Int J Software Informatics, Volume 8, Issue 1 (2014), pp.1–17 Tel: +86-10-62661040
International Journal of Software and Informatics, ISSN 1673-7288 http://www.ijsi.org

Email: ijsi@iscas.ac.cn, ijsi2007@gmail.com

Recent Progress in Software Testing, Debugging and Analysis: A Survey

Dan Hao and Hong Mei

(Key Laboratory of High Confidence Software Technologies (Peking University),
Ministry of Education, Beijing 100871, China)

Abstract This article is a survey on the recent progress of some hot topics of software engineering. The survey is based on the review on the papers of three premier conferences on software engineering from 2012 to 2013. Through the quantitative analysis on these papers, there are three hot topics identified, software testing, debugging, and analysis. Focusing on these three topics, this article summarizes some new achievements in these fields, analyzes the characteristics of these works, and points out some directions for future research.
Key words: software testing; software debugging; software analysis

Dan H, Hong M. Recent progress in software testing, debugging and analysis: a survey. *Int J Software Informatics*, **Vol.8, No.1 (2014): 1–17. http://www.ijsi.org/1673-7288/8/i180.htm**

1 Introduction

Software engineering concerns with the development of software with high quality and productivity, and is an important direction of computer science. Among various topics in software engineering, this article presents a survey on the recent progress of software testing, debugging, and analysis, which have been identified as three main hot topics of software engineering.

In particular, this article presents a study on three premier conferences (i.e., International Conference on Software Engineering (abbreviated as ICSE), ACM SIGSOFT Symposium on the Foundations of Software Engineering (abbreviated as FSE), International Conference on Automated Software Engineering (abbreviated as ASE)) on software engineering from 2012 to 2013. Borrowing the procedures of a

Funding by the High-Tech Research and Development Program of China under Grant No. 2012AA011202, the National Basic Research Program of China (973) under Grant No. 2011CB302604, the National Natural Science Foundation of China under Grant No. 61121063, 91118004, 61272157, and the Key program of Ministry of Education, China under Grant No. 313004.
Corresponding author: Hong Mei, Email: meih@pku.edu.cn
Received 2014-04-11; Revised 2014-04-18; Accepted 2014-04-18; Published online 2014-04-29.

systematic review, this study collected the research papers from these conferences, and classified these papers based on their topics.

The statistics of these papers is summarized by Table 1, which shows the number of papers within each topic accepted by each conference. The last three rows of Table 1 present the numbers of papers on these topics. From the last column, each of these topics (i.e., software testing, software debugging, and software analysis) contains at least 10% of the papers published in the recent two years, which is much higher than the percentage of any other topics.

Table 1 Basic Statistics[1]

Conference	ICSE 2013	FSE 2013	ASE 2013	ICSE 2012	FSE 2012	ASE 2012	Summary
Total Number	85	51	43	87	34	27	327
# Software Testing	13	10	7	8	5	3	46
# Software Debugging	11	2	3	11	1	5	33
# Software Analysis	7	10	5	14	7	1	44

Among the three hot topics, software testing aims to identify faults by executing the target software, whereas software debugging aims to identify the location of faults and correct the faults. These two activities are very important to assure the quality of software, and thus have attracted many researchers in the past. Software-engineering data collected during software development and runtime contain useful information. Therefore, many researchers focus on analyzing these data (including documents, bug repositories, version histories, runtime log and so on) to aid and optimize current and future software projects. According to the definition of software, a software system consists of programs, documents, and relevant data. In this article, software analysis is used to cover program analysis and software-engineering data analysis. Software analysis, especially program analysis, can be viewed as hammers to solve various problems in software engineering, including software testing and debugging.

The remaining of this article is organized as follows. Sections 2, 3, and 4, survey the recent works on software testing, debugging, and analysis. Section 5 concludes this article and gives further discussion. Figure 1 presents the topics this article consists of.

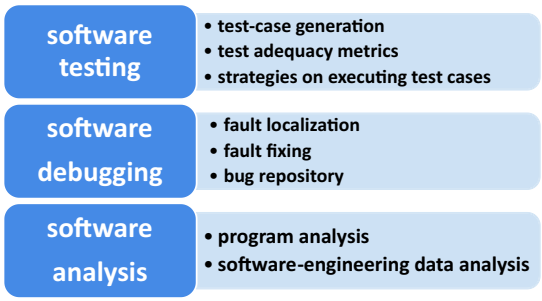

Figure 1. Organization of this article.

[1] Following the procedures of a systematic review, we gave some keywords (e.g., test, bug, API) to describe these papers, and then clustered the papers based on their words.
The actual number of papers within each topic may be larger than the number in this table, because some papers of these topics may be ignored due to the authors' carelessness.

2 Software Testing

Software testing aims to guarantee the quality of software by running test cases on the target software and observing whether any faults are revealed. Due to its importance in ensuring software quality, software testing has been widely studied in the literature of software engineering. Although software testing includes many important research topics, such as test-case generation, test repairs, regression testing, and so on, due to space limit, this article concerns with only the works in Table 1, which do not cover all the topics in software testing. For ease of presentation, this article classifies most of these works into test-case generation, test adequacy metrics, and strategies on executing test cases, which will be presented by the following sections.

2.1 Test-case generation

As software testing is recognized as an expensive activity in software development, many works in software engineering focus on automating software testing. Before running test cases on the target software, it is necessary to prepare high-quality test cases for the target software. As it is costly for developers to manually write test cases, various approaches for automated test-case generation have been proposed in the literature.

