. The artifact characteristics considered in SAAI are presented in Section 5.1.

Machine learning algorithms create *classifiers*, using selected sets of artifact characteristics and alerts with a known classification as actionable or unactionable. Hypothesis 2 states that a systematic AAIT using machine learning can accurately identify actionable alerts. The SAAI technique creates classifiers by training them on subsets of the alerts generated over the history of a project. The accuracy of the classifiers is evaluated in comparison to each other and other AAIT from literature.

The scope of alerts used to create an AAIT have varied in literature. Ruthruff, et al. [41] created a prioritization model using alerts from multiple projects within a company. Kim and Ernst [26] built prioritization models on two projects, and found low correlations between the models for the two project. Kremenek, et al. [30] investigated the prioritization of each detector separately. Hypothesis 3 investigates this inconsistency in scope by positing that systematic AAITs using machine learning are project specific, implying that a model
generated for one project is not appropriate for usage on another project. For hypothesis 3, we are interested in applying the selected artifact characteristics and machine learning classifier selected for one project to another project.

SAA1 uses machine learning to predict actionable and unactionable alerts. The input to SAA1 is a set of inspected alerts, their artifact characteristics, and their classification as either actionable or unactionable. These alerts build the resulting models, and the most predictive model is selected to supplement the ASA used. SAA1 of four steps:

1) Step One: data collection;
2) Step Two: artifact characteristics selection;
3) Step Three: machine learning model creation; and
4) Step Four: model evaluation.

Weka [49] is an open source tool and library, developed by Witten and Frank, that supports machine learning tasks. The SAA1 process outlined is supported by Weka. The remainder of this section describes the four steps of the SAA1 process.

5.1 Step One: Data Collection

SAA1 predicts actionable and unactionable alerts; therefore, the dependent variable for SAA1 is the alert’s classification of actionable or unactionable. The independent variables, or predictors, are the artifact characteristics associated with the alerts, which are described in the subsections below. The input to SAA1 consists of the independent and dependent variables as the set of alerts, their artifact characteristics, and their classification. Alerts closed via a
source code change are given an actionable classification and alerts suppressed or ignored by developers are given an unactionable classification.

SAAI considers the alert history for a project, similar to [26, 27, 41, 48]. The data collection process is the same as the data collection process for FAULTBENCH [18] described in Section 3.2. The data collection process gathers alerts over the project history; generates the alert history and creates the alert oracle; and generates the artifact characteristics for each of the alerts in the alert history. The following subsections describe the artifact characteristics generated during the data collection process. A more detailed discussion of the artifact characteristic generation process may be found in [17]. When artifact characteristics are inspired from related work, the citation to the appropriate paper is provided. Artifact characteristics that created as part of this research have a justification for why they may be predictive of actionable or unactionable alerts.

5.1.1 Alert Identifiers and Alert History

A static analysis tool generates alert identifiers (the first eight characteristics below) at alert creation, and the alert history (the last characteristic below) is generated via a program\textsuperscript{26} that compares the alerts between software revisions.

- **Project name:** [19].
- **Package name:** the package name could be generalized to the folder containing a source file [18-20, 30].
- **File name:** [30].

\textsuperscript{26} The program used to generate the number of alert modifications as part of the alert history is provided on the FAULTBENCH website: http://agile.csc.ncsu.edu/faultbench.

63
• **Method signature**: name and parameter types of the method or function containing the alert [18-20, 30]. An alert may not have an enclosing method.

• **Alert type**: the type of potential anomaly (e.g. null pointer, etc.) [18-20, 26, 27, 41, 48].

• **Alert category**: a high level categorization of alert types (e.g. security, correctness) [19, 41].

• **Priority**: the priority of the alert defined by a static analysis tool [19, 26, 41].

• **File extension**: [19, 41].

• **Line number**: the line number of the alert in the source code.

• **Marker type**: the ASA tool that generated the alert.

• **Test file**: the file that contains the automated unit tests generated for an alert [12].

• **Number of alert modifications**: the number of times an alert’s line number or priority has been changed over the alert’s lifetime[19].

### 5.1.2. Software Metrics

Nagappan et al. [34] show that code complexity metrics correlate with failure-prone modules. Additionally, Bell et al. [4, 37] have utilized code size metrics to predict fault counts. Actionable alerts could be considered faults; therefore, software metrics could be predictive of actionable alerts. For characteristics containing several granularities (e.g. method, file, etc.) the metric is collected for each level. The software metric characteristics used for this study are below:

• **Size**: the number of non-comment source statements (NCSS) within the method, file [41], or package declaration containing the alert [19].
• **Number of methods:** collected at the class and package levels [19].

• **Number of classes:** collected at the file (e.g. the class and any inner classes) and package levels [19].

• **Cyclomatic complexity:** measures the number of paths through a method [39] containing an alert [19].

• **Alert depth in file:** measures the depth of the alert as a percentage of file size [41].

### 5.1.3 Source Code History

The models by Williams and Hollingsworth [48], Kim and Ernst [26, 27], and Ruthruff et al. [41] use artifact characteristics obtained from a project’s source code repository to predict actionable alerts. Revisions are used instead of dates to track time. A *revision* is a set of changes committed to the source code repository together. For all of the characteristics listed below, except developers, the revision number is recorded. Below are the source code history characteristics:

• **Alert open revision** [19, 26].

• **Developers:** set of developers who made changes to the file containing an alert between the alert’s open revision and the prior revision analyzed [19, 26].

• **File creation revision:** [19, 41].

• **File deletion revision:** Alerts closed due to a file deletion are not considered actionable [19, 26, 27, 41]. These alerts are removed if the file deletion revision is less than or equal to the closure revision.
• **Latest modification revision:** last modification to a file, package, or project on or before the last revision [19].

• **Close revision type:** the type of code change reported in the reposition – fault or non-fault change [27].

### 5.1.4 Source Code Churn

Source code churn measures the amount of change made to a file, package, or project over time [41]. Each of the general code churn metrics are measured between the prior analyzed revision and the open revision for the alert. The churn metrics are measured for the file, package, and project that contain the alert. The source code churn characteristics are below:

- **Added lines** [19, 41].
- **Deleted lines** [19, 41].
- **Modified lines** [41].
- **Growth:** the difference between added and deleted lines [19, 41].
- **Total modified lines:** the sum of added and deleted lines [19, 41].
- **Percent modified lines:** percent of total modified lines out of all modified lines for the project [19, 41].

### 5.1.5 Aggregate Characteristics

Aggregate candidate artifact characteristics come from the above artifact characteristics and provide a deeper understanding about an alert. Prior models measure age in days [26, 41]. Instead, we measure age as the number of revisions between two revisions. Using revisions
is still a measure of time, but also provides a measure of work. The aggregate characteristics are below:

- **Total alerts for revision**: number of alerts identified on or before an alert’s open revision [19]. If an alert is opened and 10 other alerts already exist, the number of alerts for the revision is 11. The alert count also contains alerts that were deleted during the history of the project.

- **Total open alerts for revision**: number of open alerts identified on or before an alert’s open revision [19]. Continuing with the above example, suppose that three of the 10 existing alerts are closed. Therefore, the number of open alerts for the revision is eight. The alert count also contains alerts that were deleted during the history of the project.

- **Alert lifetime**: the age of the alert [19, 26]. For a closed alert, the alert lifetime is the difference between the close and open revisions. Otherwise, the lifetime is the difference between the last revision in the study and the open revision.

- **File age**: the age of the file [19, 41]. For a deleted file, the file age is the difference between the deletion and creation revision. Otherwise, the file age is the difference between the last revision in the study and the file creation revision.

- **Alerts for an artifact**: the number of alerts in the method [18, 20], file [18, 20, 41], package [18, 20], or project [41] containing an alert across all revisions [19].

- **Staleness**: amount of time between last revision and the last change of the file, package, or project [19, 41].
5.2 Step Two: Artifact Characteristic Selection

Artifact characteristic selection is important in machine learning because redundant and irrelevant characteristics reduce classifier performance [49]. Additionally, there could be diminishing returns when an artifact characteristic contributes so insignificantly to a model that the time for collection of an artifact characteristic outweighs the small increase in predictive power. We want to choose the best subset of candidate artifact characteristic to use when classifying alerts as actionable or not. Attribute selection algorithms identify the artifact characteristics that are associated with alert history’s classification, and any algorithms appropriate to the data under analysis can be applied.

Weka [49] supplies attribute selection algorithms for identifying the most predictive attributes in a data set. Attribute selection consists of a search strategy and an evaluation method for finding and evaluating the artifact characteristics in the alert history. Sixteen search strategy and evaluation method combinations are considered for generating sets of predictive artifact characteristics. As discussed in Section 5.2.1, the search strategies used are Best First, Greedy Stepwise, Rank Search with Information Gain, and Rank Search with Gain Ratio. As discussed in Section 5.2.2, the evaluation methods used are Cfs Subset Evaluation, Classifier Subset Evaluation, Consistency Subset Evaluation, and Wrapper Subset Evaluation. The default values for each search strategy and evaluation method were used to generate the artifact characteristic subsets.
5.2.1 Search Strategies

Search strategies are methods for searching the attribute subset space. Subsets of attributes are then evaluated using the associated evaluation method. Best first searches the attribute subset space starting from either no attributes (forward searching) or from all attributes (backwards searching) and evaluates attribute subsets with the evaluation method [49]. The search backtracks when one or more attribute subsets are not demonstrating predictive improvements [49]. Greedy stepwise searches the attribute subset space similarly to best first, but does not backtrack [49].

Rank search orders the attributes from best to worst from the calculation of a single-attribute evaluator [49]. The attribute subset space is then explored using a forward search strategy whereby the attribute subset of only the best artifact characteristics is considered followed by the two best artifact characteristics, etc. [49]. Information gain measures the amount of information that an attribute contains in predicting actionable and unactionable alerts by considering the alert classification mixture of the value subsets of the attribute [49]. Any numeric attributes are discretized to form categories of numeric results. Gain ratio is a similar calculation to information gain, but considers the sizes of the possible subsets for an artifact characteristic [49]. Therefore, artifact characteristics with many possible values and few alerts with those values are not considered as more important than an artifact characteristic that does well at classifying alerts from a few possible values [49].
5.2.2 Evaluation Methods

Evaluation methods appraise attribute subsets by calculating a numerical value representing how predictive a set of artifact characteristics are [49]; in our case we examine how predicative the artifact characteristics are of actionable and unactionable alerts. Cfs subset evaluation filters the artifact characteristics to find highly predictive artifact characteristics with low intercorrelation [49]. Consistency subset evaluation identifies attribute subsets where the classifications of the alerts with those characteristics are either mostly or all actionable or mostly or all unactionable [49].

Classifier subset evaluation uses a machine learner for appraising attribute subsets [49]. Wrapper subset evaluation also uses a machine learner for appraising attribute subsets, but also incorporates cross-validation [49]. For this research, we used J4.8, which is an open implementation of C4.5 decision tree machine learner in classifier subset evaluation and wrapper subset evaluation. The attribute subset will not increase the performance when applied to the J5.8 classifier for machine learning model creation, but the subset will help other machine learning classifiers, like nearest neighbor classifiers [49].

5.3 Step Three: Machine Learning Model Creation

Machine learning algorithms are applied to each of the selected artifact characteristic subsets to create candidate models using the SAAI process. Models may be cross validated using 10, 10-fold cross validations [49]. In cross validation, the set of alerts are randomly separated into ten approximately equal sets, and nine of the sets train the model that is tested by the last set. Each of the ten sets is a test set, and the process is repeated ten times. SAAI models may
also be validated by separating the alerts into training and test sets from the revision history. The model is trained on the alert history between revision 1 and revision \( n-m \), where \( n \) is the number of revisions and \( m \) is the number of revisions in the test set. The trained model is evaluated on the test set of alerts in the last \( m \) revisions.

