Report from Dagstuhl Seminar 18151

# Program Equivalence

**Edited by**

**Shuvendu K. Lahiri[1], Andrzej Murawski[2], Ofer Strichman[3], and Mattias Ulbrich[4]**

1 **Microsoft Research – Redmond, US,** shuvendu.lahiri@microsoft.com
2 **University of Oxford, GB,** andrzej.murawski@cs.ox.ac.uk
3 **Technion – Haifa, IL,** ofers@ie.technion.ac.il
4 **KIT – Karlsruher Institut für Technologie, DE,** ulbrich@kit.edu

## Abstract

Program equivalence is the problem of proving that two programs are equal under some definition of equivalence, e.g., input-output equivalence. The field draws researchers from formal verification, semantics and logics.

This report documents the program and the outcomes of Dagstuhl Seminar 18151 "Program Equivalence". The seminar was organized by the four official organizers mentioned above, and Dr. Nikos Tzevelekos from Queen-Mary University in London.

**Seminar** April 8–13, 2018 – http://www.dagstuhl.de/18151
**2012 ACM Subject Classification** Software and its engineering → Software verification, Software and its engineering → Semantics
**Keywords and phrases** program equivalence, regression-verification, translation validation
**Digital Object Identifier** 10.4230/DagRep.8.4.1

## 1 Executive summary

*Shuvendu K. Lahiri*
*Andrzej Murawski*
*Ofer Strichman*
*Mattias Ulbrich*



Program equivalence is arguably one of the most interesting and at the same time important problems in formal verification. It has attracted the interest of several communities, ranging from the field of denotational semantics and the problem of Full Abstraction, to software verification and Regression Testing. The aim of this meeting was to bring together the different approaches and techniques of the current state of the art and to facilitate the cross-pollination of research between these areas.

This interdisciplinary community met once before in the workshop on program equivalence in London (April 2016). There was a general agreement among the participants that a research community around this topic should be established in the form of a workshop and eventually a conference, and that the interest in this topic continuously grows around the world, including a growing interest in the industry. Furthermore, currently there is little overlap in the conferences that some of the key players attend, to the point that many participants were little aware of other participants' work.

We were happy to witness that indeed participants learned greatly from this week, collaborations were established, and cross fertilization between the communities occurred. We hope to meet again in Dagstuhl in the future!

## 2 Table of Contents

# 3 Overview of Talks

## 3.1 Relational Logic with Framing and Hypotheses (status report)

*Anindya Banerjee (NSF – Alexandria, US) and David A. Naumann (Stevens Institute of Technology – Hoboken, US)*


**Joint work of** Anindya Banerjee, David Naumann, Mohammad Nikouei

Relational properties arise in many settings: relating two versions of a program that use different data representations, noninterference properties for security, etc. The main ingredient of relational verification, relating aligned pairs of intermediate steps, has been used in numerous guises, but existing relational program logics are narrow in scope. We are investigating a logic based on novel syntax that weaves together product programs to express alignment of control flow points at which relational formulas are asserted. Correctness judgments feature hypotheses with relational specifications, discharged by a rule for the linking of procedure implementations. The logic supports reasoning about program-pairs containing both similar and dissimilar control and data structures. Reasoning about dynamically allocated objects is supported by a frame rule based on frame conditions amenable to SMT provers. In this talk we give an overview of the project ideas and status.

## 3.2 Semantic Differencing for HipHop Bytecode

*Nick Benton (Facebook – London, GB)*



We describe a semantic differencing tool used to compare the bytecodes generated by two different compilers for Hack/PHP at Facebook. The tool is a prover for a simple relational Hoare logic for low-level code and is used in testing, allowing the developers to focus on semantically significant differences between the outputs of the two compilers.

## 3.3 Verification with Reusing Exchangeable Results (Conditions, Witnesses, Precisions)

*Dirk Beyer (LMU München, DE)*



This presentation covers the topic of exchangable verification results. First, we explain how the conditions of conditional model checking [1] can be used to pass information from one verifier to another, in particular, the first verifier describes in the condition the parts of the state space that it was able to successfully verify, while the second verifying can use the condition of the first verifier in order to concentrate on the state space that the first verifier did not succeed on [2]. Second, abstraction based approaches (cf. CEGAR) need to compute the abstract model, i.e., specify a precision that defines the level of abstraction (set of predicates for predicate abstraction, set of variables for value analysis). The precision

is a valuable piece of information that should be reused when verifying a similar program (as, e.g., in regression verification) /citePrecisionReuse. Third, verification witnesses are exchangeable objects that contain information that another verification tool (validator) can use to re-establish the verification result [4, 5, 3]. Witnesses enable many new opportunities to improve the value of verification tools for the user, e.g., by supporting verification-based debugging [6].

