Report from Dagstuhl Seminar 13061

Fault Prediction, Localization, and Repair

Edited by

Mary Jean Harrold[1], Friedrich Steimann[2], Frank Tip[3], and Andreas Zeller[4]

1 **Georgia Tech, US,** harrold@cc.gatech.edu
2 **FernUniversität in Hagen, DE,** steimann@acm.org
3 **University of Waterloo, CA,** ftip@uwaterloo.ca
4 **Universität des Saarlandes, DE,** zeller@cs.uni-saarland.de

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 13061 "Fault Prediction, Localization, and Repair".

Software debugging, which involves localizing, understanding, and removing the cause of a failure, is a notoriously difficult, extremely time consuming, and human-intensive activity. For this reason, researchers have invested a great deal of effort in developing automated techniques and tools for supporting various debugging tasks. In this seminar, we discussed several different tools and techniques that aid in the task of Fault Prediction, Localization and Repair.

The talks encompassed a wide variety of methodologies for fault prediction and localizing, such as

- statistical fault localization,
- core dump analysis,
- taint analysis,
- program slicing techniques,
- dynamic fault-comprehension techniques,
- visualization techniques,
- combining hardware and software instrumentation for fault detection and failure prediction,
- and verification techniques for checking safety properties of programs.

For automatically (or semi-automatically) repairing faulty programs, the talks covered approaches such as

- automated repair based on symbolic execution, constraint solving and program synthesis,
- combining past fix patterns, machine learning and semantic patch generation techniques,
- a technique that exploits the intrinsic redundancy of reusable components,
- a technique based on memory-access patterns and a coverage matrix,
- a technique that determines a combination of mutual-exclusion and order relationships that, once enforced, can prevent buggy interleaving.

In addition, this seminar also explored some unusual topics such as Teaching Debugging, using Online Courses. Another interesting topic covered was the low representation of females in computing, and how programming and debugging tools interact with gender differences.

Seminar 03.–08. February, 2013 – www.dagstuhl.de/13061

1998 ACM Subject Classification D.3 Programming Languages, D.2 Software Engineering, D.2.5 Testing and Debugging, D.2.4 Software/Program Verification, F.3 Logics and Meanings of Programs, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Program analysis, Automated debugging, Fault prediction, Fault repair, Fault localization, Statistical debugging, Change impact analysis

Digital Object Identifier 10.4230/DagRep.3.2.1

Edited in cooperation with Mangala Gowri Nanda

1 Executive Summary

Mary Jean Harrold
Friedrich Steimann
Frank Tip
Andreas Zeller

Even today, an unpredictable part of the total effort devoted to software development is spent on debugging, i.e., on finding and fixing bugs. This is despite the fact that powerful static checkers are routinely employed, finding many bugs before a program is first executed, and also despite the fact that modern software is often assembled from pieces (libraries, frameworks, etc.) that have already stood the test of time. In fact, while experienced developers are usually quick at finding and fixing their own bugs, they too spend too much time with fixing the interplay of components that have never been used in combination before, or just debugging the code of others. Better automated support for predicting, locating, and repairing bugs is therefore still required.

Due to the omnipresence of bugs on the one side and the vastly varying nature of bugs on the other, the problems of fault prediction, localization, and repair have attracted research from many different communities, each relying on their individual strengths. However, often enough localizing a bug resembles the solution of a criminal case in that no single procedure or evidence is sufficient to identify the culprit unambiguously. It is therefore reasonable to expect that the best result can only be obtained from the combination of (insufficient) evidence obtained by different, and ideally independent, procedures. One main goal of this seminar is therefore to connect the many different strands of research on fault prediction, localization, and repair.

For researchers it is not always obvious how debugging is embedded in the software production process. For instance, while ranking suspicious program statements according to the likelihood of their faultiness may seem like a sensible thing to do from a research perspective, programmers may not be willing to look at more than a handful of such locations when they have their own inkling of where a bug might be located. On the other hand, commercial programmers may not be aware of the inefficiency of their own approaches to debugging, for which promising alternatives have been developed by academics. Bringing together these two different perspectives is another goal of this seminar.

Last but not least, the growing body of open source software, and with it the public availability of large regression test suites, provide unprecedented possibilities for researchers to evaluate their approaches on industrial-quality benchmarks. In fact, while standard benchmarks such as the so-called Siemens test suite still pervade the scientific literature on debugging, generalization of experimental results obtained on such a small basis is more than questionable. Other disciplines, such as the model checking or the theorem proving communities, have long established competitions based on open benchmarks to which anyone can submit their problems. Based on such benchmarks, progress would be objectively measurable, and advances in research would be better visible. It is another goal of this seminar to establish a common understanding for the need of such benchmarks, and also to initiate the standards necessary for installing them.

2 Table of Contents

3 Overview of Talks

[Day 1] Session: Introduction [Mon, Feb 04, 09:30-10:30]

3.1 Automated Debugging: Are We There Yet?

Alessandro Orso (Georgia Tech, US)

Joint work of Orso, Alessandro; Parnin, Chris
Main reference C. Parnin, A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in Proc. of the 2011 Int'l Symp. on Software Testing and Analysis (ISSTA 2011), pp. 199–209, ACM, 2011.
URL http://dx.doi.org/10.1145/2001420.2001445

Software debugging, which involves localizing, understanding, and removing the cause of a failure, is a notoriously difficult, extremely time consuming, and human-intensive activity. For this reason, researchers have invested a great deal of effort in developing automated techniques and tools for supporting various debugging tasks. Although potentially useful, most of these techniques have yet to fully demonstrate their practical effectiveness. Moreover, many current debugging approaches suffer from some common limitations and rely on several strong assumptions on both the characteristics of the code being debugged and how developers behave when debugging such code. This talk will provide an overview of the state of the art in the broader area of software debugging, discuss strengths and weaknesses of the main existing debugging techniques, present a set of open challenges in this area, and sketch future research directions that may help address these challenges.

