Quantitative Analysis of Fault and Failure Using Software Metrics

Shital V. Tate[1]

[1] Bharati Vidyapeeth Deemed University

Received: 11 December 2011 Accepted: 31 December 2011 Published: 15 January 2012

Abstract

It is very complex to write programs that behave accurately in the program verification tools. Automatic mining techniques suffer from 90â??"99

Index terms—

1 Introduction

oftware remains buggy and testing is still the leading approach for detecting software errors. Incorrect and buggy behaviour in deployed software costs up to $70 billion each year in the US [1]. Thus debugging, testing, maintaining, optimizing, refactoring, and documenting software, while timeconsuming, remain significantly important. Such maintenance is reported to consume up to 90% of the total cost of software projects [2]. Maximum maintenance time is spent studying existing software since maintenance concern is incomplete documentation.

Consistently, however, verification tools require specifications that describe some aspect of program accuracy. Creating accurate specifications is difficult, time-consuming and error-prone. Verification tools can only point out disagreements between the program and the specification. Even assuming a sound and complete tool, an defective specification can still yield false positives by pointing out non-bugs as bugs or false negatives by failing to point out real bugs. Crafting specifications typically requires program-specific knowledge.

Specification mining can be compared to learning the rules of English grammar by reading essays written by high school students; we propose to focus on the essays of passing students and be doubtful of the essays of failing students. We claim that existing miners have high false positive rates in large part because they treat all code equally, even though not all code is created equal. For example, consider an execution trace through a recently modified, rarely-executed piece of code that was copied-and-pasted by an inexperienced developer. We argue that such a trace is a poor guide to correct behaviour when compared with a well-tested, infrequently-changed, and commonly-executed trace.

Various pre-existing software projects are not yet formally specified [3]. Formal program specifications are difficult for humans to construct [4], and incorrect specifications are difficult for humans to debug and modify [5]. Accordingly, researchers have developed techniques to automatically infer specifications from program source code or execution traces [6], [7], [8], [9]. These techniques typically produce specifications in the form of finite state machines that describe legal sequences of program behaviours.

Unfortunately, these existing mining techniques are insufficiently precise in practice. Some miners produce large but approximate specifications that must be corrected manually [5]. As these large specifications are indefinite and difficult to debug, this article focuses on a second class of techniques that produce a larger set of smaller and more precise candidate specifications that may be easier to evaluate for correctness. These specifications typically take the form of two-state finite state machines that describe temporal properties, e.g. "if event a happens during program execution, event b must eventually happen during that execution." Twostate specifications are limited in their expressive power; comprehensive API specifications cannot always be expressed as a collection of smaller machines [8].

Recognize and illustrate lightweight, automatically collected software features that fairly accurate source code quality for the purpose of mining specifications. In this approach explain how to lift code quality metrics to metrics on traces, and empirically measure the utility of our lifted quality metrics when applied to previous static specification mining techniques. To avoid false positives recommend two novel specification mining techniques that use our automated quality metrics to learn temporal safety specifications. The main contributions of this project are: ? A set of source-level features related to software engineering processes that capture the trustworthiness of code for specification mining. We analyze the relative analytical power of each of these features. ? Experimental evidence that our notions of trustworthy code serve as a basis for evaluating the trustworthiness of traces. We provide a characterization for such traces and show that offthe-shelf specification miners can learn just as many specifications using only 60% of traces. ? A novel automatic mining technique that uses our trust-capturing features to learn temporal safety specifications with few false positives in practice. We evaluate it on over 800,000 lines of code and explicitly compare it to two previous approaches.

2 II.

3 On going methodology

Our basic mining technique learns specifications that locate more safety-policy violations than previous miners (740 vs. 426) while presenting far fewer false positive specifications (107 vs. 567). When focused on precision, our technique obtains a low 5% false positive rate, an order-of-magnitude improvement on previous work, while still finding specifications that locate 265 violations. To our knowledge, this is the first specification miner that produces multiple candidate specifications and has a false positive rate under 90%. i.

4 Approach

In this approach present a specification miner that works in three stages: 1. Statically estimate the trustworthiness of each code fragment. 2. Lift that judgment to traces by considering the code visited along a trace. 3. Weight the contribution of each trace by its trustworthiness when counting event frequencies for specification mining.

The code is most trustworthy when it has been written by experienced Programmers who are familiar with the project at hand, when it has been well-tested, and when it has been mindfully written.