Most existing test-case generation techniques generate test cases by analyzing (e.g., using symbolic execution) the software under test. Besides these test-case generation techniques, random test-case generation attracts much attention due to its low cost. Although test-case generation has been studied for long, some challenges still exist in this domain. For example, most existing test-case generation techniques cannot well deal with arrays and pointers, loops within a program. Random test-case generation is less effective in terms of fault detection when comparing with other test-case generation techniques due to the possible biased coverage on the input space.

In the past two years, some researchers focused on addressing the problems in existing test-case generation techniques and have achieved some improvement. In particular, as coverage is the main problem in random test-case generation, Randoop[1] was proposed to alleviate the problem in 2007, which is generally very effective in the early stage of testing. However, as it has other practical issues, e.g., hard to cover deep object interactions or branches, hard to be applied to C/C++ programs, Garg et al.[2] proposed to use concolic execution with feedback-directed random search in a testing framework in 2013.

Some works focused on test-case generation for some specific applications or software, e.g., multithreaded software, web applications, and mobile applications. In particular, although test-case generation for multithread software has been studied before, it is time-consuming to generate multithreaded test cases. To reduce time cost in generating multithreaded test cases, Nistor et al.[3] proposed an automated random generation technique BALLERINA, in which each thread executes a single randomly selected method. For software with complex inputs, e.g., browsers or compilers, constraint-based generation is widely used to generate test cases, whose major challenge lies in the scalability. To alleviate this problem, Zaeem and Khurshid[4] proposed to use recursive predicates to represent constraints and then

solve the constraints by using dynamic programming. That is, the complex inputs are divided into several smaller inputs with the same structure. To perform functional testing on enterprise applications efficiently, Thummalapenta et al.[5] proposed an automated technique to generate test cases driving a web-based application on only interesting behaviors, not the large number of behaviors. Different from existing test-case generation works, the challenge of this work lies in covering the interesting behaviors by creating test cases. As mobile apps become increasingly prevalent, researchers like Machiry et al.[6] proposed to generate relevant test inputs to unmodified Android apps.

Besides these technical works on test-case generation, some researchers investigated other issues by using test-case generation as a sample domain. For example, Fraser and Arcuri[7] questioned the choice of employed artifacts in case studies of software engineering and thus conducted an empirical study on the choice of artifacts in test-case generation. Similarly, Xiao et al.[8] focused on loop problems and thus investigated the presence and influence of loop problems in some specific domains, e.g., test-case generation.

In summary, test-case generation for traditional software has slight improvement. Most of these test-case generation works aim to generate test cases for some specific applications or software. Moreover, most of its improvement is resulting from the improvement of other techniques, e.g., symbolic execution. As it becomes more and more difficult to solve some long-history challenges in test-case generation, it is meaningful to switch to specific applications or software in order to ease the challenges in test-case generation.

2.2 Test adequacy metrics

Due to limited resources (e.g., developers' efforts and time), it is impossible to run all enumerable test cases on the target software. Therefore, in the practice of software testing, testers usually generate a limited number of test cases and test the software by the test suite consisting of these test cases. To learn whether a test suite is sufficient, it is necessary to measure the efficacy of test cases within a test suite. However, in practice it is impossible to measure the efficacy of test cases based on the number of faults they reveal due to the following reasons. First, as the number of faults actually in the software under test is unknown in practice, testers have no idea whether existing test cases have revealed all the faults. Second, in some testing scenarios (e.g., test-case generation and regression testing), it is even impossible to know whether existing test cases may reveal faults within the software under test. Therefore, many approaches have been proposed on defining test adequacy metrics, which are widely used in test-case generation, selection and so on. In particular, to generate effective test cases, test-case generation techniques usually specify some test adequacy metrics to achieve when generating test cases, whereas test-case selection, reduction, and minimization techniques usually use some test adequacy metrics as criteria to select test cases.

In the last two years, researchers have proposed some novel criteria based on existing ones. Modified condition/decision coverage (MC/DC) criterion is an important adequacy metrics, which has been evaluated to be effective in some NASA projects. However, as the effectiveness of MC/DC relies on the syntactic

structure of the program under test, Whalen et al.[9] proposed an observable MC/DC coverage criterion, which combines the coverage of decisions required by MC/DC with a path condition so as to increase the likelihood of detecting a fault. The observable MC/DC coverage criterion has been evaluated to be much more effective than MC/DC both in fault-detection and in robustness to change structure. Based on the widely used branch coverage criterion, Hassan and Andrews[10] proposed a family of coverage criteria, Multi-Point Stride Coverage, which generalize branch coverage by branch tuple coverage and have been evaluated to be more effective than the definition-use coverage criterion. Besides these general test adequacy criteria, Tasharofi et al.[11] proposed three schedule coverage for Actor programs, which are concurrent programs.

In summary, most works on test adequacy metrics are improvement on existing test adequacy metrics, rather than totally novel metrics. As most existing test adequacy metrics are usually based on program structure and the structure of a program has been fully studied, studies on test adequacy metrics are very mature. Therefore, it may be more useful to study test adequacy metrics based on other information rather than program structure since in many scenarios (e.g., web application) the source code of a program under test is not available.

2.3 Strategies on executing test cases

To guarantee the quality of the software under test, it is ideal to prepare and run all the test cases of a program. However, considering the input space of the software and various settings of its configuration, the number of test cases for the software usually is so large that it is impossible to run all the test cases. Therefore, due to such practical concerns, it is important to determine the proper subset of test cases and the execution order of these test cases, which are viewed as strategies on executing test cases.