[Day 1] Session: Recovery and Reconstruction [Mon, Feb 04, 11:00-12:15]

3.2 Automatic Recovery from Runtime Failures

Alessandra Gorla (Universität des Saarlandes, DE)

Joint work of Gorla, Alessandra; Carzaniga, Antonio; Mattavelli, Andrea; Perino, Nicolò; Pezzè, Mauro

We present a technique to make applications resilient to failures. This technique is intended to maintain a faulty application functional in the field while the developers work on permanent and radical fixes. We target field failures in applications built on reusable components. In particular, the technique exploits the intrinsic redundancy of those components by identifying workarounds consisting of alternative uses of a faulty component that avoid the failure. The technique is currently implemented for Java applications but makes little or no assumptions about the nature of the application and works without interrupting the execution flow of the application and without restarting its components. We demonstrate and evaluate this technique on four mid-size applications and two popular libraries of reusable components affected by real and/or seeded faults. In these cases the technique is effective, maintaining the application fully functional, with between 19% and 48% of the failure-causing faults, depending on the application. The experiments also show that the technique incurs an acceptable runtime overhead in all cases.

3.3 Reconstructing Core Dumps

Jeremias Rößler (Universität des Saarlandes, DE)

Main reference J. Rößler, A. Zeller, G. Fraser, C. Zamfir, G. Candea, "Reconstructing Core Dumps," in Proc. of the 6th IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST'13), March 2013, to appear.
URL http://www.st.cs.uni-saarland.de/publications/files/roessler-icst-2013.pdf

When a software failure occurs in the field, it is often difficult to reproduce. Guided by a memory dump at the moment of failure (a "core dump"), our RECORE test case generator searches for a series of events that precisely reconstruct the failure from primitive data. Applied on seven non-trivial Java bugs, RECORE reconstructs the exact failure in five cases without any runtime overhead in production code.

[Day 1] Session: Repair 1 [Mon, Feb 04, 13:30-15:00]

3.4 Combining Machine Learning with Combinatorial Search in Program Repair

Satish Chandra (IBM India – Bangalore, IN)

We consider the problem of automatically generating repair suggestions for a defective database program, one that behaves incorrectly due to an error in WHERE condition of a SELECT statement. A common setting in database programs is that the output is incorrect only for part of the data, e.g. for certain key values. In this paper, we use techniques from machine learning to take advantage of the information revealed by the defect-free data. Our basic approach is to learn a decision tree from correct behavior–including correct behavior on the defect-inducing data–of the SELECT statement. This decision tree can give valuable hints, if not directly the correct WHERE condition. Our novelty is in the crucial step of determining the correct behavior of the defect-inducing data. We do this using a combination of SAT-based search and prediction generated by support vector machines (SVMs). Our insight is that SVMs can learn from the behavior of the defect-free data to predict the behavior of defect-inducing data with high accuracy, with SAT-based search bridging over any deficit in the accuracy efficiently. We have implemented this approach and have done preliminary experiments on suite of programs and data sets obtained from real- world setting.

3.5 The Design of Bug Fixes–ICSE 2013

Thomas Zimmermann (Microsoft Research – Redmond, US)

Joint work of Murphy-Hill, Emerson; Zimmermann, Thomas; Bird, Christian; Nagappan, Nachiappan
Main reference E. Murphy-Hill, T. Zimmermann, C. Bird, N. Nagappan, "The Design of Bug Fixes," in Proc. of the 35th Int'l Conf. on Software Engineering (ICSE'13), pp. 332–341, IEEE/ACM, 2013.
URL http://dl.acm.org/citation.cfm?id=2486833

When software engineers fix bugs, they may have several options as to how to fix those bugs. Which fix is chosen has many implications, both for practitioners and researchers: What

is the risk of introducing other bugs during the fix? Is the bug fix in the same code that caused the bug? Is the change fixing the cause or just covering a symptom? In this paper, we investigate the issue of alternative fixes to bugs and present an empirical study of how engineers make design choices about how to fix bugs. Based on qualitative interviews with 40 engineers working on a variety of products, 6 bug triage meetings, and a survey filled out by 326 engineers, we found that there are a number of factors, many of them non-technical, that influence how bugs are fixed, such as how close to release the software is. We also discuss several implications for research and practice, including ways to make bug prediction and localization more accurate.

3.6 Lower and upper bounds of coverage-based fault localization accuracy

Friedrich Steimann (Fernuniversität in Hagen, DE)

Ever since the first publications on coverage-based fault localization, an empirical evaluation of the diagnostic accuracy, or of the (theoretical) effort of localizing a fault, has been a condition sine qua non. Although accuracy measures, which are usually based on rankings of program elements to be inspected in search for a fault, may be questioned as indicators of the usefulness of fault locators, they are certainly good for evaluating their performance, in a theoretical setting at least. However, in absence of absolute bounds, evaluations of the accuracies of fault locators are usually relative, that is, by comparison with other fault locators. With his talk, the speaker wishes to share and discuss his thoughts about establishing absolute lower and upper bounds of the accuracy of coverage-based fault localization, which allow one to assess the performance of any individual fault locator independently of all others.

[Day 1] Session: Fault Localization 1 [Mon, Feb 04, 15:30-16:30]

3.7 Avoiding Confoundings in Delta Debugging type of Causality Inference

Xiangyu Zhang (Purdue University, US)

Joint work of William Nick Sumner, Zhang, Xiangyu;
Main reference W.N. Sumner, X. Zhang, "Comparative causality: explaining the differences between executions," in Proc. of the 35th Int'l Conf. on Software Engineering (ICSE'13), pp. 272–281, IEEE/ACM, 2013.
URL http://dl.acm.org/citation.cfm?id=2486825

In this talk, I will briefly introduce a number of challenges we have over-come in making delta debugging more practical. In particular, I will focus on improving the causality inference engine. Delta debugging type of techniques reason about causality through state replacement, that is, replacing part of the program state at an earlier point to observe whether the failure can be induced. However, such replacement often causes undesirable entangling of the replaced state and the original state and does not properly handle errors caused by

execution omission. I will introduce our new causality inference engine that leverages a recently developed program slicing technique to perform better state replacement.