5 b) Mining Temporal Specification for Error Detection

If we use implicit language-based specifications (e.g., null pointers should not be dereferenced) or to reuse standard library specifications then it can reduce the cost of writing specifications. More recently, however, a variety of attempts have been made to conclude program-specific temporal specifications and API usage rules automatically. These specification mining techniques take programs (and possibly dynamic traces, or other hints) as input and produce candidate specifications as output. Basically specifications could also be used for documenting, refactoring, testing, debugging, maintaining, and optimizing a program. Centre of attention is that finding and evaluating specifications in a particular context: given a program and a generic verification tool, what specification mining technique should be used to find bugs in the program and thereby improve software quality? Thus we are concerned both with the number of ”real” and ”false positive” specifications produced by the miner and with the number of ”real” and ”false positive” bugs found using those ”real” specifications.

In this methodology propose a novel technique for temporal specification mining that uses information about program error handling. Our miner assumes that programs will generally adhere to specifications along normal execution paths, but that programs will likely violate specifications in the presence of some run-time errors or exceptional situations. Intuitively, error-handling code may not be tested as often or the programmer may be unaware of sources of run-time errors. Taking advantage of this information is more important than ranking candidate policies. i.

6 Contributions

7 ? Propose a novel specification mining technique

based on the observation that programmers often make mistakes in exceptional circumstances or along uncommon code paths. ? Present a qualitative comparison of five miners and show how some miner assumptions are not wellsupported in practice. ? Finally, we give a quantitative comparison of our technique's bug-finding powers to generic ”library” policies. For our domain of interest, mining finds 250 more bugs. We also show the relative unimportance of ranking candidate policies. In all, we find 69 specifications that lead to the discovery over 430 bugs in 1 million lines of code.

8 III.

Proposed system for quantitative analysis of fault and failure

In proposed system, aim to develop a system which can be used to measure the quality of the code considering different aspects affecting the quality of the code. The term quality of the code can be explained using different factors such as code clone, author rank, code churn, code readability, path feasibility etc.

To Present a new specification miner that works in three stages. First, it statically estimates the quality of source code fragments. Second, it lifts those quality judgments to traces by considering all code visited along a trace. Finally, it weights each trace by its quality when counting event frequencies for specification mining.

9 Global Journal of Computer Science and Technology

Volume XII Issue XII Version I develops an automatic specification miner that balances true positives -as required behaviours -with false positives -non-required behaviours. We claim that one important reason that previous miners have high false positive rates is that they falsely assume that all code is equally likely to be correct. For example, consider an execution trace through a recently modified, rarely-executed piece of code that was copied and-pasted by an inexperienced developer. We believe that such a trace is a poor guide to correct behaviour, especially when compared with a well-tested, stable, and commonly-executed piece of code. Patterns of specification adherence may also be useful to a miner: a candidate that is violated in the high quality code but adhered to in the low quality code is less likely to represent required behaviour than one that is adhered to on the high quality code but violated in the low quality code. We assert that a combination of lightweight, automatically collected quality metrics over source code can usefully provide both positive and negative feedback to a miner attempting to distinguish between true and false specification candidates.

Code quality information may be gathered either from the source code itself or from related artefacts, such as version control history. By augmenting the trace language to include information from the software engineering process, we can evaluate the quality of every piece of information supporting a candidate specification (traces that adhere to a candidate as well as those that violate it and both high and low quality code) on which it is followed and more accurately evaluate the likelihood that it is valid.

The system architecture of the system is as in following figure, which explains the modules to be generated.

10 Conclusion

Testing, maintenance, optimization, refactoring, documentation, and program repair these are the various applications of formal specification. Though human programmers should not produce and verify such specification manually. These technique is also problematic since it treat all parts of program as equally indicative as correct behaviour. We encode this intuition using dependability metrics such as analytical execution frequency, copy paste code measurements, code duplication software readability or path feasibility. We compare the bug finding power of various miners. This technique improves the performance of existing trace based miners by focusing on high quality traces. Our technique is also useful to improve the quality of code through specification mining. [1]

Figure 1:

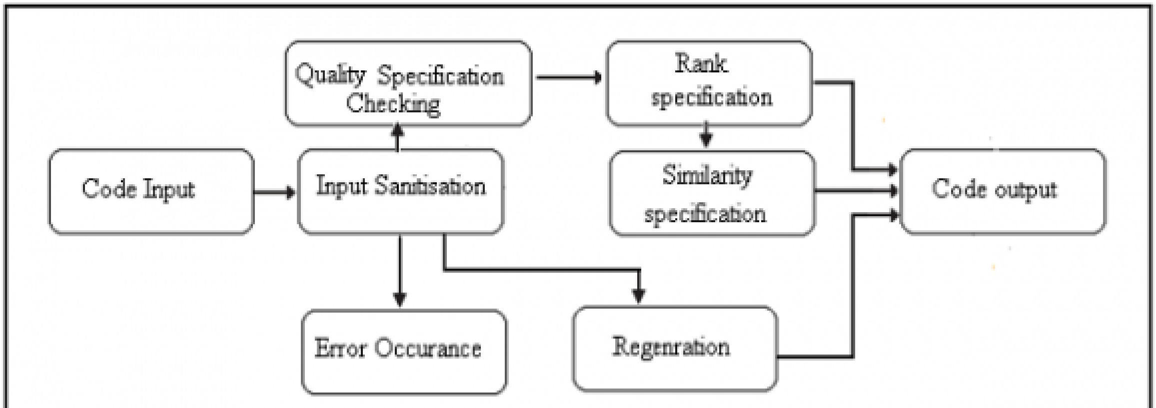
1 

fig: Proposed Architecture

Figure 2: Figure 1 :

