Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 55 Issue: 10:2022
DOI10.17605/OSF.IO/F3ZCE

FAULT LOCALIZATION MODEL FOR EVALUATING THE CAPABILITIES OF AUTOMATIC PROGRAM REPAIR WITH CHECKPOINTS

T. MAMATHA
Research Scholar, Dept. of CSE, JNTUA University, Anantapuramu, Andhra Pradesh, India.
Email: mamathat7@gmail.com

B. RAMA SUBBA REDDY
Professor, Dept. of CSE, Anantha Lakshmi Institute of Technology and Sciences, Anantapuramu, AP, India.
Email: rsreddyphd@gmail.com

C SHOBA BINDU
Professor, Dept. of CSE, JNTUA University, Anantapuramu, Andhra Pradesh India.
Email: shobabindhu@gmail.com

Abstract

Fault localization is a key phase in the automatic repair of programs because correct identification of program areas that are most closely linked with a fault significantly influences the efficiency of the patching. Most Automated Program Repair (APR) tools use common fault location methods, which are not integrated tightly into the overall program repair process and hence provide only a poor efficiency. Each year, software businesses spend several hours debugging and correcting faults for developers. Automated program repair can decrease debugging expenses. Existing automated repair solutions, such as Genprog, TSP Repair, and Sketch Fix, are highly promising but cannot repair all defects. We analyzed fault localization on automatic repair techniques with fault reduction techniques. The main challenges are to discover code semantically comparable to defective code and integrate it into the faulty program. By extending the time needed to discover a possible remedy, APR performance will be degraded. Furthermore, the correctness of the program repair will be compromised since APR will update fault-free declarations that have a higher repair priority than a real incorrect declaration. In this research, an Effective Model for Fault Localization using Checkpoints for Automatic Program Repair (EMFLC-APR) is proposed that improves the performance of the system and the time for spending faults can also be reduced. The proposed technique provides a feedback loop between the operations for identifying the sources of the issue and the work to generate and evaluate possible remedies. The feedback loop allows partial assessment findings of potential remedies to be used to more correctly discover errors and ultimately lead to processes with enhanced efficiency and effectiveness. The proposed model exhibits better efficiency in fault localization for automatic program repair.

Keywords: Fault Localization, Automatic Program Repair, Debug, Error Correction Time, Faulty Code.

1. Introduction

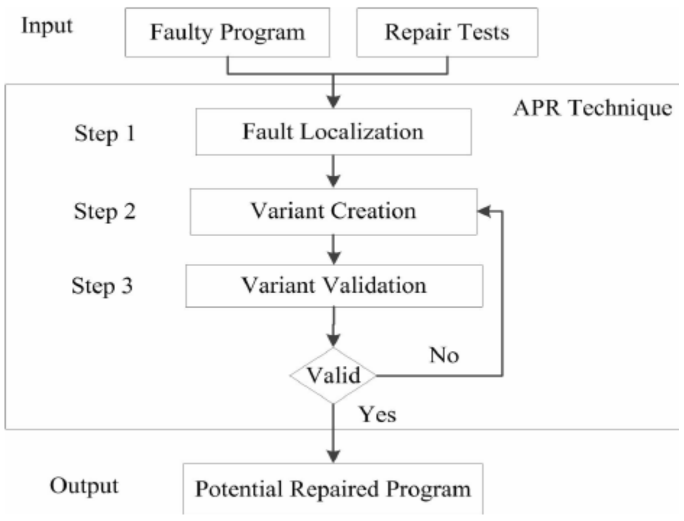
A debugging process includes finding and fixing software bugs. Program repair using automated processes is known as Automated Program Repair (APR), and it promises to reduce debugging costs significantly [1]. Fault location (FL) is used in APR techniques to guide a repair tool and divides the code into code segments with a higher probability of error. When using an APR tool [2], the code that is most likely to have errors can be

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 55 Issue: 10:2022
DOI10.17605/OSF.IO/F3ZCE

changed. Using FL techniques, a suspiciousness rating is calculated for each statement to indicate the probability of an error. The calculated suspicion will be used to build an ordered list of discrete statements by the repair tool [3]. An APR's efficiency, performance, and repair accuracy can all be impacted by the FL technology [4] that is used to execute the repair. APR effectiveness is a problem-solving capacity, whereas efficiency is the time or sequence of operations required to discover a viable repair [5], and repair accuracy reflects the retention of functional requirements by a potentially repaired program [6].