The remainder of this section describes the 16 different machine learning algorithms, in five high level categories, systematically evaluated when creating the SAAI AAIT and the justification for why the machine learning algorithm is appropriate for predicting actionable and unactionable alerts.

### 5.3.1 Classification Rules

Classification rules consist of one or more antecedents and a consequent [49]. Antecedents are a set of tests that if true predict the consequent. In our work, the consequent is the classification of an alert as actionable or unactionable. Classification rules provide an encapsulation of important information and are easy to interpret [49]. Classification rules may also mimic the decisions a developer may make when inspecting an alert, which makes these machine learners worthy of consideration. An example rule is \( \text{alertLifetime} > 474 \Rightarrow \text{classification} = \text{unactionable} \), which means if the alert lifetime for an alert in the test set is greater than 474 revisions, then the alert is predicted to be unactionable.

The *decision table* machine learner builds rules where the consequent is the majority classification for the antecedents [49]. The artifact characteristics that make up the antecedents for rules in a decision table come from a best first search of the artifact characteristics subsets. The *conjunctive rule* machine learner generates one rule whose
antecedent is created by calculating the information gain for artifact characteristics and finding the most predictive antecedent. Conjunctive rule provides a baseline of the minimum rule to classify alerts. If an alert does not meet the antecedent of the conjunctive rule, then the alert is predicted to be the default classification, which is typically the majority classification within all alerts not covered by the conjunctive rule [49].

The part machine learner creates rules from the J4.8 decision tree (discussed in Section 5.3.2) [49]. Ridor is an iterative process whereby a single rule is provided with exceptions [49]. The JRip machine learner is an implementation of the RIPPER machine learning algorithm [49]. RIPPER considers the possible classifications, starting with the classification with the fewest instances and generates rules for that classification and removes rules that are not predictive enough [49].

5.3.2 Decision Trees

Decision trees are a divide and conquer algorithm for predicting actionable and unactionable alerts [49]. Starting with the top node of a decision tree and following the paths provide a prediction for an alert. Like classification rules, decision trees can potentially model developer decisions when inspecting ASA alerts. J4.8 is an implementation of the C4.5 algorithm. The information gain or gain ratio metrics are used to decide which artifact characteristic to split on at each level of the tree [49]. J4.8 can handle both numeric and categorical data and the tree is minimized by utilizing pruning algorithms[49].

The REPTree machine learner builds a decision tree using information gain and is optimized for speed [49]. LMT is a decision tree with logistic regression equations at the leaf
nodes [49]. *ADTree* is optimized for dependent variables with two classifications, like actionable and unactionable alerts [49].

### 5.3.3 Functions

Functions are mathematical formulas used for predictions [49]. Mathematical models provide an easily implementable model and some indication into how artifact characteristics contribute to the overall prediction. *Simple logistic* predicts categorical variables and works best for artifact characteristics with no intercorrelation [49].

### 5.3.4 Nearest Neighbor

Nearest neighbor models is an instance-based machine learning algorithm, where inspected alerts close to an uninspected alert are used to predict the classification for the uninspected alert [49]. Additionally, nearest neighbor models and computationally less expensive than other machine learning models because the model is only run when classifying an instance [49]. Using nearby or similar inspected alerts to predict uninspected alerts is similar to the homogeneity premise proposed by Kremenek, et al. [30] and continued by Heckman and Williams [18, 20].

*IB1* uses the Euclidean distance to find the nearest inspected alert to an uninspected alert, and the uninspected alert is predicted to have the same classification as the inspected alert [49]. Euclidean distance is a linear measurement of distance [49]. If categorical variables are used, then differing values are given a distance of 1 while the same value is given a distance of 0 [49]. The distances of each artifact characteristic are normalized so that all artifact characteristics are considered equally [49]. *IBk* is the same as IB1, but considers k
nearest neighbors. The classification with the most instances is predicted for the uninspected alert. We consider k=2, which means that the two closest neighbors to an uninspected alert are considered.

KStar uses a series of transformations to change an uninspected alert into an inspected alert [49]. The probability of each possible transformation is used to order the inspected alerts, and the classification of the alert with the highest probability is given to the uninspected alert. LWL generates weights for each inspected instance of an alert from the distance from an uninspected alert [49]. The weights are used to build a Naïve Bayes model for classifying the uninspected alert [49].

5.3.5 Bayesian Networks

Bayesian networks are probabilistic models where the probabilities of a classification are generated by considering the likelihood of each classification for the selected artifact characteristics and normalizing the resulting likelihoods [49]. Numeric attributes are discretized. Naïve bayes calculates the probability an alert is actionable or unactionable using the above method where each artifact characteristic is considered to be “equally important and independent” [49]. Bayes net adds the additional assumption of no missing values in the training set of alerts. The process of artifact characteristic selection discussed in FAULTBENCH [18] ensures no missing values by providing a default of -1 for any values that cannot be calculated (e.g. the close revision for an alert that is still open the latest revision).
5.4. Step Four: Model Evaluation

The best SAAI model is selected by comparing the accuracy, precision, recall, AUC for the Receiver Operator Characteristic (ROC) curve, the f-measure, and the error rates. The best model had the highest accuracy followed by the highest precision and recall. If there were several models that had the same accuracy, precision, and recall, then the SAAI with the highest f-measure and lowest error is selected. Additional considerations for model selection are that the model does not predict only one classification, but instead utilizes at least one of the selected artifact characteristics.
6 SAAI Case Studies

SAAI is a systematic AAIT process for predicting actionable and unactionable alerts when using ASA. Evaluating SAAI on the subject programs in FAULTBENCH [18] v0.3 provides support in accepting or rejecting the following three hypotheses:

- **Hypothesis 1**: The artifact characteristics of an alert and the surrounding source code are predictive of the actionability of an alert.
- **Hypothesis 2**: A systematic actionable alert identification technique using machine learning can identify alerts with accuracy greater than 90%.
- **Hypothesis 3**: A systematic actionable alert identification technique using machine learning is project specific.

The AAIT evaluation process presented in Section 3.2.5 is used to evaluate SAAI on the FAULTBENCH subject programs. Five treatments are considered for evaluation of SAAI. Treatment 70 considers the alerts in the first 70% of revisions as the training set for the SAAI candidate models and the alerts in the remaining 30% of revisions are the test set. Treatment 80 and 90 are similar to treatment 70, but consider 80% and 90% of the revisions, respectively. Treatment 99 uses revisions 1 to revision $n - 1$ as training data and revision $n$ as test data. Treatment 100 is the resubstitution treatment where by all alerts are used to train the model and the model is tested via 10, 10-fold cross validations. Table 6.1 summarizes the number of alerts in each training and test set for the subject programs.
Table 6.1. **Train and test alert counts.** The number of alerts in the training set and test set for the three subject programs in FAULTBENCH v0.3. The test set contains alerts that were open at the last revision of the training set of alerts.

<table>
<thead>
<tr>
<th>Percent Revisions</th>
<th>jdom Train</th>
<th>jdom Test</th>
<th>runtime Train</th>
<th>runtime Test</th>
<th>logging Train</th>
<th>logging Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>339</td>
<td>318</td>
<td>476</td>
<td>355</td>
<td>65</td>
<td>39</td>
</tr>
<tr>
<td>80</td>
<td>405</td>
<td>302</td>
<td>554</td>
<td>290</td>
<td>66</td>
<td>37</td>
</tr>
<tr>
<td>90</td>
<td>436</td>
<td>284</td>
<td>596</td>
<td>76</td>
<td>66</td>
<td>36</td>
</tr>
<tr>
<td>99</td>
<td>452</td>
<td>267</td>
<td>585</td>
<td>41</td>
<td>66</td>
<td>36</td>
</tr>
<tr>
<td>100</td>
<td>454</td>
<td>454</td>
<td>590</td>
<td>590</td>
<td>67</td>
<td>67</td>
</tr>
</tbody>
</table>

6.1 **Hypothesis 1**

Hypothesis 1 theorizes that artifact characteristics of an alert and the surrounding source code are predictive of the actionability of an alert. Attribute selection is a process whereby the actionability of artifact characteristics is ascertained for use in machine learning models. Sixteen different attribute selection processes were run on the artifact characteristics for the three subject programs. The number of times an artifact characteristic was selected by an attribute selection process for treatment 100 of the subject programs is presented in Table 6.2 for each subject program.
Table 6.2. Artifact characteristic selection counts for FAULTBENCH subject programs. Sixteen artifact characteristics subsets were generated for each subject program.

<table>
<thead>
<tr>
<th>Artifact Characteristic</th>
<th>jdom</th>
<th>runtime</th>
<th>logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>alert category</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>alert lifetime</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>alerts in file</td>
<td>5</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>alerts in method</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>alerts in package</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>alerts in project</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>alert type</td>
<td>6</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>cyclomatic complexity</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>developers</td>
<td>0</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>file added lines</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>file age</td>
<td>1</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>file creation revision</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>file deleted lines</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>file deletion revision</td>
<td>7</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>file extension</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>file growth lines</td>
<td>7</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>file modified lines</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>file name</td>
<td>6</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>file percent modified lines</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>file size</td>
<td>7</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>file staleness</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>file total modified lines</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>latest file modification</td>
<td>3</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>latest package modification</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>latest project modification</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>line number</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>marker type</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>method name</td>
<td>7</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>method size</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>num alert modifications</td>
<td>11</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>number of classes in file</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>number of classes in package</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>number of functions in file</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>number of functions in package</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>open revision</td>
<td>3</td>
<td>11</td>
<td>0</td>
</tr>
<tr>
<td>open revision type</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Table 6.2 Continued

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>package added lines</td>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>package deleted lines</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>package modified lines</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>package name</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>package percent modified lines</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>package size</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>package total modified lines</td>
<td>2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>package staleness</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>priority</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>project added lines</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>project deleted lines</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>project growth lines</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>project modified lines</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>project name</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>project percent modified lines</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>project staleness</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>project total modified lines</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>test file</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>total alerts for revision</td>
<td>3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>total open alerts for revision</td>
<td>7</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

Out of the 57 artifact characteristics considered in SAAI, four (e.g. cyclomatic complexity, file extension, file modified lines, and open revision type) were not selected in any artifact characteristic subset. The artifact characteristic selection counts are displayed in Figure 6.1, summed across all subject programs. The most commonly selected artifact characteristic is the alert lifetime, which is in every artifact characteristic subset. Most of the churn artifact characteristics were chosen for an artifact characteristic subset in the jdom project, which may imply that the source code churn is more predictive in jdom than the other subject programs. The number of alert modifications was selected 26 times over all three subject
programs followed by method name, bug type, total open alerts for revision, and open revision. On average, 11 artifact characteristics were selected per subset for jdom, 10 artifact characteristics were selected per subset for runtime, and 4 artifact characteristics were selected per subset for logging. The selection of all but four of the proposed artifact characteristics demonstrate these characteristics are predictive of actionable and unactionable alerts supporting Hypothesis 1. However, there is a core set of artifact characteristics that are selected for each of the subject programs, with one or two supporting artifact characteristics. This core set of artifact characteristics, as evidenced by selections in many artifact characteristic subsets, may narrow the search space allowing for a potentially more efficient SAAI process.
Figure 6.1. Histogram of artifact characteristic selection counts across subjects. The artifact characteristics are ordered by the selection count.
6.2 Hypothesis 2

Hypothesis 2 theorizes that a systematic actionable alert identification technique using machine learning can accurately identify actionable alerts. Applying SAAI to the FAULTBENCH subject programs and using the evaluation metrics to select the best model will provide a measure of SAAI accuracy.

