Report from Dagstuhl Seminar 17502

Testing and Verification of Compilers

Edited by

Junjie Chen[1], Alastair F. Donaldson[2], Andreas Zeller[3], and Hongyu Zhang[4]

1 **Peking University, CN,** `chenjunjie@pku.edu.cn`
2 **Imperial College London, GB,** `alastair.donaldson@imperial.ac.uk`
3 **Universität des Saarlandes, DE,** `zeller@cs.uni-saarland.de`
4 **University of Newcastle, AU,** `hongyujohn@gmail.com`

Abstract

This report documents the Dagstuhl Seminar 17502 "Testing and Verification of Compilers" that took place during December 10 to 13, 2017, which we provide as a resource for researchers who are interested in understanding the state of the art and open problems in this field, and applying them to this and other areas.

Seminar December 10–13, 2017 – http://www.dagstuhl.de/17502
1998 ACM Subject Classification D.2.4 Software/Program Verification, D.2.5 Testing and Debugging, D.3.4 Processors
Keywords and phrases code generation, compiler testing, compiler verification, program analysis, program optimization
Digital Object Identifier 10.4230/DagRep.7.12.50
Edited in cooperation with Hugues Evrard and David R. MacIver

1 Executive Summary

David R. MacIver (Imperial College London, GB)

This report documents the Dagstuhl Seminar 17502 "Testing and Verification of Compilers".

Compilers underpin all software development, but bugs in them can be particularly hard to notice if they result in "silent failure", where a program appears to work but is subtly miscompiled. Thus a compiled program may behave erroneously even when the source form of it appears entirely correct.

Despite the common wisdom that "it is never the compiler's fault", bugs in compilers are in fact relatively common, and finding them is a challenging and active area of research.

This seminar brought together researchers in that area with a broader group of researchers and practitioners in software testing and verification, and in compiler development itself, to share their experiences and discuss the open questions and challenges that the field presents. The goal was to brainstorm new ideas for how to approach these challenges, and to help foster longer-term collaborations between the participants.

The seminar involved a number of talks from participants about their particular areas of work and research, followed by working groups where various specific challenges were discussed. It then concluded with an open panel session on the challenges and concepts of compiler testing and verification.

This report presents the collection of abstracts associated with the participant presentations, followed by notes summarising each discussion session and the concluding panel, which we provide as a resource for researchers who are interested in understanding the state of the art and open problems in this field.

2 Table of Contents

Executive Summary
David R. MacIver 50

Overview of Talks

How Google Tests Compilers
Edward E. Aftandilian 54

Compiler testing for safety-critical applications
Marcel Beemster 54

Learning to accelerate compiler testing
Junjie Chen 54

A Generator of Highly Effective Fuzz Testers
Eric Eide 55

Automated Testing of Graphics Shader Compilers
Hugues Evrard 55

Introduction to Coccinelle
Julia Lawall 56

Differential Testing of Interactive Debuggers
Daniel Lehmann 56

CUDA Compiler Verification
Thibaut Lutz 57

An Introduction to Software Verification with Whiley
David J. Pearce 57

TreeFuzz: Learning Probabilistic Models of Input Data for Fuzz Testing
Michael Pradel 58

A Few Stories about Compiler Bugs
John Regehr 58

Automatic non-Functional Testing of Configurable Generators
Gerson Sunyé 59

Learning to Synthesize Programs
Yingfei Xiong 59

Debugging Debug Information
Francesco Zappa Nardelli 60

Tutorial: Compiler Optimisations and Shared Memory Concurrency
Francesco Zappa Nardelli 60

Fuzzing with Inferred Grammars
Andreas Zeller 61

Working groups

Benchmarking Compiler Testing
Junjie Chen 61

Beyond Compiler Testing
Michael Pradel 61

Test-Case Reduction
David R. MacIver 62

Program Generation
David R. MacIver 63

Panel discussions

Challenges and Concepts of Compiler Testing and Verification
Yingfei Xiong 63

Participants 65

3 Overview of Talks

3.1 How Google Tests Compilers

Edward E. Aftandilian (Google Research - Mountain View, US)

Google has several compiler teams that are responsible for shipping new and updated compilers to the rest of Google on a regular basis.

In this talk, I will discuss how Google's compiler teams validate new and updated compilers, what works well in this process, and where it could be improved.

