International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 07 Issue: 03 | March - 2023 Impact Factor: 7.185 ISSN: 2582-3930

Automated Debugging : Still a Dream ?

Deepak Varma, Alwala Nehansh, Jatin Adya B, Anshul Gupta

Hyderabad Institute of Technology and Management

Abstract - Software debugging is the process of finding and fixing incorrect statements in programs. The process of debugging takes a lot of time and is challenging. Therefore, the field of automated debugging, which is focused on automating the discovery and correction of a failure's underlying cause, has made huge progress in the past. By applying automated approaches to identify and correct any erroneous statements in a program, the cost of producing software may be significantly decreased while also improving the quality of the final product. The purpose of this paper is to shed light on the application of automated debugging in the current market scenario. Techniques like Delta Debugging, Path-based Weakest Preconditions, Fault Localization, and Program Slicing have been demonstrated to be quite effective in dealing with the identification and resolution of inconsistencies. This paper also aims to examine the question, "Is Automated Debugging still a dream? ".

Key Words: Automated Debugging, Delta Debugging, Program Slicing.

1.INTRODUCTION

The process of developing, designing, implementing, and maintaining software is known as software development.. With the growing demand for software in the industry, programmers must provide a plethora of new features to keep customers satisfied, which may increase the number of bugs. A programmer must debug a program when it fails to fix the problem. A program's bugs are found and fixed during the debugging process. Three crucial actions are used to achieve this. The first step, fault localization, involves identifying the specific program statements that caused the failure. The second step, fault understanding, entails figuring out where the failure occurred. The third activity, fault correction, involves changing the existing code and, in some cases, the programming strategy. Debugging is a tedious and expensive process that significantly raises the cost of software maintenance. Software development costs can be significantly reduced by using automated methods to identify and correct incorrect statements in the program.

Several research methodologies have been created in recent years to aid in the automation or semi-automation of a variety of debugging jobs. In the history of automated debugging, One of the first methods for assisting program slicing was proposed by Weiser [4,5]. Slicing identifies all statements in a program P that have the potential to change the value of a variable v used at a statement s in P. Although slicing can produce sets of relevant statements, these sets are frequently too large to aid in debugging [8].

Several approaches to dynamic slicing have been proposed to address this issue in the years since Korel and Laski presented dynamic slicing, which computes slices for specific executions, such as critical slices, relevant slices [7], data-flow slices, and pruned slices [8]. These methods can significantly reduce the size of slices, which may help with debugging.

Because the groups of relevant statements discovered are generally quite large, slicing-based debugging techniques are rarely used in practice. Other methods for detecting potentially problematic code include comparing the characteristics of unsuccessful program executions to those of successful executions. This broad category of techniques has the drawback that they are only interested in minimizing the number of statements that developers must look at when analyzing a failure, presuming that looking at a flawed statement in isolation is sufficient for a developer to find and correct the corresponding bug.

With approaches like BigSift, which makes automated debugging a reality, the future of automated debugging seems bright. Given a test function, BigSift finds a minimum set of error-causing input records accountable for an undesirable output.

For a long time, programmers have waited for usable automated debugging tools, and we've come a long way since debugging began. More research should be done on more promising paths that take into account how programmers debug in real-world circumstances to advance the current status of debugging.

2. Debugging Techniques and Tools

Over the years, researchers have defined increasingly complex debugging strategies, moving from primarily manual to highly automated techniques. The execution of each task as well as the change in the state of the software can be monitored using the event log. Even for distributed and large software programs, reading through an event log can take some time. Because of this, cutting-edge methods like program slicing and delta debugging have been created.

2.1 Delta Debugging

Every bug in the database describes a complex scenario that leads to software failure. Because they may contain a lot of irrelevant information, many bug reports may be ignored. Delta debugging is an automated technique that analyzes a test case that results in a bug and converts bug reports into minimal test cases in which every part of the input is tested significantly in reproducing the failure, resulting in simpler bug reports as shown in Fig.1.Examining an output file would be absurd to simplify the input file while still generating the same failure. The delta debugging technique automates this method of reducing input through repeated trials[7].

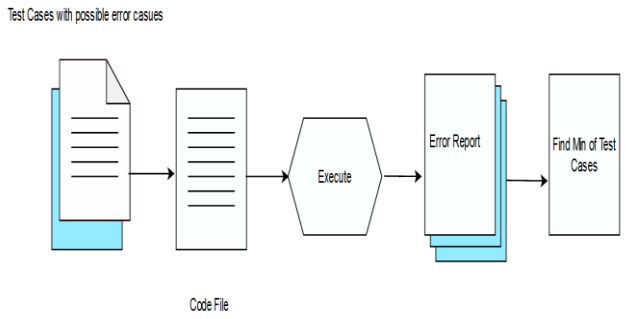

Fig. 1. Delta Debugging workflow

Before we can describe the algorithm, we must first define the process. The delta debugging technique, in general, deals with circumstances that can cause a change in program behavior. All of the program's and its environment's possible behaviors are included in these variable circumstances. Other applications of Delta debugging include locating failure-inducing code changes in programs. Given two versions of a program, one that works correctly and the other that fails, the delta debugging algorithm can be used to search for changes that are responsible for the failure.

