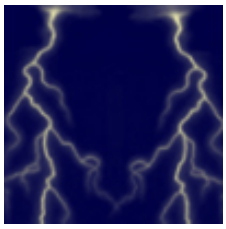
Crossroads
The ACM Student Magazine

Adaptively Ranking Alerts Generated from Automated Static Analysis

by ***Sarah Smith Heckman***



Abstract

Static analysis tools are useful for finding common programming mistakes that often lead to field failures. However, static analysis tools regularly generate a high number of false positive alerts, requiring manual inspection by the developer to determine if an alert is an indication of a fault. The adaptive ranking model presented in this paper utilizes feedback from developers about inspected alerts in order to rank the remaining alerts by the likelihood that an alert is an indication of a fault. Alerts are ranked based on the homogeneity of populations of generated alerts, historical developer feedback in the form of suppressing false positives and fixing true positive alerts, and historical, application-specific data about the alert ranking factors. The ordering of alerts generated by the adaptive ranking model is compared to a baseline of randomly-, optimally-, and static analysis tool-ordered alerts in a small role-based health care application. The adaptive ranking model provides developers with 81% of true positive alerts after investigating only 20% of the alerts whereas an average of 50 random orderings of the same alerts found only 22% of true positive alerts after investigating 20% of the generated alerts.

Introduction

Software applications typically contain developer mistakes that can often lead to failures in the field. Static analysis tools are useful in finding common programming mistakes that can lead to field failures [**2**, **5**, **10**]. An **alert** is a notification to the programmer, in the form of a warning message, of a potential fault in the source code that has been identified via static analysis [**10**]. The term **automated static analysis** (ASA) refers to the use of static analysis tools.

Some ASA can analyze source code from the beginning of the development process. Depending on the tool, ASA can also be performed on small portions of the code after a change has been made [**2**]. However, static analysis tools tend to report a high number of false positives due to the approximations needed for the tool to run in a timely manner [**2**, **5**, **10**]. Therefore, manual inspection of reported ASA alerts by the developer is required to determine if a fault actually exists in the source code [**2**, **5**].

The objective of this research is to create and validate an adaptive model to rank alerts generated by automated static analysis by the likelihood that an alert is an indication of an actual fault. By using historical data of developer feedback to help distinguish an alert from false and true positive faults, we hope to increase the number of faults detected before the number of investigated false positives discourages use of ASA altogether[**6**, **7**].

The adaptive ranking model (ARM) 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. A **ranking factor** is an alert characteristic used to predict whether another alert that shares the same characteristic is an indication of a fault. The current version of the model, ARM v0.4, uses two factors to rank alerts generated from ASA:

- Alert type accuracy (ATA): a value representing the homogeneity of alerts sharing the same alert type (e.g., null pointer and cast error).
- Code locality (CL): a value representing the homogeneity of alerts in the same location of code (e.g., method, class, and source folder) [6].

The **Automated Warning Application for Reliability Engineering** (AWARE) [11] provides ARM with ASA alerts, data on the developer's actions to suppress or close alerts, and historical information about the ranking factors. AWARE is currently being developed as a plug-in for the **Eclipse** integrated development environment, and focuses on analysis of the Java programming language. AWARE v0.4 gathers static analysis alerts generated by the FindBugs [5] Eclipse plug-in. AWARE then presents the ranked listing of alerts to the developer on the continuum [-1, 1]. A value of -1 represents an alert predicted to be a false positive, while a value of 1 represents an alert predicted to be a true positive. A value of 0 represents an alert where ARM cannot predict the likelihood an alert indicates a fault.

A case study investigated the feasibility of ARM in ranking alerts generated from ASA tools with the alert type accuracy and code locality ranking factors. iTrust, the subject of the case study, is a role-based health care program developed in a graduate level testing and reliability class at North Carolina State University. The purpose of the experiment was to compare the performance of ARM against random, optimal, and Eclipse orderings of the ASA alerts generated by FindBugs.

The rest of the paper is organized as follows: The next section covers the related work, followed by an overview of ARM. The following sections cover a case study investigating the efficacy of ARM and the results of the case study. The last section presents the conclusions and future work.

