Automated Fault Localization and Debugging

Ian Drosos
North Carolina State University
Raleigh, North Carolina
izdrosos@ncsu.edu

Aniket Ramesh Dhuri
North Carolina State University
Raleigh, North Carolina
adhuri@ncsu.edu

Karan Jadhav
North Carolina State University
Raleigh, North Carolina
kjadhav@ncsu.edu

ABSTRACT

Of all the phases in the software development life cycle, debugging takes the most time and effort. Researchers have been working on ways to lessen the impact of this phase through automated fault localization and debugging. Automated fault localization can help developers quickly find the likely location of a bug, rather than having developers search through thousands of lines of code. Automated debugging can offer potential solutions to a developer, saving time that would otherwise be used thinking up a solution. This paper is a survey of the progression of research in automated fault localization and debugging and offers ideas on the next steps for future work.

Keywords

Automated Fault Localization, Automated Debugging, Spectrum-based Fault Localization

1. INTRODUCTION

Fault localization and debugging is a costly phase of the software development lifecycle. The National Institute of Standards and Technology (NIST) lists the impact of inadequate software testing infrastructure on the US economy at $59.5 billion dollars [20]. A majority of these costs are from software users that have to avoid software errors or mitigate their effects. The rest of the costs are from additional testing resources required due to inadequate testing infrastructure, like the tools and methods used. NIST also estimates that there is a potential $22.2 billion dollar cost reduction through the improvement of software testing infrastructure. Many research efforts have been launched to improve testing infrastructure in order to solve this costly issue. Of interest in this paper, we look at research in automated fault localization and debugging. The goal of this research is to take faults discovered through failed test cases and locate them through a spectrum-based (or coverage-based) algorithm. This provides developers with a framework to quickly debug software by investigating ranked candidate fault locations rather than manually search for the location on their own. In this paper we present the procession of research in automated fault localization and debugging and their potential uses in improving software testing infrastructure.

Throughout this paper we will speak on research in frameworks and tools that support automated fault localization and debugging, the algorithms behind these frameworks, and the adoptability of the tools by professionals. We will touch on the contributions to the research area that each paper made and how their research can be improved. Finally, we will propose a path forward in automated fault localization and debugging research and what alternatives there are to automated fault localization.

ASE Fall 2016 NCSU

ACM ISBN none.

DOI: none

2. RELATED WORK

2.1 MINTS - A General Framework and Tool for Supporting Test-suite Minimization

MINTS is a general framework and tool that allows testers to encode a wide range of test-suite minimization problems, handle problems that have any number of criteria, and compute the optimal solution using modern integer linear programming solvers [11]. Test-suite minimization eliminates redundant test cases from a test-suite by using metrics like coverage. If 2 test cases have the same coverage, it may be possible to remove one of the test cases to lower testing overhead. MINTS, or MINimizer for Test Suites, can plug in with any binary ILP solver that uses the Pseudo Boolean Evaluation 2007 format. The authors performed an empirical evaluation to gauge the effectiveness of MINTS using six different ILP solvers on several programs like tcas, LogicBlox, and Eclipse.

2.1.1 Keywords

- **Regression test suite:** a test suite used in regression testing software. Regression testing is done after software has had some changes made to it, like new functions, to make sure the software remains accurate and correct.
- **Binary ILP problem:** or pseudo-Boolean problems. Binary ILP problems deal with the optimization of a linear function where all unknown variables have values 0 or 1, all while satisfying a set of linear equality/inequality constraints. Used by MINTS to represent test-suite minimization problems.

- **Test-suite minimization:** techniques that eliminate redundant test cases from a test-suite. These techniques use criteria such as coverage and fault-detection capability to detect what tests that can be minimized. MINTS is a test-suite minimization framework that aims to find the optimum solution.

2.1.2 Study Instruments

With respect to an empirical evaluation the authors asked 3 research questions which could also be asked of other test-suite minimization heuristics and tools seeking to compare results and gauge their effectiveness: (1) How often can MINTS find an optimal solution for a test-suite minimization problem in a reasonable time? (2) How does the performance of MINTS compare with the performance of a heuristic approach? (3) To what extent does the use of a specific solver affect the performance of the approach? The authors also used the Siemens suite of programs and test cases to provide a standard benchmark that other frameworks can use to compare their results against MINTS.

2.1.3 Motivation

Regression test-suites must be maintained for software as the software is changed. The problem is that test suites grow too large and eventually cannot be run as a whole over time. A large test suite can make regression testing impractical, especially when manual work is involved in the testing. MINTS is a framework that aims to minimize test-suites to prevent this problem. MINTS allows for a wide range of these test-suite minimization problems and allows for testers to encode all the constraints they believe relevant to the test cases. MINTS then produces solutions for these test-suite minimization problems that take in the constraints thought relevant by the tester that are optimal to the constraints specified.

2.1.4 Related Work

Chavatal presents a greedy heuristic that selects a test case with the most coverage of requirements that still need to be covered, and does this until all requirements are covered. A similar heuristic was offered by Harrold that improved on Chavatal's. Agrawal offered an approach that exploited implications among coverage requirements. Tallam and Gupta combined these approaches into Delay-Greedy. All of these approaches only focus on a single criterion and only give approximate solutions. MINTS extends and generalizes these techniques but also can compute optimal solutions with specified prioritized criteria.

2.1.5 Future Work and Conclusion

The authors of MINTS want to perform additional empirical studies with more subjects to see if the MINTS approach is scalable. The authors would like to collect larger programs that have test cases and test data and then replicate their experiments on these programs. They would also like to analyze their results to get an explanation to why different solvers had different performance. Lastly the authors mentioned extending their approach to test-case prioritization instead of just minimization. They would like to do this because certain ordering of test cases can maximize the likelihood of finding a fault earlier.

