

Report from Dagstuhl Seminar 16201

Synergies among Testing, Verification, and Repair for Concurrent Programs

Edited by

Julian Dolby¹, Orna Grumberg², Peter Müller³, and Omer Tripp⁴

- 1 IBM TJ Watson Research Center – Yorktown Heights, US, dolby@us.ibm.com
- 2 Technion – Haifa, IL, orna@cs.technion.ac.il
- 3 ETH Zürich, CH, peter.mueller@inf.ethz.ch
- 4 Google Inc. – Mountain View, US, trippo@google.com

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 16201 “Synergies among Testing, Verification, and Repair for Concurrent Programs”. This seminar builds upon, and is inspired by, several past seminars on program testing, verification, repair and combinations thereof. These include Dagstuhl Seminar 13021 “Symbolic Methods in Testing”; Dagstuhl Seminar 13061 “Fault Prediction, Localization and Repair”; Dagstuhl Seminar 14171 “Evaluating Software Verification Systems: Benchmarks and Competitions”; Dagstuhl Seminar 14352 “Next Generation Static Software Analysis Tools”; Dagstuhl Seminar 14442 “Symbolic Execution and Constraint Solving”; and Dagstuhl Seminar 15191 “Compositional Verification Methods for Next-Generation Concurrency”. These were held in January 2013; February 2013; April 2014; August 2014; October 2014; and May 2015, respectively. Two notable contributions of Dagstuhl Seminar 16201, which distinguish it from these past seminars, are (i) the focus on concurrent programming, which introduces significant challenges to testing, verification and repair tools, as well as (ii) the goal of identifying and exploiting synergies between the testing, verification and repair research communities in light of common needs and goals.

Seminar May 16–20, 2016 – <http://www.dagstuhl.de/16201>

1998 ACM Subject Classification B.8.1 Reliability, Testing, and Fault-Tolerance, D.1.3 [Concurrent Programming] Distributed Programming, Parallel Programming, D.2.4 [Software/Program Verification] Assertion Checkers, Correctness Proofs, Model Checking, D.4.1 [Process Management] Concurrency, I.2.2 [Automatic Programming] Automatic Analysis of Algorithms, Program Modification, Program Transformation, Program Verification

Keywords and phrases (automatic) bug repair, concurrency bugs, concurrent programming, deductive verification, interactive verification, linearizability, synchronization, testing

Digital Object Identifier 10.4230/DagRep.6.5.56



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

Synergies among Testing, Verification, and Repair for Concurrent Programs, *Dagstuhl Reports*, Vol. 6, Issue 5, pp. 56–71

Editors: Julian Dolby, Orna Grumberg, Peter Müller, and Omer Tripp



DAGSTUHL
REPORTS Dagstuhl Reports

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

1 Executive Summary

Omer Tripp

Julian Dolby

Orna Grumberg

Peter Müller

License © Creative Commons BY 3.0 Unported license
© Omer Tripp, Julian Dolby, Orna Grumberg, and Peter Müller

Context and Motivations

Major trends in computing infrastructure, such as multicore processors and data centers, increase the demand for concurrent software that utilizes the available resources. However, concurrent programs are notoriously difficult to develop. They are susceptible to a number of specific errors that do not occur in sequential code, such as data races, deadlock, atomicity violations, starvation, and violations of consistency models. These errors typically manifest themselves only in certain executions (for instance, under certain thread schedules), which makes them extremely difficult to detect, reproduce, localize, and repair. Established techniques for testing, verifying and repairing sequential programs are insufficient to handle concurrent software. In particular, they do not address the following challenges:

- *State space explosion*: The execution of a concurrent program depends not only on the inputs but also on the thread schedule and optimizations, such as memory reordering. This results in a state space that is orders of magnitude larger than for sequential programs. Bug-finding techniques, such as testing and bounded model checking, require effective ways of pruning the state space. Static verification techniques, such as deductive verification and abstract interpretation, require suitable abstractions that allow one to reason about all possible program behaviors. Finally, program repair requires techniques to predict the impact of a program change on the set of possible executions.
- *Modularity*: Modular techniques, such as unit testing or compositional verification, scale to large applications. However, for many properties of concurrent programs there are no modular techniques, or they require a large annotation overhead, for instance to denote the locations protected by a lock or to specify a locking order (or discipline) that ensures deadlock freedom. It is crucial to develop techniques that allow programs to be checked and repaired modularly, for instance to fix an atomicity violation by adding more thread synchronization, but without introducing a deadlock globally.
- *Specifications*: Testing, verification and repair may rely on specifications that express the intended program behavior, for instance in the form of test oracles or program invariants. In addition to functional properties, specifications for concurrent programs also have to express how threads cooperate, for instance via a global locking strategy. While various specification approaches exist for concurrent programs, there is no uniform formalism that handles the full range of concurrency idioms and that supports testing, verification and repair.
- *Error reporting*: Testing, verification and repair techniques need to disambiguate true problems from spurious defects, which is often difficult in concurrent programs. For instance, a data race is not necessarily a bug. If a race occurs within a lock-free data structure, then it may be admissible as part of some higher-level transactional behavior enforced by the data-structure operation. Moreover, it is important to present bugs in an understandable manner, for instance by providing reports with only a small number of threads and by determining whether a bug is inherently concurrent or may also arise in a sequential context.

