International Journal of Scientific Research in Computer Science, Engineering and Information Technology
© 2018 IJSRCSEIT | Volume 3 | Issue 1 | ISSN : 2456-3307

An Approach to Mutation Testing with Automated Debugging Tools for Software Testing

Rajalakshmi S[*1], Aathira P[1], Sakthi Kumaresh[2]

[1]Research Scholar, BCA, Department of Computer Science, M.O.P. Vaishnav College for Women, Chennai, Tamil Nadu, India

[2]Associate Professor, BCA, Department of Computer Science, M.O.P. Vaishnav College for Women, Chennai, Tamil Nadu, India

ABSTRACT

Debugging is an extremely difficult and time consuming task in software testing. Individuals have put in a great deal of effort in creating automated tools and techniques for supporting different debugging tasks. Most techniques that are in current practice focus on picking subsets of possibly erroneous statements and prioritizing them based on some standard. A program faces a failure in certain circumstances. The overall objective of this study is to examine how software developers/testers utilize and attain benefit from these automated tools. We also perceive on possible directions for future work in the zone of automated debugging and try to combine automated debugging techniques (designed based on delta debugging algorithm) and mutation testing with a specific end goal to lessen the measure of cost and time involved in the Software Testing phase.

Keywords : Statistical debugging, user studies, testing, delta debugging, debugging aids

I. INTRODUCTION

On occurrence of a software failure, developers perform three main tasks to eliminate the cause for the failure. Fault localization is the first task involving of identifying the statements in the program responsible for failure [18]. The next is fault understanding, which involves understanding root cause for the failure. Finally, fault correction involves in determining how code can be modified to remove the root cause. These three tasks are collectively termed as debugging.

Debugging is always a dreary and tedious experience that plays a critical part of cost in maintaining software [1]. Hence, reducing the cost of debugging through methods that can enhance efficiency and effectiveness of such tasks is vital. Over the most recent couple of years, there have been an exceptional number of research techniques that help automated debugging activities [2, 3, and 20]. However, there are many difficulties in these techniques that must be tended before proceeding to place them in the hands of developers.

In this paper, we give an insight on how mutation testing can be done with automated debugging tools to prove and isolate failure causes and speedup software testing. Basically, this method sets up subsets of the original circumstances, and tests these configurations whether the failure still occurs. Eventually, these methods return a subset of circumstances where every single circumstance is pertinent for delivering the failure.

Automated Debugging Techniques

Throughout the years, analysts have characterized progressively complex debugging methods, moving from for the manual to profoundly automated ones. Simultaneously, foundation to help these tasks has

also been developed. Therefore, the aggregate work done is wide [17].

The delta debugging algorithm sums up and simplifies the failure causing test cases to a small test case that still produces the failure. It secludes the differences between a failing and a passing test case [13].

II. Literature Review

Weiser proposed one of the first techniques for supporting automated debugging and, in particular, faults localization: program slicing [4, 5]. Given a program P and a variable v used at a statement s in P, slicing computes all of the statements in P that may affect the value of v at s. By definition, if the value of v in s is erroneous, then the faulty statements that led to such erroneous value must be in the slice. Any statement can be safely ignored during debugging if it is not in the slice. Although slicing can generate sets of related statements, in most sensible cases these sets are too substantial to be helpful in any way for debugging [6]. To address this issue, researchers found distinctive varieties of slicing went for diminishing the span of the processed cuts. Dynamic slicing figures slices for a specific execution. In the upcoming years, diverse variations of dynamic slicing have been proposed with regards to debugging, for example, pruned slices [6], data-flow slices [7], relevant slices [8] and critical slices [9]. These methods can extensively diminish the size of slices, and hence possibly enhance debugging. Yet, these debugging techniques are rarely used in practice.

Studies with Programmers

In the initial study of Weiser [4], 3 programs were examined by 21 programmers whose size scaled between 75 and 150 LOC. This research did not directly assess if programmers could productively debug with slicing. Overall, slices were perceived altogether significantly more regularly than other different fragments, which recommend that software engineers have a tendency to follow the flow of execution when exploring an error while debugging [16].

The broadest assessment of a debugging approach till date is the experimental investigation of the Whyline tool [10]. Whyline centers on helping amateur clients with defining theories and making natural inquiries about a program's conduct. Whyline demonstrates what a matured program slicing tool can accomplish by blending perception, dynamic cutting, programmed thinking, and a smooth UI in a single tool. Members that utilized Whyline could finish the job twice as quick than members utilizing just a conventional debugger [11].

In outline, as the short study in this area appears, experimental proof of the convenience of numerous automated troubleshooting approaches is restricted, on account of slicing for most other types of systems, when not totally absent. This circumstance makes it hard to evaluate the pragmatic adequacy of the procedures proposed to understand which qualities of a strategy can make it fruitful.

The limitations of slicing-based approaches can be overcome by following a different philosophy, aimed by an alternative family of debugging techniques. These techniques observe the characteristics of the executions of failing programs and compare them to characteristics of passing executions, thereby identifying faulty code. This type of information can be gathered only by rerunning the program against the input that caused the failure. In general, the potential effectiveness of such techniques remains unknown without a clear understanding of how developers would use these in practice.