Test-case reduction and prioritization, firstly proposed in the literature of regression testing, aims to reduce the number of test cases and schedule the execution order of test cases. Before 2012, a large number of works on test-case reduction and prioritization were proposed, most of which explored novel algorithms (e.g., genetic algorithm) to improve effectiveness, combined with other factors (e.g., running time) in reducing or prioritization test cases, or empirically studied the existing techniques.

In the last two years, more and more works on test-case reduction and prioritization focus on configuration testing[12-14] or other domains (e.g., fault localization[15]) rather than traditional testing or regression testing[16]. In particular, Zhang et al.[17] proposed a spectrum of strategies between the total and additional strategies, which are widely used in the evaluation of new test-case prioritization approaches due to the effectiveness of the additional strategy. According to Zhang et al.'s work[17], the works on test-case prioritization have become mature enough so that little improvement exists in the current research direction. As most software systems are highly configurable and have an increasing number of configuration options, which will definitely increase testing difficulties, researchers began to investigate the possibility on reducing the number of configuration option combinations. To alleviate this problem, combinatorial interaction testing (CIT)

selects an interaction strength t and computes a covering array containing all t-way configuration options. As a complement to existing works, Blue et al.[12] proposed greedy algorithms to minimize an interaction-based test suite, whose effectiveness may be improved by applying existing test-suite reduction techniques. However, as CIT may unlikely generate important high-strength interactions, Song et al.[13] proposed an algorithm iTree to identify sets of configurations that include important high-strength interactions missed by CIT. Petke et al.[14] identified the constrained prioritized interaction testing problem for higher strengths. Generally speaking, the preceding techniques on configuration testing are similar to traditional test-case reduction and prioritization in regression testing to some extent.

In summary, comparing with the works before 2012, more and more researchers began to focus on configuration testing, especially on presenting various strategies in executing test cases in configuration testing. Furthermore, it is natural to borrow existing techniques proposed in traditional test-case reduction and prioritization to facilitate configuration testing due to the similarity between these two problems.

2.4 Summary and discussion

Works on software testing mainly focus on test-case generation, test adequacy metrics, and strategies on executing test cases. These problems are important and have been studied for long.

Besides these topics, in software testing there are many other important problems that have rarely or seldom been touched. For example, test oracles are widely recognized as one of the most important problems in software testing, which provide mechanisms to tell whether the output of a test case is as expected. Unlike other topics in software testing (e.g., test-case generation), test oracles are far from being solved. Furthermore, in the last two years only two papers are related to test oracles. In particular, Nguyen et al.[18] conducted an empirical study to compare the cost and effectiveness of three types of automated oracles (i.e., data invariants, temporal invariants, and finite state automata), Staats et al.[19] proposed a mutation analysis based approach to selecting a set of variables from the program as the oracle data. Although test oracles are important, a little improvement has been achieved in the past two years due to its difficulty. To facilitate software testing, before generating test oracles automatically, it is also important and helpful to provide some aids to facilitate test oracle generation for developers in practice (e.g., selecting variables observed in software testing and determining expected values of variables).

Besides these problems identified and discussed in the research communities, there still exist many practical software testing problems that are rarely touched by the academy. For example, integration testing is always an important phrase in software testing, and there are many important issues in integration testing (e.g., fault aggregation and fault localization) influencing the efficiency of integration testing. It is important and necessary for the research communities to focus on these practical problems so as to facilitate software testing in practice.

3 Software Debugging

Software debugging aims to identify the location of faults and then fix the

faults by replacing the faulty statements by correct ones. According to the two phases of software debugging, most works on debugging can be classified into fault localization and fault fixing. Bug repositories, consisting of bug reports, which describe bugs submitted by developers, testers or end-users, play an important role in software development for both commercial projects and open-source projects. However, due to the data privacy of commercial projects, researchers in software debugging mainly started their research on analyzing and utilizing bug repositories recently when open-source projects became popular and a large number of bug repositories became available.

3.1 Fault localization

Fault localization aims to reduce the searching scope of statements that are possible to contain faults or even identify the location of faulty statements. As the first step of software debugging, fault localization has been studied for long. Based on the type of faults to localize, the works on fault localization can be classified into works on general faults (e.g., logical faults) and works on some specific faults (e.g., memory leaking problem and redundant code issue). In the past (especially before 2002), most of the researchers in this domain studied how to localize specific faults rather than general faults. However, between 2002 and 2010 more researchers focused on localizing general faults and most of their works utilized test information to aid fault localization.

In the last two years, few published papers on debugging studied general faults as such works have been fully studied before. In particular, before 2010, a large number of papers on spectrum-based fault localization have been published, aiming to localize suspicious statements by analyzing the difference between failed executions and passed executions. That is, spectrum-based fault localization usually targets at general faults. Following existing works on spectrum-based fault localization, in 2012, Gopinath et al.[20] proposed to use specification-based analysis in spectrum-based fault localization so as to improve the effectiveness of the latter. To facilitate the use of spectrum-based fault localization, Gong et al.[15] proposed a test-case selection technique based on diversity maximization speedup to reduce the number of test cases that are used to maximize the effectiveness of spectrum-based fault localization. Besides spectrum-based fault localization, there are a small number of works on locating general faults based on bug reports, which will be introduced in Section 3.3.

Most of the published papers on fault localization are concerned with identifying and finding some specific faults, e.g., performance faults[21,22], concurrency faults[23,24], configuration faults[25], memory leaking problem in Android applications[26], and so on.