3.8 Combining Hardware and Software Instrumentation for Fault Detection and Failure Prediction

Cemal Yilmaz (Sabanci University – Istanbul, TR)

Joint work of Cemal, Yilmaz; Porter, Adam
Main reference C. Yilmaz, A. Porter, "Combining hardware and software instrumentation to classify program executions," in Proc. of the 18th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering (FSE '10), pp. 67–76, ACM, 2010.
URL http://dx.doi.org/10.1145/1882291.1882304

Many data-driven program analysis approaches have studied ways to infer properties of software systems by using execution data gathered from the running systems, usually with software-level instrumentation. These approaches instrument the source code and/or binaries of programs, collect execution data from program executions every time the instrumentation code is exercised, and analyze the collected data to help shape future software development efforts. Two specific applications of this general approach, which are the focus of this paper, is fault detection, i.e., distinguishing failed executions from successful executions, and runtime failure prediction, i.e., predicting the manifestation of failures at runtime before they actually occur. A fundamental assumption of these and similar approaches is that there are identifiable and repeatable patterns in program executions and that similarities and deviations from these patterns can be used to perform many quality assurance tasks. While existing efforts appear to produce promising results, one less well-understood issue is how best to limit the runtime overhead introduced by these approaches and whether and how tradeoffs can be made between overhead and analysis accuracy. This issue is important because these approaches have been targeted at deployed software systems; excessive runtime overhead is generally undesirable. Therefore, it is important to limit instrumentation overhead as much as possible while still supporting the highest levels of analysis accuracy.

In this work we conjecture that large overhead reductions may derive from reducing the cost of the measurement instruments themselves. To evaluate this conjecture, we have designed and evaluated improved approaches in which most of the data collection work is pushed onto the hardware via the use of hardware performance counters. The data is augmented with further data collected by a minimal amount of software instrumentation that is added to the system's software. We contrast this approach with other approaches implemented purely in hardware or purely in software. Our empirical evaluations, conducted on widely-used open source projects, strongly suggest that 1) there are identifiable and repeatable patterns in program executions, 2) our hybrid hardware and software instrumentation approach is as good or better than other approaches in capturing these patterns; and they do so at a fraction of the cost of using purely software-based instrumentation, and 3) identifying similarities to these patterns and/or deviations from them can reliably detect faults and predict failures at runtime.

[Day 2] Session: Tools 1 [Tue, Feb 05, 09:00-10:30]

3.9 Males and Females Debugging: Are the Tools Getting in the Way?

Margaret M. Burnett (Oregon State University, US)

Joint work of Burnett, Margaret M.; Beckwith, L; Wiedenbeck, S.; Fleming, Scott D.; Cao, Jill; Park, Thomas H.; Grigoreanu, Valentina; Rector, Kyle
Main reference M.M. Burnett, L. Beckwith, S. Wiedenbeck, S.D. Fleming, J. Cao, T.H. Park, V. Grigoreanu, K. Rector, "Gender Pluralism in Problem-Solving Software," Interacting with Computers, 23(5), Elsevier, September 2011, pp. 450–460.
URL http://dx.doi.org/10.1016/j.intcom.2011.06.004

Although there has been recent investigation into how to understand and ameliorate the low representation of females in computing, there has been little research into how programming and debugging tools interact with gender differences. This talk reports the investigations my collaborators and I have conducted into whether and how software tools' features affect males' and females' debugging performance differently, and describes the beginnings of work on promising tool changes that help both male and female software developers across populations, ranging from end-user programmers to software professionals.

[Day 2] Session: Tools 2 [Tue, Feb 05, 11:00-12:15]

3.10 GZoltar: An Eclipse plug-in for Testing and Debugging

Rui Abreu (University of Porto, PT)

Joint work of Abreu, Rui; Riboira, André; Campos, José; Perez, Alexandre
Main reference J. Campos, A. Riboira, A. Perez, R. Abreu, "GZoltar: An Eclipse Plug-In for Testing and Debugging," in Int'l Conf. on Automated Software Engineering (ASE'12), pp. 378–381, ACM, 2012.
URL http://dx.doi.org/10.1145/2351676.2351752

Testing and debugging is the most expensive, error-prone phase in the software development life cycle. Automated testing and diagnosis of software faults can therefore drastically improve the efficiency of this phase, this way improving the overall quality of the software.

In this talk, we present a toolset for automatic testing and fault localization, dubbed GZoltar, which hosts techniques for (regression) test suite minimization and automatic fault diagnosis (namely, spectrum-based fault localization). The toolset provides the infrastructure to automatically instrument the source code of software programs to collect runtime data needed by the underlying techniques. Subsequently, the data is analyzed to both minimize the test suite just executed and compute a visual diagnostic report of the source code entities (methods, statements, etc). The toolset is a plug-and-play plug-in for the Eclipse IDE to ease world-wide adoption.

The toolset can be downloaded at www.gzoltar.com. If interested in the slides of the presentation, visit http://www.gzoltar.com/dagstuhl/.

3.11 Programming with Rosette: Code Checking, Fault Localization, Angelic Execution, and Synthesis in 20 Minutes

Emina Torlak (University of California – Berkeley, US)

Joint work of Torlak, Emina; Bodik, Rastislav

Decision procedures have helped automate key aspects of programming: coming up with a code fragment that implements a desired behavior (synthesis); establishing that an implementation satisfies a desired property (code checking); locating code fragments that cause an undesirable behavior (fault localization); and running a partially implemented program to test its existing behaviors (angelic execution). Each of these aspects is supported, at least in part, by a family of formal tools. Most such tools are built on infrastructures that are tailored for a particular purpose, e.g., Boogie for verification and Sketch for synthesis. But so far, no single infrastructure provides a platform for automating the full spectrum of programming activities, making it hard to share advances (in encodings, abstractions, and domain-specific optimizations) across different families of tools.

