A Comparative Evaluation of Static Analysis Actionable Alert Identification Techniques

Sarah Heckman
North Carolina State University
Campus Box 8206
Raleigh, NC 27695-8602
+1-919-515-2042
heckman@csc.ncsu.edu

Laurie Williams
North Carolina State University
Campus Box 8206
Raleigh, NC 27695-8602
+1-919-513-4151
williams@csc.ncsu.edu

ABSTRACT
Automated static analysis (ASA) tools can identify potential source code anomalies that could lead to field failures. Developer inspection is required to determine if an ASA alert is important enough to fix, or an *actionable alert*. Supplementing current ASA tools with automated identification of actionable alerts could reduce developer inspection overhead, leading to an increase in industry adoption of ASA tools. The goal of this research is to inform the selection of an actionable alert identification technique for ranking the output of automated static analysis through a comparative evaluation of actionable alert identification techniques. We investigated six actionable alert identification techniques on three subject projects. Among these six techniques, the systematic actionable alert identification (SAAI) technique reported an average accuracy of 82.5% across the three subject projects when considering both ASA tools evaluated. Check 'n' Crash reported an average accuracy of 85.8% for the single ASA tool evaluated. The other actionable alert identification techniques had average accuracies ranging from 42.2%-78.2%.

Categories and Subject Descriptors
D.2.4 [**Software Engineering**]: Software/Program Verification – *statistical methods, validation.*

General Terms
Measurement, Reliability, Experimentation, Verification.

Keywords
Actionable static analysis alert identification, automated static analysis, comparative evaluation

1. INTRODUCTION
Automated static analysis (ASA) tools can identify potential source code anomalies that could lead to field failures or increased maintenance costs. 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" [14]. These potential anomalies identified by ASA, which we call *alerts*, require inspection by a developer to determine if the alert is important enough to fix. If a developer determines the alert is an important, fixable anomaly, then we call the alert an *actionable alert* [3-6, 8-12, 21]. When the developer determines that an alert is not an indication of an actual code anomaly or the developer feels the anomaly is unimportant or inconsequential to the project's functionality, we call the alert an *unactionable alert* [3-6, 8-12, 21].

Static analysis tools generate many alerts with empirical observations reporting an alert density of 40 alerts per thousand lines of code (KLOC) [3, 8]. Empirical observations of the percentage of unactionable alerts vary widely; from 35% to 91% of reported alerts [1, 3, 8-9, 15-18]. 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 the overhead necessary to inspect each alert [3, 16-18]. Suppose a tool reports 1000 alerts and each alert requires five minutes for inspection. Almost ten and a half uninterrupted eight-hour workdays would be required to inspect all 1000 alerts reported by an ASA tool. Using a tool or process to predict the 35% to 91% unactionable alerts before inspection could lead to a corresponding inspection time savings.

Most ASA tools present a list of alerts to developers in an arbitrary order that may not relate to the context of the alert in the source code. Actionable alert identification techniques (AAITs, e.g., [1, 3, 5-6, 8-12, 15-18, 21-23]) supplement ASA to produce a classified or prioritized listing of the alerts most likely to be actionable after ASA has been run on a code base. AAITs utilize models or processes for building project specific-models for identifying actionable alerts. Information about the code under analysis, such as the code surrounding the alert and the development history of the project, called artifact characteristics, are inputs for predicting the actionability of an alert via the AAIT model. *Alert classification* AAITs divide ASA alerts into two groups: alerts likely to be actionable and alerts likely to be unactionable. *Alert prioritization* AAITs order ASA alerts such that alerts likely to be indications of important anomalies are at the top of an alert list presented to developers. Alert prioritization AAITs become classification AAITs when a cut-off value separates the alerts into actionable and unactionable groups.

Nineteen AAITs have been proposed in literature to supplement ASA. The documented evaluations of 18 of these techniques use different metrics, preventing meta-analysis [10] based upon published data. The goal of this research is *to inform the selection of an actionable alert identification technique for ranking the output of automated static analysis through a comparative evaluation of six actionable alert identification techniques.*

PROMISE'13, October 09, 2013 Baltimore, MD, USA

The comparative evaluation utilizes FAULTBENCH [8], a benchmark for evaluating and comparing AAITs. FAULTBENCH v0.3 contains three subject projects (jdom, runtime, and logging); an evaluation process; and evaluation metrics. The contributions of this paper is a comparative evaluation of six AAIT using FAULTBENCH, which builds on a previous comparative evaluation [2]. Our work builds on the previous comparative evaluation of Allier, et al. [2] by 1) considering a classification AAIT in addition to ranking AAIT and therefore using different metrics to evaluate the selection of the "best" AAIT and 2) breaking data into training and test sets by revision, which simulates actual use of AAIT in practice.

The rest of this paper is organized as follows: Section 2 describes Allier, et al.'s [2] comparative evaluation as related work; Section 3 describes the set-up of our comparative evaluation; Section 4 describes the results of the comparative evaluation; Section 5 presents the threats to validity; Section 6 describes future work; and Section 7 concludes.

2. RELATED WORK

Allier, et al. [2] report on a comparative evaluation of six AAIT, five from [10] and a more recent AAIT, EFindBugs [22]. The comparative evaluation investigates ranking algorithms and evaluates the algorithms using an effort metric from [18] and the Fault Detection Rate Curve (FDRC). Effort is defined as the "average number of alerts one must inspect to find an actionable one" in the first x% of ranked alerts, where the ranking is in descending order of actionability. The FDRC plots the number of faults detected against the number of alerts inspected [2, 10]. The subject programs consisted of three Java programs from FAULTBENCH [8] v0.3 (e.g., jdom, runtime, and logging) and three SmallTalk programs [2]. The ASA considered for the Java applications were PMD[1] and FindBugs [13]. The addition of the SmallTalk language required the use of a SmallTalk ASA, specifically SmallLint [20]. The selected AAIT were AWARE (which we call APM) [8], FeedbackRank (FeedbackRank was not considered in our comparative evaluation) [17], RPM (which we call LRM) [20], ZRanking (ZRanking was not considered in our comparative evaluation) [18], AlertLifetime (which we call ATL-R and ATL-D) [15], and EFindBugs (EFindBugs was not considered in our comparative evaluation) [22].

