Project Description

Applicant: Assoc. Prof. Pavel Parízek

Project Name: Automated Debugging of Large Software Systems that Perform Long-Running Computations

1. Introduction: Motivation and Overview

An essential part of the software development and maintenance process is debugging, which means locating root causes of observed runtime errors and properly fixing the corresponding bugs in program source code. While there is some tool support for debugging programs, it is a very hard, tedious, non-trivial, and time-consuming task, because many related and complicated activities still have to be done manually by software developers. In particular, when an error is observed during program execution or reported by users, the developer has to do the following: inspect parts of the error report (possibly including the execution trace), reproduce the error in the development environment, try to locate the root cause of the error in the program source code (i.e., the faulty statements), modify the source code in order to fix the corresponding bugs, and check that the bug fix is correct and does not introduce any new errors into the program execution. Note also that all these steps are typically performed many times before a particular runtime error is really eliminated. Therefore, it is clear that software developers are in great need of techniques and tools that make debugging easier and faster through (1) a higher level of automation, (2) more efficient test execution and bug reproduction, and (3) support for log analysis.

Related issues are much more apparent in the case of large software systems and long-running computations (processes), where usage of interactive online debuggers (in IDEs such as IntelliJ, Eclipse and Visual Studio) is rather limited and impractical. Actually, this project proposal is motivated by our personal experience with debugging large applications, deployed in real-world settings, in the context of applied research projects with software companies. Debugging of large software systems, especially those performing long computations, very often involves manual inspection of log files of specific runtime events. Since the log files can be really huge in practice, because of a very high amount of possibly relevant data about runtime events and the overhead associated with logging of events, developers usually specify — either by inserting new ”printf” statements into the program source code or through configuration of the logging framework — recording of the minimal number of events and information that is really needed for the particular current step of the whole debugging session. However, like in the general case, developers then have to repeat the following few steps many times: reading some part of a log file, identifying what additional information about events and program states they need to know, updating the program code and/or configuration to record these events and information, and running the whole application or a test suite again. One iteration of this process may take several hours.

In practice, multiple iterations of the process are often needed also because developers learn only gradually, during the debugging session, what information about the program execution they really need to locate the root cause of some error. For example, developers typically do not know in advance the names of procedures whose calls they need to track, and the names of program variables whose runtime values they need to inspect. Developers are also often interested in executions of a specific procedure at all relevant points during the run of a given application, so that they can inspect runtime values of selected program variables in different executions — and to make that easier, they desire tool support for jumping forward and backward over the whole execution trace just between points where the respective procedure was called, ignoring other parts of the trace. Manual usage of interactive debuggers in these scenarios is tedious and cumbersome, since each run of a subject application takes a very long time especially within a debugging session.

The overall topic of this project is the design, implementation, and evaluation of techniques and tools that would help to address the challenges and issues described above. In particular, we will focus on making the whole process of debugging large systems and long-running computations much more automated, faster, and easier for software developers. For this purpose, as our overall vision, we propose the process (methodology) of debugging that consists of the following five steps (that will be performed repeatedly):

1. Developer specifies, either manually or with the help of a tool, the set of application procedures and components whose execution should be tracked, and determines also the names of program variables whose runtime values should be recorded (e.g., because they are likely relevant to the root cause of the error in question). Tracking execution of a specific procedure means recording every call of the procedure, together with actual runtime values of its arguments.

2. Fully automated dynamic analysis is used to record all the necessary information during a run of the subject application. The necessary information would include, besides the values of arguments for each call of every tracked procedure, also relevant parts of the internal state of the program (component) — specifically, runtime values of global variables and contents of global data structures. Existing dynamic analysis frameworks will be used for this purpose.

3. A set of test cases is generated automatically for the respective tracked application components and procedures, using information recorded by the dynamic analysis. Specifically, at least one test case has to be generated for each call of every tracked application procedure, because the developer does not know (at the beginning of a debugging session) which argument values trigger the runtime error.

