Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science00 (2014) 000–000

Procedia Computer Science

www.elsevier.com/locate/procedia

Recent Developments in the Field Of Bug Fixing

Varun Mittal[a] , Shivam Aditya[b] ,*

[a]*Department of Computer Science and Engineering, VIT University, Vellore, India*
[b]*Department of Computer Science and Engineering, VIT University, Vellore, India*

Abstract

In recent times, there have been lot of work done in the field of bug fixing in the software development process. We hereby have conducted a review of the seven recent techniques in the field of bug fixing and have made a report on it.

Keywords: Bug fixing; errors; recent; software; developement

1. Introduction

Debugging is one of the most time-consuming activities that software developers engage in. programmers may need to intensively inspect code, employ a debugger, communicate with fellow programmers, write test cases, apply possible fixes, and do regression tests in order to identify the presence of the bugs in the program. A NIST report estimated that debugging costs industry billions of dollars each year. Given the cost of finding a fix to an existing bug, it is desirable that the fix be applied in all the places where the bug actually occurs. Nevertheless, programmers often fail to do so.

*Corresponding author. Tel.: +91-9600703668;
E-mail address: varun.mittal2011@vit.ac.in

*Corresponding author. Tel.: +91-9597361736;
E-mail address:sa.shivam.aditya@gmail.com

2. Recent Papers in the Field

2.1. Generating Fixes from Object Behavior Anomalies

The paper was made by Valentin Dallmeier, Andreas Zeller of Dept. of Computer Science, Saarland University, Germany and Bertrand Meyer, Chair of Software Engineering, ETH Zürich Switzerland [1].

This paper is about the process of debugging. So, when a program fails, debugging starts i.e. the process of locating and fixing the bug that causes the failure. Recent years have seen considerable advances in automated debugging.

Even with automated bug localization, the programmer must still assess these locations to choose where and how to fix the program. The goal of this work was to effectively automate the entire debugging process for a significant subset of programming errors.

It also tells us about the advances in recent years that have made it possible in some cases to locate bugs automatically. But debugging is also about correcting bugs. The results reported in this paper, from the new PACHIKA tool, suggested that such a goal may be reachable.

PACHIKA leverages differences in program behavior to generate program fixes directly. It does so by automatically inferring object behavior models from executions, determining differences between passing and failing runs, generating possible fixes, and assessing them via the regression test suite. Evaluated on the ASPECTJ bug history, PACHIKA generates a valid fix for 3 out of 18 crashing bugs; every fix pinpoints the bug location and passes the ASPECTJ test suite.

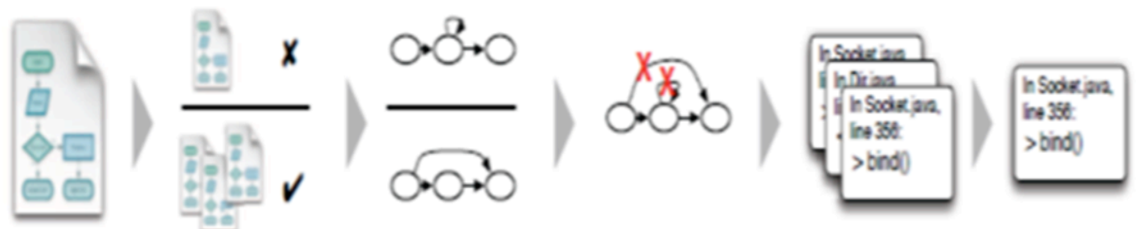

Fig 1. PACHIKA tool process

PACHIKA takes a Java program and out of its passing and failing runs, it mines object behavior models. From differences between the models , it derives fix candidates which it then validates against the regression test suite. Only validated fixes are preserved.

It concludes that the future of automated debugging lies in the automatic generation of fixes. Applied to real-life Java programs, our PACHIKA tool can generate fixes for 3 out of the 18 post-release bugs that crash ASPECTJ. By leveraging the difference between normal and abnormal behavior, we successfully constrain the search space to quickly generate potential fixes that not only remove the problem at hand, but also have a high diagnostic quality. Starting with behavioral differences, coupled with strict filtering via the test suite ensures a zero rate of false positives, ensuring that PACHIKA increases productivity. The approach can easily be extended to quality assurance beyond testing: As soon as a specification can be automatically validated, PACHIKA can leverage it to filter fix candidates—such that only true corrections remain.