The SAAI process generates 256 possible models for each artifact characteristic subset and machine learning algorithm. The 256 models are created by 16 artifact characteristic subsets applied to 16 machine learners. The best model is selected by considering the accuracy followed by the precision, recall, AUC, f-measure, and error rates. Additionally, the generated model is checked to ensure that one or more of the selected artifact characteristics is used in the model, implying that the model is not classifying all alerts as either actionable or unactionable. Finally, the model selected must have the best accuracy and have at least one true positive alert if there is at least one actionable alert in the test set, unless none of the generated models can identify a true positive alert. The best models for the five treatments of jdom, runtime, and logging are presented in Tables 6.3, 6.4, and 6.5, respectively. The artifact characteristic subset for the best SAAI is presented. Where more than one artifact characteristics subset was best and the characteristics were different, the common characteristics were reported since those are the ones likely used by the resulting machine learner.
Table 6.3. SAAI best models for jdom treatments. For each jdom treatment, the classifier, artifact characteristics, classification metrics, and AUC for the selected SAAI model are presented.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Treat.</th>
<th>Classifier</th>
<th>Chosen Artifact Characteristics</th>
<th>Accuracy [CI]</th>
<th>Precision</th>
<th>Recall</th>
<th>AUC [CI]</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdom</td>
<td>70</td>
<td>Naïve Bayes</td>
<td>alert lifetime, num alert mods., num. classes in file, open revision, pkg, total mod lines, total open for rev.</td>
<td>82.70% [0.78,0.86]</td>
<td>90.91%</td>
<td>15.63%</td>
<td>88.09% [0.85,1.0]</td>
</tr>
<tr>
<td>jdom</td>
<td>80</td>
<td>IBk</td>
<td>alert lifetime num. alert mods., package size, total for rev. total open for rev.</td>
<td>86.09% [0.82,0.90]</td>
<td>100.00%</td>
<td>12.50%</td>
<td>68.80% [0.64,1.0]</td>
</tr>
<tr>
<td>jdom</td>
<td>90</td>
<td>PART</td>
<td>alert lifetime, alerts in pkg., alert type, file name, method name, pkg. name, pkg. size</td>
<td>89.79% [0.86,0.93]</td>
<td>66.67%</td>
<td>6.67%</td>
<td>54.48% [0.44,0.65] p-value = 0.2</td>
</tr>
<tr>
<td>jdom</td>
<td>99</td>
<td>J4.8</td>
<td>alert lifetime, alerts in file, file del. rev., file growth lines, file size, num. alert mods., num. classes in pkg.</td>
<td>94.01% [0.90,0.96]</td>
<td>0.00%</td>
<td>0.00%</td>
<td>51.10% [0.32,0.70] p-value = 0.46</td>
</tr>
<tr>
<td>jdom</td>
<td>100</td>
<td>J4.8</td>
<td>alert lifetime, num alert mods., num. classes in file, open revision, pkg, total mod lines, total open for rev.</td>
<td>97.80% [0.96,0.99]</td>
<td>98.97%</td>
<td>96.00%</td>
<td>98.69% [0.97,1.0]</td>
</tr>
</tbody>
</table>
Table 6.4. **SAAI best models for runtime treatments.** For each runtime treatment, the classifier, artifact characteristics, classification metrics, and AUC for the selected SAAI model are presented.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Treat.</th>
<th>Classifier</th>
<th>Chosen Artifact Characteristics</th>
<th>Accuracy [CI]</th>
<th>Precision</th>
<th>Recall</th>
<th>AUC [CI]</th>
</tr>
</thead>
<tbody>
<tr>
<td>runtime 70</td>
<td>LWL</td>
<td>alert lifetime, alert type, developer, method name</td>
<td>49.13% [0.44,0.54]</td>
<td>90.00%</td>
<td>47.52%</td>
<td>56.83% [0.49,0.64] p-value = 0.038</td>
<td></td>
</tr>
<tr>
<td>runtime 80</td>
<td>Conjunctive Rule</td>
<td>alert lifetime</td>
<td>65.17% [0.60,0.70]</td>
<td>91.11%</td>
<td>65.86%</td>
<td>63.41% [0.54,0.72] p-value = 0.0024</td>
<td></td>
</tr>
<tr>
<td>runtime 90</td>
<td>Simple Logistic</td>
<td>alert lifetime, latest file mod., open revision</td>
<td>82.89% [0.73,0.90]</td>
<td>100.00%</td>
<td>62.86%</td>
<td>100.00% [0.96,1.0]</td>
<td></td>
</tr>
<tr>
<td>runtime 99</td>
<td>IB1</td>
<td>alert lifetime, num. alert mods., open revision, pkg. added lines, pkg. deleted lines, pkg. total mod lines, total for rev., total open for rev.</td>
<td>100.00% [0.91,1.0]</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00% [0.1,0]</td>
<td></td>
</tr>
<tr>
<td>runtime 100</td>
<td>Simple Logistic, LMT</td>
<td>alert lifetime, num. alert mods., open revision, pkg. growth lines</td>
<td>100.00% [0.99,1.0]</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00% [0.99,1.0]</td>
<td></td>
</tr>
</tbody>
</table>

Table 6.5. **SAAI best models for logging treatments.** For each logging treatment, the classifier, artifact characteristics, classification metrics, and AUC for the selected SAAI model are presented.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Treat.</th>
<th>Classifier</th>
<th>Chosen Artifact Characteristics</th>
<th>Accuracy [CI]</th>
<th>Precision</th>
<th>Recall</th>
<th>AUC [CI]</th>
</tr>
</thead>
<tbody>
<tr>
<td>logging 70</td>
<td>Simple Logistic, LMT, Naïve Bayes</td>
<td>alert lifetime</td>
<td>89.74% [0.76,0.96]</td>
<td>0.00%</td>
<td>0.00%</td>
<td>88.88% [0.78,1.0]</td>
<td></td>
</tr>
<tr>
<td>logging 80</td>
<td>Naïve Bayes</td>
<td>alert lifetime, method name, total for rev.</td>
<td>97.30% [0.86,1.0]</td>
<td>0.00%</td>
<td>0.00%</td>
<td>97.22% [0.88,1.0]</td>
<td></td>
</tr>
<tr>
<td>logging 90</td>
<td>ADTree</td>
<td>alert lifetime, method name, total for rev.</td>
<td>100.00% [0.90,1.0]</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00% [0.1,0]</td>
<td></td>
</tr>
<tr>
<td>logging 99</td>
<td>ADTree</td>
<td>alert lifetime, method name, total for rev.</td>
<td>100.00% [0.90,1.0]</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00% [0.1,0]</td>
<td></td>
</tr>
<tr>
<td>logging 100</td>
<td>Simple Logistic, LMT</td>
<td>alert lifetime, method name, total open for rev.</td>
<td>98.51% [0.92,1.0]</td>
<td>96.88%</td>
<td>100.00%</td>
<td>99.46% [0.95,1.0]</td>
<td></td>
</tr>
</tbody>
</table>
All of the models, except treatment 70 on runtime have an accuracy greater than 65%. Eight of the 15 treatments have models with accuracy greater than 90% demonstrating that SAAI can accurately identify actionable and unactionable alerts, supporting Hypothesis 2. The confidence level for the confidence intervals calculated for accuracy and AUC are 95%. The p-value is provided for the AUC if the confidence interval was calculated using the method provided by Hanley and McNeil [15].

The subject program runtime is an interesting case due to the large differences between accuracy in the five treatments. Runtime had a total of 590 alerts, of which 41 were unactionable. At treatment 70, there were the 41 unactionable alerts and 435 actionable alerts training the model. The test set consisted of 344 alerts, 41 of which were unactionable. The LWL model generated for that treatment identified almost half of the actionable alerts, but the remaining were false negatives. Additionally, the LWL model identified most of the unactionable alerts, but there were still 16 false positives. In contrast, treatment 100 had a model with perfect accuracy that correctly identified all the actionable and unactionable alerts. The approximately 100 additional alerts in the training set were able to contribute to SAAI such that the best model had 100% accuracy.

The treatments with 0% precision, recall, and AUC are treatments where no actionable alerts were in the test sets. For these treatments, the accuracy at correctly identifying unactionable alerts and reducing the false positives was important. One of the key goals with supplementing ASA with AAIT is to find the actionable alerts out of the generated alerts, and false positives provide developers with additional alerts to inspect and discard rather than
focusing on the important alerts. However, in a security domain, the interest is in reducing
the number of false negatives, and the recall would be a more important measure than
precision [10].

There were many candidate models that could have been selected as the best model,
especially since many of the models reported accuracy within the confidence interval of the
best model. Additionally, the confidence interval for the AUC ROC curves is sufficiently
large to allow for many models to fall within the bounds of the confidence interval. An
implication of this result is that there may be many best models. Therefore, models that are
less costly to build and use could be considered based on the development environment
where the model would be deployed. Additionally, new data generated as ASA is used by a
development team may not require a change of the model. The model may only require
updating when the accuracy of the prediction falls outside the confidence interval of the
original test set. This is demonstrated in the transition from treatment 70 to 80 for jdom,
where the accuracy for treatment 80 of 86.09% falls just slightly outside the confidence
interval for treatment 70 (e.g. [0.78,0.86]). When comparing treatments 80 and 90 for jdom,
the accuracy for treatment 90 of 89.79% falls within the confidence interval of [0.82,0.90].

6.3 Hypothesis 3

Hypothesis 3 theorizes that a systematic AAIT using machine learning is project specific.
Previous work has applied AAIT across projects, to individual projects, and to specific ASA.
Results by Kim and Ernst [26] in addition to the inclusion of artifact characteristics that are
project specific (e.g. the alert’s location) lead to the creation of Hypothesis 3. The artifact
characteristics and selected model vary by subject program; therefore, the SAAI generated for one project may not be applicable to another project. When a specific SAAI model contains project specific artifact characteristics like project name, package name, file name, method name, and developers the specific model for a subject will not transfer to another subject. However, the artifact characteristics and machine learning algorithm selected via the SAAI process may work on another subject program. Table 6.6 presents the accuracy when applying the artifact characteristics and machine learning algorithm from the subject in the first column to another subject by column on treatment 100.

**Table 6.6. Accuracy when switching models for treatment 100 SAAI.** The artifact characteristics and machine learning classifier that make up an SAAI model from a project in a column are applied to a project in a row. The accuracy of applying the SAAI model for treatment 100 of jdom on the runtime project is 98.6%, which falls outside of the confidence interval of [0.99,1.0] for the best selected runtime model.