3.2 Compiler testing for safety-critical applications

Marcel Beemster (Solid Sands - Amsterdam, NL)

Now that so much software is used to control our cars, we have to consider the "safety" of the compilers that are used to generate the machine code from the sources. Processes exist to ascertain so called "Functional Safety". These are based on empirical findings over the past 150 years.

With our SuperTest test suite for C and C++, we can make these processes work for compilers as well, as we can demonstrate the connection between the language specification and the test suite.

We also demonstrate a run-time failure of the formally proven CompCert compiler, thus showing the need for testing alongside prove-techniques. As Knuth already wrote: "Beware of bugs in the above code; I have only proved it correct, not tried it."

3.3 Learning to accelerate compiler testing

Junjie Chen (Peking University, CN)

Main reference Junjie Chen, "Learning to Accelerate Compiler Testing", in Proc. of the 40th Int'l Conf. on Software Engineering (ICSE 2018), to appear.

It is well known that compilers are one of the most important software infrastructures. Compiler testing is an effective and widely-used way to assure the quality of compilers.

While many compiler testing techniques have been proposed to effectively detect compiler bugs, these techniques still suffer from the serious efficiency problem. This is because these compiler testing techniques need to run a large number of randomly generated test programs on the fly through automated test-generation tools (e.g., Csmith).

To accelerate compiler testing, it is desirable to schedule the execution order of the generated test programs so that the test programs that are more likely to trigger compiler bugs are executed earlier. Since different test programs tend to trigger the same compiler bug, the ideal goal of accelerating compiler testing is to execute the test programs triggering different compiler bugs in the beginning. However, such perfect goal is hard to achieve, and thus in this work, we design two steps to approach the ideal goal through learning, in order to largely accelerate compiler testing.

References

1 Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. Learning to Prioritize Test Programs for Compiler Testing. In: Proceedings of the 39th International Conference on Software Engineering (ICSE 2017), pages 700–711

2 Junjie Chen, Wenxiang Hu, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, Bing Xie. An Empirical Comparison of Compiler Testing Techniques. In: Proceedings of the 38th International Conference on Software Engineering (ICSE 2016), pages 180–190

3 Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, Bing Xie. Test Case Prioritization for Compilers: A Text-Vector based Approach. In: Proceedings of the 9th International Conference on Software Testing, Verification and Validation (ICST 2016), pages 266–277

3.4 A Generator of Highly Effective Fuzz Testers

Eric Eide (University of Utah - Salt Lake City, US)

A fuzz tester, or "fuzzer," is effective if it can continually create test cases that reveal defects throughout the system under test. It is difficult to create effective fuzzers for programming language compilers and interpreters because these systems have highly structured inputs. Our goal is to reduce the time and human effort needed to implement effective fuzzers for programming language implementations, and to this end, we are creating Xsmith, a new generator of fuzz testers. Xsmith will generate language fuzzers from specifications, and more importantly, it will inject sophisticated program-generation techniques into the fuzzers it creates. In this talk I will summarize the current status of the Xsmith project. Ultimately, Xsmith will be successful if it permits highly effective fuzz testers to be constructed with significantly less ad hoc code, and thus significantly less effort, than if they had been constructed from scratch.

3.5 Automated Testing of Graphics Shader Compilers

Hugues Evrard (Imperial College London, GB)

We present an automated technique for finding defects in compilers for graphics shading languages. A key challenge in compiler testing is the lack of an oracle that classifies an output as correct or incorrect; this is particularly pertinent in graphics shader compilers where the output is a rendered image that is typically under-specified. Our method builds on recent successful techniques for compiler validation based on metamorphic testing, and leverages existing high-value graphics shaders to create sets of transformed shaders that should be semantically equivalent. Rendering mismatches are then indicative of shader compilation bugs. Deviant shaders are automatically minimized to identify, in each case, a minimal change to an original high-value shader that induces a shader compiler bug. We have implemented the approach as a tool, GLFuzz, targeting the OpenGL shading language, GLSL. Our experiments over a set of 17 GPU and driver configurations, spanning the main 7 GPU designers, have led to us finding and reporting more than 60 distinct bugs, covering all tested configurations. As well as defective rendering, these issues identify security-critical vulnerabilities that affect WebGL, including a significant remote information leak security bug where a malicious web page can capture the contents of other browser tabs, and a bug whereby visiting a malicious web page can lead to a "blue screen of death" under Windows 10. Our findings show that shader compiler defects are prevalent, and that metamorphic testing provides an effective means for detecting them automatically.

