Expediting Programmer AWAREness of Anomalous Code

Sarah E. Smith, Laurie Williams, Jun Xu
North Carolina State University
{sarah_smith, lawilli3, jxu}@ncsu.edu

Abstract

Long fix latency, the amount of time between fault injection and fault removal, could substantially increase the cost of a fault fix. To mitigate this cost, software engineers could be made aware of potentially-anomalous code almost as soon as it is written. Test-driven development is a style of programming in which code and tests are written in tight cycles, therefore providing feedback to software engineers early and often. Enhancing test-driven development feedback loops to automatically and continuously provide ranked, prioritized, and filtered alerts to the software engineer on the correctness and security of their code implementation should reduce the cost of fixing software faults. The Automated Warning Application for Reliability Engineering (AWARE) tool is being developed to support enhanced test-driven development.

1. Introduction

A software fault[1] may be detected only after it is revealed in testing or in code analysis, often after a large amount of code has been written. Long *fix latency*, the time between fault injection and fault removal, could substantially increase the cost of the fault fix. By expediting programmer awareness of faults, erroneous code could be more easily corrected because the programmer will be more cognizant of recent changes that could have caused the fault.

Test-driven development (TDD) [1] is a style of programming in which code and tests are written in tight cycles. By design, these tight cycles help to reduce the time between fault injection and removal since feedback is provided to the software engineer early and often. In the unit test cycle of TDD, the programmer works in rapid cycles of writing code, unit tests, and running the tests.

Enhancing TDD to include both static and dynamic analysis will increase the scope of faults detected. Static analysis reveals more Assignment and Checking defects while testing reveals more Function and Algorithm defects [7] in the Orthogonal Defect Classification [3] scheme. However, static analysis does have the disadvantage of a high number of false positives [7].

The objective of the research is to enhance TDD feedback loops to automatically and continuously provide ranked, prioritized, and filtered alerts to the software engineer on the correctness and security of their code implementation during development. The research builds upon previous research in continuous testing [6, 5] and automatic test case generation to reduce static analysis false positives [2]. The **A**utomated **W**arning **A**pplication for **R**eliability **E**ngineering (AWARE) tool is being built for the Eclipse[2] platform to support enhanced TDD.

2. AWARE

AWARE automatically and continuously provides the software engineer a fault listing based upon compilation, unit testing, and static analysis. Several development environments, such as Eclipse, maintain compilation of source code. AWARE utilizes continuous compilation and builds upon two other tools: Continuous Testing [6, 5] and Check 'n' Crash [2] (CnC). Continuous Testing continuously and automatically runs automated unit tests using spare processor cycles while the programmer continues to work [6, 5]. CnC conducts static analysis using ESC/Java2[3]; false positives are reduced via automatically generated test cases [2]. In AWARE, the alerts produced by each technique and the response of the programmer to these alerts form a feedback loop to reduce the reported false positives as the framework is being used.

The following terms define specific points of time in the fault lifecycle of interest to the research:

- **Fault injection:** the point at which a fault is introduced in the program source code.
- **First alert:** the point at which a programmer is notified of a potential fault in the code.
- **Action alert:** the point at which the programmer clicks on an alert to obtain more information.
- **Close point:** the point at which the alert goes away because the fault has been fixed or because it was intentionally rejected by the programmer.

Each of these points in time is detectable by the presence or absence of a failing test case or static analysis alert. Therefore, the fault injection and close point times may be inaccurate by at most one alert run cycle. Based on

[1] The term "fault" is used to describe anomalies that may never surface as failures in operational use of the product.

[2] http://eclipse.org

[3] http://secure.ucd.ie/products/opensource/ESCJava2/

the above definitions, the following metrics are defined for experimental evaluation. The relationship between these metrics is demonstrated pictorially in Figure 1.

- **Fault fix latency:** the time between the fault injection and fault close point.
- **Ignorance time:** the time between fault injection and action alert when the programmer becomes consciously aware of details of the reported fault.
- **Fix time:** the amount of time a programmer spends to fix the reported fault.
- **Action latency:** the time between the first fault alert and the first programmer action to address the alert.