The empirical evaluation answered each of the 3 research questions. For research question 1, MINTS was able to find an optimal solution for every minimization problem it was presented in at most 40 seconds. More impressive was that most of the solutions were found in under 2 seconds. For the most complex problem, minimization of test cases for Eclipse, MINTS found optimal solutions for most of the minimization problems in less than 10 seconds. These first results are promising, but we agree with the author's' assessment that MINTS needs to increase the breadth of the programs in their evaluation. For research question 2, MINTS was able to provide greater minimization of test suites than the heuristic based approach HGS. That said, MINTS did not improve upon the performance for the Siemens set of programs, equalling the performance of HGS. For research question 3, CPLEX and GLPPB performed the best among solvers used by MINTS. This means that instead of plugging in multiple solvers, MINTS can just use the best performing ones which saves time.

Though MINTS is one of the oldest systems we looked at, it provided a good base for other automated fault localization and debugging techniques. MINTS does not find faults but instead prunes test cases that are unnecessary. This saves time by not running test cases that will already pass since another test case with the same coverage has already ran. MINTS' ability to plug in any binary ILP solver is an important feature for future frameworks, as better algorithms are found MINTS can still be used by software engineers to perform test case minimization.

2.2 Spectrum-based fault localization

Spectrum-based fault localization, or SFL, is a lightweight automated diagnosis technique that shortens the test, diagnose, repair cycle by reducing effort to debug software [1]. SFL can be integrated into testing schemes to discover statistical coincidences between the failure in code and the activity going on in the code during the failure. The authors used the Siemens set of programs and the space program as study instruments. This allows for empirical evaluations of SFL with any other algorithm for fault localization. Further, the authors also evaluated SFL's usage against real-world industry case studies to show its usefulness outside the lab.

2.2.1 Keywords

- **Test data analysis:** the inspecting, cleaning, transforming, and modeling of test data so useful conclusions can be made. This test data is generated from test cases being executed by a system. The output is then analyzed for the purpose of finding failures in the system.

- **Software fault diagnosis:** finding the causes of faults in software. The goal is to monitor the software and detect when a fault occurs. When a fault is detected its cause and the location of that cause is found, diagnosing the fault in the software.

- **Program spectra:** a distribution of an aspect of a program's run-time behavior. This could be the distribution of all the different paths through the code each execution takes.

- **Real-time and embedded systems:** a real-time system is a system subject to real-time constraints.

This system must respond to an event within a deadline specified for the system and do so correctly. Embedded systems usually have real-time constraints and are embedded in a device in which the embedded system provides a dedicated and specific function.

2.2.2 Study Instruments

The paper uses a common benchmark (the Siemens set used by MINTS and the space program) which is easily available for others to run their tools to compare their results to the results of spectrum-based fault localization. The Siemens set is a collection of benchmark faults in several C programs and the space program is a large program with several faults. These sets of programs and the results of SFL against these programs is presented in the paper. Fault diagnosis rates of around 80% are given, which means only 20% of the code needs to be manually searched for the fault.

2.2.3 Motivation

As mentioned, the testing and debugging process is a major cost in the software development cycle. The reason debugging is such a major cost is that diagnosing faults is a very labor-intensive and manual task. The authors posit spectrum-based fault localization as a way to more efficiently diagnose faults through automation. SFL can reduce fault localization effort which then reduces the cost of the testing and debugging process.

2.2.4 Related Work

Program spectra were introduced in 2000 to diagnose bugs due to the year 2000. Several practical debugging tools like Pinpoint were mentioned that are based on SFL. The Tarantula tool (also mentioned in the GZoltar and VIDA papers) works with statement hit spectra [14] and is compared to SFL results (and is seen as a variant of SFL). Finally white box and black box diagnosis techniques were discussed. SFL lends itself to a black box classification, similar to Nearest Neighbor and dynamic program slicing.

2.2.5 Future Work and Conclusion

The authors plan to study the "influence of the granularity (statement, function level) of a program spectra on the diagnostic accuracy of spectrum-based fault localization". That is, the distribution of aspects of the program's run-time behavior. The authors would also like to investigate improving accuracy of fault localization through integrating static and dynamic program slicing. Finally the authors would like to extend SFL to multiple-fault cases, rather than just single faults.

Spectrum-based fault localization is a valuable technique for automated fault localization. SFL offers a good accuracy of diagnosis of around 80%. This leaves only 20% of code that needs to be manually searched by a developer to find a fault. This is an accuracy acceptable by most practitioners when evaluating the adoptability of the research tools. Adding more failed test cases to the evaluation by SFL can improve accuracy, but adding passing test cases can degrade performance. This means SFL requires test suite minimization to prevent duplicate passing test cases from lowering the accuracy of the fault localization effort. The authors also used the Siemens set of programs which many tools and techniques use. This allows researchers to compare their new work with SFL to see if any improvements have been made. Some tools (GZoltar and VIDA) do not provide the program set they used so no useful comparison can be done. Current issues with SFL include not being able to properly handle multiple-fault cases. Most large systems will have more than one fault, so SFL must be able to handle this situation. The authors say that SFL can extend to multiple-fault cases by applying the technique over and over until all faults are repaired one by one, but this is inefficient. SFL has shown through industry case studies of embedded software systems that it can work in the real-world and can scale to large software systems. This is necessary for the adoption of SFL by software practitioners working on these complex systems.

2.3 VIDA - Visual Interactive Debugging

VIDA is a visual interactive debugging tool. It is integrated with the Eclipse IDE to assist a programmer in debugging their code. It assists programmers by recommending breakpoint locations via an outline that visualizes the code and producing a static dependency graph [5]. VIDA aims to provide this assistance through the same debugging flow a programmer usually goes through during debugging. The authors believe that tools that require a separate flow than the programmer's usually flow will not be adopted by programmers. VIDA also adapts its recommendations based on feedback by the programmer, potentially improving recommendations to the programmer. This assumes the programmer is correct in their assumption though, a potential pitfall if the programmer is confident in the incorrect estimation of where a breakpoint should go.