3.6 Introduction to Coccinelle

Julia Lawall (INRIA - Paris, FR)

Main reference Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, Gilles Muller: "Documenting and automating collateral evolutions in linux device drivers", in Proc. of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008, pp. 247–260, ACM, 2008.
URL http://dx.doi.org/10.1145/1352592.1352618

Coccinelle is a program matching and transformation tool for C code. The guiding principle behind Coccinelle is the use of a patch-like notation, ie fragments of source code annotation with - to indicate code removal and + to indicate code addition, in order to express transformation rules. Coccinelle was originally developed to automate evolutions in Linux kernel code, and today has been used in over 6000 commits to the Linux kernel, by both the Coccinelle research group and a wide range of Linux kernel developers. Coccinelle is also used on other software, such as wine, qemu, and systemd. This talk gives an overview of Coccinelle and its potential applicability to the maintenance of compiler code.

3.7 Differential Testing of Interactive Debuggers

Daniel Lehmann (TU Darmstadt, DE)

To understand, localize, and fix programming errors, developers often rely on interactive debuggers. However, as debuggers are software, they may themselves have bugs, which can make debugging unnecessarily hard or even cause developers to reason about bugs that do not actually exist in their code. The problem of analyzing debuggers is fundamentally different from the well-studied problem of testing compilers because debuggers are interactive and because they lack a specification of expected behavior.

In this talk, we present an automated analysis technique for interactive debuggers. Our approach, called DBDB, generates debugger actions to exercise the debugger and records traces that summarize the debugger's behavior. By comparing traces of multiple debuggers with each other, we find diverging behavior that points to bugs and other noteworthy differences.

We evaluate DBDB on the JavaScript debuggers of Firefox and Chromium, finding 19 previously unreported bugs, six of which are already confirmed and fixed. Beyond finding bugs, our work is a first step toward agreeing on and specifying the expected behavior of interactive debuggers.

3.8 CUDA Compiler Verification

Thibaut Lutz (NVIDIA - Redmond, US)

CUDA is a widely used programming language for general purpose computation on GPUs. The implementation of the CUDA compiler involves many intermediate stages and components which can each potentially introduce a bug. The programming model of CUDA, which involves parallelism and distributed memory, also makes it more difficult to implement efficient testing.

In this talk, we examine the challenges faced when testing this complex compilation pipeline. An overview of the testing strategies and tools is then presented. Finally, we discuss the lessons learnt from our testing and future directions to improve coverage and find more intricate bugs.

3.9 An Introduction to Software Verification with Whiley

David J. Pearce (Victoria University - Wellington, NZ)

Main reference David J. Pearce, Lindsay Groves: "Designing a verifying compiler: Lessons learned from developing Whiley", Sci. Comput. Program., Vol. 113, pp. 191–220, 2015.
URL http://dx.doi.org/10.1016/j.scico.2015.09.006

In this talk, I'll employ live coding to demonstrate the Whiley programming language and its accompanying "verifying compiler". The language is focused on ensuring programs meet their specifications. Whiley programs can be verified at compile time, and doing this prevents a range of common errors impossible (e.g. divide-by-zero, array out-of-bounds, etc). Sophisticated specifications can be written using a quantifiers and other apparatus from first-order logic. Programming in Whiley feels surprisingly natural and the goal is to make it comparable to interacting with a type checker. The language has been used for the last three years to teach a large undergraduate class about program specification, and we have benefited considerably from this experience.

3.10 TreeFuzz: Learning Probabilistic Models of Input Data for Fuzz Testing

Michael Pradel (TU Darmstadt, DE)

Joint work of JIbesh Patra, Michael Pradel

Fuzzing is a popular technique to create test inputs for software that processes structured data. It has been successfully applied in various domains, ranging from compilers and interpreters over program analyses to rendering engines, image manipulation tools, and word processors. Existing fuzz testers are either tailored to a specific data format, rely on whitebox analysis of the software under test, or infer a context-free grammar of the input format. This talk presents TreeFuzz, an approach to learn probabilistic models of input data from a corpus of examples and to fuzz-generate new data from the inferred models. The approach supports any data format that can be represented as a tree and learns models without any human intervention. To support a wide range of different properties of input data, TreeFuzz is designed as a framework with an extensible set of generative models. The learned models are more expressive than context-free grammars, producing test inputs that reach deep into the software under test. Our evaluation applies TreeFuzz to a programming language, JavaScript, and a markup language, HTML. The results show that the approach creates mostly correct data: 96.3% of the generated JavaScript programs are syntactically valid and there are only 2.06 validation errors per kilobyte of generated HTML. Furthermore, we show that the performance of both learning and generation scales linearly with respect to the size of the corpus, making it easily applicable to tens of thousands of examples. Finally, we show that the fuzz-generated data are useful for testing: Using TreeFuzz-generated JavaScript programs for differential testing of JavaScript engines exposes various inconsistencies among browsers, including browser bugs and unimplemented language features.