- *Liveness*: Whereas for most sequential programs, termination is the only relevant liveness property, liveness (such as fairness or the absence of livelocks) is often more prevalent in concurrent programs. It is, therefore, important to develop techniques to check and enforce progress.

Program testing, verification, and repair each offer partial solutions to these challenges. This seminar was conceived with the goal of bringing together these three communities in order to develop a common understanding of the issues as well as to enable collaboration at the level of techniques and tools.

Main Themes

The first step toward exposing, and enabling, synergies between the three main threads of research on correctness and reliability of concurrent programs – verification, testing and repair – is to analyze the challenges and contributions pertaining to each of these areas in isolation. We survey work that has been done in each of these communities, based on the available literature and presentations given in the seminar, to summarize the current state of the three communities.

Verification

A main challenge in verification of concurrency properties is the prohibitive state space unfolded by thread interleavings. A hybrid solution to this problem is to specialize the static abstraction according to necessary proof conditions, arising during dynamic runs, such that the verification algorithm can scale with fine-grained abstractions (Naik, Yang). Another approach is to retain correlations among local thread states as well as the shared program state (Sagiv, Segalov). In this way, useful invariants can be proved and exploited by the verifier even if an unbounded number of threads is assumed. Refinement techniques are useful when little information is required about the environment to prove a property (Gupta). A useful idea in error reporting is to pinpoint concurrency-specific bugs (differentiating them from sequential bugs) by also running a sequential verifier and performing delta analysis (Joshi). Much like other techniques, verification greatly benefits from user specifications. For example, a parallelizing compiler is more likely to prove disjointness between loop iterations if relevant data structures (or operations) are specified as linearizable (Rinard, Diniz). This also provides a measure of modularity, enabling the separation between library linearizability checking and client verification. Modern program logics (O’Hearn, Parkinson, Gardner) provide a way of constructing correctness proofs for concurrent programs, though in general modular verification of concurrent software remains a hard problem.

Testing

Similarly to verification, testing techniques are also challenged by the state-space problem. Several ideas have been proposed in response to this problem. Open-world testing, whereby data structures or libraries referenced by an application are tested in isolation for concurrency bugs (e.g., atomicity violations), reduces the scope of testing considerably (Shacham). Interestingly, even open-world issues that cannot be recreated within the client application are often fixed by developers, which encourages further research into modular consistency properties (e.g., linearizability) (Shacham). Predictive analysis is a recent form of testing that holds the promise of high coverage at an affordable cost (Smaragdakis). Starting from a

concrete trace, predictive analysis applies feasibility-preserving transformations (reordering trace events, typically through constraint solving) to detect concurrency bugs, such that soundness is guaranteed (Dolby, Huang). Another source of state-space reduction is to exploit high-level semantic guarantees, like atomicity, to abstract away intermediate trace transitions (Shacham, Tripp). This also relates to error reporting, where certain read/write conflicts give rise to spurious conflicts that can be eliminated with a higher-level view of conflict as lack of commutativity between atomic operations (Koskinen, Kulkarni). Contrary to memory-level conflict detection, commutativity-based testing requires a specification (Shacham, Tripp). Another form of specification refers to consistency relaxations, e.g. permitting certain types of read/write conflict (Thies) or specifying a computation as nondeterministic (Burnim, Tripp).