[J. ACM ()] , *J. ACM* 2005. 52 (3) p. .

[Mccabe ()] ‘A complexity measure’. T J Mccabe . *IEEE Trans. Software Eng* 1976. 2 (4) p. .

[Chidamber and Kemerer ()] ‘A metrics suite for object oriented design’. S R Chidamber , C F Kemerer . *IEEE Trans. Softw. Eng* 1994. 20 (6) p. .

[Krinke ()] *A study of consistent and inconsistent changes to code clones*, J Krinke . 2007. WCRE. IEEE Computer Society. p. .

[Whaley et al. (ed.) ()] *Automatic extraction of object-oriented component interfaces*, J Whaley , M C Martin , M S Lam . ISSTA (ed.) 2002.

[Engler et al. ()] ‘Bugs as inconsistent behavior: A general approach to inferring errors in systems code’. D R Engler , D Y Chen , A Chou . *Symposium on Operating Systems Principles*, 2001. p. .

[Engler et al. ()] ‘Bugs as inconsistent behaviour: A general approach to inferring errors in systems code’. D R Engler , D Y Chen , A Chou . *Symposium on Operating System Principles*, 2001. p. .

[Kapser and Godfrey ()] *Cloning Considered Harmful” in WCRE*, C Kapser , M W Godfrey . 2006. p. .

[Ammons et al. ()] ‘Debugging temporal specifications with concept analysis’. G Ammons , D Mandelin , R Bod´?k , J R Larus . *Programming Language Design and Implementation*, 2003. p. .

[Kayed and Chang (2010)] ‘FiVaTech: Page-Level Web Data Extraction from Template Pages’. Mohammed Kayed , Chia-Hui Chang . *IEEE Transactions On Knowledge And Data Engineering* February 2010. 22 (2) .

[Das ()] ‘Formal specifications on industrial-strength code from myth to reality’. M Das . *Computer-Aided Verification*, 2006. p. 1.

[Le Goues and Weimer] ‘Measuring code quality to improve specification mining’. Claire Le Goues , Westely Weimer . *IEEE Trans. Software Eng*

[Ammons et al. ()] ‘Mining specifications’. G Ammons , R Bodik , J R Larus . *Principles of Programming Languages*, 2002. p. .

[Weimer and Necula ()] ‘Mining temporal specifications for error detection’. W Weimer , G C Necula . *TACAS* 2005. p. .

[Sanchez et al. (2007)] *On the Sustained Use of a Test Driven Development Practice at IBM,” in Agile*, J C Sanchez , L Williams , E M Maximilien . 2007. August 2007. IEEE Computer Society. p. .

[Weimer and Mishra ()] ‘Privately finding specifications’. W Weimer , N Mishra . *IEEE Trans. Software Eng* 2008. 34 (1) p. .

[Seacord et al. ()] R C Seacord , D Plakosh , G A Lewis . *Modernizing Legacy Practices*, 2003.

[Chen et al. ()] ‘Setuid demystified’. H Chen , D Wagner , D Dean . *USENIX Security Symposium*, 2002. p. .

[Detlefs et al.] *Simplify: a theorem prover for program checking*, D Detlefs , G Nelson , J B Saxe .

[Goues and Weimer ()] ‘Specification mining with few false positives’. C , Le Goues , W Weimer . *TACAS*, 2009. p. .

[Gabel and Su ()] ‘Symbolic mining of temporal specifications’. M Gabel , Z Su . *ICSE*, 2008. p. .

[The economic impact of inadequate infrastructure for software testing (2002)] *The economic impact of inadequate infrastructure for software testing*, may 2002. National Institute of Standards and Technology (Tech. Rep. 02-3)

[Nagappan and Ball ()] ‘Using software dependencies and churn metrics to predict field failures: An empirical case study’. N Nagappan , T Ball . *ESEM*, 2007. p. .

[Penta and German ()] ‘Who are source code contributors and how do they change?’. M , Di Penta , D M German . *in Working Conference on Reverse Engineering*, 2009. IEEE Computer Society. p. .