In summary, research on localizing general faults has achieved slight progress as it is difficult to significantly improve the effectiveness of these techniques. At the same time, more and more researchers began to investigate the works on specific faults, since these works have more input information and may produce mature results to be applied in practice. In practice, software debugging is still mainly manual works depending on developers' experience and understanding. Therefore, researchers may consider improving the effectiveness of existing fault localization by combining developers' feedback.

3.2 Fault fixing

Although fault fixing is a problem with a long history in software engineering, research on fixing general faults actually became popular recently. Before 2009, most works on fixing general faults are specification based approaches, which may produce some repairs that break other functionalities and require the existence of specification. However, in practice, formal specifications are rarely available, and thus specification based approaches did not attract too much attention in the past.

In 2009 Weimer et al.[27] proposed a generic programming based approach GenProg to automatically fixing faults, which used generic programming to maintain a set of variants for the program and modified these variants using two generic algorithm operations to find a variant that passed all the test cases. As GenProg showed a possible automated way on fixing general faults, more and more researchers focused on fixing general faults. In particular, as GenProg[27] repaired only faults in off-the-shelf C programs, Goues et al.[28] in 2012 proposed a novel algorithmic improvement to repair faults in large programs, which has been confirmed to repair 105 faults from 8 open-source programs. Different from existing specification-based fault fixing, these works aim to generate a fix that passes all the test cases. Therefore, it is interesting and necessary to learn whether the program with such a fix actually works as expected.

Besides the works on fixing general faults, some works still focused on specific faults. In particular, Samimi et al.[29] proposed an approach to helping developers find and fix HTML generation errors in PHP programs by using string constraint solving, Coker and Hafiz[30] proposed three program transformations to fix C integer problems.

In summary, although fault fixing is a long-term problem, many works on fault fixing focused on specific faults rather than general faults. As in practice faults are usually fixed by developers manually rather than by some automatic tool, it is worth and interesting to learn how developers fix faults so as to facilitate the works on fixing faults. Based on this motivation, Murphy-Hill et al.[31] conducted an empirical study on how developers make design choices in fixing faults. Following this empirical study, it may be also important to find a way to simulate developers' decisions in fixing faults so as to facilitate automated fault fixing.

3.3 Bug repository

During software development, many commercial projects and open-source projects use bug repositories or issue tracking systems to record various problems reported by users or developers. However, due to the access limit of commercial projects and their various resources including bug repositories, few researchers focus on bug repositories until the popularity of open-source projects. A large number of bug reports exist in the repositories of open-source projects. For example, in the open-source version of Eclipse, average 29 bug reports are submitted each day[32]. A bug report is a document recording the identification and solving history of a bug. With the availability of a large number of bug repositories provided by open-source projects, more and more works focused on bug repositories, including bug report quality, bug report duplication, bug report assignment, debugging based on bug reports, and so on, to aid the software development based on bug reports.

Works on bug report assignment aim to automatically assign bug reports to appropriate developers. Most existing works on bug report assignment[32,33] use various information of bug reports (e.g., summary and description[34], identities of bug reports and those of developers[35,36], comments[37]) to predict the fixer of a bug report based on some machine-learning techniques. In the last two years, bug report assignment continued attracting researchers' interest and most of its works focused on improving the accuracy of bug report assignment. In particular, Servant and Jones[38] combined fault localization, history mining, and expertise assignment to automatically assign a fault (represented by a failing test case) to the most appropriate developer. Xuan et al.[39] used a socio-technical approach to modeling developers' prioritization in bug repositories to facilitate three predicting tasks around bug repositories. As it is inefficient for developers to read bug reports one by one, Bortis and Hoek[40] proposed Porchlight, which grouped bug reports based on tags so that developers can work on bugs in meaningful groups.

As bug reports tell the existence of faults, some researchers began to investigate how to debug based on bug reports. Given a bug report, existing works usually identified the positions of faults in the source code by their relevance to a bug report[41], or recommended possible fixes by analyzing history information (e.g., similar bugs[42]). Works in the last two years focused on improving the effectiveness of such a debugging process by combining new information or new techniques. In particular, to locate relevant files for fixing a fault, Zhou et al.[43] proposed an information retrieval based technique, which ranked the files based on the similarity between the initial bug report and the source code using a revised Vector Space Model by considering similar bugs fixed before. As natural language information retrieval based techniques produce localization results with low accuracy, Saha et al.[44] proposed a structural information retrieval based fault localization technique instead.

As bug reports play an important role in software development, especially software debugging, it is interesting to study and improve the quality of bug reports. For example, during bug report triaging, developers have to look through a large number of bug reports to find similar problems and thus bug report summarization is proposed to reduce the amount of data developers have to read. In 2012, Mani et al.[45] proposed four unsupervised summarization techniques to summarize bug reports. To improve the accuracy of duplicate bug detection, Nguyen et al.[46] proposed a duplicate bug report detection technique based on the combination of information-retrieval features and topic features.

In summary, many works focused on bug repositories (including bug reports) since a large number of bug reports are available in practice and useful information may be extracted from these bug reports to assist various software-engineering tasks. As most existing works on bug repositories are based on some specific projects, it is necessary to study whether the findings from these specific projects can be generalized to others.

3.4 Summary and discussion

As fault localization has been fully studied, many researchers in software debugging switched their interests to fault fixing instead. However, existing

fault-localization approaches are far from practice because of the following reasons. First, these approaches cannot locate the positions of faults correctly. Second, these approaches can hardly be combined in developers' manual debugging process so that few of them have been applied in practice. Moreover, existing works on fault fixing have a practical problem because we cannot tell whether a fix is correct or wrong just based on the results of test cases, although such knowledge eases the difficulty of research works on fault fixing. As open-source projects provide a large number of available bug reports, works on bug repositories attract much attention and thus become popular. This problem is new and interesting. However, considering the diversity of projects, it may be interesting to study the commonality and difference of bug repositories for different projects so that the findings from one project may be generalized to others.

