Statement of Research

Kaituo Li

January 16, 2015

NIST estimates that testing, debugging and maintaining typically take more than half of the resources for an average software project. Still, industry pays $22 billion to $60 billion per year in the U.S. due to software failures. We could all spend less if software engineers could have a wider choice of bug detecting tools, better understanding of program behaviors, more effective methods of analyzing huge volumes of data, and more explicit guidance on reasoning about concurrency at cloud-scale. With these goals in mind, I have worked on four directions — bug detection [4, 3], invariant inference [5], test case generation [2], and debugging [1].

More broadly, my research has been on software engineering and applied programming languages, especially in program analysis and automated testing. I am particularly interested in solving four critical challenges. First, how can we increase the confidence of bug warnings produced by automated bug-finding tools in the existence of an error, as well as its severity? Second, how can we automatically infer formal specification from existing test suites and source code? Third, how can we systematically generate representative example data or test cases for a complex program in order to understand its behavior? Fourth, how can we detect, reproduce, and fix concurrency related bugs in local and distributed systems?

1 Research Themes

There are some consistent themes and approaches in my past work that I expect to also pursue vigorously in the future:

- ***Combination*** of static and dynamic analysis: Both static and dynamic program analyses have their advantages. Dynamic analysis excels in control-flow accuracy; static analysis is good at data-flow richness/generality. One cannot make a statement that static analysis is always better than dynamic analysis or vice versa. This had led me to devise combinations of static and dynamic analyses (e.g., [3]) to achieve the best of both worlds.

- ***Machine learning***, data mining, and natural language processing: Computers are good at identifying linear and non-linear patterns in huge data sets. I am interested in designing a suite of algorithms and techniques that draw insights from machine learning, data mining, and natural language algorithms to guide the exploration of other challenges. For example, during a Data Scientist internship, I have created a data mining system that can forecast the time required to complete a job in a SaaS platform, which can vary for a number of reasons: variance in input data set size, hardware resources, software modifications, network availability, network traffic, etc. This work includes statistical modeling, and building infrastructure to efficiently store, analyze and extract data.

- ***Constraint solving***: Many computer science problems, including software and hardware verification, type inference, static program analysis, test-case generation, scheduling, planning, and graph problems, can be transformed into tasks of finding properties for all possible sequences of program behaviors or for some particular sequences. These properties can be expressed in arithmetic constraints, boolean constraints, and equality constraints. Later, we can encode the constraints in a solver that are amenable to automated reasoning. I have used constraint solvers extensively in my research and I have worked with a state-of-the art Satisfiability Modulo Theories (SMT) solver Z3 a lot these days.

- ***Big data*** systems: The complexity of big data systems presents not only challenges unique to this domain (e.g. whether protocols of big data systems work properly on some network topology), but also opportunities to revisit traditional questions of single-server systems (e.g. how a package scales to large problems). On the

one hand, big data systems could benefit from the techniques used in software engineering and programming language. There are a number of well-understood language features and program analyses that could be adapted to big data systems. For example, I have adapted traditional dynamic symbolic execution to map-reduce programs [2]. Such adaptions have many applications: performance monitoring, anomaly detection, and reengineering, to name a few. On the other hand, big data systems are rather different from the single-server system for which most of existing languages and tools were developed. This is a great opportunity to upgrade existing languages and tools by addressing new challenges. For example, I have designed a domain-specific language for big data systems that provides significant ease of use for test condition formation, test construction and test running [1].

2 Enhanced Dynamic Invariant Inference Using Second-Order Constraints

An invariant is a proposition that holds true at certain program points, such as the boolean expression in a Java assert statement. Dynamic invariant inference generates invariants with the help of a test suite that exercises the functionality of an application. An ideal automatic tool should postulate propositions that are both true (on all executions) and relevant (to the programmer, or to some client analysis such as a theorem prover or test data generator). A crucial aspect of the dynamic invariant inference process is its dependency on the quality of test suite. The more representative the test suite is, the more thoroughly the application is exercised, the more accurate and relevant the inferred invariants are. Prior techniques can derive many erroneous and irrelevant invariants without a representative test suite.

