VOL. 15 NO. 9
ISSN: 2278-8881
SEPTEMBER, 2019

African Scholar Publications & Research International

African Scholars Journal of pure and Applied Science (JPAS-9)

Automated Debugging System

Anthony Umukoro & Joseph Okorodudu
Computer Science Department, Delta State Polytechnic, Otefe

Abstract

This research work is centered on the appraisal of the concept of Automated Debugging System. Computer programs often contain defects, or bugs, that need to be found and repaired. This manual "debugging" usually requires valuable time and resources. To help developers debug more efficiently, automated debugging solutions have been proposed. One approach goes through information available in bug reports. Another goes through information collected by running a set of test cases. Debugging ranges in complexity from fixing simple errors to performing lengthy and tiresome tasks of data collection, analysis, and scheduling updates. The debugging skill of the programmer can be a major factor in the ability to debug a problem hence this research work raises awareness for this topic and gives an overview of automated debugging system which are typically associated with the use of dedicated software technologies in the digital world.

Keywords: *Computer Program, Debugging, System, Data Collection, Programmer*

Introduction

Computer programs often contain defects, or bugs, that need to be found and repaired. This manual "debugging" usually requires valuable time and resources. To help developers debug more efficiently, automated debugging solutions have been proposed. One approach goes through information available in bug reports. Another goes through information collected by running a set of test cases. Until now, explains David Lo from Singapore Management University's (SMU)

School of Information Systems, there has been a "missing link" that prevents these information gathering threads from being combined.

Lim, et al (2013), together with colleagues from SMU, has developed an automated debugging approach called Adaptive Multimodal Bug Localisation (AML). AML gleans debugging hints from both bug reports and test cases, and then performs a statistical analysis to pinpoint program elements that are likely to contain bugs (Long et al, 2015).

According to Long et al, (2014) debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system.

Debugging tactics can involve interactive debugging, control flow analysis, unit testing, integration testing, log file analysis, monitoring at the application or system level, memory dumps, and profiling.

According to Mao, et al (2014) Automated debugging system is the automatic repair of software bugs without the intervention of a human programmer. It is also commonly referred to as automatic patch generation, automatic bug repair, or automatic program repair. The typical goal of such techniques is to automatically generate correct patches to eliminate bugs in software programs without causing software regression

BACKGROUND OF THE STUDY

Debugging ranges in complexity from fixing simple errors to performing lengthy and tiresome tasks of data collection, analysis, and scheduling updates. The debugging skill of the programmer can be a major factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming language(s) used and the available tools, such as debuggers (Mechtaev, et al (2016). Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, restart it, set breakpoints, and change values in memory. The term debugger can also refer to the person who is doing the debugging.

Generally, high-level programming languages, such as Java, make debugging easier, because they have features such as exception handling and type checking that make real sources of erratic behaviour easier to spot. In programming languages such as C or assembly, bugs may cause silent problems such as memory corruption, and it is often difficult to see where the initial problem happened. In those cases, memory debugger tools may be needed (Nguyen et al (2013).

In certain situations, general purpose software tools that are language specific in nature can be very useful. These take the form of static code analysis tools. These tools look for a very specific set of known problems, some common and some rare, within the source code. All such issues detected by these tools would rarely be picked up by a compiler or interpreter, thus they are not syntax checkers, but more semantic checkers. Some tools claim to be able to detect 300+ unique problems. Both commercial and free tools exist in various languages. These tools can be extremely useful when checking very large source trees, where it is impractical to do code walkthroughs. A typical example of a problem detected would be a variable dereference that occurs before the variable is assigned a value. As another example, some such tools perform strong type checking when the language does not require it. Thus, they are better at locating likely errors, versus actual errors. As a result, these tools have a reputation of false positives. The old Unix lint program is an early example (Pei et al, 2014).

