

# Symbolic Execution and Constraint Solving

Edited by

Cristian Cadar<sup>1</sup>, Vijay Ganesh<sup>2</sup>, Raimondas Sasnauskas<sup>3</sup>, and  
Koushik Sen<sup>4</sup>

- 1 Imperial College London, GB, [c.cadar@imperial.ac.uk](mailto:c.cadar@imperial.ac.uk)
- 2 University of Waterloo, CA, [vganesh@uwaterloo.ca](mailto:vganesh@uwaterloo.ca)
- 3 University of Utah, US, [rsas@cs.utah.edu](mailto:rsas@cs.utah.edu)
- 4 University of California, Berkeley, US, [ksen@cs.berkeley.edu](mailto:ksen@cs.berkeley.edu)

---

## Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 14442 “Symbolic Execution and Constraint Solving”, whose main goals were to bring together leading researchers in the fields of symbolic execution and constraint solving, foster greater communication between these two communities and exchange ideas about new research directions in these fields.

There has been a veritable revolution over the last decade in the symbiotic fields of constraint solving and symbolic execution. Even though key ideas behind symbolic execution were introduced more than three decades ago, it was only recently that these techniques became practical as a result of significant advances in constraint satisfiability and scalable combinations of concrete and symbolic execution. Thanks to these advances, testing and analysis techniques based on symbolic execution are having a major impact on many sub-fields of software engineering, computer systems, security, and others. New applications such as program and document repair are being enabled, while older applications such as model checking are being super-charged. Additionally, significant and fast-paced advances are being made in research at the intersection of traditional program analysis, symbolic execution and constraint solving. Therefore, this seminar brought together researchers in these varied fields in order to further facilitate collaborations that take advantage of this unique and fruitful confluence of ideas from the fields of symbolic execution and constraint solving.

**Seminar** October 27–30, 2014 – <http://www.dagstuhl.de/14442>

**1998 ACM Subject Classification** D.2.4 Software/Program Verification (Formal methods), D.2.5 Testing and Debugging (Symbolic execution, Testing tools (e.g., data generators, coverage testing), Tracing), B.2.3 Reliability, Testing, and Fault-Tolerance (Test generation)

**Keywords and phrases** Symbolic Execution, Software Testing, Automated Program Analysis, Constraint Solvers

**Digital Object Identifier** 10.4230/DagRep.4.10.98


## 1 Executive Summary

*Cristian Cadar*

*Vijay Ganesh*

*Raimondas Sasnauskas*

*Koushik Sen*

**License**  Creative Commons BY 3.0 Unported license  
© Cristian Cadar, Vijay Ganesh, Raimondas Sasnauskas, and Koushik Sen

Symbolic execution has garnered a lot of attention in recent years as an effective technique for generating high-coverage test suites, finding deep errors in complex software applications,



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 3.0 Unported license

Symbolic Execution and Constraint Solving, *Dagstuhl Reports*, Vol. 4, Issue 10, pp. 98–114

Editors: Cristian Cadar, Vijay Ganesh, Raimondas Sasnauskas, and Koushik Sen



DAGSTUHL  
REPORTS

Dagstuhl Reports  
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and more generally as one of the few techniques that is useful across the board in myriad software engineering applications. While key ideas behind symbolic execution were introduced more than three decades ago, it was only recently that these techniques became practical as a result of significant advances in constraint satisfiability and scalable combinations of concrete and symbolic execution. The result has been an explosion in applications of symbolic execution techniques in software engineering, security, formal methods and systems research. Furthermore, researchers are combining symbolic execution with traditional program analysis techniques in novel ways to address longstanding software engineering problems. This in turn has led to rapid developments in both constraint solvers and symbolic execution techniques, necessitating an in-depth exchange of ideas between researchers working on solvers and symbolic techniques, best accomplished through dedicated workshops.

Hence, one of the main goals of this Dagstuhl seminar was to bring together leading researchers in the fields of symbolic execution and constraint solving, foster greater communication between these two communities and discuss new research directions in these fields. The seminar had 34 participants from Canada, France, Germany, Norway, Singapore, South Africa, Spain, Switzerland, The Netherlands, United Kingdom and United States, from both academia, research laboratories, and the industry. More importantly, the participants represented several different communities, with the topics of the talks and discussions reflecting these diverse interests: testing, verification, security, floating point constraint solving, hybrid string-numeric constraints, debugging and repair, education, and commercialization, among many others.