4. With the help of tools, developers will execute the generated tests and discover some patterns of differences between passing and failing tests, e.g. in terms of procedure call arguments and input program states, to determine the likely root cause of the runtime error in question.

5. In order to confirm or refute a particular hypothesis about the root cause, the developer will thoroughly inspect the given execution trace where the runtime error has been observed. We plan to provide tool support that will enable developers to move forward and backward over the recorded execution trace and inspect specific executions of tracked procedures (including runtime values of program variables). Executions of the individual generated test cases will be used for this purpose. Stepping over the execution trace will be possible only at the granularity of tracked procedures and components, in the sense that other procedures will be ”invisible”. More precisely, developers will be able to inspect a projection of the full execution trace that shows just procedures specified in the first step, which saves them a lot of time. The respective tool support will allow the developers, for example, to quickly jump to another call (execution) of a specific procedure with different runtime argument values and an input program state.

Note that most of the debugging tasks within a single session will be performed off-line, using outputs of the dynamic analysis that is executed just once in advance.

Our goal for this project is to design all techniques and implement tools needed to realize this vision, and solve the associated research problems in order to make the whole process efficient and scalable. Many research and technical challenges are associated with individual steps of the whole process outlined above, covering the full spectrum from theory (algorithms) up to implementation and optimizations. Important challenges include high scalability and good performance in terms of practical memory consumption and low speed overhead. The whole system will be evaluated on a realistic case study. Details about specific goals are provided below in the following sections of this project proposal.

The main benefit of the successful realization of this project would be a much more efficient process of debugging, and therefore also more efficient process of software development and maintenance in general, that exploits the higher level of automation provided by our system. Developers then would not have to, repeatedly, insert additional logging statements into the program source code, wait until yet another long run of a subject application finishes, and read huge log files to determine possible locations of faulty code.

2. Current State of the Art

In this section, we provide an overview of work that has already been done in research areas closely related to the topic of this project. We also discuss limitations of published techniques and existing tools, highlighting challenges and open problems that need to be addressed.

Interactive debuggers. Software developers in the industry predominantly use interactive debuggers, available in mainstream development environments such as Eclipse, IntelliJ IDEA and MS Visual Studio, for online inspection of the program state and executed instructions. For some application domains and settings, including embedded systems and debugging of applications running on remote server hardware, tools like GDB with command-line or text user interface are often used. While all these interactive debuggers provide many useful features, including breakpoints and single-stepping, that help developers to identify faulty statements in the program source code, their usage is quite cumbersome for large applications with long execution traces, especially when one also needs to go back and forth over the execution trace in order to inspect the runtime state and events at various points on the trace.

There also exist some extensions for interactive debuggers that provide a higher degree of automation through scripting and support advanced navigation over execution traces, but still not at the level needed by our proposed approach. For example, the Expositor system [25] is an extension over GDB that enables developers (i) to define scripts in order to filter the execution trace and create projections (subtraces) that capture just the interesting events and states, (ii) query and search the trace, and (iii) move forward and backward in program execution over the projected trace (e.g., by jumping to the next or previous point of interest). Relevant points of interest may be runtime events, such as every call of a specific procedure. The scripting language supported by Expositor allows the developers also to quickly reach a specific point on the program execution trace, where they can inspect the nearby code interactively.

Automated debugging. Lot of work has already been done by researchers worldwide on methods and tools for automating common debugging tasks, with the goal of making the whole process of debugging much easier and less time-consuming. This includes especially the following: techniques for locating root causes of runtime errors in terms of faulty statements and relevant variables [14], generating minimal tests (in terms of source code complexity) that reproduce a given failure [10], static and dynamic program slicing [48], and tools for visualizing and simplifying execution traces. However, most of these techniques were evaluated just on small and middle-sized programs that run just for a short time.

A notable approach, called statistical fault localization [24, 27, 29], is based on statistical comparison of failing and successful test executions. It can help developers by automatically identifying possible faulty statements and ranking them according to their suspiciousness, but recent studies [38, 40, 50] show limitations of this approach — in particular, that it is not effective in practice for large software systems and real faults.