<table>
<thead>
<tr>
<th></th>
<th>jdom</th>
<th>runtime</th>
<th>logging</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdom</td>
<td>n/a</td>
<td>98.6%</td>
<td>95.5%</td>
</tr>
<tr>
<td>runtime</td>
<td>95.6%</td>
<td>n/a</td>
<td>98.5%</td>
</tr>
<tr>
<td>logging</td>
<td>90.1%</td>
<td>98.8%</td>
<td>n/a</td>
</tr>
<tr>
<td>Treatment 100</td>
<td>97.8%</td>
<td>100.00%</td>
<td>98.5%</td>
</tr>
<tr>
<td></td>
<td>[0.96,0.99]</td>
<td>[0.99,1.0]</td>
<td>[0.92,1.0]</td>
</tr>
</tbody>
</table>

The accuracy of the jdom SAAI for treatment 100 on runtime and logging is high, but not as high as the best models for runtime and logging by 1.4%-3.0%. The accuracy of the jdom SAAI for treatment 100 on runtime falls below the confidence interval for accuracy, but is within the confidence interval for logging. The same relationship holds for the other SAAI, except for the SAAI generated for logging using
runtime's artifact characteristics and machine learning algorithm, where the accuracy is the same as logging's selected model. Again, the accuracy generated when applying another subject model to jdom and runtime falls outside the confidence interval for the best jdom and runtime models, but is within the confidence interval for logging. Because the accuracy of the SAAIs were over the 90% high accuracy mark and for logging the accuracy falls within the confidence interval for the best candidate model, hypothesis 3 is not supported for the three FAULTBENCH v0.3 subject program, but cannot be rejected outright. There may be a core set of artifact characteristics and machine learning algorithms that work well on most subject programs. Finding the core set of artifact characteristics and machine learning algorithms will increase the efficiency of creating SAAI candidate models.

6.4 Limitations

The limitations to this work come from the FAULTBENCH threats to validity found in Section 3.5. In summary, there could be a threat to construct validity through our measurement of and calculations of the artifact characteristics. The threat to internal validity is from the tools written to automate FAULTBENCH v0.3 and the SAAI process. The threat to external validity is the generalizability of the results, which is partially mitigated through use of FAULTBENCH v0.3.

6.5 Summary

SAAI was run on the three subject programs from FAULTBENCH and the most accurate artifact characteristics and machine learning algorithms were selected for each treatment. Three hypotheses were evaluated. Hypothesis 1 is supported by showing that all but one of the
artifact characteristics was selected as part of an artifact characteristic subset. Hypothesis 2 is supported by showing that eight of 15 treatments have a reported accuracy greater than 90% for the generated SAAI models. Hypothesis 3 investigates the applicability of the artifact characteristics and machine learning algorithms generated for one subject program on another subject program. The reported accuracy of these models when applied to other subject programs is above the 90% accuracy cutoff and falls within the confidence intervals for logging’s best model, which demonstrates that hypothesis 3 is not supported for theFAULTBENCH V0.3 subject programs. Refinement of hypothesis 3 through investigation of additional subject programs is required.
7 Comparative Evaluation

Many AAIT have been proposed in literature, but no comprehensive comparative evaluation of AAIT exists. One of the reasons for FAULTBENCH is to compare AAIT, and FAULTBENCH is used for further evaluation of hypothesis 2 by comparing SAAI to four AAITs available in the literature and the APM model from my feasibility study.

- Hypothesis 2: A systematic actionable alert identification technique using machine learning can accurately identify actionable alerts.

The focus of the comparative evaluation is on AAITs that classify or prioritize alerts generated by ASAs for the Java programming language. Table 2.1 shows which AAIT from literature are applicable for the domain of the comparative study. There are six AAIT that were created for the Java programming language. The AAIT by Ayewah, et al. [3] does not provide a specific classification model, but is based on prioritizing alert types by actionability, and was not considered. The paper containing the explanation of the AAIT by Meng, et al. [33] did not provide enough detail for recreation, and an email to the authors was unanswered. The AAIT selected for the comparative evaluation are listed below:

- APM: Adaptive Prioritization Model [18] – Chapter 4
- ATL: Alert Type Lifetime developed by Kim and Ernst [26] and discussed in Section 2.3.4. Two variations of the model are considered. Kim and Ernst measure the lifetime in days (ATL-D). An additional model that measures the lifetime in revisions is also considered (ATL-R).
• CNC: Check ‘n’ Crash developed by Csallner, et al. [12] and discussed in Section 2.2.6. The CNC model could only be evaluated on alerts generated by ESC/JAVA [14].

• LRM: Logistic Regression Model developed by Ruthruff, et al. [41] and discussed in Section 2.3.10. Not all of the artifact characteristics used by Ruthruff, et al. could be reproduced, including the Google specific artifact characteristics and the indentation measure.

• HWP: History-based Warning Prioritization developed by Kim and Ernst [27] and discussed in Section 2.3.5.

For this comparative evaluation, the FAULTBENCH threats to validity apply. Additionally, the first author automated all but one (i.e. LRM) of the AAITs from the associated source papers and the programs to evaluate the finding of the AAITs. AAITs were manually tested. Additional threats to validity come from the AAITs themselves, and the limitations to each of the AAITs are discussed in Chapter 2.

7.1 jdom

The evaluation metrics for jdom are summarized in Table 7.1. The lines highlighted in light grey are the AAIT with the highest accuracy for each treatment when using both static analysis tools, and are the best models. Two of the treatments also have CNC as the best model; however, CNC only considers alerts generated by CHECK ‘N’ CRASH because there were no generated test cases for FINDBUGS. The treatments where CNC was the best model are highlighted in dark grey. The AAIT are ordered by accuracy within each treatment. The metric UA50 is the number of unactionable alerts identified before 50% of actionable alerts
and IFAA is the number of inspections before the first actionable alert and IAAA is the number of alerts inspected before all actionable alerts are identified. The UF50, IFAA, and IAAA metrics are not reported for any of the SAAI models because the models were treated strictly as classification models. The calculations for these metrics are in Section 3.3.3.
Table 7.1. **jdom summary data.** The evaluation metrics (Section 3.3.3) are presented for each AAIT and treatment for **jdom.** The lines highlighted in light grey are the non-CNC AAIT with the highest accuracy for a treatment. If CNC has the highest accuracy for a treatment, the line is highlighted in dark grey.

<table>
<thead>
<tr>
<th>Treatment</th>
<th>AAIT</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
<th>Accuracy (%)</th>
<th>% AUC Max Area</th>
<th>UF50</th>
<th>IFAA</th>
<th>IAAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>SAAI - NAÏVE BAYES</td>
<td>90.91%</td>
<td>15.63%</td>
<td>82.70%</td>
<td>88.09%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>70</td>
<td>APM</td>
<td>46.15%</td>
<td>9.38%</td>
<td>79.56%</td>
<td>59.96%</td>
<td>136</td>
<td>0</td>
<td>317</td>
</tr>
<tr>
<td>70</td>
<td>ATL-R</td>
<td>31.82%</td>
<td>10.94%</td>
<td>77.36%</td>
<td>56.91%</td>
<td>121</td>
<td>10</td>
<td>316</td>
</tr>
<tr>
<td>70</td>
<td>CNC</td>
<td>100.00%</td>
<td>6.25%</td>
<td>72.73%</td>
<td>56.25%</td>
<td>25</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>70</td>
<td>LRM</td>
<td>36.78%</td>
<td>64.00%</td>
<td>72.35%</td>
<td>82.79%</td>
<td>17</td>
<td>0</td>
<td>210</td>
</tr>
<tr>
<td>70</td>
<td>ATL-D</td>
<td>25.93%</td>
<td>21.88%</td>
<td>71.70%</td>
<td>61.89%</td>
<td>105</td>
<td>3</td>
<td>306</td>
</tr>
<tr>
<td>70</td>
<td>HWP</td>
<td>18.95%</td>
<td>73.44%</td>
<td>31.45%</td>
<td>60.75%</td>
<td>80</td>
<td>0</td>
<td>316</td>
</tr>
<tr>
<td>80</td>
<td>SAAI - IBK</td>
<td>100.00%</td>
<td>12.50%</td>
<td>86.09%</td>
<td>68.80%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>80</td>
<td>APM</td>
<td>41.67%</td>
<td>10.42%</td>
<td>83.44%</td>
<td>59.03%</td>
<td>100</td>
<td>0</td>
<td>276</td>
</tr>
<tr>
<td>80</td>
<td>ATL-D</td>
<td>20.00%</td>
<td>2.08%</td>
<td>83.11%</td>
<td>60.74%</td>
<td>108</td>
<td>2</td>
<td>290</td>
</tr>
<tr>
<td>80</td>
<td>ATL-R</td>
<td>23.53%</td>
<td>8.33%</td>
<td>81.13%</td>
<td>60.78%</td>
<td>123</td>
<td>0</td>
<td>300</td>
</tr>
<tr>
<td>80</td>
<td>CNC</td>
<td>100.00%</td>
<td>9.09%</td>
<td>80.00%</td>
<td>52.56%</td>
<td>25</td>
<td>0</td>
<td>46</td>
</tr>
<tr>
<td>80</td>
<td>LRM</td>
<td>34.67%</td>
<td>55.32%</td>
<td>76.43%</td>
<td>79.49%</td>
<td>22</td>
<td>0</td>
<td>238</td>
</tr>
<tr>
<td>80</td>
<td>HWP</td>
<td>15.17%</td>
<td>66.67%</td>
<td>35.43%</td>
<td>59.76%</td>
<td>102</td>
<td>0</td>
<td>300</td>
</tr>
<tr>
<td>90</td>
<td>CNC</td>
<td>0.00%</td>
<td>0.00%</td>
<td>95.12%</td>
<td>37.50%</td>
<td>25</td>
<td>25</td>
<td>27</td>
</tr>
<tr>
<td>90</td>
<td>SAAI - PART</td>
<td>66.67%</td>
<td>6.67%</td>
<td>89.79%</td>
<td>54.48%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>90</td>
<td>ATL-D</td>
<td>20.00%</td>
<td>3.33%</td>
<td>88.38%</td>
<td>68.02%</td>
<td>43</td>
<td>2</td>
<td>259</td>
</tr>
<tr>
<td>90</td>
<td>APM</td>
<td>0.00%</td>
<td>0.00%</td>
<td>86.97%</td>
<td>46.64%</td>
<td>170</td>
<td>12</td>
<td>260</td>
</tr>
<tr>
<td>90</td>
<td>ATL-R</td>
<td>23.53%</td>
<td>13.33%</td>
<td>86.27%</td>
<td>64.66%</td>
<td>49</td>
<td>0</td>
<td>282</td>
</tr>
<tr>
<td>90</td>
<td>LRM</td>
<td>32.08%</td>
<td>58.62%</td>
<td>82.86%</td>
<td>79.60%</td>
<td>26</td>
<td>0</td>
<td>217</td>
</tr>
<tr>
<td>90</td>
<td>HWP</td>
<td>8.67%</td>
<td>56.67%</td>
<td>32.39%</td>
<td>56.33%</td>
<td>76</td>
<td>33</td>
<td>282</td>
</tr>
<tr>
<td>99</td>
<td>CNC</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>39</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>99</td>
<td>ATL-D</td>
<td>0.00%</td>
<td>0.00%</td>
<td>95.13%</td>
<td>54.73%</td>
<td>120</td>
<td>25</td>
<td>199</td>
</tr>
<tr>
<td>99</td>
<td>SAAI - J48</td>
<td>0.00%</td>
<td>0.00%</td>
<td>94.36%</td>
<td>51.10%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>99</td>
<td>APM</td>
<td>0.00%</td>
<td>0.00%</td>
<td>93.63%</td>
<td>27.80%</td>
<td>179</td>
<td>147</td>
<td>243</td>
</tr>
<tr>
<td>99</td>
<td>ATL-R</td>
<td>0.00%</td>
<td>0.00%</td>
<td>89.51%</td>
<td>77.94%</td>
<td>54</td>
<td>17</td>
<td>205</td>
</tr>
<tr>
<td>99</td>
<td>LRM</td>
<td>0.00%</td>
<td>0.00%</td>
<td>84.64%</td>
<td>68.14%</td>
<td>101</td>
<td>34</td>
<td>207</td>
</tr>
<tr>
<td>99</td>
<td>HWP</td>
<td>2.22%</td>
<td>30.77%</td>
<td>30.71%</td>
<td>74.43%</td>
<td>52</td>
<td>51</td>
<td>123</td>
</tr>
<tr>
<td>100</td>
<td>SAAI - J48</td>
<td>98.97%</td>
<td>96.00%</td>
<td>97.80%</td>
<td>98.69%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>100</td>
<td>APM</td>
<td>97.96%</td>
<td>96.00%</td>
<td>97.36%</td>
<td>99.68%</td>
<td>0</td>
<td>0</td>
<td>291</td>
</tr>
<tr>
<td>100</td>
<td>LRM</td>
<td>82.49%</td>
<td>73.00%</td>
<td>81.28%</td>
<td>93.65%</td>
<td>0</td>
<td>0</td>
<td>369</td>
</tr>
<tr>
<td>100</td>
<td>ATL-R</td>
<td>70.42%</td>
<td>25.00%</td>
<td>62.33%</td>
<td>73.34%</td>
<td>116</td>
<td>0</td>
<td>442</td>
</tr>
<tr>
<td>100</td>
<td>HWP</td>
<td>52.87%</td>
<td>92.00%</td>
<td>60.35%</td>
<td>48.15%</td>
<td>207</td>
<td>60</td>
<td>426</td>
</tr>
<tr>
<td>100</td>
<td>ATL-D</td>
<td>100.00%</td>
<td>3.00%</td>
<td>57.27%</td>
<td>80.07%</td>
<td>58</td>
<td>0</td>
<td>442</td>
</tr>
<tr>
<td>100</td>
<td>CNC</td>
<td>100.00%</td>
<td>2.63%</td>
<td>51.95%</td>
<td>75.36%</td>
<td>15</td>
<td>0</td>
<td>73</td>
</tr>
</tbody>
</table>
The accuracy for each AAIT and treatment are presented in Figure 7.1. The accuracy grew over the treatments for APM and SAAI. ATL-D, ATL-R, CNC, and LRM all declined at treatment 100 because they misclassified many actionable alerts as unactionable. At treatment 100 for jdom, SAAI and APM both identified 192 of the 200 actionable alerts. The SAAI model had two fewer false positives than the APM model. The other models had more FPs and FNs, demonstrating that the models are less predictive of actionable and unactionable alerts. The HWP model at treatment 100 identified 184 of the 200 actionable alerts, but also reported 164 FPs; almost one FP for every TP. At treatment 99, all of the models except HWP reported no TPs, but had high accuracy due to identifying the unactionable alerts correctly. HWP at treatment 99 identified four of the 13 possible actionable alerts at the cost of 176 FPs. CNC shows high accuracy at treatments 90 and 99 due to testing fewer alerts, most of which were unactionable.
Figure 7.1. Accuracy across treatments for each AAIT on jdom. The SAAI model has accuracy close to the APM and ATL-D models. CNC had the highest accuracy at treatments 90 and 99 because the number of alerts that CNC could predict on in the test set were fewer (41 alerts at revision 90 compared to 284 alerts for the other models and 39 alerts at revision 99 compared to 267 alerts for the other models) and the accuracy was better within the smaller set of alerts.