2.2. Propagating Bug Fixes with Fast Subgraph Matching

The paper was made by Boya Sun, Gang Shu, Andy Podgurski, Shirong Li, Shijie Zhang, Jiong Yang of Department of Electrical Engineering and Computer Science, Case Western Reserve University [2].

In this research paper, they presented a powerful and efficient approach to the problem of propagating a bug fix to all the locations in a code base to which it applies. Their approach represented bug and fix patterns as subgraphs of a system dependence graph, and it employed a fast, index-based subgraph matching algorithm to discover unfixed bug-pattern instances remaining in a code base. They also developed a graphical tool to help programmers specify bug patterns and fix patterns easily. They evaluated their approach by applying it to bug fixes in four large open-source projects. The results indicated that the approach exhibits good recall and precision and excellent efficiency.

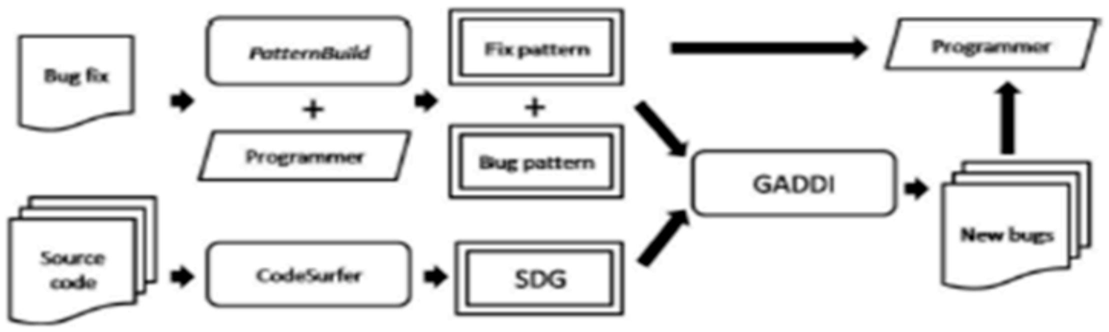

Fig 2. The GADDI Process

They have presented an efficient and effective approach to solve the bug fix propagation problem by finding matches of a query graph representing a bug pattern. An easy-to-use graphical tool PatternBuild is used for specifying bug and fix patterns, and fast subgraph matching based on graph indexing is used to find bug instances in a code base.

Empirical evaluation on large open source projects indicated that the subgraph matching algorithm used for detecting potential bugs achieved very high recall, which indicates that this approach is able to propagate bug fixes nearly completely. The GADDI algorithm was shown to be very efficient when used in our approach.

The precision of the proposed approach averaged a little more than 50%. Although this is acceptable, better precision is required. Improper node labeling is the main cause of false positives. In order to help ensure that the same label is assigned to all semantically equivalent nodes supervised learning approaches seem to be appropriate. Features can be selected to characterize all the factors that can affect labeling: surrounding dependences, ASTs (abstract syntax tree) text of source code, etc. Another cause of false positives is that some bug and fix patterns are not universally applicable. To solve this problem and also to prune results, functionality was provided to let programmers define more constraints on the bug and fix patterns.

They also hoped to add functionality to support semiautomatic correction of buggy code. This seems to be plausible since they highlight code changes according to the graph edit distance algorithm, so we can get an edit script relating the bug pattern and the fix pattern, which might be used to change a potential bug instance into a fix instance.

2.3. Debugging in the Large via Mining Millions of Stack Traces

The paper was made by Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, Microsoft Research Asia and Tao Xie, North Carolina State University [3].

The paper tells us about how given limited resource and time before development-site testing, software release, and debugging become more and more insufficient to ensure satisfactory software performance. The counterpart for debugging in the large pioneered by the Microsoft Windows Error Reporting (WER) system focusing on crashing/hanging bugs, the emergence of the performance debugging in the large has been possible due to the available infrastructure support to collect execution traces with performance issues from a huge number of users at the deployment sites. Performance analysts face a serious challenge in the form of performance debugging the numerous and complex traces at various deployment sites. An innovative approach to enable performance debugging in the large called StackMine has been proposed in this paper that mines callstack traces to help performance analysts effectively discover highly impactful performance bugs (e.g., bugs impacting many users with long response delay). As of now StackMine has been applied in performance-debugging activities at a Microsoft team for performance analysis that has been used for a large no. of execution traces which is a clear indication of a successful technology-transfer effort since December 2010, The example of performance analysts conducting an evaluation of StackMine on performance debugging in the large for Microsoft Windows 7 is an example of real-time usage of StackMine. The results of the evaluation on a third-party application by performance analysts highlighted substantial benefits offered by StackMine in performance debugging in the large for large-scale software systems.