Allier, et al. [2] defined the best AAIT as the AAIT with the lowest measured effort. They measured effort for the first 20%, 50%, and 80% of alerts, and found that for the subject programs, AWARE [8] had the lowest effort followed by FeedbackRank [17]. By comparing the way that the different AAIT analyzed alerts, Allier, et al. [2] conclude that ranking algorithms that consider alerts individually rather than aggregating by the alert type or rule produce better results.

Our work builds on the previous comparative evaluation of Allier, et al. [2] by 1) considering a classification AAIT in addition to ranking AAIT and therefore using different metrics to evaluate the selection of the "best" AAIT and 2) breaking data into training and test sets by revision, which simulates actual use of AAIT in practice.

3. COMPARATIVE EVALUATION SETUP

For the comparative evaluation, we ran the six AAITs on three subject projects in FAULTBENCH [8] v0.3. The goal of this research is *to inform the selection of an actionable alert identification technique for ranking the output of automated static analysis through a comparative evaluation of six actionable alert identification techniques*. The remainder of Section 3 describes the experimental set up for the goal. Section 3.1 describes the FAULTBENCH benchmark and the associated subject projects. Section 3.2 describes the selection of AAIT in the comparative evaluation. Section 3.3 provides the procedure for applying the FAULTBENCH benchmark to the six AAIT under comparison. Section 3.4 describes the evaluation metrics.

3.1 Subjects

FAULTBENCH v0.3 uses three freely-available open source subject projects for the comparative evaluation of AAITs. Table 1 lists the defining characteristics of the three subject projects. Due to the heavy processing load of generating all possible artifact characteristics for all AAIT, we sampled the revisions from a subject's source code repository, starting with the first revision and including the last revision. For jdom and runtime, we sampled every 25th revision. For logging, we sampled every 15th revision. Each checked out revision was automatically built using build scripts and other resources associated with the project. However, several of the checked out revisions would not build, typically early revisions. The ASA tools used in the comparative evaluation required compiling code; therefore, no data were collected for revisions that would not build.

Table 1. FAULTBENCH v0.3 subject projects [8, 12].

	jdom	runtime	logging
Domain	data format	software dev.	logging library
Size (LOC)	9035-13146	2066-15516	355-1785
Time Frame (mm/dd/yy)	05/27/2000 12/17/2008	05/05/2001 08/07/2008	08/02/2001 09/20/2008
# of Revisions	1168	1324	710
# of Sampled Revisions	48	54	48
# Built Revisions	30	48	42
Total Alerts	489	632	91
Actionable Alerts	200	549	31
Unactionable Alerts	254	41	36
Deleted Alerts	35	42	24

Since we are looking at the history of a project, we present a range of sizes, measured by lines of code, for the project's history. The total alerts for a project are the distinct alerts generated across the project history from the ASA tools FindBugs [13] and Check 'n' Crash [6]. Automated classification of alerts as actionable and unactionable is part of the FAULTBENCH process (Section 3.3.2). Alerts associated with files that were deleted over the course of the subject's history were removed from consideration when creating the test sets of alerts. If the alert was not deleted before the training-test division break, the deleted alert was considered part of the training set, modeling how AAIT would be used in practice.

3.1.1 jdom

The subject project jdom[2] is an open source library for XML. The jdom code at the time of analysis was kept in a CVS source code repository. The jdom project consists of three sub-projects, all of

[1] PMD is available at: http://pmd.sourceforge.net/.

[2] jdom may be found at: http://www.jdom.org/.

which were considered for the comparative evaluation: jdom, jdom-contrib, and jdom-test. For the remainder of this paper, the three projects will be referred to collectively as, jdom. All three projects are built using Ant[3].

3.1.2 runtime

The subject project org.eclipse.core.runtime is a core plug-in of the Eclipse[4] integrated development environment. The plug-in, which we shorten to runtime for this paper, was stored in a CVS repository. 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 plug-ins required to build runtime varied over runtime's history. The build step consists of a headless Eclipse build, which means the build started an instance of Eclipse that compiled the projects without loading Eclipse's user interface. Eclipse v3.3.1.1 was used for the headless runtime build. Check 'n' Crash was not run on runtime due to inconsistent library usage throughout the project's history.

3.1.3 logging

The subject project logging is a Java logging library that is part of Apache Commons[5]. The logging project is maintained in a SVN source code repository. All libraries required to build the logging project were obtained through Maven[6] for each build and the project was built using Ant.

3.2 AAIT

Heckman and Williams [10] identify 18 AAIT that classify or prioritize alerts after an ASA run. Of the 18 listed AAIT, we selected six for a comparative evaluation using the FAULTBENCH benchmark [8]. The selected AAIT meet the following criteria:

- The AAIT would classify or prioritize alerts generated by ASA for the Java programming language;
- An implementation of the AAIT is described in the literature allowing for replication;
- The AAIT is fully automated and does not require manual intervention or inspection of alerts as part of the process.

The above criteria lead to the selection of APM [8], AlertLifetime (ATL) [15], Check 'n' Crash (CnC) [6], HW09 (SAAI) [9], HWP [16], and RPM08 (LRM) [21]. EFindBugs [22], a model introduced after Heckman's and Williams' literature review [10], was excluded because the technique requires a manual inspection of alerts to initialize the *defect likelihood* for ranking each alert type.

3.2.1 Actionable Prioritization Models (APM)

Heckman and Williams [8] propose an AAIT that is an adaptive model for prioritizing individual ASA alerts using the alert's type and location in the source code. The adaptive prioritization model (APM) re-ranks alerts after each developer inspection, which incorporates developer's feedback about inspected alerts into the model. The APM ranks alerts on a continuous scale from -1 to 1, where alerts close to -1 are more likely to be unactionable, and alerts close to 1 are more likely to be actionable. When no prediction about the actionability of an alert can be made, the alert is given a ranking of 0. For example, a prediction of 0 would be returned when the alert's type and location have not been inspected before. The APM model makes the assumption that alerts sharing an alert type or code location are likely to be all actionable or all unactionable, i.e., the alerts are expected to be homogeneous within a group that share a characteristic. The assumption comes from the observations of homogeneous alerts at a given location by Kremenek, et al. [17].