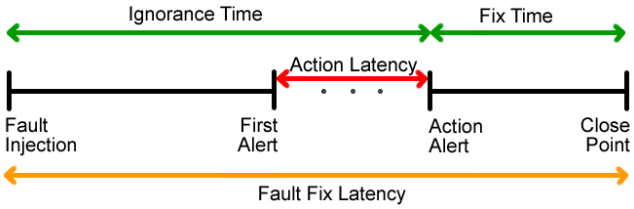

Figure 1: Evaluation Metric Relationships

Research [6, 5] indicates that fix time is correlated with ignorance time; by reducing ignorance time, fix time will be reduced. Ignorance time should be reduced by providing programmers with prioritized alerts from test failures and static analysis results. Therefore the reliability and security of code will be improved through more attention to code correctness.

3. Alert Ranking and Prioritization

The programmer interacts with AWARE by reacting to the alerts, potentially repairing the faults indicated, and giving feedback on the accuracy and relevance of the alert information. AWARE adapts its internal ranking and filtering of alerts based on this feedback and on the results of automatically-generated test cases, to improve its future performance.

AWARE will utilize a single, unified metric to rank the alerts from different sources. The ultimate criterion for ranking is the expected accuracy of an alert. The factors listed below will be initially used for alert ranking.

- **Type accuracy (TA):** The probability that a particular type of alert is a true positive. This will be done by categorizing different fault types and the observed accuracy of a type.
- **Redundancy factor (RF):** The probability that an alert is a true positive given that multiple tools have reported the same alert. This can be facilitated by standardizing the classifications and outputs of multiple tools.
- **Code locality (CL):** The probability that an alert in a given code location is a true positive. Kremenek et al. [4] show that alerts reported by static analysis tools frequently cluster by code locality. It is more likely a new alert is a true positive if other nearby alerts are true positives.
- **Test coverage (TC):** The probability that an alert is a true positive despite the amount of testing. We expect an inverse relationship between the probability of true positives versus the amount of test coverage.

Using the list of factors, we can compute the ranking of a set of reported alerts by finding the conditional probability that a particular alert α is a true positive (TP) given the factors listed above. Formally, the probability is expressed in the following formula given the alert α: $Pr(\alpha \text{ is } TP \mid TA(\alpha), RF(\alpha), CL(\alpha), TC(\alpha))$. The value of the formula can be computed using elementary probability theory. As a programmer inspects the alerts, he or she will determine whether an alert is a true or false positive. Once a response is known, we can use the feedback to recompute the prior probability values for TA, RF, CL, and TC.

4. Conclusions and Future Work

By extending TDD to include static analysis cycles, more code vulnerabilities should be discovered earlier during development, thereby reducing the ignorance time. The reduction of ignorance time should lead to the reduction of fix time. Future work will investigate the relationship between the metrics discussed in the paper to determine if the enhancement of TDD reduces the fault fix time.

Acknowledgements

This work is supported by the NCSU Center for Advanced Computing and Communication.

References

[1] K. Beck, *Test Driven Development -- by Example*. Boston: Addison Wesley, 2003.

[2] C. Casllner and Y. Smaragdakis, "Check 'n' Crash: Combining Static Checking and Testing," in *International Conference in Software Engineering*. St. Louis, MO, USA, 2005

[3] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal Defect Classification - A Concept for In-Process Measurements," *IEEE Transactions on Software Engineering*, vol. 18, pp. 943-956, 1992.

[4] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation Exploitation in Error Ranking," presented at International Symposium on Foundations of Software Engineering (FSE), Newport Beach, CA, 2004.

[5] D. Saff and M. D. Ernst, "Reducing Wasted Development Time Via Continuous Testing," presented at International Symposium on Software Reliability Engineering, Denver, CO, 2003.

[6] D. Saff and M. D. Ernst, "An Experimental Evaluation of Continuous Testing During Development," presented at International Symposium on Software Testing and Analysis, Boston, MA, USA, 2004.

[7] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. Hudepohl, and M. Vouk, "A Study of Static Analysis for Fault Detection in Software," North Carolina State University, Raleigh, NC TR-2005-26, 2005.