This talk introduces Rosette, a new shared infrastructure for computer-aided programming. Rosette is a high-level functional language with symbolic reasoning capabilities, designed to enable rapid prototyping of domain-specific programming tools. The Rosette language is itself a small EDSL (embedded domain-specific language) that inherits and exposes extensive support for meta programming from its host language, Racket. To prototype a tool in Rosette, we first define the target programming model or EDSL by writing an interpreter for it. Depending on the purpose of the tool, the next step is to allow some constructs in the target language to produce symbolic, rather than just concrete, values. In the last step, we define the tool's behavior by formulating a suitable satisfiability query about programs in the target language (e.g., a code checking tool searches for an input that satisfies the program's pre-condition and a negation of its post-condition). Rosette's symbolic engine then does the rest: executing a program in the target EDSL yields a symbolic encoding of the program's semantics, which is used to instantiate and discharge the tool's satisfiability query. We show how to use Rosette to prototype a program checker, a fault localizer, an angelic oracle, and an inductive synthesizer for a tiny DSL.

3.12 An Overview of EzUnit

Marcus Frenkel (FernUniversität in Hagen, DE)

Joint work of Frenkel, Marcus; Steimann, Friedrich
URL http://www.fernuni-hagen.de/ps/prjs/EzUnit5/

Unit testing is one of the important tools for ensuring the quality of software. Unit tests are able to indicate the presence of errors introduced into program code, but normally they don't provide much evidence where the fault actually might be. This task is fulfilled by coverage-based fault locators, which can use the coverage information provided by the unit tests to compute more likely and less likely locations for faults. EzUnit is an Eclipse plug-in which supports programmers in their fault localization task by applying various implemented fault locators to the test results provided by JUnit, as well as allows the implementation and

evaluation of new fault locators. The key feature of EzUnit is its modular structure which allows for an easy extension or replacement of its parts; these parts include interfaces e.g. to the tracer and test runner, fault injectors and locators, or a repository to draw test probands or coverage matrices from for the purpose of evaluation. In this talk, the EzUnit framework shall be introduced, along with its possibilities to implement and evaluate novel approaches for fault localization, and the benefits it has to use EzUnit as an easily alterable platform for fault location evaluations.

[Day 2] Session: Fault Localization and Repair 2 [Tue, Feb 05, 13:30-15:00]

3.13 Basics of Causal Fault Localization from Observational Data

Andy Podgurski (Case Western Reserve University – Cleveland, US)

Statistical Fault Localization (SFL) techniques use statistical measures of association between occurrences of program failures and occurrences of certain runtime events, such as coverage of individual program statements, to identify "suspicious" program locations that may contain faults. However, most proposed SFL metrics are subject to confounding bias that can seriously distort suspiciousness scores so they are not helpful for fault localization. Fortunately, important techniques (developed in recent decades by researchers from several disciplines) for making principled causal inferences from observational data can be used to reduce confounding bias and markedly improve the performance of SFL. In this talk, we explain the basic ideas behind these techniques, including causal effect measures, confounding, potential outcome random variables, conditional exchangeability, causal graphs, Pearl's Back-Door Criterion, and the construction of causal graphs from program dependence graphs.

3.14 Understanding the Relationship Between Recent Fault-Localization Techniques and Causality

George K. Baah (Georgia Tech, US)

Several dynamic analysis techniques have been developed to find the location of faults in programs. The techniques include experimental approaches such as Delta debugging and statistical approaches such as Tarantula. In this talk, I will show analytically the limitations of the techniques (e.g., Tarantula, Ochiai) that rely on statistical metrics for solving the fault-localization problem. I will then present empirical results supporting the analytical results and the challenges that must be overcome to accurately find the causes of software failures.

3.15 SEMFIX: Automated Program Repair using Semantic Analysis

Abhik Roychoudhury (National University of Singapore, SG)

Joint work of Nguyen, H; Qi, Dawei; Roychoudhury, Abhik; Chandra, Satish
Main reference H. Nguyen, D. Qi, A. Roychoudhury, S. Chandra, "SEMFIX: Program Repair via Semantic Analysis," in Proc. of the 35th Int'l Conf. on Software Engineering (ICSE'13), pp. 772–781, IEEE/ACM, 2013.
URL http://dl.acm.org/citation.cfm?id=2486890

Debugging consumes significant time and effort in any major software development project. Moreover, even after the root cause of a bug is identified, fixing the bug is non-trivial. Given this situation, automated program repair methods are of value. In this work, we present an automated repair method based on symbolic execution, constraint solving and program synthesis. In our approach, the requirement on the repaired code to pass a given set of tests is formulated as a constraint. Such a constraint is then solved by iterating over a layered space of repair expressions, layered by the complexity of the repair code. We compare our method with recently proposed genetic programming based repair on SIR programs with seeded bugs, as well as fragments of GNU Coreutils with real bugs. On these subjects, our approach reports a higher success-rate than genetic programming based repair, and produces a repair faster.

[Day 2] Session: Teaching debugging [Tue, Feb 05, 15:30-16:30]

3.16 Teaching Debugging

Andreas Zeller (Universität des Saarlandes, DE)

URL http://www.whyprogramsfail.com/

I present a course on debugging, based on an online course I developed for the online education pioneer Udacity. The course comes in six units:

1. How Debuggers Work (How failures come to be / The scientific method / Interactive debugging)
2. Asserting Expectations (Preconditions, postconditions, invariants / Inferring invariants)
3. Simplifying Failures (Delta debugging / Simplifying fuzz inputs / Causes and causality)
4. Tracking Origins (The art of deduction / Dependencies / Slices)
5. Reproducing Failures (Capturing inputs / Capturing coverage / Statistical debugging)
6. Learning from Mistakes (Bug reports / Bug distributions / Mining software repositories)

I combined this online course with six-week student projects at my university. Students were working on topics as diverse as delta debugging on LaTeX, novel combinations of debugging techniques, or fuzzers for SQL and Python. The key to having students do such large tasks in only six weeks was to use a language like Python, which allows easy access to the interpreter.