3.2.2 Alert Type Lifetime (ATL)

Kim and Ernst [15] propose an AAIT process for building subject-specific models using the alert type's lifetime (ATL). ATL models prioritize alert types by the average lifetime of alerts sharing the same type over the history of the subject project. The lifetime of an alert or set of alerts sharing the same type in each file is measured, in days, from the time the first instance of an alert type appears until closure of the last instance of an alert type. The premise is that alerts of a given type that are fixed quickly are more important to developers. Therefore, a newly-generated alert is expected to be actionable if the alert's type has had a shorter average lifetime throughout the project's history. Alert types that remain in the file at the last studied revision are penalized 365 days. One limitation of ATL is that the assumption that important alerts are fixed quickly may be incorrect. Instead, alert types that are fixed quickly may be associated with the easiest underlying faults to fix and not the most important alerts [15].

Two variations of the ATL model are considered in the empirical comparison of AAIT for this research. Kim and Ernst [15] measure the average alert type lifetime in days, which is represented by model ATL-D. When using a model that measures time by days, some alert types may demonstrate a longer alert lifetime due to a long period of development inactivity. Considering the number of revisions, as recorded by a source code repository, between changes is an alternative measurement of time during development that would mitigate the potential problem with gaps in development activity. We have created a variation of Kim and Ernst's ATL-D model, ATL-R, which uses the same technique as ATL-D to generate the alert type lifetime but uses revisions rather than days. For alerts that are never closed, the alert's lifetime is penalized by adding a penalty value of 365 days for ATL-D and half of the maximum revisions in the project's lifetime for ATL-R. Because ATL is a prioritization model a cutoff value allows for the classification of the alerts into actionable and unactionable groups. For the ATL models, we consider the penalty value as the cutoff for alert classification, which is similar to how actionable and unactionable alert oracles (Section 3.3.2) are generated.

3.2.3 Check 'n' Crash (CnC)

Csallner, et al. [6] propose an AAIT tool, called Check 'n' Crash (CnC) that generates automated unit test cases from some ASA alerts associated with runtime exceptions generated by the ESC/Java [7] ASA tool. If a test case fails, then the associated alert is actionable. The test case provides a set of inputs that lead to a runtime exception, and may help identify the underlying fault. The severity of the alert is determined from the test failure information. An alert with an associated test failure receives a severity of 1. If test cases associated with an alert did not fail, then the severity of the alert is 3. In some cases, tests were not generated for the alerts due to the implementation of the Check 'n' Crash ASA. Those

[3] Ant is a build library for Java programs: http://ant.apache.org/.

[4] Eclipse is an open source integrated development environment that may be found at: http://www.eclipse.org/

[5] Apache Commons is a top-level Apache project containing common libraries. Logging may be found at: http://commons.apache.org/logging/.

[6] Maven is a build management tool: http://maven.apache.org/.

alerts received a priority of 2. Alerts given a priority of 1 were classified as actionable, and all other alerts were classified as unactionable. Due to inconsistent library usage, CnC was not run on the runtime subject project.

3.2.4 History-Based Warning Prioritization (HWP)

Kim and Ernst [16] propose an AAIT process for building subject-specific models using the commit messages and code changes in the source code repository to prioritize alert types. The history-based warning prioritization (HWP) weights alert types by the number of alerts closed by fault- and non-fault fixes as recorded in the source code repository. 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, α. Kim and Ernst [16] considered three values for α: 0.5, 1, and 0.9. For each non-fault-fix, the weight increases by 1-α. The final step normalizes the sum of the fix change weights for an alert type by the number of alerts sharing the type. A higher weight implies that alerts with a given type are more likely to be actionable due to fault fixes. 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 alert types that do not have at least one associated fault-fix or non-fault-fix, which may happen when the alert has never appeared over the course of the project's history used to build the model.

Unlike Kim and Ernst [16] who gather fault-fix data for every revision, our implementation of HWP gathers ASA alerts from every 25th revision and the final revision. The time required to run the analysis for every revision was prohibitively expensive. Therefore, we cannot determine atomic fault-fixes. We define a fault-fix as a set of changes where at least one of the changes in the 25 covered revisions was a fault-fix in that the change fixed a bug or other problem. We used the following keyword bases (e.g. we allowed for endings like 's', 'ed', or 'ing') to identify fault-fixes in the repository commit notes: bug, fix, patch, bugfix. The value α was set to 0.9; the value used in Kim and Ernst's experiments [16].

3.2.5 Logistic Regression Models (LRM)

Ruthruff, et al. [21] use a logistic regression model (LRM) to predict actionable alerts. Thirty-three artifact characteristics are considered as independent variables for LRM. Ruthruff, et al. [21] generate logistic regression models that predict the probability an alert is actionable (or unactionable) given a set of artifact characteristics. Reducing the number of characteristics from 33 to a more manageable number of at least six factors for inclusion in the LRM is done via a screening process. Ruthruff, et al. [21] use the cut-off of at least six factors "to ensure we were left with at least some factors for model building."

There are four stages to the screening methodology. At each stage of screening artifact characteristics for inclusion in the final model, an increasing percentage of alerts are used to build a model. Additionally, the contribution of each alert to the generated model is measured using an Analysis of Deviance. Artifact characteristics with p-values for the Chi-squared tests less than a specified cutoff point are moved to the next phase of screening. Artifact characteristics that do not have a p-value less than the cutoff are eliminated from the model because the effect of that factor was negligible in predicting the actionability of an alert. Table 2 presents the percentage of alerts considered and the p-value cut off for each stage of screening. The screening process ends when fewer than six artifact characteristics remain or when the 4th stage is completed. Final model creation occurs on a sample of the alerts. A limitation of the LRM is that the resulting model may contain collinearity because the screening process may select related artifact characteristics [21].

Table 2. Cutoff information for screening stages [21]

Stage	Percentage of Alerts	p-value cutoff
1	5%	0.80
2	25%	0.50
3	50%	0.20
4	100%	0.10

For our implementation of the LRM model, the only exceptions made to the screening process described occurred when there were no artifact characteristics with a p-value less than or equal to the cutoff. When all the p-values for all artifact characteristics was greater than the cutoff, we either selected the artifact characteristic with a p-value less than 1.0, or if all p-values were 1.0, then we moved to the second stage of the screening process.

