Macro and Micro Perspectives on Strategic Software Quality Assurance in Resource Constrained Environments

Tao Xie David Notkin
Department of Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA
+1 206 616 1844
{taoxie, notkin}@cs.washington.edu

ABSTRACT

Software quality assurance (SQA) plays a key role in software development process. Software quality assurance methods include testing, inspection, formal method (program verification, model checking, etc.), static code analysis, and runtime verification, etc. A disciplined approach to meeting benefit, cost, schedule, and quality constraints is in need. In this paper, we propose two perspectives (macro and micro) on strategic software quality assurance in resource constrained environments. We present a survey and discuss a variety of research opportunities and challenges with these two perspectives. Finally we present our research work on test case prioritization based on boundary value coverage to tackle strategic SQA problems with these two perspectives.

Keywords: Software Quality Assurance, Economic Driven Software Engineering, Regression testing, Test Case Prioritization

1. INTRODUCTION

The activities of software quality assurance (SQA) are different from other activities in a software development process in that the investment on different SQA methods is to achieve the same goal, making software products high quality, or finding and fixing bugs in software products more specifically. However, in other activities during software development process, usually only one method or approach is adopted to attain the goal, such as those activities in phases of design and implementation, etc. One of the reasons why people usually do not choose only one SQA method to achieve high software quality is that so far few or even no single SQA method can assure satisfactory quality alone. In early days, software testing and inspection were commonly used as SQA methods. Owe to recent advances on theory and programming languages, static defect detection and model checking techniques have been used in SQA activities more extensively. Osterweil et al. [23] suggest that different SQA techniques and tools could be integrated to provide value considerably beyond what the separate techniques can provide. Gunter et al. [13] suggest developing methods for combining the strengths of different methods for analyzing software system to improve its quality. In this paper, we propose two perspectives in strategic software quality assurance. The macro perspective focuses on the integration of different SQA methods. One of the issues is resource distribution among different SQA methods and is related to following question:

- What SQA methods are to be adopted? How to distribute the constrained resources on these methods?

The micro perspective focuses on the strategic resource distribution within certain SQA method and is related to following question:

- How to distribute the constrained resources on the artifacts involved on certain SQA method?

2. MACRO PERSPECTIVE

Provided with various SQA methods to choose, the SQA people have to decide on which ones shall be used in their SQA process and how to distribute the constrained resources among them. An exemplary resource distribution on different SQA methods is showed in Figure 1.

The cost-effective analysis of each SQA method needs to be conducted to support the decisions on them. One direction to improve cost-effectiveness of a SQA method is to reduce the cost while keeping reasonable effectiveness. For example, generally formal methods require upfront expensive investments. Wong et al. suggested some ways of tailoring formal methods to suit a constrained environment [29].

EDSER-4 '02, May 21, 2002, Orlando, Florida.

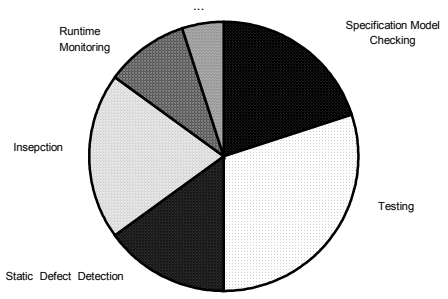

Figure 1. Exemplary Resource Distribution on Different SQA Methods

The other direction to improve cost-effectiveness of a SQA method is to increase its effectiveness while keeping the reasonable cost. Effectiveness of different SQA methods might be dependent on specific applications or application domains, including the types of contained faults. Although test case generation in testing method can be fault-based, testing method is generally not fault-specific. Theoretically exhaustive testing can detect all faults in the program. However, model checking or static defect detection is generally property-specific or specific to those faults that violate particular properties. Therefore, investigating the effectiveness of these SQA methods needs to take the types of potential faults in the system into consideration.