## 2 Table of Contents

### Executive Summary

*Cristian Cadar, Vijay Ganesh, Raimondas Sasnauskas, and Koushik Sen* . . . . . 98

### Overview of Talks

Automated White-Box Testing Beyond Branch Coverage

*Sébastien Bardin* . . . . . 102

Dynamic Symbolic Execution: State of the Art, Applications and Challenges

*Cristian Cadar* . . . . . 102

Reaching Verification with Systematic Testing

*Maria Christakis* . . . . . 103

Superoptimizing LLVM

*Peter Collingbourne* . . . . . 103

Experiences with SMT in the GPUVerify Project

*Alastair F. Donaldson* . . . . . 104

On the Challenges of Combining Search-Based Software Testing and Symbolic Execution

*Juan Pablo Galeotti* . . . . . 104

Impact of Community Structure on SAT Solver Performance

*Vijay Ganesh* . . . . . 105

Combining FSM Modeling and Bit-Vector Theories to Solve Hybrid String-Numeric Constraints

*Indradeep Ghosh* . . . . . 105

Symbolic Path-Oriented Test Data Generation for Floating-Point Programs

*Arnaud Gotlieb* . . . . . 106

Segmented Symbolic Analysis

*Wei Le* . . . . . 106

Commercial Symbolic Execution

*Paul Marinescu* . . . . . 107

Solving Non-linear Integer Constraints Arising from Program Analysis

*Albert Oliveras* . . . . . 107

Constraint Solving in Symbolic Execution

*Hristina Palikareva* . . . . . 108

Provably Correct Peephole Optimizations with Alive

*John Regehr* . . . . . 108

Symbolic Techniques for Program Debugging

*Abhik Roychoudhury* . . . . . 109

Generating Heap Summaries from Symbolic Execution

*Neha Rungta* . . . . . 109

MultiSE: Multi-Path Symbolic Execution using Value Summaries

*Koushik Sen* . . . . . 110

The Symbiosis of Network Testing and Symbolic Execution  
*Oscar Soria Dustmann* . . . . . 110

Symbolic Execution and Model Counting  
*Willem Visser* . . . . . 111

Feedback-Driven Dynamic Invariant Discovery  
*Lingming Zhang* . . . . . 111

**Thought-provoking Talks** . . . . . 112

**Programme** . . . . . 112

**Participants** . . . . . 114

### 3 Overview of Talks

#### 3.1 Automated White-Box Testing Beyond Branch Coverage

*Sébastien Bardin (CEA LIST, FR)*

**License** © Creative Commons BY 3.0 Unported license  
© Sébastien Bardin

**Joint work of** Bardin, Sébastien; Kosmatov, Nikolai; Delahaye, Mickaël; Chebaro, Omar  
**Main reference** S. Bardin, N. Kosmatov, F. Cheynier, “Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria,” in Proc. of the 2014 IEEE 7th Int’l Conf. on Software Testing, Verification and Validation (ICST’14), pp. 173–182, IEEE, 2014.  
**URL** <http://dx.doi.org/10.1109/ICST.2014.30>

Automated white-box testing is a major issue in software engineering. The last decade has seen tremendous progress in Automatic test data generation thanks to Symbolic Execution techniques. Yet, these promising results are currently limited to very basic coverage criteria such as statement coverage or branch coverage, while many more criteria can be found in the literature. Moreover, other important issues in structural testing, such as infeasible test requirement detection, have not made so much progress.

In order to tackle these problems, we rely on a simple and concise specification mechanism for (structural) coverage criteria, called labels. Labels are appealing since they can faithfully encode many existing coverage criteria, allowing to handle all of them in a unified way. They are the corner stone of FRAMA-C/LTEST, a generic and integrated toolkit for automated white-box testing of C programs, providing automatic test generation and detection of infeasible test requirements through a combination of static and dynamic analyzes.

We will overview the platform and focus our presentation on the following points: (1) present a new and efficient Symbolic Execution algorithm dedicated to label coverage, and (2) describe recent results on the sound detection of infeasible test requirements.

##### References

- 1 Sébastien Bardin, Nikolai Kosmatov, François Cheynier: Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria. In: ICST 2014. IEEE (Los Alamitos)
- 2 Sébastien Bardin, Omar Chebaro, Mickaël Delahaye, Nikolai Kosmatov: An All-in-One Toolkit for Automated White-Box Testing. In: TAP 2014. Springer (Heidelberg)

#### 3.2 Dynamic Symbolic Execution: State of the Art, Applications and Challenges

*Cristian Cadar (Imperial College London, GB)*

**License** © Creative Commons BY 3.0 Unported license  
© Cristian Cadar

In this talk, I start by presenting the state of the art in dynamic symbolic execution, including its main enablers, namely mixed concrete/symbolic execution, better and faster constraint solvers and novel path exploration algorithms. I then survey several applications of symbolic execution, including bug finding, security, high-coverage test generation, software debugging, patch and document recovery, etc. I finish the talk by discussing some of the ongoing challenges in terms of path explosion, verification, concurrency and constraint solving.

### 3.3 Reaching Verification with Systematic Testing

*Maria Christakis (ETH Zürich, CH)*

**License** © Creative Commons BY 3.0 Unported license  
© Maria Christakis

**Joint work of** Christakis, Maria; Godefroid, Patrice

**Main reference** M. Christakis, P. Godefroid, “Proving memory safety of the ANI Windows image parser using compositional exhaustive testing,” in Proc. of the 16th Int’l Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI’15), LNCS, Vol. 8931, pp. 370–389, Springer, 2015.