2.3.1 Keywords

- **Static dependency graph:** a directed graph that represents dependencies of some number of objects towards each other. For example, a class X that imports the string class and uses string class functions within itself can be said to be dependent on the string class. A static graph shows every possible run of the program, rather than just one execution.
- **Fault localization:** the process of finding the location of faults (bugs) by identifying abnormal behaviors and the root cause of such behaviors.
- **Breakpoint(s):** signals to the debugger that tell it to pause execution of the program. Programmers can attach breakpoints to lines in their code to determine where they would like these pauses to happen. A paused program can then be stepped through line-by-line or resumed, which will cause another pause if the resumed execution hits another breakpoint.

2.3.2 Study Instruments

This paper does not provide any potential programs to empirically evaluate VIDA. The difficulty with VIDA is that it's results are mostly objective. One developer might find the breakpoints recommended by VIDA to be useful and something they would not have chosen, but a different developer might disagree with where VIDA recommends a breakpoint. One way to gauge VIDA's effectiveness would be to use a set of programs with known defects and measure VIDA's accuracy by taking the breakpoints recommended and seeing if they cover the line of code that the defect is on.

2.3.3 Motivation

Software debugging costs more than 50% of software development and maintenance budget, so providing a tool to ease this process should lower the effects of debugging on the software development process. Fault localization is seen as the more time consuming process of debugging so the VIDA focuses finding bugs. It is expected that VIDA will help a programmer during the debugging process by providing candidate breakpoints, providing a global view (static dependency graph) of the program, and adjusting/improving these recommendations and views based on programmer feedback.

2.3.4 Related Work

Tarantula, a fault localization tool, visualizes statements with different colors [14]. A red statement in Tarantula means it is suspicious of the statement and could be where the fault is. The problem with Tarantula is that it is not in line with the usual debugging process a programmer follows. VIDA takes in feedback by the programmer, similar to algorithmic debugging, during conventional debugging to tailor results to the programmer.

2.3.5 Conclusion

VIDA recommends breakpoint candidates to programmers to ease the cost of debugging software. Using analysis of execution information and the user's own estimation of breakpoint correctness, VIDA provides a list of breakpoints to the programmer that are ordered by a suspicion rating. The more suspicious that VIDA is of a line of code, the higher on the list the breakpoint appears. VIDA adapting to input of the user is quite valuable. Until automated fault localization and debugging has a high enough accuracy for programmers to trust the tool (see section 2.8), it will not be adopted. VIDA allows the programmer to correct the suspicion level of breakpoints so that VIDA can provide the user with the best list of breakpoints. Further, the authors' understanding that the workflow of the tool must match that of the programmer is insightful. Most programmers are unwilling to change a flow they believe to be the best for them, so a tool that enhances their flow rather than changes it is preferred.

2.4 GZoltar

GZoltar is a plug-in for the Eclipse IDE that provides a framework for automated testing and debugging, specifically "(regression) test suite minimization and fault localization" [15]. GZoltar uses spectrum-based fault localization to do this. This tool can automatically analyze source code of programs to produce run-time data. GZoltar takes in a list of JUnit test cases and executes them. The results of the executed test cases, passing or failing, is saved and presented to the user. GZoltar then uses RZoltar to minimize the test suite by using a code coverage matrix. It then filters solutions and shows them to the user. The user can then choose to re-execute test cases or view various metrics about the tests that were ran. GZoltar provides the user a GUI for all of this functionality for ease of use.

2.4.1 Keywords

- **Eclipse plug-in:** a tool set that is embedded in the Eclipse IDE that provides developers with some set of functions, e.g. a git plug-in that allows pulling and pushing of code from repos while inside Eclipse.
- **Automatic testing:** the execution of tests, reporting of test outcomes, and comparison of test results with expected results done through automation using software.
- **Automatic Debugging:** the automation of the identification and removal of errors in software being tested and debugged.

2.4.2 Study Instruments

Unfortunately this paper did not provide any study instruments to evaluate the performance of GZoltar. This paper was more focused on providing information on the framework GZoltar provides for automated fault localization and debugging. Because of this we are unable to see if GZoltar actually aids developers in finding faults quicker than any other framework, or even manually. Theoretically any Java project open in the Eclipse IDE can be used, allowing GZoltar to use any study instruments used in the other papers, but the authors did not run this type of evaluation.

2.4.3 Motivation

Testing and debugging is the "most expensive, error-prone phase in the software development cycle". So software that can automate testing and debugging can lessen the impact of the testing phase of the development cycle, improving the overall quality of the software being tested. GZoltar is presented as a toolset that can automate finding faults in the code. There are other test suites that provide unit testing, but do not offer the same capabilities in test suite reduction and fault localization (where in the code the fault is likely to exist) as GZoltar does.

2.4.4 Related Work

Most IDEs have some form of debugging tool that allows for iterating through code to find errors (via breakpoints or line-by-line iteration) but are manual. Other automated works include Tarantula, but Tarantula does not integrate with an IDE and does not support unit tests as GZoltar does. Finally Zoltar provides automatic debugging but only runs on Linux and works only with projects in C.

2.4.5 Future Work and Conclusion

In the future the authors of GZoltar would like to provide more techniques that minimize test suites, like MINTS, and provide more diagnostic report visualizations. In order to reduce overhead from collecting information, they also plan to add dynamically instrumenting (monitoring and measuring performance) the source code as a capability of GZoltar.

GZoltar's implementation as a plug-in for the Eclipse IDE increases its adoptability by software engineering practitioners. Rather than being a standalone tool, developers simple need to download and install GZoltar into their IDE. As software developers we can certainly see the benefits of GZoltar's visualizations and functionality beyond the usual IDE debugging tools. These visualizations allow developers to quickly find failing test cases and GZoltar provides functionality to quickly go to the line of code that most likely caused the test case to fail, allowing quicker bug fixes. As mentioned in section 2.4.2, the authors of this paper did not present us with any empirical evaluation of GZoltar. Despite this, we can see GZoltar's potential as a framework for test suite minimization and automated fault localization.