Repair

In program repair, error reporting (or localization) plays a key role, deciding the effective scope and nature of the fix. Pinpointing the exact conditions that give rise to a concurrency bug is thus critical, emphasizing the need for better testing and verification tools. Importantly, incorrect fixing may introduce concurrency bugs (e.g., a deadlock resulting from additional synchronization to fix an atomicity violation), which again highlights the need for better synergy between repair and testing/verification (Liu). Incorrect fixing also turns liveness into a concrete concern: Assuming the program previously terminated, does it also terminate after the fix? Existing solutions that ensure termination rely on iterative transformation methods as well as specialized models like Petri nets (Liu, Zhang). A common assumption in the repair community, to hold back the state-space challenge, is that concurrency bugs involve a small number of threads (typically 2) (Liblit, Liu). The hope is that better synergy with testing and verification can work toward relaxing this assumption. Semantic lifting of the concrete code, exploiting e.g. linearizability, has recently been demonstrated as a useful means to apply bottom-up/top-down fixing: First, the code is lifted into an abstract workflow, and then the workflow is concretized into a correct reimplementation (Liu, Tripp). This motivates further exploration of useful specification media for repair of concurrency defects.

Goals of the Seminar

The goal of the seminar was to promote cross fertilization among the verification, testing and repair communities, as they seem to be running into the same challenges, thereby solving increasingly similar problems. At the extreme, verification is about all possible program behaviors, testing is about running the program to see what it does, and repair is about generating new code. However, many techniques in all communities now blur the distinction. Use of dynamic information to guide abstractions in verification is one example; another is how predictive testing looks for bugs in possible executions close to a dynamic one, leading to a form of verification; finally, program repair increasingly uses solvers to synthesize new programs and test them, which overlaps with techniques from the other areas. We intended for the seminar to bring out further areas in which these fields are closely related, and inspire further techniques that fuse these areas, which was fulfilled by some of the discussions throughout the seminar.

Below are concrete examples of connections that we meant to expose, some of which were discussed throughout the seminar:

Benchmarks

Each area has a variety of benchmarks and competitions, and many of them ultimately focus on concurrency-specific challenges like interleavings. It seems likely that the different communities could benefit from sharing. For instance, predictive testing and verification could surely share many benchmarks, and a more standard set of benchmarks could make evaluations easier. At the same time, potential users could help ensure that any benchmarks actually measure what they care about.

Infrastructure

Much progress in both testing and verification has been made possible by progress in solver technology, and a variety of solvers are now common in both areas. There is room to share the infrastructure itself and the common remaining challenges.

Hybrid tools

The path-specific focus of testing and the global focus of verification can aid each other, e.g. current work such as CLAP using a control flow from a specific execution to make model checking more scalable.

Though the seminar touched on techniques and approaches that generalize beyond analysis and repair of concurrent software, we feel that the overall focus on challenges posed by concurrency was justified. With this focus, we were able to stir concrete discussion and tightly connected talks.

2 Table of Contents

Executive Summary	
<i>Omer Tripp, Julian Dolby, Orna Grumberg, and Peter Müller</i>	57
Organization of the Seminar	62
Overview of Talks	63
Predicate abstraction for bounded and unbounded concurrency <i>Alastair Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl</i>	63
Starling: simpler concurrency proofs <i>Mike Dodds, Matthew J. Parkinson, and Matthew Windsor</i>	64
Automated Program Bug Repair <i>Orna Grumberg</i>	64
Tutorial: Automated Repair of Concurrency Bugs <i>Ben Liblit</i>	65
Making the Java Memory Model Safe <i>Andreas Lochbihler</i>	65
Stateless Model Checking of Event-Driven Applications <i>Anders Møller</i>	66
Partial Verification Results <i>Peter Müller</i>	66
ISSTAC: Integrated Symbolic Execution for Space-Time Analysis of Code (Side-Channel Analysis) <i>Corina Pasareanu</i>	67
Reasoning about non-linearizable concurrent objects <i>Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and German Andres Delbianco</i>	68
Actor Services: Modular Verification of Message Passing Programs <i>Alexander J. Summers</i>	68
Tutorial: Deductive Verification Tools <i>Alexander J. Summers</i>	69
Fixing Linearizability Violations in Map-based Concurrent Operations Automatically <i>Omer Tripp</i>	70
Bringing Abstract Interpretation to Termination and Beyond <i>Caterina Urban</i>	70
Participants	71

3 Organization of the Seminar

The seminar lasted four days. We launched it with an introduction session featuring 2-minute lightning talks by each of the participants. These brief talks provided background on the person and relevant research experience and expertise. This enabled interaction and discussion from the very onset. The remainder of the seminar consisted of three types of talks: tutorials, demos and presentations.

Throughout the seminar, there were four tutorials. The first two were given on the first day of the seminar. The remaining two were given the next day. The goal of the tutorials was to expose the different communities to one another in a thorough and explicit manner (modulo time limitations). For this reason, the tutorials were given during the first part of the seminar. The tutorials were on the following topics:

- Model checking techniques for concurrent software
- Deductive verification of concurrent programs
- Repair of concurrency bugs
- Testing of, and test generation for, concurrent software

The tutorials provided general background, specific techniques as well as discussion of challenges and future research directions.