Published techniques for automated simplification of recorded execution traces [20, 21, 22] target especially concurrent programs, in which case error traces are typically very long and hard-to-understand for developers because they contain actions from multiple threads.

Dynamic analysis frameworks. Mature dynamic analysis frameworks are available for programs written in all major languages used to develop complex and large enterprise applications. The list of available frameworks includes RoadRunner [16] and DiSL [35] for programs running on the Java Virtual Machine, Pin [33] and Valgrind [36] for C/C++ programs, and SharpDetect [13] for programs written in C# to be run on the .NET platform. All these frameworks are highly configurable and extensible with custom analysis plugins, and equipped with features needed for use cases relevant to this project, but they also have certain limitations. The biggest challenge related to the usage of dynamic analysis in the context of this project is runtime performance overhead (with respect to speed and memory consumption), which is caused by the need to (i) precisely track the execution of specific procedures and (ii) record detailed information about events such as procedure calls and runtime values of program variables.

Generating test cases and reduction. While the task of generating the source code for a set of test cases, to be performed in step 3 of the proposed debugging process, is mostly an engineering problem for which great tool support is already available, several challenges related to generating and execution of test cases still have to be tackled. In the context of this project, we see as the most relevant challenge the need to generate a set of test cases that is as small as possible but still has the desired coverage of (i) target program behavior and (ii) information recorded by dynamic analysis in step 2. This is very important because repeated execution of an unnecessarily large set of tests would take lot of time, in the order of several hours, while our goal is to make the debugging process very effective from the perspective of software developers. A popular approach is to create a possibly very large set of candidate test cases first, and then apply some techniques of test suite reduction (cf. [19, 44, 47, 51, 52]) to prune the set of tests, for example, to eliminate redundancy.

Machine learning. In recent years, researchers also started applying methods of machine learning in software engineering, program analysis, automated bug detection, and program synthesis. The main benefits are (i) higher automation of the respective tasks and (ii) the ability to extract valuable knowledge from the vast amount of program code and metadata in open-source repositories. Specific published applications of machine learning include the following: inference of certain aspects of program code (e.g., type annotations) [43], improved usefulness of statistical fault localization [28], searching logs from test execution to locate entries that describe errors and failing test runs [1], automated synthesis of programs in domain specific languages through learning

from big datasets [42], generating likely correct patches based on an application-independent model of correct code learned from patches created by human developers [30], creating static analyzers by learning the inference rules from large datasets of programs [5], and learning bug detectors in the form of a classifier that distinguishes correct code from incorrect [41]. This list, however, contains just a selection of techniques that we could use as the basis of our work in this project.

3. Specific Goals of the Project

In order to realize the overall vision outlined in the first section, we need to address many research and technical challenges that correspond to individual steps of the debugging process. The following list of specific goals (where each goal is related to one step of the whole process) covers most of the challenges we already know.

- G1 Design efficient and precise techniques for automatic inference of the names of application procedures and variables that should be tracked (step 1). Since fully automated inference may not be possible in many cases, a practical variant of this goal is to support the following process: first the developers annotate just a small subset of program variables, and then our fully automated technique identifies other variables and procedures into which may flow the runtime values of variables explicitly annotated by developers.

- G2 Extend and optimize the selected dynamic analysis frameworks (e.g., RoadRunner and SharpDetect) to enable gathering of all the necessary information about runtime events and state during execution of a subject program with sufficient performance (step 2). In particular, we would like to minimize the performance overhead associated with observing and recording information during the run of dynamic analysis.

- G3 Design an efficient procedure that generates test cases for application components and procedures based on information recorded by the dynamic analysis (step 3). The generated set of tests should be as small as possible, while ensuring sufficient coverage of program behavior, to achieve practical (short) execution time of tests within a debugging session.

- G4 Implement tools that enable developers to inspect the recorded execution trace and step over it both in the forward and backward direction (step 5).

