Automatic Debugging Approaches: A literature Review.

Geoffrey Mariga Wambugu
gwmariga@yahoo.com; gmariga@mut.ac.ke
Department of Information Technology
Murang'a University of Technology, Murang'a Kenya

Kevin Mwiti Njeru
njerukevin@gmail.com;
Department of Information Technology
Murang'a University of Technology, Murang'a Kenya

Abstract
Fixing failed computer programs involves completing two fundamental debugging tasks: first, the programmer has to reproduce the failure; second, s/he has to find the failure cause. Software debugging is the process of locating and correcting erroneous statements in a faulty program as a result of testing. It is extremely time consuming and very expensive. The term debugging collectively refers to fault localization, understanding and correction. Automated tools to locate and correct the erroneous statements in a program can significantly reduce the cost of software development and improve the overall quality of the software. This paper discusses fault localization, program slicing and delta debugging techniques. It identifies statistical fault localization tools such as Tarantula, GZoltar and others such as dbx and Microsoft Visual C++ debugger that provides a snapshot of the program state at various break points along an execution path. In conclusion we note that most software development companies spend a huge amount of resources in testing and debugging. A lot more research need to be conducted to fully automate the debugging process thereby reducing software production cost, time and improve quality.

Keywords
Automated debugging; Fault localization; Program Slicing; Execution Synthesis

1. Introduction

When a computer program fails, a programmer must debug the program to fix the problem. This is done by completing two fundamental debugging tasks: first, the programmer has to reproduce the failure; second, s/he has to find the failure cause. Both tasks can result in a tedious, long-lasting, and boring work on the one hand, and can be a factor that significantly drives up costs and risks on the other hand. The field of automated debugging aims to ease the search for failure causes.

Software defects, commonly known as bugs, present a serious challenge for system reliability and dependability. Once a program failure is observed, the debugging activities to locate the defects are typically nontrivial and time consuming. Software debugging is the process of locating and correcting erroneous statements in a faulty computer program. Debugging a program consists primarily of stopping your program under certain conditions and then

examining the state of the program stack and the values stored in program variables. You stop execution of your program by setting breakpoints in your program. Breakpoints can be unconditional, in which case they always stop your program when encountered, or conditional, in which case they stop your program only if a test condition that you specify is true.

Parnin & Orso (2011) identifies three activities that must be performed when a software failure occurs. (1) Fault localization which consists of identifying the program statement(s) responsible for the failure, (2) Fault understanding that involves considering the root cause of the failure and (3) Fault correction to determine how to modify the code in order to remove such root cause. The term debugging collectively refers to fault localization, understanding, and correction. According to He & Gupta (2004), debugging is an expensive and challenging activity requiring understanding of the program and is often done manually by the programmers. Automated tools to locate and correct the erroneous statements in a program can significantly reduce the cost of software development.

Debugging software deployed in the real world is hard, frustrating, and typically requires deep knowledge of the code (Zamfir, et al 2013). Researchers have therefore invested a considerable amount of effort in developing automated techniques and tools for supporting various debugging tasks (Parnin & Orso, 2011). Most existing automated debugging techniques just focus on selecting a set of suspicious statements that may cause failures and ranking them in terms of suspiciousness (Lei, et.al 2012). This paper identifies the following tools from literature and discusses each one of them, giving a state of the art in automated debugging in the conclusion.

2. Fault localization

Fault localization is the activity of identifying the exact locations of program faults (Wong & Debroy, 2009). It is a very expensive and time consuming process. 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.
Software fault localization is one of the most expensive, tedious and time consuming activities in program debugging (Wong & Debroy, 2009). Therefore, there is a high demand for automatic fault localization techniques that can guide programmers to the locations of faults, with minimal human intervention. Fault localization can be divided into two major phases. The first part is to use a method to identify suspicious code that may contain program bugs. The second part is for programmers to actually examine the identified code to decide whether it indeed contains bugs.

Fault localization has been an active area of research, leading to the creation of several tools, such as Tarantula and GZOLTAR to address the first phase of fault localization. Spectrum-based Fault Localization (SFL), the technique behind the outlined tools, is a statistical debugging technique that relies on code coverage information.

