Report from Dagstuhl Seminar 14171

Evaluating Software Verification Systems: Benchmarks and Competitions

Edited by

Dirk Beyer[1], Marieke Huisman[2], Vladimir Klebanov[3], and Rosemary Monahan[4]

1 **University of Passau, DE**
2 **University of Twente, NL**
3 **Karlsruhe Institute of Technology, DE**
4 **National University of Ireland, IE**

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 14171 "Evaluating Software Verification Systems: Benchmarks and Competitions". The seminar brought together a large group of current and future competition organizers and participants, benchmark maintainers, as well as practitioners and researchers interested in the topic. The seminar was conducted as a highly interactive event, with a wide spectrum of contributions from participants, including talks, tutorials, posters, tool demstrations, hands-on sessions, and a live competition.

Seminar April 21–25, 2014 – http://www.dagstuhl.de/14171
1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs
Keywords and phrases Formal Verification, Deductive Verification, Automatic Verification, Theorem Proving, Model Checking, Program Analysis, Competition, Comparative Evaluation
Digital Object Identifier 10.4230/DagRep.4.4.1

1 Executive Summary

Dirk Beyer
Marieke Huisman
Vladimir Klebanov
Rosemary Monahan

The seminar aimed to advance comparative empirical evaluation of software verification systems by bringing together current and future competition organizers and participants, benchmark maintainers, as well as practitioners and researchers interested in the topic.

The objectives of the seminar were to (1) advance the technical state of comparative empirical evaluation of verification tools, (2) achieve cross-fertilization between verification communities on common/related issues such as selection of relevant problems, notions of correctness, questions of programming language semantics, etc., (3) explore common evaluation of different kinds of verification tools, its appropriate scope and techniques, (4) raise mutual awareness between verification communities concerning terminology, techniques and trends, and (5) promote comparative empirical evaluation in the larger formal methods community.

Evaluating Software Verification Systems: Benchmarks and Competitions, *Dagstuhl Reports*, Vol. 4, Issue 4, pp. 1–19
Editors: Dirk Beyer, Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan

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

Altogether, 43 researchers and practitioners have attended the seminar. A vast majority of the attendees (almost 90%) have participated either in the SV-COMP or in the VerifyThis (and related, e. g., VSComp/VSTTE) verification competitions. For lack of better terms, we tend to refer to these communities as the "automatic verification" and "deductive verification" community respectively, though, as Section 1 discusses in more detail, these labels are based on pragmatics and history rather than on technical aspects.

The presentations, hands-on sessions, and discussions provided valuable feedback that will help competition organizers improve future installments. To continue the effort, a task force will compile a map of the—in the meantime very diverse—competition landscape to identify and promote useful evaluation techniques. It was agreed that evaluation involving both automatic and deductive verification tools would be beneficial to both communities as it would demonstrate the strengths and weaknesses of each approach. Both SV-COMP and VerifyThis will be associated with the ETAPS conference in 2015.

A call to the public: It has been reported that competition-verification challenges have been used as homework for students. The seminar organisers would appreciate feedback and experience reports from such exercises.

Seminar Structure

The seminar was structured as a highly interactive event, rather than a sequence of talks, compared to workshops or conferences. The seminar opened with a session of *lightning talks* that gave every participant two minutes to introduce themselves and mark their activities and interests in verification and in comparative evaluation in particular.

In order to give participants insight into different verification techniques and practical capabilities of some existing verification tools, the seminar featured short overviews of the state of the art in deductive verification resp. automatic verification, as well as several tutorials, hands-on sessions, and accompanying discussions. These included a longer tutorial on deductive verification with the Dafny system together with a hands-on session, as well as short mini-tutorials on the automatic verifiers CPAchecker and CBMC, and deductive verifiers KIV and VeriFast. Another hands-on session concluded the seminar.

Discussions on evaluation techniques and setups were initiated with presentations by competition organizers and benchmark collectors. The presented evaluation vehicles included: VerifyThis competition (deductive verification), SV-COMP (automatic verification), VSTTE competition (deductive verification), SMT-COMP (Satisfiability Modulo Theories), Run-time Verification competition, RERS challenge (Rigorous Examination of Reactive Systems), INTS benchmarks (Integer Numerical Transition Systems).

Since evaluation must be grounded with the requirements of current and prospective users of verification technology, the seminar incorporated contributions from industrial participants. Among them were a talk on the use of the SPARK deductive verification tool-set as a central tool for the development of high-integrity systems at Altran UK, a talk on the use of automatic verification tools in the Linux Driver Verification project, an accompanying discussion, as well as statements by other industry representatives (from GrammaTech, Galois, LLNL, and Microsoft Research).