Fault location techniques that are ineffective may lead to incorrect repair decisions by failing to find declarations where faults wait, giving a poor score to the true problem statement, or identifying an excessive number of statements that may contain faults. APR efficiency will be reduced if there are no incorrect statements. Because the APR alters numerous fault-free statements inefficiently before reaching a problematic statement [7], assigning low values to a defective statement has no effect on APR efficiency, but it affects APR competence [8] and causes erroneous repairs. Too many reports, on the other hand, could improve APR efficiency by increasing the likelihood of discovering an available repair [9], but the repair will almost certainly be subpar and wrong. Using a fault locating technique in the worst-case scenario can cause APR's performance to drop because it will mark all assessment focuses as potentially incorrect. The process of fault localization and bug fixing is shown in Figure 1.

Fig 1: Fault Localization Process and Bug Fixing



The number of variants formed by an APR technique to discover various repair outcomes is reduced when using a fault location technique that finds fewer statements or places a problematic assertion at the head of the list of potentially faulty statements (LPFS) [10]. Reducing danger of changing non-fault statements while improving repair accuracy are only two advantages to use APR [11]. It's only been recently that academics have begun

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 55 Issue: 10:2022
DOI10.17605/OSF.IO/F3ZCE

to look into how different technologies, expectations, and modifications to fault localization approaches, as well as validation methodologies for patching, effect APR tool performance.

To identify over-fitted modifications based on their source code attributes, machine learning models make use of contemporary patch validation methodologies [12]. By leveraging information gleaned from software components in a natural language [13], the proposed method aims to improve auto repair procedures. By strengthening the limits placed on potential repairs throughout the repairing and validation steps, the increased fault location should allow APR tools [14] to develop patches for suitable fault locations, resulting in better patch quality.

Retrospective fault placement, as well as localization, are proposed in this model as novel locations for faults that improve precision while also boosting effectiveness and integrating closely with regular automatic program repair methods [15]. Retrospective fault location increases the search space for feasible repairs by giving a more fault location technique. To improve the accuracy of fault localization [16], retroactive fault location makes use of mutation-based fault location. Due to the notoriously long time, it takes to carry out mutation-based fault localization, it's critical to do so as a step in the program's normal repair process. In other words, the retrospective fault localization creates a feedback loop that re-uses the candidate instead of just dismissing them, such that the validation fails [17] to improve the accuracy of the fault location [18]. To improve fault location for future analogous updates, candidate fixes that pass particular tests that fail the originating program that is utilized.

2. Literature Survey

The three automated software repair methods are fault localization, patch creation, and patch validation. Fault localization procedures are used immediately after receiving a bug report to pinpoint the wrong code section analyzed by Gay et al. [1]. The modification of a problematic code fragment following repair criteria based on adaptive computation or code-based collaborations might create numerous potential patches. When evaluating the validity of a candidate patch, regression testing is frequently employed, which includes negative test cases and positive instances suggested by Liu et al. [4]. Until a patch is found, this method can be used indefinitely. If a patch passes all of these tests, it is considered authentic. Recent studies have centered on automated techniques of repair.

A new study area in broadly automated program repair has opened up because of GenProg's use of genetic programming to mend programs with no specifications. This topic has been studied before, however, as evidenced by the existence of earlier calls for contracts to be met for the pre-and post-conditions analyzed by Goffi et al. [6]. There hasn't been any real-world testing of the effectiveness of the JAFF model, which utilizes

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 55 Issue: 10:2022
DOI10.17605/OSF.IO/F3ZCE

an evolutionary approach to correct broken java programs. HTML production mistakes in PHP applications can be automatically resolved with PHP Repair.