We consider 30 of the 33 artifacts utilized by Ruthruff et al. [21], with the following exceptions. We do not consider the alert's depth in the source file because our file size measurements only consider source lines rather than comments and whitespace. The actual line number of an alert may exceed the measured size of the file, creating an incorrect measure of file depth. Additionally, we consider the cyclomatic complexity of the alert's method (if the alert is in a method) rather than indentation because indentation may not provide the correct measure of cyclomatic complexity for a method as a whole due to possible incorrectness or inconsistency in indentation by different developers. Finally, we do not consider the BugRank and BugRank Range artifact characteristics. These two values are internal measures created by Google; therefore, we do not have access to the method for calculation. The screening process, model building, and model evaluation for LRM were performed using R[7] 2.6.2. If the generated model contained a categorical variable and the test set contained values not in the training set, those alerts in the test set were not classified in the evaluation. Therefore, the total count of alerts evaluated could be fewer than the total number of alerts for a treatment. These numbers are reported in the evaluation section where appropriate.

3.2.6 Systematic Actionable Alert Identification (SAAI)

Heckman and Williams [9, 12] present a process for using machine learning techniques to identify key artifact characteristics and the best models for prioritizing static analysis alerts for specific projects. The process, called systematic actionable alert identification (SAAI) consists of four steps: 1) gathering artifact characteristics about alerts generated from static analysis; 2) selecting important, unrelated sets of characteristics; 3) using machine learning algorithms and the selected sets of characteristics to build candidate models; and 4) selecting the best candidate model using classification evaluation metrics, with an emphasis on

[7] R is an open source statistical computing environment: http://www.r-project.org/.

accuracy and precision. Models that predicted that all alerts were either actionable or unactionable were not considered best models.

SAAI considers 42 artifact characteristics during model building [12]. The open source machine-learning library, Weka [24], automated the SAAI process for artifact characteristic selection and generation of the candidate models. For Step 2, we considered 16 attribute selection algorithms to identify the most predictive artifact characteristics [9]. Attribute selection algorithms combined a search strategy (e.g. cfs subset, classifier, consistency, and wrapper) for searching the artifact characteristics space and an evaluation method (e.g. best first, greedy, and rank search using either information gain or the gain ratio of each artifact characteristic) for identifying the best subset of artifact characteristics [9, 24]. Many of the attribute selection algorithms included steps to reduce collinearity or redundant artifact characteristics (e.g., cfs subset) or included cross-validation (e.g., wrapper) [24]. For each of the artifact characteristic subsets, we generated 16 candidate models using the following machine learning algorithms: BayesNet, NaïveBayes, ADTree, J48, LMT, REPTree, ConjunctiveRule, Decision Table, JRip, PART, Ridor, SimpleLogistic, IB1, IBk (where k=2), KStar, and LWL [9, 24]. The first three steps outlined above were automated.

Step 4 required evaluation by the researchers to select the "best" model out of 256 candidate models. The best model for SAAI was selected by first considering accuracy, followed by the precision, recall, area under the curve, f-measure, and error rates, all generated by Weka [24]. Since SAAI is a classification model, we evaluated accuracy, precision, and recall first. For SAAI, the best model was checked to ensure that one or more of the selected artifact characteristics were used in the model to ensure the model is not simply classifying all alert as actionable or unactionable. A model without any selected artifact characteristics would classify alerts as either actionable or unactionable from the majority type in the training set. Since there are many models to choose from, another model with a similar level of evaluation metrics could be selected. The phenomena of a model with no artifact characteristics was seen in practice, but was never the best model for the subject projects.

3.3 FAULTBENCH Procedure

This section describes the specifics of implementing the five-step FAULTBENCH procedure for the comparative evaluation of the six AAIT. Each step of the FAULTBENCH procedure is described in the following subsections.

3.3.1 Step 1: Gather Alert and Artifact Characteristic Data Sources

The first step of the FAULTBENCH procedure generated the primary sources of data for the evaluation of all AAITs. The automation included the checkout of the subject project's source code from a repository, the generation of ASA alerts, and the artifact characteristics.

For the comparative evaluation, we ran two ASA tools on the subject projects: FindBugs [13] and Check 'n' Crash [6]. FindBugs is an open source ASA tool that generates an alert for an area of code that matches a bug pattern. Check 'n' Crash generates JUnit test cases from alerts generated by ESC/Java [7], runs these test cases, and reports test failures to the developer. Check 'n' Crash focuses on alerts that may generate runtime exceptions, and therefore crash a program, which is a subset of the alerts that ESC/Java can generate. Check 'n' Crash was used in the comparative evaluation as both a static analysis tool and as one of the AAIT under evaluation. For the comparative evaluation, all of the alerts generated by Check 'n' Crash, were considered for the AAITs. The passing or failing of the generated JUnit test cases was only considered when evaluating the Check 'n' Crash AAIT. From this point forward, we will refer to the ASA tool as Check 'n' Crash and the associated AAIT under evaluation as CnC.

The FindBugs alerts are recorded in XML files. The alerts generated by Check 'n' Crash through the underlying ESC/Java ASA and the failing test cases for CnC were recorded in text files. The alerts generated by both ASAs provide the following artifact characteristics: package name, file name, method signature, alert type, alert category, priority, file extension, line number, marker type, test file (in the case of Check 'n' Crash), and severity (in the case of Check 'n' Crash's failing test cases).

Generation of the artifact characteristics classified as software metrics were generated using JavaNCSS[8]. JavaNCSS is an open source tool for generating source code metrics, specifically non-commented source statements (NCSS), method and class counts, and cyclomatic complexity. The data generated by JavaNCSS provide the following artifact characteristics: size, number of methods (at the file and package levels), number of classes (at the file and package levels), and cyclomatic complexity.

The artifact characteristics related to the source code history and the source code churn were generated by processing the revision histories of the subject projects. The revision histories were mined from the source code repository. The specific metrics for these categories were generated during Step 2. The aggregate artifact characteristics, including the number of alert modifications, were also generated from the above data sources in Step 2.

The ASA tools and JavaNCSS required compiled projects. The subject projects were initially compiled with Java 1.5 revision 17. If the project built correctly, FindBugs v1.3.8 and the metrics program, JavaNCSS v29.50, were run on the code. Some of the constructs in very early version of the runtime subject project were not understandable to JavaNCSS v29.50. If a parsing error occurred when running JavaNCSS v29.50, the metrics were rerun using an earlier version of JavaNCSS, v21.41.

3.3.2 Step 2: Artifact Characteristic and Alert Oracle Generation

The data processing step consists of two parts: 1) associating the artifact characteristics with each generated alert; and 2) generating the alert oracles. Not all of the artifact characteristics come directly from the data files generated as part of Step 1. Therefore, some additional processing must be done to generate the artifact characteristics, the details of which are presented in [9].