An important goal, common to all five steps of the whole debugging process, is to design techniques that (i) scale well to large systems and (ii) have just a low overhead in terms of memory consumption, speed, and manual effort required from the developers.

Besides the goals mentioned above, we plan to perform a solid and large-scale experimental evaluation in the form of case studies that will involve realistic software systems of industrial size and complexity.

We would also like to answer several research questions, especially regarding the scalability of individual techniques to large software systems and their practical limitations, during our work on this project. An obvious research question (Q1) is whether the overall process (methodology) of debugging large applications with long-running computations, described in the introductory section, is realistic and useful in practice. Other relevant questions are, for example, the following: (1) whether developers would follow the methodology, and (2) what changes would increase its adoption by developers.

4. Proposed Approach and Timeline

Here we describe the general schedule of this project, in particular the order in which we plan to work on specific tasks, and provide details regarding our planned approach to solve the goals and answer research questions.

Overall schedule. The whole project has a specified duration of three years. We plan to start our work by creating a basic naive implementation of all steps of the whole debugging process outlined in the first section. This will serve as a proof of concept that we then use to perform a set of initial experiments and to show that our overall approach is feasible. All of this would be our main focus during the first year of the project.

Later in the second and third year of the project, our main tasks will be (1) to fulfill the goals G1-G4 specified in the previous section through design and implementation of respective techniques, and (2) to answer the research questions defined in the previous section as well. Based on our previous work and experience, we expect that we will have to develop new program analysis techniques and extend the current state-of-the-art to fulfill these goals, especially to make the whole debugging process work well in practice. We will also attempt

to solve the associated theoretical challenges that will emerge along the way during our work on the design and implementation of various techniques and their optimizations. Each new developed technique or component will be used as a replacement for the corresponding part of the original naive implementation, thus improving the performance and usefulness of the affected individual steps of the debugging process. The detailed plans and schedule for the second and third year of the project will be created with respect to results that we achieve in the first year and also considering the results of other researchers in this field.

Note also that while in the first and second year of this project we will focus mainly on single-threaded programs and solve the important challenges mentioned above in the context of debugging such programs, in the third year we will extend the methodology, developed techniques and tool support towards multithreaded systems. That means we will focus on addressing issues related to concurrency, such as (1) the need to consider interference between concurrent threads and (2) dependency of recorded information (e.g., observed runtime values of program variables and the order of events in a log file) on a specific thread interleaving.

Goal G1. Automatic inference of procedure names and variable names will involve some kind of dependency analysis, similar to those used in program slicing [48], that will be computed with respect to the program location (statement) where the error that is a subject of a particular debugging session manifests itself. In addition, we plan to use well-known static program code analyses (including data-flow and aliasing), together with extraction of relevant information from error traces (reports) and program execution logs.

Goal G2. In order to fulfill this goal, we plan to extend the respective dynamic analysis frameworks (e.g. RoadRunner and SharpDetect) through custom plugins that will be responsible for collecting the necessary information about runtime events and the state of the application subject to analysis. We expect the amount of information recorded and stored by dynamic analysis to be very large, comparable to the size of log files created through the manual approach described in the overview section. However, usage of dynamic analysis together with an automated inference of relevant procedures and variable names (see the goal G1) in this way has one important benefit — developers then would not have to manually insert logging statements into the source code of the target application (that is subject to debugging), enjoying much higher level of automation provided by our system and thus saving lot of time. Furthermore, in our work on the other task related to this goal, which concerns the development and evaluation of performance and scalability optimizations, we will most likely adapt some ideas from techniques proposed specifically for dynamic detection of concurrency errors [15, 17, 6, 46].

