Roundtable: Research Opportunities and Challenges for Large-Scale Software Systems

Article *in* Journal of Computer Science and Technology · September 2016

DOI: 10.1007/s11390-016-1668-9

6 authors, including:

Xusheng Xiao
Case Western Reserve University
39 PUBLICATIONS **796** CITATIONS
SEE PROFILE

Jian-Guang Lou
Microsoft
69 PUBLICATIONS **1,599** CITATIONS
SEE PROFILE

David Shepherd
ABB
1 PUBLICATION **0** CITATIONS
SEE PROFILE

Xiao X, Lou JG, Lu S *et al.* Roundtable: Research opportunities and challenges for large-scale software systems. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 31(5): 851–860 Sept. 2016. DOI 10.1007/s11390-016-1668-9

Roundtable: Research Opportunities and Challenges for Large-Scale Software Systems

Xusheng Xiao [1], Jian-Guang Lou [2], *Member, ACM, IEEE*, Shan Lu [3], David C. Shepherd [4], Xin Peng [5], and Qian-Xiang Wang [6], *Member, CCF, IEEE*

[1] *NEC Laboratories America, Princeton, NJ 08540, U.S.A.*

[2] *Microsoft Research Asia, Beijing 100080, China*

[3] *Department of Computer Science, University of Chicago, Chicago, IL 60637, U.S.A.*

[4] *ABB Corporate Research, Raleigh, NC 27606, U.S.A.*

[5] *School of Computer Science, Fudan University, Shanghai 201203, China*

[6] *School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China*

E-mail: xsxiao@nec-labs.com; jlou@microsoft.com; shanlu@uchicago.edu; david.shepherd@us.abb.com
pengxin@fudan.edu.cn; wqx@pku.edu.cn

Received August 22, 2016; revised August 23, 2016.

Abstract For this special section on software systems, six research leaders in software systems, as guest editors for this special section, discuss important issues that will shape this field's future research directions. The essays included in this roundtable article cover research opportunities and challenges for large-scale software systems such as querying organization-wide software behaviors (Xusheng Xiao), logging and log analysis (Jian-Guang Lou), engineering reliable cloud distributed systems (Shan Lu), usage data (David C. Shepherd), clone detection and management (Xin Peng), and code search and beyond (Qian-Xiang Wang). — Tao Xie, Leading Editor of Software Systems.

Keywords organization-wide software behavior, log analysis, reliable cloud distributed system, usage data, clone detection and management, code search

1 Querying Organization-Wide Software Behaviors (Xusheng Xiao)

Today, computer systems are quite complex due to the numerous installed software programs. System experts usually have limited visibility into these software programs since the tools they use often give a partial view of the complex systems. Without monitoring and understanding the behaviors of these software programs, it is a challenging task for system experts to maintain the proper functioning of computer systems. For example, if a software program is compromised, the security of the customer data cannot be guaranteed; if certain software programs cause the system to fail, the services hosted in the system may be interrupted. Existing application logs (e.g., web logs and firewall logs) provide only partial information about specific software's behaviors[1], and thus are insufficient and ineffective for monitoring behaviors of various software programs. This motivates the recent trend of leveraging system monitoring logs to offer intelligence in system management.

System monitoring data is not bound to specific applications and has specific log structures, which can be collected by system-level monitoring tools, such as auditd①, ETW②, DTrace③, and sysdig④. System

Survey
Special Section on Software Systems 2016

①The linux audit framework. https://www.suse.com/documentation/sled10/audit_sp1/data/book_sle_audit.html, Aug. 2016.

②ETW events in the common language runtime. https://msdn.microsoft.com/en-us/library/ff357719(v=vs.110).aspx, Aug. 2016.

③DTrace. http://dtrace.org/, Aug. 2016.

④Sysdig. http://www.sysdig.org/, Aug. 2016.

monitoring data records all interactions among various software programs and system resources (e.g., processes, files, network sockets) over time as a sequence of events. Based on such data characteristics, system monitoring data can be represented as a temporal graph, with system entities as heterogeneous nodes and system events as edges with timestamps (i.e., pointing from the initiator node to the target node). Fig.1 gives the example of modeling system monitoring data as a temporal graph. In Fig.1, we have nine events annotated with the timestamps $t1 \sim t9$. For example, at timestamp $t1$, the initiator process $P1$ originated from some software program opens the target network connection $I1$ to send or receive some data, as indicated by the type of the interaction "open". By modeling system monitoring data as temporal graphs, we can easily understand the temporal event nature of the data and observe the software behaviors with respect to their interactions with system resources.