### References


1 D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. 2012. Conditional Model Checking: A Technique to Pass Information between Verifiers. In *Proc. FSE*. ACM, Article 57, 57:1–57:11 pages.ISBN:978-1-4503-1614-9, https://doi.org/10.1145/2393596.2393664

2 Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. 2018 Reducer-Based Construction of Conditional Verifiers. In *Proc. ICSE*. ACM. https://doi.org/10.1145/3180155.3180259

3 D. Beyer and P. Wendler. 2013 Reuse of Verification Results: Conditional Model Checking, Precision Reuse, and Verification Witnesses. In *Proc. SPIN LNCS 7976)*. Springer, 1–17. https://doi.org/10.1007/978-3-642-39176-7_1

4 D. Beyer,M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. 2015 Witness Validation and Stepwise Testification across Software Verifiers. In *Proc. FSE*. ACM, 721–733. ISBN: 978-1-4503-3675-8. https://doi.org/10.1145/2786805.2786867

5 D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. 2016. Correctness Witnesses: Exchanging Verification Results Between Verifiers. In *Proc. FSE*. ACM, 326–337. https://doi.org/10.1145/2950290.2950351

6 Dirk Beyer and Matthias Dangl. 2016. Verification-Aided Debugging: An Interactive Web-Service for Exploring Error Witnesses. In *Proc. CAV (2) LNCS 9780)*. Springer, 502–509. https://doi.org/10.1007/978-3-319-41540-6_28


## 3.4 Validating Optimizations of Concurrent C/C++ Programs

*Soham Chakraborty (MPI-SWS – Kaiserslautern, DE)*



Compilation of C/C++ shared memory concurrent programs faces many challenges. On the one hand, C/C++ concurrency enable multiple transformations on shared memory accesses and fences. On the other hand, not all transformations which are correct for sequential programs are correct in the concurrent setting. Thus, compiler writers have to perform careful analysis to determine which transformations are correct.

In this talk I will present our work on validating the optimizations of LLVM, a state-of-the-art C/C++ compiler. Our work has revealed some previously unknown bugs in LLVM concerning the compilation of concurrent C/C++ programs.

## 3.5 Semantics-Parametric Program Equivalence

*Stefan Ciobaca (University AI. I. Cuza – Iasi, RO)*


**Joint work of** Stefan Ciobaca, Dorel Lucanu

The operational semantics of any programming language can be modeled as a set of constrained rewrite rules of the form "l rewrites into r if b", where l and r are terms representing program configurations and where b is a logical constraint. The rewrite rules are interpreted in an algebra of program configurations and the reduction relation generated by the rewrite rules is the one-step transition relation.

Using this encoding, we can prove program equivalence in a semantics-parametric manner. We build an equivalence checker E(P, Q, L, R) that takes as input not only two programs P and Q that we want to prove equivalent, but also the operational semantics L and R of the programming languages of P and Q.

We implement the equivalence checker and show that it works on several examples, which cover both imperative and functional languages. This is work-in-progress.

This work was supported by a grant of the Romanian National Authority for Scientific Research and Innovation, CNCS/CCCDI – UEFISCDI, project number PN-III-P2-2.1-BG-2016-0394, within PNCDI III.

## 3.6 Abstract Semantic Diffing of Evolving Concurrent Programs

*Constantin Enea (University Paris-Diderot, FR)*


**Joint work of** Ahmed Bouajjani, Constantin Enea, Shuvendu K. Lahiri
**Main reference** Ahmed Bouajjani, Constantin Enea, Shuvendu K. Lahiri: "Abstract Semantic Diffing of Evolving Concurrent Programs", in Proc. of the Static Analysis – 24th International Symposium, SAS 2017, New York, NY, USA, August 30 – September 1, 2017, Proceedings, Lecture Notes in Computer Science, Vol. 10422, pp. 46–65, Springer, 2017.
**URL** http://dx.doi.org/10.1007/978-3-319-66706-5_3

We present an approach for comparing two closely related concurrent programs, whose goal is to give feedback about interesting differences without relying on user-provided assertions. This approach compares two programs in terms of cross-thread interferences and data-flow, under a parametrized abstraction which can detect any difference in the limit. We introduce a partial order relation between these abstractions such that a program change that leads to a "smaller" abstraction is more likely to be regression-free from the perspective of concurrency. On the other hand, incomparable or bigger abstractions, which are an indication of introducing new, possibly undesired, behaviors, lead to succinct explanations of the semantic differences.