On the last day of the seminar, there was a session dedicated to demos, where three live demos were given. These showed use of both research and commercial tools. The topics were as follows:

- Directed model checking of JavaScript code (with asynchronous event handlers)
- Test generation for concurrent libraries
- Modular and interactive verification of concurrent programs

Finally, there were six technical sessions. Within each session, we intentionally combined talks on testing, verification and repair to expose synergies and encourage discussion. Topics that were covered include the following:

- Stateless model checking of event-driven applications
- Interpolation in model checking
- Predicate abstraction for bounded and unbounded concurrency
- Automated bug repair
- Making use of partial verification results
- Interactive verification of concurrent software using Dafny
- Reasoning about non-linearizable concurrent objects
- Algorithmic logic-based verification
- The Rely/Guarantee framework for verifying concurrent programs
- Proving termination via abstract interpretation
- Pervasive verification of multi-core systems
- The Java memory model
- Repairing linearizability violations in map-based operations
- Modular verification of message-passing programs
- Verification of event-driven JavaScript programs
- Concurrent specification of POSIX
- Using symbolic execution for space-time analysis of code

As the different types of talks and many topics that were covered illustrate, the challenges that were addressed in this seminar are complex and demand discussion and collaboration across the testing, verification and repair communities. There are natural links that, to date, have not been exploited sufficiently. Immediate examples include verification of programs

after a concurrency bug has been automatically patched, guiding testing by the results of partial verification (which is what commercial tools typically support), and integrating verification with bounded model checking. These and various other points of synergy were explored and discussed throughout the seminar both during sessions and in ad-hoc meetings and forums (which is a great advantage of meeting at Dagstuhl).

As the reader can learn from the rest of this report, the seminar has achieved the goals of promoting discussion across the different communities, exposing common challenges and approaches to address these challenges, as well as shaping “hybrid” research directions that take their inspiration from the problems faced by both the testing and the verification and the repair communities. Participants provided highly positive feedback following the seminar, and expressed interest in follow-up events. The organizers are also very supportive of more seminars in the spirit of this seminar, which aim to explore and exploit links and synergies between the communities. There is room for more focused discussion on specific topics that were touched upon during the seminar. There should hopefully also be an opportunity in the future to reflect on, and shape, research directions that were borne out of this seminar or match its spirit.

4 Overview of Talks

4.1 Predicate abstraction for bounded and unbounded concurrency

Alastair Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl

License © Creative Commons BY 3.0 Unported license

© Alastair Donaldson, Alexander Kaiser, Daniel Kroening, Michael Tautschnig, and Thomas Wahl

Main reference A. Kaiser, D. Kroening, T. Wahl, “Lost in Abstraction: Monotonicity in Multi-threaded Programs”, *Information and Computation*, 2016; pre-print available from author’s webpage.

URL <http://dx.doi.org/10.1016/j.ic.2016.03.003>

URL <http://www.ccs.neu.edu/home/wahl/Publications/kkw16.pdf>

Predicate abstraction, a technique for overapproximating system-level software by programs over Boolean-valued variables, has proved to be a success story in sequential program verification, especially for control-intensive programs. In recent years there has been some effort to extend this technique to multi-threaded programs. In this talk I discuss the requirements for shared-variable concurrent predicate abstraction, both for the case of a known number of threads, as well as for the unbounded case, where the number of threads is initially unknown or may dynamically change at runtime.


The result is that, while concurrent predicate abstraction reduces data complexity substantially (as in the sequential case), we have to pay for that by an increase in concurrency control complexity, even in the bounded-thread case.

This work as published primarily in the following two papers (journal versions of preceding conference papers):

- Alexander Kaiser and Daniel Kroening and Thomas Wahl. Lost in Abstraction: Monotonicity in Multi-threaded Programs. *Information and Computation*, 2016.
- Alastair Donaldson and Alexander Kaiser and Daniel Kroening and Michael Tautschnig and Thomas Wahl. Counterexample-guided abstraction refinement for symmetric concurrent programs. *Formal Methods in System Design*, 2012.

4.2 Starling: simpler concurrency proofs

Mike Dodds (University of York, GB), Matthew J. Parkinson, and Matthew Windsor

License  Creative Commons BY 3.0 Unported license
© Mike Dodds, Matthew J. Parkinson, and Matthew Windsor


Modern program logics have made it feasible to reason about the most complex kinds of concurrent algorithm. However, many modern logics are enormously complex and difficult to understand, and most logics lack any kind of automated tool support.