**URL** [http://dx.doi.org/10.1007/978-3-662-46081-8\\_21](http://dx.doi.org/10.1007/978-3-662-46081-8_21)

We describe how we proved memory safety of a complex Windows image parser written in low-level C in only three months of work and using only three core techniques, namely (1) symbolic execution at the x86 binary level, (2) exhaustive program path enumeration and testing, and (3) user-guided program decomposition and summarization. As a result of this work, we are able to prove, for the first time, that a Windows image parser is memory safe, that is, free of any buffer-overflow security vulnerabilities, modulo the soundness of our tools and several additional assumptions regarding bounding input-dependent loops, fixing a few buffer-overflow bugs, and excluding some code parts that are not memory safe by design.

### 3.4 Superoptimizing LLVM

*Peter Collingbourne (Google Inc. – Mountain View, US)*

**License** © Creative Commons BY 3.0 Unported license  
© Peter Collingbourne

**Joint work of** Collingbourne, Peter; Regehr, John; Sasnauskas, Raimondas; Taneja, Jubi; Chen, Yang; Ketema, Jeroen

Compiler development, as with any engineering task, involves certain tradeoffs. In particular, the tradeoff between the expected performance improvement of a specific optimization and the engineer time required to implement it can be difficult to assess, especially given the long tail of potential uncaught optimizations that may be lurking within a large codebase.

Souper is an open source superoptimizer based on LLVM which can automatically extract potential peephole optimizations from real C and C++ programs, use a SMT solver to verify their correctness, apply them automatically, and use static and dynamic profiling to identify the most profitable optimizations. It pushes the state of the art in optimizer development forward in two primary ways: directly, by instantly providing a way for users to automatically apply a variety of previously unimplemented peephole optimizations to their code; and indirectly, by allowing compiler developers to focus their energy on implementing the most profitable optimizations in the baseline compiler.

This talk described Souper, giving details of the translation from LLVM to SMT predicates, and results from compiling LLVM with Souper.

### 3.5 Experiences with SMT in the GPUVerify Project

*Alastair F. Donaldson (Imperial College London, GB)*

**License** © Creative Commons BY 3.0 Unported license  
© Alastair F. Donaldson

**Joint work of** Bardsley, Ethel; Betts, Adam; Chong, Nathan; Collingbourne, Peter; Deligiannis, Pantazis; Donaldson, Alastair F.; Ketema, Jeroen; Liew, Daniel; Qadeer, Shaz

**Main reference** E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, S. Qadeer, “Engineering a Static Verification Tool for GPU Kernels,” in Proc. of the 26th Int’l Conf. on Computer Aided Verification (CAV’14), LNCS, Vol. 8559, pp. 226–242, Springer, 2014.

**URL** [http://dx.doi.org/10.1007/978-3-319-08867-9\\_15](http://dx.doi.org/10.1007/978-3-319-08867-9_15)

The GPUVerify project has investigated techniques for automatically proving freedom from data races in GPU kernels, implemented in a practical tool. To achieve efficiency and a relatively high degree of automation, we have put a lot of effort into designing encodings of properties of GPU kernels into SMT formulas that avoid the use of quantifiers, which are notoriously hard to reason about automatically. In our experiments with various encodings, using the Z3 and CVC4 solvers, we have observed a great deal of variation in response time between solvers, and across encodings with respect to a single solver. In this talk I gave an overview of one SMT encoding of the data race-freedom property for GPU kernels, and presented experimental results illustrating the variation in results we have observed. The results demonstrate the need to evaluate optimizations to a verification technique with respect to large set of benchmarks, and to evaluate SMT-based optimizations using multiple solvers.

### 3.6 On the Challenges of Combining Search-Based Software Testing and Symbolic Execution

*Juan Pablo Galeotti (Universität des Saarlandes, DE)*

**License** © Creative Commons BY 3.0 Unported license  
© Juan Pablo Galeotti

**Joint work of** Galeotti, Juan Pablo; Gordon Fraser; Andrea Arcuri; Matthias Hörschele; Andreas Zeller

**Main reference** J.P. Galeotti, G. Fraser, A. Arcuri, “Improving search-based test suite generation with dynamic symbolic execution,” in Proc. of the 2013 IEEE 24th Int’l Symp. on Software Reliability Engineering (ISSRE’13), pp. 360–360, IEEE, 2013.

**URL** <http://dx.doi.org/10.1109/ISSRE.2013.6698889>

In recent years, there has been a tremendous advance in Search-Based Software Testing and Symbolic Execution. Search-based testing (SBST) can automatically generate unit test suites for object oriented code, but may struggle to generate specific values necessary to cover difficult parts of the code. Symbolic execution (SE) efficiently generates such specific values, but may struggle with complex datatypes, in particular those that require sequences of calls for construction.

In this talk I will present a hybrid approach for automatic unit-test generation that integrates the best of both worlds [1]. Also, I will focus on the challenges we recently faced when applying SBST and SE to multi-layered software comprised of an object-oriented upper layer and a native-platform dependent native layer [2].