Goal G3. The reason why we actually plan to automatically generate and run tests for individual procedures and components in the third step of the whole debugging process is the following: it is technically very hard (1) to exactly reproduce a particular concrete saved runtime state of a program at a specific location in another execution of the program, and (2) to resume or start execution of a given program ”from the middle”, for example at a location right before the call of a specific procedure (cf. [4, 18]). Even in the case of test generation, it is a technical challenge to generate tests such that certain parts of the runtime state are set, during test initialization, based on information provided by dynamic analysis. We need to do that as precisely as possible. The specific approach that we will try as the first option is to generate multiple unit tests for every relevant procedure and component from the subject application, where each test captures one recorded possible input state and reproduces it at the beginning just before entry into a target procedure or component. Here the term ”input state” refers to arguments of procedure calls and relevant parts of the internal state of the subject program (component).

In addition, when designing and implementing the algorithm for test case generation, we plan to adapt, use, and possibly improve existing techniques for test suite reduction [19, 44, 47, 51, 52] so that the actual number of generated tests is not unnecessarily large due to redundancy.

Since all the necessary combinations of test input values for each application procedure or component subject to generated tests will be fully determined by dynamic analysis, we do not need to use well-established automated methods for collecting test inputs that guarantee high coverage, such as methods based on symbolic execution [11], concolic execution [39, 45], and recording values during previous executions of a test suite [37].

Foundational techniques. In the scope of the whole project, we plan to build upon existing foundational techniques from the areas of program analysis and logic-based methods, extending and optimizing them where needed to achieve high scalability. This also includes, for example, usage of symbolic execution and SMT solvers, which is quite common in the fields related to software verification [7], testing [12], and debugging.

Support for multithreading and concurrency. Debugging software systems that involve multiple concurrent threads is much more difficult (than in the case of single-threaded programs) also because a given observed runtime error can be typically reproduced only under a specific thread schedule that is unknown in advance. Specifically, it is therefore very hard to collect relevant information about program behavior that would help in locating a root cause of the error. In our planned debugging approach, this applies both to dynamic analysis (step 2) and execution of tests (steps 4 and 5). A possible way to address this challenge, that we will try first, is to execute the program and tests in a special environment that supports precise control over the thread scheduler. For example, Java Pathfinder [53] is just such a tool that we need. It is a verification framework based upon a special virtual machine (1) that exposes all the necessary APIs (mechanisms) that can be used to enforce a specific thread schedule during execution of the subject program, and (2) it can explore many different interleavings of program threads. We will also build upon the (few) already published debugging techniques (i.e., locating root causes of errors and repairing) that focus explicitly on multithreaded systems (cf. [32, 31, 34]).

Other research questions and directions. Research questions defined in the previous section will be answered mostly based on the results of experiments and studies that we plan to conduct within the scope of this project. For example, to answer the question Q1, we will have to perform a user study likely involving groups of students. Our current plan is to find answers to all these questions over the whole duration of this project.

Another important task is to identify typical ways how developers behave (i.e., what actions they make) when debugging large applications and long-running computations, in particular whether and how much developers use the "`printf`" approach to debugging, how they inspect and process log files, and how they use runtime monitoring frameworks and other tools (including online interactive debuggers) within the debugging sessions. Here, we will build upon previous studies [38, 3, 9, 26]. Based on the results of this study, i.e. after we find out how developers actually behave when debugging, we plan to enable multiple different usage scenarios of the whole system through the configuration of individual techniques and components that we develop.

In the last year of this project, we would also like to investigate possible ways of using selected techniques of machine learning within the context of the proposed debugging methodology. We believe that machine learning could help especially in the following two ways — (1) automate the processing and analysis of test results and execution traces in step 5 of the debugging process, and (2) navigate developers more quickly towards locations of faulty statements based on the specific information and general knowledge extracted from outputs of test execution — but we still need to perform experiments in order to confirm or refute this hypothesis. Since the usage of machine learning techniques in software engineering, specifically in program code analysis (search for bugs) and debugging, is a very active research field, we will create a more detailed plan for this task at the end of the second year, considering the progress and results achieved by other researchers worldwide in the meantime.