## 3.7 Property Directed Equivalence via Abstract Simulation

*Grigory Fedyukovich (Princeton University, US)*


**Joint work of** Grigory Fedyukovich, Arie Gurfinkel, Natasha Sharygina
**Main reference** Grigory Fedyukovich, Arie Gurfinkel, Natasha Sharygina: "Property Directed Equivalence via Abstract Simulation", in Proc. of the Computer Aided Verification – 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II, Lecture Notes in Computer Science, Vol. 9780, pp. 433–453, Springer, 2016.
**URL** http://dx.doi.org/10.1007/978-3-319-41540-6_24

Numerous versions of software have to be designed, developed, and verified before the product is ready for a release. Each version suffers from bugs which have to be fixed and should not appear again in the future. Once a version is formally verified for safety, its proof should be made available for verification of the coming versions. However, vast majority of verification tools are tailored to verification of each new program version in isolation from the version history. We present an approach for incremental verification based on constrained Horn clauses that lifts the proofs across program modifications. The key idea behind our approach is to establish a property directed equivalence between pairs of program versions, and we propose a way to do it through synthesis of simulation relations. We present the implementation and evaluation of the algorithm supporting our hypothesis that incremental verification can be performed efficiently, even if the program modifications are non-trivial. In cases when the complete proof lifting is impossible, our tool lifts the proof partially, which further allows the generation of the missing parts of the proof, or the calculation of a change impact certificate.

## 3.8 A graph-rewriting refinement of the $\beta$ law

*Dan R. Ghica (University of Birmingham, GB)*


**Joint work of** Dan R. Ghica, Koko Muroya, Todd Waugh Ambridge

The newly developed Dynamic Geometry of Interaction is a graph-rewriting abstract machine based on Girard's semantics of linear logic proofs. It can model in a unified setting all the reduction strategies of the lambda calculus, also giving accurate cost models for execution. Using it we can refined the standard beta law of the lambda calculus into four, simpler, graph-rewriting laws.

## 3.9 Regression Verification of Multi-Threaded Programs

*Arie Gurfinkel (University of Waterloo, CA) and Ofer Strichman (Technion – Haifa, IL)*


**Joint work of** Arie Gurfinkel, Sagar Chaki, Ofer Strichman
**Main reference** Sagar Chaki, Arie Gurfinkel, Ofer Strichman: "Regression verification for multi-threaded programs (with extensions to locks and dynamic thread creation)", Formal Methods in System Design, Vol. 47(3), pp. 287–301, 2015.
**URL** http://dx.doi.org/10.1007/s10703-015-0237-0
**Main reference** Sagar Chaki, Arie Gurfinkel, Ofer Strichman: "Regression Verification for Multi-threaded Programs", in Proc. of the Verification, Model Checking, and Abstract Interpretation – 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings, Lecture Notes in Computer Science, Vol. 7148, pp. 119–135, Springer, 2012.
**URL** http://dx.doi.org/10.1007/978-3-642-27940-9_9

Regression verification is the problem of deciding whether two similar programs are equivalent under an arbitrary yet equal context, given some definition of equivalence. So far this problem has only been studied for the case of single-threaded deterministic programs. We present a method for regression verification to establish partial equivalence (i.e., input/output equivalence of terminating executions) of multi-threaded programs. Specifically, we develop two proof-rules that decompose the regression verification between concurrent programs to that of regression verification between sequential functions, a more tractable problem. This ability to avoid composing threads altogether when discharging premises, in a fully automatic way and for general programs, uniquely distinguishes our proof rules from others used for classical verification of concurrent programs.

## 3.10 Model-checking contextual equivalence of higher-order programs with references

*Guilhem Jaber (ENS – Lyon, FR)*



This talk will present SyTeCi, a general automated tool to check contextual equivalence for programs written in a typed higher-order language with references (i.e. local mutable states), corresponding to a fragment of OCaml. After introducing the notion of contextual equivalence, we will see on some examples why it is hard to prove such equivalences (reentrant calls, private states). Then, we will introduce SyTeCi, a tool to automatically check such equivalences. This tool is based on a reduction of the problem of contextual equivalence of two programs to the problem of reachability of "error states" in a transition system of memory configurations. Contextual equivalence being undecidable (even in a finitary setting), so does the non-reachability problem for such transition systems. However, one can apply model-checking techniques (predicate abstraction, analysis of pushdown systems) to check non-reachability via some approximations. This allows us to prove automatically many non-trivial examples of the literature, that could only be proved by hand before. We will end this talk by the presentation of a prototype implementing this work.