#### References

- 1 Juan Pablo Galeotti, Gordon Fraser, Andrea Arcuri: *Improving search-based test suite generation with dynamic symbolic execution*. In ISSRE 2013, pages 360–369
- 2 Matthias Hörschele, Juan Pablo Galeotti, Andreas Zeller: *Test generation across multiple layers*. In SBST 2014, pages 1–4

### 3.7 Impact of Community Structure on SAT Solver Performance

*Vijay Ganesh (University of Waterloo, CA)*

**License** © Creative Commons BY 3.0 Unported license  
© Vijay Ganesh

**Joint work of** Newsham, Zack; Ganesh, Vijay; Fischmeister, Sebastian; Audemard, Gilles; Simon, Laurent  
**Main reference** Zack Newsham, V. Ganesh, S. Fischmeister, G. Audemard, L. Simon, “Impact of Community Structure on SAT Solver Performance,” in Proc. of the 17th Int’l Conf. on Theory and Applications of Satisfiability Testing (SAT’14), LNCS, Vol. 8561, pp. 252–268, Springer, 2014.  
**URL** [http://dx.doi.org/10.1007/978-3-319-09284-3\\_20](http://dx.doi.org/10.1007/978-3-319-09284-3_20)

Modern CDCL SAT solvers routinely solve very large industrial SAT instances in relatively short periods of time. It is clear that these solvers somehow exploit the structure of real-world instances. However, to-date there have been few results that precisely characterise this structure. In this paper, we provide evidence that the community structure of real-world SAT instances is correlated with the running time of CDCL SAT solvers. It has been known for some time that real-world SAT instances, viewed as graphs, have natural communities in them. A community is a sub-graph of the graph of a SAT instance, such that this sub-graph has more internal edges than outgoing to the rest of the graph. The community structure of a graph is often characterised by a quality metric called  $Q$ . Intuitively, a graph with high-quality community structure (high  $Q$ ) is easily separable into smaller communities, while the one with low  $Q$  is not. We provide three results based on empirical data which show that community structure of real-world industrial instances is a better predictor of the running time of CDCL solvers than other commonly considered factors such as variables and clauses. First, we show that there is a strong correlation between the  $Q$  value and Literal Block Distance metric of quality of conflict clauses used in clause-deletion policies in Glucose-like solvers. Second, using regression analysis, we show that the the number of communities and the  $Q$  value of the graph of real-world SAT instances is more predictive of the running time of CDCL solvers than traditional metrics like number of variables or clauses. Finally, we show that randomly-generated SAT instances with  $0.05 \leq Q \leq 0.13$  are dramatically harder to solve for CDCL solvers than otherwise.

### 3.8 Combining FSM Modeling and Bit-Vector Theories to Solve Hybrid String-Numeric Constraints

*Indradeep Ghosh (Fujitsu Labs of America Inc. – Sunnyvale, US)*

**License** © Creative Commons BY 3.0 Unported license  
© Indradeep Ghosh

**Joint work of** Ghosh, Indradeep; Li, Goudong  
**Main reference** G.Li, I. Ghosh, “PASS: String Solving with Parameterized Array and Interval Automaton,” in Proc. of the 9th Int’l Haifa Verification Conference (HVC’13), LNCS, Vol. 8244, pp. 15–31, Springer, 2013.  
**URL** [http://dx.doi.org/10.1007/978-3-319-03077-7\\_2](http://dx.doi.org/10.1007/978-3-319-03077-7_2)

This talk focused on the challenges and techniques for solving hybrid String-Numeric constraints. These types of constraints arise as path conditions during symbolic execution of industrial application, especially enterprise applications written in Java and JavaScript programming languages. Two types of solving techniques were presented: Finite state machine modeling of strings and Bit-vector models of strings. Their pros and cons were discussed and ways to merge them using parameterized arrays and interval automaton were presented.



### 3.9 Symbolic Path-Oriented Test Data Generation for Floating-Point Programs

*Arnaud Gotlieb (Simula Research Laboratory – Lysaker, NO)*

**License** © Creative Commons BY 3.0 Unported license  
© Arnaud Gotlieb

**Joint work of** Bagnara, Roberto; Carlier, Matthieu; Gori, Roberta; Gotlieb, Arnaud

**Main reference** R. Bagnara, M. Carlier, R. Gori, A. Gotlieb, “Symbolic path-oriented test data generation for floating-point programs,” in Proc. of the 2013 IEEE 6th Int’l Conf. on Software Testing, Verification and Validation (ICST’13), pp. 1–10, IEEE, 2013.

**URL** <http://dx.doi.org/10.1109/ICST.2013.17>

Verifying critical numerical software involves the generation of test data for floating-point intensive programs. As the symbolic execution of floating-point computations presents significant difficulties, existing approaches usually resort to random or search-based test data generation. However, without symbolic reasoning, it is almost impossible to generate test inputs that execute many paths with floating-point computations. Moreover, constraint solvers over the reals or the rationals do not handle the rounding errors. In this paper, we present a new version of FPSE, a symbolic evaluator for C program paths, that specifically addresses this problem. The tool solves path conditions containing floating-point computations by using correct and precise projection functions. This version of the tool exploits an essential filtering property based on the representation of floating-point numbers that makes it suitable to generate path-oriented test inputs for complex paths characterized by floating-point intensive computations. The paper reviews the key implementation choices in FPSE and the labeling search heuristics we selected to maximize the benefits of enhanced filtering. Our experimental results show that FPSE can generate correct test inputs for selected paths containing several hundreds of iterations and thousands of executable floating-point statements on a standard machine: this is currently outside the scope of any other symbolic execution test data generator tool.