Verification Communities: Remarks on Commonalities, Differences, and Terminology

Important goals of the seminar were to raise mutual awareness and to foster cross-fertilization between verification communities. We may habitually refer to communities as "automatic verification" or "deductive verification", but as time passes, these labels become less adequate.

A trend is apparent that different types of tools are slowly converging, both technically and pragmatically. Instances of both automatic and deductive verifiers may use symbolic execution or SMT solvers. Automatic verifiers can synthesize (potentially quantified) invariants, verify infinite-state systems, or systems that are heap-centric.

The pace of development is high and the surveys are costly (the last comprehensive survey on automatic verification appeared in 2008). As a consequence, community outsiders typically have an outdated—sometimes by decades—view on verification technology that does not reflect the state of the art. We expect publications from competitions to fill the void between more comprehensive surveys.

One of the terminological pitfalls concerns the use of the attribute "automatic". For instance, model checking and data-flow analysis are typically advertised as "automatic". This is indeed true in the sense that the model-checking user does not have to supply proof hints such as loop invariants to the tool. On the other hand, using a model checker in practical situations may as well require user interaction for the purpose of creating test drivers, choosing parameters, tuning abstractions, or interpreting error paths (which can be quite complex). These aspects are typically abstracted away during evaluation of automatic verifiers, which allows better standardization but does not capture all aspects that are relevant in practice.

The situation is further confused by the fact that some deductive verifiers are also advertised as "automatic", even though all established deductive verification systems require user interaction and the amount of interaction that is needed with different tools is not radically different. The main meaningful differences are rather

1. whether user interaction happens only at the beginning of a single proof attempt or whether the user can/has to intervene during proof construction, and
2. whether user interaction happens in a purely textual manner or whether non-textual interaction is possible/required.

The seminar has confirmed the need for improved terminology, as well as made an attempt to eliminate misconceptions and communication pitfalls. Unfortunately, there is still no widely-accepted and usable terminology to communicate these distinctions.

2 Table of Contents

3 Tutorials and Hands-on Sessions

3.1 Introduction to Deductive Software Verification

Rosemary Monahan (NUI Maynooth, IE)

Deductive software verification is a program verification technique where a program and its specification are input into a tool which verifies that the given program meets its specification. A human user contributes in two ways: through formalizing an informally stated specification for a program and through providing guidance to a verification system to show formally the conformance of the program to its specification. The specification is the form of a contract, typically containing preconditions, which state the conditions under which the program should be executed; post-conditions, which will be achieved by executing the program; and frame conditions which specify what the program execution is permitted to modify. Deductive verification tools generate and prove the verification conditions necessary to verify that the given program meets its specification. The tool user often contributes to this process providing proof tips that direct the verifier, as well as lemmas and assertions that can assist the proof. This interaction is required due to the strong properties being verified and the range of different tools in the verification landscape. An overview of how these verification tools and proofs are performed in a range of state-of-the-art tools is given with reference to examples from many tools that took part in the VerifyThis 2012 verification competition at FM 2012.

3.2 Introduction to Automatic Software Verification

Dirk Beyer (University of Passau, DE)

Automatic program verification is a verification technique that takes as input a program and a (temporal) property and without user interaction constructs a witness for correctness (program invariants, proof certificate) or violation (counterexample). The talk gives an overview over techniques that are implemented in software verifiers that participate in SV-COMP. Those techniques include abstraction refinement, predicate abstraction, CEGAR, interpolation, shape analysis, bounded model checking, large-block encoding, and conditional model checking.

3.3 Dafny Crash Course and Hands-on Session

Rustan Leino (Microsoft Research, US) and Rosemary Monahan (NUI Maynooth, IE)

URL http://research.microsoft.com/en-us/projects/dafny/

In this specialized mini-tutorial, features of Dafny, the programming language and interactive program verifier, are presented. Material is tailored to a program verification exercise

announced later in the afternoon, allowing participants to use what they learned in the Dafny crash course. Tutorial materials and the coincidence count challenges are available on the web.[1] 33 participants took part in the tutorial and hands-on session.

3.4 VeriFast: A Mini-tutorial

Bart Jacobs (KU Leuven, BE)

URL http://people.cs.kuleuven.be/~bart.jacobs/verifast/

During this mini-tutorial I introduced the seminar members to the VeriFast program verification tool that I have been developing with my group since 2008. It is a tool for sound modular automatic verification of safety and correctness properties of annotated single-threaded and multi-threaded C and Java programs. It requires that each function/method be annotated with a pre-condition and post-condition expressed in a variant of separation logic. It verifies each function/method separately using symbolic execution.