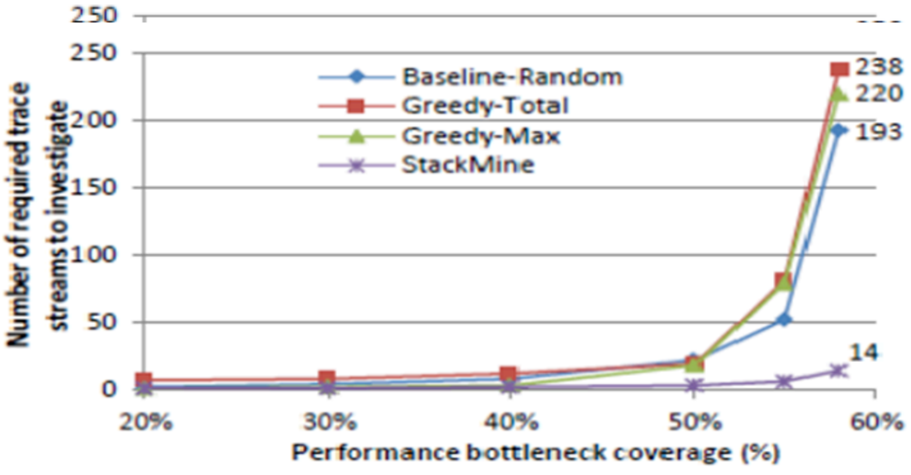

Fig 3. Performance Chart

The evaluations on two large-scale real-world software products (Microsoft Windows 7 and a third-party application) demonstrated StackMine's substantial benefits in performance debugging in the large. Exemplified by WER and StackMine, we envision and advocate a game-changing paradigm for software quality assurance in the large based on usage data collected from the real world, in order to cope with increasingly large and complex modern software systems, like ultra-large-scale systems.

2.4. Where Should the Bugs Be Fixed?

The paper was made by Jian Zhou, Hongyu Zhang, and David Lo, School of Software, Tsinghua University, Beijing 100084, China [4].

Software quality is vital for the success of a software project. Although many software quality assurance activities (such as testing, inspection, static checking, etc) have been proposed to improve software quality, in reality software systems are often shipped with defects (bugs). For a large and evolving software system the project team could receive a large number of bug reports over a long period of time. For example, around 4414 bugs were reported for the Eclipse project in 2009.

Once a bug report is received and confirmed, the project team should locate the source code files that need to be changed in order to fix the bug. However, it is often costly to manually locate the files to be changed based on the initial bug reports, especially when the numbers of files and reports are large. For a large project consisting of hundreds or even thousands of files, manual bug localization is a painstaking and time-consuming activity. As a result, the bug fix time is often prolonged, maintenance cost is increased and customer satisfaction rate is hampered.

In recent years, some researchers have applied information retrieval techniques to automatically search for relevant files based on bug reports [16, 25, 31, 32]. They treat an initial bug report as a query and rank the source code files by their relevance to the query. The developers can then examine the returned files and fix the bug. These methods are information retrieval based bug localization methods.

Unlike spectrum-based fault localization techniques [1, 18, 19, 22, 23], information retrieval (IR) based bug localization does not require program execution information (such as passing and failing traces). They locate the bug-relevant files based on initial bug reports.

Many of the existing IR-based bug localization methods are proposed in the context of feature/concept location, using a small number of selected bug reports.

Thus, the project team could receive a large number of bug reports for a large and evolving software system . A daunting task is to find the source code files that need to be altered in order to fix the bugs. When a bug report is received by the developers it is quite desirable that they have all the information about the files that are to be changed in order to fix the bugs. In this paper, they have proposed BugLocator, an information retrieval based method for locating the relevant files for fixing a bug.