Other debugging tools such as dbx and Microsoft VC++ debugger allow users to set break points along a program execution and examine values of variables as well as internal states at each break point. These tools provide a snapshot of the program state at various break points along an execution path. dbx is an interactive, source-level, command-line debugging tool. You can use it to run a C or C++, program in a controlled manner and to inspect the state of a stopped program.

dbx gives you complete control of the dynamic execution of a program, including collecting performance and memory usage data, monitoring memory access, and detecting memory leaks. dbx enables you to: Examine a core file from a program that has crashed; Set breakpoints; Step through your program; Examine the call stack; Evaluate variables and expressions; Use runtime checking to find memory access problems and memory leaks and Use fix-and-continue to modify and recompile a source file and continue executing without rebuilding the entire program. dbx debugger can be used on the command line, graphically through the Oracle Solaris Studio IDE, or through a separate graphical interface called dbxtool. Microsoft Visual C++ is a Graphical User Interface (GUI) debugger that allows interactive debugging from within the Integrated Development Environment (IDE) through the editor window.

One major disadvantage of this approach is that it requires users to develop their own strategies to avoid examining too much information for nothing. Another significant disadvantage is that it cannot reduce the search domain by prioritizing code based on the likelihood of containing faults on a given execution path.

3. Program Slicing

Program slicing is a technique that focuses on those parts of a program that could have contributed to the failure. This approach yields a subset of the program execution—called program slice that is relevant for a specific state or behavior. Slices are based on dependencies between statements: A statement S2 depends on a statement S1, if S1 can influence the program state accessed by S2. Starting from a statement, the transitive closure over all dependencies forms a program slice. In debugging, computing the backward slice for a failing statement returns all statements that could have influenced the failure. An important distinction is made between static and dynamic slicing. While a static slice applies to all possible runs, and therefore is computed without making any assumptions about a concrete (failing) program run, a dynamic slice just applies to the failing run and thus is more precise. An example of program slicing tool is CodeSurfer which is a commercial tool for performing static slicing on C programs. CodeSurfer is a code-understanding tool for C and C++ source code. CodeSurfer performs a deep semantic analysis of a program and provides sophisticated queries for understanding your code. It enables you to automatically identify and navigate the deep structure of your program: the semantic threads that reveal exactly how your program works. CodeSurfer can be used either interactively or programmatically. Another tool is Indus. At present, there are 3 modules that are part of Indus. (1) **Indus** is a module that houses the implementation pertaining to algorithms and data structures common to analyses and transformations that are part of or are planned to be part of Indus. This module contains interface definition common to most analyses and transformations to provide a framework in which various implementations of analyses/transformations can be combined to form systems with ease. (2) **StaticAnalyses** module intended to be the collection of static analyses such as object-flow analysis, escape analysis, and dependence analyses. The analyses in this module use common interfaces and implementations from *Indus* and may define/provide new interfaces/implementations specific to new analyses. Existing analyses are: Object-flow Analysis (OFA) which is a points-to analysis for Java; Escape Analysis which is an extended implementation of the escape analysis; a collection of dependence analyses: entry-based control, exit-based control, identifier-based data, reference-based data, interference, ready, synchronization, and divergence, required by analyses/transformations such program slicing and partial evaluation; Side-Effect Analysis which provides method-level side-

effect information; Monitor Anlaysis is a simple analysis that provides monitor/lock graph information for the given system; Safe Lock Analysis is an analysis that conservatively discovers if a lock (monitors) will not be held indefinitely and Atomicity Analysis which provides information about atomicity in the given system. (3) **Java Program Slicer** module contains the core implementation of Java program slicer along with adapters that deliver the slicer in other applications such as Bandera and Eclipse.

4. Delta debugging

Each bug in the database describes complex scenarios which cause software to fail. They may contain a lot of irrelevant information therefore a great deal of the bug reports could be equivalent. Delta debugging is an automated technique which takes a test case that causes a bug and turns the bug reports into minimal test cases where every part of the input would be significant in reproducing the failure thereby producing simplified bug reports (Zeller & Hildebrandt, 2000). It is usually hard to figure out what the real cause of the failure is by just inspecting an output file. Simplifying the input file and still generate the same failure would be very helpful in finding the error. Delta debugging technique automates this approach of repeated trials for reducing the input. To describe the algorithm we first need to define the process.