## 3.11 Distinguishing between Communicating Transactions

*Vasileios Koutavas (Trinity College Dublin, IE)*


**Joint work of** Vasileios Koutavas, Maciej Gazda, Matthew Hennessy
**Main reference** Vasileios Koutavas, Maciej Gazda, Matthew Hennessy: "Distinguishing between communicating transactions", Inf. Comput., Vol. 259(1), pp. 1–30, 2018.
**URL** http://dx.doi.org/10.1016/j.ic.2017.12.001

Communicating transactions is a form of distributed, non-isolated transactions which provides a simple construct for building concurrent systems. We will explore the observable behaviour of such systems through different nominal modal logics which share standard communication modalities, but have distinct past and future modalities involving transactional commits. We will discuss how, although quite different, the distinguishing power of these logics is identical. Furthermore, they are equally expressive because there are semantics-preserving translations between their formulae. Using the logics we can clearly exhibit subtle example inequivalences between communicating transactions, sheding light on the behaviour of such constructs.

## 3.12 DEEPSEC: Deciding Equivalence Properties in Security Protocols

*Steve Kremer (INRIA Nancy – Grand Est, FR)*


**Joint work of** Vincent Cheval, Steve Kremer, Itsaka Rakotonirina
**Main reference** Vincent Cheval, Steve Kremer, Itsaka Rakotonirina: "DEEPSEC: Deciding Equivalence Properties in Security Protocols – Theory and Practice". In Proc. of the 39th IEEE Symposium on Security and Privacy (S&P'18), pp. 525–542, IEEE Computer Society Press, San Francisco, CA, USA, May 2018.
**URL** http://doi.ieeecomputersociety.org/10.1109/SP.2018.00033

Automated verification has become an essential part in the security evaluation of cryptographic protocols. Recently, there has been a considerable effort to lift the theory and tool support that existed for reachability properties to the more complex case of equivalence properties. In this talk I will report on our recent advances in theory and practice of this verification problem. We establish new complexity results for static equivalence, trace equivalence and labelled bisimilarity and provide a decision procedure for these equivalences in the case of a bounded number of sessions. Our procedure is the first to decide trace equivalence and labelled bisimilarity exactly for a large variety of cryptographic primitives—those that can be represented by a subterm convergent destructor rewrite system. We implemented the procedure in a new tool, DEEPSEC. We showed through extensive experiments that it is significantly more efficient than other similar tools, while at the same time raises the scope of the protocols that can be analysed.

## 3.13 Interprocedural Relational Verification in SymDiff and Applications

*Shuvendu K. Lahiri (Microsoft Research – Redmond, US)*



**Main reference** Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, Chris Hawblitzel: "Differential assertion checking", in Proc. of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pp. 345–355, ACM, 2013.
**URL** http://dx.doi.org/10.1145/2491411.2491452
**Main reference** Chris Hawblitzel, Ming Kawaguchi, Shuvendu K. Lahiri, Henrique Rebêlo: "Towards Modularly Comparing Programs Using Automated Theorem Provers", in Proc. of the Automated Deduction – CADE-24 – 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings, Lecture Notes in Computer Science, Vol. 7898, pp. 282–299, Springer, 2013.
**URL** http://dx.doi.org/10.1007/978-3-642-38574-2_20
**Main reference** Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, Henrique Rebêlo: "SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs", in Proc. of the Computer Aided Verification – 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, Lecture Notes in Computer Science, Vol. 7358, pp. 712–717, Springer, 2012.
**URL** http://dx.doi.org/10.1007/978-3-642-31424-7_54
**Main reference** Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, Sam Blackshear: "Verification modulo versions: towards usable verification", in Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom – June 09 – 11, 2014, pp. 294–304, ACM, 2014.
**URL** http://dx.doi.org/10.1145/2594291.2594326
**Main reference** Shaobo He, Shuvendu K. Lahiri, Zvonimir Rakamaric: "Verifying Relative Safety, Accuracy, and Termination for Program Approximations", J. Autom. Reasoning, Vol. 60(1), pp. 23–42, 2018.
**URL** http://dx.doi.org/10.1007/s10817-017-9421-9

In this talk, I describe the SymDiff tool that is a verifier for proving properties of program differences. Differential program verification concerns with proving interesting properties over program differences, as opposed to the program itself. Such properties include program equivalence, but can also captures more general differential/relational properties. SymDiff provides a specification language to state such differential (two-program) properties using the concept of mutual summaries that can relate procedures from two versions. It also provides proof system for checking such differential specifications along with the capability of generating simple differential invariants.