Although fault localization and fixing are old problems, researchers discover many other related problems (e.g., test-case selection to facilitate fault localization) by further studying these two old problems. Moreover, works on these related problems might further facilitate the works on fault localization and fixing as well. Since bug repositories have not been studied long, many problems exist and need further studied. However, considering the existence of a large number of bug repositories, it is important to study how to investigate the repositories of different structures to support further development. Since there exist a large number of bug repositories and future development may continue using various bug repositories, it is also important to consider how to combine bug reports and bug repositories.

4 Software Analysis

Program analysis is a process that analyzes the behavior of a program and thus is widely used for solving various software-engineering problems such as software testing and debugging. Nowadays, besides the programs themselves, the other data of projects produced during development and runtime become more and more important. Such data are no less important than the programs themselves, and thus it is necessary to provide some infrastructure to assist analysis on these data, which are called software-engineering data in this article. According to the definition, software consists of programs, documents, and the data collected during software development and runtime. Therefore, this section studies software analysis, which is divided into program analysis and software-engineering data analysis.

4.1 Program analysis

Program analysis, aims to analyze the behavior of a program, and its techniques have been widely used to solve various software-engineering problems. Therefore, many domains in software engineering, e.g., software testing and program comprehension, can be facilitated by the improvement on program analysis.

In the last two years, some program analysis techniques were revisited and researchers focused on addressing the existing problems in these techniques. Symbolic execution, firstly proposed before 1980, is an important technique widely used in test-case generation[47] since 2000. As it is difficult to determine symbolic values for loops and library calls in traditional symbolic analysis, Le[48] in 2013 proposed a hybrid technique to address such a challenge by partitioning a program

on demand and invoking concrete executions on selected code segment. To reduce the invalid inputs and unidentified infeasible traces in symbolic analysis, Braione et al.[49] proposed to use program invariants over the lazy initialization process to characterize valid input states and add term rewriting mechanisms to symbolic execution. To reduce the instrumentation overhead in dynamic analysis, Saha et al.[50] proposed a distributed trace collection framework that collects traces following the divide-and-conquer strategy. Similarly, to aid dynamic analysis on multithread programs with low overhead, Ganai et al.[51] proposed a framework based on dynamic taint analysis.

Due to the wide usage of JavaScript applications, researchers began to focus on analyzing the behavior of programs in JavaScript. As the dynamic nature of JavaScript, some existing analysis techniques (including static and dynamic techniques) for C++/Java cannot be applied to programs in JavaScript. Therefore, in the last two years, Feldthaus et al.[52] proposed a scalable field-based flow analysis to construct approximate call graphs for JavaScript, Sen et al.[53] presented a heavy-weight dynamic analysis framework JALANGI, Madsen et al.[54] proposed a technique combining traditional pointer analysis with novel use analysis to deal with library code in JavaScript application.

In summary, researchers mainly focused on improving existing program analysis techniques. Program analysis techniques have important influence in software engineering, and their slight improvement may also facilitate other software engineering tasks[55-57]. Although program analysis has been widely used to solve various software engineering problems, few program analysis tools are available with friendly interface and scalability. To facilitate the use of program analysis techniques, researchers may consider improve the scalability of their program analysis techniques, and provide program analysis tools as well.

4.2 Software-engineering data analysis

During software development and execution, a large amount of development and runtime data are collected for both commercial projects and open-source projects. However, due to the access limit, researchers can hardly acquire the data of commercial projects and use these data in their works. Recently as open-source projects became popular, there emerge a large number of projects whose various artifacts (e.g., source code, documents, bug repositories, and runtime log) are available. These data are the products generated during software development and runtime. Software-engineering data analysis is the process of analyzing various data of projects, including documents, and other relevant data of projects.

Software-engineering data analysis is different from program analysis on their major subjects and the analysis techniques. In particular, program analysis takes only programs as subjects under analysis, whereas software-engineering data analysis takes various data as subjects under analysis. Although these data include programs, programs are neither the major subjects nor the only subjects in software-engineering data analysis. Furthermore, program analysis usually focuses on the syntax and semantics of programs and analyzes programs based on its syntax (in essence, it follows the rule-based approach), whereas software-engineering data analysis usually deals with various data uniformly (i.e., ignores the syntax of programs) by mostly

using data mining techniques.

Some works on software-engineering data analysis aim to acquire some information explicitly by using some data-mining technique. To facilitate various software engineering tasks, researchers proposed to mine different information (e.g., specification[58,59], API usage[60], and requirements[61]). In particular, Schur et al.[62] proposed to mine behavior models from web applications to automate testing of web application, Sun and Khoo[63] proposed to mine signatures for a buggy program based on two sets of execution profiles of the program so as to identify the cause or effect of a bug, Fahland et al.[59] proposed to mine branching-time scenario based specification from a set of execution traces. Nowadays, more and more open-source projects emerge and thus a large number of project data are available. From these "big" data, researchers can mine useful information for the current project and future projects.

Some works on software-engineering data analysis aim to support some specific tasks in software engineering by analyzing project data, including fault detection[21,64,65], fault fixing[43,66], duplicate bug report detection[46], and so on. In particular, Palomba et al.[64] proposed an approach to detecting five types of bad smells by analyzing the history information from versioning system, Nguyen et al.[66] proposed a multi-layered approach to recovering the bug reports and fixes by learning the association relations between bug reports and changed source code, Zhou et al.[43] proposed an approach to finding relevant files for fixing a fault by mining the bug reports, Han et al.[21] proposed an approach to detecting performance faults by mining callstack traces, Pradel and Gross[65] proposed an approach to revealing API protocol violation by combining protocol mining, test-case generation, and protocol checking.