To enhance the dynamic invariant detection approach, my work [5] identifies and describes the idea of second-order constraints that can help improve the consistency and relevance of dynamically inferred invariants. I call these constraints second-order because they are constraints over constraints: they relate classes of invariants (first-order constraints). For instance, even though the invariants describing the behavior of two functions f1 and f2 may be unknown, I may know that any valid input for f1 is also valid for f2, i.e., the precondition of f1 implies that of f2. Such second-order constraints can be known even though the invariants are unknown. I define a vocabulary of common and useful second-order constraints and provided an implementation of the constraints. I further introduce techniques for dynamically discovering potential second-order constraints that the programmer can subsequently approve or reject. Evaluations have show that dynamic inference of second-order constraints together with minimal human effort can significantly influence the produced (first-order) invariants: *all of the additional invariants were correct and most were **insightful**, while spurious invariants were **eliminated**.*

3 Residual Investigation: Predictive and Precise Bug Detection

False error reports are the bane of static bug detection tools. Programmers are quickly frustrated and much less likely to trust a static tool if they observe that reported errors are often not real errors, or are largely irrelevant in the given context. This is in contrast to error detection at early stages of program development, where guarantees of detecting all errors of a certain class (e.g., type soundness guarantees) are desirable. Programmers typically welcome conservative sanity checking while the code is actively being developed, but prefer later warnings (which have a high cost of investigation) to be issued only when there is high confidence that the error is real, even at the expense of possibly missing errors.

To reduce false or low-value warnings for static tools, I introduce the concept of residual investigation for program analysis [3, 4]. Residual investigation provides a **general recipe** for **designing predictive and precise dynamic analysis for the first time**. This work received a **ACM SIGSOFT Distinguished Paper Award** at ISSTA 2012. A residual investigation is a dynamic check installed as a result of running a static analysis that reports a possible program error. The purpose is to observe conditions that indicate whether the statically predicted program fault is likely to be realizable and relevant. The key feature of a residual investigation is that it has to be much more precise (i.e., with fewer false warnings) than the static analysis alone, yet significantly more general (i.e., reporting more errors) than the dynamic tests in the programs test suite pertinent to the statically reported error. I enhance the static analyzer FindBugs with several residual investigations, appropriately tuned to the static error patterns in FindBugs, and apply it to 9 large open-source systems and their native test suites. FindBugs with residual

investigation *lowers* the false positive rate from $\geq$ **90%** to $\leq$ **23%** in several large Open Source systems. I observe similarly dramatic improvements with race detection.

4 Dynamic Example Data Generation for Dataflow programs

Dataflow programming (e.g. map-reduce) has emerged as an important data processing paradigm in the area of big data analytics. When a user writes a dataflow program, he/she will typically employ example data or test cases to validate it. Validating with large real data is impractical, both for reasons of efficiency (running on large data sets takes a long time) and for reasons of ease-of-validation (it is hard to tell whether the result is what was expected). One alternative is to sample the real data available. The sample data need to thoroughly exercise the program, covering all key behavior of each dataflow operator. This is very hard to achieve via random sampling, however. Another alternative is to synthesize representative data. Such data synthesis is complicated by the complexity of dataflow language operators as well as by the presence of user-defined functions.

My SEDGE system [2] addresses these completeness problems: for every dataflow operator, I produce data aiming to cover all cases that arise in the dataflow program (e.g., both passing and failing a filter). SEDGE relies on transforming the program into symbolic constraints, and solving the constraints using a symbolic reasoning engine (a powerful SMT solver), while using input data as concrete aids in the solution process. The approach resembles dynamic-symbolic (a.k.a. concolic) execution in a conventional programming language, adapted to the unique features of the dataflow domain. SEDGE is the ***first*** *dynamic symbolic execution engine for dataflow programs.*

5 ReproLite: A Lightweight Tool to Quickly Reproduce Hard System Bugs

Cloud systems have become ubiquitous today – they are used to store and process the tremendous amounts of data being generated by Internet users. These systems run on hundreds of commodity machines, and have a huge amount of non-determinism (thousands of threads and hundreds of processes) in their execution. Therefore, bugs that occur in cloud systems are hard to understand, reproduce, and fix. The state-of-the-art of debugging in the industry is to log messages during execution, and refer to those messages later in case of errors.