5. Research Methodology

During our work on this project, we plan to follow the current recommended practice for solid research in software engineering, testing, and program analysis [49, 8]. We will create a mathematical description, including proofs when applicable, for every algorithmic program analysis technique developed within the scope of this project. A precise mathematical description is relevant specifcally for techniques designed to fulfill the goals G1 and G2. We will create a prototype implementation of the core system that should perform all five steps of the debugging process outlined in the first section, together with prototype implementations of all techniques and optimizations, and release everything in the form of open-source software packages. In addition, we plan to implement a plugin for some popular IDE, such as Eclipse or IntelliJ, that will allow users (software developers) to manage the whole debugging process from within their favorite working environment. This would be needed especially for solid evaluation of the proposed debugging methodology by the means of user studies.

The whole system and every new technique or component will be evaluated experimentally on programs retrieved from public repositories of open source software, such as GitHub and GitLab. Here we plan to include several large realistic software systems, in addition to small programs (microbenchmarks) used for testing.

For the purpose of user case studies, i.e., experiments involving developers and university students, we plan to identify and collect real errors, not yet fixed, in various open-source software projects through manual inspection of the respective public repositories and their issue-tracking systems. Then, during the actual experiments, we plan to (i) ask participants to find the root causes of such real errors with the help of our system and (ii) collect relevant metrics. We will also consider specific details of the methodology, setup, and processing of results in

existing published studies [2, 23], when preparing our own experiments.

All the work in this project will build upon the results of previous research projects of the main applicant, including the project "13-12121P: Practical Program Verification Using Combination of Static and Dynamic Analysis" (2013-2015) and the project "18-17403S Automated Incremental Verification and Debugging of Concurrent Systems" (2018-2020), both also funded by the Czech Science Foundation.

6. Expected Outputs of the Project

All outputs and results of this project can be divided into three groups: (1) new techniques and optimizations needed to make the whole envisioned process of debugging work well, (2) publications in proceedings of international conferences and international scientific journals, and (3) prototype implementations that will be released under an open-source license, so that other researchers and software companies may use them.

We expect to produce 6-8 publications at leading conferences on software engineering, testing, and program analysis techniques (such as ICSE, ASE, OOPSLA, ISSTA, FSE, and TACAS), which means approximately 2-3 each year, and towards the end of the project we will also publish two articles in journals with an impact factor. This combination of publication media is typical for computer science research, where top-quality conferences with reviewed proceedings still represent the primary means of publishing results, especially in the areas of software engineering and program analysis. Such conferences have a very serious peer-reviewing process with a low acceptance ratio of 1/3-1/5, but also a very high impact in the community.

In our publications, we will also report on practical experience with the application of the proposed debugging process on large and complex software systems, which may lead to additional research challenges that could be pursued in the future.

7. Project Team

The project team includes two researchers, Pavel Parízek and Jan Kofroň, from the Department of Distributed and Dependable Systems (D3S) at Faculty of Mathematics and Physics, Charles University, and two students — namely, Filip Kliber, who is a doctoral student at the same department, and one additional doctoral student or an exceptional master student. Each member of the project team will be responsible for specific tasks. Pavel Parízek will lead the whole project, coordinate research activities of students, and work on dynamic analysis, test generation, and tool support for actual debugging (step 5 of the whole process). Jan Kofroň will contribute especially to the design and evaluation of precision improvements and performance optimizations, based on his experience in the fields of software verification and static program analysis. All students will work on the design of new algorithmic techniques, as well as on their implementation and experimental evaluation. Filip Kliber should focus mostly on dynamic analysis techniques and tools. The second student, yet to be hired at the beginning of this project, will focus on some of the other steps of the whole debugging methodology and related goals — including the support for automated inference of tracked procedures and variables, test case generation, and inspection of a recorded execution trace by the means of single-stepping. We expect that students will also help with organization of user studies.

The Department of Distributed and Dependable Systems is sufficiently equipped with hardware and software necessary for successful completion of this project.

Charles University has published the Gender equality plan and all other relevant documents on its website, in the section Equal opportunities / Rovné příležitosti (`https://cuni.cz/UK-11530.html`).

In the scope of this project, we will also continue in our ongoing collaboration with research teams at foreign institutions, including University of Lugano (Prof. Sharygina) and University of Waterloo (Prof. Lhoták).