I started the tutorial by interactively verifying, in the VeriFast IDE, a simple example C program, illustrating modular symbolic execution, the symbolic store, and the symbolic path condition. Then I moved to an example involving dynamic memory allocation, illustrating the symbolic heap, and the production and consumption of heap chunks. Then, through the example of a modular specification and verification of the functional behavior of a stack data structure, I showed how recursive data structures can be specified and verified, through the use of inductive separation logic predicates and inductive data types. To conclude, I briefly discussed my VeriFast solution to the coincidence count challenge.

3.5 KIV: A Mini-tutorial

Gidon Ernst and Gerhard Schellhorn (University of Augsburg, DE)

URL http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/

The tutorial demonstrated the interactive theorem prover KIV to the seminar participants. It highlighted some of its core features:

- Sequent calculus for Higher Order Logic.
- An automatic simplification strategy based on user-defined conditional rewrite rules.
- Structured algebraic data types with lots of predefined ones in KIV's library.
- A weakest precondition calculus for imperative programs over these abstract data types. Programs can be verified by symbolic execution and induction.
- An elaborate graphical interface that graphically displays specification hierarchies ("development graphs"). Proof trees of sequent calculus, that can be inspected and replayed.
- A correctness management that takes care to invalidate only a minimal set of theorems after changes.
- Context-sensitive application of rewrite rules by just clicking on function symbols.

[1] http://www.rise4fun.com/dafny/eOCY
http://www.rise4fun.com/dafny/wJHw
http://www.rise4fun.com/dafny/hcvan

The tutorial then demonstrated how the coincidence count example introduced in the Dafny tutorial is proved in KIV. A second example was the deletion of the minimal element of a binary search tree from the VerifyThis competition at FM 2012, using KIV's separation library and a direct induction which avoids the use of complex invariants.

3.6 Bounded Software Model Checking using CBMC: A Mini-tutorial

Michael Tautschnig (Queen Mary University of London, UK)

URL http://www.cprover.org/cbmc/

CBMC implements bit-precise bounded model checking for C programs and has been developed and maintained for more than ten years. Only recently support for efficiently checking concurrent programs, including support for weak memory models, has been added. CBMC verifies the absence of violated assertions under a given loop unwinding bound by reducing the problem to a Boolean formula. The formula is passed to a SAT solver, which returns a model if and only if the property is violated. In the tutorial I will provide an overview of the key components of CBMC, underlining its straightforward pipeline. Then a number of examples will be presented, including floating point and concurrent programs, as well as a full SAT solver (PicoSAT).

3.7 Predicate Abstraction with CPAchecker: A Mini-tutorial

Philipp Wendler (University of Passau, DE)

URL http://cpachecker.sosy-lab.org/

Predicate abstraction is a traditional abstract domain used for model checking. In this talk we learn how it works in combination with CEGAR on a small C program. We then see how the verification framework CPAchecker analyzes the same program with predicate abstraction, and study the produced proof. An overview of CPAchecker and a tutorial on its usage conclude the talk.

3.8 Concluding Hands-on Session

The seminar concluded with another hands-on session. A verification challenge based on a real bug encountered in the Linux kernel source was chosen by the seminar organizers (description below). Participants were encouraged to build teams of up to three people, in particular mixing attendees from different communities. Some teams have applied several different tools to the problem. The challenge was as follows (abridged description):

The accompanying C file implements a doubly-linked list with integer payloads. The central function of the program sorts the list using a bubble-sort-style algorithm in ascending order of payloads. The data structure also supports nesting, but this is not used for sorting. Separation between nesting pointers and list-linkage pointers should be maintained though.
The program contains a basic correctness checker consisting of assert statements ('inspect' function). The checker does not check the complete specification given above.
Verify the program w.r.t. the given assertions. Should you find bugs, please fix them and proceed to verify. If your tool does not support C, we ask you to reimplement the core data structure/functionality in the language of your choice. Please try to stay as faitful to the original code as possible.

- For verifiers that do not need user-supplied invariants, checking that the assertions pass would be a starting point. Feel free to add assertions that are meaningful w.r.t. the informal specification. If you can't verify all of the included assertions, please produce a maximal set of assertions that you can verify.
- For verifiers that use more expressive specification formalisms and user-supplied invariants, we encourage you to prove a more complete functional specification rather than just checking the assertions.

After the allocated two hours elapsed, we have received eleven submissions. The program indeed contained a bug resulting from a typo in the list initialization code.[2] The applied automatic verifiers could detect the assertion violation easily though interpreting the error path, but locating the bug required considerable effort in some cases. Unsurprisingly, proving the program correct after fixing the bug was not easy for the automatic verifiers. If at all, correctness could be shown only within a small bounded scope, with one exception. The program analyzer Predator, which is geared towards verifying heap-manipulating programs, could synthesize the invariant that is necessary to show that the assertions are never violated on lists of arbitrary length.