After evaluating the accuracy, we next evaluated the precision. The precision measures how many actionable alerts were correctly predicted out of all the alerts predicted as actionable. The precision is an important measure because ASA becomes less useful when the developer must inspect many unactionable alerts to find the actionable alerts; therefore,
we are interested in a higher precision over a higher recall. The precision for each AAIT and treatment are presented in Figure 7.2. CNC reports 100% precisions at treatments 70, 80, and 100 by identifying the one actionable alert associated with a failing test case. SAAI reports the highest precision for treatments 70, 80, and 90 when alerts from both ASA are considered. ATL-D reports 100% precision at treatment 100. The precision of the AAITs decline as fewer revisions are considered for the test set, which may imply that using most of the project history to predict the latest revisions may not be useful or that the predictor of interest for the model may work best when considering only recent history. All AAIT’s except HWP and SAAI failed to find any actionable alerts at treatment 99. HWP found four of the 13 actionable alerts; however, HWP reported 176 FP classifications for a precision of 2%. SAAI found one actionable alert at the cost of three FP classifications resulting in a precision of 25%.
Figure 7.2. Precision across treatments for each AAIT on jdom. Precision measures how many of the actionable alert predictions were correct. A high precision means that there were few FP classifications. For treatments 90 there were 30 actionable alerts out of 284 alerts and at treatment 99 there were 13 actionable alerts out of 267 alerts. Most models found 17 or fewer actionable alerts at treatment 90 and four or fewer actionable alerts at treatment 99. At treatments 90 and 99, the models with the higher precision had fewer FP classifications. The best models at treatment 100 reduced the number of FP classifications. However, the number of correctly identified actionable alerts could be low. For example, ATL-D had a precision of 100% at treatment 100, find six actionable alerts with no FP classifications. However, ATL-D missed the other 194 actionable alerts.

Figure 7.3 presents the recall for each AAIT and treatment. The recall measures how well an AAIT finds actionable alerts. The recall for CNC is poor, which shows that CNC
works well at finding actionable alerts when associated test cases fail, but other alerts with passing test cases may also be actionable, and CNC is unable to find those actionable alerts. For treatments 70 through 99, there were at most 64 actionable alerts in the test set. The recall was low for APM, ATL-D, ATL-R, CNC, and SML for treatments 70 through 99; however, the number of FP classifications to TP classifications tended to be smaller. While HWP and LRM had high recall for treatments 70 through 99, the number of FP classifications to TP classifications ranged from 2:1 to 44:1, the latter of which may be too high for use. Like the precision for jdom, the recall declined as fewer revisions were considered for the test set. The recall for treatment 100 was greater than 90% for HWP, APM, and SAAI demonstrating that these models can identify all actionable alerts; however, APM and SAAI had a high precision with only four and two FP classifications, respectively. HWP had 164 FP classifications at treatment 100.
Figure 7.3. Recall across treatments for each AAIT on jdom. Recall measures how many actionable alerts were correctly predicted as actionable. For treatments 70, 80, 90, and 99, all the models, except HWP and LRM, reported low recall at the cost of many FP classifications.

The average UF50 ranged from 0 unactionable alerts identified before 50% of the actionable alerts were identified at treatment 100 for APM and LRM to 207 unactionable alerts identified before 50% of the actionable alerts at treatment 100 for HWP. At treatments 70, 80, 90, and 100, 16 of the AAITs reported an IFAA of 0, which means an actionable alert was reported first in the prioritization. The first actionable alert ranged from being reported first to 147th. The number of inspections required to find all actionable alerts ranged from 27 on
treatment 90 for CNC to 442 on treatment 100 for ATL-R. There were 200 actionable alerts in the jdom subject, but the number of actionable alerts varied by treatment. Therefore, the average IAAA value of 195 at treatment 99 (excluding CNC due to prioritizing only a subset of alerts) shows that 195 inspections were required on average to identify the 13 actionable alerts in the test set. However, SAAI did not report the UF50, IFAA, and IAAA metrics, so the SAAI model cannot be compared with the AAIT from literature.

7.2 runtime

The runtime evaluation metrics are summarized in Table 7.2. The lines highlighted in light grey have the highest accuracy for each treatment, and are the best models. The AAIT are ordered by accuracy within each treatment. The metric UA50 is the number of unactionable alerts identified before 50% of actionable alerts and IFAA is the number of inspections before the first actionable alert and IAAA is the number of alerts inspected before all actionable alerts are identified. The UF50, IFAA, and IAAA metrics are not reported for any of the SAAI models because the models were treated strictly as classification models. The calculations for these metrics are in Section 3.3.3.
Table 7.2. *runtime* summary data. The evaluation metrics (Section 3.3.3) are presented for each AAIT and treatment for *runtime*. The lines highlighted in light grey are the AAIT with the highest accuracy for a treatment.