Although many faults in real-world Java applications have been fixed successfully with Par proposed by Sergey Mechtaev et al. [18], it is unknown if Sem Fix [27], which employs symbolic execution to correct flawed C programs, scales adequately to large-scale applications. Fixed templates generated from human-written remedies indicate that semantic analysis might be costly. Gene-based algorithms are used in genetic programming to find software programs that are specifically built to accomplish a given task. To evolve populations and find more effective solutions in genetic programming, similar to traditional genetic algorithms, genetic operations such as selection, crossover, and mutation are used.

To automatically and comprehensively patch software maintenance flaws, GenProg is a potential automated program repair solution suggested by Qi Xin et al. [20]. An algorithm in GenProg, which uses genetic programming to create patches, directs the patch generation process. Geng must first implement two crucial components, according to YingfeiXionget al. [15]: a recognition of the resolution and a definition of fitness. The patched program's AST is used by GenProg to illustrate the presentation problem. GenProg's fitness function picks patches with high fitness that pass a large number of test cases based on the results of test cases used to evaluate each patch. These patches will be put to good use in the evolutionary process that moves forward. The only way to know if a patch is good enough is to put it through a series of tests. FL methods produce a suspiciousness score for each expression in the source code, indicating the likelihood that the statement contains a bug. After that, statements are categorized based on how much suspicion they arouse in the listener.

Developers can use the suspiciousness score to prioritize their investigation efforts. However, Spectrum-Based Fault Localization (SBFL) is commonly used in software to compare the behavior of successful execution to an unsuccessful one when it comes to FL. SBFL keeps track of the dynamic characteristics of program execution for each test in the suite. A suspiciousness score is calculated for each statement using SBFL methods based on how many tests pass or fail. Statements that are executed more frequently in the program have a higher suspiciousness score because they are more likely to include errors during a failure run. According to Martin Monperrus et al. [25], many strategies have been proposed to compute proposition suspicion ratings. CURE is a new NMT-based APR technique with three major novelties. First, CURE pre-trains a programming language (PL) model on a large software codebase to learn developer-like source code before the APR task. Second, CURE designs a new code-aware search strategy that finds more correct fixes by focusing on compilable patches and patches that are close in length to the buggy code. Finally, CURE uses a sub word tokenization technique to generate a smaller search space that contains more correct fixes.

3. Fault Localization Model with Checkpoints

For a long time, fault localization and automatic program repair have been integrated. Traditionally, given a flaw in the software, fault localization recommends locations inside the program that may be the source of the bug. Automatic Program Repair then tries to alter those questionable areas to get rid of the issue [19]. Bad fault localization may result in the omission of potential repairs if restrictive, or in the creation of unnecessary work if it is too permissive [20]. According to studies, incorrect fault localizations occur frequently in practice for test-based repair. This identifies the requirement for fault localization [21], which can limit the space of alternatives while still ensuring that plausible causes for an issue are not overlooked.

Let's consider an Input. JP is a Java program made up of classes, and TC is the test case for JP. The tests for JP are classified into two groups: those who pass (TP) and those who fail (TF). It's safe to expect that TP will just feature tests that show how much stronger the program is after doing strength training. Using Fault Localization, pinpointing can be done exactly where in a program an error occurred. Using algorithms based on dynamic and static metrics, each screenshot is given a suspiciousness value. The more suspicious a screenshot appears to be, the higher its score will be.

The suspiciousness levels are calculated from the program code as

$$SL = \sqrt{\frac{\Sigma_{i=1}^{N}(TP_i - \delta)^2 + \max(TF)}{\max(TC) + \max(TP)}} + Th \qquad (1)$$

Here δ is the count of faults that need not to be considered. This is the threshold value considered that is dynamic based on the lines of code.