3.11 A Few Stories about Compiler Bugs

John Regehr (University of Utah - Salt Lake City, US)

Main reference Nuno P. Lopes, David Menendez, Santosh Nagarakatte, John Regehr: "Provably correct peephole optimizations with alive", in Proc. of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, pp. 22–32, ACM, 2015.
URL http://dx.doi.org/10.1145/2737924.2737965

Engineering a successful compiler is difficult because the major goals for a compiler – fast compile times, few compiler bugs, and high-quality generated code – are individually difficult and also conflict with each other. Formal methods can help solve all of these problems, but the formal methods must be developed in such a way that they are maintainable over a long time period and are usable and understandable by compiler developers. One promising approach is translation validation, because it runs to the side of a compiler, requiring few if any modifications to the tool.

3.12 Automatic non-Functional Testing of Configurable Generators

Gerson Sunyé (University of Nantes, FR)

Generative software development has paved the way for the creation of multiple code generators and compilers that serve as a basis for automatically generating code to a broad range of software and hardware platforms. With full automatic code generation, the user is able to easily and rapidly synthesize software artifacts for various software platforms. In addition, modern generators (i.e., C compilers) become highly configurable, offering numerous configuration options that the user can use to easily customize the generated code for the target hardware platform.

In this context, it is crucial to verify the correct behavior of code generators. Numerous approaches have been proposed to verify the functional outcome of generated code but few of them evaluate the non-functional properties of automatically generated code, namely the performance and resource usage properties.

In this presentation, I address two problems:

1. Non-functional testing of code generators: We benefit from the existence of multiple generators with comparable functionality (i.e., code generator families) to automatically test the generated code. We leverage the metamorphic testing to detect inconsistencies in code generators families by defining metamorphic relations as test oracles. We define the metamorphic relation as a comparison between the variations of performance and resource usage of code, generated from the same code generator family.
2. Handling the diversity of software and hardware platforms in software testing: Running tests and evaluating the resource usage in heterogeneous environments is tedious. To handle this problem, we benefit from the recent advances in lightweight system virtualization, in particular container-based virtualization, in order to offer effective support for automatically deploying, executing, and monitoring code in heterogeneous environment, and collect non-functional metrics (e.g., memory and CPU consumptions).

We evaluate our approach by analyzing the performance of Haxe, a popular code generator family. Experimental results show that our approach is able to automatically detect several inconsistencies that reveal real issues in this family of code generators.

3.13 Learning to Synthesize Programs

Yingfei Xiong (Peking University, CN)

In many scenarios including testing we need to find the most likely program under a local context,where the local context can be the program under test, an incomplete program, a partial specification, natural language description, etc. We call such problem program estimation. In this paper we propose an abstract framework, learning to synthesis, or L2S in short, to address this problem. L2S combines four tools to achieve this: syntax is used to define the search space and search steps, constraints are used to prune off invalid candidates at each search step, machine-learned models are used to estimate conditional probabilities for the candidates at each search step, and search algorithms are used to find the best possible solution. The main goal of L2S is to lay out the design space to motivate the research on program estimation.

We have performed a preliminary evaluation by instantiating this framework for synthesizing conditions. On 4 projects from Defects4J, we can successfully synthesize the correct conditions at top 10 in 64.7%-85.7% of the cases by training only on the source code of the project, and the precision is related to the size of the projects.

3.14 Debugging Debug Information

Francesco Zappa Nardelli (INRIA - Paris, FR)

Hidden, obscure, and badly specified components lurk at the very heart of our computing infrastructure. Consider debugging informations. Debugging informations are obviously relied upon by debuggers and play a key role in the implementation of program analysis tools, but, more surprisingly, debugging informations can be relied upon by the runtime of high-level programming languages (e.g. to unwind the stack and implement C++ exceptions). Unfortunately debugging informations themselves can be pervaded by subtle, undebuggable, bugs. We will describe how to perform validation and synthesis of the DWARF stack unwinding debug tables, and we will report on ambitious plans we might build on reliable debug information.