BugLocator ranks all files based on the textual similarity between the initial bug report and the source code using a revised Vector Space Model (rVSM), taking into consideration information about similar bugs that have been fixed before.

More than 3,000 bugs were localized and reported when large-scale experiments were performed on four open source projects which was a clear indication that files containing bugs can be effectively and efficiently be located by BugLocator For example, buggy files of Eclipse 3.1 containing 62.60% bugs are ranked in the top ten among 12,863 files. Their experiments also show that BugLocator outperforms existing state-of-the-art bug localization methods.

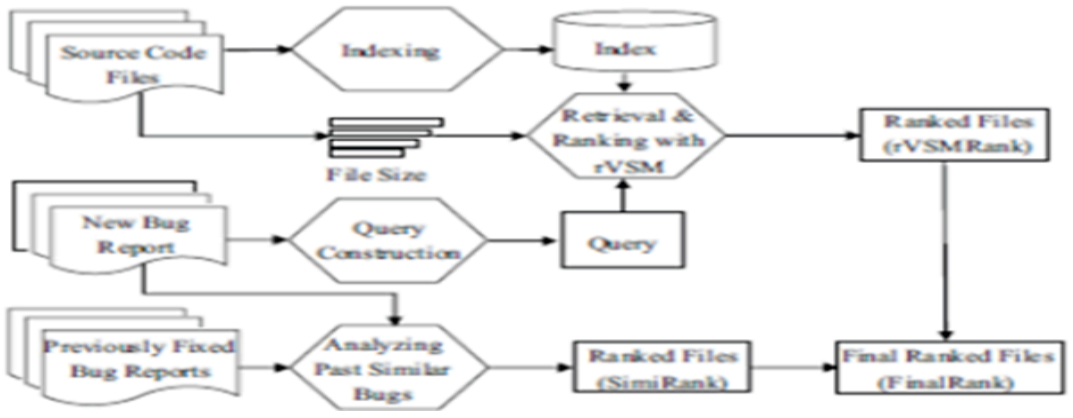

Fig 4. Structure of Bug Locator

2.5. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each

The paper was made by Claire Le Goues Michael Dewey-Vogt (Computer Science Department, University of Virginia, Charlottesville, VA), Stephanie Forrest (Computer Science Department, University of New Mexico, Albuquerque, NM) and Westley Weimer (Computer Science Department, University of Virginia, Charlottesville, VA.) [5]

The paper tells us that program evolution and repair are major components of software maintenance, which accounts for a sizable portion of the total cost of software production. Automated techniques to reduce their costs are therefore especially beneficial. Developers for large software projects must confirm, triage, and localize defects before fixing them and validating the fixes. Although there are a number of tools available to help with triage, localization, validation and even confirmation, generating repairs remains a predominantly manual, and thus expensive, process. At the same time, cloud computing, in which virtualized processing power is purchased cheaply and on-demand, is becoming commonplace.

Research in automated program repair has focused on reducing defect repair costs by producing candidate patches for validation and deployment. Recent repair projects include ClearView, which dynamically enforces invariants to patch overflow and illegal control-flow transfer vulnerabilities; AutoFix-E, which can repair programs annotated with design-by-contract pre- and post-conditions; and AFix, which can repair single-variable atomicity violations.

In their previous work, they introduced GenProg, a general method that uses genetic programming (GP) to repair a wide range of defect types in legacy software (e.g., infinite loops, buffer overruns, segfaults, integer overflows, incorrect output, format string attacks) without requiring a priori knowledge, specialization, or specifications. GenProg searches for a repair that retains required functionality by constructing variant programs through computational analogs of biological processes.

The goal of this paper is to evaluate dual research questions: "GenProg can repair what fraction of bugs existing in the given process ?" and "What is the cost of repairing a bug with GenProg?" Three important insights were combined together to answer these questions. Their key algorithmic insight is to represent candidate repairs as patches, rather than as abstract syntax trees. These changes were critical to GenProg's scalability to millions of lines of code which was a crucial component of our evaluation. They introduced new search operators that dovetail with this representation to reduce the number of ill-formed variants and improve performance. Their key performance insight is to use off-the-shelf cloud computing as a framework for exploiting search-space parallelism as well as a source of grounded cost measurements. Their key experimental insight is to search version control histories exhaustively, focusing on open-source C programs, to identify revisions that correspond to human bug fixes as defined by the program's most current test suite.