Fix Generation produces several enhancements to the input program JP for each step in ascending order of suspiciousness. To prevent the suspicious limit condition, the improvements aim to alter JP's behavior. Fix generation generates a list of potential fixes that must be tested before being implemented. Fix Validation looks at each potential repair to determine whether it genuinely fixes the fault that T uncovered. Using fix validation, which has been utilized in traditional automated program repair, all available tests T are performed against the other fix candidate, and only fix candidates that transfer all tests graded according to the unreliability of the snapshots from which they were derived are produced.

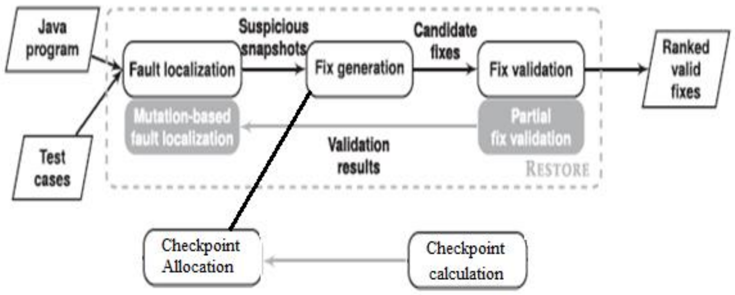
The similarity difference between the fixed and non-fixed bugs are calculated as

$$\lambda = \frac{\Sigma_{i=1}^{N} SL_N + T_h}{(TP - TF + \delta)} \qquad (2)$$

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 55 Issue: 10:2022
DOI10.17605/OSF.IO/F3ZCE

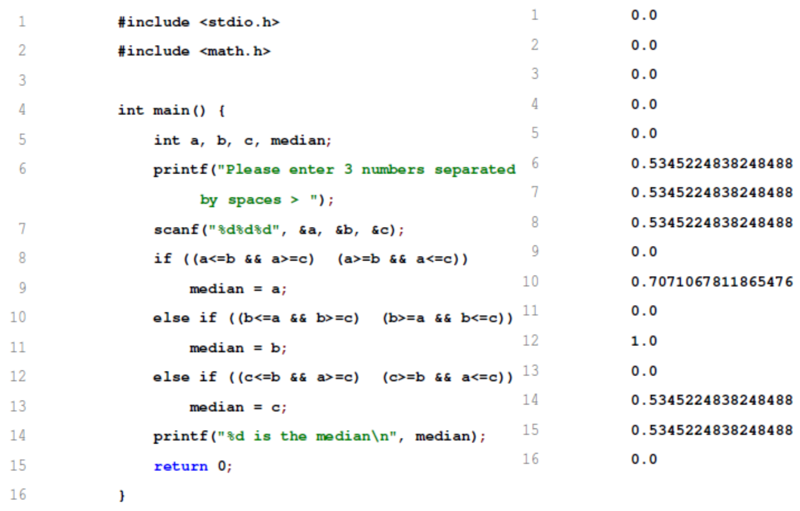
To provide retroactive fault localization, the EMFLC-APR Fix Validation method of bug fix validation is used. To successfully uncover changes in behavior in some stages, partial fix verification uses only a subset of the currently available tests TC to operate on program JP under fix. The bug fixing process is depicted in Figure 2.

Fig 2: Bug Fixing Process



Each patch varies depending on the software's difficulty, manual review might take anywhere from a few minutes to several hours. Authors are also unavoidable when comparing machine patches to human fixes, which is why the manual review approach requires the participation of two writers at a minimum. If a patch doesn't need to be removed, it's considered a correct patch; if it needs to be removed by all methods and no additional investigation has been done, it's considered incorrect; and if it needs to be removed by only one or two methods, further investigation is required. The suspiciousness of every instruction is represented in Figure 3.