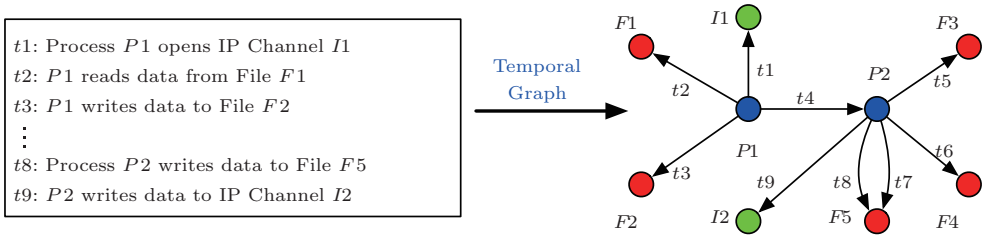


Fig.1. System monitoring data as a temporal graph.

To better comprehend the system status, an important step is to query the system monitoring data for interested software behaviors. For example, the system experts may want to query risky behaviors (either from publicly available knowledge bases⑤~⑨, or from their own definitions for suspicious behaviors specific to the organization). A typical query to find suspicious software behavior could be finding whether there exists any software program except Apache listening to the port 80 for the past week. In an enterprise environment where the majority are developer hosts, it is suspicious to have a process, not Apache, listening to the port 80. Another query to find suspicious software behavior could be finding whether there exists any software program accessing more than 100 IP addresses in the past minute. Such abnormal behaviors usually indicate that some malicious programs try to connect to a list of malicious domains. Knowing whether certain software behaviors occur at what time range provides insightful information for the system experts to diagnose different types of problems, such as malware infection and data exfiltration.

In order to help system experts query interested software behaviors over system monitoring data, I foresee that our research community needs to address the following challenges.

First, we need to develop a domain-specific language that can *intuitively* and *concisely* express interested software behaviors with temporal information. Existing querying languages, such as database querying languages SQL⑩ and Cypher⑪ and generic log querying languages provided by Splunk⑫ and Elasticsearch⑬, are designed for data retrieval based on data schema or regular expressions. Thus, to express behaviors, users need to translate the behaviors into data constraints[2], which becomes time-consuming and error-prone when

⑤McAfee. http://www.mcafee.com/, Aug. 2016.

⑥Exploit Database. https://www.exploit-db.com/, Aug. 2016.

⑦Common Vulnerabilities and Exposures (CVE). https://cve.mitre.org/, Aug. 2016.

⑧Symantec. https://www.symantec.com/, Aug. 2016.

⑨Kaspersky. www.kaspersky.com/, Aug. 2016.

⑩ISO/IEC 9075-1: 2008, Information technology–Database languages–SQL–Part 1: Framework (SQL/Framework). http://www.iso.org/iso/catalogue_detail.htm?csnumber=45498, Aug. 2016.

⑪Cypher Query Language. http://neo4j.com/developer/cypher/, Aug. 2016.

⑫Splunk. http://www.splunk.com/, Aug. 2016.

⑬Elasticsearch. https://www.elastic.co/, Aug. 2016.

the behavior involves three or more events. Also, it is verbose and cumbersome to specify temporal relationships using existing query languages. Temporal relationships are required in expressing software behaviors with multiple steps (e.g., file creation happening after net work accesses), and such relationships are important in inferring dependency relations among events. Using SQL, the query could easily include about 30 constraints for expressing a software behavior that consists of three events and two relationships. There is a strong need in developing a domain-specific language to ease the task of expressing interested software behaviors with temporal information and optimizing the query execution.

Second, we need to develop techniques that progressively return the search results of the interested software behaviors. Since every software program could have the potential to cause changes of the system status, we need to monitor and collect system monitoring data from every host in the organization, e.g., the whole enterprise. As certain complex behaviors (e.g., malicious behaviors involved in advanced persistent threat (APT) attacks[14]) may require investigation in a month of data or even longer term of data, organizations often need to keep several months to one year worth of data. Based on recent studies[3], monitoring software behaviors derived from system-level audit events produces huge amount of daily logs: an average desktop computer produces over one million events per day, and an organization consisting of 100 computers each day could produce databases over 50 GB. Also, the system experts often cannot understand the system status through exactly one query, and they usually diagnose the system via interactive querying, where they can write queries, get results, and refine the queries, etc. Due to the large amount of the data, directly storing the data into databases (e.g., PostgreSQL[15]) and querying over the data could easily cause the query execution to go up to more than 10 minutes, making interactive querying difficult to realize. To support effective and efficient interactive querying of system monitoring data, techniques that can split the query into sub-queries and progressively return the query results are much needed.