The alert oracles (i.e. the actual actionability of the alert) were automatically generated by traversing the history of the alerts in a subject project as available in the sampled revisions. If an alert was opened and closed between sampled revisions, the alert is not considered in the study. Model generation for use in the field should consider all revisions in the history or all revisions for a window in the project's history when model building.

[8] JavaNCSS may be found at: http://www.kclee.de/clemens/java/javancss/.

The state of the alert at the last revision is the alert's oracle. If the alert has been closed, the alert is classified as actionable. If the alert has been suppressed, the alert is classified as unactionable. If the alert is still open at the last revision, the alert is classified as unactionable, even if the alert was just opened in the latest few revisions. The assumption for classifying all open alerts as unactionable is that if developers have not fixed the anomaly associated with the alert during the history of the project thus far, the alert must not be important to the developer(s) for the project at that point in time.

3.3.3 Step 3: Training and Test Sets

For Step 3 of the FAULTBENCH procedure, we considered three treatments for training and testing AAITs. Each treatment splits the alerts for a subject project into two sets: one set is used to generate an AAIT model (e.g. the training set) and the second set is used to evaluate the model (e.g. the test set). The treatments simulate how an AAIT could be used in practice by using the alerts in the first X% of revisions to train the models and the alerts in the remaining revisions to test the models. We considered training data for the first 70, 80, and 90% of revisions. We expect that most industry teams that would adopt an AAIT technique would start by evaluating their project's history for a given ASA tool, and use the generated model on future revisions. By picking some percentage of revisions from the project history to build the model, we are able to test the model on "new" data that arrives after the cutoff revision, more closely simulating how we think developers would use the AAIT in practice.

The set of alerts open and closed at an X% revision are used to train a model and the model is applied to the set of alerts opened and closed after the revision. The training and test sets do contain overlapping alerts: those alerts open at the cutoff revision. While the overlap is a threat to validity, the overlap would occur when using an AAIT in addition to ASA in practice. Additionally, the training set may include alerts that are removed from analysis in the test set because of a file deletion. The numbers of alerts in the training and test sets for each subject projects are provided in Section 4's discussion of the comparative evaluation.

3.3.4 Step 4: Model Building

Step 4 builds a model for each AAIT and each treatment using the training set of alerts. For the LRM model, the alerts selected at each stage of the screening process were selected randomly. Due to the random selection of alerts from the test set to generate the model, the LRM model building process was repeated three times. The most predictive model was selected for the comparative evaluation.

3.3.5 Step 5: Model Evaluation

For Step 5, the generated models were evaluated using the test set of alerts, the set of alert oracles, and the evaluation metrics listed in Section 3.4. The SAAI technique generated and evaluated 256 different models [9]. Selection of the "best" model for SAAI is discussed in Section 3.2.6.

3.4 Evaluation Metrics

FAULTBENCH [8] v0.3 provides both classification and prioritization metrics for evaluating AAITs. Since we are evaluating one AAIT that only generates classifications rather than rankings, we only consider classification metrics in our evaluation.

Alert classification techniques predict whether alerts are actionable or unactionable. After classification is complete, we evaluate the classification by comparing the predicted value with the actual value for each alert in the test set. Figure 1 shows the possible outcomes when comparing predicted and actual values.

Using the classification metrics listed in Figure 1, we can calculate several other measures [8-10, 12, 16, 23-25], which evaluate how well AAITs classify static analysis alerts as actionable or unactionable. The model's prediction of actionable and unactionable (on the rows) is compared with the generated alert oracles (on the columns), as discussed in Section 3.3.2. The precision, recall, and accuracy calculations only consider the alerts in the test set. When evaluating the AAIT, we look at accuracy, precision, and then recall. Accuracy tells us how well the models are classifying the alerts. When using ASA, many FPs may discourage use of the ASA tool [4]. 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. However, from a security perspective, missing a TP may be critical [4], and other metrics (e.g., those in [19]) could be used.

		Anomalies are observed (oracle)	
		Actionable	Unactionable
Model predicts alerts	Actionable	True positive (TP)	False positive (FP)
	Unactionable	False negative (FN)	True negative (TN)

Figure 1: Classification Table [8, 12] (adapted from Zimmerman, et al. [25])

- **Precision [9]**: the proportion of correctly classified anomalies (TP) out of all alerts predicted as anomalies (TP + FP). The precision calculation is presented in Equation 1.

$$precision = \frac{TP}{TP + FP} \quad (1)$$

- **Recall (also called True Positive Rate or Sensitivity) [9]**: the proportion of correctly classified anomalies (TP) out of all possible anomalies (TP + FN). The recall calculation is presented in Equation 2.

$$recall = \frac{TP}{TP + FN} \quad (2)$$

- **Accuracy [9]**: the proportion of correct classifications (TP + TN) out of all classifications (TP + TN + FP + FN). The accuracy calculation is presented in Equation 3.

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} \quad (3)$$

4. COMPARATIVE EVALUATION RESULTS

We present the evaluation results for each subject project below.

4.1 jdom

The jdom project contained 489 total alerts generated by FindBugs and Check 'n' Crash. Of the 489 alerts, 200 were actionable, which means the alert was closed due to a code change during the history of the project. Two hundred and fifty-four (254) of the jdom alerts were unactionable because they remained open at the end of the analysis period. The remaining 35 alerts were associated with files that were deleted over the course of the project; however, if the file

deletion occurred in the revisions covered by the test set, the alert was considered when training the model.

Tables 3 and 4 present the training and test data about FindBugs and Check 'n' Crash alerts reported in jdom. Alerts in the training set are not distinct from alerts in the test set. Any alert open at the last revision considered for the training set is also part of the test set. The number of new alerts decreased over time in the jdom project as the project reached maturity. There were only 31 additional alerts in the training data between treatment 80 and treatment 90. The number of alerts in the test set of the LRM model differed from the number of alerts in the test sets of the other models. If the LRM model included an artifact characteristic that contained categorical data, like the bug type, there may have been test data that contained bug types not in the training set; therefore, no prediction could be made when using Weka's models. Alerts with new categories could be provided a default prediction when a model is used in practice. Alerts with bug types not in the training set were removed from the test set. At treatment 70, there were 264 alerts in LRM's test set. At treatment 80, there were 297 alerts in LRM's test set, and at treatment 90, there were 279 alerts in LRM's test set.

