Survey Paper

Naveen Patinjaraeveetil Ravikumar
Department of Computer Science and Engineering
North Carolina State University
Raleigh
npatinj@ncsu.edu

Vivek Nair
Department of Computer Science and Engineering
North Carolina State University
Raleigh
vnair2@ncsu.edu

***Abstract*—Software debugging is one of the most expensive,time consuming and tedious among the activities in software development cycle. This means that fully autonomic software debugging techniques is the need to the day. A autonomic debugging technique should be able to guide developers and perform fault localization with minimum human intervention. This generated a lot of interest in the software engineering community and as a result a various methods have been proposed. Recntly, there has been also a push to use statistical methods to use historical data to guide the debugging process. In this article we provide an overview of the techniques and recent advances in this field.**

***Keywords*-Program debugging, software fault localization, testing, code completion, static analysis**

I. Introduction

The work is run by software and a look around is an evidence to that. From the household appliances to the most advanced satellites all are run by software. Given the centrality of software in the modern life of our species and given the propensity of making mistakes, software reliability has become a issues of concern. To reduce the number of bugs in the software attempts are made by spending more resources(testing and debugging contribute to 50% to 80%[ref1] of the development time) and make smart tool which make the process of debugging less tedious and time consuming. With the introduction of concurrent programming, the process of debugging has become more complex. We would also like to draw attention to the fact that over the years, time to market a software has become extremly crutial and development/testing time has reduced. This is another reason why smarter tools are required to circumvent the problem of bugs.

The survey paper aims to review recent contributions in the field of static analysis, bug localization, software repair and code completion. We aim to look at the trends prevalent in the software community regarding the use of these methodologies. There are many static analysis tools available for the community. Some of the papers presented in ICSE14, introduces novel techniques in areas of Static Analysis, Bug localization etc. For Static analysis we are particularly interested in two papers: an automated approach to detect violation[1] and use of the statistical prediction[2]. We are also plan to investigate bug localization, which tries to find bugs in code using the information directly from the bug repository. We would be studying a bug localization technique using the Markov chain logic [3] and explore BOAT [4], which is an experimental platform to evaluate different bug localization techniques. We will also look into the techniques used in the current software repair strategies such as MintHint[5], which automatically synthesizes repair hints. Code completion is another field which is of interest to us. We will focus on Cookbook[6] which provides in-situ code completion using edit recipes learned from examples and CodeHint[7], which provides dynamic and interactive synthesis of repair hints.

II. Body

A. Static Analysis

Static analysis is a method used by the software community to check the code for defects without running the code. Running a static analyzer on the code would help developers to identify problems with the code and would act as a testing tool for the developers. A substantial amount of work has been done in the past to detect defect and vulnerabilities. Lint program is considered to be the first static analysis tool, which enforces or flags style issues such as, naming conventions etc. Though the lint tool is obsolete now, it has paved way for tools which detect defects which lead to security vulnerabilities such as SQL injection and cross-site scripting.Find-bugs is one such tool that looks for coding defects[1].

With all the advantages of static analysis there are various limitations which needs to be addressed before static analysis becomes a one stop solution to the code defect problems. Static analysis cant tell the user whether they have got the requirements wrong. This is because the code doesnt know what the tool needs to do. On the same line of thought, the tool also cant detect mistakes in logic even though mistakes in logic might or might not be caught in the unit testing phase. It has also been pointed out in some previous studies that different static analysis tool return different results which show almost no overlap between them. This means that ideally a developer should run multiple static analyzer on the code base before she can breathe a sigh of relief. There is also a debate whether manual review should be considered rather than using bug finding tools, which are licensed most of the time and is very expensive. A study was conducted by Wagner et. al[16] shows that the tools are strongly tied to the personal programming style and the design of the software since the results differed strongly from project to project. Mantyla et. al[12] points out the bug finding tools found “found more problems with maintainability than real defects than real defects. Testing phase of the tool found fewer bugs than reviews and as mentioned before has no overlap between the bugs found during reviews and static analysis. Wedyan et. al[18] conducted a study, where two static analysis tools were run on two open source projects and the effectiveness of the tools were measured with respect to the actual defects that were fixed by the developers (using the bug reports). The results shows that fewer than 3% of the detected faults correspond to the coding concerns reported by ASA tools. This shows that static analysis tools are good at finding problems that “could happen rather than that “do happen. Johnson et. al[8], conducted a study with 20 developers and found that even though developers think that static analysis tool should be used but the number of false positives, the way the warning is presented among other issues which deters in the widespread adoption of these tools. We cant stress enough on the importance of keeping the the number of false negatives as low as possible. Blackstones formulation succinctly put is as “It is better that ten guilty persons escape than that one innocent suffers.