Third, we need to develop a domain-specific language that can query interested software behaviors directly over the data stream of the system monitoring data. Storing the system monitoring data into databases and querying interested behaviors with temporal information cannot provide real-time detection on interested software behaviors, and is inefficient in detecting statistics-based abnormal behaviors, such as most frequent behaviors in using network connections. To address these problems, stream processing can be leveraged to manage the data stream of the system monitoring data before the data is stored into databases. Existing stream processing tools often provide generic stream query languages that suffer from the similar problems as SQL on the system monitoring data: verbose and difficult to express interested software behaviors. A domain-specific language is much needed to easily query software behaviors over the data stream of the system monitoring data.

Moreover, the events of system monitoring data exhibit strong *spatial* and *temporal* properties: events in different hosts are independent and events in the same host are independent along the time. Based on such *spatial* and *temporal* properties of the data, there are various research opportunities to optimize the querying of software behaviors over the data, such as partitioning a query into several sub-queries that search a non-overlapping portion of the data and leverage *parallelism* to execute the queries in parallel.

System monitoring data has great values in capturing software behaviors with respect to their interactions with system resources. Such data is quite valuable in diagnosing various problems in reliability, performance, and security for computer systems in enterprises and governments. However, people are lacking effective and efficient automated tools to help them manage and query such data. More importantly, due to the nature of these problems, I foresee that many of them can be addressed by building upon existing software engineering techniques such as domain-specific languages, program analysis (e.g., light-weight analysis to provide more precise file access information), and verification (e.g., verifying temporal logic over interested software programs).

2 Logging and Log Analysis (Jian-Guang Lou)

As large-scale online SaaS systems, such as those of Microsoft, Google, and Amazon, are getting increasingly large and complex — they often contain hundreds of distributed components and support a large num-

[14] Trustwave Global Security Report 2015. https://www2.trustwave.com/rs/815-RFM-693/images/2015_TrustwaveGlobalSecurityReport.pdf, Aug. 2016.

[15] PostgreSQL. http://www.postgresql.org/, Aug. 2016.

ber of concurrent users, log analysis has become more and more important. Typically, software engineers first test an online service system in a lab (testing) environment and then deploy the system in production (actual) environment. The lab environment often has a small, pseudo cloud setting with a limited amount of data, while the production environment has a large and complex cloud infrastructure supporting a huge amount of data. Because of the differences between the lab and the production environments, online service systems often encounter unexpected problems, even they are well tested in the lab environment. As debugging tools (e.g., an IDE debugger) are often inapplicable in production settings, log analysis has become a principal way to perform various software-engineering tasks for an online SaaS system, e.g., monitoring, debugging, and troubleshooting. Log analysis now is also a very important technical pillar of Testing in Production (TiP), which has been considered as a great way to get exposure to the diversity of the real world, and to uncover bugs you just will not find in the lab for online services.

2.1 Logging

Given the importance of log analysis, it is crucial to ensure the quality of recorded log messages is high enough for engineers to perform their tasks. The quality of logging not only directly influences the effectiveness of log analysis for diagnosis, but also largely impacts the runtime performance of software as logs come at a cost of CPU or I/O overhead.

In the state-of-the-art practice of software engineering, there is no strong incentive for engineers to put efforts and attentions on log quality. Many engineers write their logging statements with an ad-hoc way. In current practice, we do not have a process to ensure the quality of logs. This is totally different from that of code quality insurance, where we have a closed-loop process (i.e., develop, code review, testing, and fixing) to ensure the code quality. Some teams do have documents of logging practice guidance. However, such logging guidance is often loosely followed. There is no review and no testing. In addition, different from code changes, there is no existing good practice for engineers to test the quality of logs. Given the importance of logs in the era of cloud computing, we argue that it is worthy to put efforts on research and developing new technologies in log quality including quality evaluation, testing, closed-loop process, and so on.

To make sure the high quality of logs with low cost, we need a good solution to answer the following questions: where do we add a logging statement? What shall we log? When do we log the information into storage? How to measure the quality of logging in terms of effectiveness and cost? We need to record enough log messages to support almost all postmortem analysis tasks. At the same time, we also try to avoid generating log messages that are not useful for our analysis tasks. It would be better to remove such useless and noisy log information at early stage to avoid potential overhead. There were some pioneer research studies that have addressed some of these questions in recent years[4-6]. However, it is still a very small step to a systematic way of the quality managing of logs like what we have done for source code. On the other hand, I believe there is a huge research potential along this direction.

2.2 Log Analysis

Log analysis is an art and science seeking to make sense out of computer-generated logs. There have been various efforts in the past years, which mainly include the following two major areas in service quality management:

• *Problem Detection*: detect potential issues based on system logs, events, counters, usage data, and customer support records.

• *Problem Localization and Diagnosis*: identify the problem site for a service live site issue, or provide information to help pinpoint the potential causes.