III. Discussion and Findings

In this section, we describe our findings about the behavior of programmers and discuss about developers' want values, overviews, and explanations.

3.1. Behaviour of Programmers

3.1.1. Programmers Fixed the Failure Sometimes without Tool, Not the Fault

Ideally, software engineers are guided based on their understanding of the failure to a root cause while debugging. However, this does not always happen in practice. Sometimes developers find that, they can stop the failure from occurring without actually fixing the fault through experimentation and control of the program.

.

Observation 1 - Developers may be ensured of fault correction with the help of automated debugging tool instead of simply patching failures

3.1.2 Developers Need Overviews, Explanations and Values

A combination of the ranking tool was used by developers (especially for locating faults) and conventional troubleshooting (especially for understanding fault)[19]. Automated Test cases are executed during automated debugging. Exploring unfamiliar code can suggest many promising starting places for developers by using automated debugging tool. The tool displays appropriate code entrance points thereby helping in program understanding even though the tool couldn't pinpoint the correct location of the fault. Developers quickly disregard the tool if they felt they could not trust the results or understand how such results were computed. Developers would be permitted to explore the failure in a more methodical and data-driven manner, if they were provided with such details. Developers are currently presented with a set of apparently disconnected statements and no additional support when using these tools, rather than working with the familiar and reliable step-by-step approach of a traditional debugger.

Observation 2–By providing information on results that include test cases, data values and information about slices, faults can be easily identified.

Threats to Validity

Our study has concentrated more on skilful developers. Students were the participants of our study, who did not have the similar experience of expert developers, which could confine what can be deduced from the research. Yet quite a few members had quite a long or little experience as developers. Our outcomes may therefore not sum up to other projects and faults, and investigations are expected to affirm our observations made in the beginning and analyses. Nevertheless, our outcomes are encouraging and enabled us to make some intriguing, yet preliminary, perceptions to explain additional studies and give a technique for governing such studies. An ultimate warning to credibility relates to the nature of the failure information.

3.3 Where does it lead to?

- Hybrid, semi-automated fault localization techniques
- Debugging of field failures (with limited information)
- Failure understanding and explanation
- (Semi-)automated repair and workarounds

IV. Proposed Methodology

4.1 Overview of the Design

To overcome the issues and failures faced by the above tools and techniques, we hereby propose our idea to combine automated debugging tools with mutation testing so that we could use these tools to identify the mutants faster and also to ensure that our program works in an expected way [14]. Delta debugging, a single algorithm, is enough to decide the situations that lead to failure. Delta debugging tests a program systematically and automatically to confine failure initiating situations such as the

program input, changes to the program code, or executed statements [12, 15].

teps involved in this methodology can be listed as follows:

Step 1: Input the faulty programs for testing.
Step 2: Generate mutants into the programs which result in patched programs.
Step 3: Input the test cases.
Step 4: Validate the patched programs against the test cases using automated debugging tools, which are designed based on delta debugging algorithm to minimize the time involved in debugging.
Step 5: The output generated is a repaired program that does not have any faults and executing without any failure.

The architectural view of how this can be implemented is shown below:

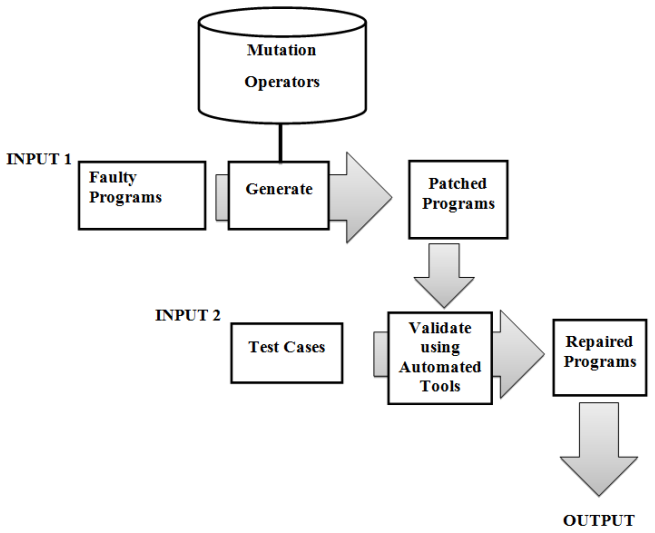

Figure 1 : Integration of mutuant operators and automated debugging

Study Results

We ran an initial experimental test through Vaultry, a bank application program coded in C++ consisting of 675LOC. The results obtained were far superior when compared to normal automated debugging tools. The tools were able to locate the bugs faster using delta debugging algorithm rather than the tools that work on normal tracing procedures. Thereby, we recommend this technique to be implemented at a large level so as to minimize testing time and cost involved in testing phase.