We describe applications of SymDiff towards interprocedural equivalence checking, cross-version compiler validation, differential assertion checking, checking the safety of approximate transformations and for semantic change impact analysis.

## 3.14 Polymorphic Game Semantics for Dynamic Binding

*James Laird (University of Bath, GB)*



**Main reference** James Laird: "Polymorphic Game Semantics for Dynamic Binding", in Proc. of the 25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 – September 1, 2016, Marseille, France, LIPIcs, Vol. 62, pp. 27:1–27:16, Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2016.
**URL** http://dx.doi.org/10.4230/LIPIcs.CSL.2016.27

We present a game semantics for an expressive typing system for block-structured programs with late binding of variables and System F style polymorphism. As well as generic programs and abstract datatypes, this combination may be used to represent behaviour such as dynamic dispatch and method overriding.

We give a denotational models for a hierarchy of programming languages based on our typing system, including variants of PCF and Idealized Algol. These are obtained by extending polymorphic game semantics to block-structured programs. We show that the categorical structure of our models can be used to give a new interpretation of dynamic binding, and establish definability properties by imposing constraints which are identical or similar to those used to characterize definability in PCF (innocence, well-bracketing, determinacy). Moreover, relaxing these can similarly allow the interpretation of side-effects (state, control, non-determinism) – we show that in particular we may obtain a fully abstract semantics of polymorphic Idealized Algol with dynamic binding by following exactly the methodology employed in the simply-typed case.

## 3.15 Program equivalences and program refinements for compiler verification

*Xavier Leroy (INRIA – Paris, FR)*



Verifying the soundness of a compiler means proving that the generated code behaves as prescribed by the semantics of the source program. There are many definitions of interest for "behaves as prescribed". Observational equivalence is appropriate for well-defined source languages such as Java. However, for C and C++, observational equivalence cannot be guaranteed because several evaluation orders are allowed for source programs, while the compiled code implements one of those evaluation orders. Moreover, C and C++ treat run-time errors such as integer division by zero or out-of-bound array accesses as undefined behaviors, meaning that the compiled code is allowed to perform any actions whatsoever, from aborting the program to continuing with random values to opening a security hole.

The CompCert compiler verification project builds on a notion of program refinement that enables the compiler to choose one among several possible evaluation orders, making the program "more deterministic", and also to optimize source-level undefined behaviors away, making the program "more defined". An example of the latter dimension of refinement is the elimination of an integer division `z = x / y` if `z` is unused later: if `y` is 0, the original program exhibits undefined behavior (division by zero), but not the optimized program.

Program refinement is proved using simulation diagrams between the labeled transition systems that define the semantics of the original and transformed program. In full generality a so-called backward simulation diagram is needed, relating every transition of the transformed program with zero, one or several transitions of the original program, provided the original program is at a safe state (a state that cannot silently reach undefined behavior). For compilation passes that preserve the amount of nondeterminism, a simpler proof is possible as a forward simulation diagram, relating transitions of the original program with sequences of transitions of the transformed program.

This notion of program refinement and the associated proof techniques have served CompCert well so far, but can be difficult to extend to aggressive optimizations, other properties of interest, or other language features of interest. For example, loop optimizations such as loop exchange or loop blocking change control flow in a non-local manner, renewing interest in more denotational or more relational alternatives to simulation diagrams. Interesting program properties that we would like to see and to prove preserved during compilation

include constant-time cryptography, i.e. the fact that secret data is never used as argument to conditional branches, memory addressing, or other operations whose execution time depend on the value of the arguments. Finally, shared-memory concurrency is a challenge for compiler verification. Many compiler optimizations and code generation scheme that are valid for sequential programs remain valid for concurrent programs that are free of data races. Controlled data races, as supported by the low-level atomics of C and C++ 2011, raise many more challenges and are only starting to be understood semantically.

## 3.16 Client-Specific Equivalence Checking – An Overview

*Yi Li (University of Toronto, CA) and Julia Rubin (University of British Columbia – Vancouver, CA)*


**Joint work of** Yi Li, Federico Mora, Marsha Chechik, Julia Rubin

Software is often built by integrating components created by different teams or even different organizations. Changes in one component may trigger a sequence of updates to its downstream clients. To avoid dealing with updates, developers often delay upgrades, negatively affecting correctness and robustness of their systems. In this work, we investigate the effect of component changes on the behaviour of their clients. We observe that changes in a component are often irrelevant to a particular client and thus can be adopted without any delays or negative effects. Following this observation, we formulate the notion of client-specific equivalence checking (CSE), lay out particular challenges and opportunities, and discuss possible solutions for checking such equivalence. We also present our early findings and propose some promising directions for further exploration.