2.2 Program Slicing

Program slicing is a method that concentrates on the areas of a program that might have caused the failure.
A program slice is what this method creates; it's a segment of the program execution that's pertinent to a particular state or behavior.On statement dependencies, slices are based:A statement S2 is dependent upon a statement S1 if S1 has an impact on the program state that S2 accesses.

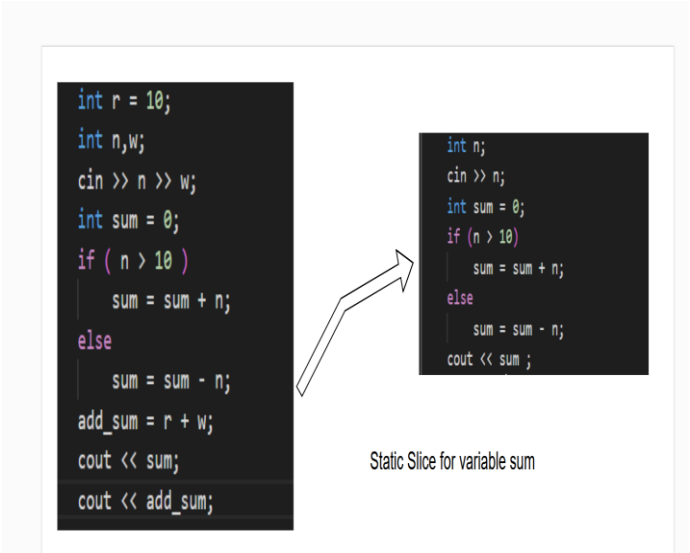

Fig. 2. Program Slicing

As shown in Fig. 2, a program slice is created by transitively closing all dependencies that begin with a statement. When debugging, computing the backward slice for a failing statement yields all statements that might have contributed to the failure. Static and dynamic slicing differ greatly from one another. A dynamic slice only applies to the failing run and is therefore more accurate, whereas a static slice applies to all possible runs and is computed without making any assumptions about a particular program run.

2.3 Indus

Indus is a module that contains the implementation of algorithms and data structures that are common to analyses and transformations that are or will be part of Indus. This module contains interface definitions common to most analyses and transformations to provide a framework in which various analyzes and transformations can be easily combined to form systems[7]. There are 2 more modules supported by Indus

- StaticAnalyses is a collection of static analyses such as object-flow analysis, escape analysis, and dependence analysis.The analyzes in this module make use of common Indus interfaces and implementations and may define/provide new interfaces/implementations for new analyses.
- The Java Program Slicer module contains the core Java programme slicer implementation as well as adapters that deliver the slicer in other applications such as Bandera and Eclipse.

2.4 Algorithmic Tracing

This method of debugging employs passive user contact, in which the user must respond but has no control over the procedure. This implies that there is no way for the user to switch between algorithms[3]. A large percentage of software uses a variety of techniques to perform various operations such as searching, sorting, and data fetching to and from modules. The most well-known algorithms include:

- Divide-and-Query(APD), which attempts a binary search on the execution tree.
- Top-down diagnosis (DED), which displays the trace in breadth-first although the execution is depth-first.

2.5 Spectrum-based debugging

Monitoring the instructions in a particular execution tree is a component of spectrum-based debugging, also referred to as spectrum-based fault location (SFL). The program spectrum is used to locate the active portions of the program runtime to achieve this.
A program spectrum is a group of runtime statistics that offers a picture of a program's changing behavior. It includes a few flags that are related to various programmatic components. Block hits/misses and function hits/misses are the two categories into which program profiles are divided.
The precise line of code that runs in response to a particular or abstract input can be located using these spectrums.

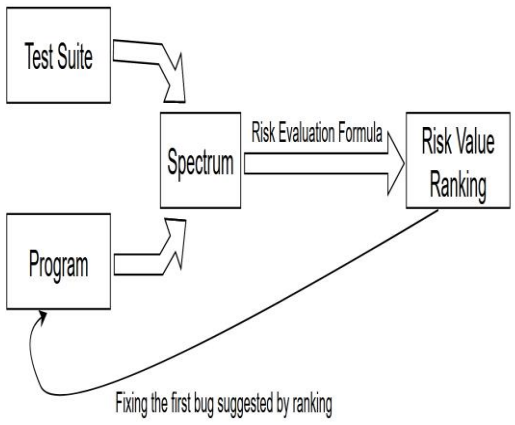

Fig. 3. Spectrum Based Debugging

2.6 Fault Localization

The process of pinpointing the exact locations of program flaws is known as fault localization. It takes a long time and costs a lot of money. Its effectiveness depends on developers' understanding of the program being debugged, their ability of logical judgment, past experience in program debugging, and how suspicious code, in terms of its likelihood of containing faults, is identified and prioritized for an examination of possible fault locations.