Software engineering community has recognized static analysis as a potential tool which could reduce the number of defects in a software system. This is an active area of research and we present two such works which shows how researchers are circumventing the problems by by building better tools.

Static analysis is performed whenever a new version of a software is released. Most of the time programmers are only interested in knowing which are the new issues which was introduced in the code base after the previous version was released. Which means programmers are only looking for warning deltas between the versions. This would expose new issues and the issue which have not been resolved since the previous version. As mentioned earlier time required to run the Static analysis tools on a fairly large code base and then post processing the results to determine issues can be very time consuming and tedious. Intuitively a programmer would want a static analysis tool which could proactively identify issues and report it to programmers.

Radhika et al.[15] proposed a similar tool and build a prototype, which would help the programmers to find new bugs proactively with the scope only being the new set of code and should avoid the overhead of running multiple tools on the entire code base. The proposed tool identifies issues by performing analysis of code across version at the time of commit of each change in the new version. Authors have also implemented a learning system, which uses the expert knowledge to improve accuracy and spread of the system. Given the length of the report generated by the current state of the art static analyzer, developers need to manually go through the output and find out the issues which needs attention and the others which are warning. Using this intuition, authors developed expert system, which can identify the patterns and context in which the issues occur. The system can use this expert knowledge to improve its prediction accuracy. In addition of accurately identifying issues, the system returns a ranked list of issues based on several factors such as confidence level, criticality etc. Authors evaluated the system by running it on three projects from the Siemens CT DC AA Healthcare sector. It was observed that there is a reduction of about 15% of false positives and detects “considerable number of false negatives (since the system has been learning from the expert). The results also showed that system reduced the time to analyze the results went down by 24%. The tool reduces the time spent running static analysis tools on the code base and encourages developers to use it.

Though this is a very novel way to solve the problem there are few limitations which we think should be taken care of. The authors have mentioned in the paper that the system would work only if there are atleast 3 version of the software under development. Though this doesnt seem to be a big problem, it would be very useful if the system could learn from other projects of the same domain. This would make sure that the system can be used even in initial stages of the project. This system also introduces bias (similar to all expert systems) to avoid this we would recommend to include some form of triangulation, which can remove most of the bias.

Rahman et al. [14] in the paper compares two different techniques: statistical defect prediction and static big-finding, and try to find synergies between the two approaches. Static bug finding(SBF) is very similar to static analysis. Static bug finding ranges from a simple pattern checking to a detailed static analysis. FindBugs[ref], a static bug finding tool, has both pattern matching,which scales well, as well as static analysis build into it. Defect Prediction(DP) on the other hand uses historical data to develop model, which simulates the behaviour of the software, to predict where the defect can occur.

Though these approaches were developed in complete isolation, they try to solve the same problem. But, it is very difficult to compare the performances of these approaches. DP tools are easy to implement, agnostic to language, platform etc. The easy availability of metric required for the tools have made DP tools to be more valuable than the other available approaches. Coarse granularity(file) is the Achilles heal of this technique. SBF on the other hand is language- and platform specific, requires a separate build process but has a finer granularity.

Authors introduce a new metric to compare these two approaches: comparability bugginess identification. This metric has two parts to it, zonation and measurement. Zonation refers to the granularity at which the tool reports errors whereas measurement deals with how spread out the error is. Based on this metric author reports results from a comparative study of two SBF tools(PMD and FindBugs) and DP based on a logistic model on five open source models and tries to answer if SBF tools compare with DF prediction and if there exists any synergy between them.