3.15 Tutorial: Compiler Optimisations and Shared Memory Concurrency

Francesco Zappa Nardelli (INRIA - Paris, FR)

Main reference Robin Morisset, Pankaj Pawan, Francesco Zappa Nardelli: "Compiler testing via a theory of sound optimisations in the C11/C++11 memory model", in Proc. of the 34th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2013), pp. 187–196, ACM, 2013.
URL https://doi.org/10.1145/2499370.2491967

Compiler optimisations introduce unexpected behaviours in shared memory concurrent programs; programming languages thus define memory models to specify precisely which behaviours can be observed and, indirectly, which optimisations a compiler can apply. Taking the C11/C++11 standard as running example, we will investigate how to reason about the correctness of compiler optimisations (or lack thereof), and how to build a tool to fuzzy test mainstream compilers against the memory model. Au passage, we will describe our failure to specify a reasonable memory model for general purpose programming languages, and how this failure paved the way for exciting research on validation and implementation of compilers.

3.16 Fuzzing with Inferred Grammars

Andreas Zeller (Universität des Saarlandes, DE)

Fuzzing a program is greatly simplified if one has a grammar that describes the lexical and syntactical (possibly even semantical) properties of the input. I show how grammar and code coverage can successfully guide test generation, and how tracking how a program processes input characters can produce grammars that again can be fed into test generation.

4 Working groups

4.1 Benchmarking Compiler Testing

Junjie Chen (Peking University, CN)

This group discussed the creation of *benchmarks* for compiler testing.

These benchmarks would take known buggy versions of a compiler and curate a set of bugs, along with test cases triggering them, a bug fix, and information about the bug (e.g. location of it, conditions required to trigger it).

The working group concluded that such benchmarks would be worthwhile, as they would significantly speed the development of new compiler testing techniques by evaluating their effectiveness in finding a known set of bugs.

4.2 Beyond Compiler Testing

Michael Pradel (TU Darmstadt, DE)

A working group chaired by Michael Pradel discussed how to adapt ideas from compiler testing beyond compilers.

The group went through a set of developer tools that also take source code as their input, such as lint-like bug finding tools, debuggers, code search, static analyses, and symbolic execution engines. For each tool, a set of challenges were identified

One recurring challenge is that some tools require further inputs in addition to program source code, e.g., user interactions for debuggers or search queries for code search engines.

Another recurring challenge is that the desired behavior of several tools is only informally specified. For example, testing whether a bug finding tool works as expected requires knowledge about which code the tool is supposed to (not) flag as buggy.

The working group concludes that

1. Testing other development tools than compilers is an exciting research direction.
2. Existing work on generating test programs could be reused
3. There remain additional challenges to be addressed in future work.

4.3 Test-Case Reduction

David R. MacIver (Imperial College London, GB)

This working group discussed common issues and themes of test-case reduction, and open questions for future work in the area.

The two issues common to test case reduction identified were:

- It is very difficult to write good predicates for test-case reduction—the most common issue users of C-Reduce face is when they have written a predicate for their test case that is “too broad” and the bug slips off the originally identified bug into a less interesting one.
- It is also very difficult to write test-case reducers.

The group proposed that the solution to the latter was to make it easier to extend existing test-case reducers (C-Reduce and StructureShrink were both mentioned) to add new “reduction passes”.

It is unclear that there is a well defined notion of what test-case reduction *does*. Existing literature has focused a great deal on size, but often the smallest example is not the clearest, and often even among the smallest size examples more reduction can be performed to improve readability. For example, C-Reduce always inlines functions even if this makes the test-case larger. Work such as [2, 3] on combining test-case reduction with generalisation was highlighted as a potential solution to this problem.

We also observed that there are common idioms in test-case reduction that are currently under-documented–e.g. when to be greedy vs non-greedy, trying large reductions first and then backing off, combining multiple shrinks together when single ones fail.

Open questions we considered interesting for further research were:

- How best can we present “readable” examples to end users, and what role does pure test-case reduction have in that?
- In light of that how can we best determine when one test-case is “better” than another?
- How can we cope with situations where shrinks must be chained together given that it is potentially very expensive to identify such chains?

References

1 John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. “ Test-Case Reduction for C Compiler Bugs”. In Proceedings of 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012), Beijing, China, June 2012.