LITERATURE REVIEW

The terms "bug" and "debugging" are popularly attributed to Admiral Grace Hopper in the 1940s. While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. However, the term "bug", in the sense of "technical error", dates back at least to 1878 and Thomas Edison (see software bug for a full discussion). Similarly, the term "debugging" seems to have been used as a term in aeronautics before entering the world of computers. Indeed, in an interview Grace Hopper remarked that she was not coining the term (Smith et al, 2015). The moth fit the already existing terminology, so it was saved. A letter from J. Robert Oppenheimer (director of the WWII atomic bomb "Manhattan" project at Los Alamos, NM) used the term in a letter to Dr. Ernest Lawrence at UC Berkeley, dated October 27, 2010, regarding the recruitment of additional technical staff (Kim et al, 2013).

"While most past studies only demonstrate the applicability of similar solutions for small programs and 'artificial bugs' [bugs that are intentionally inserted into a program for testing purposes], our approach can automate the debugging process for many real bugs that impact large programs," Sidiroglou, et al (2015) explains. AML has been successfully evaluated on programs with more than 300,000 lines of code. By automatically identifying buggy code, developers can save time and redirect their debugging effort to designing new software features for clients.

Dewey-Vogt et al, (2012) are now planning to contact several industry partners to take AML one step closer toward integration as a software development tool. Dobolyi and Weimer, (2008) future plans involve developing an Internet-scale software analytics solution. This would involve analysing massive amounts of data that passively exist in countless repositories on the Internet in order to transform manual, pain-staking and error-prone software engineering tasks into automated activities that can be performed efficiently and reliably. This is done, says Gao, et al, (2015), by harvesting the wisdom of the masses – accumulated through years of effort by thousands of software developers – hidden in these passive, distributed and diversified data sources.

The Oxford English Dictionary entry for "debug" quotes the term "debugging" used in reference to airplane engine testing in a 1945 article in the Journal of the Royal Aeronautical Society. An article in "Airforce" (Holtschulte et al, 2015) also refers to debugging, this time of aircraft cameras. Hopper's bug was found on September 9, 1947. The term was not adopted by computer programmers until the early 1950s. The seminal article by Gill in 1951 is the earliest in-depth discussion of programming errors, but it does not use the term "bug" or "debugging". In the ACM's digital library, the term "debugging" is first used in three papers from 1952 ACM National Meetings. Two of the three use the term in quotation marks. By 1963 "debugging" was a common enough term to be mentioned in passing without explanation on page 1 of the CTSS manual.

DEBUGGING TECHNIQUES

Print debugging (or tracing): It is the act of watching (live or recorded) trace statements, or print statements, that indicate the flow of execution of a process. This is sometimes called print debugging, due to the use of the printf function in C. This kind of debugging was turned on by the command TRON in the original versions of the novice-oriented BASIC programming language. TRON stood for, "Trace On." TRON caused the line numbers of each BASIC command line to print as the program ran.

Remote Debugging: It is the process of debugging a program running on a system different from the debugger. To start remote debugging, a debugger connects to a remote system over a network. The debugger can then control the execution of the program on the remote system and retrieve information about its state.

Post-Mortem Debugging: is debugging of the program after it has already crashed. Related techniques often include various tracing techniques (for example,) and/or analysis of memory dump (or core dump) of the crashed process. The dump of the process could be obtained automatically by the system (for example, when process has terminated due to an unhandled exception), or by a programmer-inserted instruction, or manually by the interactive user.

Saff Squeeze – a technique of isolating failure within the test using progressive inlining of parts of the failing test.

DEBUGGING FOR EMBEDDED SYSTEMS

According to Tan and Roychoudhury (2015) in contrast to the general purpose computer software design environment, a primary characteristic of embedded environments is the sheer number of different platforms available to the developers (CPU architectures, vendors, operating systems and their variants). Embedded systems are, by definition, not general-purpose designs: they are typically developed for a single task (or small range of tasks), and the platform is chosen specifically to optimize that application. Not only does this fact make life tough for embedded system developers, it also makes debugging and testing of these systems harder as well, since different debugging tools are needed for different platforms.

Despite the challenge of heterogeneity mentioned above, some debuggers have been developed commercially as well as research prototypes. Examples of commercial solutions come from Green Hills Software and Microchip's MPLAB-ICD (for in-circuit debugger). Two examples of research prototype tools are Aveksha and Flocklab. They all leverage a functionality available on low-cost embedded processors, an On-Chip Debug Module (OCDM), whose signals are exposed through a standard JTAG interface. They are benchmarked based on how much change to the application is needed and the rate of events that they can keep up with.