The authors find that SDF tools do better than PMD in most of the cases which DP doesnt fare well again Findbugs. Authors find that SBF warnings based on DP improves upon the native SBF priority while vice-versa is not true. [TODO: Talk about threats to validity]

B. Bug Localization

Testing is the fundamental part of the software development project. A lot of effort is spent on testing a program and bug removal(introduction) happens continuously throughout the development process. During program debugging, fault localization is required to figure out which part of the program is buggy. Static Analysis, Machine learning based methods, model based method, data mining based method are just a few to name. Fault localization can be viewed more as an art rather than a automated mechanical process. Techniques like static analysis helps us to narrow down the scope of the search domain to a particular method or a file. Certain methods [5][19] only selects a single failed test case and a single successful test case to locate bug. These methods fail to leverage other information to make a more informed choice. Alternate methods [10][11] uses combination of data from multiple failed and successful test to locate a bug. A general rule of thumb for the fault localization techniques is to provide a high "suspiciousness to the code that contains bugs low "suspiciousness to part of the code which contains less bugs. This would help to generate a ranked list of bugs, which can then be used by developers to debug effectively and reduce time required for debugging. There is a rich collection of literature that explores various ways to facilitate fault location, but they are far from perfect. As more and more sophisticated techniques are being developed we need to note that the software systems are being increasingly complex which means that the challenges posed are also growing. We would look at two methods which increases our understanding about fault localization.

Zhang et al. points out that most coverage based approaches has two common limitation. Firstly they work with an underlying assumption that the buggy statements are sequential and separate. These bugs ignore the inter-statement relation in a program ie. a error in one statement effects the following statements. Second, these method do not exploit multiple source of information to narrow down the scope of the fault ie. as mentioned before only single failed and pass test case is considered for fault localization. The authors hypothesize joint inference to exploit global information as well as historical information can open up new avenues for the research of bug localization.

Authors propose a new bug localization technique based on Markov logic(MLN). MLN when applied to a bug localization problem permits to combine different information source into a comprehensive solution. Authors define MLN has a first-order knowledge base with a weight attached to each formula. Which means rather than having hard constraint like a first-order knowledge base, MLN softens these constraints, so that if the world violates any of the constraints then it becomes less probable rather than impossible. Authors use three program features to predict a bugs location namely statement coverage, program structure and prior bug knowledge. With MLN, one can encode the program features into a joint inference system and conduct inference. The model contains first order formula with predicates and negations. After the program features have been encoded it is important to train the MLN. Authors use the Alchemy toolkit for this purpose.

Authors build a prototype(MLNDebugger) which takes a set of passed and failed tests as input and outputs a list of suspicious statements. This prototype was evaluated using four subject programs from the Siemens suite and was tested against Tarantula which is a state of the art statement based approaches. Their experimental results show that MLNDebugger consistently outperform Tarantula. Authors also report that their tool is fairly portable and doesnt require enormous amount of computing resources.

Though the results look promising there is one aspect which we think is a red flag. It is mentioned in the paper that the current inference program assumes that there is only one bug in the tested program and argues that this is a common practice in modern software development. We dont think this is true since a lot of developers work with legacy systems and assuming that a developers starts out with a bug free system is a big assumption. There needs to be more work to get this working.

With all the approaches we can see that it is very difficult to compare these techniques. This is because the techniques are tested on different datasets and is hard to compare their effectiveness. Since, the datasets are not made publicly available, it is often hard to replicate their results. Then to top it all some of the techniques have been used in a small number of subsets. To circumvent this problem Wang et al.[17] proposes a platform,BOAT that can be used to compare various techniques and reduce the threat of external validity.

The platform build by the authors consists of three components: data collection component, local debugging component and remote execution component. The data collection component processes the project and collect all the data related to the project (details of the bug from the bug tracking system, delta between files before and after the fix). The local debugging component allows the users to interact with BOAT in the local environment and configure the set up. After the system is configured, jobs are specified and run by the remote execution component. An email is sent out by the system to inform the user about the progress/failure of the system.

BOAT as a web based platform would enable researchers to compare and replicate various approaches over a large dataset of bug report. Authors have also done considerable amount of work to collect considerable number of bug reports from various projects. Though authors have claimed that they have made their tool publicly available, the link is not active.