We propose an antidote in Starling, a prototype tool for automated verification of concurrent algorithms. Starling takes a proof outline written in an intuitive predicate-based style, and converts it into proof obligations that can be discharged by Z3 or a Horn clause solver. Starling's underlying approach is based on the Views framework, which means it can be applied to many kinds of reasoning system. Starling can automatically verify several challenging examples including the Linux ticketed lock.

Starling is in active development on github: <http://github.com/septract/starling-tool>.

4.3 Automated Program Bug Repair

Orna Grumberg (Technion – Haifa, IL)

License  Creative Commons BY 3.0 Unported license
© Orna Grumberg
Joint work of Bat-chen Rothenberg, Orna Grumberg

This is a work in progress.

The work presents a novel approach for automatically repairing a program with respect to a given set of assertions. Programs are repaired using a predefined set of mutations. We impose no assumptions on the number of erroneous locations in the program. We refer to a bounded notion of correctness.

The repaired programs are returned one by one, in increasing number of mutations. Only minimal sets of mutations are applied. That is, if a program can be repaired by applying a set of mutations *Mut*, then no superset of *Mut* is later considered. This is based on the understanding that the programmer would like to get a repaired program which is as close to the original program as possible.

Our approach is based on formal methods. In particular, we exploit both SMT and SAT solvers, both incrementally. The SMT solver verifies whether a mutated program is indeed correct. The SAT solver restricts the search space of mutated programs to only those obtained by a minimal mutation set. Thus, an efficient search of all minimal repaired program is achieved.

We implemented a prototype of our algorithm and got very encouraging results.

4.4 Tutorial: Automated Repair of Concurrency Bugs

Ben Liblit (University of Wisconsin – Madison, US)

License © Creative Commons BY 3.0 Unported license
© Ben Liblit

Joint work of Guoliang Jin, Shan Lu, Ben Liblit

Main reference G. Jin, W. Zhang, D. Deng, B. Liblit, S. Lu, “Automated Concurrency-Bug Fixing”, in Proc. of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI’12), pp. 221–236, USENIX Association, 2012.

URL <https://www.usenix.org/conference/osdi12/technical-sessions/presentation/jin>

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

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

References

- 1 Dongdong Deng, Guoliang Jin, Marc de Kruijf, Ang Li, Ben Liblit, Shan Lu, Shanxiang Qi, Jinglei Ren, Karthikeyan Sankaralingam, Linhai Song, Yongwei Wu, Mingxing Zhang, Wei Zhang, and Weimin Zheng, “Fixing, Preventing, and Recovering From Concurrency Bugs.” In Science China Information Sciences, volume 58, number 5, May 2015. Invited paper.
- 2 Guoliang Jin, Wei Zhang, Dongdong Deng, Ben Liblit, and Shan Lu, “Automated Concurrency-Bug Fixing.” In Tenth USENIX Symposium on Operating Systems Design and Implementation (OSDI 2012), October 2012.
- 3 Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit, “Automated Atomicity-Violation Fixing.” In Proceedings of the ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation (PLDI 2011), June 2011.

4.5 Making the Java Memory Model Safe

Andreas Lochbihler (ETH Zürich, CH)

License © Creative Commons BY 3.0 Unported license
© Andreas Lochbihler

Main reference A. Lochbihler, “Making the Java Memory Model Safe”, ACM Trans. Program. Lang. Syst., 35(4), Paper 12, 2014.

URL <http://dx.doi.org/10.1145/2518191>

Type safety and the Java security architecture distinguish the Java programming language from other mainstream programming languages like C and C++. Another important feature of Java is its built-in support for multithreading and the Java memory model. In this talk, I discuss how the current Java memory model affects type safety and Java’s security guarantees.

The findings are based on a formal model of Java and the Java memory model. It includes dynamic memory allocation, thread spawns and joins, infinite executions, the wait-notify mechanism, and thread interruption, all of which interact in subtle ways with the memory model. The language is type safe and provides the data race freedom guarantee. The model and proofs have been checked mechanically in the proof assistant Isabelle/HOL.

References

- 1 Andreas Lochbihler. *Making the Java Memory Model Safe*. ACM Trans. Program. Lang. Syst. 35(4):12, 2014

4.6 Stateless Model Checking of Event-Driven Applications

Anders Møller (Aarhus University, DK)

License © Creative Commons BY 3.0 Unported license
© Anders Møller

Main reference C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, M. Vechev, “Stateless model checking of event-driven applications”, in Proc. of the 2015 ACM SIGPLAN Int’l Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015), pp. 57-73, ACM, 2015.