#### References

- 1 R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. Symbolic path-oriented test data generation for floating-point programs. In Proc. of the 6th IEEE Int. Conf. on Software Testing, Verification and Validation (ICST’13), Luxembourg, Mar. 2013.
- 2 B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability (STVR)*, 16(2):97–121, June 2006

### 3.10 Segmented Symbolic Analysis

*Wei Le (Iowa State University – Ames, US)*

**License** © Creative Commons BY 3.0 Unported license  
© Wei Le

**Main reference** W. Le, “Segmented Symbolic Analysis,” in Proc. of the 2013 35th Int’l Conf. on Software Engineering (ICSE ’13), pp. 212–221, IEEE, 2013.


**URL** <http://dx.doi.org/10.1109/ICSE.2013.6606567>

Symbolic analysis is indispensable for software tools that require program semantic information at compile time. However, determining symbolic values for program variables related to loops and library calls is challenging, as the computation and data related to loops can have statically unknown bounds, and the library sources are typically not available at compile time. In this talk, I present segmented symbolic analysis, a hybrid technique that enables fully automatic symbolic analysis even for the traditionally challenging code of library calls

and loops. The novelties of this work are threefold: 1) we flexibly weave symbolic and concrete executions on the selected parts of the program based on demand; 2) dynamic executions are performed on the unit tests constructed from the code segments to infer program semantics needed by static analysis; and 3) the dynamic information from multiple runs is aggregated via regression analysis. We developed the Helium framework, consisting of a static component that performs symbolic analysis and partitions a program, a dynamic analysis that synthesizes unit tests and automatically infers symbolic values for program variables, and a protocol that enables static and dynamic analyses to be run interactively and concurrently. Our experimental results show that by handling loops and library calls that a traditional symbolic analysis cannot process, segmented symbolic analysis detects 5 times more buffer overflows. The technique is scalable for real-world programs such as `putty`, `tightvnc` and `snort`.

### 3.11 Commercial Symbolic Execution

*Paul Marinescu (Imperial College London, GB)*

**License**  Creative Commons BY 3.0 Unported license  
© Paul Marinescu

Symbolic execution is a powerful automatic testing technique which recently received a lot of attention in academic circles, but has not found its way into mainstream software engineering because reaching a balance between usability, effectiveness and resource requirements has proved elusive thus far. This talk looks at the challenges that need to be overcome to make symbolic execution a commercial success, comparing it with static analysis-based commercial solutions.

### 3.12 Solving Non-linear Integer Constraints Arising from Program Analysis

*Albert Oliveras (Polytechnic University of Catalonia, ES)*

**License**  Creative Commons BY 3.0 Unported license  
© Albert Oliveras

**Joint work of** Larraz, Daniel; Oliveras, Albert; Rodriguez-Carbonell, Enric; Rubio, Albert

**Main reference** D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, A. Rubio, “Minimal-Model-Guided Approaches to Solving Polynomial Constraints and Extensions,” in Proc. of the 17th Int’l Conf. on Theory and Applications of Satisfiability Testing (SAT’14), LNCS, Vol. 8561, pp. 333-350, Springer, 2014.

**URL** [http://dx.doi.org/10.1007/978-3-319-09284-3\\_25](http://dx.doi.org/10.1007/978-3-319-09284-3_25)

We present new methods for deciding the satisfiability of formulas involving integer polynomial constraints. In previous work we proposed to solve SMT(NIA) problems by reducing them to SMT(LIA): non-linear monomials are linearized by abstracting them with fresh variables and by performing case splitting on integer variables with finite domain. When variables do not have finite domains, artificial ones can be introduced by imposing a lower and an upper bound, and made iteratively larger until a solution is found (or the procedure times out). For the approach to be practical, unsatisfiable cores are used to guide which domains have to be relaxed (i.e., enlarged) from one iteration to the following one. However, it is not clear then how large they have to be made, which is critical.

Here we propose to guide the domain relaxation step by analyzing minimal models produced by the SMT(LIA) solver. Namely, we consider two different cost functions: the number of violated artificial domain bounds, and the distance with respect to the artificial domains. We compare these approaches with other techniques on benchmarks coming from constraint-based program analysis and show the potential of the method. Finally, we describe how one of these minimal-model-guided techniques can be smoothly adapted to deal with the extension Max-SMT of SMT(NIA) and then applied to program termination proving.

### 3.13 Constraint Solving in Symbolic Execution

*Hristina Palikareva (Imperial College London, GB)*

**License** © Creative Commons BY 3.0 Unported license  
© Hristina Palikareva

**Joint work of** Palikareva, Hristina; Cadar, Cristian

**Main reference** H. Palikareva, C. Cadar, “Multi-solver Support in Symbolic Execution,” in Proc. of the 25th Int’l Conf. on Computer Aided Verification (CAV’13), LNCS, Vol. 8044, pp. 53–68, Springer, 2013; pre-print available from author’s webpage.