Fig 3: Suspiciousness Levels of every Instruction



```
#include <stdio.h>
#include <math.h>

int main() {
    int a, b, c, median;
    printf("Please enter 3 numbers separated
         by spaces > ");
    scanf("%d%d%d", &a, &b, &c);
    if ((a<=b && a>=c)  (a>=b && a<=c))
        median = a;
    else if ((b<=a && b>=c)  (b>=a && b<=c))
        median = b;
    else if ((c<=b && a>=c)  (c>=b && a<=c))
        median = c;
    printf("%d is the median\n", median);
    return 0;
}
```

1	0.0
2	0.0
3	0.0
4	0.0
5	0.0
6	0.5345224838248488
7	0.5345224838248488
8	0.5345224838248488
9	0.0
10	0.7071067811865476
11	0.0
12	1.0
13	0.0
14	0.5345224838248488
15	0.5345224838248488
16	0.0

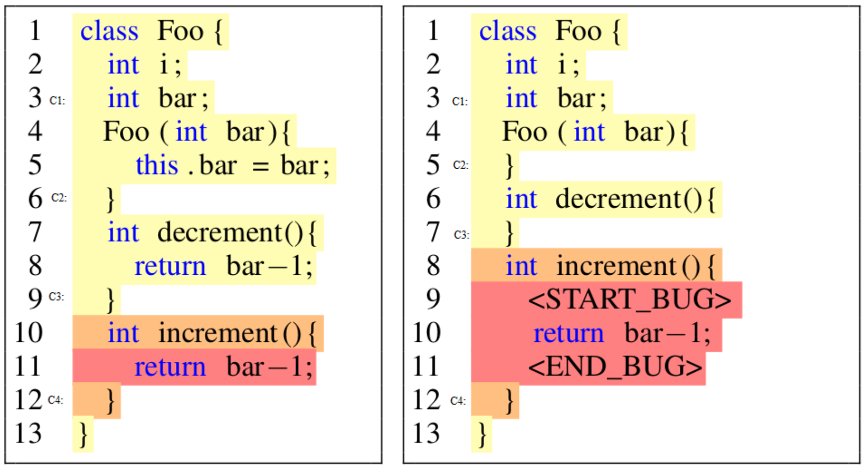
Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 55 Issue: 10:2022
DOI10.17605/OSF.IO/F3ZCE

For each test case TC, the proposed model runs the program and records whether or not it succeeds. Based on the frequency of successful and unsuccessful test scenarios, it updates the suspiciousness score. The suspicious level update is performed as

$$suspiciousness\,(s) = \sqrt{\frac{TF\,(\lambda)}{total\_failed} \times \frac{TP\,(\lambda)}{SL}} \quad (3)$$

The bug identification and checkpoint fixing is represented in Figure4. If statement s is suspect, the suspiciousness score will reflect that. The greater the level of suspicion, the greater the likelihood that s is flawed. A test failure rate is represented by the number of failed tests that are carried out. To sum it up, the total number of failed tests is equal to the sum of all the test failures, both those that do not execute and those that do. S represents the number of suitable testing cases that are executed.

Fig 4: Checkpoint Fixing



```
class Foo {
  int i;
  int bar;
  Foo (int bar){
    this.bar = bar;
  }
  int decrement(){
    return bar-1;
  }
  int increment(){
    return bar-1;
  }
}
```

```
class Foo {
  int i;
  int bar;
  Foo (int bar){
  }
  int decrement(){
  }
  int increment(){
    <START_BUG>
    return bar-1;
    <END_BUG>
  }
}
```

The checkpoints are set to the code that is not having errors. If any occurs in the program the execution process is maintained at a previous checkpoint and then the occurred errors are repaired. The checkpoints are fixed by using the equation

$$\mathrm{Checkpoint}(SL^{N}) = \frac{1}{\mathrm{LOC(JP)}} \sum_{i=1}^{N} \mathrm{TP}(N) + \lambda^{N}_{i,j} + \delta \quad (4)$$