Related Work

Alert ranking strategies are used to prioritize alerts returned from ASA by the likelihood that the alert is a true positive [7]. Z-Ranking [7] is an algorithm that uses statistical analysis to rank alerts generated by the MC [3] static analysis tool for the C programming language. Another ranking algorithm, FEEDBACK-RANK [6] adjusts the alert ranking after each alert is inspected based on its location in the source code.

Kremenek et al. [6] have shown that true and false positive alerts tend to cluster by their location in the source code, or the alert's code locality. This clustering can occur at the function, class, and directory or source folder level. Kremenek et al. [6] show that for two systems (Linux and an unnamed commercial system), a function, file, and directory contain more than one alert 35%, 76%, and 97.5% of the time on average, respectively. Of those code locations with more than one alert, 88% of the methods, 52% of the files, and 13% of the directories have homogeneous alert populations. A homogeneous population means the alerts in a code locality are either all true positives or all false positives. Therefore, if a programmer provides feedback that an alert is a true positive, then other alerts in the same code location are likely to

also be true positives and vice versa for false positives [6].

Brun and Ernst [1] utilize machine learning to rank program properties by the likelihood that they will reveal faults. The authors gather semantic properties about the program's execution. However, the premise behind their ranking is extendable to other program properties such as static analysis. The authors describe the technique as "learning from fixes" by comparing faulty and fixed versions of a program. Similarly, ARM "learns from fixes," however, ARM also learns from what was *not* fixed.

Adaptive Ranking Model

This section describes ARM's inputs, calculation, contributions, and limitations.

ARM Inputs and Outputs

ARM takes three inputs: 1) the alerts generated by ASA, 2) developer feedback in the form of suppressing false positive alerts and closing true positive alerts, and 3) historical data from previous releases about the ranking factors. The premise behind ARM is to predict which alerts indicate a fault based on the actions that a developer took on past alerts. The developer's feedback modifies the ranking adaptively, while the historical data is represented as coefficients in ARM to provide initial data for the ranking, or as a control for team, organization, or company policies. For example, if a company is concerned about software field failures due to null pointers, modifications to the coefficients in ARM could increase or decrease the importance of the ranking.

ARM ranks each alert on a continuum from -1 to 1. A value in the range [-1,0) implies an alert is likely to be a false positive while a value in (0,1] implies an alert is likely to be a true positive. A value of 0 means that the model does not have enough data to predict whether an alert is a true or false positive.

Suppressing an alert 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 found to be a false positive or an indication of a fault that he or she does not elect to fix (e.g., a style 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.

ARM Specifics

The ranking factors in ARM v0.4 are alert type accuracy and code locality. If a ranking factor does not demonstrate strong statistical contribution to the ranking of ASA alerts, then the factor will be eliminated from future versions of ARM.

The total ranking of an alert *a* being a true or false positive in the software is the weighted average combination of alert type accuracy and code locality, as described in Equation 1. The ranking of the alert is presented to the developer as a value on a continuum between -1 and 1. The $ß_{ATA}$ and $ß_{CL}$ coefficients in Equation 1 weight the contribution of the alert type accuracy and code locality factors, respectively. Currently, the $ß_{ATA}$ and $ß_{CL}$ coefficients have an equal weight of 0.5, suggesting that the alert type accuracy

and code locality factors contribute equally to the ranking. However, the values of these coefficients will change as the importance of each ranking factor to a developer is determined through empirical analysis. Similar to Nagappan et al.'s approach [**8**], principal component analysis (PCA) is used to determine the contribution of each ranking factor to the overall ranking of ASA alerts.

$$\text{Ranking}(a) = (ß_{ATA} \bullet ATA(a)) + (ß_{CL} \bullet CL(a)), \text{ where } ß_{ATA} + ß_{CL} = 1 \quad (1)$$

The ranking of an ASA alert with an alert type accuracy value of 0.285 and code locality value of -0.141 would be as follows: Ranking(*a*) = (0.5 • 0.285 + 0.5 • -0.141) = 0.214. The positive ranking of 0.214 means that the alert is more likely to be an indication of a fault.