2.5 Test case purification for improving fault localization

Xuan and Monperrus present us with a modification of spectrum-based fault localization, spectrum-based purifying of test cases to improve fault localization [21]. The goal of this purification is to take test cases and divide them into smaller, purified, test cases. These purified test cases can be used with test oracles to localize faults. Using the purified test cases with existing fault localizers, like Tarantula, results in a better ranking for the program statements. Meaning that purified test cases will give more accurate locations of the faults.

A purified test case is a test case with only one assertion, rather than the potentially many assertions of non-purified test cases. To handle the purifying of test cases the authors present a framework with three steps: test case atomization, test case slicing, and refining ranks. First, a failing test case goes through test case atomization. Only one of the original assertions in the failing test case are kept. Then the atomized test case goes through slicing where irrelevant statements in the test case are removed and then a new test case generated, the purified test case. Finally, using this new purified test case the statements are re-ranked via existing fault localization tools.

2.5.1 Keywords

- **Test case purification:** the act of creating a short test case with only one assertion. These are generated through the removal of several statements from a failing test case. The purpose of this is to improve existing techniques on fault localization.
- **Assertion:** a binary expression that indicates the expected behaviors of a program. An exception is thrown when an assertion is not met. This causes the test case with the assertion to be marked as a failure.
- **Test case atomization:** where a test case with n assertions is turned into n test cases with one assertion through test case purification. This generates a set of test cases for each test case that fails.
- **Dynamic program slicing:** the act of generating purified test cases by the removal of irrelevant statements. Dynamic slicing keeps the executed statements in a dynamic execution. This reduces the size of test cases needing to be executed.

2.5.2 Study Instruments

The authors used six open-source Java libraries in their experiment to discover the performance of test case purification. These six programs are used by other researchers of fault localization so using these libraries allows for a comparison of performance between previous and future works. The authors also mutated the code to create faults in the code, a strategy used by their cited works. The authors also used six fault localization techniques to test the effects of test case purification vs non-purified solutions.

2.5.3 Motivation

The paper gives a figure that shows a fraction of a Java program (the Apache Commons Lang lib). The authors took this program and injected a fault at line 20 (negating a conditional expression). After the fault is injected the test cases of AC Lang are run and only one test case fails (t1). The test case covers lines 3 to 21, but when modern fault localization frameworks like Tarantula are run it gives a rank for all lines (3-21) as equal. Because of this result it is hard to know the fault is at line 20. The authors used this issue as motivation in their research on test case purification. Since fault localization is an essential and time-consuming activity in software engineering, various fault localization techniques (spectrum-based, mutations, slicing) were created to identify faults in code based on execution traces of test cases. The paper posits that test case purification can improve fault localization performance by separating existing test cases into smaller, "purified", test cases. When combined with the original fault localization techniques, test case purification should result in better ranking of the program statements.

2.5.4 Related Work

Tarantula is a framework to localize and visualize faults [14]. Ochiai, Jaccard, and Tarantula are seen as the best in spectrum-based fault localization. This paper talks about test case purification as a way to make better use of existing test cases and can be applied to existing techniques like the above. There is also mutation and slicing based fault localization listed in related works. FIFL injects faults (mutants) and ranks edits based on suspiciousness of mutants. Slicing-based fault localization has been used to reduce the size of programs to avoid distributing irrelevant statements or to identify faulty statements. In this work the authors only change test cases for fault localization, not the programs themselves.

2.5.5 Future Work and Conclusion

The authors would like to conduct experiments on other Java projects as a measure of performance for the framework. They plan to design new ranking methods and explore how to reduce the time cost of test case slicing for test case purification. Lastly, they would like to apply test case purification to other software issues like regression testing and automatic software repair.

Test case purification had great results when applied to existing fault localization techniques. The majority of the purified test cases provided improvements on non-purified test cases, with only 2.4% causing worse results. Test case purification can be used with any framework doing automated fault localization. The breadth at which this new technique displays it's benefits makes test case purification a powerful framework for future fault localization research to use to enhance their tools. Despite test case purification being used to feed automated fault localization techniques new test cases, we believe it is also a good design pattern for software developers to follow. These simplified and targeted test cases lower coverage per test case, meaning if a purified test case fails, it should even quicken manual debugging efforts.

2.6 NOPOL: Automatic repair of buggy if conditions and missing preconditions with SMT

DeMarco et al. presented NOPOL, a framework that can automatically fix IF conditions as well as code missing preconditions [7]. NOPOL takes in a program and the test suite for the program to repair these types of bugs. Results ex-

tracted from passing and failing test cases are transformed into a Satisfiability Modulo Theory (SMT) problem. The result from this problem is then turned into a fix that can be merged into the program to fix the bug causing the failed test case. NOPOL goes beyond fault localization by actually producing code fixes that would normally require a developer to write. Not only does NOPOL potentially save time during the debugging phase, but also the development phase.

2.6.1 Keywords

- **Automatic repair:** the automatic repair of bugs like buggy IF conditions using a given approach.
- **Buggy IF condition:** conditional statements that have some kind of bug that cause the program execution to lead to a different branch in the execution path, leading to unexpected behavior. NOPOL aims to fix these bugs automatically.
- **Missing precondition:** a common bug relating to branching. Preconditions distinguish different values of variables and when they are missing it can lead to bugs like null pointer exceptions. NOPOL aims to fix these bugs automatically.
- **Angelic fix localization:** an algorithm that identifies repair locations and repair oracles. It is a technique of modifying the program state to find angelic pairs.