<table>
<thead>
<tr>
<th>Treatment</th>
<th>AAIT</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
<th>Accuracy (%)</th>
<th>% AUC Max Area</th>
<th>UF50</th>
<th>IFAA</th>
<th>IAAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>LRM</td>
<td>88.08%</td>
<td>100.00%</td>
<td>88.08%</td>
<td>93.93%</td>
<td>2</td>
<td>0</td>
<td>344</td>
</tr>
<tr>
<td>70</td>
<td>HWP</td>
<td>87.89%</td>
<td>74.26%</td>
<td>68.31%</td>
<td>90.36%</td>
<td>25</td>
<td>0</td>
<td>343</td>
</tr>
<tr>
<td>70</td>
<td>SAAI - LWL</td>
<td>90.00%</td>
<td>47.52%</td>
<td>49.13%</td>
<td>56.83%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>70</td>
<td>APM</td>
<td>87.50%</td>
<td>32.34%</td>
<td>36.34%</td>
<td>88.96%</td>
<td>24</td>
<td>0</td>
<td>344</td>
</tr>
<tr>
<td>70</td>
<td>ATL-R</td>
<td>93.10%</td>
<td>26.73%</td>
<td>33.72%</td>
<td>92.72%</td>
<td>10</td>
<td>0</td>
<td>343</td>
</tr>
<tr>
<td>70</td>
<td>ATL-D</td>
<td>92.31%</td>
<td>7.92%</td>
<td>18.31%</td>
<td>92.68%</td>
<td>12</td>
<td>0</td>
<td>344</td>
</tr>
<tr>
<td>70</td>
<td>ATL-R</td>
<td>93.10%</td>
<td>26.73%</td>
<td>33.72%</td>
<td>92.72%</td>
<td>10</td>
<td>0</td>
<td>343</td>
</tr>
<tr>
<td>70</td>
<td>ATL-D</td>
<td>92.31%</td>
<td>7.92%</td>
<td>18.31%</td>
<td>92.68%</td>
<td>12</td>
<td>0</td>
<td>344</td>
</tr>
<tr>
<td>80</td>
<td>LRM</td>
<td>87.54%</td>
<td>99.60%</td>
<td>87.41%</td>
<td>87.23%</td>
<td>16</td>
<td>0</td>
<td>286</td>
</tr>
<tr>
<td>80</td>
<td>HWP</td>
<td>84.72%</td>
<td>73.49%</td>
<td>65.86%</td>
<td>89.48%</td>
<td>23</td>
<td>0</td>
<td>289</td>
</tr>
<tr>
<td>80</td>
<td>SAAI - CR</td>
<td>91.11%</td>
<td>65.86%</td>
<td>65.17%</td>
<td>63.41%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>80</td>
<td>ATL-R</td>
<td>93.68%</td>
<td>35.74%</td>
<td>42.76%</td>
<td>92.70%</td>
<td>10</td>
<td>0</td>
<td>289</td>
</tr>
<tr>
<td>80</td>
<td>APM</td>
<td>70.49%</td>
<td>17.27%</td>
<td>22.76%</td>
<td>84.54%</td>
<td>26</td>
<td>1</td>
<td>209</td>
</tr>
<tr>
<td>80</td>
<td>ATL-D</td>
<td>81.82%</td>
<td>3.61%</td>
<td>16.55%</td>
<td>89.94%</td>
<td>12</td>
<td>0</td>
<td>290</td>
</tr>
<tr>
<td>70</td>
<td>SAAI - LOG</td>
<td>100.00%</td>
<td>62.86%</td>
<td>82.89%</td>
<td>1.00%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>70</td>
<td>ATL-R</td>
<td>55.26%</td>
<td>60.00%</td>
<td>59.21%</td>
<td>71.53%</td>
<td>12</td>
<td>0</td>
<td>75</td>
</tr>
<tr>
<td>70</td>
<td>ATL-D</td>
<td>100.00%</td>
<td>2.86%</td>
<td>55.26%</td>
<td>69.91%</td>
<td>17</td>
<td>0</td>
<td>75</td>
</tr>
<tr>
<td>70</td>
<td>LRM</td>
<td>49.30%</td>
<td>100.00%</td>
<td>52.63%</td>
<td>73.33%</td>
<td>12</td>
<td>1</td>
<td>68</td>
</tr>
<tr>
<td>70</td>
<td>APM</td>
<td>46.51%</td>
<td>57.14%</td>
<td>50.00%</td>
<td>66.30%</td>
<td>20</td>
<td>1</td>
<td>77</td>
</tr>
<tr>
<td>90</td>
<td>LRM</td>
<td>45.31%</td>
<td>82.86%</td>
<td>46.05%</td>
<td>70.55%</td>
<td>20</td>
<td>0</td>
<td>71</td>
</tr>
<tr>
<td>90</td>
<td>APM</td>
<td>46.51%</td>
<td>57.14%</td>
<td>50.00%</td>
<td>66.30%</td>
<td>20</td>
<td>1</td>
<td>77</td>
</tr>
<tr>
<td>100</td>
<td>SAAI - LOG</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>1.00%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>100</td>
<td>ATL-D</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>100</td>
<td>ATL-R</td>
<td>0.00%</td>
<td>0.00%</td>
<td>97.56%</td>
<td>0.00%</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>100</td>
<td>APM</td>
<td>0.00%</td>
<td>0.00%</td>
<td>57.50%</td>
<td>0.00%</td>
<td>40</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>100</td>
<td>HWP</td>
<td>0.00%</td>
<td>0.00%</td>
<td>29.27%</td>
<td>0.00%</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>100</td>
<td>SAAI - IB1</td>
<td>0.00%</td>
<td>0.00%</td>
<td>12.20%</td>
<td>0.00%</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>100</td>
<td>ATL-D</td>
<td>0.00%</td>
<td>0.00%</td>
<td>12.20%</td>
<td>0.00%</td>
<td>41</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>100</td>
<td>SAAI - LOG, LMT</td>
<td>100.00%</td>
<td>100.00%</td>
<td>100.00%</td>
<td>1.00%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>100</td>
<td>APM</td>
<td>100.00%</td>
<td>100.00%</td>
<td>95.25%</td>
<td>99.85%</td>
<td>0</td>
<td>0</td>
<td>569</td>
</tr>
<tr>
<td>100</td>
<td>LRM</td>
<td>95.59%</td>
<td>98.72%</td>
<td>94.58%</td>
<td>99.63%</td>
<td>0</td>
<td>0</td>
<td>584</td>
</tr>
<tr>
<td>100</td>
<td>HWP</td>
<td>94.01%</td>
<td>100.00%</td>
<td>94.07%</td>
<td>94.12%</td>
<td>20</td>
<td>6</td>
<td>587</td>
</tr>
<tr>
<td>100</td>
<td>ATL-R</td>
<td>95.71%</td>
<td>93.44%</td>
<td>90.00%</td>
<td>97.42%</td>
<td>11</td>
<td>0</td>
<td>590</td>
</tr>
<tr>
<td>100</td>
<td>ATL-D</td>
<td>100.00%</td>
<td>14.94%</td>
<td>20.85%</td>
<td>97.27%</td>
<td>11</td>
<td>0</td>
<td>590</td>
</tr>
</tbody>
</table>

The accuracy of the AAITs on *runtime* varied greatly over the treatments, as seen in Figure 7.4. SAAI’s accuracy increased from treatment 70 to 99 and the selected SAAI reported
100% accuracy at treatments 99 and 100. At treatment 99, there were no actionable alerts in the test set, and SAAI correctly identified all the alerts as unactionable, which none of the other models did. At treatment 100, SAAI correctly split all of the alerts into actionable and unactionable groups. All of the AAIT, except ATL-D showed high accuracy at treatment 100, demonstrating that the AAIT performed well at identifying some of the unactionable alerts, even on a skewed dataset like runtime, where only 7% of the alerts were unactionable.

ATL-D prioritized alert types by the alert’s lifetime using days as a measure of time. The variation of the accuracy within the different treatments, especially compared to ATL-R at treatment 100, may suggest that using days to measure time is less stable than revisions. Part of the reason for the poor performance of ATL-D is that the last two analyzed runtime revisions were separated by eight and eighteen months from the prior revisions, which would significantly increase the alert lifetime of the alerts still open in these revisions. Using revisions in ATL-R, removed the effect of the long time between revisions, and produced a better model at treatment 100.
Figure 7.4. Accuracy across treatments for each AAIT on runtime. SAAI increases across treatments by correctly identifying the 7% of unactionable alerts, while the other AAITs vary by treatment.

The precision of AAITs on runtime decreased from treatment 70 to 99, except for SAAI and ATL-D, which increased at treatment 90, as shown in Figure 7.5. The precision for all of the AAITs increased to over 94% at treatment 100 showing that all the AAIT tended to correctly prediction actionable alerts. The precision is 0 at treatment 99 because no actionable alerts were in the test set. ATL-D and SAAI both reported perfect precision; however, SAAI identified all of the actionable alerts while ATL-D found only 82 of the 549
actionable alerts. The number of TPs and FPs were relatively close in treatments 70, 80, and 90 due to a large number of misclassifications for the APM, ATL-D, ATL-R, and SAAI models. The HWP and LRM models had higher precision at treatments 70 and 80 because they found most of the actionable alerts. At treatment 90, HWP and LRM had many FPs due to the overwhelming number of actionable alerts in the training data.

Figure 7.5. Precision across treatments for each AAIT on runtime. The precision is 0 for all AAITs at treatment 99 because no actionable alerts were in the test set.
The recall of AAITs on runtime varied greatly by AAIT, as shown in Figure 7.6. LRM had the highest recall for treatments 70, 80, and 90, at the cost of identifying most of the unactionable alerts as actionable. The recall for the other AAIT ranged from 2% to 83%. Most of the models had a high number of FNs, but fewer FPs than HWP and LRM. The recall of all of the models improved at treatment 100 to almost 100% except for ATL-D, which was 15%.

Figure 7.6. Recall across treatments for each AAIT on runtime. The recall at treatment 99 is 0 because there are no actionable alerts in the test set.
Since revision 99’s test set contained no actionable alerts, the UF50, IFAA, and IAAA were 41 (except for LRM, which only had 40 alerts with a ranking due to an alert being excluded because the bug type was not in the test set). For the remaining treatments, on average, 20 unactionable alerts required inspection before 50% of the actionable alerts were inspected. Nine unactionable alerts required inspection before the first actionable alert was found, on average. For treatment 100, 584 alerts, on average, required inspection before all actionable alerts were found. Out of the 590 alerts, six alerts did not require inspection, on average. However, SAAI did not report the UF50, IFAA, and IAAA metrics, so the SAAI model cannot be compared with the AAIT from literature.

### 7.3 logging

The logging evaluation metrics are summarized in Table 7.3. The lines highlighted in light grey are the AAIT with the highest accuracy for each treatment when using both static analysis tools, and are the best models. Three of the treatments also have CNC as the best model; however, CNC only considers alerts generated by CHECK ‘N’ CRASH because there were no generated test cases for FINDBUGS. The treatments where CNC was the best model are highlighted in dark grey. The AAIT are ordered by accuracy within each treatment. The metric UA50 is the number of unactionable alerts identified before 50% of actionable alerts and IFAA is the number of inspections before the first actionable alert and IAAA is the number of alerts inspected before all actionable alerts are identified. The UF50, IFAA, and
IAAA metrics are not reported for any of the SAAI models because the models were treated strictly as classification models. The calculations for these metrics are in Section 3.3.3.
Table 7.3. **logging** summary data. The evaluation metrics (Section 3.3.3) are presented for each AAIT and treatment for logging. The lines highlighted in light grey are the non-CNC AAIT with the highest accuracy for a treatment. If CNC has the highest accuracy for a treatment, the line is highlighted in dark grey.

<table>
<thead>
<tr>
<th>Treatment</th>
<th>AAIT</th>
<th>Precision (%)</th>
<th>Recall (%)</th>
<th>Accuracy (%)</th>
<th>% AUC Max Area</th>
<th>UF50</th>
<th>IFAA</th>
<th>IAAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>ATL-D</td>
<td>0.00%</td>
<td>0.00%</td>
<td>92.31%</td>
<td>29.78%</td>
<td>35</td>
<td>9</td>
<td>38</td>
</tr>
<tr>
<td>70</td>
<td>ATL-R</td>
<td>0.00%</td>
<td>0.00%</td>
<td>92.31%</td>
<td>33.33%</td>
<td>35</td>
<td>5</td>
<td>38</td>
</tr>
<tr>
<td>70</td>
<td>SAAI - Imt, log, Naïve Bayes</td>
<td>0.00%</td>
<td>0.00%</td>
<td>89.74%</td>
<td>88.88%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>70</td>
<td>LRM</td>
<td>25.00%</td>
<td>100.00%</td>
<td>76.92%</td>
<td>96.44%</td>
<td>2</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>70</td>
<td>CNC</td>
<td>0%</td>
<td>0%</td>
<td>66.67%</td>
<td>30.77%</td>
<td>4</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>70</td>
<td>HWP</td>
<td>8.11%</td>
<td>100.00%</td>
<td>32.00%</td>
<td>18.22%</td>
<td>36</td>
<td>20</td>
<td>39</td>
</tr>
<tr>
<td>80</td>
<td>CNC</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>80</td>
<td>ATL-D</td>
<td>0.00%</td>
<td>0.00%</td>
<td>97.30%</td>
<td>97.22%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>80</td>
<td>SAAI - Naïve Bayes</td>
<td>0.00%</td>
<td>0.00%</td>
<td>97.30%</td>
<td>91.78%</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>80</td>
<td>ATL-R</td>
<td>0.00%</td>
<td>0.00%</td>
<td>97.30%</td>
<td>91.78%</td>
<td>16</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>80</td>
<td>APM</td>
<td>14.29%</td>
<td>100.00%</td>
<td>83.78%</td>
<td>100.00%</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>80</td>
<td>LRM</td>
<td>4.00%</td>
<td>100.00%</td>
<td>35.14%</td>
<td>36.99%</td>
<td>23</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>90</td>
<td>SAAI - adTree</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>90</td>
<td>CNC</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>90</td>
<td>ATL-D</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>90</td>
<td>ATL-R</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>90</td>
<td>APM</td>
<td>0.00%</td>
<td>0.00%</td>
<td>91.67%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>90</td>
<td>LRM</td>
<td>0.00%</td>
<td>0.00%</td>
<td>83.33%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>90</td>
<td>HWP</td>
<td>0.00%</td>
<td>0.00%</td>
<td>33.33%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>99</td>
<td>SAAI - adTree</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>99</td>
<td>CNC</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>99</td>
<td>ATL-D</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>99</td>
<td>ATL-R</td>
<td>0.00%</td>
<td>0.00%</td>
<td>100.00%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>99</td>
<td>APM</td>
<td>0.00%</td>
<td>0.00%</td>
<td>91.67%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>99</td>
<td>LRM</td>
<td>0.00%</td>
<td>0.00%</td>
<td>83.33%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>99</td>
<td>HWP</td>
<td>0.00%</td>
<td>0.00%</td>
<td>33.33%</td>
<td>0.00%</td>
<td>36</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>100</td>
<td>SAAI - Imt, log</td>
<td>96.88%</td>
<td>100.00%</td>
<td>98.51%</td>
<td>99.46%</td>
<td>n/a</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>100</td>
<td>APM</td>
<td>100.00%</td>
<td>90.32%</td>
<td>91.04%</td>
<td>98.94%</td>
<td>0</td>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>100</td>
<td>LRM</td>
<td>83.33%</td>
<td>64.52%</td>
<td>77.61%</td>
<td>91.36%</td>
<td>1</td>
<td>0</td>
<td>55</td>
</tr>
<tr>
<td>100</td>
<td>ATL-R</td>
<td>100.00%</td>
<td>38.71%</td>
<td>71.64%</td>
<td>84.53%</td>
<td>4</td>
<td>0</td>
<td>61</td>
</tr>
<tr>
<td>100</td>
<td>ATL-D</td>
<td>100.00%</td>
<td>16.13%</td>
<td>61.19%</td>
<td>79.08%</td>
<td>8</td>
<td>0</td>
<td>63</td>
</tr>
<tr>
<td>100</td>
<td>CNC</td>
<td>100.00%</td>
<td>20.00%</td>
<td>55.56%</td>
<td>50.77%</td>
<td>4</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>100</td>
<td>HWP</td>
<td>46.67%</td>
<td>67.74%</td>
<td>49.25%</td>
<td>69.31%</td>
<td>15</td>
<td>8</td>
<td>64</td>
</tr>
</tbody>
</table>
The accuracy for each of the AAIT and treatments for logging, as shown in Figure 7.7, is very similar to the accuracy results for jdom. For logging, there were only three actionable alerts in treatment 70, one actionable alert in treatment 80, and no actionable alerts in treatments 90 and 99. Therefore, the high accuracy reported for ATL-D, ATL-R, and SAAI are from correctly identifying the unactionable alerts. There were only six alerts, two of which were actionable, generated by CNC at treatment 70 and four unactionable alerts generated at treatments 80, 90, and 99. CNC was unable to identify any of the actionable alerts, but correctly identified all of the unactionable alerts at treatments 80, 90, and 99. At treatment 100, SAAI had a single FP and no FNs while APM had three FPs and three FNs. The other AAIT had a large number of either FPs or FNs, leading to a low accuracy.
Figure 7.7. Accuracy across treatments for each AAIT on logging. The accuracy of most of the AAIT was high due to the correct classification of the mostly unactionable test sets for treatments 70, 80, 90, and 99. SAAI and APM had the most correct classifications at treatment 100. HWP had many FPs, resulting in the low accuracy.