Research on fault localization has been ongoing, and as a result, several tools, including Tarantula and GZOLTAR, have been developed to address the initial stages of fault localization. The tools described here are based on the statistical debugging method known as "spectrum-based fault localization" (SFL), which uses code coverage data.
Users of other debugging tools, like dbx and the Microsoft VC++ debugger, can set breakpoints along the execution of a program and inspect variable values and internal states at each breakpoint. These tools offer a snapshot of the program's state at different execution-path breakpoints. dbx is a command-line debugging tool that is interactive and source-level.
This method's primary drawback is that users must create their own strategies to avoid going through excessive amounts of information for nothing. Another significant drawback is that it is unable to narrow the search domain by giving priority to code based on how likely it is to contain errors along a particular execution path.

2.7 GZOLTAR

GZoltar is an Eclipse plug-in that uses cutting-edge spectrum-based fault localization algorithms to produce precise fault localization information and provides the most recent research on regression testing. Additionally, it produces simple and interactive diagnostic report visualizations like Sunburst and Treemap.
Eclipse integration is incredibly helpful. GZoltar constructs the System Under Test (SUT) structure for the visualization view using Eclipse's standard features, such as detecting open projects in the workspace and their classes. To make the debugging process easier, GZoltar also seamlessly integrates the code editor and the standard Eclipse warnings generation with the offered visual diagnostic reports.

3. CONCLUSION

With this paper, we have attempted to present the current state of automated debugging in the industry, as well as the approaches and tools that are available, as well as the gaps in those tools, and future study and work that needs to be carried out. Debugging is crucial for any form of software, especially in safety-sensitive systems, therefore developers would benefit from having additional alternatives to investigate this topic. The goal of automated debugging is to make it easier to locate the source of a failure. As most software development organizations spend a significant amount of time and money on testing and debugging, automated debugging might save them both time and money. Returning to the original question, "Is Automated Debugging Still a Dream?" No, as demonstrated by the development of technologies like Bigshift, SPIN, Path-Based Weakest Preconditions, Language Consistency Checking, Plan Recognition, and others, automated debugging is not only possible but also potentially quite useful and valuable when used efficiently. Even if there is a shortage of understanding in this area, the growing need for error-free software will make it simpler for organizations to reduce uncertainty and provide successful outcomes.

3. FUTURE WORK

In this part, we discuss future research prospects.
Extending the period of a systematic literature review to obtain additional data on a relevant issue might help to improve the study.
Without a doubt, present debugging approaches can meet the needs of consumers, but they have some shortcomings that may be addressed to improve the user experience. As this study attempts to determine the possibility of automated debugging, progress in the categories below would be extremely beneficial.

- **Windows Integration**: The vast majority of currently available debugging approaches, such as GZoltar, Eclipse plugins, and expanded Delta Debugging variants, are designed to operate with the Unix operating system.The growing popularity of the Windows operating system demands the availability of tools for that operating system.

- **Integration of testing tools with IDEs**: While Eclipse supports some Delta Debugging plug-ins, it is only available on unix platforms.VS Code, for example, is a well-known IDE.Integrating debugging tools / plug-ins with IDEs such as VS Code and Atom would help users to save time while also eliminating ambiguity.

- **Improving from previous test results**: Learning from prior test results, whether positive or negative, may be a highly useful metric for future testing since the tool will gather experience from previous tests.

REFERENCES

1. Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? In Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11). Association for Computing Machinery, New York, NY, USA, 199–209. https://doi.org/10.1145/2001420.2001445.

2. J. Rößler, "How helpful are automated debugging tools?," 2012 First International Workshop on User Evaluation for Software Engineering Researchers (USER), 2012, pp. 13-16, doi: 10.1109/USER.2012.6226573.

3. M.Decasse and A. . -M. Emde, "A review of automated debugging systems: knowledge, strategies and techniques," Proceedings. [1989] 11th International Conference on Software Engineering, 1988, pp. 162-171, doi: 10.1109/ICSE.1988.93698.

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. 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.

7. WAMBUGU, Geoffrey Mariga; NJERU, Kevin Mwiti. The Automatic Debugging approaches. International Journal of Applied Computer Science (IJACS), [S.l.], v. 1, n. 1, p. 1-5, sep. 2017. ISSN 2522-6258.

8. 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.