**URL** [http://dx.doi.org/10.1007/978-3-642-39799-8\\_3](http://dx.doi.org/10.1007/978-3-642-39799-8_3)

**URL** <http://srg.doc.ic.ac.uk/files/papers/kee-multisolver-cav-13.pdf>

Dynamic symbolic execution is an automated program analysis technique that employs an SMT solver to systematically explore paths through a program. It has been acknowledged in recent years as a highly effective technique for generating high-coverage test suites as well as for uncovering deep corner-case bugs in complex real-world software, and one of the key factors responsible for that success are the tremendous advances in SMT-solving technology. Nevertheless, constraint solving remains one of the fundamental challenges of symbolic execution, and for many programs it is the main performance bottleneck.

In this talk, we will present the results reported in our CAV 2013 paper on integrating support for multiple SMT solvers in the dynamic symbolic execution engine KLEE. In particular, we will outline the key characteristics of the SMT queries generated during symbolic execution, describe several high-level domain-specific optimisations that KLEE employs to exploit those characteristics, introduce an extension of KLEE that uses a number of state-of-the-art SMT solvers (Boolector, STP and Z3) through the metaSMT solver framework, and compare the solvers’ performance when run on large sets of QF\_ABV queries obtained during the symbolic execution of real-world software. In addition, we will discuss several options for designing a parallel portfolio solver for symbolic execution tools. We will conclude the talk by proposing the introduction of a separate division at the annual SMT competition targeted specifically at symbolic execution tools.

### 3.14 Provably Correct Peephole Optimizations with Alive

*John Regehr (University of Utah, US)*

**License** © Creative Commons BY 3.0 Unported license  
© John Regehr

Compilers should not miscompile. Our work addresses problems in developing peephole optimizations that perform local rewriting to improve the efficiency of LLVM code. These optimizations are individually difficult to get right, particularly in the presence of undefined behavior; taken together they represent a persistent source of bugs. This paper presents Alive,

a domain-specific language for writing optimizations and for automatically either proving them correct or else generating counterexamples. Furthermore, Alive can be automatically translated into C++ code that is suitable for inclusion in an LLVM optimization pass. Alive is based on an attempt to balance usability and formal methods; for example, it captures—but largely hides—the detailed semantics of three different kinds of undefined behavior in LLVM. We have translated more than 300 LLVM optimizations into Alive and, in the process, found that eight of them were wrong.

### 3.15 Symbolic Techniques for Program Debugging

*Abhik Roychoudhury (National University of Singapore, SG)*

**License** © Creative Commons BY 3.0 Unported license

© Abhik Roychoudhury

**Joint work of** Roychoudhury, Abhik; Chandra, Satish

**URL** <http://www.slideshare.net/roychoudhury/abhik-satishdagstuhl-40988809>

In recent years, there have been significant advances in symbolic execution technology, driven by the increasing maturity of SMT and SAT solvers as well as by the availability of cheap compute resources. This technology has had a significant impact in the area of automatically finding bugs in software. In this tutorial, we review ways in which symbolic execution can be used not just for finding bugs in programs, but also in debugging them! In current practice, once a failure-inducing input has been found, humans have to spend a great deal of effort in determining the root cause of the bug. The reason the task is complicated is that a person has to figure out manually how the execution of the program on the failure-inducing input deviated from the "intended" execution of the program. We show that symbolic analysis can be used to help the human in this task in a variety of ways. In particular, symbolic execution helps to glean the intended program behavior via analysis of the buggy trace, analysis of other traces or other program versions. Concretely, the tutorial provides a background in symbolic execution, and then covers material from a series of recent papers (including papers by the authors) on determination of root cause of errors using symbolic techniques.

### 3.16 Generating Heap Summaries from Symbolic Execution

*Neha Rungta (NASA – Moffett Field, US)*

**License** © Creative Commons BY 3.0 Unported license

© Neha Rungta

**Joint work of** Hillery, Ben; Mercer, Eric; Rungta, Neha; Person, Suzette

A fundamental challenge of using symbolic execution for software analysis has been the treatment of dynamically allocated data. State-of-the-art techniques have addressed this challenge through either (a) use of summaries that over-approximate possible heaps or (b) by materializing a concrete heap lazily during generalized symbolic execution. In this work, we present a novel heap initialization and analysis technique which takes inspiration from both approaches and constructs precise heap summaries lazily during symbolic execution. Our approach is 1) scalable: it reduces the points of non-determinism compared to generalized symbolic execution and explores each control-flow path only once for any given set of isomorphic heaps, 2) precise: at any given point during symbolic execution, the symbolic heap represents the exact set of feasible concrete heap structures for the program under

analysis, and 3) expressive: the symbolic heap can represent recursive data structures. We demonstrate the precision and scalability of our approach by implementing it as an extension to the Symbolic PathFinder framework for analyzing Java programs.