The main take away points are:

- Online courses are effective
  - Videos for individual pace; Quizzes for continuous self-assessment
  - But: Presenting or discussing scientific literature does not fit well (reading groups!)

- Online courses are efficient
  Time spent in classroom is better spent on working with students directly
- Python is a great language for tracing
  Assertions, invariants, delta debugging, statistical debugging, etc. are easy
  But: Little support for dynamic dependencies and symbolic reasoning
- Python is great for experimenting
  Delta debugging on states: 90 lines, 3 hours. (Read this again and again.)
- Projects unleash student creativity
  Attract students to your research topics

[Day 3] Session: Solvers and Satisfiability [Wed, Feb 06, 09:00-10:30]

3.17 SAT Solvers for Software Reliability, Security and Repair

Vijay Ganesh (University of Waterloo, CA)

In recent years SAT solvers have had a huge impact on software engineering research and practice. The reason for this is the impressive improvement in the efficiency of SAT solvers and their use as part of SMT solvers (e.g., DPLL(T) and as a backend for solvers for theories of bit-vectors and arrays). In this talk, I will highlight the 4 heuristics that have played a key role in this dramatic improvement in SAT solver performance, namely, 1) conflict-driven clause-learning with backjumping, 2) branching heuristics, 3) restarts and 4) efficient implementation of Boolean constant propagation.

I will also introduce my work on programmatic SAT solvers, where a user can specialize the SAT solver to their specific domain using an API that allows them to add code to influence the SAT solver's search and branching heuristics. This idea is a variation on the idea of DPLL(T), and I will highlight the differences between the two.

3.18 Fault Localization using Maximum Satisfiability

Rupak Majumdar (MPI for Software Systems – Kaiserslautern, DE)

Joint work of Majumdar, Rupak; Jose, Manu
Main reference M. Jose. R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," in Proc. of the 32nd ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'11), pp. 437–446, ACM, 2011.
URL http://dx.doi.org/10.1145/1993498.1993550

Several verification tools exist for checking safety properties of programs and reporting errors. However, a large part of the program development cycle is spend in analyzing the error trace to isolate locations in the code that are potential causes of the bug. Currently, this is usually performed manually, by stepping through the error trace in a debugger. We present Bug-Assist, a tool that assists programmers localize error causes to a few lines of code. Bug-Assist takes as input an ANSI-C program annotated with assertions, performs model checking internally to find potential assertion violations, and for each error trace returned by

the model checker, returns a set of lines of code which can be changed to eliminate the error trace. Bug-Assist's algorithm formulates error localization as a MAX-SAT problem and uses scalable MAX-SAT solvers.

We discuss a set of open problems related to this idea, for example, the use of similar techniques for compiler bugs, for regression testing, for concurrency testing, as well as possible techniques for scaling up to large programs.

[Day 3] Session: Repair 3 [Wed, Feb 06, 11:00-12:15]

3.19 On the biodiversity of source code: rigid or plastic repair?

Benoit Baudry (INRIA Bretagne Atlantique – Rennes, FR), Martin Monperrus (INRIA Nord Europe – Lille, FR)

Joint work of Baudry, Benoit; Monperrus, Martin
Main reference M. Monperrus, M. Mezini, "Detecting Missing Method Calls as Violations of the Majority Rule", ACM Transactions on Software Engineering and Methodology (TOSEM), 22(1), pp. 7:1–7:25, 2013.
URL http://dx.doi.org/10.1145/2430536.2430541

In this talk, we first present a static code analysis that finds a specific kind of bug (missing method calls when using third-party frameworks) based on a radical abstraction over the code (type usages: a list of method calls on a variable of a given type). Furthermore, the analysis provides a partial diagnostic on how to repair the bug. Those diagnostics enabled us to provide valid patches on a million LOC unknown code base – valid in the sense that they were incorporated in the code base by the lead developers. Further experiments were done to understand the dynamics of type-usages at the code ecosystem level. Those experiments revealed that some Java library classes give birth to many different type usages that we call "usage diversity". A high "usage diversity" indicates a great "plasticity" of the class itself, and we have empirical evidence of a relation between the usage diversity of a class and its success (the class is used by many client classes). Those novel observations open new research questions: how does the code plasticity relate to the kind of repair techniques? how to step from code diversity to automated code diversification?

3.20 Leveraging Software Text Analytics To Detect, Diagnose, and Fix Bugs

Lin Tan (University of Waterloo, CA)

Joint work of Liu, Chen; Yang, Jinqiu; Tan, Lin; Hafiz, Munawar
Main reference C. Liu, J. Yang, L. Tan, M. Hafiz, "R2Fix: Automatically Generating Bug Fixes from Bug Reports," in Proc. of the 6th IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST'13), March, 2013, to appear.
URL https://ece.uwaterloo.ca/~lintan/publications/r2fix-icst13.pdf

Software bugs seriously hurt software reliability. In this talk, I will present our recent research on leveraging software textual information in program comments, source code, and bug reports to detect, diagnose, and fix software bugs. R2Fix automatically generates bug-fixing patches from bug reports written free-form in a natural language. R2Fix combines past

fix patterns, machine learning techniques, and semantic patch generation techniques to fix bugs automatically. We evaluate R2Fix on three projects, i.e., the Linux kernel, Mozilla, and Apache, for three important types of bugs: buffer overflows, null pointer bugs, and memory leaks. R2Fix generates 57 patches correctly, 5 of which are new patches for bugs that have not been fixed by developers yet. We reported all new patches to the developers; 4 have already been accepted and committed to the code repositories. The 57 correct patches generated by R2Fix could have shortened the bug fixing time by up to 63 days on average. In addition, they could save developers' time and effort in diagnosing and fixing bugs. R2Fix is safe as patches are not applied until developers have confirmed the correctness of the patches. iComment, aComment, and @tComment automatically extract specifications from source code and code comments written in a natural language, and use these specifications to detect comment-code inconsistencies, i.e., software bugs and bad comments.

