International Journal of Engineering Research and General Science Volume 3, Issue 3, Part-2 , May-June, 2015
ISSN 2091-2730

Measuring Code Quality to Improve Specification Mining

Roshan Deshmukh, Prof.Umesh Kulkarni

KVCT's ARMIET, roshan982@gmail.com

Abstract— Every software Industry requires the quality of code. Formal specifications are mathematically based techniques whose purposes are to help with the implementation of systems and software. They are used to describe a system, to analyze its behavior, and to aid in its design by verifying key properties of interest through rigorous and effective reasoning tools. These specifications are formal in the sense that they have syntax, their semantics fall within one domain, and they are able to be used to infer useful information. Formal specifications can help with program testing, optimization, refactoring. However, they are difficult to write manually, and automatic mining techniques suffer from 90–99% false positive rates. To address this problem, we propose to augment a temporal-property miner by incorporating code quality metrics. We measure code quality by extracting additional information from the software engineering process, and using information from code that is more likely to be correct as well as code that is less likely to be correct.

Keywords— Specification mining, machine learning, software engineering, code metrics, program understanding

INTRODUCTION

Incorrect and buggy behavior in deployed software costs up to $70 billion each year in the US [7]. Thus debugging, testing, maintaining, optimizing, refactoring, and documenting software, while time-consuming, remain critically important.

Such maintenance is reported to consume up to 90% of the total cost of software projects .A key maintenance concern is incomplete documentation up to 60% of maintenance time is spent studying existing software(e.g.,[8]). Human processes and especially tool support for finding and fixing errors in deployed software often require formal specifications of correct program behavior(e.g.,[9]); it is difficult to repair a coding error without a clear notion of what "correct" program behavior entails. Unfortunately, while low-level program annotations are becoming more and more prevalent, comprehensive formal specifications remain rare.

Many large, preexisting software projects are not yet formally specified. Formal program specifications are difficult for humans to construct .and incorrect specifications are difficult for humans to debug and modify. Accordingly, researchers have developed techniques to automatically infer specifications from program source code or execution traces [2]. These techniques typically produce specifications in the form of finite state machines that describe legal sequences of program behaviors.

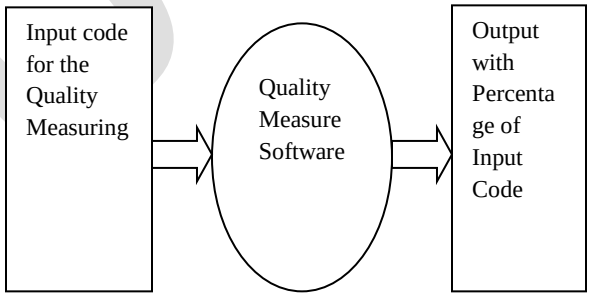

Fig.1.Block Diagram

LITERATURE SURVEY

API-based and Information Theoretic Metrics for Measuring the Quality of Software Modularization [10]. This system is developed using Object oriented software system. Create a set design principles for code modularization and produce set of metrics. Modularization quality is calculated using metrics such as structural, architectural and notions. There are three contributions such as coupling, cohesion and complexity metrics to modularize the software. This metrics seek to characterize a body of software according to the enunciated principles. Provide two types of experiments to validate the metrics.

Whaley *et al.* propose a static miner [1] that produces a single multi-state specification for library code. The JIST[2] miner refines Whaley *et al.*'s static approach by using techniques from software model checking to rule out infeasible paths. Gabel and Su [3] extend Perracotta using BDDs, and show both that two-state mining is NP-complete and some specifications cannot be created by composing two-state specifications. Lo *et al.* use learned temporal properties, such as those mined in this article, to *steer* the learning of finite state machine behavior models [4].Shoham *et al.* [5] mine by using abstract interpretation, where the abstract values are specifications

MINING CHARACTERISTICS

This section shows the underlying concepts of mining techniques and their limitations which encourage the researchers to step into incorporating code quality metrics. Specification mining techniques produces specifications but still they have high false positive rates. The Comparison between most of these approaches is provided in the Table 1.

In WN miner [6] the specification mining was motivated by the observations of run-time error handling mistakes. In other approaches examining such mistakes, the code frequently violates simple API specifications in exceptional situations. Despite the proliferation of specification-mining research, there is not much report on issues pertaining to the quality of specification miners. This technique is same as that of Engler et al. but is based on assumptions about run time errors, chooses candidate event pairs differently, presents significantly fewer candidate specifications and ranks presented candidates differently.