2.6.2 Study Instruments

NOPOL used several case studies as study instruments that other systems in the past have used as well as future systems can use to compare results to NOPOL and others. The first case study was a classical program, Tcas, that was used as an example by the related work SemFix. Tcas is a collision avoidance system in the Software-artifact Infrastructure Repository (SIR) and is made up of 125 lines of code. The patch created by NOPOL differs from the patch made by SemFix but both pass all test cases and are an acceptable fix. The second case study was the Commons-Math library in Apache. Commons-Math is a lightweight library for common math/stats problems and is made up of 5000 lines of code with 353 test cases. This was a real-world use case as the bug NOPOL found and fixed was an actual bug in the library. The last case study was the missing precondition example created by the authors to show how NOPOL can fix missing preconditions. In order to fix an ArrayIndexOutOfBounds exception preconditions had to be generated by NOPOL and this SMT solution is converted as the final patch to fix the bug.

2.6.3 Motivation

The authors list the motivation of automatic software repair as decreasing the cost of fixing bugs. The automatically-produced patches can be used by developers as potential solutions or even put in as is if time is of the essence. Giving developers a potential solution should ease the time needed for the developers to understand the bug and design a fix for it.

2.6.4 Related Work

GenProg is a test-suite based program repair approach that uses genetic programming by viewing a program as an AST and generating another AST patch to provide multiple candidate patches and use the candidate that passes all test cases [3]. GenProg has an average success rate of 77 percent. SemFix is a program repair approach that uses semantic analysis. Unlike GenProg, SemFix generates patches by combining symbolic execution, constraint solving, and program synthesis. SemFix has a higher accuracy rate and executes in less time than GenProg. NOPOL can repair missing parameters which SemFix cannot. Related to the use of SMT solvers, Jha et al. mined program oracles based on examples and used SMT solvers to create constraints. This addressed the same problem as NOPOL by encoding the synthesis constraint with a first-order logic formula.

2.6.5 Future Work and Conclusion

For future work, the authors plan to evaluate their approach on more real-world bugs. NOPOL will be extended to fix bugs in conditionals for loop structures like while and for loops. Another limitation of NOPOL is the inability to deal with object parameters, a conditional statement that returns if a list contains a certain object cannot be generated through NOPOL. We believe this is a valid direction as real-world bugs are not as clean as the bugs they fixed during their experiments. Loop structures are very common in programming, so NOPOL must be able to deal with bugs within them to have impact in the real-world. In the current state, NOPOL can only generate solutions that are also quickly and easily created by software developers.

Despite these current limitations, NOPOL goes beyond the previous fault localization techniques by generating fixes for the bugs it detects. The goal of automated fault localization and debugging is to lessen the impact that the debugging phase has on the software development process. Pairing fault localization and fix generation hastens the debugging phase when the potential fixes are accurate and can deal with many types of errors. That said, developers may be hesitant to let generated fixes go into production code due to code quality and security concerns. At the very least, NOPOL can present developers with a good base solution that they can use to develop their own.

2.7 DynaMoth: Dynamic Code Synthesis for Automatic Program Repair

DynaMoth extends the NOPOL system by providing a new code synthesis engine rather than the original SMT engine (SMTSynth) [6]. DynaMoth uses GZoltar to localize fault statements and uses Spoon to do analyze and angelic value mine Java source code [19]. The main advantage DynaMoth has over SMTSynth is it's ability to generate more complex method calls in it's patches, while SMTSynth is limited to simple patches containing arithmetic, first order logic operators, and method calls without parameters. This enhancement is done by using dynamic exploration of tentative expressions. Theoretically, DynaMoth should be able to generate patches for all bugs that SMTSynth can. In practice there are bugs with DynaMoth's implementation that prevent it from generating the amount of fixes that SMTSynth does, but DynaMoth has proven that it can fix more complicated faults and generates simpler and more readable patches. These more readable patches are important since practitioners will have to integrate and maintain them in the codebase.

2.7.1 Keywords

- **Fault Classes:** classes of bugs where fault occurs and the system NOPOL targets.
- **Evaluated-Expression:** denoted as EEXP (c) is a pair e, v where e is a valid Java expression and v the value of the Java expression in a specific run-time context c.
- **Test-suite-based Repair:** test suite based repair uses a test suite as the specification of the correct behavior of the program.
- **Angelic Value Mining:** the angelic value mining step determines the required value of the buggy Boolean expression to make the failing test cases to pass.

2.7.2 Study Instruments

The authors of DynaMoth used the Defects4J dataset which contains projects with faults. Defects4J is billed as âĂIJthe largest open and structured database of real-world Java bugsâĂİ. The authors also used Grid5000, a high performance computing platform, to deal with the long running execution time involved for DynaMoth running on Defects4J. The use of this dataset allowed DynaMoth to produce empirical results for three research questions: What is the bug fixing ability of DynaMoth on Defects4J? Are the patches created by DynaMoth more readable than those by the SMT engine? And what is the run-time performance of DynaMoth? This gives future research a dataset and goal metrics to compare their results with those of DynaMoth.

2.7.3 Motivation

The paper describes implementation and details synthesis engine called as DynaMoth for NOPOL which initially used SMTSynth and now DynaMoth. DynaMoth uses dynamic exploration of tentative expressions based on Angelic Value Mining to collect run-time context.The evaluation shows that Nopol with DynaMoth is capable of repairing bugs that have never been repaired so far. DynaMoth's main design goal is to be capable to synthesize patches with complex operators and those involving method calls with parameters which yields a huge search space.

2.7.4 Related Work

CodeHint is a closest synthesis engine as DynaMoth, which synthesizes Java code from run-time data inside the development environment by using JRE run-time debug interface to collect run-time data and call methods [13]. However major difference between CodeHint and DynaMoth would be that CodeHint has a goal to help developers whereas DynaMoth tries to achieve automatic repair.Also, CodeHint synthesize an expression for single run-time context whereas DynaMoth has to generate expression that is valid for many run-time contexts at same time which increases the DynaMoth's complexity in terms of synthesis engine. Other related work would be GenProg which uses genetic algorithm implemented in C to copy existing code in the program instead of synthesizing code. SemFix is another such tool that uses symbolic execution for fixing assignment and conditions, which uses same algorithm as SMTSynth for synthesis with which the paper compares in detail.