Table 3: jdom Training Data

Treat.	Train Total	CnC Total	CnC Act.	FB Total	FB Act.	% Act.
70	339	65	37	274	142	53%
80	405	66	37	339	153	47%
90	436	75	38	361	161	46%

Table 4: jdom Test Data

Treat.	Test Total	CnC Total	CnC Act.	FB Total	FB Act.	% Act.
70	318	55	16	263	48	20%
80	302	50	11	252	37	16%
90	284	41	2	243	28	11%

The results of calculating the evaluation metrics for jdom are summarized in Table 5. The cells highlighted in light grey are the AAIT with the highest accuracy for each treatment and are considered to be the best models. If CnC was the AAIT with the highest accuracy for a given treatment, the second highest AAIT is also highlighted since CnC is limited to only the Check 'n' Crash alerts.

Table 5: jdom Evaluation Metrics

	Accuracy (%)			Precision (%)			Recall (%)		
Rev.	**70**	**80**	**90**	**70**	**80**	**90**	**70**	**80**	**90**
APM	80	83	87	46	42	0	9	10	0
ATL-D	72	83	88	26	20	20	22	2	3
ATL-R	77	81	86	32	24	24	11	8	13
CnC	73	80	95	100	100	0	6	9	0
HWP	31	35	32	19	15	9	73	67	57
LRM	72	76	83	37	35	32	64	55	59
SAAI	83	86	90	92	100	67	16	13	7

The overall average accuracy of the AAIT increased from 69.7% at treatment 70 to 80.25% at treatment 90. Individually, all AAIT followed the increase of accuracy over the treatments, except for HWP. One possibility for the decrease in accuracy for HWP at treatment 90 may be due to the maturity of the jdom project. If most of the alerts were closed due to fault fixes, the model may predict many unactionable alerts as actionable, which is supported by the precision numbers.

The AAIT with the highest accuracy for identifying actionable and unactionable Check 'n' Crash and FindBugs alerts was SAAI. SAAI identified the most actionable alerts while minimizing FPs and FNs. LRM and HWP did better numerically at identifying actionable alerts, which is demonstrated with a high recall, but at the cost of a large number of FPs, leading to a lower accuracy and precision. At treatment 70, LRM identified 32 actionable alerts and HWP identified 47 actionable alerts while SAAI only identified 10 actionable alerts. LRM predicted approximately two FPs for every TP. HWP predicted four to ten FPs for every TP. ATL-D and ATL-R also had a high ratio of FP alerts to TP alerts, which were at three to four FPs to TPs and two to three FPs to TPs, respectively.

The ratio to FP to TP alerts is also reflected in the precision of the AAIT for each treatment. The average precision of all models decreased across treatments, from 50.1% at treatment 70 to 21.56% at treatment 90. SAAI reports the highest precision when alerts from both ASA are considered.

Like the average precision, the average recall decreased across treatments, from 28.8% on average at treatment 70 to 19.8% on average at treatment 90. The recall was low for APM, ATL-D, ATL-R, CnC, and SAAI for treatments 70 through 90 implying that the models, especially those with higher accuracy, are better at identifying unactionable alerts than actionable alerts.

The precision and recall of the AAITs decline as more revisions are considered for the training set. Numerically, the reason is that there are very few actionable alerts compared with many unactionable alerts in the test sets at treatment 90. For treatment 90, there are 30 actionable alerts and 254 unactionable alerts. The high accuracy at treatment 90 shows the models do a good job with predicting unactionable alerts, but the low precision and recall shows that the models cannot identify the few actionable alerts in the test set. These results may imply that using most of the project history to predict the latest revisions may not be useful. The precision was much higher for treatments 70 and 80, which use less of the revision history for model building. However, at treatments 70 and 80 the recall was low for most models showing that actionable alerts were not well identified. Another reason for the poor precision and recall at treatment 90 may be that the set of actionable alerts in the test set are unrelated to any alerts in the training set. Further analysis of the false negatives at treatment 90 could lead to additional insights into the low precision and recall and better AAIT models.

4.2 Runtime

The runtime project contained 632 total alerts generated by FindBugs. Of the 632 alerts, 549 were actionable, which means the alert was closed due to a code change during the history of the project. Forty-one (41) of the runtime alerts were unactionable, and remained open at the end of the analysis period. The remaining 42 alerts were associated with files that were deleted over the course of the project; however, if the file deletion occurred in the revisions covered by the test set, the alert was considered when training the model.

Tables 6 and 7 present the training and test data about FindBugs alerts reported in runtime. Alerts in the training set are not distinct from alerts in the test set. Any alert open at the last revision considered for the training set is also part of the test set. The number

of alerts in the test set of the LRM model differed from the number of alerts in the test sets of the other models. If the LRM model included an artifact characteristic that contained categorical data, like the bug type, there may have been test data that contained bug types not in the training set. Alerts with bug types not in the training set were removed from the test set. For treatment 70, there were 325 alerts in LRM's test set. For treatment 80, there were 286 alerts in LRM's test set. For treatment 90, there were no alert types in the test set not in the training set, so no alerts were removed.

Table 6: runtime Training Data

Treat.	Train Total	Deleted	Actionable	% Actionable
70	476	11	440	92%
80	554	16	513	93%
90	596	12	549	92%

Table 7: runtime Test Data

Treat.	Test Total	Actionable	% Actionable
70	344	303	88%
80	290	249	86%
90	76	35	46%

Table 8 presents runtime evaluation metrics. The cells highlighted in grey are the AAIT with the highest accuracy for each treatment and are considered to be the best models. The average accuracy of the models increased from 49.0% at treatment 70 to 57.7% at treatment 90, mirroring the increase in average accuracy for jdom. The accuracy of the AAITs on runtime did not generally increase or decrease for all models across treatments. For runtime, there were many more actionable alerts in the test set than unactionable alerts; however, most models had a difficult time identifying the difference between actionable and unactionable alerts. LRM reported the highest accuracy at treatments 70 and 80 while SAAI reported the highest accuracy at treatment 90.