With deductive verifiers, the situation was more varied. Several teams succeeded in verifying parts of the code respective to a subset of assertions. Success factors were support for verifying C programs (as otherwise time was lost translating the subject program into the language supported by the verifier) and having found the bug first either by testing or automatic verification as auxiliary technique. A followup on the challenge is planned. An interesting question is whether and how the automatically synthesized safety invariant can be used in a deductive verifier.

4 Poster Presentations and Tool Demonstrations

The following posters and tool demonstrations have been presented at the seminar.

Posters:

- Annotations for All, David Cok
- OpenJML, David Cok

[2] The challenge is part of the SV-COMP database and can be found at https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/heap-manipulation/bubble_sort_linux_false-unreach-call.c.

- AutoProof, an auto-active verifier for Eiffel, Nadia Polikarpova
- Ultimate Automizer, Matthias Heizmann
- Cryptol: The Language of Cryptography, Joe Kiniry
- The Astrée Static Analyser, Antoine Miné
- KeY System 2.0, Mattias Ulbrich, Vladimir Klebanov
- IVIL, Mattias Ulbrich
- CPAchecker: The Configurable Software Verification Platform, Phillip Wendler
- SPARKSkein, Angela Wallenburg
- VERCORS, Marieke Huisman

Tool demonstrations:

- UFO, Aws Albarghouthi
- SPEEDY, David Cok
- OpenJML, David Cok
- Why3, Andrei Paskevich and Jean-Christophe Filliâtre
- AutoProof, Nadia Polikarpova
- Frankenbit, Arie Gurfinkel
- Ultimate Automizer, Matthias Heizmann
- Cryptol, Joe Kiniry
- Astrée, Antoine Miné
- SIL, Peter Müller
- GROOVE, Arend Rensink
- HSE, Andrey Rybalchenko
- KIV, Gerhard Schellhorn
- CodeThorn, Markus Schordan
- LLBMC, Carsten Sinz
- CBMC, Michael Tautschnig
- KeY/IVIL, Mattias Ulbrich
- CPAchecker, Phillip Wendler
- GRASShopper, Thomas Wies

5 Discussion on Comparative Evaluation

5.1 Deductive Verification

The following is an incomplete summary of remarks voiced during the discussion on evaluation issues by the deductive verification community. As in all good discussions a range of, sometimes conflicting, views was presented. Nineteen participants of the seminar took part in the discussion.

5.1.1 Sourcing Competition Challenges

The issue of sourcing good challenges for deductive verification tools was discussed at length. It was agreed that a call for challenges could be issued well in advance of the competition with contributions from industrial users particularly welcome. Those who submit challenges should submit a solution and scoring scheme. If they participate in the competition, they would be eliminated from that competition challenge. The ACM programming and similar competitions were also mentioned as a potential source for challenges.

5.1.2 Range of Competition Challenges

The following suggestions were made with respect to competition challenges:

- A selection of challenges could be offered with teams expected to complete a subset of challenges (e. g., offer five challenges; teams submit their best three).
- Challenges should focus on program verification rather than theorem proving. Emphasis should be put on programming language constructs such as iterators, threads, reflection, concurrency, complicated data structures, and interaction between the user and the program.
- Challenges that we cannot complete but that can be mapped to simpler problems (e. g., by mapping to the integer domain) are motivating.
- Challenge areas could be announced in advance of the competition to encourage tool developers to improve tools prior to the competition.
- On-site competitions need to be selective about the problems so that a small number of challenges with a focus on modular specification are presented.
- Challenges should progress from previous competitions and from challenge to challenge within the competition.
- Good challenges will motivate tools to evolve in different directions, with the more interesting challenges presenting a large gap between their specification and their implementation.
- Providing challenges in pseudo-code is favored over program languages, because the input to verification tools differs significantly. Time to implement the solution in the language of the tool must therefore be allocated.
- Challenges with errors should be included so that error detection as well as verification of program properties are part of the problem set.

5.1.3 Types of Competition

It was agreed that short on-site competitions, longer off-site competitions and benchmark collections (such as VACID-0) all have a role in motivating tool development and comparison. Challenges need to be self-contained, involving a maximum of two days work (to be realistic regarding the time constraints of those who will participate).

5.1.4 Relevance to Industry

Challenges should focus tool developers on building industrial-strength tools, allowing users to use features of real-world programming languages, rather than forcing users to encode problems in the way that the tool requires.