In a normal Table1.A Comparison study execution, events „a' and „b' may be separated by other events and difficult to discern as a pair. After an error has occurred, however, the cleanup code is usually much less cluttered and contains only operations required for correctness. The candidate specifications are filtered using varied criteria such as exceptional control flow, one error, data path etc.

This highlights the practical importance of the algorithmic assumptions, in particular the use of exceptional control flow. It can serve as a requirement for acceptance. It can even assist inspections by helping to target effort at parts of a program that may need improvement. Though this miner select specifications from software artifacts and finds per-program specifications for error detection, it does not have profound results in bug finding.

Strauss, ECC and WN technique were all good at yielding specifications that found bugs. The WN technique found all bugs reported by other techniques on these benchmarks and did so with the fewest false positives.

QUALITY METRICS

Code metrics like LOC and Cyclomatic Complexity examines the internal complexity of a procedure whereas this structure metrics examines the relationship between a section of code and the rest of the system. Process oriented metrics are used through the different phases of the software life cycle. Measurement on quality should concentrate on the early phases in the life cycle to improve the quality of software and decrease of development and maintenance costs. Defects must be tracked to the release origin which is the portion of the code that contains the defects and at what release the portion was added, changed, or enhanced.

When calculating the defect rate of the entire product, all defects are used; when calculating the defect rate for the new and changed code, only defects of the release origin of the new and changed code are included. On the one hand, the process quality metrics simply means tracking defect arrival during formal machine testing for some organizations. On the other hand, some software organizations with well-established software metrics programs cover various parameters in each phase of the development cycle.

Miners	Characteristics	Comment
Engler et al.	Use two state temporal properties.	High false positive rates
Whaley et al.	Produces Single multi state specification	Human intervention
Strauss	focuses on machine learning to learn a Single specification from traces	Use of single specification is not sufficient
JIST	Refines Whaley et al. technique to mainly disregard infeasible paths	Handles only simple subset of Java
WN miner	Selecting specifications from software artifacts	Does not have profound results in finding bugs
Claire	Use measurements of trustworthiness of source code to mine specifications	Does not give adequate results over precision.

ACKNOWLEDGMENT

I would like to express my sincere gratitude towards my guide Prof.Umesh Kulkarni for the help, guidance an encouragement in the development of this methodology. They supported me with scientific guidance, advice and encouragement, and were always helpful and enthusiastic and this inspired me in my work. I have benefitted from numerous discussions with guide and other colleagues.

CONCLUSION

Formal specifications have a variety of applications including testing, maintenance, optimization, refactoring, documentation, and program repair. However, such specifications are difficult for human programmers to produce and verify manually, and existing

automatic specification miners that discover two-state temporal properties have prohibitively high false positive rates. The goal of this survey is to support the study on the legacy of generating specifications to the new automatic techniques. It helps to get an insight into this dynamic field of study in Specification Mining. Since the object orientation is emerging in all kinds of applications, it is also welcome in the specification mining process. It is mentioned to be dynamic, as these approaches are under development and it steps higher everyday to achieve efficiency in capturing specifications.

REFERENCES:

[1]J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," in ISSTA, 2002.

[2]R.Alur, P.Cerny, P.Madhusudan, and W. Nam, "Synthesis of interface specifications for Java classes," in POPL, 2005.

[3]M. Gabel and Z. Su, "Symbolic mining of temporal specifications, "in ICSE, 2008, pp. 51–60.

[4]D. Lo, L. Mariani, and M. Pezz`e, "Automatic Steering of Behavioral Model Inference," in FSE. ACM, 2009, pp. 345–354.

[5]S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *International* Symposium on Software Testing and *Analysis*, 2007, pp. 174–184.

[6]W. Weimer and G.C. Necula, "Mining Temporal Specifications for Error Detection," Proc. Int'l Conf. Tools and Algorithms for the Construction and Analysis of Systems, pp. 461-476, 2005.

[7]National Institute of Standards and Technology, "The economic impacts of inadequate infrastructure for software testing," Tech. Rep. 02-3, May 2002.

[8]S. L. Pfleeger, Software Engineering: Theory and Practice. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[9]D. Malayeri and J. Aldrich, "Practical exception specifications."In Advanced Topics in Exception Handling Techniques, 2006, pp. 200–220.

[10]Sarkar, S.; Kak, A.C.; Rama, G.M. "Metrics for Measuring the Quality of Modularization of Large-Scale Object-Oriented Software", Software Engineering, IEEE Transactions on, on page(s): 700 - 720 Volume: 34, Issue: 5, Sept.-Oct. 2008