Adjustment Factor

The adjustment factor represents the homogeneity of a population *p* of alerts from developer feedback and is used to modify the computation of ranking factors alert type accuracy and code locality. A **population** of alerts it the subset of all reported alerts that share some characteristic, such as the same alert type or code location. The adjustment factor is scaled from -1 to 1. An adjustment factor value of -1 implies that all of the alerts with developer feedback in a population have been suppressed. An adjustment factor value of 1 implies that all of the alerts with developer feedback in a population have been closed. An adjustment factor value of 0 implies either that there are no closed or suppressed alerts in the population or that the number of closed and suppressed alerts is equal.

Equation 2 describes the adjustment factor calculation. The numerator describes the homogeneity of the population, while the denominator describes the developer's overall contribution of inspecting alerts in that population. Suppose there are ten alerts in a population, and seven of which have been inspected. Furthermore, of the seven inspected alerts, the developer closed three and suppressed four. The adjustment factor for the remaining three alerts in the population is $AF_p(a)$ = (3 closed - 4 suppressed) / (3 closed + 4 suppressed) = -1 / 7 = -0.14. The adjustment factor implies that the uninspected alerts in the population are more likely to be false positives.

$$AF_p(a) = (\#closed_p - \#suppressed_p) / (\#closed_p + \#suppressed_p) \quad (2)$$

Alert Type Accuracy

Alert type accuracy is the weighted average of historical data from the observed true positive rates of ASA and the actions a developer has taken to suppress and close alerts reported by ASA, as shown in Equation 3. In the absence of any historical data from the developer, an initial alert type accuracy value is obtained by investigating the true positive rates of alert types detected by ASA on similar or previous releases of source code or a literature review. The initial alert type accuracy value is in the range of [-1,1] where -1 means that ASA reports all false positives for an alert type and a value of 1 means that ASA reports all true positives for an alert type. A value of 0 implies that an alert type reported by ASA has an equal chance of being a true or false positive or that there is no data for that alert type.

$$ATA(a) = (x \cdot t_{type}) + (y \cdot AF_{type}(a)), \text{ where } x + y = 1 \quad (3)$$

For this research, the initial alert type accuracy value of 51 of FindBugs' 270 alert types were calculated from the data presented by Hovemeyer and Pugh [5]. Table 1 provides the alerts, descriptions, number of subalerts, and alert type accuracy values. For further explanation of the alerts, see [5]. Equation 3 describes the alert type accuracy calculation. The values of *x* and *y* are used to weight the contribution of the initial alert type accuracy value *t* and the adjustment factor for the alert type. In the absence of other empirical data, *x* and *y* are both given a weight of 0.5. The values of *x* and *y* will later be modifiable by the developer such that he or she can place more confidence in the initial alert type accuracy value or the adaptive ranking from the adjustment factor.

Table 1: Initial type accuracy values of selected FindBugs alerts.

General Alert Type Category [5]	Number Subalerts	Initial ATA Value
EC: Suspicious Equals Comparison	6	0.015
NP: Null Pointer Dereference	20	0.110
UR : Uninitialized Read in Constructor	1	0.009
DC: Double Checked Locking	1	0.181
IS2: Inconsistent Synchronization	1	0.175
NS: Non-Short-Circuit Boolean Operator	2	0.023
OS: Open Stream	2	-0.003
RCN: Redundant Comparison to Null	5	0.005
RR: Read Return Should Be Checked	2	0.021
RV: Return Value Should Be Checked	8	0.015
UW: Unconditional Wait	1	0.001
Wa: Wait Not in Loop	2	-0.012

Continuing with the previous example, assume there are 10 alerts sharing the same alert type. There are still three closed and four filtered alerts, therefore the AF_{type} = -0.14. Based on a historical study of previous releases of the system, we obtain t_{type} = 0.71. The default value of 0.5 is used for *x* and *y* , therefore *ATA(a)* = 0.5 • 0.71 + 0.5 • -0.14 = 0.285. Because of the strong historical trend towards alerts of the given type being true positives, the alert type accuracy value is in the true positive range.