Problem detection is very important for SaaS providers to timely detect problems of their online services to avoid possible SLA (Service Level Agreement) violation. Over the years, various methods have been proposed to detect problems including change detection, outlier detection, multi-dimensional spike detection, trend detection, invariants based detection and PCA (Principal Component Analysis) based detection, etc. In the era of cloud computing, anomaly detection methods mainly have the following common requirements:

• *Scalability*: scalable and have low runtime complexity;

• *Adaptability*: adaptive to environment changes, take into account workload changing patterns including day of the week and hour of the day;

• *Usability*: easy to use, require no configuration, less parameter tuning is better;

• *Accuracy*: result in a low number of false positives and alarms to gain the trust of the user.

Understanding and characterizing normal behavior is the most important part of each technique. In most scenarios, we often do not have any concrete answer of "how to define a problem" rather than a rough idea of "an instance that deviates from an expected trend or normal behavior". Therefore, almost all techniques use historical data (or data from some healthy peers, a lab environment) to build models which we use to determine if a new data point is anomalous or not. A model is used to describe the normal data and represents all assumptions we supply about the normal data. The assumptions include and are not limited to the probability distribution which we may think fits the data. A range of models such as simple Gaussian model, general likelihood ratio model, ARIMA, Holt-Winters and sequential relationships have been used in previous literatures. Choosing a proper model and model parameters is very important as the model encodes your assumption of the data. Wrong assumption often means wrong results. Because in many cases, we do not know anything about the data your algorithm is monitoring for anomalous behavior, we cannot afford to make assumptions about the underlying data. In such cases, the fewer assumptions made, the better the approach will perform and scale. However, there is no any guidance or tool that can help software practitioners select a proper model (and model parameters) for a given scenario, which is a potential research topic in the future. Another research question that has not be touched is that: can we improve the performance of our anomaly detection by utilizing different types of data from different sources? In practice, we have many types of data from different sources including source code, configuration, code change history data, logs, service dependency, and so on. How to fully utilize these data sources to obtain a holistic view of anomaly detection is still an open research question now.

Problem diagnosis techniques localize likely causes of a problem to some locations or components, or find hints (e.g., some metrics, or log messages) that can directly/indirectly point to problem causes. Analyzing logs for problem diagnosis has been an active research area[7-12]. Many diagnosis techniques have been proposed in literatures including rule-based, model-based, statistical, machine-learning, and count-and-thresholding ones. Every technique has its own advantages and limitations. Different scenarios and data types need different kinds of techniques or models as each technique often has a special assumption on log data. One type of data source can only show one side of system. If engineers want to get a comprehensive diagnosis of a system, they must combine all analysis results from different data sources of the system to obtain a holistic view. However, there are very few tools in practice that can provide a holistic analysis of a system. How to get a comprehensive analysis result from various types of log data can be a potential research topic.

The insights obtained from data-driven log analysis are usually correlation rather than causation. There is still a gap between our analysis results and root causes. Root cause analysis often needs decent knowledge about the service system and deep experience in software engineering. Although log analysis techniques have demonstrated their usefulness in many scenarios, there are still some issues that need intensive manual efforts during their first-time occurrences. Studying on proper methodologies for representing complex domain knowledges and integrating them with data-driven techniques is still a very important part of future research[13].

3 Engineering Reliable Cloud Distributed Systems (Shan Lu)

In the big data and cloud computing era, software systems are getting more complex and new intricate bugs continue to create reliability issues that cause major economic losses. For example, outages caused by software bugs have happened to all major cloud service providers, including Google, Amazon, Microsoft Azure, and many more. Studies in the past have shown that a 5-minute outage could cost these companies almost a million dollars revenue. On one hand, the reliability of these cloud distributed systems will get increasingly crucial with our society's increasing reliance on them. On the other hand, maintaining the reliability of cloud distributed systems is much harder than that for traditional single-machine systems. More software engineering research is desired to tackle the reliability problems in these systems.

Cloud distributed systems run on hundreds or thousands of machines, execute complicated distributed protocols, and face a variety of hardware faults. They contain not only more bugs of traditional types, but also more bugs with unique types. We briefly discuss a few such bugs below.

• *Distributed Concurrency Bugs*[14-15]. These bugs are caused by unexpected timing in distributed systems. They are different from single-machine concurrency bugs in that they hide in a program state space that is

much larger and much more difficult to model, as distributed systems contain inter-node concurrency, intra-node process-level concurrency, intra-process thread-level concurrency, and intra-thread asynchronous concurrency. It is challenging to detect or expose these bugs, given the large and complicated program state space. It is also challenging to fix these bugs, because enforcing ordering across multiple machines is difficult and expensive.