URL <http://dx.doi.org/10.1145/2814270.2814282>

Modern event-driven applications, such as, web pages and mobile apps, rely on asynchrony to ensure smooth end-user experience. Unfortunately, even though these applications are executed by a single event-loop thread, they can still exhibit nondeterministic behaviors depending on the execution order of interfering asynchronous events. As in classic shared-memory concurrency, this nondeterminism makes it challenging to discover errors that manifest only in specific schedules of events. In this work we propose the first stateless model checker for event-driven applications, called R4. Our algorithm systematically explores the nondeterminism in the application and concisely exposes its overall effect, which is useful for bug discovery. The algorithm builds on a combination of three key insights: (i) a dynamic partial order reduction (DPOR) technique for reducing the search space, tailored to the domain of event-driven applications, (ii) conflict-reversal bounding based on a hypothesis that most errors occur with a small number of event reorderings, and (iii) approximate replay of event sequences, which is critical for separating harmless from harmful nondeterminism. We instantiate R4 for the domain of client-side web applications and use it to analyze event interference in a number of real-world programs. The experimental results indicate that the precision and overall exploration capabilities of our system significantly exceed that of existing techniques.

4.7 Partial Verification Results

Peter Müller (ETH Zürich, CH)

License © Creative Commons BY 3.0 Unported license
© Peter Müller

Most techniques to detect program errors, such static program analysis, do not fully verify all possible executions of a program. They leave executions unverified when they do not check certain properties, fail to verify properties, or check properties under certain unsound assumptions such as the absence of arithmetic overflow.

In this talk, we present a technique to complement partial verification results by automatic test case generation. We annotate programs to reflect which executions have been verified, and under which assumptions. These annotations are then used to guide dynamic symbolic execution toward unverified program executions. We have implemented our technique for the .NET static analyzer Clousot and the dynamic symbolic execution tool Pex. It produces smaller test suites (by up to 19.2%), covers more unverified executions (by up to 7.1%), and reduces testing time (by up to 52.4%) compared to combining Clousot and Pex without our technique.

4.8 ISSTAC: Integrated Symbolic Execution for Space-Time Analysis of Code (Side-Channel Analysis)

Corina Pasareanu (NASA – Moffett Field, US)

License © Creative Commons BY 3.0 Unported license
© Corina Pasareanu

Attacks relying on the inherent space-time complexity of algorithms implemented by software systems are gaining prominence. Software systems are vulnerable to such attacks if an adversary can inexpensively generate inputs that cause the system to consume an impractically large amount of time or space to process those inputs, thus denying service to benign users or otherwise disabling the system. The adversary can also use the same inputs to mount side-channel attacks that aim to infer some secret from the observed space-time system behavior.

Our project, ISSTAC: Integrated Symbolic Execution for Space-Time Analysis of Code, aims to develop automated analysis techniques and implement them in an industrial-strength tool that allows the efficient analysis of software (in the form of Java bytecode) with respect to space-time complexity vulnerabilities. The analysis is based on symbolic execution, a well-known analysis technique that systematically explores program execution paths and also generates inputs that trigger those paths. We are building a cloud-based symbolic execution engine for Java that includes new and improved algorithms for the symbolic space-time complexity and side-channel analysis of programs and a novel model counting constraint solver needed for quantifying the analysis results.

This is a 4-year collaborative project between Vanderbilt University, CMU, UC Santa Barbara and Queen Mary University, London. The project will build upon existing and mature symbolic execution tools (Symbolic PathFinder). I will give an overview of the project and highlight recent advancements on side-channel analysis. The ISSTAC website is: <https://www.cmu.edu/silicon-valley/research/isstac/index.html>.

4.9 Reasoning about non-linearizable concurrent objects

Ilya Sergey (University College London, GB), Aleksandar Nanevski, Anindya Banerjee, and German Andres Delbianco

License © Creative Commons BY 3.0 Unported license

© Ilya Sergey, Aleksandar Nanevski, Anindya Banerjee, and German Andres Delbianco

Main reference I. Sergey, A. Nanevski, A. Banerjee, G. Andrés Delbianco, “Hoare-style Specifications as Correctness Conditions for Non-linearizable Concurrent Objects”, arXiv:1509.06220v3 [cs.LO], 2015.