### 3.17 MultiSE: Multi-Path Symbolic Execution using Value Summaries

*Koushik Sen (University of California, Berkeley)*

**License** © Creative Commons BY 3.0 Unported license  
© Koushik Sen

**Joint work of** Sen, Koushik; Necula, George; Gong, Liang; Choi, Wontae

**Main reference** K. Sen, G. Necula, L. Gong, P. W. Choi, “MultiSE: Multi-Path Symbolic Execution using Value Summaries,” Technical Report, UCB/EECS-2014-173, University of California, Berkeley, 2014.

**URL** <http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-173.html>

Concolic testing of dynamic symbolic execution (DSE) has been proposed recently to effectively generate test inputs for real-world programs. Unfortunately, concolic testing techniques do not scale well for large realistic programs, because often the number of feasible execution paths of a program increases exponentially with the increase in the length of an execution path.

In this talk, I will describe MultiSE, a new technique for merging states incrementally during symbolic execution, without using auxiliary variables. The key idea of MultiSE is based on an alternative representation of the state, where we map each variable, including the program counter, to a set of guarded symbolic expressions called a value summary. MultiSE has several advantages over conventional DSE and state merging techniques: 1) value summaries enable sharing of symbolic expressions and path constraints along multiple paths, 2) value-summaries avoid redundant execution, 3) MultiSE does not introduce auxiliary symbolic values, which enables it to make progress even when merging values not supported by the constraint solver, such as floating point or function values.

We have implemented MultiSE for JavaScript programs in a publicly available open-source tool. Our evaluation of MultiSE on several programs shows that MultiSE can run significantly faster than traditional symbolic execution.

### 3.18 The Symbiosis of Network Testing and Symbolic Execution

*Oscar Soria Dustmann (RWTH Aachen University, DE)*

**License** © Creative Commons BY 3.0 Unported license  
© Oscar Soria Dustmann

Creating new adaptive Internet technologies as envisioned by the MAKI project requires the interaction of a plethora of different modules, submodules and hardware platforms. Implementation-defined behaviour and the inherent concurrency and communication delay of these systems are the primary sources of many potential types of errors. As can already be observed in current highly distributed systems such errors tend to hide in sometimes quite obscure corner-cases. For example, a catastrophic protocol lock-up might manifest only for an unlikely, unexpected packet reordering. In addition, incompatibilities stemming from the heterogeneity of subsystems, demand further attention for testing approaches. This work aims at devising methodologies that address particularly timing and heterogeneity related issues in distributed and networked systems, with a focus on event-driven software.

### 3.19 Symbolic Execution and Model Counting

*Willem Visser (Stellenbosch University – Matieland, ZA)*

**License** © Creative Commons BY 3.0 Unported license  
© Willem Visser

**Joint work of** Visser, Willem; Filieri, Antonio; Pasareanu, Corina; Dwyer, Matt; Geldenhuys, Jaco  
**Main reference** A. Filieri, C.S. Păsăreanu, W. Visser, J. Geldenhuys, “Statistical symbolic execution with informed sampling,” in Proc. of the 22nd ACM SIGSOFT Int’l Symp. on Foundations of Software Engineering (FSE’14), pp. 437–448, ACM, 2014.  
**URL** <http://dx.doi.org/10.1145/2635868.2635899>

Symbolic execution has become a very popular means for analysing program behaviour. Traditionally it only considers whether paths are feasible or not. We argue there is a wealth of interesting new research directions that open up when we also consider how likely it is that a path is feasible. We show that by using the notion of model counting (how many solutions there are rather than just whether there is a solution to a constraint) we can calculate how likely an execution path through the code is to be executed. We show how this can be used to augment program understanding, calculate the reliability of the code and also as a basis for test coverage calculations. In order to speed up calculations we use the Green framework and will give a quick primer on how to use Green. Green is a framework to allow one to reuse results across analysis runs. It speeds up satisfiability checking and model counting at the moment, but can do anything you like in a flexible framework.

### 3.20 Feedback-Driven Dynamic Invariant Discovery

*Lingming Zhang (University of Texas at Dallas, US)*

**License** © Creative Commons BY 3.0 Unported license  
© Lingming Zhang

**Joint work of** Zhang, Lingming; Yang, Guowei; Rungta, Neha; Person, Suzette; Khurshid, Sarfraz  
**Main reference** L. Zhang, G. Yang, N. Rungta, S. Person, S. Khurshid, “Feedback-driven dynamic invariant discovery,” in Proc. of the 2014 International Symposium on Software Testing and Analysis (ISSTA’14), pp. 362–372, ACM, 2014.  
**URL** <http://dx.doi.org/10.1145/2610384.2610389>

Program invariants can help software developers identify program properties that must be preserved as the software evolves, however, formulating correct invariants can be challenging. In this work, we introduce iDiscovery, a technique which leverages symbolic execution to improve the quality of dynamically discovered invariants computed by Daikon. Candidate invariants generated by Daikon are synthesized into assertions and instrumented onto the program. The instrumented code is executed symbolically to generate new test cases that are fed back to Daikon to help further refine the set of candidate invariants. This feedback loop is executed until a fix-point is reached. To mitigate the cost of symbolic execution, we present optimizations to prune the symbolic state space and to reduce the complexity of the generated path conditions. We also leverage recent advances in constraint solution reuse techniques to avoid computing results for the same constraints across iterations. Experimental results show that iDiscovery converges to a set of higher quality invariants compared to the initial set of candidate invariants in a small number of iterations.