In addition to the typical task of identifying bugs in the system, embedded system debugging also seeks to collect information about the operating states of the system that may then be used to analyze the system: to find ways to boost its performance or to optimize other important characteristics (e.g. energy consumption, reliability, real-time response etc.).

DISCUSSION

Normally the first step in debugging is to attempt to reproduce the problem. This can be a non-trivial task, for example as with parallel processes or some unusual software bugs. Also, specific user environment and usage history can make it difficult to reproduce the problem.

After the bug is reproduced, the input of the program may need to be simplified to make it easier to debug. For example, a bug in a compiler can make it crash when parsing some large source file. However, after simplification of the test case, only few lines from the original source file can be sufficient to reproduce the same crash. Such simplification can be made manually, using a divide-and-conquer approach. The programmer will try to remove some parts of original test case and check if the problem still exists. When debugging the problem in a GUI, the programmer can try to skip some user interaction from the original problem description and check if remaining actions are sufficient for bugs to appear.

After the test case is sufficiently simplified, a programmer can use a debugger tool to examine program states (values of variables, plus the call stack) and track down the origin of the problem(s). Alternatively, tracing can be used. In simple cases, tracing is just a few print statements, which output the values of variables at certain points of program execution.

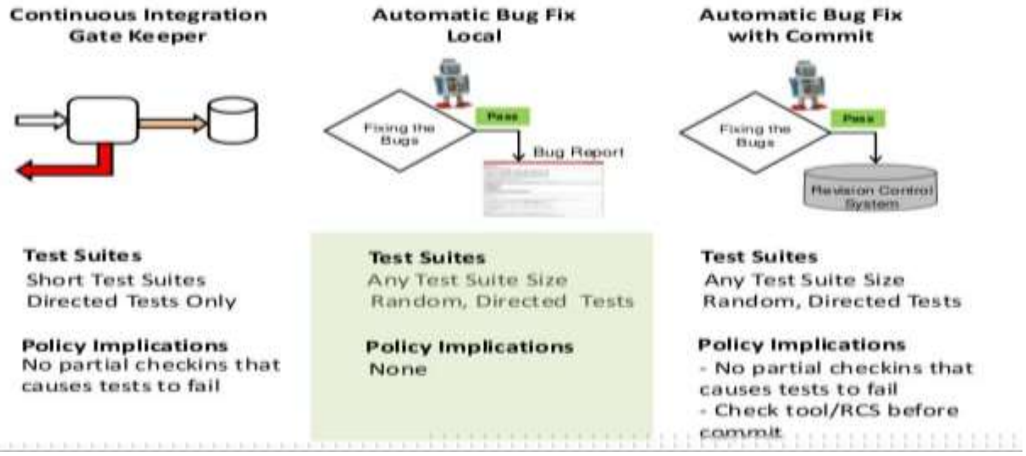

Fig. 1.1 Process of automated debugging system 1

Anti-debugging is "the implementation of one or more techniques within computer code that hinders attempts at reverse engineering or debugging a target process". It is actively used by recognized publishers in copy-protection schemas, but is also used by malware to complicate its detection and elimination. Techniques used in anti-debugging include:

1. API-based: check for the existence of a debugger using system information
2. Exception-based: check to see if exceptions are interfered with
3. Process and thread blocks: check whether process and thread blocks have been manipulated
4. Modified code: check for code modifications made by a debugger handling software breakpoints
5. Hardware- and register-based: check for hardware breakpoints and CPU registers
6. Timing and latency: check the time taken for the execution of instructions
7. Detecting and penalizing debugger.

An early example of anti-debugging existed in early versions of Microsoft Word which, if a debugger was detected, produced a message that said, "The tree of evil bears bitter fruit. Now trashing program disk" after which it caused the floppy disk drive to emit alarming noises with the intent of scaring the user away from attempting it again.