Above two directions are to improve cost-effectiveness for a particular SQA method. If some SQA methods have relatively good cost-effectiveness for our target system, we might invest the constrained resources on them rather than the less cost-effective ones. Indeed, in practice, distributing constrained resources among SQA methods is more complicated than that being affected by other factors. For example, besides the cost-effectiveness, the severity cost of faults that are detectable by certain SQA methods can greatly affect the decision whether or not to use that method. Chandra et al. claim that for application domains where the cost of just a handful of high-severity faults is high, model checking is a valuable complement to traditional SQA methods [6].

Another way is to improve the interaction of different SQA methods to gain more value than the separate techniques can provide, which is focused by this section. Traditionally, different SQA methods have been viewed as competing ways of achieving high quality software, perhaps complementing each other, but essentially separate. Integration of different SQA methods in an effective way has recently caught the attention of practitioners and researchers. The effective integration of different SQA methods can provide better cost-effectiveness than performing them independently. Gunter et al. [13] exemplified a scenario of SQA method integration that the result of type checking or other static analysis techniques could be used to assist theorem proving, and when a theorem prover failed to establish a proof obligation, the information about how the prover failed could be used to generate test cases.

The integration of different SQA methods can be categorized as loose integration and tight integration based on the extent of coupling between these SQA methods.

2.1 Loose Integration

The final artifact or product of a SQA method is the one that is produced by that method to directly aid the fault detection. For example, the final artifacts of testing methods are a list of executed test cases that detect the faults together with their execution results. By incorporating the fault localization technique, the final artifacts of testing methods can include the locations of the detected faults in the code. The final artifacts of specification model checking method are a list of violated properties and their counter-examples in specification.

The intermediate artifact or by product of a SQA method is the one that is used by that method to produce the final artifacts. For example, the intermediate artifacts of the specification model checking method include the symbolic model that is feed to the model checker. The intermediate artifacts of testing method include the test case execution trace.

In loose integration of different SQA methods, one SQA method makes use of the intermediate artifacts produced by another SQA method.

2.1.1 Dynamic Analysis and Static Defect Detection

Daikon [10], an invariant detection tool, can discover likely specification, e.g. invariants, from program executions by instrumenting the target program to trace the variables, running the instrumented program over a test suite, and inferring invariants over the instrumented values. Dynamically detected invariants can annotate a program or provide goals for static verification. These invariants, the intermediate artifacts of testing, are fed to the static checker ESC-Java [22]. If the static checker finds the conditions under which the invariants collected from correct runs are invalidated, the potential faults are reported.

2.1.2 Model Checking and Testing

Gunter et al. [13] claim that success for specifications will come from providing tangible benefits, such as test oracles and test generation, not just the potential for verification. Reusing specification among different SQA methods is a loose integration of them commonly used by practitioners and researchers.

By modeling the negation of test purpose or coverage criteria as a temporal formula, the test case generation problem is formulated as finding counterexamples during

the model checking [9][14]. Another similar approach is to model the negation of certain properties to be covered and generate the counterexamples as test cases [12]. Application of mutation analysis in model checking can generate counterexamples as test cases to distinguish the variants from original specification [1].

TestEra [20] models the correctness criteria for the program in Alloy, a first-order relational language, and specifies abstraction and concretization translations between instances of Alloy models and Java data structures. It produces concrete Java inputs as test case counterexamples to violated correctness criteria by using the Alloy Constraint Analyzer [16].

VeriSoft [6] implements model checking algorithms for systematically testing concurrent reactive software and has been used in a large industrial telecommunication system. Its applications show that complementing model checking with traditional testing can significantly contribute to increasing the confidence that a software system is ready to ship.

2.1.3 Theorem Proving and Testing

Given a formal proof of the correctness of an abstract model of a system to be developed, testing data can be extracted to test concrete implementations of that system. Carrying out a correctness proof often entails detailed analysis of the input domain of a program, partitioning that domain into subdomains for which a uniform argument is used to establish correctness. This partitioning by-product of the proof process can be used as the basis for extracting test cases to be used for validating the implementations of the specification [19][5]. On the other hand, automated testing is used to gain confidence in the likely correctness of the software, prior to investing in proofs [26].