Table 8: runtime Evaluation Metrics

	Accuracy (%)			Precision (%)			Recall (%)		
AAIT	**70**	**80**	**90**	**70**	**80**	**90**	**70**	**80**	**90**
APM	36	23	50	88	70	47	32	17	57
ATL-D	18	17	55	92	82	100	8	4	3
ATL-R	34	43	59	93	94	55	27	36	60
HWP	68	66	46	88	85	45	74	73	83
LRM	88	87	53	88	87	49	100	100	100
SAAI	49	65	83	90	91	100	48	66	63

The average precision of AAITs on runtime decreased from 89.8% at treatment 70 to 66.1% at treatment 90. The precision for the individual models generally decreased over treatments except for ATL-D and SAAI. ATL-D and SAAI both reported perfect precision at treatment 90; however, SAAI identified 22 of the 35 actionable alerts while ATL-D found only one of the 35 actionable alerts. For all treatments, there was no more than 1.2 FP alerts identified for each TP. 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. The HWP and LRM models tended to identify most alerts as actionable leading to a high number of FPs

The average recall of AAITs on runtime increased from 48.1% at treatment 70 to 60.9% at treatment 90. 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 2.8% to 83.8%. Most of the models had a high number of FNs, but fewer FPs than HWP and LRM.

4.3 logging

The logging project contained 91 total alerts generated by FindBugs and Check 'n' Crash. Of the 91 alerts, 31 were actionable, which means the alert was closed due to a code change during the history of the project. Thirty-six (36) of the logging alerts were unactionable, and remained open at the end of the analysis period. The remaining 24 alerts were associated with files that were deleted over the course of the project; however, if the file deletion occurred in the revisions covered by the test set, the alert was considered when training the model.

Tables 9 and 10 present the training and test data about FindBugs alerts reported in logging. The LRM dataset required no removal of alerts due to categorical coefficients.

Table 9: logging Training Data

Treat.	Train Total	CnC Total	CnC Act.	FB Total	FB Act.	% Act.
70	65	9	5	56	26	48%
80	66	9	5	57	26	47%
90	66	9	5	57	26	47%

Table 10: logging Test Data

Treat.	Test Total	CnC Total	CnC Act.	FB Total	FB Act.	% Act.
70	39	6	2	33	1	8%
80	37	4	0	33	1	3%
90	36	4	0	32	0	0%

The logging evaluation metrics are summarized in Table 11. The cells highlighted in light grey are the AAIT with the highest accuracy for each treatment and are considered to be the best models. If CnC was the AAIT with the highest accuracy for a given treatment, the second highest AAIT is also highlighted since CnC is limited to only the Check 'n' Crash alerts.

Table 11: logging Evaluation Metrics

	Accuracy (%)			Precision (%)			Recall (%)		
AAIT	**70**	**80**	**90**	**70**	**80**	**90**	**70**	**80**	**90**
APM	85	89	92	0	0	0	0	0	0
ATL-D	92	97	100	0	0	0	0	0	0
ATL-R	92	97	100	0	0	0	0	0	0
CnC	67	100	100	0	0	0	0	0	0
HWP	32	35	33	8	4	0	100	100	0
LRM	77	84	83	25	14	0	100	100	0
SAAI	90	97	100	0	0	0	0	0	0

The average accuracy increased from 76.4% at treatment 70 to 86.9% at treatment 90. For logging, there were only three actionable alerts in treatment 70, one actionable alert in treatment 80, and no actionable alerts in treatment 90. Therefore, the high accuracy reported for ATL-D, ATL-R, and SAAI are from correctly identifying the unactionable alerts.

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. The precision of all AAIT is zero at treatments 90 because there were no actionable alerts in the test sets.

Both LRM and HWP had 100% recall at treatments 70 and 80, suggesting that these models are good at finding the few actionable alerts in the test set. However, by finding all of the actionable alerts, the LRM and HWP models also predicted that many of the unactionable alerts were actionable. HWP identified many more FPs than LRM. All AAIT had recall of zero at treatments 90 because there were no actionable alerts in the test sets.

The logging project is a unique project since there were so few actionable alerts in the test set. The lack of actionable alerts may be due to the maturity of the project. The project is also smaller than the other subject programs. The logging project may be a better subject for AAIT that seek to maximize recall over accuracy and precision. We will reevaluate the inclusion of logging as a FAULTBENCH subject.

5. Threats to Validity

Many threats to validity come from FAULTBENCH limitations [8]. The following are threats to the validity of our work:

5.1 Construct Validity

There could be a threat to construct validity through our measurement of and calculations of the artifact characteristics. The build of every sampled revision was automated as was the collection of artifact characteristics and the building of the AAIT models. While the automation was tested, there could be flaws. The tools to automate FAULTBENCH are available online at http://www.realsearchgroup.org/faultbench/.

5.2 Internal Validity

A threat to internal validly is from the algorithm to generate alert oracles. If the alert was closed during the history of the project then the alert was considered actionable unless the alert was closed due to a file deletion. All other alerts were considered unactionable because they had not yet been fixed by the developers on the project. An alert just opened would be considered unactionable even if the alert has not yet been inspected by a developer and would eventually be fixed. Due to sampling the revisions, some alerts may be missed; specifically those that were opened and closed between two sampled revisions. Sampling was done to reduce data collection time. Running data collection processes in parallel or collecting data on all revisions in a smaller window of history can further inform the prediction capabilities of AAITs.

Inspection of the alerts would provide a more accurate oracle of developer actions. Obtaining actual developer's actions would require input from the subject project developers, which is not possible due to studying the history of the project. Future work could incorporate studies of more recent projects where ASA is actively used and decisions for filtering or fixing alerts are recorded.

The percentage of revisions used to generate the training and test sets of data are another threat to internal validity. The goal of the threshold is to model how people would use the tools in practice, but different cutoffs may lead to different results. Due to the definition of the alert oracles, there were alerts that were in both the training and test set: those alerts open at the cutoff revision. Because the alert is in the training set, the alert may be easier to predict in the test set.

5.3 External Validity

The threat to external validity is the generalizability of the results, which is partially mitigated through the use of FAULTBENCH v0.3. Additional work to expand the FAULTBENCH benchmark would reduce threats to external validity on future studies of AAIT. Additional threats to validity come from the AAITs themselves and the modifications made to the algorithms due to the post-hoc nature of the comparative evaluation. The limitations to each of the AAITs are discussed in Section 3.2.

6. FUTURE WORK

Future work will incorporate studies of more recent projects where ASA is actively used and alerts are actively filtered. Additional work will focus on identifying which artifact characteristics are most predictive of actionable and unactionable alerts, which will contribute to a better AAIT. Analysis may find that a combination of AAIT models, switching between AAIT from project characteristics, or using a smaller window of history to create models may be beneficial when using AAIT.