C. Code Completion

Code completion is a feature that enables the programmer to find the suitable code that needs to be used based on some static analysis. Auto complete is one such feature found in Eclipse IDE wherein the API that need to be used are provided as recommendations. This is very easy to use and reduces the load on the developers as we do not need to remember the names of various method calls. This is also helpful if we are importing an unknown API and is not aware of the different methods in the same.

It is seen that many of the tools rely on static techniques for finding the code fragments. We are surveying 2 papers on code completion. The first one is CodeHint: Dynamic and Interactive Synthesis of Code Snippets[4] and the second one is Cookbook: In Situ Code Completion using Edit Recipes Learned from Examples[6]. These 2 techniques use novel ideas which can be integrated in the IDE that we use today and the results are amazing. We will now look into these 2 techniques in more detail.

[4] suggests that many of the code completion tools that are used today rely heavily on the static techniques for finding code fragments. The CodeHint uses a dynamic(giving accurate results and allowing programmers to reason about concrete executions), easy-to-use and interactive method(allowing users to refine the candidate code snippets). It is seen that the search and autocomplete have limitations when using a new or unknown API. We need to have the static information about the return calls to use them and usually generates tiny code fragments. The CodeHint approach generates a code fragment at a programmer chosen location at run time while executing a specific concrete input. The main advantages of CodeHint above others include running in dynamic context unlike the static which enables the filtering of the desired results. It supports a wide variety of specifications and also interactive letting user give inputs.

In summary the main contributions of CodeHint are:

1) A new dynamic method for synthesizing code that is easy to use as well as interactive
2) An efficient algorithm that exploits the dynamic context to generate candidate statements that can include advanced features of the host language such as I/O, reflection, and native calls.
3) An implementation as a Eclipse IDE plugin that synthesizes the JAVA code as well as Android programs
4) Evaluation based on 2 user studies

We will now look briefly into the working of the CodeHint. Imagine a user who writes the code given in Fig1 and wants to write the code at line 6 to find the menu bar for the window that contains the graphical tree and store it in the variable o. We know that the Menubar is represented by a JMenuBar object but there is no information as to how this object is obtained. This is where the CodeHint comes useful. We will now shed some light on how a programmer can use the same. the user can set a breakpoint after line 5 (e.g., near the comment on line 6) and run the program to that breakpoint, which is the current context. He can also provide the specification o' instanceof JMenuBar to CodeHint called as the pdspec. Given this query, CodeHint will begin a search for expressions that it can assign to o to try to satisfy the pdspec. This iterative search will start with local variables and generate larger expressions with operations such as addition and method calls. CodeHint will evaluate these expressions in the current context, undoing side effects as they occur, to enable it to get precise results that satisfy the users pdspec. A probabilistic model is also used to make sure that upto 5 best results are displayed to the user. Once the results are obtained the user can choose one of them that best satisfies his requirement. At this point the user can add few more testcases and eliminate some of the results or choose one of them.

It can also be used to detect the clicks on a graphical tree of elements. In Java objects have a toString method, which can be used to find the object which initiated the click event by checking if dspec o .toString().contains(”Alice”), who initiated this. We can now check for different element ”Bob” by clicking on it. Also when he does a NULL click, we can evaluate the candidate expressions to point to the one that evaluates pdSpec o = NULL , it eliminates all but one candidate and find the correct code: o = ((JTree)tree).getPathForLocation(x, y);

It can also be used in the skelton which CodeHint will use to guide its search. Specifically, we can enter the skeleton o.getPathComponent(??) where the ??represents the missing portion of the code. More information about the pdspec and the sythesis algorithm used is seen in [4].

2 user studies and an empirical evaluation were conducted to evaluate the usability of CodeHint. The empirical evaluation clearly showed that CodeHint is sufficiently scalable.

The first study focused on constrained single line code edits and the second focused on larger open-ended tasks and used an improved version of CodeHint. The evaluation was done based on the following measures:

1) Task completion time: Time taken to either complete or abandon a task.
2) Task completion rate: Percentage of tasks users successfully completed.
3) Code quality: Number of bugs in participants task code.
4) Tool choice: Fraction of tasks in the choice condition for which participants opted to use CodeHint.

The studies clearly showed that CodeHint significantly improves programmer productivity.