They combined these insights and presented a novel, scalable approach to automated program repair based on GP, and then evaluated it on 105 real-world defects taken from open-source projects totaling 5.1 MLOC and including 10,193 test cases.

Thus, we can tell that there are more bugs in real-world programs than human programmers can realistically address. This paper evaluated two research questions: "What fraction of bugs can be repaired automatically?" and "What is the cost of repairing a bug automatically?" As said earlier, in previous work, they presented GenProg, which uses genetic programming to repair defects in off-the-shelf C programs. To answer these questions they: (1) proposed novel algorithmic improvements to GenProg that allow it to scale to large programs and find repairs 68% more often, (2) exploited GenProg's inherent parallelism using cloud computing resources to provide human competitive cost measurements, and (3) generated a large, indicative benchmark which was used for systematic evaluations. They evaluated GenProg on 105 defects from 8 open-source programs totaling 5.1 million lines of code and involving 10,193 test cases.

GenProg automatically repairs 55 of those 105 defects. This kind of evaluation is the largest available of its kind, and is often two orders of magnitude larger than previous work in terms of code or test suite size or defect count. Public cloud computing prices allow our 105 runs to be reproduced for $403; a successful repair completes in 96 minutes and costs $7.32, on average.

Concluding, their overall goal was to reduce the costs associated with defect repair in software maintenance. GenProg requires test cases and developer validation of candidate repairs, but reduces the cost of actually generating a code patch. While these results are only a first step, they have implications for the future of automated program repair. For example, part of the high cost of developer turnover may be mitigated by using the time saved by this technique to write additional tests, which remain even after developer's leave, to guide future repairs. GenProg could also be used to generate fast, cheap repairs that serve as temporary bandages and provide time and direction for developers to find longer-term fixes.

They directly measured the time and monetary cost of their technique by using public cloud computing resources. Their 105 runs can be reproduced for $403: this can be viewed as $7.32, and 96 minutes, for each of 55 bug repairs. While we do not have a quantitative theory that fully explains how GenProg works, the results of the systematic benchmark suite will allow us to investigate such issues in the future.

We consider our results to be strongly competitive, and hope that they will increase interest in this research area.

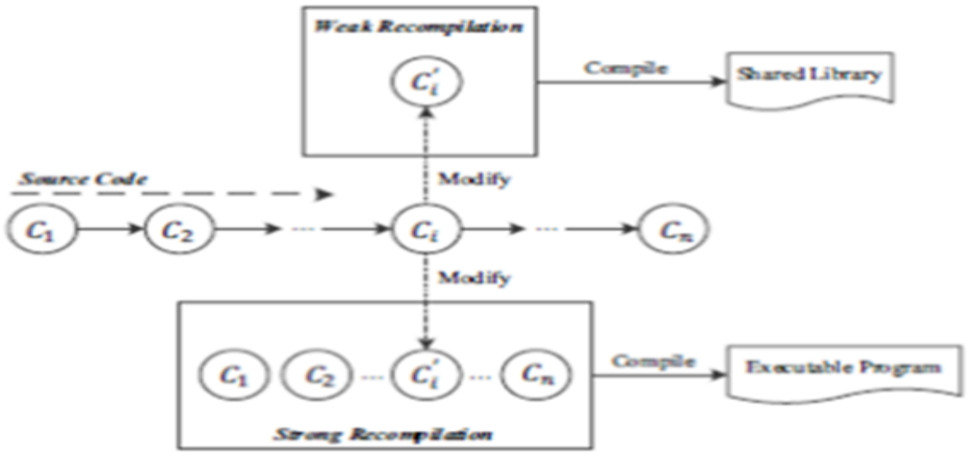

Fig 5. Weak recompilation versus strong compilation

2.6. Making Automatic Repair for Large-scale Programs More Efficient Using Weak Recompilation

This paper was made by Yuhua Qi, Xiaoguang Mao and Yan Lei Department of Computer Science and Technology National University of Defense Technology, Changsha, China [6].

The paper tells us about how program repair is a tedious and difficult activity that requires lots of resources spent on locating and fixing program bugs. And recently there is some promising work, on automated program repair, reducing the cost of software maintenance. In general, the repair process can be divided into three phases: locate the program bug; generate the candidate patches in light of some specified rules; validate these patches through some test cases again and again until a valid patch is found. To the best of our knowledge, current works mainly focus on the second phase: how to generate the candidate patches. However, cost reduction techniques in the process of program repair are rarely considered. This paper seeked to reduce the computational cost by optimizing the program recompilation process.