2.2 Tight Integration

In tight integration of different SQA methods, one SQA method makes use of the final artifacts or products produced by another SQA method. Fewer progresses have been made in tight integration of different SQA methods than in loose integration. There exist some challenging research opportunities in this area.

The final artifacts of a SQA method, generally in form of a list of reported faults, can be categorized as follows:

- Actual faults, those reported faults that are actually faults in the system.
- False positives, those reported faults that are actually not faults in the system.

In addition, false negatives are those faults that are left undetected in the system by this SQA method.

Generally testing methods might have relatively fewer false positives but other SQA methods might report certain number of false positives caused by the information loss due to abstraction or other unsound static analysis techniques. Sometimes it is beyond the capability of human inspection to identify those warnings to be false positives or actual faults. Those unresolved warnings are called uncertain faults. One potential research direction to tackle this problem is to generate and run those test cases to exercise the dangerous conditions, e.g. certain paths or certain variable values, under which these uncertain faults are expected to expose. However, there are two challenges: the first is to associate the faults with the dangerous conditions; the second one is to generate or identify the test cases based on these dangerous conditions. Another direction is to monitor those conditions that the uncertain faults are related to by using runtime verification techniques.

When resources are constrained, it is reasonably practical to select or prioritize the test cases in test case pool that possibly exercise those dangerous conditions related to uncertain faults. Another coarse-grained approach with less complexity is to select or prioritize those test cases that exercise those structural entities, e.g. statements or functions, where those uncertain faults are reported.

By estimating the false negatives left in system by certain SQA method based on the reported faults and characteristics of the system, SQA people can schedule another SQA method to complement to detect these false negatives.

3. MICRO PERSPECTIVE

Different from macro perspective, which focuses on the interactions between SQA methods, micro perspective focuses on the resource distribution within certain SQA method. More specifically, it concerns the distribution of constrained resources on the artifacts on certain SQA method. These artifacts can be modules in modular static defect detection techniques, subsystems of a specification in model checking techniques, components in runtime verification techniques, and test cases in testing techniques.

3.1 Selection Techniques

When resources are constrained and cannot afford to invest on all artifacts in the system, selection techniques are used to select those artifacts to invest based on certain criteria. The criteria might take importance, mission-criticality, cost, and fault potential into account when the artifacts are functional units in the system. Similar selection techniques are used in software product line scoping process to define the scope of product line [8] or in requirement scoping process to define the scope of requirement to fulfill [17].

For artifacts of test cases or test suites in testing techniques, the criteria might comprise rate of fault detection, test case execution cost, structural coverage, and data coverage, etc.

3.2 Prioritization Techniques

Since the artifact selection is done under conditions of uncertainty and incomplete knowledge, the scope of selection might be dynamically adjusted after selection decision is made. For example, when the cost of the SQA method is underestimated, there are not enough resources to spend on all selected artifacts. When the cost of the SQA method is overestimated, there are still some available resources left for SQA people to decide which artifacts of unselected ones are to be spent.

Even when the scope of selection is not changed along the way, it remains advantageous to detect the faults as early as possible. Therefore, prioritization techniques are used to complement selection techniques in addressing these issues induced by uncertainty.

In software development process, Microsoft [7] breaks down large products into a priority-ordered set of manageable product features. This provides a mechanism for incorporating customer inputs, setting priorities, completing the most important parts first, and changing or cutting less important features. Various approaches to prioritize requirements are proposed by researchers [4][18].

Test case prioritization techniques sort the test cases for regression testing in certain order such that those test cases that are more important measured in some way are scheduled to run earlier in regression testing [28][24]. Integration of test case selection and prioritization are used in practice by prioritizing the test cases selected in test case selection process [28][25]. Most existing test case prioritization techniques are based on structural coverage, e.g. function coverage or statement coverage, etc.