Code Locality

The code locality contributes to the ranking of an alert from the historical data based on alerts that the developer suppressed and closed in the same area of code. When looking at code locality, a population is a particular area of code, such as a specific method *m*, class *c*, or source folder *s* [6]. The data found by Kremenek et al. [6] show that up to 88% of the location populations, especially at the function level, tend

to contain a homogeneous population of alerts.

The AF location describes the ratio of true to false positives in a particular location of code. ARM calculates the adjustment factor at the function, class, and source folder levels. Equation 4 is used to compute the code locality of a given alert using coefficients $ß_m$, $ß_c$, $ß_s$. The values for the coefficients $ß_m$, $ß_c$, $ß_s$ were obtained by normalizing the percentage of observed skew at the function, file, and directory levels in [6], and the values obtained are $ß_m = 0.575$, $ß_c = 0.34$, $ß_s = 0.085$.

$$CL(a) = (ß_m \bullet AF_m(a)) + (ß_c \bullet AF_c(a)) + (ß_s \bullet AF_s(a)), \text{ where } ß_m + ß_c + ß_s = 1 \quad (4)$$

Back to our example, suppose the 10 alerts all belong to the same method, class, and source folder in the source code under analysis. Based on the adjustment factor, AF location for all of the alert locations is -0.14. Therefore, $CL(a)$ = 0.575 • -0.14 + 0.34 • -0.14 + 0.085 • -0.14 = -0.141. The negative value for the code locality implies that the alert is more likely to be a false positive.

Expected Contribution

The expected contribution of this research is to reduce the cost of false positives via ARM for ranking alerts generated by ASA tools. ARM's ranking should provide more true positive alerts at the top of the listing, making the ASA tools more useful for developers. While this research focuses on the Java programming language, the generation of a probabilistic model for other languages should follow.

ARM Limitations

The initial ranking of ASA alerts across the entire code base is important to a developer's initial impression of the usefulness of static analysis and, therefore, to his or her continued use of ASA [6]. Continued use is important because the developer's actions cause the model to adapt and increase the predictability.

ARM's initial ranking is dependent on historical data about the ASA with respect to the project. In ARM v0.4 the initial ranking is dependent on knowledge about the ratio of true positives to reported alerts for a particular alert type, which was gathered from [5]. When data was not available about an alert type, an initial alert type accuracy value of 0 was used. The initial alert type accuracy values used in ARM v0.4 may not be predictive of all ATA distributions. The initial alert type accuracy could come from an investigation of previous versions of the code. An alert is a true positive when an alert identified in an earlier version of code is no longer reported by ASA on a later version of the code.

Case Study

To determine the efficacy of the ranking strategy of ARM, the alert ranking generated for iTrust by ARM was compared against the random, optimal, and Eclipse orderings of the same alerts.

Overview of iTrust

iTrust is a role-based health care application developed as part of a project in CSC 712: Software Testing and Reliability at North Carolina State University. The case study was done on a project from a pair of students in the Fall 2006 Software Testing and Reliability class. The teaching assistant from the class picked an exceptional project turned in for the assignment. The student team had never run static analysis on the

iTrust project used for this case study. The iTrust project contained 19 source classes and 25 test classes with 1964 lines of source code and 3903 lines of test code.

The iTrust application contains two types of automated test: 1) unit test cases written using the **JUnit** framework and 2) black box test cases written using the **FIT** framework. JUnit is a regression testing framework for automating the running of unit tests. FIT is a framework for integrated testing, which provides a vehicle for collaboration between developers, testers, and customers. FIT tests consist of tests in HTML tables and Java classes, called fixtures, that tie the HTML based test to the application. Alerts generated by FindBugs also included faults in the test cases. The analysis of test code is also important because a mistake in a test case may affect a developer's confidence in an application.

The experiment was run in Eclipse 3.2 as a runtime workbench on a 1.66 GHz T1300 processor with 1.00 GB of RAM. The combination of a run of ASA with FindBugs on the full source code base and ranking the alerts with the AWARE tool took less than 90 seconds. When AWARE was run on the code, 163 alerts were generated covering 18 FindBugs alert types. Of the 163 alerts, 27 alerts were determined to be faults through inspection of the code by the author. A fault was defined as an execution path through the code that could lead to a program failure.