5.1.5 Solution Strategies and Variations

Grading of challenges could take solution strategies into account, with extra points for using different approaches to the problem, e. g., a maximum of X points for an iterative solution and a maximum of Y points for a recursive solution.

Another suggestion was to provide challenges with a (rough) solution. This would evaluate how easy it is to encode the solution in a particular tool and reduce influence of the user on the result. Of course, different tools might require very different solutions.

5.1.6 Motivating Competition Participation

To date, post-competition publications have been an important motivating factor for competition participation, yet the sustainability of this approach is unclear. Tool developers also benefit through comparison of tools and through advancing tools to meet the verification challenges. Other motivators to develop further include:

- Cash prizes obtained via sponsorship, which would motivate student teams.
- Prizes sponsored by professional societies, e. g., an ACM software verification prize.
- Different competition tracks, e. g., undergraduate, postgraduate, developers, users.
- Opportunities for discussion of tools and solutions, as well as exchange of tool development ideas.
- Bringing student supervisors on board so that they allow student time for competitions, integrate the challenges and competitions into case studies, and develop course exercises based on competition challenges.
- Guest presentations and registration at conferences.

5.1.7 Use of Libraries

The use of libraries in competition should be encouraged. The largest challenge here is the lack of specified libraries that are available for use. Some verification challenges could include verification of library code.

5.1.8 Tool Integration

The integration of tools and techniques is necessary to allow scalability in program verification. A combination of automatic and deductive verification techniques should be explored. A good starting point for case studies would be to take solutions from deductive verification competition challenges, inject bugs and ask automatic verifiers to locate the bugs. The main drawback at present seems to be the focus of automatic verifiers on C programs as input and their restricted use of theories. Competitions that require two tools per team would provide for tool integration, e. g., automatic invariant generation could assist many deductive verifiers at present.

5.2 Automatic Verification

The following is an incomplete summary of remarks voiced during the discussion on evaluation issues by the automatic verification community. Fourteen participants of the seminar took part in the discussion.

5.2.1 How to Grow the Benchmark Set of Verification Tasks

The discussion included some concrete plans for obtaining more benchmark verification tasks for programs involving arrays, programs involving floating-point arithmetic, generated random loop benchmarks, generated random concurrent benchmarks, and more verification tasks from Linux systems code.

5.2.2 New Category on Bug Finding

The participants at the SV-COMP community meeting have agreed on introducing a demonstration category on bug finding. The goal in this category is to find as many bugs as possible within a global time limit of 4 h. More precisely, a participating verifier is given the full set of verification tasks of SV-COMP 2015 and a computing resource for 4 h (CPU time), and the verifier reports all bugs that it can find. The ranking is based on the number of correctly identified bugs. The bugs are accompanied by a witness path that is checked, and the bug report is counted if the witness is successfully validated.

5.2.3 Directory of Verifiers

A proposal was made to create an "electronic book of verification tools". A common pattern for the description of the tools is applied, such that users/readers quickly get an overview over technologies, features, and applicability of the verification tool to certain use cases.

5.2.4 Verification Results with Confidence Levels?

A discussion regarding the evaluation of challenge solutions centered around confidence levels. Is a solution's confidence level simply true or false, or is there a more fine-grained level, such as a confidence level from 0 to 9? It was agreed that numeric confidence levels are helpful only for a fixed set of examples. Confidence may be applicable more to humans rather than to verification tools. How should results be compared if confidence levels are used? If used in practice of competitions, both a numeric scheme and a non-numeric scheme is needed. Every tool should then include a description of what is achieved, feedback to the user, perhaps counterexamples. However, this would make the competition rules quite complicated, and simple rules are important to not confuse readers/users while interpreting the results. Therefore, it was voiced that the rules should be kept as simple as possible. No matter how complicated and detailed an evaluation would be made—there are always different criteria for evaluation by different audiences—winning a competition does not automatically mean that the winner is the best tool (from the consumers' point of view).

6 Competition Overviews

6.1 The VerifyThis Verification Competition

Vladimir Klebanov (Karlsruhe Institute of Technology, DE)

Joint work of Huisman, Marieke; Klebanov, Vladimir; Monahan, Rosemary
Main reference M. Huisman, V. Klebanov, R. Monahan, "On the Organization of Program Verification Competitions," in Proc. of the 1st Int'l Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE'12), CEUR-WS, Vol. 873, pp. 50–59, 2012.
URL http://ceur-ws.org/Vol-873/papers/paper_2.pdf

We discuss the challenges that have to be addressed when organizing program verification competitions. Our focus is on competitions for verification systems where the participants both formalize an informally stated requirement and (typically) provide some guidance for the tool to show it. We draw insights from our experiences with organizing VerifyThis, a program verification competition aiming (1) to bring together those interested in formal verification, and to provide an engaging, hands-on, and fun opportunity for discussion; (2) to

evaluate the usability of logic-based program verification tools in a controlled experiment that could be easily repeated by others. We discuss in particular the following aspects: challenge selection, on-site versus online organization, team composition and judging.