## 3.17 Semantic Program Repair Using a Reference Implementation

*Sergey Mechtaev (National University of Singapore, SG)*


**Joint work of** Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske and Abhik Roychoudhury
**Main reference** Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske and Abhik Roychoudhury: "Semantic Program Repair Using a Reference Implementation" In Proc. of 40th Int'l Conference on Software Engineering, Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18)
**URL** https://doi.org/10.1145/3180155.3180247

The goal of program repair is to automatically modify a given incorrect program to eliminate the observable failures. One of the key challenges of this technology is that a formal specification of the intended behavior is typically not available in practice, and the use of a test suite as a correctness criteria often leads to the generation of incorrect patches that merely overfit the tests.

Semantic program repair aims to understand the meaning of software defects by means of semantic program analysis. This approach has two advantages over previous syntactic techniques. First, semantic analysis helps to efficiently navigate the conceptually large search space of patches. Second, semantic analysis helps to compensate the lack of correctness specification in real-world software.

In this talk, we discuss a semantic approach of generating program patches using a reference implementation. Specifically, this approach extracts specification from a reference implementation and generates a patch that enforces conditional equivalence of the patched and the reference programs w.r.t. a user-defined inputs condition. We demonstrate the effectiveness of this techniques in our experiments with GNU Coreutils and Busybox that implement the same set of UNIX utilities. We also discuss how this technique contributes to a broader vision of a general-purpose program repair system that will able to address many types of defects in commodity software and have applications in software development, security and education.

## 3.18 An introduction to game semantics

*Andrzej Murawski (University of Oxford, GB)*



I will give an introductory talk on game semantics, which is a modelling theory for higher-order programming languages based on the metaphor of game playing. Over the last 25 years, game semantics has been used to obtain the first full abstraction results for a wide spectrum of programming languages (full abstraction means that interpretations of two programs coincide exactly when the programs are equivalent). More recently, game models have been exploited to classify decidable (wrt program equivalence) fragments of various programming languages based on their type signatures. I will give a brief survey of the results and mention the kinds of automata that have turned out useful in capturing the dynamics of game models.

## 3.19 Trace Equivalence For Android Malware Detection

*Julia Rubin (University of British Columbia – Vancouver, CA)*


**Joint work of** Khaled Ahmed, Mieszko Lis, Julia Rubin

In this talk, we will present a novel approach we propose for efficiently detecting Android malware at runtime. Our approach relies on monitoring application execution, collecting execution traces, and then reasoning about equivalent vs. different traces. We will discuss characteristics of Android malware and identify a notion of trace equivalence that helps detect such malware. We will then show that the optimal notion of equivalence is impractical to implement due to the event-driven and multi-threaded nature of the Android system, and examine other possible solutions.

## 3.20 Proofs for Performance

*Rahul Sharma (Microsoft Research India – Bangalore, IN)*


**Joint work of** Eric Schkufza, Berkeley Churchill, Alex Aiken
**Main reference** Rahul Sharma, Eric Schkufza, Berkeley R. Churchill, Alex Aiken: "Data-driven equivalence checking", in Proc. of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013, pp. 391–406, ACM, 2013.
**URL** http://dx.doi.org/10.1145/2509136.2509509

Automated formal reasoning has the potential to significantly improve the quality of compiler generated code. We describe a data-driven approach to such reasoning: the proof steps are assisted by analysis applied to data gathered from program executions. We show how data-driven equivalence checking proves the correctness of code generated by production compilers (such as GCC with all optimizations enabled) by generating a formal proof of equivalence between a C source and the compiler generated x86 binary. Moreover, this equivalence checker lets us generate provably correct code that is up to 70% faster than the compiler generated code. Furthermore, we show how data-driven precondition inference lets us generate code that can be multiple times faster than compiler generated code.