Some works on software-engineering data analysis give some basic infrastructures to support the analysis process. Nowadays, researchers tend to mine useful information from existing large open-source projects, which are collected by some software repositories, e.g., Sourceforge, GitHub, and Google Code. Due to the ultra-large-scale of these software repositories, it is hard for every researcher to systematically extract data from these repositories. To aid analysis on such ultra-large-scale software repositories, Dyer et al.[67] proposed a domain-specific language and infrastructure (i.e., Boa) to support works on software-repository mining. Furthermore, to aid developers on big data analytics applications, which are a new category of applications, Shang et al.[68] proposed an approach to verifying the runtime execution of such applications after deployment. Although software-engineering data analysis targets at different software-engineering tasks, the subjects of such analysis are all the large number of data stored in various repositories. To aid such analysis by avoiding repeated works, it is useful to construct some infrastructures for such analysis.

In summary, due to the emerging of big data analytics applications and existence of ultra-large-scale software repositories, software-engineering data analysis has become a very hot topic. However, the data under analysis are usually from different projects and of different structures, researchers should consider how to deal with the diversity of data and generalize the findings from some specific projects.

4.3 Summary and discussion

This section presents the recent progress of software analysis, which is the process of analyzing the behavior and data of projects. Data of various formats for a project are important artifacts of a project. Moreover, these data (e.g., documents and bug repositories) are no less important than the programs of a project due to the following main reasons. First, some data (e.g., pictures in the webpage) are as necessary as programs. Second, some data of a project may help developers or maintainers to understand the programs of the project. Third, some data of projects may provide useful information or knowledge (e.g., API usage, debugging aid) for future projects.

In program analysis, besides works on improving existing program analysis techniques, many researchers investigated the analysis techniques for programs in JavaScript since existing analysis techniques for C++/Java programs can hardly be applied to JavaScript programs. On the other hand, program analysis techniques have been widely used to solve various software engineering tasks e.g., test-case generation. However, it is costly to apply these techniques because these techniques are lack of robust and friendly tool support. That is, tools are important in facilitating the practical usage of existing program analysis techniques.

Software-engineering data analysis is a new topic, which attracts much attention immediately as it emerges. Similar to works on bug repositories, it is also important to study software-engineering data analysis via different projects considering the diversity of data of different projects.

5 Conclusion and Discussion

Based on the quantitative analysis on the papers of software engineering in the last two years, this article identified three hot topics, software testing, debugging, and analysis. For each topic, this article briefly reviewed its works in the last two years and discussed its characteristics compared with its prior works. The characteristics of these hot topics are summarized as follows. Most works in software testing are still concentrating on the kernel problems of this domain. Most works in software debugging focus on bug repositories and fault fixing. Software-engineering data analysis becomes very hot in the area of software analysis.

Although various techniques have been proposed in software testing, debugging, and analysis, many of these techniques use or borrow some common techniques, e.g., machine learning, data mining, and statistical analysis. For example, in software testing, Zhang et al.[17] defined the basic and extended models of test-case prioritization using the probability theory; in software debugging, most spectrum-based fault-localization techniques calculated the suspicions of statements using the statistical theory; software analysis usually discovered the secret behind the large number of project data using data-mining techniques. Besides the widely used program analysis techniques, many problems in software testing, debugging, and analysis tend to borrow some data analysis techniques. That is, software-engineering data analysis plays an important role in software engineering.

Furthermore, existing works on software testing, debugging, and analysis usually used some benchmarks from research communities to demonstrate their effectiveness. However, even if a technique has been evaluated to be effective on some benchmark, it

may not be applied to industry due to possible implementation complexity. Moreover, benchmarks may be not representative of industrial projects, and thus the conclusion based on benchmarks may be not generalized to industry projects as well. Therefore, considering practical issues, researchers are suggested to evaluate their techniques on industrial projects rather than simplified benchmarks.

Note: It is necessary to point out that, due to the limit of authors' knowledge, the comments or points may not be totally correct. It is only for readers' information.

References

[1] Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. Proc. of the 29th International Conference on Software Engineering. 2007. 75–84.

[2] Garg P, Ivancic F, Balakrishnan G, Maeda N, Gupta A. Feedback-directed unit test generation for C/C++ using concolic execution. Proc. of the 35th International Conference on Software Engineering. 2013. 132–141.

[3] Nistor A, Luo Q, Pradel M, Gross TR, Marinov D. BALLERINA: Automatic generation and clustering of efficient random unit tests for multithreaded code. Proc. of the 34th International Conference on Software Engineering. 2012. 727–737

[4] Zaeem RN, Khurshid S. Test input generation using dynamic programming. Proc. of the 20th ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2012. 34.

[5] Thummalapenta S, Lakshmi KV, Sinha S, Sinha N, Chandra S. Guided test generation for web applications. Proc. of the 35th International Conference on Software Engineering. 2013. 162–171.

[6] Machiry A, Tahiliani R, Naik M. Dynodroid: An input generation system for Android apps. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 224–234.

[7] Fraser G, Arcuri A. Sound empirical evidence in software testing. Proc. of the 34th International Conference on Software Engineering. 2012. 178–188.