References

[1] A. Amar and P.C. Rigby. Mining Historical Test Logs to Predict Bugs and Localize Faults in the Test Logs. ICSE 2019, IEEE CS.

[2] N. Ayewah and W. Pugh. The Google FindBugs fixit. ISSTA 2010, ACM.

[3] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman. On the Dichotomy of Debugging Behavior among Programmers. ICSE 2018, ACM.

[4] T. Bergan, D. Grossman, and L. Ceze. Symbolic Execution of Multithreaded Programs from Arbitrary Program Contexts. OOPSLA 2014, ACM.

[5] P. Bielik, V. Raychev, and M.T. Vechev. Learning a Static Analyzer from Data. CAV 2017, LNCS 10426.
[6] S. Biswas, J. Huang, A. Sengupta, and M.D. Bond. DoubleChecker: Efficient Sound and Precise Atomicity Checking. PLDI 2014, ACM.
[7] N. Bjorner, K.L. McMillan, and A. Rybalchenko. Program Verification as Satisfiability Modulo Theories. SMT 2012.
[8] S.M. Blackburn, A. Diwan, M. Hauswirth, P.F. Sweeney, J.N. Amaral, T. Brecht, L. Bulej, C. Click, L. Eeckhout, S. Fischmeister, D. Frampton, L.J. Hendren, M. Hind, A.L. Hosking, R.E. Jones, T. Kalibera, N. Keynes, N. Nystrom, and A. Zeller. The Truth, The Whole Truth, and Nothing But the Truth: A Pragmatic Guide to Assessing Empirical Evaluations. ACM Transactions on Programming Languages and Systems, 38(4), 2016, ACM.
[9] M. Bohme, E.O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller. Where Is the Bug and How Is It Fixed? An Experiment with Practitioners. ESEC/FSE 2017, ACM.
[10] M. Burger and A. Zeller. Minimizing Reproduction of Software Failures. ISSTA 2011, ACM.
[11] C. Cadar, D. Dunbar, and D.R. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. OSDI 2008, USENIX.
[12] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. Communications of the ACM, 56(2), 2013.
[13] A. Cizmarik and P. Parizek. SharpDetect: Dynamic Analysis Framework for C#/.NET Programs. RV 2020, LNCS 12399.
[14] H. Cleve and A. Zeller. Locating Causes of Program Failures. ICSE 2005, ACM.
[15] C. Flanagan and S.N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. PLDI 2009, ACM.
[16] C. Flanagan and S.N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. PASTE 2010, ACM.
[17] C. Flanagan and S.N. Freund. RedCard: Redundant Check Elimination for Dynamic Race Detectors. ECOOP 2013, LNCS 7920.
[18] P. Godefroid. Micro Execution. ICSE 2014, ACM.
[19] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel. On-Demand Test Suite Reduction. Proceedings of ICSE 2012, IEEE CS.
[20] J. Huang and C. Zhang. An Efficient Static Trace Simplification Technique for Debugging Concurrent Programs. SAS 2011, LNCS 6887.
[21] J. Huang and C. Zhang. LEAN: Simplifying Concurrency Bug Reproduction via Replay-supported Execution reduction. OOPSLA 2012, ACM.
[22] N. Jalbert and K. Sen. A Trace Simplification Technique for Effective Debugging of Concurrent Programs. FSE 2010, ACM.
[23] B. Johnson, Y. Song, E.R. Murphy-Hill, and R.W. Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? ICSE 2013, IEEE CS.
[24] J.A. Jones and M.J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. ASE 2005, ACM.
[25] Y.P. Khoo, J.S. Foster, and M. Hicks. Expositor: Scriptable Time-Travel Debugging with First-class Traces. ICSE 2013, IEEE CS.
[26] P.S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' Expectations on Automated Fault Localization. ISSTA 2016, ACM.
[27] T.-D.B. Le, D. Lo, C. Le Goues, and L. Grunske. A Learning-to-Rank Based Fault Localization Approach using Likely Invariants. ISSTA 2016, ACM.
[28] T.-D.B. Le, D. Lo, and F. Thung. Should I follow this fault localization tool's output? - Automated prediction of fault localization effectiveness. Empirical Software Engineering, 20(5), 2015, Springer.
[29] X. Li, S. Zhu, M. d'Amorim, and A. Orso. Enlightened Debugging. ICSE 2018, ACM.
[30] F. Long and M. Rinard. Automatic Patch Generation by Learning Correct Code. POPL 2016, ACM.
[31] C. Torres Lopez, R. Gurdeep Singh, S. Marr, E. Gonzalez Boix, and C. Scholliers. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper). ECOOP 2019, LIPIcs 134.
[32] B. Lucia, B.P. Wood, and L. Ceze. Isolating and Understanding Concurrency Errors Using Reconstructed Execution Fragments. PLDI 2011, ACM.