4. Results

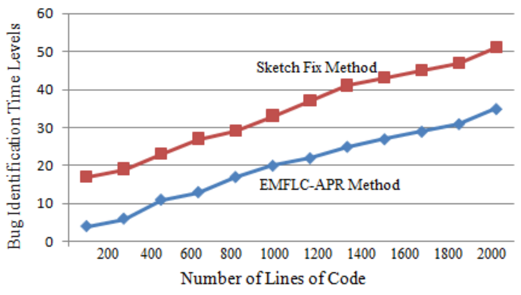
In this research work, an Effective Model for Fault Localization using Checkpoints for Automatic Program Repair (EMFLC-APR) is proposed for improving the performance levels in APR. We evaluated the tests using the Defects4J benchmark that contains 357 defects from 5 open-source Java projects. The proposed model is compared with the

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 55 Issue: 10:2022
DOI10.17605/OSF.IO/F3ZCE

existing Sketch Fix model in terms of Bug Identification Time Levels, Program Repair Time Levels, Fault Localization Accuracy Level and Checkpoint Allocation Time Levels, and Accuracy Levels in Automatic Program Repair.

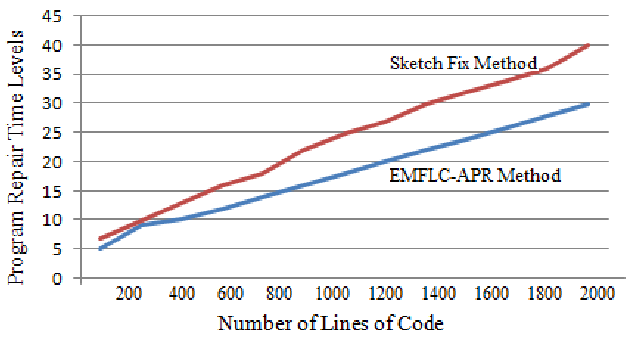
Using an automated bug-fixing system, software issues can be repaired without the need for human participation. Automatic patch generation, automatic bug correction, and automatic program repair are all terms that refer to the same concept. Error, flaw, or fault in software creates an unwanted behavior, such as producing an inaccurate or unanticipated outcome. A program is considered buggy if it has numerous defects or if the bugs severely impair its functionality. The bug identification time levels of the proposed and traditional models are shown in figure 5.

Fig 5: Bug Identification Time Levels



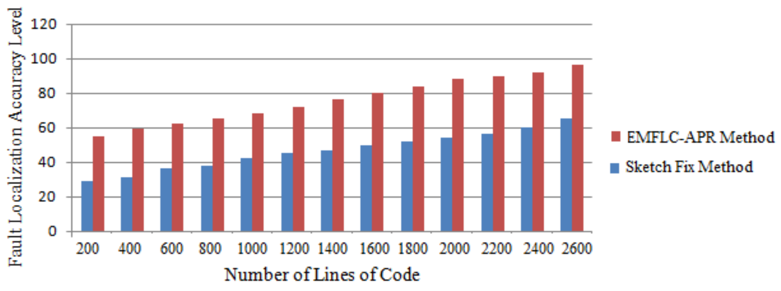
Program repair automation has the potential to reduce these burdensome processes by suggesting software bug fixes that are more likely to work. As an example, a program and a statement of the accuracy criteria that the static program must meet are inputs to these techniques. Test suites are commonly used in scientific research since failing one or more of them indicates the existence of a bug that has to be corrected while passing one or more tests indicates that the behavior should remain as it is. A set of software changes that fixes the bug without disrupting other behavior is the final aim. The program repair time levels of the proposed and traditional models are depicted in Figure 6.

Fig 6: Program Repair Time Levels



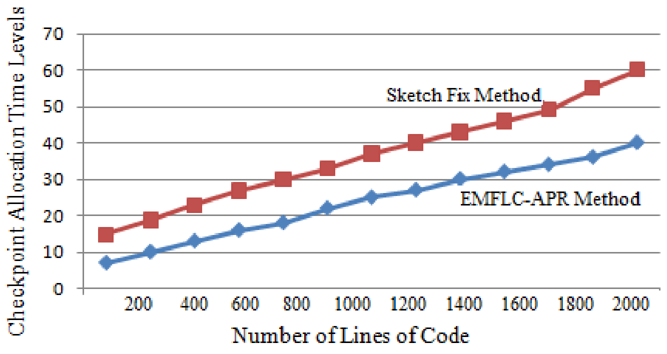
Fault localization is an essential part of fault management systems since it helps pinpoint the exact cause of network problems that have been identified. Unreachable hosts or systems, delayed response, high utilisation, and so on are examples of fault symptoms. The fault localization accuracy levels of the proposed and existing models are represented in Figure 7.