[Day 4] Session: Program Analysis 1 [Thu, Feb 07, 09:00-10:45]

3.21 Genetic Mutation Conditioned Amorphous Parametric Hybrid "Dual" Slicing

Jens Krinke (University College London, GB)

This talk highlights the most important ideas, problems, and solutions in the history of program slicing. It turns out that most of the technical problems have been solved in the past 10 years while there have not been a lot of advances in the academic research on the long standing challenges.

In addition, the talk presents current trends in the usage of dependence analysis and program slicing. It shows how dependence form large clusters of statements that are indistinguishable in terms of impact. The talk also shows how slicing can be used in information flow control to check security policies including declassification.

3.22 Information Flow Analysis for JavaScript

Christian Hammer (Universität des Saarlandes, DE)

Joint work of Just, Seth; Cleary, Alan; Shirley, Brandon; Hammer, Christian
Main reference S. Just, A. Cleary, B. Shirley, C. Hammer, "Information flow analysis for JavaScript," in Proc. of the 1st ACM SIGPLAN Int'l Workshop on Programming Language and Systems Technologies for Internet Clients (PLASTIC'11), pp. 9–18, ACM, 2011.
URL http://dx.doi.org/10.1145/2093328.2093331

Modern Web 2.0 pages combine scripts from several sources into a single client-side JavaScript program with almost no isolation. In order to prevent attacks from an untrusted third-party script or cross-site scripting, tracking provenance of data is imperative. However, currently no browser offers this security mechanism. This work presents the first information flow control mechanism for full JavaScript. We track information flow dynamically as much as possible but rely on intra-procedural static analysis to capture implicit flow. Our analysis

handles even the dreaded eval function soundly and incorporates flow based on JavaScript's prototype inheritance. We implemented our analysis in a production JavaScript engine and report both qualitative as well as quantitative evaluation results.

3.23 Gaining inSight into Programs that Analyze Programs – By Visualizing the Analyzed Program

Mangala Gowri Nanda (IBM India Research Lab. – New Delhi, IN)

Joint work of Nanda, Agastya; Nanda, Mangala Gowri
Main reference A. Nanda, M, Gowri Nanda, "Gaining insight into programs that analyze programs: by visualizing the analyzed program," in Proc. of the 24th ACM SIGPLAN Conf. Companion on Object Oriented Programming Systems Languages and Applications, pp. 1023–1030, ACM, 2009.
URL http://dx.doi.org/10.1145/1639950.1640074

Visualization of a program typically entails low level views of the program execution state showing, for example, method invocations or relations amongst heap objects. In most cases, this would imply visualization of the executable program. However there is a certain genre of programs that analyze or transform other programs. These programs could be compilers, static bug detectors, test suite analyzers, model to model transformers etc. In such cases, very often, it helps to visualize what is happening to the input program rather than the analyzer program. It is for such programs that we describe a configurable, analysis framework. For ease of exposition, we call the analyzer program the "manipulate" program, and the input program the "puppet" program. To facilitate the visualization, we instrument the manipulate program to generate a dump as it analyzes the puppet program. Using the "dump", we reconstruct the interprocedural control flow graph of the puppet program and then visualize the flow of the manipulate program over the puppet program. In particular, our visualization consists of rendering the control flow graph of the puppet program, and then tracking the behavior of the manipulate program by watching its interaction with the puppet program. We use colors to highlight different events in the manipulate program. Using this scheme, we are able to (1) gain insight into the manipulate program; (2) collect useful information / statistics about the puppet program. We have implemented the visualizer in a tool called "Insight". We ran Insight on a static debugging tool (the manipulate program) called Xylem. Xylem applies static analysis to find potential null pointer exceptions in a puppet program, as for example, the Apache Ant program. We report the insights gained by running Xylem through Insight on Ant and other puppet programs.

[Day 4] Session: Program Analysis 2 [Thu, Feb 07, 11:15-12:15]

3.24 Blended Taint Analysis for JavaScript

Barbara G. Ryder (Virginia Polytechnic Institute – Blacksburg, US)

Joint work of Ryder, Barbara G.; Wei, Shiyi

JavaScript is ubiquitous in website code, introducing challenges to static analyses by its heavily used dynamic features such as dynamic code generation and variadic functions. Taint analysis is an important tool for finding data integrity problems in programs. The blended analysis paradigm combines a dynamic representation of program calling structure , with

static analysis applied to that calling structure. Recently, we developed a blended taint analysis for JavaScript website code that is more scalable and more accurate than a pure static analysis. Because taint analysis is posed as reachability on a program representation between tainted sources and sensitive sinks, there is the possibility of the analysis gathering a witness path for each reported vulnerability.

3.25 Accurate and Scalable Security Analysis of Web Applications

Marco Pistoia (IBM TJ Watson Research Center – Hawthorne, US)

Joint work of Pistoia, Marco; Omer Tripp; Patrick Cousot; Radhia Cousot; Salvatore Guarnieri

Security auditing of industry-scale software systems mandates automation. Static taint analysis enables deep and exhaustive tracking of suspicious data flows for detection of potential leakage and integrity violations, such as cross-site scripting (XSS), SQL injection (SQLi) and log forging. Research in this area has taken two directions: program slicing and type systems. Both of these approaches suffer from a high rate of false findings, which limits the usability of analysis tools based on these techniques. Attempts to reduce the number of false findings have resulted in analyses that are either (i) unsound, suffering from the dual problem of false negatives, or (ii) too expensive due to their high precision, thereby failing to scale to real-world applications. In this paper, we investigate a novel approach for enabling precise yet scalable static taint analysis. The key observation informing our approach is that taint analysis is a demand-driven problem, which enables lazy computation of vulnerable information flows, instead of eagerly computing a complete data-flow solution, which is the reason for the traditional dichotomy between scalability and precision. We have implemented our approach in ANDROMEDA, an analysis tool that computes data-flow propagations on demand, in an efficient and accurate manner, and additionally features incremental analysis capabilities. ANDROMEDA is currently in use in a commercial product. It supports applications written in Java,.NET and JavaScript. Our extensive evaluation of ANDROMEDA on a suite of 16 production-level benchmarks shows ANDROMEDA to achieve high accuracy and compare favorably to a state-of-the-art tool that trades soundness for precision.