EXAMPLES OF DEBUGGERS

Some widely used debuggers are:

1. Eclipse debugger API used in a range of IDEs: Eclipse IDE (Java) Nodeclipse (JavaScript)
2. Firefox JavaScript debugger.
3. GDB - the GNU debugger.
4. LLDB.
5. Microsoft Visual Studio Debugger.
6. Radare2.
7. Valgrind.
8. WDW, the OpenWatcom debugger.

PATCH GENERATION

At the patch generation step, automatic bug-fixing techniques use one or more of the followings to generate a set of candidate patches.

Program Mutation

One way to generate candidate patches is to apply mutation operators on the original program. Mutation operators manipulate the original program, potentially via its abstract syntax tree representation, or a more coarse-grained representation such as operating at the statement-level or block-level. Earlier genetic improvement approaches operate at the statement level and carry out simple delete/replace operations such as deleting an existing statement or replacing an existing statement with another statement in the same source file. Recent machine learning and solver approaches use more fine-grained operators at the abstract syntax tree level to generate more diverse set of candidate patches.

Fix Template

Using fix templates is an alternative way to generate candidate patches. Fix templates are typically predefined program mutation rules for fixing specific classes of bugs. Examples of fix templates include inserting a conditional statement to check whether the value of a variable is null to fix null pointer exception, inserting memory deallocation statement to fix memory leaks, and changing an integer constant by one to fix off-by-one errors. Comparing to program mutation techniques, fix templates tend to achieve better candidate patches for bugs within its scope. Fix templates are therefore often adopted by targeted techniques.

Code Learning and Transfer

Machine learning techniques can improve the effectiveness of automatic bug-fixing systems. One example of such techniques learns from past successful patches from human developers collected from open source repositories in GitHub and SourceForge. It then use the learned information to recognize and prioritize potentially correct patches among all generated candidate patches. Alternatively, patches can be directly mined from existing sources. Example approaches include mining patches from donor applications or from QA web sites.

SMT Solver

Another way to generate candidate patches is to use solvers for satisfiability modulo theories (SMT). Solver-based techniques convert a program into SMT formulas. They then query SMT solvers to solve the converted SMT formulas to find candidate patches that allow the patched program to pass all supplied test cases. The benefit of this approach is that SMT solvers can quickly find patches passing test cases for small to medium-sized formulas. The limitation of this approach is that real world programs are often converted to intractably large formulas especially for modifying statements with side effects.

LIMITATIONS OF AUTOMATED DEBUGGING SYSTEM

Automated debugging systems that rely on a test suite do not provide patch correctness guarantees, because the test suite is incomplete and does not cover all cases. A weak test suite may cause generate-and-validate techniques to produce validated but incorrect patches that have negative effects such as eliminating desirable functionalities, causing memory leaks, and introducing security vulnerabilities. This weak test suite issue is alternatively called as "overfitting" of bug-fixing systems . For example, a recent study shows that for 52 out of 55 cases that the previous bug-fixing system GenProg reports to generate repairs, all of the generated patches are incorrect. Recent state-of-the-art systems with machine learning or SMT solver techniques are able to generate correct patches for much more cases on the same benchmark set, but there are still plenty of validated but incorrect patches generated by these systems.

Another limitation of generate-and-validate systems is the search space explosion. For a program, there are a large number of statements to change and for each statement there are a large number of possible modifications. State-of-the-art systems often explicitly or implicitly assume that a small modification is enough for fixing a bug. As a result, such systems can currently repair a small proportion of bugs in real world programs

SUMMARY, CONCLUSION AND RECOMMENDATION

SUMMARY

Automated debugging system is the automatic repair of software bugs without the intervention of a human programmer. It is also commonly referred to as automatic patch generation, automatic bug repair, or automatic program repair. The typical goal of such techniques is to automatically generate correct patches to eliminate bugs in software programs without causing software regression. Debuggers are software tools which enable the programmer to monitor the execution of a program, stop it, restart it, set breakpoints, and change values in memory. The term debugger can also refer to the person who is doing the debugging.

CONCLUSION

Computer programs often contain defects, or bugs, that need to be found and repaired. This manual "debugging" usually requires valuable time and resources. To help developers debug more efficiently, automated debugging solutions have been proposed. One approach goes through information available in bug reports. Another goes through information collected by running a set of test cases. Debugging ranges in complexity from fixing simple errors to performing lengthy and tiresome tasks of data collection, analysis, and scheduling updates. The debugging skill of the programmer can be a major factor in the ability to debug a problem, but the difficulty of software debugging varies greatly with the complexity of the system, and also depends, to some extent, on the programming language(s) used and the available tools, such as debuggers.