V. Future Work

Delta debugging speeds up the main issue in debugging – the timeframe. Therefore, by developing automated tools which can work on delta debugging algorithms, the computation can be done at a faster rate. Our long-term vision is that, to debug a program, one should setup an appropriate function for testing. At that point, one can allow the computer to do the debugging, separating failure situations by using a blend of program analysis and automated testing. Automatic isolation of failure is no longer past the cutting edge. It is just a question of how much computing power and program analysis you are willing to spend on it.

VI. Conclusion

Researchers have envisioned how automated debugging tools can help developers fix defects in code for 30 years. In this paper, we obtained both positive and negative results by having real programmers act this vision out. We find that the ranking tool is considered no more effective than traditional debugging for our more challenging task even when an artificially-high rank is used. The defects observed in the ranking tool may show general shortcomings in today's automated debugging techniques that limit their viability. The proposed methodology checks for the effectiveness and accuracy of a program to identify the faults or errors in the system framework within a little timeframe and at the least cost. Developers have been waiting a long time for usable automated debugging tools, and we have officially gone far from the beginning of debugging. We must steer research towards more promising directions, to further advance the level of development in this area, that take into account the way programmers actually debug in real scenarios.

VII. REFERENCES

[1]. I. Vessey. Expertise in debugging computer programs.International Journal of Man-Machine Studies: A Process Analysis, 23(5):459{494, 1985.

[2]. T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counter example traces. In Proceedings of the Symposium on Principles of Programming Languages (POPL 03), pages 97{105, New Orleans, LA, USA, 2003.

[3]. H. Cleve and A. Zeller. Finding failure causes through automated testing. In Proceedings of the International Workshop on Automated Debugging (AADEBUG 00), Munich, Germany, 2000.

[4]. M. Weiser. Program slicing.In Proceedings of the International Conference on Software Engineering (ICSE 81), pages 439{449, San Diego, CA, USA, 1981.

[5]. M. Weiser. Program slicing.IEEE Transactions on Software Engineering, 10(4):352{357, 1984.

[6]. X. Zhang, N. Gupta, and R. Gupta. Pruning dynamic slices with confidence. In Proceedings of the Conference on Programming Language Design and Implementation (PLDI 06), pages 169{180, New York, NY, USA, 2006.

[7]. X. Zhang, R. Gupta, and Y. Zhang.Precise dynamic slicing algorithms.In Proceedings of the International Conference on Software Engineering (ICSE 03), pages 319{329, Washington, DC, USA, 2003.

[8]. T. Gyimothy, A. Beszedes, and I. Forgacs.An efficient relevant slicing method for debugging.In Proceedings of the European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 99), pages 303{321, London, UK, 1999.

[9]. R. A. DeMillo, H. Pan, and E. H. Spafford.Critical slicing for software fault localization.In Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 96), pages 121{134, San Diego, CA, USA, 1996.

[10]. A. J. Ko and B. A. Myers. Debugging reinvented: asking and answering why and why not questions about program behavior. In Proceedings of the International Conference on Software Engineering (ICSE 08), pages 301{310, Leipzig, Germany, 2008.

[11]. A. J. Ko and B. A. Myers. Finding causes of program output with the Java Whyline. In Proceedings of the International Conference on Human Factors in Computing Systems (CHI 09), pages 1569{1578, Boston, MA, SA, 2009.

[12]. M. Ducass´e (ed). In Proceedings of the Fourth International Workshop on Automated Debugging (AADEBUG 2000), August 2000, Munich.

[13]. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," in IEEE Transactions on Software Engineering, vol. 28, no. 2, pp. 183-200, Feb 2002.

[14]. W. Visser, "What makes killing a mutant hard," 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), Singapore, 2016, pp. 39-44.

[15]. Groce M. A. Alipour C. Zhang Y. Chen J. Regehr "Cause reduction: Delta debugging even without bugs" Software Testing Verification & Reliability vol. 26 no. 1 pp. 40-68 Jan. 2016.

[16]. T. S. Gadge and N. Mangrulkar, "Approaches for automated bug triaging: A review," 2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA), Bangalore, 2017, pp. 158-161.

[17]. E. C. Campos and M. d. A. Maia, "Common Bug-Fix Patterns: A Large-Scale Observational Study," 2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, ON, 2017, pp. 404-413.

[18]. KOCHHAR, Pavneet Singh; XIA, Xin; David LO; and LI, Shanping. Practitioners' expectations on automated fault localization.

(2016). Proceedings of the 25th ACM International Symposium on Software Testing and Analysis: ISSTA 2016, Saarbrucken, Germany; 2016 July 18-20.

[19]. Tien-Duy B. Le1 , David Lo1 , Claire Le Goues , and Lars Grunske. A learning-to-rank based fault localization approach using likely invariants, 2016. In the Proceedings of the 25th ACM International Symposium on Software Testing and Analysis: ISSTA 2016, Saarbrucken, Germany; 2016 July 18-20.

[20]. Muhammad Ali Gulzar, MatteoInterlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. “Automated Debugging in Data-Intensive Scalable Computing.”In ACM Symposium on Cloud Computing September 25, 2017. Santa Clara, California.