In large-scale programs, it consumes a lot of time for recompiling and reinstalling the modified (patched) program which is the result of automatically repairing a bug by modifying the program source code. Thus, a recompilation technique called weak recompilation was described in this paper which was used for suppressing the above time cost and to make the repair process more efficient.

The assumption of weak recompilation is that a program is assumed to be constructed from a set of components, and for each candidate patch only the changed code fragment in term of one component is recompiled to a shared library; the behaviors of patched program are observed by executing the original program with an instrumentation tool which can wrap specified function.

The advantage of weak recompilation is that reinstallation cost will be cut down completely. And redundant recompilation cost can be also suppressed, They also built WAutoRepair, a system which enables scalability to fix bugs in large-scale C programs with high efficiency. The results of the experiments confirmed that their repair system significantly outperforms Genprog which was a famous approach for automatic program repair. For the wireshark program containing over 2 million lines of code, WAutoRepair spent only 0.222 seconds in recompiling one candidate patch and 8.035 seconds in totally repairing the bug, compared to Genprog separately taking about 20.484 and 75.493 seconds, on average.

Concluding, for automated program repair they were the first to present and try to address the problem of expensive recompilation cost for large-scale programs.

Due to relatively high efficiency by applying weak recompilation, WAutoRepair spends less time on validating a candidate patch. That is, more candidate patches can be validated within a limited time bound; and thus it is possible that more complex modification rules (meaning that more trials) can be adopted to repair defective programs with large scales by WAutoRepair.

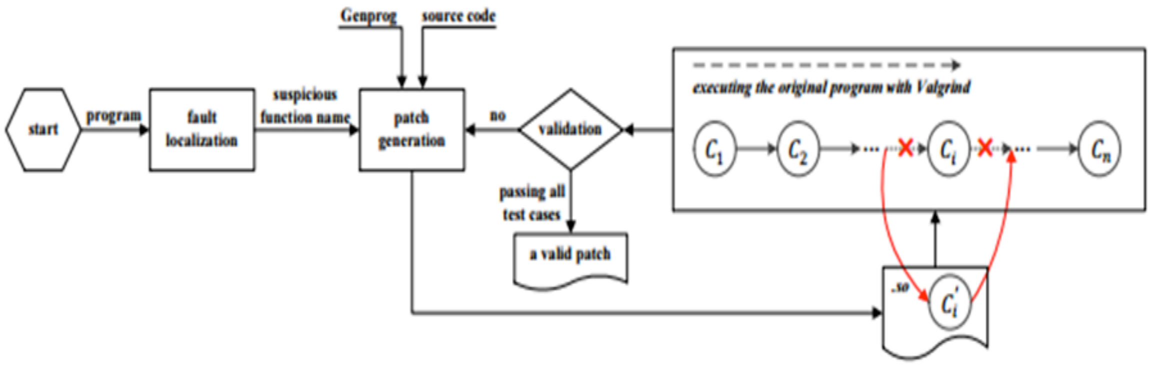

Fig 6.The Framework of WAutoRepair

2.7. R2Fix: Automatically Generating Bug Fixes from Bug Reports

This paper was made by Chen Liu, Jinqiu Yang and Lin Tan (University of Waterloo, ON, Canada) and Munawar Hafiz (Auburn University, AL, USA) [7].

In this paper, the authors told about how every day, an overwhelming number of bugs are reported.

For example, the Mozilla bug database, with a total of 670,359 bug reports, receives an average of 135 new bug reports daily. The corresponding bugs hurt security and software reliability which are not improved until the bugs are repaired.

Upon receiving a bug report, developers diagnose the root cause of the bug, generate a patch that can repair the bug, and commit the patch to the source code repository. They combined the first two steps (diagnosis and patch generation) under the label of repairing a bug, which is the emphasis of this paper. Developers' bug-fixing process is primarily manual; therefore the time required for producing a fix and its accuracy depend on the skill and experience of individuals.

Furthermore, Developers often need to fix more bugs than their time and resources allow. Although developers spend almost half of their time fixing bugs, bugs take years to be fixed on average.