6.2 SV-COMP: Competition on Software Verification

Dirk Beyer (University of Passau, DE)

Main reference D. Beyer, "Status Report on Software Verification," in Proc. of the 20th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14), LNCS, Vol. 8413, pp. 373–388, Springer, 2014.
URL http://dx.doi.org/10.1007/978-3-642-54862-8_25

SV-COMP is a thorough evaluation of verification tools that take as input software programs and run a fully automatic verification of a given property. The overview describes the definitions, rules, setup, and procedure of SV-COMP. The verification tasks of the competition consist of nine categories containing a total of 2 868 C programs, covering bit-vector operations, concurrent execution, control-flow and integer data-flow, device-drivers, heap data structures, memory manipulation via pointers, recursive functions, and sequentialized concurrency. The specifications include reachability of program labels and memory safety. The most recent (3rd) edition of the competition had 15 participants. http://sv-comp.sosy-lab.org/

6.3 RERS Challenge and Property-Driven Benchmark Generation

Bernhard Steffen (TU Dortmund, DE)

Joint work of Steffen, Bernhard; Isberner, Malte; Naujokat, Stefan; Margaria, Tiziana; Geske, Maren
Main reference B. Steffen, M. Isberner, S. Naujokat, T. Margaria, M. Geske, "Property-Driven Benchmark Generation," in Proc. of the 20th Int'l Symp. on Model Checking Software (SPIN'13), LNCS, Vol. 7976, pp. 341–357, Springer, 2013.
URL http://dx.doi.org/10.1007/978-3-642-39176-7_21

We present a systematic approach to the automatic generation of platform-independent benchmarks of realistic structure and tailored complexity for evaluating verification tools for reactive systems. The idea is to mimic a systematic constraint-driven software development process by automatically transforming randomly generated temporal-logic-based requirement specifications on the basis of a sequence of property-preserving, randomly generated structural design decisions into executable source code of a chosen target language or platform. Our automated transformation process steps through dedicated representations in terms of Büchi automata, Mealy machines, decision diagram models, and code models. It comprises LTL synthesis, model checking, property-oriented expansion, path condition extraction, theorem proving, SAT solving, and code motion. This setup allows us to address different communities via a growing set of programming languages, tailored sets of programming constructs, different notions of observation, and the full variety of LTL properties—ranging from mere reachability over general safety properties to arbitrary liveness properties. The paper illustrates the corresponding tool chain along accompanying examples, emphasizes the current state of development, and sketches the envisioned potential and impact of our approach.

6.4 The 2nd Verified Software Competition

Jean-Christophe Filliâtre (University Paris South, FR)

Joint work of Filliâtre, Jean-Christophe; Paskevich, Andrei; Stump, Aaron
Main reference J.-C. Filliâtre, A. Paskevich, A. Stump, "The 2nd Verified Software Competition: Experience Report," in Proc. of the 1st Int'l Workshop on Comparative Empirical Evaluation of Reasoning Systems (COMPARE'12), CEUR-WS, Vol. 873, pp. 36–49, 2012.
URL http://ceur-ws.org/Vol-873/papers/paper_6.pdf

We report on the second verified software competition. It was organized by the J.-C. Filliâtre, A. Paskevich, and A. Stump on a 48 hours period on November 8–10, 2011. We describe the competition, present the five problems that were proposed to the participants, and give an overview of the solutions sent by the 29 teams that entered the competition.

6.5 First International Competition of Software for Runtime Verification

Ezio Bartocci (TU Wien, AT)

Joint work of Bartocci, Ezio; Bonakdarpour, Borzoo; Falcone, Yliès

We report the description of the procedures, the participating teams, the submitted benchmarks, the evaluation process and the results of the 1st International Competition of Software for Run-time Verification. This competition is held as a satellite event of the 14th International Conference on Run-time Verification (RV) and is organized in three main tracks: offline monitoring, online monitoring of C programs and online monitoring of Java programs.