4. TEST CASE PRIORITIZATION BASED ON BOUNDARY VALUE COVERAGE

Since a large number of faults tend to occur at boundaries of the input domain, testing boundary values has been extensively discussed in software testing handbooks [21][2][3]. Most past research work in boundary value coverage has generally focused on test case generation instead of test case assessment, which can be used in regression testing, e.g. test case selection or test case prioritization, due in part to lack of tool support to collect boundary value coverage information without a priori specification.

Since very few programs in practice are equipped with specification or assertion-like annotations containing the predicates to infer boundary values, we propose a new approach of applying boundary value coverage in test case prioritization without requiring a priori specification. We use an invariant detection tool called Daikon to dynamically infer specification, e.g. invariants, preconditions and post-conditions for each function [10].

Daikon discovers likely specification, e.g. invariants, from program executions by instrumenting the target program to trace the variables, running the instrumented program over a test suite, and inferring invariants over the instrumented values. Daikon infers invariants at specific program points such as loop heads, entries and exits of functions. The invariant detector is provided with variable traces that contain the values of all variables in scope at specific program points for each test case execution. We develop a boundary predicate extractor to extract the boundary predicates for specific program points from invariants produced by Daikon. Boundary value coverage calculator is developed to calculate the boundary value coverage for certain test case based on the boundary predicates and the variable trace for that test case. An overview of above procedures is showed in Figure 2.

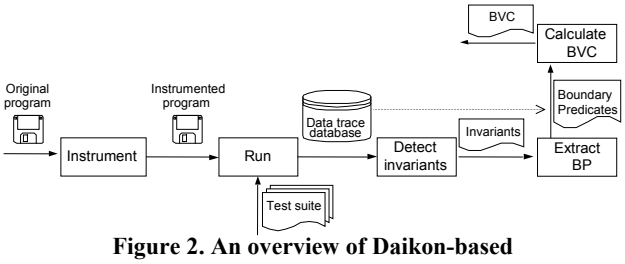

Figure 2. An overview of Daikon-based boundary value coverage technique.

Based on the boundary value coverage information collected for each test case execution, we calculate the reward value for each test case and order them based on these reward values. An experiment is conducted on seven small-size siemens programs created by Siemens researchers [15] and one medium-size space program originally created by Vokolos and Frankl [27]. Our initial results show that incorporating boundary value coverage information in prioritization can improve the effectiveness of structural coverage techniques in average. This work is tackling strategic SQA prioritization problem with micro perspective. At the same time, this approach is a form of loose integration of Daikon, a dynamic analysis approach, and testing approach with macro perspective.

5. CONCLUSIONS

The macro perspective on strategic SQA focuses on the integration of different SQA methods and the resource distribution among them. The micro perspective focuses on the strategic resource distribution within certain SQA method. This paper presents a survey and proposes the research directions and challenges with these two perspectives. Our initial work on test case prioritization based on boundary value coverage is an attempt to tackle strategic SQA problems with these two perspectives.

6. ACKNOWLEDGMENTS

Discussions with students in the Software Engineering Seminar, CSE 590n, at the University of Washington, Winter 2002, were helpful and inspiring. This work was supported in part by the National Science Foundation under grant ITR 0086003.

7. REFERENCES

[1] P. Ammann , P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In Proceedings of ICFEM, pages 46-54, December 1998.

[2] B. Beizer, Software Testing Techniques, 2nd edition, Van Nostrand Reinhold, New York, USA.

[3] B. Beizer, Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley, New York, 1995

[4] B. Boehm, H. In. Identifying quality-requirement conflicts, IEEE Software, Volume: 13 Issue: 2, March 1996, Page(s): 25 –35

[5] S. Burton, J. Clark, A. Galloway and J. McDermid. Automated V&V for High Integrity Systems: A Targeted Formal Methods Approach. In the Proceedings of the 5th NASA Langley Formal Methods Workshop. June 2000.