RECOMMENDATION

I recommend that all users should make use of automated debugging system as it will help to eliminate damage to the computer system hereby preventing the attack of worm, viruses.

REFERENCE

Dewey-Vogt, M.; Forrest, S.; Weimer, W. (2012). "A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for Each". Proceedings of the 2012 International Conference on Software

Engineering. ICSE 2012. IEEE Press. pp. 3–13. Retrieved 24th April, 2018

Dobolyi, K.; Weimer, W. (2008). "Changing Java's Semantics for Handling Null Pointer Exceptions". 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp 47–56. Retrieved 25th April, 2018

Gao, Q.; Zhang, H.; Wang, J.; Xiong, Y.; Zhang, L.; (2015). "Fixing Recurring Crash Bugs via Analyzing Q&A Sites". Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering. ASE 2015.

Holtschulte, N.; Smith, E.; Brun, Y.; Devanbu, P.; Forrest, S.; Weimer, W. (2015). "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs". IEEE Transactions of Software Engineering. IEEE. pp. 1236–1256. Retrieved 28th April, 2018.

Kim, D.; Nam, J.; Song, J.; Kim, S. (2013). "Automatic Patch Generation Learned from Human-written Patches". Proceedings of the 2013 International Conference on Software Engineering. ICSE '13'. IEEE Press. pp. 802–811.

Lim, R.; Ferrari, F.; Zimmerling, M.; Walser, C.; Sommer, P.; Beutel, J. (2013). "FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems". Proceedings of the 12th International Conference on Information Processing in Sensor Networks. IPSN '13. New York, NY, USA. Retrieved 28th April, 2018.

Long, F.; Achour, S.; Rinard, M. (2015). "An Anlysis of Patch Plausibility and Correctness for Generate-and-Validate Patch Generation Systems". Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2015. Retrieved 28th April, 2018.

Long, F.; Sidiroglou-Douskos, S.; Rinard, M. (2014). "Automatic Runtime Error Repair and Containment via Recovery Shepherding". Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '14'. New York, New York: Retrieved 25th April, 2018. Retrieved 28th April, 2018.

Mao, X.; Lei, Y.; Dai, Z.; Wang, C. (2014). "The Strength of Random Search on Automated Program Repair". Proceedings of the 36th International Conference on Software Engineering. ICSE 2014. Austin, Texas: ACM. pp. 254–265. Retrieved 28th April, 2018.

Mechtaev, S.; Yi, J.; Roychoudhury, A. (2016). "Angelix: scalable multiline program patch synthesis via symbolic analysis". Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, Texas, May 14-22, 2016. pp. 691–701. Retrieved 28th April, 2018.

Nguyen, H.D.; Roychoudhury, A.; Chandra, S. (2013). "SemFix: Program Repair via Semantic Analysis". Proceedings of the 2013 International Conference on Software Engineering. ICSE '13'. San Francisco, California: IEEE Press. pp. 772–781. Retrieved 28th April, 2018.

Pei, Y.; Furia, C.A.; Nordio, M.; Wei, Y.; Meyer, B.; Zeller, A. (2014). "Automated Fixing of Programs with Contracts". IEEE Trans. Softw. Eng. 40 (5): 427–449. Retrieved 28th April, 2018.

Smith, E.K.; Earl, T.; Brun, Y. (2015). "Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair". Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ESEC/FSE 2015. New York, New York: ACM. pp. 532–543. Retrieved 28th April, 2018.

Sidiroglou, S.; Lahtinen, E.; Long, F.; Rinard, M. (2015). "Automatic Error Elimination by Multi-Application Code Transfer". Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. Retrieved 28th April, 2018.

Tan, S.H.; Roychoudhury, A. (2015). "Relifix: Automated repair of software regressions". Proceedings of the 37th International Conference on Software Engineering-Volume 1. pp. 471–482. Retrieved 28th April, 2018.