6.6 SMT-COMP: The SMT Competition

Alberto Griggio (Bruno Kessler Foundation – Trento, IT)

Joint work of Cok, David R.; Griggio, Alberto; Bruttomesso, Roberto; Deters, Morgan
Main reference D. R. Cok, A. Griggio, R. Bruttomesso, M. Deters, "The 2012 SMT Competition," in Proc. of the 10th Int'l Workshop on Satisfiability Modulo Theories (SMT'12), EPiC Series, Vol. 20, pp. 131–142, EasyChair, 2012.
URL http://www.easychair.org/publications/?page=527924520

The talk presents the lessons learned from participating in the annual SMT solvers competition SMT-COMP, both as a competitor and as an organizer. I describe the organization of the competition, highlight its positive impacts on the community, but also discuss some of its current limitations, concluding with some suggestions for possible future improvements.

6.7 Numerical Transition Systems

Philipp Rümmer (Uppsala University, SE)

Joint work of Hojjat, Hossein; Konecný, Filip; Garnier, Florent; Iosif, Radu; Kuncak, Viktor; Rümmer, Philipp
Main reference H. Hojjat, F. Konecný, F. Garnier, R. Iosif, V. Kuncak, P. Rümmer, "A Verification Toolkit for Numerical Transition Systems," in Proc. of the 18th Int'l Symp. on Formal Methods (FM'12), LNCS, Vol. 7436, pp. 247–251, Springer, 2012.
URL http://dx.doi.org/10.1007/978-3-642-32759-9_21

Verification systems accept software programs or hardware designs as input in a wide variety of formats, some of which are intricate or partly underspecified. This leads to challenges for the development of verification systems, as well as the collection of verification benchmarks and competitions. This talk introduces a standardized format for verification problems, Numerical Transition Systems, which is applicable both for the representation of benchmarks and as a simple yet expressive intermediate verification language. Numerical Transition Systems support a variety of data types, including arithmetic and arrays, and provide features such as procedures and parallelism.

7 Industrial Applications

7.1 SPARK 2014 – Beyond Case Studies

Angela Wallenburg (Altran, UK)

Formal software verification has been successfully applied and demonstrated to scale to industrial projects. While many case studies have been successful, few formal methods have reached the take-up and maturity level where industrial non-experts continue to use the method for project after project, and where this formal method is a permanent part of the business of industrial software development. However, there are some notable exceptions: for example the SPARK language and tool-set for static verification has been applied for many years in on-board aircraft systems, control systems, cryptographic systems, and rail systems. SPARK has been developed and used at Altran UK (formerly Praxis) for almost 30 years. The grand challenge of building a verifying compiler for static formal verification of programs aims at bringing formal verification to non-expert users of powerful programming languages. This challenge has nurtured competition and collaboration among verification tool builders such as the participants in this Dagstuhl seminar. In this talk I describe our approach to popularizing formal verification in the design of the SPARK 2014 language and the associated formal verification tool GNATprove (co-developed by Altran UK and AdaCore). In particular, I will describe our solution to combining tests and proofs, which provides a cost-competitive way to develop software to standards such as DO-178. At the heart of our technique are executable contracts, and the ability to both test and prove those. I will also report on experiences in evaluation of verification tools from an industrial perspective.

7.2 The Experience of Linux Driver Verification

Vadim Mutilin ands Alexey Khoroshilov (Russian Academy of Sciences, Moscow, RU)

The talk presents experience of Linux Verification Center in verification of Linux Device Drivers. For that purpose we use software model checkers which solve reachability problems, i. e. they prove that labeled error locations in the program can not be reached by any execution starting from an entry point. But Linux device drivers have neither entry point no error location. That is why the driver needs to be prepared for the verification. The driver is essentially asynchronous. On driver loading, the kernel core invokes the initialization function of the driver which registers callback functions. Then these functions are called by the kernel core on receiving events from user space and from hardware. So for the verification we need to prepare the model environment with explicit calls of driver callbacks. The model environment should reproduce the same scenarios of interactions with the driver as in the real kernel and at the same time it should be simple enough to be analyzed by existing static verification tools. For that purpose we proposed a method of environment modeling based on pi-calculus where the environment is represented as a set of communicating processes interacting with driver via messages. From the pi model we generate a C program which being combined with the driver becomes valid input for software model checkers with an entry point emulating all feasible paths in driver's code. But this code still have no error locations representing erroneous behavior of the driver. So the second task is to prepare specification on driver-kernel interfaces to be checked by software model checkers. The specification is weaved into the driver source code with the help of C Instrumentation Framework (CIF) and it becomes a source of error locations.