[8] Xiao X, Li S, Xie T, Tillmann N. Characteristic studies of loop problems for structural test generation via symbolic execution. Proc. of the 28th IEEE/ACM International Conference in Automated Software Engineerig. 2013. 246–256.

[9] Whalen M, Gay G, You D, Heimdahl MPE, Staats M. Observable modified condition/decision coverage. Proc. of the 35th International Conference on Software Engineering. 2013. 102–111.

[10] Hassan MM, Andrews JH. Comparing multi-point stride coverage and data flow coverage. Proc. of the 35th International Conference on Software Engineering. 2013. 172–181.

[11] Tasharofi S, Pradel M, Lin Y, Johnson R. Bita: Coverage-guided, automatic testing of actor programs. Proc. of the 28th IEEE/ACM International Conference in Automated Software Engineerig. 2013. 114–124.

[12] Blue D, Segall I, Tzoref-Brill R, Zlotnick A. Interaction-based test-suite minimization. Proc. of the 35th International Conference on Software Engineering. 2013. 182–191.

[13] Song C, Porter A, Foster JS. Itree efficiently discovering high-coverage configurations using interaction trees. Proc. of the 34th International Conference on Software Engineering. 2012. 903–913.

[14] Petke J, Yoo S, Cohen MB, Harman M. Efficiency and early fault detection with lower and higher strength combinatorial interaction testing. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 26–36.

[15] Gong L, Lo D, Jiang L, Zhang H. Diversity maximization speedup for fault localization. Proc. of the 27th IEEE/ACM International Conference in Automated Software Engineerig. 2012. 30–39.

[16] Hao D, Zhang L, Wu X, Mei H, Rothermel G. On-demand test suite reduction. Proc. of the 34th International Conference on Software Engineering. 2012. 738–748.

[17] Zhang L, Hao D, Zhang L, Rothermel G, Mei H. Bridging the gap between the total and

additional test-case prioritization strategies. Proc. of the 35th International Conference on Software Engineering. 2013. 192–201.

[18] Nguyen CD, Marchetto A, Tonella P. Automated oracles: An empirical study on cost and effectiveness. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 136–146.

[19] Staats M, Gay G, Heimdahl MPE. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. Proc. of the 34th International Conference on Software Engineering. 2012. 870–880.

[20] Gopinath D, Zaeem RN, Khurshid S. Improving the effectiveness of spectra-based fault localization using specifications. Proc. of the 27th IEEE/ACM International Conference in Automated Software Engineerig. 2012. 40–49.

[21] Han S, Dang Y, Ge S, Zhang D, Xie T. Performance debugging in the large via mining millions of stack traces. Proc. of the 34th International Conference on Software Engineering. 2012. 176–186.

[22] Nistor A, Song L, Marinov D, Lu S. Toddler detecting performance problems via similar memory-access patterns. Proc. of the 35th International Conference on Software Engineering. 2013. 562–571.

[23] Zhou J, Xiao X, Zhang C. Stride: Search-based deterministic replay in polynomial time via bounded linkage. Proc. of the 34th International Conference on Software Engineering. 2012. 890–900.

[24] Farzan A, Madhusudan P, Razavi N, Sorrentino F. Predicting null-pointer dereferences in concurrent programs. Proc. of the 20th ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2012. 47.

[25] Zhang S, Ernst MD. Automated diagnosis of software configuration errors. Proc. of the 35th International Conference on Software Engineering. 2013. 312–321.

[26] Guo C, Zhang J, Yan J, Zhang Z, Zhan Y. Characterizing and detecting resource leaks in Android applications. Proc. of the 28th IEEE/ACM International Conference in Automated Software Engineerig. 2013. 389–398.

[27] Weimer W, Nguyen T, Goues CL, Forrest S. Automatically finding patches using genetic programming. Proc. of the 31st International Conference on Software Engineering. 2009. 364–374.

[28] Goues CL, Dewey-Vogt M, Forrest S, Weimer W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. Proc. of the 34th International Conference on Software Engineering. 2012. 3–13.

[29] Samimi H, Schafer M, Artzi S, Millstein T, Tip F, Hendren L. Automated repair of HTML generation errors in PHP applications using string constraint solving. Proc. of the 34th International Conference on Software Engineering. 2012. 277–287.

[30] Coker Z, Hafiz M. Program transformations to fix C integers. Proc. of the 35th International Conference on Software Engineering. 2013. 792–801.

[31] Murphy-Hill E, Zimmermann T, Bird C, Nagappan N. The design of bug fixes. Proc. of the 35th International Conference on Software Engineering. 2013. 332–341.

[32] Anvik J, Hiew L, Murphy GC. Who should fix this bug? Proc. of the 28th International Conference on Software Engineering. 2006. 361–370.

[33] Xuan J, Jiang H, Ren Z, Yan J, Luo Z. Automatic bug triage using semi-supervised text classification. Proc. of the 22nd International Conference on Software Engineering and Knowledge Engineering. 2010. 209–214.

[34] Cubranic D, Murphy GC. Automatic bug triage using text categorization. Proc. of the 16th International Conference on Software Engineering and Knowledge Engineering. 2004. 92–97.

[35] Bhattacharya P, Neamtiu I. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. Proc. of the 26th International Conference on Software Maintenance. 2010. 1–10.

[36] Tamrawi A, Nguyen TT, Al-Kofahi J, Nguyen TN. Fuzzy set-based automatic bug triaging. Proc. of the 33rd International Conference on Software Engineering. 2011. 884–887.

[37] Baysal O, Godfrey MW, Cohen R. A bug you like: A framework for automated assignment

of bugs. Proc. of the 17th IEEE International Conference on Program Comprehension. 2009. 297–298.