Experimental Setup

This experiment investigated the following hypotheses:

- Hypothesis 1: ATA is a contributing predictor of future true positive alerts.
- Hypothesis 2: CL is a contributing predictor of future true positive alerts.
- Hypothesis 3: The adaptive ranking of alerts based upon ARM will present the developer with more true positive alerts at the top of the ranking than a random or Eclipse ordering .

Hypothesis 1 and Hypothesis 2 were investigated through observation of the ranking of the alerts by population. Hypothesis 3 compared the ARM ranking of ASA alerts to a random, optimal, and Eclipse ordering of the same alerts. An inspection of the alert at the top of the list determined if the alert was a true or a false positive. For the ARM, an adjustment to the ranking of the remaining alerts occurred. The number of true positive alerts identified after each inspection was reported.

The number of inspections required to close all of the true positive alerts for ARM was compared to a random, optimal, and Eclipse ordering of alerts. A random ordering of alerts is a viable ranking strategy when no other ranking data are available [**6**], therefore random ranking was the baseline of the experiment. In an optimal ordering of alerts, each inspection uncovers a fault until all ASA detectable faults are uncovered. Eclipse's "Problems" view orders ASA alerts (and all other Eclipse problems) by the alert description.

Results

This section reports the results of the research hypotheses and the limitations of the case study.

Hypothesis 1

FindBugs reported 18 distinct alert types on the iTrust code. Of the 18 alert types, only three consisted of

mixed true and false positive alerts. All other alert types were either all true positives or all false positives. Because a majority of the alert types populations were homogeneous, the alert type accuracy value is a good predictor of future true or false positive alerts.

Hypothesis 2

Of the 36 classes with ASA alerts, only 11 of the classes have nonhomogeneous populations. Of the 25 homogeneous nonsingleton populations, one class is predictive of true positives and 19 classes are predictive of false positive alerts. Alerts at the method level tended to be singletons or were predictors towards the false positive end of the ranking. The contribution of the source folder is a good predictor for false positive alerts. After one inspection of a false positive alert in a given source folder, all other alerts in that folder are moved to the bottom of the ranking. The predictors for the mixed folders tend towards the false positive end of the spectrum.

Hypothesis 3

The ranking of ARM was compared to three baselines: a random ordering of alerts, an optimal ordering of alerts, and Eclipse's ordering of alerts. The random baseline was the average of 50 unique random permutations (generated via **MatLab**) of the alerts. Multiple random permutations were used to ensure that our random comparison point exhibited no special ordering characteristic. The optimal ordering of alerts positions all true positive alerts at the top of the ranking, therefore ensuring that each inspection will find a fault until all true positive alerts are exhausted. The default of Eclipse's Problems View is to present alerts alphabetized by the alert description.

The true positive alert detection rate for ARM, the random baseline, optimal ordering, and the Eclipse ordering of alerts are presented in Figure 1. The x-axis represents the number of inspected alerts. Each alert inspection can uncover either a true or a false positive. The y-axis represents the number of true positive alerts found at, or before, a given inspection. The data line is parallel to the x-axis when false positives are uncovered. The data was generated by determining if a true or a false positive alert was uncovered for the different orderings of the alerts. The random, optimal, and Eclipse ordering were static, but ARM's ranking adjusted after each true positive correction or false positive suppression.