## 3.21 Program equivalence problems in computational science

*Stephen Siegel (University of Delaware – Newark, US)*


**Main reference** Stephen F. Siegel, Manchun Zheng, Ziqing Luo, Timothy K. Zirkel, Andre V. Marianiello, John G. Edenhofner, Matthew B. Dwyer, Michael S. Rogers: "CIVL: the concurrency intermediate verification language", in Proc. of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2015, Austin, TX, USA, November 15-20, 2015, pp. 61:1–61:12, ACM, 2015.
**URL** http://dx.doi.org/10.1145/2807591.2807635

A number of interesting program equivalence problems arise in computational science. Many algorithms used in that domain have a straightforward implementation–e.g., matrix multiplication. But these straightforward implementations are then transformed in innumerable ways, e.g., to reduce the number of floating-point operations, to use cache more efficiently, and to take advantage of parallel hardware. The transformed programs are expected to be equivalent – in some sense – to the original simple versions. In this talk I will describe several examples of such problems, and our use of symbolic execution tools (CIVL and TASS) to solve them. In many cases, equivalence can be established within some small bounds (on the sizes of inputs, number of processes, etc.), but some progress has been made on proofs without such bounds.

## 3.22 Proving Mutual Termination

*Ofer Strichman (Technion – Haifa, IL)*


**Joint work of** Dima Elenbogen, Shmuel Katz, Ofer Strichman
**Main reference** Dima Elenbogen, Shmuel Katz, Ofer Strichman: "Proving mutual termination", Formal Methods in System Design, Vol. 47(2), pp. 204–229, 2015.
**URL** http://dx.doi.org/10.1007/s10703-015-0234-3

Two programs are said to be mutually terminating if they terminate on exactly the same inputs. We suggest inference rules and a proof system for proving mutual termination of a given pair of procedures <f, f'> and the respective subprograms that they call under a free context. Given a (possibly partial) mapping between the procedures of the two programs, the premise of the rule requires proving that given the same arbitrary input $in$, $f(in)$ and $f'(in)$ call procedures mapped in the mapping with the same arguments. A variant of this proof rule with a weaker premise allows to prove termination of one of the programs if the other is known to terminate. In addition, we suggest various techniques for battling the inherent incompleteness of our solution, including a case in which partial equivalence (the equivalence of their input/output behavior) has only been proven for some, but not all, the outputs of the two given procedures. We present an algorithm for decomposing the verification problem of whole programs to that of proving mutual termination of individual procedures, based on our suggested inference rules. In this talk I will survey our work on proving mutual termination of programs and demo our prototype implementation of this algorithm.

## 3.23 The Software Analysis Workbench

*Aaron Tomb (Galois – Portland, US)*


**Joint work of** Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, Aaron Tomb
**Main reference** Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, Aaron Tomb: "Constructing Semantic Models of Programs with the Software Analysis Workbench", in Proc. of the Verified Software. Theories, Tools, and Experiments – 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers, Lecture Notes in Computer Science, Vol. 9971, pp. 56–72, 2016.
**URL** http://dx.doi.org/10.1007/978-3-319-48869-1_5

The Software Analysis Workbench (SAW) is a tool for transforming programs into models of their functional behavior, manipulating those models, and using various third-party reasoning tools to prove properties of those models. Although it is in principle more than a program equivalence checking tool, it is most highly tuned for proving equivalence.

SAW currently uses symbolic execution to construct program models, and uses path merging and path satisfiability checking to increase the class of programs for which it can generate a single, complete model. As a result, the current behavior is roughly an instance of bounded model checking, with the bounds provided by the program rather than some fixed constant.

SAW integrates closely with Cryptol, a domain-specific functional language originally designed for the high-level description of cryptographic algorithms, and generally well-suited to describing finite programs of the sort that are most amenable to analysis with SAT

and SMT. We have used SAW to prove functional equivalence between many imperative implementations of cryptographic algorithms and high-level specifications written in Cryptol.

This talk describes some of the techniques used in SAW along with some examples of the concrete implementations we have used it to verify.

## 3.24 Nominal Games: A Semantics Paradigm for Effectful Languages

*Nikos Tzevelekos (Queen Mary University of London, GB)*


**Joint work of** Andrzej Murawski, Steven Ramsay, Dan Ghica, Guilhem Jaber, Thomas Cuvillier, Nikos Tzevelekos

Game semantics has been developed since the 90's as a denotational paradigm capturing observational equivalence of functional languages with imperative features. While initially introduced for PCF variants, the theory can nowadays express effectful languages ranging from ML fragments and Java programs to C-like code. In this talk we present recent advances in devising game models for effectful computation. Central in this approach is the use of names for representing in an abstract fashion different forms of notions and effects, such as references, higher-order values and polymorphism. We moreover look at automata models relevant to nominal games and how can they be used for model checking program equivalence.