[38] Servant F, Jones JA. WHOSEFAULT: Automatic developer-to-fault assignment through fault localization. Proc. of the 34th International Conference on Software Engineering. 2012. 36–46.

[39] Xuan J, Jiang H, Ren Z, Zou W. Developer prioritization in bug repositories. Proc. of the 34th International Conference on Software Engineering. 2012. 25–35.

[40] Bortis G, Hoek AVD. PorchLight: A tag-based approach to bug triaging. Proc. of the 35th International Conference on Software Engineering. 2013. 342–351.

[41] Lukins SK, Kraft NA, Etzkorn LH. Source code retrieval for bug localization using latent dirichlet allocation. Proc. of the 15th Working Conference on Reverse Engineering. 2008. 155–164.

[42] Ashok B, Joy J, Liang H. DebugAdvisor: A recommender system for debugging. Proc. of the 17th ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2009. 373–382.

[43] Zhou J, Zhang H, Lo D. Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. Proc. of the 34th International Conference on Software Engineering. 2012. 14–24.

[44] Saha RK, Leasey M, Khurshid S, Perry DE. Improving bug localization using structured information retrieval. Proc. of the 28th IEEE/ACM International Conference in Automated Software Engineerig. 2013. 345–355.

[45] Mani S, Catherine R, Sinha VS, Dubey A. AUSUM: Approach for unsupervised bug report summarization. Proc. of the 20th ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2012. 11.

[46] Nguyen AT, Nguyen TT, Nguyen TN, Lo D, Sun C. Duplicate bug report detection with a combination of information retrieval and topic modeling. Proc. of the 27th IEEE/ACM International Conference in Automated Software Engineerig. 2012. 70–79.

[47] Zhang L, Xie T, Zhang L, Tillmann N Halleux JD, Mei H. Test generation via dynamic symbolic execution for mutation testing. Proc. of the 26th International Conference on Software Maintenance. 2010. 1–10.

[48] Le W. Segmented symbolic analysis. Proc. of the 35th International Conference on Software Engineering. 2013. 212–221.

[49] Braione P, Denaro G, Pezzè M. Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 411–421.

[50] Saha D, Dhoolia P, Paul G. Distributed program tracing. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 180–190.

[51] Ganai M, Lee D, Gupta A. DTAM: dynamic taint analysis of multi-threaded programs for relevancy. Proc. of the 20th ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2012. 46.

[52] Feldthaus A, Schaefer M, Sridharan M, Dolby J, Tip F. Efficient construction of approximate call graphs for JavaScript IDE services. Proc. of the 35th International Conference on Software Engineering. 2013. 752–761.

[53] Sen K, Kalasapur S, Brutch T, Gibbs S. Jalangi: A selective record-replay and dynamic analysis framework for JavaScript. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 488–498.

[54] Madsen M, Livshits B, Fanning M. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 499–509.

[55] Wang X, Zhang L, Xie T, Xiong Y, Mei H. Automating presentation changes in dynamic web applications via collaborative hybrid analysis. Proc. of the 20th ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2012. 16.

[56] Wang X, Dang Y, Zhang L, Zhang D, Lan E, Mei H. Can I clone this piece of code here? Proc. of the 27th IEEE/ACM International Conference in Automated Software Engineerig. 2012. 170–179.

[57] Meng S, Wang X, Zhang L, Mei H. A history-based matching approach to identification of framework evolution. Proc. of the 34th International Conference on Software Engineering. 2012. 353–363.

[58] Kumar S, Khoo SC, Roychoudhury A, Lo D. Inferring class level specifications for distributed systems. Proc. of the 34th International Conference on Software Engineering. 2012. 914–924.

[59] Fahland D, Lo D, Maoz S. Mining branching-time scenarios. Proc. of the 28th IEEE/ACM International Conference in Automated Software Engineerig. 2013. 443–453.

[60] Uddin G, Dagenais B, Robillard MP. Temporal analysis of API usage concepts. Proc. of the 34th International Conference on Software Engineering. 2012. 804–814.

[61] Carreño LVG, Winbladh K. Analysis of user comments: An approach for software requirements evolution. Proc. of the 35th International Conference on Software Engineering. 2013. 582–591.

[62] Schur M, Roth A, Zeller A. Mining behavior models from enterprise web applications. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 422–432.

[63] Sun C, Khoo SC. Mining succinct predicated bug signatures. Proc. of the 21st ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2013. 576–586.

[64] Palomba F, Bavota G, Penta MD, Oliveto R, Lucia AD, Poshyvanyk D. Detecting bad smells in source code using change history information. Proc. of the 28th IEEE/ACM International Conference in Automated Software Engineerig. 2013. 268–278.

[65] Pradel M, Gross TR. Leveraging test generation and specification mining for automated bug detection without false positives. Proc. of the 34th International Conference on Software Engineering. 2012. 288–298.

[66] Nguyen AT, Nguyen TT, Nguyen HA, Nguyen TN. Multi-layered approach for recovering links between bug reports and fixes. Proc. of the 20th ACM SIGSOFT International Symposum on the Foundations of Software Engineering. 2012. 63.

[67] Dyer R, Nguyen HA, Rajan H, Nguyen TN. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. Proc. of the 35th International Conference on Software Engineering. 2013. 422–431.

[68] Shang W, Jiang ZM, Hemmati H, Adams B, Hassan AE, Martin P. Assisting developers of big data analytics applications when deploying on hadoop clouds. Proc. of the 35th International Conference on Software Engineering. 2013. 402–411.