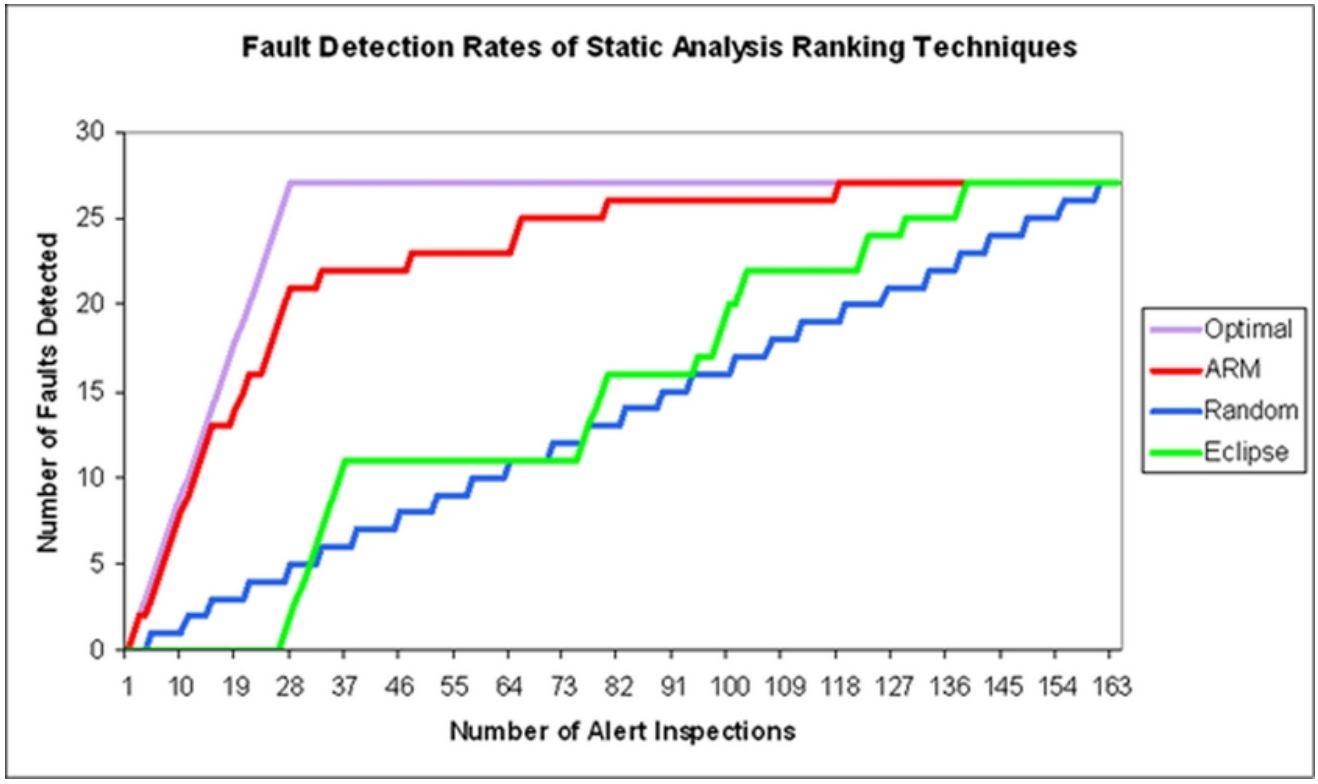


Figure 1: Fault detection rates for alert inspections.

Figure 1 shows that ARM performs better than random and the Eclipse ordering of alerts by presenting more true positive alerts to the developer early in the alert listing. ARM performs very close to the optimal ordering of alerts by discovering 81% of the true positive alerts in the first 20% of inspections. The random baseline ordering of alerts uncovered 22% of the true positive alerts in the first 20% of inspections. ARM's performance then degrades because the remaining alerts belong to nonhomogeneous populations. However, ARM found all the true positive alerts after 73% of the possible alert inspections. The random baseline found all 27 true positives after inspecting 98% of the alerts and the Eclipse ordering found all the true positives after inspecting 85% of the alerts.

Kremenek et al. [6] conducted a similar experiment using C static analyzers on an unnamed commercial system. Using their code locality adaptive ranking system, they achieved a 2-3 times improvement over a random baseline. ARM includes one other ranking factor, ATA, and currently gathers alerts on Java applications. The ARM showed a 1.34 times improvement over the random baseline. However, our overall alert population was 2.7 times larger than the alerts investigated by Kremenek.