[Day 4] Session: Mending the Web [Thu, Feb 07, 13:30-15:00]

3.26 An Overview of the Apollo Project: Test Generation and Fault Localization for Web Applications

Shay Artzi (IBM – Littleton, US), Frank Tip (University of Waterloo, CA), and Julian Dolby (IBM TJ Watson Research Center – Hawthorne, US)

Joint work of Tip, Frank; Dolby, Julian; Artzi, Shay; Pistoia, Marco

The Apollo project at IBM Research was aimed at developing practical automated techniques for finding, localizing, and fixing bugs in web applications. We adapted an existing dynamic test generation technique that applies concrete and symbolic execution to the domain of web applications written in PHP, and used it to find dozens of failures in open-source PHP

applications. To help programmers with localizing the faults that cause these failures, we adapted existing fault localization techniques that predict in which statements a fault is located by applying a statistical analysis to execution data gathered from multiple tests. Our results indicate that, using our best technique, nearly 90% of faults are localized to within 1% of all executed statements. We also address the question of how to localize a fault when the programmer is confronted with a failure but no test suite is available that can be used for fault localization. In such cases, our new directed test generation technique is capable of generating small test suites with high fault-localization effectiveness.

This presentation is based on papers published at ISSTA'08, ICSE'10, ISSTA'10, IEEE TSE'10 and IEEE TSE'12.

3.27 Automated Repair of HTML Generation Errors in PHP Applications Using String Constraint Solving

Hesam Samimi (Univ of California – Los Angeles, US)

Joint work of Schäfer, Max; Artzi, Shay; Millstein, Todd; Tip, Frank; Hendren, Laurie
Main reference H. Samimi, M. Schäfer, S. Artzi, T. Millstein, F. Tip, L.e Hendren, "Automated repair of HTML generation errors in PHP applications using string constraint solving," in Proc. of the 2012 Int'l Conf. on Software Engineering (ICSE'12), pp. 277–287, IEEE Press, 2012.
URL http://dl.acm.org/citation.cfm?id=2337223.2337257

PHP web applications routinely generate invalid HTML. Modern browsers silently correct HTML errors, but sometimes malformed pages render inconsistently, cause browser crashes, or expose security vulnerabilities. Fixing errors in generated pages is usually straightforward, but repairing the generating PHP program can be much harder. We observe that malformed HTML is often produced by incorrect "string literal prints", i.e., statements that print string literals, and present two tools for automatically repairing such HTML generation errors. PHPQuickFix repairs simple bugs by statically analyzing individual prints. PHPRepair handles more general repairs using a dynamic approach. Based on a test suite, the property that all tests should produce their expected output is encoded as a string constraint over variables representing string literal prints. Solving this constraint describes how string literal prints must be modified to make all tests pass. Both tools were implemented as an Eclipse plugin and evaluated on PHP programs containing hundreds of HTML generation errors, most of which our tools were able to repair automatically.

[Day 4] Session: Miscellaneous [Thu, Feb 07, 15:30-16:30]

3.28 Demo of the Tarantula Fault-Localization Tool

Jake Cobb (Georgia Tech, US)

Joint work of Cobb, Jake; Harrold, Mary Jean; Jones, James A.
URL http://aristotleresearch.com/

I will give a demo of an implementation of the Tarantula statistical fault-localization technique. It includes fault-localization calculations using a modification of the Ochiai formula, a SeeSoft visualization of results, navigation based on the fault-localization results, and a fault-localization-based test-case clustering algorithm. The tool currently supports Java programs

using the popular JUnit testing framework. The data import step supports integration with both Ant and Maven, two of the dominant build tools for Java projects. Finally, coverage information can be obtained from either the proprietary Clover or open-source Cobertura coverage tools.

3.29 Three Projects Related to Fault Prediction, Localization, and Repair

Milos Gligoric (University of Illinois – Urbana, US)

This talk briefly presents three projects from our research group that are closely related to the three topics of the seminar: instead of "Fault Prediction", "Fault Localization", and "Fault Repair", I will present "Failure Prediction", "Dependency-Aware Fault Localization", and "Test Repair". In "Failure Prediction" we estimate the likelihood of triggering a failure in a specific part of a system by running the system with a large number of real-world inputs. The results of applying this technique on the Eclipse refactoring engine estimate the number of failures for each refactoring and show that the Eclipse refactoring engine is less buggy than we expected. In "Dependency-Aware Fault Localization" we use knowledge about changes of the system under test to improve fault localization by identifying unchanged statements that cannot affect the results of any test. The results on a number of programs demonstrate that dependency-aware fault localization can reduce the number of statements to be inspected before the fault is identified. In "Test Repair" we present a solution for repairing broken unit tests after the system under test is changed by analyzing dynamic execution of the tests. The results show that this approach can repair many common test failures and that its suggested repairs match developers' expectations.