In the talk we discussed the architecture of Linux Driver Verification (LDV) Tools and showed its analytical and error trace visualization features. The comparison of CPAchecker with BLAST on the Linux drivers showed that while the total number of found bugs is the same the tools find different bugs. Thus it is worth to use the tools together. For now, LDV Tools helped to find more than 150 bugs which were approved and fixed in the latest Linux kernels. In the talk we discussed lessons learned. We found that it is important that the software model checker supports a full set of language features and could parse it. Moreover, the tool should not fail if it does not support some feature. If it cannot prove correctness it still may, for example, continue the search for unsafe trace if possible. We found that for verification tools it is even more important to ignore thousands irrelevant transitions than efficiently handle relevant ones. Also, the engineering efforts can help to get significantly better results and we shared experience in speeding up BLAST from 8 times on small-sized drivers and to 30 times on medium-sized drivers. On the base of the lessons learnt we make several conclusions how to improve Software Verification Competition (SV-COMP). First of all, there are two different use cases for software model checkers. The first one targets to prove correctness of code under analysis regarding some properties, the second one targets to find as much bugs as possible. Currently, rules and scoring scheme of SV-COMP evaluate tools from the first point of view, while it would be useful to evaluate the tools from the second point of view as well. Another conclusion is that benchmarks we produced so far target current generation of software model checkers and these benchmarks can not help to evaluate next generation tools. But fore verification of device drivers we need much more features, for example, better pointer analysis support, specifications in terms of sets and maps, verification of parallel programs, data race detection, support for function pointers.

We should produce benchmarks which target new functionality required for device driver verification, not only existing one.

The effective analysis of error traces produced by software model checkers in case of specification violation is crucial for the industrial use. At least, it should be possible to analyze the errors traces without knowing implementation details of the software model checker. A common representation of error traces in the competition would ease usage of the tools for driver verification as far as a single trace converter could be developed for Error Trace Visualizer component inside LDV Tools which is used for trace analysis. Users waits for verification results in terms of wall time, not the CPU time. The use of CPU time for the competition does not encourage the developers to utilize the available resources for parallelization. For example, the tools may use the CPU cores available on the machine instead of using a single one, thus reducing the wall time. It would be good if competition rules would encourage to reduce wall time of verification as well.

Participants

- Aws Albarghouthi
University of Toronto, CA
- Ezio Bartocci
TU Wien, AT
- Bernhard Beckert
KIT – Karlsruher Institut für Technologie, DE
- Dirk Beyer
Universität Passau, DE
- Marc Brockschmidt
Microsoft Research UK – Cambridge, GB
- David Cok
GrammaTech Inc. – Ithaca, US
- Gidon Ernst
Universität Augsburg, DE
- Marie Farrell
NUI Maynooth, IE
- Jean-Christophe Filliâtre
University Paris South, FR
- Bernd Fischer
University of Stellenbosch, ZA
- Alberto Griggio
Bruno Kessler Foundation – Trento, IT
- Radu Grigore
University of Oxford, GB
- Arie Gurfinkel
Carnegie Mellon University, US
- Matthias Heizmann
Universität Freiburg, DE
- Marieke Huisman
University of Twente, NL
- Bart Jacobs
KU Leuven, BE
- Alexey Khoroshilov
Russian Academy of Sciences – Moscow, RU
- Joseph Roland Kiniry
Galois Inc. – Portland, US
- Vladimir Klebanov
KIT – Karlsruher Institut für Technologie, DE
- K. Rustan M. Leino
Microsoft Res. – Redmond, US
- Stefan Löwe
Universität Passau, DE
- Antoine Miné
ENS – Paris, FR
- Rosemary Monahan
NUI Maynooth, IE
- Wojciech Mostowski
University of Twente, NL
- Peter Müller
ETH Zürich, CH
- Petr Müller
Brno Univ. of Technology, CZ
- Vadim Mutilin
Russian Academy of Sciences – Moscow, RU
- Andrei Paskevich
University Paris South, FR
- Nadia Polikarpova
ETH Zürich, CH
- Arend Rensink
University of Twente, NL
- Philipp Rümmer
Uppsala University, SE
- Andrey Rybalchenko
Microsoft Research UK – Cambridge, GB
- Gerhard Schellhorn
Universität Augsburg, DE
- Markus Schordan
Lawrence Livermore National Laboratory, US
- Carsten Sinz
KIT – Karlsruher Institut für Technologie, DE
- Bernhard Steffen
TU Dortmund, DE
- Jan Strejcek
Masaryk University – Brno, CZ
- Michael Tautschnig
Queen Mary University of London, GB
- Mattias Ulbrich
KIT – Karlsruher Institut für Technologie, DE
- Jaco van de Pol
University of Twente, NL
- Angela Wallenburg
Altran UK – Bath, GB
- Philipp Wendler
Universität Passau, DE
- Thomas Wies
New York University, US