## 4 Thought-provoking Talks

In addition to the regular 25-minutes conference-style presentations, we encouraged the attendees to give 5-minutes talks on thought-provoking ideas. A key goal behind these short talks was to promote discussions among the attendees and to rethink our future research directions on symbolic execution and constraint solving. We got warm participation from the speakers and the audience. The topics discussed in these short talks include

- the application challenges of symbolic execution,
- the challenges in processing the test cases during symbolic execution of networked code,
- the idea of bringing static analysis and symbolic execution techniques together, and
- the application of symbolic execution in security.

Overall, we found these short-talk sessions to be engaging and fun. We encourage future workshop organizers to include similar sessions in their meetings.

## 5 Programme

### Programme for Tuesday 28th of October

- Organizers: Welcome and introductions
- Cristian Cadar: Dynamic Symbolic Execution: State of the Art, Applications and Challenges
- Willem Visser: Symbolic Execution and Model Counting
- Koushik Sen: MultiSE: Multi-Path Symbolic Execution using Value Summaries
- Wei Le: Segmented Symbolic Analysis
- Peter Collingbourne: Superoptimizing LLVM
- John Regehr: Provably Correct Peephole Optimizations with Alive
- Arnaud Gotlieb: Symbolic Path-Oriented Test Data Generation for Floating-Point Programs
- Thought-provoking talks & discussions
- Indradeep Ghosh: Combining FSM Modeling and Bit-Vector Theories to Solve Hybrid String-Numeric Constraints
- Thought-provoking talks & discussions

### Programme for Wednesday 29th of October

- Abhik Roychoudhury and Satish Chandra: Symbolic Techniques for Program Debugging
- Thought-provoking talks & discussions
- Vijay Ganesh: Impact of Community Structure on SAT Solver Performance
- Albert Oliveras: Solving Non-linear Integer Constraints Arising from Program Analysis
- Alastair F. Donaldson: Experiences with SMT in the GPUVerify Project
- Hristina Palikareva: Constraint Solving in Symbolic Execution
- Juan Pablo Galeotti: On the Challenges of Combining Search-Based Software Testing and Symbolic Execution
- Paul Marinescu: Commercial Symbolic Execution
- Maria Christakis: Reaching Verification with Systematic Testing
- Thought-provoking talks & discussions

**Programme for Thursday 30th of October**

- Neha Rungta: Generating Heap Summaries from Symbolic Execution
- Oscar Soria Dustmann: The Symbiosis of Network Testing and Symbolic Execution
- Lingming Zhang: Feedback-Driven Dynamic Invariant Discovery
- Istvan Haller: Symbolic execution in the field of security
- Sébastien Bardin: Automated White-Box Testing Beyond Branch Coverage
- Nicky Williams: Introduction to PathCrawler



## Participants

- Sébastien Bardin  
CEA LIST – Paris, FR
- Earl Barr  
University College London, UK
- Cristian Cadar  
Imperial College London, UK
- Satish Chandra  
Samsung Electronics –  
San Jose, US
- Maria Christakis  
ETH Zürich, CH
- Peter Collingbourne  
Google Inc. –  
Mountain View, US
- Jorge R. Cuéllar  
Siemens AG – München, DE
- Morgan Deters  
New York University, US
- Alastair F. Donaldson  
Imperial College London, UK
- Juan Pablo Galeotti  
Universität des Saarlandes –  
Saarbrücken, DE
- Vijay Ganesh  
University of Waterloo, CA
- Indradeep Ghosh  
Fujitsu Labs of America Inc. –  
Sunnyvale, US
- Arnaud Gotlieb  
Simula Research Laboratory –  
Lysaker, NO
- Istvan Haller  
Free Univ. of Amsterdam, NL
- Wei Le  
Iowa State Univ. – Ames, US
- Paul Marinescu  
Imperial College London, UK
- Benjamin Mehne  
University of California –  
Berkeley, US
- Martin Ochoa  
TU München – DE
- Albert Oliveras  
Polytechnic University of  
Catalonia – Barcelona, SP
- Hristina Palikareva  
Imperial College London, UK
- Ruzica Piskac  
Yale University – New Haven, US
- John Regehr  
University of Utah – Salt Lake  
City, US
- Abhik Roychoudhury  
National University of  
Singapore – SG
- Neha Rungta  
NASA – Moffett Field, US
- Raimondas Sasnauskas  
University of Utah – Salt Lake  
City, US
- Koushik Sen  
University of California –  
Berkeley, US
- Oscar Soria Dustmann  
RWTH Aachen, DE
- Nikolai Tillmann  
Microsoft Corporation –  
Redmond, US
- Willem Visser  
Stellenbosch University –  
Matieland, ZA
- Klaus Wehrle  
RWTH Aachen, DE
- Nicky Williams  
CEA LIST – Paris, FR
- Christoph M. Wintersteiger  
Microsoft Res. – Cambridge, UK
- Lingming Zhang  
University of Texas at Dallas, US