Therefore, support to make it easier and faster for developers to fix bugs is in high demand. The capability to automatically generate patches from bug reports could: (1) save programmers' time and effort in diagnosing bugs and generating patches, allowing developers to fix more bugs or focus on other development tasks; and (2) improve software reliability and security by shortening the bug-fixing time.

In short, many bugs, even those that are known and documented in bug reports, remain in mature software for a long time due to the lack of the development resources to fix them. The authors proposed a general approach, R2Fix, which was used free-form bug reports to automatically generate bug-fixing patches. R2Fix combines machine learning techniques, past fix patterns and semantic patch generation techniques to fix bugs automatically. They evaluated R2Fix on three projects, i.e. Mozilla, Linux kernel and Apache, for three important types of bugs: null pointer bugs, buffer overflows, and memory leaks. R2Fix generates 57 patches correctly, 5 of which are new patches for bugs that have not been fixed by developers yet. They reported all 5 new patches to the developers; 4 have already been accepted and committed to the code repositories. The 57 correct patches generated by R2Fix could have shortened and saved up to an average of 63 days of bug diagnosis and patch generation time.

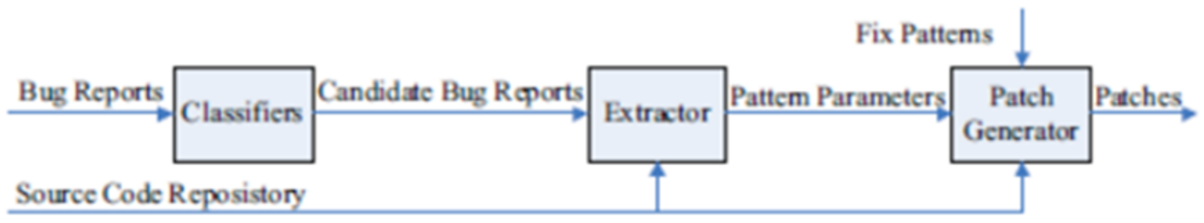

Fig 7. The Architecture of R2Fix

3. Conclusion

This research paper describes a review of the latest technologies in the field of Bug Fixing. Seven research papers have been studied and reviews about them have been given accompanied by diagrams. These reviews can help people and companies identify the appropriate bug fixing mechanisms needed for them saving themselves both time and money.

References

1. Generating Fixes from Object Behavior Anomalies by Valentin Dallmeier, Andreas Zeller and Bertrand Meyer, 2009 IEEE/ACM International Conference on Automated Software Engineering, 1527-1366/09 $29.00 © 2009 IEEE DOI 10.1109/ASE.2009.15.
2. Propagating Bug Fixes with Fast Subgraph Matching by Boya Sun, Gang Shu, Andy Podgurski, Shirong Li, Shijie Zhang and Jiong Yang, 21st International Symposium on Software Reliability Engineering 1071-9458/10 $26.00 © 2010 IEEE DOI 10.1109/ISSRE.2010.36, 2010 IEEE 21st International Symposium on Software Reliability Engineering
3. Performance Debugging in the Large via Mining Millions of Stack Traces by Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang and Tao Xie, ICSE 2012, 978-1-4673-1067-3/12/$31.00, 2012 IEEE 145
4. Where the Bugs Should Be Fixed? by Jian Zhou, Hongyu Zhang and David Lo, ICSE 2012, 978-1-4673-1067-3/12/$31.00, 2012 IEEE
5. A Systematic Study of Automated Program Repair:Fixing 55 out of 105 Bugs for $8 Each by Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest and Westley Weimer, ICSE 2012, 978-1-4673-1067-3/12/$31.00, 2012 IEEE
6. Making Automatic Repair for Large-scale Programs More Efficient Using Weak Recompilation by Yuhua Qi, Xiaoguang Mao and Yan Lei, 28th IEEE International Conference on Software Maintenance (ICSM), 2012, 978-1-4673-2312-3/12/$31.00 "c 2012 IEEE
7. R2Fix: Automatically Generating Bug Fixes from Bug Reports by Chen Liu, Jinqiu Yang and Lin Tan and Munawar Hafiz, 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, 978-0-7695-4968-2/13 $26.00 © 2013 IEEE DOI 10.1109/ICST.2013.24