• *Distributed Fault-Handling Bugs*[16]. These bugs are caused by improper handling of node failures and network failures in distributed systems. These bugs are common, as hardware faults are common in a large-scale systems. They are also unique: traditional single-machine software mostly does not need to handle hardware faults, while distributed systems are expected to continue functioning in face of hardware faults. It is challenging to expose these bugs, as their triggering conditions are special. It is also challenging to diagnose these bugs, as their symptoms are always mixed with other types of failure symptoms.

• *Distributed/Big-Data Performance Problems*[17-18]. Performance slowdowns in one node could propagate to other nodes and eventually bring down the whole system. Processing *big* data naturally makes the system vulnerable to memory management and out-of-memory problems. These problems are often made worse by the semantic gap between the cloud infrastructure systems and the cloud application systems: the infrastructure systems do not know what exactly the application is doing; the application language hides low-level details and makes application writers unable to make the best use of the system resource.

Overall, big data and cloud computing systems bring us easy-to-access computing power and new scientific and commercial opportunities. At the same time, they also bring us unique challenges in software reliability. More software engineering research is needed to tackle these important and challenging problems.

4 Usage Data (David C. Shepherd)

For many years now personal computers have been powerful enough to automatically collect and upload anonymous usage data without causing any noticeable performance degradation for end users. Developers at Eclipse⑯, Microsoft⑰, Autodesk, ABB Inc. and many others have collected anonymous usage data as part of efforts to continually improve their product. Unfortunately this data, usually a sequence of commands or events, has proven difficult to analyze and has primarily been used to generate aggregate data[19]. This data, which includes ranked lists of most used commands, breakdown of users by country, or analysis of which features are used most, has proven to be useful, but perhaps not as useful as its initial promise. For instance, the Eclipse Usage Data project was discontinued after several years due to lack of value⑱. It is clear that our community has not yet found a killer application for usage data.

Since aggregate data has not yet provided compelling findings, our community needs to investigate whether high value applications of this data exist. While straightforward applications have proved to under-deliver, there are potential applications of this data that, if feasible, would be high impact.

Perhaps one promising application of this data would be to use it as input during product planning. If usage data could influence which features to enhance or which workflows to smooth, the next sprint could better target the most pressing needs first. The challenge for this application lies in automating the mapping between usage data and programming tasks.

A tantalizing, but more challenging application of usage data would be to leverage it to replicate errors that are occurring in the field. Since developers often report that reproducing a problem is usually more difficult than fixing it, this application would prove to be valuable. However, because usage data is anonymous, requiring that customer-specific information such as the file or data the user was working with when the error occurred is removed, this high-value application will be difficult to realize.

Another interesting potential application of usage data is to refine user interfaces. Given data from thousands or millions of users designers could optimize the placement of buttons, panes, and commands to ease the end user's workflow. Unfortunately, current usage data collection facilities of many applications do not contain enough information for user interface analysis. For instance, Eclipse's UDC collected which underlying commands were executed, not which button or keyboard shortcut was used to trigger them. Without this information, automated user interface analysis is infeasible.

⑯http://www.eclipse.org/org/usagedata, Aug. 2016.

⑰http://en.wikipedia.org/wiki/Windows_10, Aug. 2016.

⑱Beaton W. Bye bye mon... udc, 2011. http://waynebeaton.wordpress.com/2011/09/16/bye-bye-mon-udc/, Aug. 2016.

I believe that we are entering an exciting period in analyzing usage data. Many applications collect the data and feedback from early usage data analysis, which is beginning to reshape exactly which details are collected. As this second generation of usage data is collected, our community has a second chance to examine this data and solidify its value by finding its killer application.

5 Clone Detection and Management (Xin Peng)

Code clones mean similar or identical code fragments, which are basically the results of code copy/paste (with or without minor modification) but may also be caused by accidental cloning (e.g., implementing similar functionalities by using the same set of APIs)[20]. Code clones are common in both commercial and open-source software systems. Code clones are code duplication, which is a well-known code smell. They are generally considered to be harmful, as they may increase the length and complexity of code and cause inconsistent changes, thus incurring additional maintenance effort and even bugs. Thus, the focus of traditional code clone research was mainly on clone detection and clone elimination (e.g., by refactoring).

However, code cloning is an efficient code reuse strategy and can be a reasonable design decision[21]. Some empirical studies have revealed that the reasons why a code clone is introduced or evolved and the harmfulness of a code clone must be interpreted and considered in a larger context from technical, personal, and organizational perspectives[22]. It is hard to say whether code clones are harmful or not and it is impossible to eliminate all the code clones. On the other hand, large-scale code and knowledge reuse causes that a software system usually has hybrid code sources from, for example, open-source projects, online communities, technical documents, legacy systems, or similar products. Therefore, the focus of code clone research has been shifted from clone detection to clone management, which encompasses a wide range of categories of activities including clone detection, clone documentation and analysis, tracking of clone evolution, and refactoring of code clones[23].