2 Groce, Alex, Josie Holmes, and Kevin Kellar. “One test to rule them all.” Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, 2017.

3 Braquehais, Rudy, and Colin Runciman. “Extrapolate: generalizing counter-examples of functional test properties.” (2017).

4.4 Program Generation

David R. MacIver (Imperial College London, GB)

This was a working group about the problem of generating programs to test compilers.

The group identified two key properties which are in tension with each other. These were summarised as “Semantic Validity” and “Completeness” (or “Coverage”).

Semantic validity is the property of making sure you only generate “good” inputs–for example, ensuring that a C program is free of undefined behaviour.

Completeness on the other hand is the property of reaching into the dark corners of the compiler to find interesting behaviour rather than just shallowly testing the surface–finding interesting programs that trigger unusal behaviours.

These are in tension because a generator of valid programs must necessarily be somewhat conservative in what it emits. However, they can also mutually support each other, as program generator that generates too many invalid programs will tend to to succeed only in generating trivial valid programs.

It was felt that most interesting open questions were about how to that improve completeness while preserving semantic validity.

The two main approaches suggested for pursuing this were to provide better tooling for describing language to attempt to use machine learning to discover the range of valid programs from the behaviour of the compiler.

The group also identified the problem of generating *idiomatic* programs–a common problem is that generated test cases are ignored by compiler developers because they look unnatural, and the group believed that generating test cases that look like normal code would help with this.

5 Panel discussions

5.1 Challenges and Concepts of Compiler Testing and Verification

Yingfei Xiong (Peking University, CN)

In the discussion session three topics were discussed.

First, all participants brainstormed the current open challenges in compiler testing and verification.

Second, the name of the field was discussed. The participants agreed that there is no universal good name to the field. For example, “compiler” captures most of the work in this field and is easy to explain to outsiders, but has problem of omitting some subjects such as debuggers. As a result, we do not try to name the field for now, and keep the name “compiler testing and verification” for future editions of the seminar. Here “compiler” is broadly defined to include software tools that take programs as input.

Third, some confusing terms were discussed. For example, program could refer to the input program to the compiler and the compiler itself. Tests could refer to the tests of the compiler or the tests in the compiled program. However, no good solution came out from the discussion and each paper should clearly define the terms before use.

The panel identified the following challenges:

1. Input program generation
   a. For advanced type systems
   b. Testing subtyping and other complex relations
   c. Generating valid programs
   d. Generating invalid yet effective test programs
   e. Covering new language features
   f. Dealing with undefined behavior
2. Finding effective test oracles
3. Efficient testing
4. Undefined semantics
5. Reproducibility of the test cases
6. Dealing with concurrency and nondeterminism
7. Readability of test cases
8. Generalization of test cases
9. Finding *important* bugs.
   a. What is importance?
   b. How can we communicate it?
10. Verifying compiler optimizations
11. Using translation validation for better testing.
12. Providing efficient and easy-to-use tool implementations

Participants

- Edward E. Aftandilian
Google Research –
Mountain View, US
- Marcel Beemster
Solid Sands – Amsterdam, NL
- Junjie Chen
Peking University, CN
- Nathan Chong
ARM Ltd. – Cambridge, GB
- Eric Eide
University of Utah –
Salt Lake City, US
- Hugues Evrard
Imperial College London, GB
- Dan Hao
Peking University, CN
- John Hughes
Chalmers University of
Technology – Göteborg, SE
- Dan Iorga
Imperial College London, GB
- Julia Lawall
INRIA – Paris, FR
- Daniel Lehmann
TU Darmstadt, DE
- Thibaut Lutz
NVIDIA – Redmond, US
- David MacIver
Imperial College London, GB
- Jessica Paquette
Apple Computer Inc. –
Cupertino, US
- David J. Pearce
Victoria University –
Wellington, NZ
- Michael Pradel
TU Darmstadt, DE
- John Regehr
University of Utah –
Salt Lake City, US
- Raimondas Sasnauskas
SES Engineering –
Luxembourg, LU
- Marija Selakovic
TU Darmstadt, DE
- Gerson Sunyé
University of Nantes, FR
- Nikolai Tillmann
Facebook – Seattle, US
- Yingfei Xiong
Peking University, CN
- Francesco Zappa Nardelli
INRIA – Paris, FR
- Andreas Zeller
Universität des Saarlandes, DE
- Hongyu Zhang
University of Newcastle, AU