[6] introduces an another code completion strategy that uses the edit recipes learned from examples for code completion.It clearly has significance compared to other methods that leverages on the pre-defined templates or match a set of user-defined APIs to complete the rest of changes. The other techniques that are currently present provide simple quick fix for API method calls. Many of the older code completion strategies rank other methods higher based on the patterss found in the version history or code base[2]. Some of them make use of the edit examples but needs the developers to manually supply the input API usage patterns[13] whereas some use the editing pattern of the user[3] but limited to the predefined refactoring operations by the IDE.

The proposed code completion, scheme is called CookBook where developers can define custom edit recipesa reusable template of complex edit operationsby specifying change examples. This helps the user to define new recipes and store them in an XML file for easy sharing of the library.Cookbook matches a developers edit stream to the individual recipes context and edit operations and ranks suitable recipes in the background.

When a user provides one or more method-level change examples by selecting their old and new versions, Cookbook generates an abstract edit recipe that shows the most specific generalization of changes demonstrated by the change examples. This edit recipe then can be applied to different target contexts where control and data flow contexts match but use different type, method, and variable names. Cookbook performs realtime matching of the incoming edit stream against the library of edit recipes and ranks them with a confidence score. Once the edit recipe is chosen then Cookbook concretizes the template edits to current context and apply rest of the changes.

Once users have a collection of recipes in Cookbook, their edit operations are matched with the recipes frequently so that any potentially applicable recipes are found as early as possible and listed for developers to select and automatically complete the rest of editing. Cookbook only requires the developers to provide one or more examples as input fdefine the new edit recipe. This can either be provided by the developers or taken from the code base.

The above figure shows the 2 recipes that is being used one being the Comment Mapper Recipe(Fig) and the other one the Event handler reciper(Fig). These are both saved to the shared recipe library. The (Fig) shows the code change that the programmer is making and maps this to the edit recipe. The change made is mapped to Comment Mapper Recipe and the auto complete is done based on that.

The main steps that are involved in the CookBook strategy are:

1) Recipe creation
2) Recipe Matching with Edit Stream
3) Recipe Ranking
4) Recipe Application

Cookbook was evaluated using 68 systematic changed methods drawn from the version history of Eclipse SWT. Results show that overhead of recipe matching in real time is imperceptible, and Cookbook is able to narrow down to a single most suitable recipe within 8.2 keystrokes on average. Cookbook is highly accurate, producing results 82% similar to the expected edits in the evaluation data set. Cookbook is able to narrow down to the most suitable recipe in 75% of the cases. It takes 120 milliseconds to find the correct suitable recipe on average, and the edits produced by the selected recipe are on average 82% similar to developers hand edit.

D. Software Repair

Debugging is a very important task for maintenance of the products, so we need to give more emphasize towards the program repair task. Although there are many tools available now they are having many limitations. Some of the limitations include the existence of the specification for the program being debugged and the repair space being very large existing techniques see a bigger search space.

MintHint[9] is a novel technique for the program repair which is a departure from the current techniques that is being used today. It is different from the approach used by the common repair tools. Most of the repair tools use the traditional fault localization information alone. MintHint does not try to find a complete repair unlike the other tools. It aims to to generate repair hints that suggest how to rectify a faulty statement and help developers find a complete, actual repair[9].

MintHint operates in 4 steps as shown in the high level diagram above[Fig]. The first step identifies the potential faulty statements using a fault localozation strategy. Thus, a list of fault statements are obtained. The fault locatization strategy used here is Ochiai approach[7]. Then it runs the hint generation algorithm in each of the fault statements obtained.

Next step involved the derivation of state transformers. Since there are no inherent specification present, this is obtained using the test suite. Basically, the specification is a function defined for all program states that reach the faulty statement in the given test suite and produces right output for each of them.

Ranking of expressions is done in the next step during which the repair space is searched for expression whose value over the input state of the state transformer are statistically correlated to the output states. The partial list of expressions are ranked then. Synthesis of repair hints is prepared after analysing the list using the pattern matching . The hint is either simple or compound one. After the hints are generated it is prioritized and given to the developers.