To achieve systematic clone management, it is desired that an organization establishes clone management mechanisms from the following aspects.

• *Management of Code Clone Genealogies.* Clone genealogies are an automatically extracted history of clone evolution from a sequence of program versions[24]. By systematically identifying and managing clone genealogies in a large scale (e.g., involving all the software projects of an organization), developers can have a comprehensive understanding of the origination and evolution of each of the code elements (e.g., files, methods, code fragments) of a project.

• *Continuous Monitoring of Cloning Events.* Cloning events (e.g., the addition, removal, or modification of code clones) are captured through continuous monitoring of code repositories and thus different stakeholders can be timely notified about relevant code cloning events in an on-demand way according to customized strategies[25]. In this way, stakeholders can take timely action when specific cloning events (e.g., a code copy/paste that violates architectural constraints) occur.

• *Consistent Editing of Code Clones.* Code templates can be extracted from code clones by code differencing. When a developer copies and pastes a code fragment with existing code clones, he/she can be recommended with modification slots of the pasted code and corresponding options in an interactive way[26]. On the other hand, some code clones need to be consistently maintained after being introduced. When a clone instance is modified, the modification needs to be propagated to the other instances of the same clone class or be notified to related developers.

6 Code Search and Beyond (Qian-Xiang Wang)

Because of the rapidly expanding code bases, nowadays, many programmers try to search for existing code snippets that implement the expected feature. Typically a programmer enters a natural language search query, and gets back one or multiple snippets. While existing code search engines, such as Ohloh Code[19] and Krugle[20], use text similarity to find code snippets in open source code repositories (e.g., GitHub, SourceForge), some new approaches take into account code characteristics, such as API usage patterns[27], encoded code patterns[28], and method call relationship[29], to recommend better snippets to programmers.

Besides this typical scenario, there are some other related interesting code search scenarios. In my opin-

[19] https://code.openhub.net/, Aug. 2016.

[20] http://www.krugle.com/, Aug. 2016.

ion, two scenarios are the most interesting: code-search-based synthesis and code-search-based bug localization.

Most snippets recommended by common code search tools just implement features similar to the expected features. Few of these recommended snippets implement the target features exactly. Based on these snippets, programmers need to modify the code, e.g., change variable names or parameter values, change expressions, or even branch conditions, so as to get the exact code. Can code search tools be improved to recommend exact expected snippets? One piece of such exploratory work is SWIM (Synthesizing What I Mean)[30], which consists of two components: "natural language to API mapper", which suggests a set of APIs given a user query in English, and "synthesizer", which generates code snippets using the suggested APIs.

Snippets recommended by common code search tools can be used not only for further modification, but also for bug localization. Stack Overflow has been widely used in recent years. Many programmers post the description of expected functionalities on Stack Overflow, along with buggy code snippets, and ask where the bug is. Then some expert may post answers, pointing out the bug location. Can we build a system to point out the bug location automatically? One piece of such exploratory work is LOBBY (LOcating Bugs By Searching the most similar sample snippet)[31]. Given a buggy code snippet, LOBBYS takes two steps to locate the bug: 1) normalize the buggy snippet, and then search for the most similar sample snippet that implements the same feature from the code base; 2) align the buggy code and sample code snippets, find the difference between the two code snippets, and generate a bug report based on the difference.

References

[1] Yuan D, Park S, Zhou Y. Characterizing logging practices in open-source software. In *Proc. the 34th ICSE*, June 2012, pp.102-112.

[2] Zong B, Xiao X, Li Z, Wu Z, Qian Z, Yan X, Singh A K, Jiang G. Behavior query discovery in system-generated temporal graphs. *PVLDB*, 2015, 9(4): 240-251.

[3] Xu Z, Wu Z, Li Z, Jee K, Rhee J, Xiao X, Xu F, Wang H, Jiang G. High fidelity data reduction for big data security dependency analyses. In *Proc. the 23rd ACM Conference on Computer and Communications Security* (*CCS*), Oct. 2016.

[4] Ding R, Zhou H, Lou J G, Zhang H, Lin Q, Fu Q, Zhang D, Xie T. Log^2: A cost-aware logging mechanism for performance diagnosis. In *Proc. the USENIX Annual Technical Conference*, July 2015, pp.139-150.

[5] Yuan D, Zheng J, Park S, Zhou Y, Savage S. Improving software diagnosability via log enhancement. In *Proc. the 16th ASPLOS*, March 2011, pp.3-14.