URL <http://arxiv.org/abs/1509.06220v3>

Designing scalable concurrent objects, which can be efficiently used on multicore processors, often requires one to abandon standard specification techniques, such as linearizability, in favor of more relaxed consistency requirements. However, the variety of alternative correctness conditions makes it difficult to choose which one to employ in a particular case, and to compose them when using objects whose behaviors are specified via different criteria. The lack of syntactic verification methods for most of these criteria poses challenges in their systematic adoption and application.

In this line of work, we argue for using Hoare-style program logics as an alternative and uniform approach for specification and compositional formal verification of safety properties for concurrent objects and their client programs. Through a series of case studies, we demonstrate how an existing program logic for concurrency can be employed off-the-shelf to capture important state and history invariants, allowing one to explicitly quantify over interference of environment threads and provide intuitive and expressive Hoare-style specifications for several non-linearizable concurrent objects that were previously specified only via dedicated correctness criteria. We illustrate the adequacy of our specifications by verifying a number of concurrent client scenarios, that make use of the previously specified concurrent objects, capturing the essence of such correctness conditions as concurrency-aware linearizability, quiescent, and quantitative quiescent consistency.

4.10 Actor Services: Modular Verification of Message Passing Programs

Alexander J. Summers (ETH Zürich, CH)

License © Creative Commons BY 3.0 Unported license

© Alexander J. Summers

Joint work of Müller, Peter

Main reference A. J. Summers, P. Müller, “Actor Services: Modular Verification of Message Passing Programs”, in Proc. of the 25th European Symposium on Programming (ESOP’16), LNCS, Vol. 9632, pp. 699–726, Springer, 2016.

URL http://dx.doi.org/10.1007/978-3-662-49498-1_27

We present actor services [1]: a novel program logic for defining and verifying response and functional properties of programs which communicate via asynchronous messaging. Actor services can specify how parts of a program respond to messages, both in terms of guaranteed future messages, and relations between the program states in which messages are received and responses sent. These specifications can be composed, so that end-to-end behaviours of parts of a system can be summarised and reasoned about modularly. We provide inference rules for guaranteeing these properties about future execution states without introducing explicit traces or temporal logics.

Actor services are ultimately derived from local actor services, which express behaviours of single message handlers. We provide a proof system for verifying local services against an implementation, using a novel notion of obligations to encode the appropriate liveness requirements. Our proof technique ensures that, under weak assumptions about the underlying system (messages may be reordered, but are never lost), as well as termination of individual message handlers, actor services will guarantee suitable liveness properties about a program, which can be augmented by rich functional properties. Our approach supports reasoning about both state kept local to an actor (as in a pure actor model), and shared state passed between actors, using a flexible combination of permissions, immutability and two-state invariants.

References

- 1 Alexander J. Summers and Peter Müller. *Actor Services: Modular Verification of Message Passing Programs*. European Symposium on Programming (ESOP) 2016, Springer-Verlag, LNCS.

4.11 Tutorial: Deductive Verification Tools

Alexander J. Summers (ETH Zürich, CH)

License © Creative Commons BY 3.0 Unported license
© Alexander J. Summers

Joint work of Müller, Peter; Schwerhoff, Malte

Main reference P. Müller, M. Schwerhoff, A. J. Summers, “Viper: A Verification Infrastructure for Permission-Based Reasoning”, in Proc. of the 17th Int’l Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI’16), LNCS, Vol. 9583, pp. 41–62, Springer, 2016.

URL http://dx.doi.org/10.1007/978-3-662-49122-5_2

This tutorial summarises the state of the art in automated deductive verification tools – those which take a program along with specifications/annotations and attempt to prove that the program satisfied its specification, reporting potential violations. An overview of a number of these tools is given, including a summary of the different applications and features of these tools, and the Chalice verifier for race-free concurrent programs (<http://chalice.codeplex.com>) is shown for a concrete demonstration.

The main two technical approaches for building such verification tools are symbolic execution (defining a custom verification engine) and verification condition generation (embedding verification problems in a lower-level verification language). Both techniques are briefly explained, and the idea of intermediate verification languages is motivated. The Viper Project (<http://viper.ethz.ch>) [1] is presented, which is a new such intermediate language and suite of generic verification tools, designed to easily support encodings of modern program logics and other reasoning methodologies. The tutorial concludes by giving demonstrations of how both traditional permission-based reasoning and recent techniques such as those addresses weak memory reasoning can be simply implemented via encodings into Viper, exploiting the reusable verifiers provided.