[Day 5] Session: Concurrency [Fri, Feb 08, 09:30-10:30]

3.30 Automated Concurrency-Bug Fixing

Ben Liblit (University of Wisconsin–Madison, US)

Joint work of Jin, Guoliang; Zhang, Wei; Deng, Dongdong; Liblit, Ben; Lu, Shan
Main reference G. Jin, W. Zhang, D. Deng, B. Liblit, S. Lu, "Automated Concurrency-Bug Fixing," in Proc. of the 10th USENIX Symp. on Operating Systems Design and Implementation (OSDI'12), pp. 221–236, USENIX Assoc., 2012.
URL https://dl.acm.org/citation.cfm?id=2387880.2387902

Concurrency bugs are widespread in multithreaded programs. Fixing them is time-consuming and error-prone. We present CFix, a system that automates the repair of concurrency bugs. CFix works with a wide variety of concurrency-bug detectors. For each failure-inducing interleaving reported by a bug detector, CFix first determines a combination of mutual-exclusion and order relationships that, once enforced, can prevent the buggy interleaving. CFix then uses static analysis and testing to determine where to insert what synchronization operations to force the desired mutual-exclusion and order relationships, with a best effort to avoid deadlocks and excessive performance losses. CFix also simplifies its own patches by merging fixes for related bugs.

Evaluation using four different types of bug detectors and thirteen real-world concurrency-bug cases shows that CFix can successfully patch these cases without causing deadlocks or excessive performance degradation. Patches automatically generated by CFix are of similar quality to those manually written by developers.

3.31 Localizing Non-deadlock Concurrency Bugs

Mary Jean Harrold (Georgia Tech, US)

Joint work of Park, Sangmin; Vudue, Richard; Harrold, Mary Jean
Main reference S.Park, R. Vuduc, M.J. Harrold, "A Unified Approach for Localizing Non-deadlock Concurrency Bugs," in Proc. of the 5th IEEE Int'l Conf. on Software Testing, Verification and Validation (ICST'12), pp. 51–60, IEEE, 2012.
URL http://dx.doi.org/10.1109/ICST.2012.85

In this talk, I will present our automated dynamic fault-comprehension technique, that provides a way to explain concurrency bugs with additional information over existing fault-localization techniques, and thus, bridges the gap between fault-localization and fault-fixing techniques. The technique inputs a list of memory-access patterns and a coverage matrix, groups those patterns responsible for the same concurrency bug, and outputs the grouped patterns along with suspicious methods and bug graphs. Griffin is the first technique that handles multiple concurrency bugs. I will also describe the implementation of our technique in Java and C++, and show the empirical evaluation which shows results show that out technique clusters failing executions and memory-access patterns for the same bug with few false positives, provides suspicious methods that contain the locations to be fixed, and runs efficiently.

4 Discussion and Open Problems

The main theme of the discussion was that techniques for fault prediction, localization, and repair as presented at this Seminar were already quite advanced, but their adoption, in industry especially, was lagging behind. One usual suspect for this is the lack of large-scale empirical studies demonstrating the usefulness of the various approaches, but this would require controlled studies, which are inherently difficult to conduct with the limited (especially monetary) resources of a computer science department. Another probable cause is a lack of (continued) education or, rather, the general lag time between the discovery and publication of a new insight, and its widespread adoption in industry.

5 Conclusion

The workshop provided a productive venue for an open exchange of ideas about various topics related to fault prediction, localization, and repair. Overall, we felt that there was an excellent balance of topics and mix of participants. The format of the workshop went well. We deliberately avoided over-scheduling by keeping the presentations relatively short (with the exception of a few well-chosen tutorial-style presentations), and long breaks between sessions. These breaks provided plenty of time for fruitful interactions between the meeting participants. Overall, we consider the workshop to have been very successful, and one of the most enjoyable and productive Dagstuhl workshops we have attended.

Participants

- Rui Abreu
University of Porto, PT
- Shay Artzi
BM – Littleton, US
- George K. Baah
Georgia Inst. of Technology, US
- Benoit Baudry
INRIA Bretagne Atlantique – Rennes, FR
- Margaret M. Burnett
Oregon State University, US
- Satish Chandra
IBM India – Bangalore, IN
- Jake Cobb
Georgia Inst. of Technology, US
- Julian Dolby
IBM TJ Watson Research Center – Hawthorne, US
- Marcus Frenkel
FernUniversität in Hagen, DE
- Vijay Ganesh
University of Waterloo, CA
- Milos Gligoric
Univ. of Illinois – Urbana, US
- Alessandra Gorla
Universität des Saarlandes, DE
- Mangala Gowri Nanda
IBM India Research Lab. – New Delhi, IN
- Christian Hammer
Universität des Saarlandes, DE
- Mary Jean Harrold
Georgia Inst. of Technology, US
- Jens Krinke
University College London, GB
- Ben Liblit
University of Wisconsin – Madison, US
- Rupak Majumdar
MPI for Software Systems – Kaiserslautern, DE
- Martin Monperrus
INRIA Nord Europe – Lille, FR
- Alessandro Orso
Georgia Inst. of Technology, US
- Marco Pistoia
IBM TJ Watson Research Center – Hawthorne, US
- Andy H. Podgurski
Case Western Reserve University – Cleveland, US
- Jeremias Rößler
Universität des Saarlandes, DE
- Abhik Roychoudhury
National Univ. of Singapore, SG
- Barbara G. Ryder
Virginia Polytechnic Institute – Blacksburg, US
- Hesam Samimi
Univ. California – Los Angeles, US
- Friedrich Steimann
Fernuniversität in Hagen, DE
- Lin Tan
University of Waterloo, CA
- Frank Tip
University of Waterloo, CA
- Emina Torlak
University of California – Berkeley, US
- Cemal Yilmaz
Sabanci Univ. – Istanbul, TR
- Andreas Zeller
Universität des Saarlandes, DE
- Xiangyu Zhang
Purdue University, US
- Thomas Zimmermann
Microsoft Res. – Redmond, US

First row: Marcus, Mary Jean, Jake, Mangala, Barbara, Lin, Margaret, Alessandra.
Second row: Jeremias, Milos, Vijay, Shay, Abhik, Christian, Cemal, Xiangyu.
Third row: Alex, Emina, Frank, Satish, Rui, Ben, Julian, Benoit.
Last row: Jens, Andreas, Thomas, Marco, Andy, Friedrich, Hesam, George, Rupak, Martin.