2.7.5 Future Work and Conclusion

The authors of the paper in future plan to implement additional optimizations, mostly to address the combinatorial explosion of the search space caused by method accepting many arguments. In additional to it, the paper mentions using many candidate Evaluated-Expressions to be used as parameters. This is an important direction for DynaMoth as developers are unlikely to use a tool that requires a long runtime to produce a solution. Perhaps the use of optimizers to deal with the large search space that exists when DynaMoth runs will lower this execution time to an acceptable level.

The results of DynaMoth complement that of NOPOL's original synthesis engine SMTSynth. For bug fixing ability, DynaMoth and SMTSynth both fixed 19 of the same bugs. SMTSynth solved 16 bugs that DynaMoth did not solve, while DynaMoth only solved 8 that SMTSynth could not. This shows that DynaMoth is not the optimal engine for NOPOL, but instead would need to be combined with SMTSynth to fix the maximum amount faults. Unfortunately, the remaining amount of bugs unfixed totaled 181. This could be due to the bugs being of a more complex nature than currently covered by NOPOL and it's engines. So while it is impressive to generate new solutions as a way to provide developers with a better debugging framework, a majority of faults are left untouched. The authors did find that DynaMoth produced patches that were simpler and easier to read by generating a smaller amount of expressions and operators. For performance, DynaMoth was slower than SMTSynth when it generated a solution for a fault. Both had comparable times amongst all executions, but this execution time is quite high. On average, for all bugs the frameworks took around 38 minutes to execute, with long running executions over one hour and 30 minutes timing out. This time drops drastically when the bugs can be patched with an average execution time of 6 minutes and 9 minutes for SMTSynth and DynaMoth respectively. We believe that this amount of time should be acceptable to practitioners when using NOPOL in the real-world. The real problems comes with the low patch rate (27/224 fixed, or 12%). This low rate means that the majority of execution times fall into the average of 38 minutes rather than the quicker solve rate of 9 minutes. Until DynaMoth and other engines implemented in NOPOL can meet a higher solve rate we do not believe it will be attractive to software engineering professionals.

2.8 Practitioners' expectations on automated fault localization

Rather than posit some framework or algorithm to perform automated fault localization and debugging, Kochhar et al. surveyed software engineering practitioners for their feedback on the line of research involving automated fault localization [17]. They surveyed 386 software engineers from over 30 countries about their expectations of the research being done in fault localization. The main focus of the survey was to investigate the willingness of software engineers to adopt the fault localization techniques defined in the research. The authors then looked at existing fault localization research and compared what the research offered to what was expected of it by the practitioners. Future research in automated fault localization can then use these expectations to inform the design goals of the frameworks they create.

2.8.1 Keywords

- **Empirical Study:** a study using empirical evidence which is knowledge acquired by means of the senses (like observation and experimentation).
- **Practitioners' Expectations:** insights gathered in the study about fault localization from 386 practitioners including their views on importance of fault localization and their thresholds and reasons for adopting or not adopting techniques that provide fault localization.
- **Trustworthiness:** the measure of a technique that rates it's ability to satisfy a minimum success rate.
- **Scalability:** the measure of a technique that rates it's ability to work on programs of multiple sizes from small programs of 1-100 lines of code to large programs of 1-1million LOC.

2.8.2 Study Instruments

The survey information is provided in the paper. First the survey collects demographics like whether the person is a professional software engineer, their experience, and their involvement in open source development. This filters respondents to remove any who might not understand the survey or might not be relevant to the survey's subject matter. The survey then gets the practitioner's expectations of fault localization research. The survey defines fault localization and asks the importance of fault localization to their work. The respondents are then asked what would drive their adoption of these fault localization techniques. Criteria such as availability of debugging data, minimum success criteria, minimum success rate, and minimum scalability are rated by the respondent. If a respondent replies they would not adopt a trustworthy technique they are asked why. Finally the survey allows for free-response from the respondent.

2.8.3 Motivation

The authors are investigating if software engineering practitioners appreciate the line of research involving fault localization. They perform an empirical study by surveying 386 practitioners about their expectations of this research as a way to inform what directions that researchers need to put effort to develop fault localization techniques that actually matter to practitioners. If the practitioners do not adopt these new techniques then the research into fault localization does not have a significant impact on the software engineering world.

2.8.4 Related Work

Lo et al. had surveyed practitioners at Microsoft on how they perceived the relevance of research published in ICSE and FSE by asking them to rate 40 selected papers by answering about how important that research was [4]. Kochhar et al. instead focus on adoption rather than relevance and specifically about fault localization rather than the entirety of software engineering studies. So the survey the authors present is much more focused than Lo et al.'s work. Perscheid et al. studied debugging practice of professional software developers [16]. Instead the authors of this paper asked what practitioners wanted for a future tool in fault localization and their likely adoption of that tool. There were also related empirical studies on fault localization. For instance, some performed an empirical study to evaluate Tarantula against other fault localization techniques. Wang et al. looked into the usability of BugLocator and found that it was only useful if bug reports come without rich and identifiable information, otherwise it harmed the developers performance [18]. This paper extends and complemented all of the related studies by analyzing the expectations of the practitioners on the importance of the research in fault localization and their willingness to adopt these techniques, investigating thresholds for that adoption, and found interesting results for many research questions that were not considered before.

2.8.5 Future Work and Conclusion

The authors plan to develop fault localization techniques that can bring current research closer to the adoption thresholds set by the respondents of the survey. They also plan to explore questions such as: why practitioners prefer some of the coarser granularity, why are more experienced developers less enthused on automated fault localization, and how different are the expectations of open source and professional practitioners?