In general the delta debugging technique deals with circumstances whose change may cause a different program behavior. These changeable circumstances refers to all the possible behaviors of the program and its environment. Other applications of Delta debugging is in finding failure inducing code changes in programs. Given two versions of a program such that one works correctly and the other one fails, delta debugging algorithm can be used to look for changes which are responsible for introducing the failure. Choi & Zeller, 2002 notes that delta debugging can also be applied in isolating failure inducing thread schedules. Given a thread schedule for which a concurrent program works and another for which the program fails, delta debugging algorithm can narrow down the differences between two thread schedules and find the locations where a thread switch causes the program to fail.

5. Execution Synthesis

According to Zamfir, & Candea, 2010, debugging real systems is hard, requires deep knowledge of the code, and is time-consuming. Bug reports rarely provide sufficient information, thus developers are forced to search for an explanation of how the program could have arrived at the reported failure point. Execution synthesis is a technique for automating this investigative work: given a program and a bug report, it automatically produces an execution of the program that leads to the reported bug symptoms. Using a combination of static analysis and symbolic execution, it "synthesizes" a thread schedule and various required program inputs that cause the bug to manifest. The synthesized execution can be played back deterministically in a regular debugger, like GDB which is the GNU DeBugger. It is used to debug code that has been compiled by GCC (the GNU Compiler Collection). It's a very powerful debugger that allows you to debug even the most sophisticated of software. This is particularly useful in debugging concurrency bugs. Our technique requires no runtime tracing or program modifications, thus incurring no runtime overhead and being practical for use in production systems. We evaluate ESD — a debugger based on execution synthesis—on popular software (e.g.,the SQLite database, ghttpdWeb server, HawkNL network library, UNIX utilities): starting from mere bug

re-ports, ESD reproduces on its own several real concurrency and memory safety bugs in less than three minutes.

6. Conclusion

This paper identifies debugging as tedious, time consuming, boring and very expensive. It notes that despite all efforts and the scientific progress made, modern software still contains bugs that not only cause pure inconveniences, but also have a negative impact on the economy. Thus, we still need techniques that help us debug software systems. Both debugging techniques of reproducing the failure and finding the defect can be a tough and risky challenge. Automated debugging aims to ease the search for failure causes. Most software development companies spend a huge amount of resources in testing and debugging. A lot more research need to be conducted to fully automate the debugging process thereby reducing software production cost, time and improve quality.

References

Brummayer, R. Lonsing, F. & Biere A. 2010, 'Theory and Applications of Satisfiability Testing SAT 2010' *Proceedings 13th International Conference, SAT 2010 Edinburgh, UK, July 11-14, 2010*

Choi, J.D. & Zeller, A. 2002, 'Isolating Failure-Inducing Thread Schedules.' *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, July 2002

He, H. & Gupta, N. 2004, Automated Debugging Using Path-Based Weakest Preconditions

Lei,Y. Wang, C Mao, X & Wu, Q 2012, ' Enhancing Contexts for Automated Debugging Techniques', *Proceedings of the Seventh International Conference on Software Engineering Advances*

Bandara, M.L. 2002, 'A guide to GDB', available at < http://www.lashi.org/ writing/guide_to_gdb_1.1.pdf> accessed on 28th November 2014

Parnin, C. Orso, A. 2011, 'Are Automated Debugging Techniques Actually Helping Programmers?' ACM 978-1-4503-0562

Wong, W.E. & Debroy, V. 2009, 'A Survey of Software Fault Localization', Technical Report UTDCS-45-09 < Available at http://www.utdallas.edu/~ewong/fault-localization-survey.pdf accessed on 19th Nov. 2014 >

Zamfir, C. & Candea, G. 2010, 'Execution Synthesis: A Technique for Automated Software Debugging' ACM 978-1-60558-577-2/10/0

Zamfir, C. Kasikci, B. Kinder, J. Edouard Bugnion, E. & Candea, G. 2013 'Automated Debugging for Arbitrarily Long Executions', *Proceedings of 14th Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Ana Pueblo, NM, May 2013.

Zeller, A. & Hildebrandt, R. 2000, 'Simplifying and Isolating Failure-Inducing Input', *IEEE Transactions on Software Engineering* 28(2), February 2002, pp. 183-200.

< http://www.grammatech.com/research/technologies/codesurfer Accessed on 30th Nov. 2014>

< http://indus.projects.cis.ksu.edu/ Accessed on 30th Nov. 2014>