Case Study Limitations

The case study investigated in this paper provides an overview of how ARM works. However, there were several limitations. The initial alert type accuracy values were gathered from data about projects investigated as part of the work in [5]. The reported data only covers a subset of the alert types, and more data could have improved ARM. Conducting a historical analysis of the delta between previous builds of the iTrust project could have also been used as initial alert type accuracy values. The coefficient values for the code locality ranking factor were obtained from analysis of the skew in [6]. However, the analysis in Kremenek et al.'s work was on C code, and the clustering of alerts in homogeneous populations at the function, class, and source folder level may not parallel the Java equivalents. Finally, the experiment

presented in this paper was conducted on only one project. Therefore, while the results are encouraging, they are not statistically significant.

Conclusions and Future Work

From the case study conducted on the iTrust role-based health care application, our data indicates that both the alert type accuracy and code locality factors contributed to the ranking of alerts generated from ASA. The data also indicates that ARM performs better than a random baseline and the ordering of alerts generated by FindBugs and reported in the Eclipse Problems View while approaching the optimal ordering of alerts.

Future work will examine how to improve the ranking of ARM by investigating the contributions of the alert type accuracy and code locality ranking factors to ARM through principal component analysis. Particular emphasis will be on improving the ranking of ARM for nonhomogeneous populations. To accomplish the improved ranking, other ranking factors will be added and the model will be validated in future open source and industrial case studies. The Generated Test Failure [**4**, **11**] ranking factor requires that test cases are automatically generated for ASA alerts.

As multiple static analysis tools are added to ARM, a ranking factor that investigates the overlap between alert types from multiple tools will be introduced and empirically investigated. Other work will be conducted to determine whether adaptive sorting heuristics or the inclusion of information gain algorithms [**6**] would improve the detection of true positive alerts. Future evaluation of ARM will parallel the test prioritization work done by Rothermel et al. [**9**] via the use of the rate of fault detection metric and the area under the curve.

Acknowledgments

This research was funded by a grant from the **Center for Advanced Computing and Communication** and by an **IBM PhD Fellowship** awarded to the author. I would like to thank the **RealSearch** reading group for providing feedback on this paper. I would also like to thank Lucas Layman and Stephen Thomas for their work in the development of AWARE and Lucas Layman, Chei-wei Ho, and Meiyappan Nagappan for their suggestions to the ARM.

References

1
Y. Brun and M. D. Ernst, "Finding Latent Code Errors via Machine Learning Over Program Executions," in *26th International Conference on Software Engineering* , Edinburgh, Scotland, 2004, pp. 480-490.

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

3
D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," in *Operating Systems Design and Implementation* no. San Diego, CA, 2000.

4
S. Heckman and L. Williams, "Automated Adaptive Ranking and Filtering of Static Analysis Alerts," in *17th International Symposium on Software Reliability Engineering, Fast Abstract* , Raleigh, NC, USA,

2006.

5 D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* , Vancouver, British Columbia, Canada, 2004, pp. 132-136.

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

7 T. Kremenek and D. Engler, "Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations," in *10th International Static Analysis Symposium* , San Diego, California, 2002.

8 N. Nagappan, T. Ball, and A. Zeller, "Mining Metrics to Predict Component Failures," in *28th International Conference on Software Engineering* , Shanghai, China, 2006, pp. 452-461.

9 G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing Test Cases For Regression Testing," *IEEE Transactions on Software Engineering,* vol. 27, no. 10, pp. 929-948, October 2001.

10 N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," in *15th IEEE International Symposium on Software Reliability Engineering* , Saint-Malo, Bretagne, France, 2004, pp. 245-256.

11 S. E. Smith, L. Williams, and J. Xu, "Expediting Programmer AWAREness of Anomalous Code," in *16th IEEE International Symposium on Software Reliability Engineering, Fast Abstract* , Chicago, IL, USA, 2005.

Biography

Sarah Smith Heckman is a PhD student in the **Department of Computer Science** at **North Carolina State University** and an intern at **IBM**. Her research interests are in software engineering, static analysis, and testing. Sarah received her MCS and BS in computer science from North Carolina State University. She is a member of the ACM and IEEE. She received an IBM PhD fellowship in 2006 and 2007. Contact her at sarah_heckman@ncsu.edu.