MintHint overcomes the limitations of the other approaches by:

1) it does not rely on a formal specification; it instead derives an operational specification (i.e., a state transformer) from the test cases available.
2) approaches that aim at deriving complete repair typically use equality with the state transformer (or an analogous entity) as a criterion for selecting a candidate repair.The statistical correlation used in MintHint is a more relaxed and robust notion than equality and can thus be more effective in identifying which expressions are likely to be part of the repaired code; this allows MintHint to synthesize more general repairs and to be effective in the presence of incomplete data or even imperfect data (i.e., state transformer mappings that do not arise in execution of the fault-free program).
3) Since MintHint looks for building blocks of repair (rather than the complete repair itself) and then combines them algorithmi- cally to generate compound hints, it can generate useful, actionable hints even when exploring an incomplete repair space.

Evaluation consists of two main parts:

1) a user study that assesses whether MintHint can improve developers productivity, and
2) an empirical study which applies MintHint to several faulty versions of three Unix utilities to further assess the effectiveness of the approach.

The study involved 10 users and consisted of a control phase and an experimental phase. In both phases, each user was provided with a single repair task along with fault localization information and a test suite. In the experimental phase, in addition, the users were given repair hints generated by MintHint. Without therepair hints, only 6 of the 10 users completed their task within 2h. With repair hints, all 10 users could complete their task within the same time limit. Moreover, the tasks completed in both phases were completed over 5 times faster by the users who used MintHints repair hints.

In addition, MintHint was evaluated on a total of 11 faulty versions of Unix utilities sed, flex, and grep . Symbolic execution timed out for one of them where For 7 of the remaining 10 faulty versions, MintHint generated hints that immediately led to a repair.

When debugging, developers productivity improved manyfold with the use of repair hintsinstead of traditional fault localization information alone.

III. Conclusion

The conclusion goes here. this is more of the conclusion

Acknowledgment

We would like to thank Dr. Christopher Parnin for his support throught the course of this paper. Standing on the shoulders of gaints, we would also like to thank various authors of the referred papers for their hardwork.

References

[1] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, 2008.

[2] M. Bruch, M. Monperrus, and M. Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[3] S. R. Foster, W. G. Griswold, and S. Lerner. Witchdoctor: Ide support for real-time auto-completion of refactorings. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 222–232. IEEE Press, 2012.

[4] J. Galenson, P. Reames, R. Bodik, B. Hartmann, and K. Sen. Codehint: dynamic and interactive synthesis of code snippets. In *ICSE*, pages 653–663, 2014.

[5] L. Guo, A. Roychoudhury, and T. Wang. Accurately choosing execution runs for software fault localization. In *Compiler Construction*, pages 80–95. Springer, 2006.

[6] J. Jacobellis, N. Meng, and M. Kim. Cookbook: in situ code completion using edit recipes learned from examples. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 584–587. ACM, 2014.

[7] T. Janssen, R. Abreu, and A. J. van Gemund. Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 662–664. IEEE Computer Society, 2009.

[8] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.

[9] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: automated synthesis of repair hints. *arXiv preprint arXiv:1306.1286*, 2013.

[10] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM, 2005.

[11] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff. Statistical debugging: A hypothesis testing-based approach. *Software Engineering, IEEE Transactions on*, 32(10):831–848, 2006.

[12] M. V. Mantyla and C. Lassenius. What types of defects are really discovered in code reviews? *Software Engineering, IEEE Transactions on*, 35(3):430–448, 2009.

[13] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen. Graph-based pattern-oriented, context-sensitive source code completion. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 69–79. IEEE Press, 2012.

[14] F. Rahman, S. Khatri, E. T. Barr, and P. T. Devanbu. Comparing static bug finders and statistical prediction. In *ICSE*, pages 424–434, 2014.

[15] R. D. Venkatasubramanyam and S. Gupta. An automated approach to detect violations with high confidence in incremental code using a learning system. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 472–475. ACM, 2014.

[16] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Testing of Communicating Systems*, pages 40–55. Springer, 2005.

[17] X. Wang, D. Lo, X. Xia, X. Wang, P. S. Kochhar, Y. Tian, X. Yang, S. Li, J. Sun, and B. Zhou. Boat: an experimental platform for researchers to comparatively and reproducibly evaluate bug localization techniques. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 572–575. ACM, 2014.

[18] F. Wedyan, D. Alrmuny, and J. M. Bieman. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*, pages 141–150. IEEE, 2009.

[19] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM, 2002.