LRM and HWP had the highest precision at treatments 70 and 80, by identifying all three of the actionable alerts for treatment 70 and the single actionable alert in treatment 80, as shown in Figure 7.8. The precision of all AAIT is zero at treatments 90 and 99 because there were no actionable alerts in the test sets. At treatment 100, ATL-D, ATL-R, and CNC all had
precisions of 100% by not identifying FPs, but these AAIT only found five, 12, and one of the actionable alerts, respectively.

![Logging Precision Diagram](image)

**Figure 7.8. Precision across treatments for each AAIT on logging.** The precision for treatments 90 and 99 is zero for all AAIT due to the lack of actionable alerts in the test sets. Only HWP and LRM identified all of the actionable alerts at treatments 70 and 80, but the number of FPs lead to lower precisions. ATL-D, ATL-R, and CNC had 100% precision at treatment 100 but only identified a few of the actionable alerts.

Both HWP and LRM had 100% recall at treatments 70 and 80, suggesting that these models are good at finding the few actionable alerts in the test set, as shown in Figure 7.9. However,
by finding all of the actionable alerts, the HWP and LRM models also predicted that many of the unactionable alerts were actionable. HWP identified many more FPs than LRM. SAAI had 100% recall at treatment 100. The other AAIT missed between three and 26 of the 31 actionable alerts at treatment 100. All AAIT had recall of zero at treatments 90 and 99 because there were no actionable alerts in the test sets.

![Diagram of Recall across treatments for each AAIT on logging](image)

**Figure 7.9. Recall across treatments for each AAIT on logging.** The precision for treatments 90 and 99 is zero for all AAIT due to the lack of actionable alerts in the test sets. HWP and LRM identified all of the actionable alerts at treatments 70 and 80. SAAI identified all of the actionable alerts at treatment 100.
For treatments 90 and 99, all the alerts in the test set were unactionable; therefore, the UF50, IFAA, and IAAA metrics are all the number of alerts in the test set. For treatment 100, on average six unactionable alerts were inspected before the first 50% of the actionable alerts were found. For CNC, none of the actionable alerts in the test sets had associated test failures except in treatment 100 due to using all alerts as the test set. For treatment 100 all of the AAITs except HWP (and SAAI due to being classification rather than prioritization) identified an actionable alert first in the list. The number of inspections required to find all actionable alerts ranged from 39 to 64 for treatment 100. SAAI did not report the UF50, IFAA, and IAAA metrics, so the SAAI model cannot be compared with the AAIT from literature using these metrics.

**7.4 Limitations**

The limitations to this work come from the FAULTBENCH threats to validity found in Section 3.5. In summary, there could be a threat to construct validity through our measurement of and calculations of the artifact characteristics. The threat to internal validity is from the tools written to automate FAULTBENCH v0.3 and the AAIT in the comparative study. One way the threat to internal validity was addressed was by using the same train and test set of alerts for each AAIT. The threat to external validity is the generalizability of the results which is partially mitigated through use of FAULTBENCH v0.3. Additional threats to validity come from the AAITs themselves, and the limitations to each of the AAITs are discussed in Chapter 2.
7.5 Summary

The goal of this comparative evaluation is to inform selection of AAITs to supplement ASA by comparatively evaluating six actionable alert identification techniques using FAULTBENCH. We compared six AAITs using three subjects from FAULTBENCH V0.3 using the accuracy, precision, recall, AUC, UA50, IFAA, and IAAA metrics. We considered five treatments for each subject where a percentage of revisions in the history of a subject were used to train an AAIT and the remaining revisions tested the generated model. SAAI was found to be the best overall model by having the highest accuracy or tying for the highest accuracy on 11 of the 15 treatments. ATL-D, ATL-R, and LRM followed by having the highest accuracy or tying for the highest accuracy on five, four, and two of the treatments, respectively. CNC also had the highest accuracy on five of the 15 treatments; however, CNC only considered alerts from one of the two ASA used in the study.

The results of the study suggest that models generated by SAAI perform better than other AAIT from literature. AAIT performs better than an AAIT created from insight into ASA. Additionally, the model underlying the SAAI AAIT changed across treatments, suggesting that periodically refreshing AAITs is important for maintaining a high level of accuracy in predicting actionable alerts. However, these results are not conclusive, suggesting that further comparisons are required on larger software projects, preferably on projects actively using ASA.
8 User Study

The comparative evaluation of the SAAI AAIT with other AAITs from literature is suggestive of the usefulness of AAITs in general and the SAAI AAIT specifically. AAITs supplement ASA, which may potentially lead to an increase of industry adoption. A user study, conducted on students in the senior level, project-based, capstone course, will evaluate how ASA and AAIT, specifically SAAI, are used in practice.

The goal of the user study is to determine if developers find ASA, particularly ASA supplemented with AAIT useful when developing software. The subjects of the study were solicited from NCSU’s senior design project course. Students are divided into teams of four to develop a project sponsored by an industry partner. Potential participants were restricted to teams using either Java or Python as their development language for the project. From these potential participants, individual students or teams were presented with the study overview. The study was approved by the IRB\textsuperscript{27}. The students who chose to participate signed a release form and were provided with instructions on how to install the appropriate tooling for the study. The remainder of this section discusses the study set up, the models generated for the subjects, and the results of the user study.

8.1 Study Setup

At the start of the study, subjects were required to install the Eclipse plug-in AWARE [20], first introduced in Chapter 4. AWARE is open-source under a Common Public License. AWARE gathers alerts generated by ASA and classifies or prioritizes the alerts using an AAIT.

\textsuperscript{27} IRB approval number 61-09-01.
Students developing their senior project using Java used the `FINDBUGS` v.1.3.7 ASA tool [21] as an Eclipse plug-in and `AWARE-Java` v1.8.0. Students developing their senior design project using Python used PyLint\(^{28}\) v0.16.0 ASA tool, the Pydev\(^{29}\) v1.4.2 Eclipse plug-in for Python development, and `AWARE-Python` v1.8.0.

The study consisted of two parts: 1) using ASA during development; and 2) using ASA supplemented by `SAAI` during development. The first part of the study used `AWARE` without `SAAI`. All ASA alerts had a value of zero. Data collected by `AWARE` over a four to six week period served as input to the `SAAI` process described in Section 5. An individual model was generated for each subject. The subjects updated `AWARE` to a version containing their personal model and continued using `AWARE` for another four to six weeks. The data generated by `AWARE` was again collected for analysis, and the students filled out a survey reflecting on their experiences using `AWARE` and the associated `SAAI` AAIT. The survey questions may be found in Appendix A.

Initially, 12 students signed up to participate in the user study; however, nine students did not collect enough data by the first collection date to continue in the study. One of the three remaining students belonged to a project team that decided to no longer use Python as their programming language, and did not continue with the study. The remaining two subjects finished the study. Both students belonged to the same team and were the primary developers for their team’s project. The project was implemented in Python.

\(^{28}\) [http://www.logilab.org/project/pylint](http://www.logilab.org/project/pylint)

8.2 Model Generation

AWARE records the current state of ASA alerts at every developer action and when ASA is run. Each of these alert state records are analogous to a revision from a software repository, except at a more granular level. The SAAI AAIT was generated by building the alert history from the alert state while AWARE was used.

During the SAAI model generation process, a subset of the potential artifact characteristics described in Section 5.1 was considered. AWARE only collected data about the ASA alerts, not the surrounding source code. The following 13 artifact characteristics were evaluated: project name, package name, file name, severity, bug type, alerts in method, alerts in file, alerts in package, alerts in project, open version (timestamp), total alerts for revision, total open alerts for revision, and alert lifetime.

For Subject 1, the model with the highest accuracy was KStar, a nearest neighbor model, with attributes bug type, alerts in method, and alert lifetime. The KStar model was chosen because the average error was less than the other models with similar accuracy, precision, and recall.

For Subject 2, the model with the highest accuracy was IBk, where k is equal to one, using the alerts in package, open version, and alert lifetime artifact characteristics. IBk is a nearest neighbors model, whereby an uninspected alert is given the classification as the closest neighbor to the alert [49]. Weka’s [49] library provided the model implementations within AWARE, which allowed for the models to be updated after each alert inspection.
8.3 Study Results

There are four research questions of interest for the user study:

1. Is there a difference in the average amount of time to close and suppress alerts before and after the SAAI AAIT was added to AWARE?

2. Where the alerts closed after the SAAI AAIT was added to AWARE predicted as actionable?

3. Were the generated SAAI AAIT user specific?

4. Did the subject perceive that ASA and ASA supplemented with AAIT as beneficial during development?

For research question 1, the time between an alert’s creation and the alert’s closure or suppression was recorded. We considered only distinct alerts recorded as closed or suppressed in AWARE’s database. The time difference was from the earliest recorded open time to the latest recorded closure or suppression time. If the last alert closure was followed by an alert reopening, then the alert was not considered in the analysis. The mean alert lifetime of alerts closed before the first data drop is compared with the mean alert lifetime of alerts closed after the first data drop. There may be some error in this calculation due to the time required to build a custom model between the first data drop and the deployment of the new model.