[6] Fu Q, Zhu J, Hu W, Lou J G, Ding R, Lin Q, Zhang D, Xie T. Where do developers log? An empirical study on logging practices in industry. In *Proc. ICSE*, May 31-June 7, 2014, pp.24-33.

[7] Yuan C, Lao N, Wen J R, Li J, Zhang Z, Wang Y M, Ma W Y. Automated known problem diagnosis with event traces. In *Proc. the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems* (*EuroSys*), April 2006, pp.375-388.

[8] Lo D, Cheng H, Han J, Khoo S C, Sun C. Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proc. the 15th SIGKDD*, June 28-July 1, 2009, pp.557-566.

[9] Xu W, Huang L, Fox A, Patterson D, Jordan M. Detecting large-scale system problems by mining console logs. In *Proc. the 22nd ACM SOSP*, Oct. 2009, pp.117-132.

[10] Lou J G, Fu Q, Yang S, Xu Y, Li J. Mining invariants from console logs for system problem detection. In *Proc. USENIX ATC*, June 2010, p.24.

[11] Reidemeister T, Jiang M, Ward P. Mining unstructured log files for recurrent fault diagnosis. In *Proc. the 12th IFIP/IEEE International Symposium on Integrated Network Management*, May 2011, pp.377-384.

[12] Menzies T, Butcher A, Cok D *et al*. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, 2013, 39(6): 822-834.

[13] Lou J G, Lin Q, Ding R, Fu Q, Zhang D, Xie T. Software analytics for incident management of online services: An experience report. In *Proc. the 28th ASE*, Nov. 2013, pp.475-485.

[14] Leesatapornwongsa T, Lukman J F, Lu S, Gunawi H S. TaxDC: A taxonomy of nondeterministic concurrency bugs in datacenter distributed systems. In *Proc. the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2016, pp.517-530.

[15] Leesatapornwongsa T, Hao M, Joshi P, Lukman J F, Gunawi H S. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proc. the 11th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2014, pp.399-414.

[16] Yuan D, Luo Y, Zhuang X, Rodrigues G R, Zhao X, Zhang Y, Jain P U, Stumm M. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proc. the 11th USENIX Conference on Operating Systems Design and Implementation*, Oct. 2014, pp.249-265.

[17] Fang L, Nguyen K, Xu G, Demsky B, Lu S. Interruptible tasks: Treating memory pressure as interrupts for highly scalable data-parallel programs. In *Proc. the 25th Symposium on Operating Systems Principles*, Oct. 2015, pp.394-409.

[18] Nguyen K, Fang L, Xu G H, Demsky B, Lu S, Alamian S, Mutlu O. Yak: A high-performance big-data-friendly garbage collector. In *Proc. the 12th USENIX Conference on Operating Systems Design and Implementation*, Nov. 2016.

[19] Murphy G C, Kersten M, Findlater L. How are Java software developers using the Eclipse IDE? *IEEE Software*, 2006, 23(4): 76-83.

[20] Roy C K, Cordy J R. A survey on software clone detection research. Technical Report, TR 2007-541, School of Computing, Queen's University at Kingston, 2007. http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf, Aug. 2016.

[21] Kapser C J, Godfrey M W. "Cloning considered harmful" considered harmful: Patterns of cloning in software. *Empirical Software Engineering*, 2008, 13(6): 645-692.

[22] Zhang G, Peng X, Xing Z, Zhao W. Cloning practices: Why developers clone and what can be changed. In *Proc. the 28th IEEE International Conference on Software Maintenance*, Sept. 2012, pp.285-294.

[23] Roy C K, Zibran M F, Koschke R. The vision of software clone management: Past, present, and future (keynote paper). In *Proc. the 2014 Software Evolution Week — IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*, Feb. 2014, pp.18-33.

[24] Kim M, Sazawal V, Notkin D, Murphy G C. An empirical study of code clone genealogies. In *Proc. the 10th ESEC/FSE*, Sept. 2005, pp.187-196.

[25] Zhang G, Peng X, Xing Z, Jiang S, Wang H, Zhao W. Towards contextual and ondemand code clone management by continuous monitoring. In *Proc. the 28th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2013, pp.497-507.

[26] Lin Y, Peng X, Xing Z, Zheng D, Zhao W. Clone-based and interactive recommendation for modifying pasted code. In *Proc. the 10th Joint Meeting on Foundations of Software Engineering*, Aug. 30-Sept. 4, 2015, pp.520-531.