[33] C.-K. Luk, R.S. Cohn, R. Muth, H. Patil, A. Klauser, P.G. Lowney, S. Wallace, V.J. Reddi, and K.M. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of PLDI 2005, ACM.

[34] N. Machado, B. Lucia, and L. Rodrigues. Production-guided Concurrency Debugging. PPOPP 2016, ACM.

[35] L. Marek, Y. Zheng, D. Ansaloni, L. Bulej, A. Sarimbekov, W. Binder, and P. Tuma. Introduction to dynamic program analysis with DiSL. Science of Computer Programming, 98, 2015, Elsevier.

[36] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. PLDI 2007, ACM.

[37] C. Pacheco, S.K. Lahiri, M.D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. ICSE 2007, IEEE CS.

[38] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? ISSTA 2011, ACM.

[39] C.S. Pasareanu, P.C. Mehlitz, D.H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. ISSTA 2008, ACM.

[40] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, and B. Keller. Evaluating and Improving Fault Localization. ICSE 2017, IEEE CS.

[41] M. Pradel and K. Sen. DeepBugs: A Learning Approach to Name-based Bug Detection. Proceedings of OOPSLA 2018, ACM.

[42] V. Raychev, P. Bielik, M.T. Vechev, and A. Krause. Learning Programs from Noisy Data. Proceedings of POPL 2016, ACM.

[43] V. Raychev, M.T. Vechev, and A. Krause. Predicting Program Properties from "Big Code". Proceedings of POPL 2015, ACM.

[44] G. Rothermel, M.J. Harrold, J. von Ronne, and C. Hong. Empirical Studies of Test-Suite Reduction. Software Testing, Verification & Reliability, 12(4), 2002, Wiley.

[45] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. ESEC/FSE 2005, ACM.

[46] T. Sheng, N. Vachharajani, S. Eranian, R. Hundt, W. Chen, and W. Zheng. RACEZ: A Lightweight and Non-Invasive Race Detection Tool for Production Applications. ICSE 2011, ACM.

[47] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, and D. Marinov. Balancing Trade-offs in Test-Suite Reduction. FSE 2014, ACM.

[48] F. Tip. A Survey of Program Slicing Techniques. Journal of Programming Languages, 3(3), 1995.

[49] J. Vitek and T. Kalibera. Repeatability, Reproducibility, and Rigor in Systems Research. Proceedings of EMSOFT 2011, ACM.

[50] X. Xie, Z. Liu, S. Song, Z. Chen, J. Xuan, and B. Xu. Revisit of Automatic Debugging via Human Focus-Tracking Analysis. ICSE 2016, ACM.

[51] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid. An Empirical Study of JUnit Test-Suite Reduction. ISSRE 2011, IEEE CS.

[52] H. Zhong, L. Zhang, and H. Mei. An Experimental Study of Four Typical Test Suite Reduction Techniques. Information & Software Technology, 50(6), 2008, Elsevier.

[53] Java Pathfinder (JPF): an extensible software verification framework for Java bytecode programs. `https://github.com/javapathfinder/jpf-core/wiki`