While the accuracy tended to increase when considering more of the alerts in a project's history, the precision and recall tended to decrease suggesting that the AAIT were better at identifying unactionable alerts than actionable alerts. Additional analysis of the alerts correctly and incorrectly identified as actionable could suggest other artifact characteristics and model possibilities for AAIT. Consideration of additional evaluation metrics, as in [19], will provide another perspective on interpreting model results when the goal is detection of all actionable alerts.

Expansion of FAULTBENCH to include more subject projects and better selection of alert oracles will decrease the threats to external validity on future comparative evaluations. Additional subject programs may allow for evaluation of AAIT with different goals.

7. CONCLUSIONS

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, and recall. We considered three 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. The AAIT were evaluated by comparing the accuracy, precision, and recall of the actionable and unactionable classification with the alert oracles as defined in section 3.3.2. Accuracy evaluates how well an AAIT identified both actionable and unactionable alerts, and for this study, was considered the most important of the measures. Precision evaluates how well actionable alerts are identified within the set predicted as actionable. Too much false positive identification may lead to rejection of the ASA tool.

SAAI was found to be the best overall model by having the highest accuracy or tying for the highest accuracy on six of the nine treatments. ATL-D, ATL-R, and LRM followed by having the highest accuracy or tying for the highest accuracy on three, three, and two of the treatments, respectively. CnC also had the highest accuracy on three of the nine treatments; however, CnC only considered alerts from one of the two ASA used in the study. LRM and HWP tended to have the highest recall by identifying more actionable alerts, but at the cost of a large number of FPs. Those interested in increasing recall may want to consider LRM or HWP.

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 and where developers are available to generate alert oracles.

8. ACKNOWLEDGMENTS

This work was supported by the first author's IBM PhD Fellowship.

9. REFERENCES

[1] A. Aggarwal and P. Jalote, "Integrating Static and Dynamic Analysis for Detecting Vulnerabilities," Proceedings of the 30th Annual International Computer Software and Applications Conference, Chicago, Illinois, USA, September 17-21, 2006, pp. 343-350.

[2] S. Allier, N. Anquetil, A. Hora, S. Ducasse, "A Framework to Compare Alert Ranking Algorithms," 2012 19th Working conference on Reverse Engineering, Kingston, Ontario, Canada, October 15-18, 2012, p. 277-285.

[3] C. Boogerd and L. Moonen, "Prioritizing Software Inspection Results using Static Profiling," Proceedings of the 6th IEEE Workshop on Source Code Analysis and Manipulation, Philadelphia, PA, USA, September 27-29, 2006, pp. 149-160.

[4] B. Chess and G. McGraw, "Static Analysis for Security," in IEEE Security & Privacy. vol. 2, no. 6, 2004, pp. 76-79.

[5] C. Csallner and Y. Smaragdakis, "Check 'n' Crash: Combining Static Checking and Testing," Proceedings of the 27th International Conference on Software Engineering, St. Louis, MO, USA, 2005, pp. 422-431.

[6] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-Crasher: A Hybrid Analysis Tool for Bug Finding," ACM Transactions on Software Engineering and Methodology, vol. 17, no. 2, pp. 1-36, April, 2008.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, Berlin, Germany, 2002, pp. 234-245.

[8] S. Heckman and L. Williams, "On Establishing a Benchmark for Evaluating Static Analysis Alert Prioritization and Classification Techniques," Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany, October 9-10, 2008, pp. 41-50.

[9] S. Heckman and L. Williams, "A Model Building Process for Identifying Actionable Static Analysis Alerts," Proceedings of the 2nd IEEE International Conference on Software Testing, Verification and Validation, Denver, CO, USA, 2009, pp. 161-170.

[10] S. Heckman and L. Williams, "A Systematic Literature Review of Actionable Alert Identification Techniques for Automated Static Code Analysis," *Information and Software Technology*, vol. 53, no. 4, April 2011, p. 363-387.

[11] S. S. Heckman, "Adaptively Ranking Alerts Generated from Automated Static Analysis," in ACM Crossroads. vol. 14, no. 1, 2007, pp. 16-20.

[12] S. S. Heckman, A Systematic Model Building Process for Predicting Actionable Static Analysis Alerts, Dissertation, Computer Science, North Carolina State University, 2009.

[13] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, British Columbia, Canada, October 24-28, 2004, pp. 132-136.

[14] IEEE, "IEEE 1028-1997 (R2002) IEEE Standard for Software Reviews," 2002.

[15] S. Kim and M. D. Ernst, "Prioritizing Warning Categories by Analyzing Software History," Proceedings of the International Workshop on Mining Software Repositories, Minneapolis, MN, USA, May 19-20, 2007, p. 27.

[16] S. Kim and M. D. Ernst, "Which Warnings Should I Fix First?," Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia, September 3-7, 2007, pp. 45-54.

[17] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation Exploitation in Error Ranking," Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Newport Beach, CA, USA, 2004, pp. 83-93.

[18] T. Kremenek and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," Proceedings of the 10th International Static Analysis Symposium, San Diego, California, 2003, pp. 295-315.

[19] T. Z. Menzies, C. J. Hihn, K. Lum, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 2-13.

[20] D. Roberts, J. Brant, R. Johnson, "A Refactoring Tool for SmallTalk," *Theory and Practice of Object Systems,* vol. 3, no. 4, 1997, p. 253-263.

[21] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting Accurate and Actionable Static Analysis Warnings: An Experimental Approach," Proceedings of the 30th International Conference on Software Engineering, Leipzig, Germany, May 10-18, 2008, pp. 341-350.

[22] H. Shen, J. Fang, J. Zhao, "EFindBugs: Effective Error Ranking for FindBugs," 2011 IEEE 4th International Conference on Software Testing, Verification and Validation, Berlin, Germany, March 21-25, 2011, p. 299-308.

[23] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," IEEE Transactions on Software Engineering, vol. 31, no. 6, 2005, pp. 466-480.

[24] I. H. Witten and E. Frank, Data Mining: Practical Machine Learning Tools and Techniques, 2nd ed. Amsterdam: Morgan Kaufmann, 2005.

[25] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting Defects in Eclipse," Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering, Minneapolis, MN, USA, May 20, 2007, p. 9.