[6] S. Chandra, P. Godefroid and C. Palm Software Model Checking in Practice: An Industrial Case Study. Proceedings of ICSE'2002, Orlando, May 2002.

[7] M. A. Cusumano , R. W. Selby, How Microsoft builds software, Communications of the ACM, v.40 n.6, p.53-61, June 1997

[8] J-M. DeBaud and K. Schmid. A Systematic Approach to Derive the Scope of Software Product Lines. Proceedings of ICSE'1999, Los Angeles, CA, USA, pp. 34–43, 1999.

[9] A. Engels, L. Feijs, S. Mauw: Test Generation for Intelligent Networks using Model Checking. In 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Enschede, the Netherlands, April 1997,

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering, 27(2): 1-25, Feb. 2001.

[11] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In International Symposium of Formal Methods Europe 2001, Lecture Notes in Computer Science, v2021, pages 500-517. Springer, March 2001.

[12] A. Gargantini, and C. Heitmeyer, Using Model Checking to Generate Tests from Requirements Specifications, ESEC/FSE '99, Toulose, France, September 1999, Lecture Notes in Computer Science, Vol. 1687.

[13] C. Gunter , J. Mitchell , D. Notkin, Strategic directions in software engineering and programming languages, ACM Computing Surveys (CSUR), v.28 n.4, p.727-737, Dec. 1996

[14] H.S. Hong, I. Lee, O. Sokolsky and S.D. Cha. Automatic Test Generation from Statecharts Using Model Checking, Proceedings of Workshop on Formal Approaches to Testing of Software, August 2001, pp. 15-30.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In Proceedings of the 16th ICSE, pages 191-200, May 1994

[16] D. Jackson, I. Schechter, I. Shlyahter: Alcoa: the alloy constraint analyzer. In proceedings of ICSE 2000: p.730-733

[17] J. Karlsson and K. Ryan. Supporting the Selection of Software Requirements. In 8th International Workshop on Software Specification and Design, 1996, pp. 146-149.

[18] J. Karlsson and K. Ryan. Prioritizing Requirements Using a Cost-Value Approach. IEEE Software. September/October issue, 1997, pp. 67-74.

[19] S. Maharaj. Towards a Method of Test Case Extraction from Correctness Proofs. Proceedings of the 14th International Workshop on Algebraic Development Techniques, Bonas, France, November 1999, pp 45-46

[20] D. Marinov and S. Khurshid. TestEra: A Novel Framework for Testing Java Programs. 16th IEEE ASE 2001, San Diego, CA. Nov 2001.

[21] G. J. Myers, The Art of Software Testing, Wiley, New York, USA, 1979

[22] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. Proceedings of First Workshop on Runtime Verification, July 23, 2001, Paris, France.

[23] L. J. Osterweil et al. Strategic directions in software quality. ACM Computing Surveys, (4):738-750, Dec. 1996.

[24] G. Rothermel, R. Untch, C. Chu and M. J. Harrold. Test case prioritization: an empirical study. In Proceedings of ICSM, pages 179-188, Aug. 1999

[25] A. Srivastava and J. Thiagarajan, Effectively Prioritizing Tests in Development Environment, MSR-TR-2002-15, Feb. 2002.

[26] N. Tracey, J. Clark, K. Mander and J. McDermid. Integrating Automated Testing with Exception Freeness Proofs for Safety Critical Systems. In the Proceedings of 4th Australian Workshop on Safety Critical Systems and Software. Australian Computer Society. November 1999.

[27] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In Proceedings of International Conference on Software Maintenance, pages 44-53, 1998.

[28] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In Proceedings of the 8th International Symposium on Software Reliability Engineering, pages 230-238, Nov. 1997.

[29] A. Wong and M. Chechik. Formal Methods When Money is Tight. In Proceedings of the First Workshop On Economics-Driven Software Engineering Research, Los Angeles, California, May 1999.