The authors found that when comparing the desired capabilities of automated fault localization research, only a few pieces of research met their requirements. Only 2 papers examined worked at method level granularity, the most preferred level by practitioners. The majority of the papers worked at statement level granularity, which practitioners found to be too coarse-grained. For the desired success rate of fault localization techniques, several papers satisfied 50% of the respondents by having a 50% success rate. The issue with these papers is that they worked on the coarse level granularity that is not preferred by the practitioners. Further, no papers provided a success rate of 75%+, the preferred rate of the majority of practitioners. On a positive note, the efficiency, scalability, and use of test cases for debugging data capabilities of current automatic fault localization research satisfies the majority of practitioners. The authors also found that the majority of respondents found fault localization "essential" or "worthwhile", with less than 10% finding it "unimportant" and "unwise".

Though this paper did not introduce a new framework or algorithm for fault localization, it's contribution to research in the area is very important. In order for research to have real-world implications it must be adopted by the real-world. In order for the researched solutions to be adopted it needs to provide some value to the adopters. This paper works to define what specifications the research solutions need to have in order to be adopted. Now researchers have a performance goal for their tools and algorithms. Researchers not meeting these goals may see their research have little impact outside of their lab or academia, but if they can meet these goals their research is more likely to impact the industry that the problem resides in. This is why we believe the further discovery of practitioner expectations on fault localization should continue and researchers should use these expectations to drive their goals in researching automated fault localization.

3. FUTURE WORK

All of these research papers work to alleviate the strain of the debugging phase on the software development life-cycle.

Debugging can be a costly phase that consumes resource time that would be better spent in design or implementation. The works mentioned in this paper have laid a strong foundation for future automated fault localization research to build upon. Many used study instruments that future works can use to compare their results to show their improvements with various metrics like execution time, accuracy, and ability. We recommend that all future works include datasets that include real-world problems. Real-world faults are complex, and are inside of programs that are complex. Simple programs made to represent a fault do not accurately represent the challenges these frameworks will face in the industry. In order to get software practitioners to adopt these frameworks, researchers must show their value on a real-world level. Most of the papers did not test their techniques on real-world problems, but instead used program suites that had well-defined test cases. This is great for comparing performance results but does not prove that the performance of the frameworks will carry into the real-world. Instead, once automated fault localization techniques are polished release them into the wild and see what practitioners do with them. Then, gather results on whether or not it actually improved the lives of the programmers using them. Did it help them find bugs faster? Did it lead them down a path they would have not otherwise thought of?

Another potential direction is to combine automated test case generation with automated fault localization. The works discussed in this paper depend heavily on coverage of test cases, both passing and failing. In order to apply these automated fault localization techniques to real-world programs they will have to deal with a sub-optimal amount of test cases, as most real-world programs rarely contain 100% coverage. A solution to this problem is to have the frameworks is to first attempt to increase test case coverage by generating new test cases. If test case generation can produce enough coverage, then the success rate of the fault localization techniques may increase. These test cases can also be purified as defined by Xuan and Monperrus to increase the effectivness of the automated fault localization techniques.

The last mentioned work, **Practitioners' expectations on automated fault localization**, is the best way forward to define performance parameters of future automated fault localization and debugging techniques. Understanding the expectations of those who would most benefit from your research will result in a more adoptable work. Kochharet al. did not find any holy grail of fault localization that fulfilled the expectations of the practitioners they interviewed. For instance, DynaMoth had a very long execution time, something unacceptable to many practitioners. To solve this issue researchers should look into optimizations for their techniques. Another issue with the current works is their ability and success rate. Practitioners expect a high success rate for automated debuggers, if these techniques cannot meet a minimum success rate then they will not be adopted. Further, the ability of the techniques must be questioned. The majority of the papers worked at method level granularity, a level too coarse-grained for many practitioners. The future frameworks of automated fault localization and debugging must aim to improve their success rates and do so at a finer level of granularity. Another requested feature is for the automated fault localization framework to present the developer with some sort of rationale as to why it chose the lines of code it did as the location of the fault. Respondents believed that rationale given would increase understanding of the fix that would need to be done. That is, the respondents were worried that when given the location of a fault they would not know why it was a fault or how they would fix it. Research into extending the frameworks to include this rationale to assist software practitioners in fixing a fault would obviously provide a desired service. We would also like to see an expanded survey to include leading companies in the software industry (Google, Facebook, Netflix, etc). A wide-scale, comprehensive survey is needed to create well-defined goals for automated fault localization and debugging techniques.

Researchers should increase automated fault localization and repair frameworks' value by implementing them in development pipelines. As DevOps practices becomes commonplace in the industry the likelihood of large commit with potential faults decreases. While automated fault localization is very useful for saving developer time on large programs, continuous integration [8], delivery, and deployment [12] practices will already allow developers to detect faults quickly. These practices have incremental software changes automatically tested, analyzed, and integrated or deployed often, rather than after long development cycles [2]. This means that it becomes easier to manually spot broken code since a smaller amount of code snippets should be committed in each push, and code with faults will be detected at each commit. Automated fault localization and repair techniques in this paper can work with DevOps instead of being phased out by it. After a successful build, the code commit should run through various tests and analysis to detect any potential errors. At this phase automated fault localization techniques can inform automated repair techniques (like NOPOL) and re-attempt to deploy passing code, removing the requirement of developer intervention. Gulsher et al. have produced a fine-tuned spectrum based fault localization technique called patterned spectrum analysis that is inspired by continuous integration [10]. Patterned spectrum analysis tested against the Defects4J dataset was more effective than the spectrum based fault localization described by Abreu et al. These are encouraging results that show CI and automated fault localization should work together to create a better debugging framework.