Fig 7: Fault Localization Accuracy Level



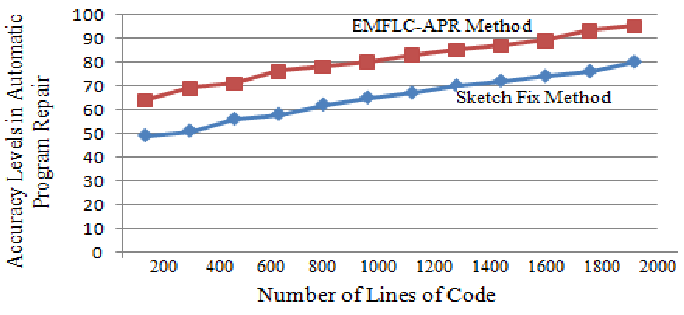
There are two crucial decisions that a user must make when attempting to run a long-running program on a cluster computer system. For the most part, on cluster systems, the user must first decide the number of processors before the calculation begins, and that number cannot be changed after the computation is underway. Processor allocation is straightforward on a system without check pointing. The application should use as many processors as possible to maximize parallelism while also minimizing execution time. It is less apparent how many processors will be allocated when check pointing is enabled. An application can't continue until a failed processor is fixed; otherwise, the system will have to be restarted till it's fully functional again. The checkpoint allocation time levels of the proposed and existing methods are shown in Figure 8.

Fig 8: Checkpoint Allocation Time Levels



The proposed model is accurate in fault localization and automatic program repair by fixing the bugs accurately and quickly. The accuracy levels in automatic program repair of the traditional and existing models are represented in Figure 9.

Fig 9: Accuracy Levels in Automatic Program Repair



5. Conclusion

Automatic Repair tools are imprecise when checked against bug benchmarks and can only produce patches for a small subset of bug types. Repair tools need to be assessed using a variety of bug standards and the types of issues that the procedures are designed to solve, according to recent research. An effective fault localization model for evaluating the capabilities of automatic program repair errors as-yet-undiscovered class of bugs using checkpoints. As a sufficiently general methodology, retrospective fault placement might be included in other start generating program repair methods, possibly with some adjustments. Modern software repair techniques generate a large number of over-fitting patches, and the automatically related test cases can be used to evaluate the accuracy of the patch in scientific investigations. It can correct programs without no passing a bug at all. It is also useful for designing domain-specific test generators to discard erroneous patches, and only a small fraction of instantaneously created test possible conditions is enough to identify incorrect bug fixes in scientific studies. One of our upcoming initiatives is the development of a cutting-edge benchmark that includes useful defects for program repair research. Fault clustering will be used in future work together with other recent advances in fault localization to make fully automated program repair research even more exciting in the years to come.

References

1) G. Gay and R. Just, “Defects4J as a challenging case for the search-based software engineering community,” in Proceedings of the international symposium on Search-Based Software Engineering (SSBSE), Oct. 2020.

2) K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyand´e, “Avatar: Fixingsemantic bugs with fix patterns of static analysis violations,” in Int. Conf. on Software Analysis, Evolution, and Reengineering, 2019, pp. 1–12.

3) K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyand´e, “LSRepair: Live search of fix ingredients for automated program repair,” in Asia-Pacific Software Engineering Conference, 2018, pp. 658–662.

4) K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyand´e,“A critical review on the evaluation of automated program repair systems,” Journal of Systems and Software, vol. 171, p. 110817, 2020.

5) M. Motwani and Y. Brun, “Automatically generating precise oracles from structured natural language specifications,” in Proceedings of the 41st International Conference on Software Engineering. IEEE Press, 2019, pp. 188–199.