Subject 1 suppressed 16 alerts before the first data drop and closed one alert after the first data drop. A t-test cannot be conducted due to only one alert in the post data drop set. The
alert lifetime of the alert closed after the data drop is smaller than the mean alert lifetime before the first data drop.

Subject 2 closed 56 and suppressed 34 alerts before the first data drop and closed 15 alerts after the first data drop. A t-test of the means of the closed alerts before and after the data drop do not show a statistically significant different between the two means (t = 1.7647, degrees of freedom = 55, and p-value = 0.08316); however the mean lifetime of the closed alerts after the data drop is smaller than before the data drop.

For research question 2, we consider the alerts closed after the first data drop. The predicted actionability of an alert are compared to the developers actions when inspecting the alerts. The states of an alert before or after inspection are not considered in the analysis and do not affect the alerts chosen for analysis. Subject 1 closed two alerts, and both alerts were predicted to be unactionable, which are a false negatives. The two alert closures vary from the count of one closed alert discussed in evaluating research question 1 is because one of the closed alerts was later reopened. Subject 2’s SAAI model correctly predicted 16 of the closed alerts were actionable and incorrectly predicted than one alert was unactionable. The SAAI AAIT for subject 2 had a 94% accuracy compared to the 0% accuracy of subject 1. The results from subject 2 are encouraging that SAAI AAIT is predictive of actionable alerts in practice. The results of subject 1 are less encouraging, but due to the small number of alert closures, the results are not discouraging.

For research question 3, we compare the models and artifact characteristics built for each subject. Both subjects had alert lifetime as a common artifact characteristic, but the other
important artifact characteristics differed. The models generated by SAAI AAIT were different implementations of the nearest neighbor model. A third model was generated for a subject that later dropped out of the study. The artifact characteristics were alerts in method, alerts in file, and alert lifetime and the machine learner was Naïve Bayes.

Research question 4 was evaluated through the survey the subjects completed at the end of the study. The survey is presented in Appendix A. The survey consisted of 14 questions about the subject’s experience with using AWARE during development. The survey responses are presented in Table 8.1, below. Both subjects found that ASA was useful during development, particularly since they had never developed in Python before. Subject 2 felt that AWARE helped their development, while subject 1 felt AWARE had no effect on their development. However, there is no conclusive difference between using just ASA and ASA supplemented with SAAI AAIT.
Table 8.1. Subject survey responses to AWARE questionnaire. The subject responses to the survey questions, which are listed in Appendix A.

<table>
<thead>
<tr>
<th>Question</th>
<th>Subject 1</th>
<th>Subject 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I ran static analysis automatically during development.</td>
<td>I ran static analysis automatically during development.</td>
</tr>
<tr>
<td>2</td>
<td>Every once in a while</td>
<td>A couple of times per coding session</td>
</tr>
<tr>
<td>3</td>
<td>has no effect on</td>
<td>helps</td>
</tr>
<tr>
<td>4</td>
<td>A little, but the lag doesn’t affect/annoy me</td>
<td>Not that I’ve noticed</td>
</tr>
<tr>
<td>5</td>
<td>I might defer looking at it until a later time. Some of the problems were useful but some were just not important. Especially the warnings on where to declare a variable.</td>
<td>I might defer looking at it until a later time. Sometimes if I’m busy I will go back and check later</td>
</tr>
<tr>
<td>6</td>
<td>Usually accurate, but sometimes the errors are not actually errors</td>
<td>Usually accurate, but sometimes the errors are not actually errors</td>
</tr>
<tr>
<td>7</td>
<td>AWARE is doing ok</td>
<td>AWARE is doing ok</td>
</tr>
<tr>
<td>8</td>
<td>Occasionally</td>
<td>Occasionally</td>
</tr>
<tr>
<td>9</td>
<td>I generally look at the highest ranked alerts</td>
<td>I just go down the list</td>
</tr>
<tr>
<td>10</td>
<td>I don’t have enough information to compare the two</td>
<td>They’re about the same to me</td>
</tr>
<tr>
<td>11</td>
<td>It brought a ranking system of the code I was developing and that was pretty interesting.</td>
<td>It seems to help me adhere to best practices when coding</td>
</tr>
<tr>
<td>12</td>
<td>None</td>
<td>It complains a lot about import statements for no reason.</td>
</tr>
<tr>
<td>13</td>
<td>The alerts were semi-useful. I wasn’t familiar with python code organization when I started but AWARE helped me dive into it.</td>
<td>When there was an actual syntax error it did a great job. Sometimes it would say stuff, like import statements needed to be fixed, which really didn’t</td>
</tr>
<tr>
<td>14</td>
<td>None</td>
<td>None</td>
</tr>
</tbody>
</table>
8.4 Limitations

The limitations for the user study come from the AWARE tooling and the small sample size of the user study. AWARE records three types of information: 1) the alert state after each code change in comma separated value files; 2) the alert state in a database, which is used when Eclipse is first opened; and 3) a listing of alert events stored in the database. Alerts that share the same project name, package name, file name, bug type, and source hash or line number are considered the same alert. Alerts that change may only be tracked if either the source hash or line number changes. If both the source hash and line number change, then the alert is closed and a new alert is opened. An additional limitation in AWARE is that when a closed or suppressed alert is reopened or unsuppressed, the alert cannot be removed from the set of alerts used for training. The Weka [49] code was modified to allow for removal when using IBk, but the remaining machine learners were unable to be modified. Finally, a limitation to the study is the sample size of two. With a sample size of two, no conclusive conclusions may be made about the user study.
9 Contributions and Future Work

ASA is a viable technique for developers to identify potential anomalies in their source code during development. The cost of ASA is that many of the reported alerts require developer inspection and many may be unactionable due to either incorrect analysis or unimportant alerts. Supplementing ASA with an AAIT provides an additional level of feedback for guiding developers to actionable alerts.

Three hypotheses were investigated for this research:

- Hypothesis 1: The artifact characteristics of an alert and the surrounding source code are predictive of the actionability of an alert.
- Hypothesis 2: A systematic actionable alert identification technique using machine learning can accurately identify actionable alerts.
- Hypothesis 3: A systematic actionable alert identification technique using machine learning is project specific.

Of the 58 proposed artifact characteristics, all but four were selected as part of at least one artifact characteristic subset. The alert lifetime was chosen for every artifact characteristic subset, showing that the alert lifetime is potentially the most predictive artifact characteristics. However, the inclusion of other artifact characteristics in addition to alert lifetime when creating SAAI is likely important for increased model accuracy. SAAI typically performed better than ATL-D and ATL-R, which are prioritize alert types by the average lifetime. Artifact characteristics chosen for one or two artifact characteristic subsets may not be as predictive or even used in the model generated for those subsets. For every artifact
characteristic subset studied, at least one of the attribute selection techniques identified predictive artifact characteristics suggesting the acceptance of hypothesis 1. Further work is required to refine the list of 58 proposed artifact characteristics into a smaller set of the most predictive artifact characteristics.

The SAAI process generated for the three FAULTBENCH [18] v0.3 subject programs, had accuracy ranging from 49.13% to 100.00%. Eight of the 15 FAULTBENCH subject treatments reported accuracy greater than 90% when using a SAAI. Additionally, when comparing SAAI models with other actionable alert identification techniques from literature on FAULTBENCH v0.3 subject programs found that SAAI models had the highest accuracy for 11 of the 15 treatments. Both of the above results support hypothesis 2.

Hypothesis 3 is not supported on the FAULTBENCH v0.3 subject programs. When applying a model, in the form of selected artifact characteristics and machine learner, from one subject on another subject, the reported accuracy is above the 90% high accuracy cutoff. Additionally, the accuracies when applying jdom and runtime models to logging are in the confidence interval for treatment 100 of logging.

The contributions of this research are as follows:

• A systematic actionable alert identification model building process to predict actionable and unactionable automated static analysis alerts;

• A benchmark, FAULTBENCH, for evaluating and comparing actionable alert identification techniques; and
• A comparative evaluation of systematic actionable alert identification models with other actionable alert identification techniques from literature.

These contributions further the knowledge about supplementing ASA with AAIT. However, more research is required to provide additional support the reported results. Further comparative evaluations should consider all AAITs presented in the Related Work section. Additionally, those evaluations should expand FAULTBENCH to include more subject programs written in new languages. By applying SAAI to a user study in Python provides evidence that SAAI is applicable to other languages beyond Java. Finally, user studies involving more subjects both in academia and industry will continue to inform the creation and usage of AAIT.
REFERENCES


User Study Questionnaire

Please circle the *best answer*

1) How frequently did you run static analysis during development?
   a. Practically never
   b. Rarely
   c. Every once in a while
   d. A couple of times per coding session
   e. I ran static analysis automatically during development

2) Do you look at the AWARE view for information on code problems identified by static analysis?
   a. Practically never
   b. Rarely
   c. Every once in a while
   d. A couple of times per coding session
   e. I frequently check AWARE view information

3) Complete this statement: “AWARE ______________ my development.”
   a. hinders
   b. helps
   c. sometimes hinders, sometimes helps
   d. has no effect on

4) While developing with AWARE, does the system lag?
   a. Yes, unreasonable delays
   b. A little, but the lag doesn’t affect/annoy me
   c. Not that I’ve noticed
   d. The system lags anyway, regardless of whether I’m using AWARE or not
5) If AWARE notified you that there is a problem in the code, do you investigate it? PLEASE PROVIDE DETAIL on your answer.

   a. I always investigate
   b. I might defer looking at it until a later time
   c. I generally ignore the information

Details:

6) If you have used AWARE to investigate code problems, have you found the problems to be accurate? Do the problems exist in your code and are they worth fixing?

   a. The problems are always accurate
   b. Usually accurate, but sometimes the errors are not actually errors
   c. The “errors” are almost always not an actual problem

7) Do you think that AWARE is doing a good job of giving a high ranking to alerts that are actually errors in the software?

   a. I don’t really use AWARE, so I can’t comment
   b. AWARE is doing a very good job of putting actual errors at the top of the list
   c. AWARE is doing a satisfactory job
   d. AWARE is going OK
   e. AWARE could be better at its ranking
   f. AWARE doesn’t rank the alerts well at all
   g. I never really noticed the ranking

8) Have often do you suppress alerts that are not of interest to you?

   a. Frequently
   b. Occasionally
   c. Once or twice
   d. Never
9) How do you use the AWARE view to decide which errors to investigate? Select all that apply.

   a. I generally look for specific types of alerts
   b. I like to look at the most severe alerts
   c. I generally look at the highest ranked alerts
   d. I just go down the list
   e. I look for alerts in the code that I’m currently working on and fix those
   f. Other. Please explain.

10) Have you found the ranking of alerts in the AWARE view to be more effective than the ordering of the bugs in the regular Eclipse Problems view?

   a. I like AWARE view’s ordering better
   b. I like the Problems view’s ordering better
   c. They’re about the same to me
   d. I don’t really use the AWARE view
   e. I don’t have enough information to compare the two.

11) What did you like most about using AWARE when developing?

12) What did you like least about using AWARE when developing?

13) Please comment about the effectiveness of AWARE at identifying alerts that you wanted to fix.

14) Any other feedback about AWARE.