Further, researchers can also look to increase the breadth of automated fault localization and debugging by moving beyond code fixes. One issue we can apply automated fault localization to is program performance. CPU, memory, and hardware metrics can be visualized by flame graphs. Flame graphs are interactive visualizations for code-path stack traces [9]. Using OS level logging, like linux's perf command, stack traces are recorded into stacks. These stack traces are then collapsed, folded, and then a flame graph created in a svg file showing how much time certain program calls spend in the CPU, how much memory they use, and more. We believe automated fault localizers can use the same stack traces to detect code-paths that consume too much CPU and memory. Rather than having developers visually check flame graphs for misbehaving code, automated fault localization frameworks can detect problematic code and report that code to developers. This gives automated fault localization techniques a breadth of impact beyond code correctness.

Finally, several of the automated fault localization tools like GZoltar and VIDA were implemented as Eclipse plugins. Developers are unlikely to want to install and main-

tain an extra piece of software for automated fault localization so integration with the developer's IDE is a good strategy to increase the adoptability of the framework. This adheres to the design pattern followed by VIDA and is confirmed by Kochhar et al. in their survey of practitioners. Since automated fault localization and debugging techniques are meant to increase the quality of life for developers, researchers should make the frameworks easy to integrate into a developers existing work flow.

4. CONCLUSION

While we believe that automated fault localization and debugging is not mature enough to take the real-world by storm, the problem it seeks to solve is an important one. The debugging process is a costly issue for the software industry, it causes frustration in developers (from beginners to experts), and requires solutions. For now, humans are still a lot better at finding and fixing faults, but perhaps the real impact of automated fault localization, debugging, and even repair is not in helping human software developers. Once computers are doing the programming, instead of humans, there will need to be algorithms that help artificial programmers find and fix faults. But for humans, the growing research and application in continuous integration, lessens the need for automated fault localization. Better professional practices are allowing developers to find newly introduced faults quickly. In order to compete, automated fault localization and debugging techniques need to be able to find a greater number of faults, be able to find more complex faults, and find these faults faster. Researchers should also continue to think about integrating frameworks that provide continuous integration and automated fault localization. Despite this, the techniques discussed in this paper provide a good foundation for researcher seeking to go the automated route for debugging. If future work by researchers can meet practitioner expectations, and does so on real-world faults, we expect automated fault localization and debugging to quickly be picked up in the software industry. Until then we remain skeptical that the automated fault localization techniques mentioned in this paper will become common place with software engineering professionals.

5. ACKNOWLEDGMENTS

We would like to thank Dr. Menzies for introducing us to the world of MASE. This has been one of the most interesting courses we have had the privilege to take.

6. REFERENCES

[1] G. R. V. G. A. Abreu R, Zoeteweij P. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, page 1780âĂŞ1792, November 2009.

[2] H. B. M. G. A. G. J. H. J. M. B. M. T. S. M. S. S. W. L. W. C. Parnin, C. Atlee. The top 10 adages in continuous deployment. *IEEE Software*, 2016.

[3] S. F. Claire Le Goues, ThanhVu Nguyen and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, pages 54–72, 2012.

[4] N. N. D. Lo and T. Zimmermann. How practitioners perceive the relevance of software engineering research. *In FSE*, 2015.

[5] L. Z. J. S. H. M. Dan Hao, Lingming Zhang. Vida: Visual interactive debugging. *In Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 583–586, May 2009.

[6] T. Durieux and M. Monperrus. Dynamoth: dynamic code synthesis for automatic program repair. *Proceedings of the 11th International Workshop on Automation of Software Test (AST 2016)*, pages 85–91, May 2016.

[7] D. L. B. M. M. Favio DeMarco, Jifeng Xuan. Automatic repair of buggy if conditions and missing preconditions with smt. *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014)*, pages 30–39, May 2014.

[8] M. Fowler. Continuous integration, http://www.martinfowler.com/articles/continuousintegration.html.

[9] B. Gregg. Flame graphs, http://www.brendangregg.com/flamegraphs.html.

[10] S. D. Gulsher Laghari, Alessandro Murgia. Fine-tuning spectrum based fault localisation with frequent method item sets. *In Proceedings of ASEâĂŹ16*, September 2016.

[11] H.-Y. Hsu and A. Orso. Mints: A general framework and tool for supporting test-suite minimization. *In Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, pages 419–429, May 2009.

[12] J. Humble. Continuous delivery vs continuous deployment, https://continuousdelivery.com/2010/08/continuous-delivery-vs-continuous-deployment/.

[13] R. B. H. J. Galenson, P. Reames and K. Sen. Codehint: Dynamic and interactive synthesis of code snippets. *In Proceedings of the 36th International Conference on Software Engineering (ACM 2014)*, page 653âĂŞ663, 2014.

[14] H. M. Jones, J.A. Empirical evaluation of the tarantula automatic fault localization technique. *In Proceedings of ASEâĂŹ05*, page 273âĂŞ282, November 2005.

[15] A. P. JosÃl' Campos, AndrÃl' Riboira and R. Abreu. Gzoltar: an eclipse plug-in for testing and debugging. *In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 378–381, September 2012.

[16] M. T. M. Perscheid, B. Siegmund and R. Hirschfeld. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal*, 2016.

[17] D. L. Pavneet Singh Kochhar, Xin Xia and S. Li. Practitioners' expectations on automated fault localization. *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 165–176, July 2016.

[18] C. P. Q. Wang and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. *In Proceedings of the 2015 International Symposium on Software Testing*, pages 1–11, 2015.

[19] N. P. C. N. R. Pawlak, M. Monperrus and L. Seinturier. Technical report 5901. *Proceedings of the 25th International Symposium on Software Testing*

and Analysis (ISSTA 2016), 2015.

[20] S. RTI Health and E. Research. The economic impacts of inadequate infrastructure for software testing final report. *RTI Project Number 7007.011*, May 2002.

[21] J. Xuan and M. Monperrus. Test case purification for improving fault localization. *In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 52–63, November 2014.