## 3.25 Relational Equivalence Proofs Between Imperative and MapReduce Algorithms

*Mattias Ulbrich (KIT – Karlsruher Institut für Technologie, DE)*


**Main reference** Bernhard Beckert, Timo Bingmann, Moritz Kiefer, Peter Sanders, Mattias Ulbrich, Alexander Weigl: "Relational Equivalence Proofs Between Imperative and MapReduce Algorithms", CoRR, Vol. abs/1801.08766, 2018.
**URL** http://arxiv.org/abs/1801.08766

MapReduce frameworks are widely used for the implementation of distributed algorithms. However, translating imperative algorithms into these frameworks requires significant structural changes to the algorithm. As the costs of running faulty algorithms at scale can be severe, it is highly desirable to verify the correctness of the translation, i.e., to prove that the MapReduce version is equivalent to the imperative original. We present a novel approach for proving equivalence between imperative and MapReduce algorithms based on partitioning the equivalence proof into a sequence of equivalence proofs between intermediate programs with smaller differences. Our approach is based on the insight that two kinds of sub-proofs are required: (1) uniform transformations rewriting the controlflow structure that are mostly independent of the particular context in which they are applied; and (2) context-dependent transformations that are not uniform but that preserve the overall structure and can be proved correct using coupling invariants. I demonstrated the feasibility of our approach by applying it to the PageRank algorithm. The potential for automation has been discussed.

## 3.26 A Behavioural Equivalence for Algebraic Effects: Logic with Modalities

*Niels Voorneveld (University of Ljubljana, SI)*


**Joint work of** Alex Simpson, Niels F. W. Voorneveld
**Main reference** Alex Simpson, Niels F. W. Voorneveld: "Behavioural Equivalence via Modalities for Algebraic Effects", in Proc. of the Programming Languages and Systems – 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Lecture Notes in Computer Science, Vol. 10801, pp. 300–326, Springer, 2018.
**URL** http://dx.doi.org/10.1007/978-3-319-89884-1_11

In this talk we investigate behavioural equivalence between programs of a functional language extended with a signature of (algebraic) effect-triggering operations. Two programs are considered behaviourally equivalent if they enjoy the same behavioural properties. To formulate this, we define a logic whose formulas specify these behavioural properties. A crucial ingredient is a collection of modalities expressing effect-specific aspects of behaviour. The construction of the logic and the theory of such modalities are outlined.

We look at examples of effects and what modalities we may choose for them. These examples include: nondeterminism, error, probabilistic choice, global store and input/output. Moreover, we will briefly look at two technical conditions for such modalities which, if satisfied, makes the logically-specified behavioural equivalence a congruence. The given examples satisfy these properties. The induced behavioural equivalence is also related to a notion of Abramsky's applicative bisimilarity.

## Participants

- Anindya Banerjee
NSF – Alexandria, US
- Gilles Barthe
IMDEA Software – Madrid, ES
- Nick Benton
Facebook – London, GB
- Dirk Beyer
LMU München, DE
- Soham Chakraborty
MPI-SWS – Kaiserslautern, DE
- Stefan Ciobaca
University AI. I. Cuza – Iasi, RO
- Constantin Enea
University Paris-Diderot, FR
- Grigory Fedyukovich
Princeton University, US
- Dan R. Ghica
University of Birmingham, GB
- Arie Gurfinkel
University of Waterloo, CA
- Guilhem Jaber
ENS – Lyon, FR
- Vasileios Koutavas
Trinity College Dublin, IE
- Steve Kremer
INRIA Nancy – Grand Est, FR
- Shuvendu K. Lahiri
Microsoft Research – Redmond, US
- James Laird
University of Bath, GB
- Xavier Leroy
INRIA – Paris, FR
- Yi Li
University of Toronto, CA
- Sergey Mechtaev
National University of Singapore, SG
- Andrzej Murawski
University of Oxford, GB
- Kedar Namjoshi
Nokia Bell Labs – Murray Hill, US
- David A. Naumann
Stevens Institute of Technology – Hoboken, US
- Julia Rubin
University of British Columbia – Vancouver, CA
- Philipp Rümmer
Uppsala University, SE
- Neha Rungta
Amazon.com, Inc. – Palo Alto, US
- Chaked Saydoff
Technion – Haifa, IL
- Rahul Sharma
Microsoft Research India – Bangalore, IN
- Stephen Siegel
University of Delaware – Newark, US
- Marcelo Sousa
University of Oxford, GB
- Ofer Strichman
Technion – Haifa, IL
- Aaron Tomb
Galois – Portland, US
- Nikos Tzevelekos
Queen Mary University of London, GB
- Mattias Ulbrich
KIT – Karlsruher Institut für Technologie, DE
- Niels Voorneveld
University of Ljubljana, SI