In ReproLite, we augment the already widespread process of debugging using logs by enabling testers to quickly and easily specify the conjectures that they form regarding the cause of an error (or bug) from execution logs, and to also automatically validate those conjectures [1]. ReproLite includes a Domain Specific Language (DSL) that allows testers to specify all aspects of a potential scenario (e.g., specific workloads, execution operations and their orders, environment non-determinism) that causes a given bug. Given such a scenario, ReproLite can enforce the conditions in the scenario during system execution. Potential buggy scenarios can also be automatically generated from a sequence of log messages that a tester believes indicates the cause of the bug. We have experimented ReproLite with 11 bugs from two popular cloud systems, Cassandra and HBase. We were able to *quickly* reproduce all of the bugs using ReproLite. We found that *correct buggy scenarios could often (**70%** of the bugs) be constructed from log messages.*

6 Ongoing and Future Research

My future research projects are along two lines. The first part of my future projects is at the core of program analysis and testing. For instance, right now there is not a strong open-source dynamic-symbolic execution engine for Java programs. I am working on a cutting-edge and open-sourced dynamic-symbolic execution engine that is actually usable for most Java programs. Having a strong open-source Java dynamic-symbolic engine will be invaluable because it greatly simplifies analysis of Java programs. The second part of my future projects lies at the intersection of systems, applied programming language, and software engineering. Numerous results are published in the programming language and software engineering community. However, we are still far from using these results on complex systems such as distributed systems, operating systems, and data management systems. There is significant value in getting language and software engineering support for these complex systems. For example, any extensions

of existing techniques for local concurrency bugs to the distributed context can greatly help developers debug their systems. The following are several immediate research opportunities.

Combination of static and dynamic analysis There is still considerable subtlety in devising combinations of static and dynamic analysis. For example, my work on residual investigation showed that the loss of recall in bug detector combining static and dynamic analysis is high due to sparse test suite. I am working on improving the recall of residual investigation by making existing test suites better. To achieve this, I plan to generate tests automatically using dynamic symbolic execution.

Invariant inference Currently there are four popular approaches for invariant inference: generate-and-test using pre-defined invariant templates, generating propositions from symbolic execution along actual program execution paths, abstract interpretation, and machine-learning. Each method has its own advantages and disadvantages in different contexts. I plan to explore the relationship among different approaches in an effort to create tools that follow a hybrid approach.

Program analysis on map-reduce programs I believe that the area of dataflow programs (esp. map-reduce pipelines) can benefit from the application of program analysis techniques, to a much greater extent than current practice. As an example, I would like to explore how to generate program data that reveal changes among program versions. The generated example data can help understand code change, augment existing test suite, and verify intended change (e.g., whether the change brings any anomaly).

Debugging for Distributed Systems Debugging distributed system often incurs high-cost or significant human effort due to exponentially large state spaces. There is a great need for tools that can sift through a large number of state spaces and help programmers come up with hypothesis of bug causes. By combining the proper kinds of data mining, log analysis, and search facilities with program analysis, one could design an inference algorithm to learn what events should be responsible for a bug.

References

[1] Kaituo Li, Pallavi Joshi, Aarti Gupta, and Malay K. Ganai. Reprolite: A lightweight tool to quickly reproduce hard system bugs. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 25:1–25:13, New York, NY, USA, 2014. ACM.

[2] Kaituo Li, C. Reichenbach, Y. Smaragdakis, Y. Diao, and C. Csallner. Sedge: Symbolic example data generation for dataflow programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 235–245, Nov 2013.

[3] Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. Residual investigation: Predictive and precise bug detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 298–308, New York, NY, USA, 2012. ACM.

[4] Kaituo Li, Christoph Reichenbach, Christoph Csallner, and Yannis Smaragdakis. Assessing the effect of screen mockups on the comprehension of functional requirements. *ACM Trans. Softw. Eng. Methodol.*, 24(2), December 2014.

[5] Kaituo Li, Christoph Reichenbach, Yannis Smaragdakis, and Michal Young. Second-order constraints in dynamic invariant inference. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 103–113, New York, NY, USA, 2013. ACM.