6) Goffi, A. Gorla, M. D. Ernst, and M. Pezz`e, “Automatic generation of oracles for exceptional behaviors,” in International Symposium on Software Testing and Analysis (ISSTA), Saarbr¨ucken, Germany, July 2016, pp. 213–224.

7) Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezz`e, and S. D. Castellanos, “Translating code comments to procedure specifications,” in International Symposium on Software Testing and Analysis (ISSTA), Amsterdam, Netherlands, 2018, pp. 242–253.

8) Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clment, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in java programs.2016.

9) S. Mechtaev, J. Yi, and A. Roychoudhury. Angelic: Scalable multiline program patch synthesis via symbolic analysis. In 2016 IEEE/ACM 38thInternational Conference on Software Engineering (ICSE), 2016.

10) Gulsher Laghari, Alessandro Murgia, and Serge Demeyer ANSYMO. Fine-tuning spectrum-based fault localization with frequent method item sets. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016.

11) Xuan-Bach D. Le. Towards efficient and effective automatic program repair. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, 2016.

12) Qi Xin and Steven P. Reiss. Leveraging syntax-related code for automated program repair. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017.

13) Lushan Chen, Yu Pei, and Carlo A. Furia. Contract-based program repair without the contracts. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017.

14) Ripon K. Saha, YingjunLyu, and Hiroaki Yoshida. Elixir: Effective object-oriented program repair. In Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), 2017.

15) Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. Identifying patch correctness in test-based automatic program repair. In Proceedings of ICSE, 2018.

16) Ming Wen, Junjie Chen, rongxinwu, Dan Hao, and Shing-Chi Cheung. Context-aware patch generation for better automated program repair. InProceedings of ICSE, 2018.

17) Xuan Bach D. Le, Ferdian Thung, David Lo, and Claire Le Goues. Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. Towards practical program repair with on-demand candidate generation.In Proceedings of ICSE, 2018

18) Sergey Mechtaev, Manh-Dung Nguyen, YannicNoller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In Proceedings of ICSE, 2018.

Tianjin Daxue Xuebao (Ziran Kexue yu Gongcheng Jishu Ban)/
Journal of Tianjin University Science and Technology
ISSN (Online):0493-2137
E-Publication: Online Open Access
Vol: 55 Issue: 10:2022
DOI10.17605/OSF.IO/F3ZCE

19) Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. Quixbugs: a multi-lingual program repair benchmark set based on th equixey challenge. In Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity Pages 55-56., 2017.

20) Qi Xin. Towards addressing the patch overfitting problem. In Software Engineering Companion (ICSE), 2017 IEEE/ACM 39th International Conference on, 2017.

21) Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. Better test cases for better automated program repair. In In Proceedings of 2017 11th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Paderborn, Germany, September 48, 2017 (ESEC/FSE17),11 pages, 2017.

22) Qi Xin and Steven P. Reiss. Identifying test-suite-overfitted patches through test case generation. In ISSTA, 2017.

23) Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. Test case generation for program repair: A study of feasibility and effectiveness. CoRR, abs/1703.00198, 2017.

24) Github repository of our study is available online.https://github.com/KTH/quixbugs-experiment, 2018.

25) Martin Monperrus. Automatic software repair: A bibliography. ACM Comput. Surv., 51(1):17:1–17:24, January 2018.

26) Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. An empirical study on mutation, statement, and branch coverage fault revelation avoid the unreliable clean program assumption. In Proceedings of the 39th International Conference on Software Engineering pages 597–608. IEEE Press, 2017.

27) Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, 2017.

28) Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on, volume 1, pages213–224. IEEE, 2016.

29) M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in IEEE/ACM International Conference on Automated Software Engineering, Singapore, 2016, pp. 262–273.

30) Koyuncu, T. F. Bissyand´e, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. L. Traon, "D&C: A divide-and-conquer approach to IR-based bug localization," ArXiv, vol. abs/1902.02703, 2019.