References

- 1 Peter Müller and Malte Schwerhoff and Alexander J. Summers. *Viper: A Verification Infrastructure for Permission-Based Reasoning*. Verification, Model Checking, and Abstract Interpretation (VMCAI) 2016, Springer-Verlag, LNCS,

4.12 Fixing Linearizability Violations in Map-based Concurrent Operations Automatically

Omer Tripp (IBM TJ Watson Research Center – Yorktown Heights, US)

License © Creative Commons BY 3.0 Unported license
© Omer Tripp

Joint work of Peng Liu, Omer Tripp, Xiangyu Zhang

Main reference P. Liu, O. Tripp, X. Zhang, “Flint: fixing linearizability violations”, in Proc. of the 2014 ACM Int’l Conf. on Object Oriented Programming Systems Languages & Applications (OOPSLA’14), pp. 543–560, ACM, 2014.

URL <http://dx.doi.org/10.1145/2660193.2660217>

Writing concurrent software while achieving both correctness and efficiency is a grand challenge. To facilitate this task, concurrent data structures have been introduced into the standard library of popular languages like Java and C#. Unfortunately, while the operations exposed by concurrent data structures are atomic (or linearizable), compositions of these operations are not necessarily atomic. Recent studies have found many erroneous implementations of composed concurrent operations.

In this talk, I address the problem of fixing nonlinearizable composed operations such that they behave atomically. Specifically, I will present an automated fixing algorithm for composed Map operations and its implementation as the Flint tool. Flint accepts as input a composed operation suffering from atomicity violations. Its output, if fixing succeeds, is a composed operation that behaves equivalently to the original operation in sequential runs and is guaranteed to be atomic.

Flint takes a first step towards fixing incorrect concurrent compositions fully automatically, encouraging more research effort in this direction. Evaluation of Flint on 48 incorrect compositions from 27 popular applications, including Tomcat and MyFaces, has yielded highly encouraging: Flint is able to correct 96% of the methods, and the fixed version is often the same as the fix by an expert programmer and as efficient as the original code.

4.13 Bringing Abstract Interpretation to Termination and Beyond

Caterina Urban (ETH Zürich, CH)

License © Creative Commons BY 3.0 Unported license
© Caterina Urban

Joint work of Miné, Antoine

Main reference C. Urban, A. Miné, “Proving Guarantee and Recurrence Temporal Properties by Abstract Interpretation”, in Proc. of the 16th Int’l Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI), LNCS, Vol. 8931, pp. 190–208, Springer, 2015.

URL http://dx.doi.org/10.1007/978-3-662-46081-8_11

Program termination is the most prominent liveness property. We design new program approximations, in order to automatically infer sufficient preconditions for program termination and synthesize piecewise-defined ranking functions, which provide upper bounds on the waiting time before termination. We also contribute an abstract interpretation framework for proving liveness properties, which comes as a generalization of the framework proposed for termination. In particular, the framework is dedicated to liveness properties expressed in temporal logic, which are used to ensure that some desirable event happens once or infinitely many times during program execution. The results presented in this talk have been implemented into a prototype analyzer. Experimental results show that it performs well on a wide variety of benchmarks and it is competitive with the state of the art.

Participants

- Mike Dodds
University of York, GB
- Julian Dolby
IBM TJ Watson Research Center
– Yorktown Heights, US
- Derek Dreyer
MPI-SWS – Saarbrücken, DE
- Philippa Gardner
Imperial College London, GB
- Orna Grumberg
Technion – Haifa, IL
- Arie Gurfinkel
Carnegie Mellon University –
Pittsburgh, US
- Cliff B. Jones
University of Newcastle, GB
- K. Rustan M. Leino
Microsoft Corporation –
Redmond, US
- Ben Liblit
University of Wisconsin –
Madison, US
- Andreas Lochbihler
ETH Zürich, CH
- Peter Müller
ETH Zürich, CH
- Anders Møller
Aarhus University, DK
- Wytse Oortwijn
University of Twente, NL
- Corina Pasareanu
NASA – Moffett Field, US
- Wolfgang J. Paul
Universität des Saarlandes, DE
- Arnd Poetzsch-Heffter
TU Kaiserslautern, DE
- Murali Krishna Ramanathan
Indian Institute of Science –
Bangalore, IN
- Malavika Samak
Indian Institute of Science –
Bangalore, IN
- Ilya Sergey
University College London, GB
- Natasha Sharygina
University of Lugano, CH
- Sharon Shoham Buchbinder
Tel Aviv University, IL
- Alexander J. Summers
ETH Zürich, CH
- Michael Tautschnig
Queen Mary University of
London, GB
- Omer Tripp
IBM TJ Watson Research Center
– Yorktown Heights, US
- Caterina Urban
ETH Zürich, CH
- Yakir Vazel
Princeton University, US
- Thomas Wahl
Northeastern University –
Boston, US