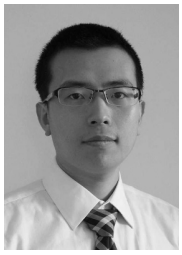
[27] Zhong H, Xie T, Zhang L, Pei J, Mei H. MAPO: Mining and recommending API usage patterns. In *Proc. the 23rd European Conference on Object-Oriented Programming*, July 2009, pp.318-343.

[28] Keivanloo I, Rilling J, Zou Y. Spotting working code examples. In *Proc. the 36th International Conference on Software Engineering*, May 31-June 7, 2014, pp.664-675.

[29] Li X, Wang Z, Wang Q, Yan S, Xie T, Mei H. Relationship-aware code search for JavaScript frameworks. In *Proc. the 24th International Symposium on the Foundations of Software Engineering*, Nov. 2016.

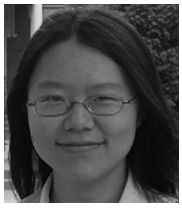
[30] Raghothaman M, Wei Y, Hamadi Y. SWIM: Synthesizing what I mean: Code search and idiomatic snippet synthesis. In *Proc. the 38th International Conference on Software Engineering*, Sept. 2016, pp.357-367.

[31] Wang Q, Li X. Bug localization via searching crowd-contributed code. In *Proc. the 6th Asia-Pacific Symposium on Internetware*, Nov. 2014, pp.1-10.



Xusheng Xiao is a researcher at NEC Laboratories America, Princeton. He received his Ph.D. degree in computer science from North Carolina State University, and was a visiting student in computer science at the University of Illinois at Urbana-Champaign, USA. His research interests are in software engineering and computer security, with a focus on software testing, bug detection, mobile security, and system/enterprise security. His work in mobile security has been selected as one of the top ten finalists for CSAW Best Applied Security Paper Award 2015. His research has been presented at top-tier venues such as ICSE, FSE, ISSTA, ASE, USENIX Security, CCS, and VLDB.

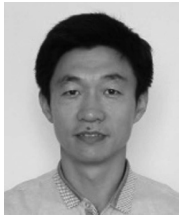
Jian-Guang Lou is a lead researcher in the Software Analytics group at Microsoft Research Asia, Beijing. His research interests include performance analysis and diagnosis of online services, data mining for software engineering. Lou received his Ph.D. degree in pattern recognition from the National Laboratory of Pattern Recognition at the Institute of Automation of the Chinese Academy of Sciences, Beijing. He is a member of IEEE and ACM.



Shan Lu is an associate professor in the Department of Computer Science at the University of Chicago, Chicago. She received her Ph.D. degree at University of Illinois, Urbana-Champaign, in 2008. She was the Clare Boothe Luce Assistant Professor of Computer Sciences at University of Wisconsin, Madison, from 2009 to 2014. Her research focuses on software reliability, particularly detecting, diagnosing, and fixing concurrency bugs and performance bugs in large software systems. Shan has won Alfred P. Sloan Research Fellow in 2014, Distinguished Alumni Educator Award from Department of Computer Science at University of Illinois in 2013, and NSF Career Award in 2010. Her co-authored papers won two ACM-SIGSOFT Distinguished Paper Awards at ICSE 2015 and FSE 2014, one Best Paper Award at USENIX FAST in 2013, an ACM-SIGPLAN Research Highlight Award in 2011, and an IEEE Micro Top Picks in 2006. Shan serves as the vice chair of ACM-SIGOPS 2015∼2018, and the information director of ACM-SIGOPS 2013∼2015. She also served as the technical program co-chair for USENIX Annual Technical Conference in 2015.

David C. Shepherd is a senior principal scientist with ABB Corporate Research, where he leads a group focused on improving developer productivity and increasing software quality. His background, including becoming employee number nine at a successful software tools spinoff and working extensively on popular open source projects, has focused his research on bridging the gap between academic ideas and viable industrial tools. His main research interests to date have centered on software tools that improve developers search and navigation behavior.

Xin Peng is a full professor of the School of Computer Science at Fudan University, Shanghai. He received his B.Sc. degree and Ph.D. degree from Fudan University, Shanghai, in 2001 and 2006 respectively. His research interests include self-adaptive system, mobile and cloud computing, software maintenance and evolution. He serves on several program committees of prestigious international conferences in software engineering area such as ICSM, RE, ICSR, SPLC, ICPC. His work won the Best Paper Award at ICSM 2011.



Qian-Xiang Wang is a professor at the School of Electronics Engineering and Computer Science, Peking University, Beijing. He received his Ph.D. degree in computer science from the Northwestern Polytechnic University, Xi'an, in 1997, and B.S. degree in computer science from National University of Defense Technology, Changsha, in 1991. He is interested in software analysis, WebIDE, and adaptive software. He is a member of CCF and IEEE.