





Information Exchange in Software Verification

Dirk Beyer^{*1}, Marieke Huisman^{*2}, Jan Strejček^{*3}, and Heike Wehrheim^{*4}

1  LMU Munich, DE. dirk.beyer@sosy.ifi.lmu.de

2  University of Twente – Enschede, NL. m.huisman@utwente.nl

3  Masaryk University – Brno, CZ. strejcek@fi.muni.cz

4  Universität Oldenburg, DE. heike.wehrheim@uni-oldenburg.de

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 25172 *Information Exchange in Software Verification*. The term “software verification” refers to the procedure of deciding the correctness of software with respect to (user-supplied or predefined) specifications. In general, software verification is an undecidable problem. Despite this undecidability, software verification is a very active research field with contributions of researchers from several areas such as theorem proving, deductive verification, static analysis, and automatic verification. The analysis techniques developed in these subareas are often complementary with respect to the type of software and specifications they can efficiently handle.

The objective of this Dagstuhl Seminar was to bring together people working in these different subareas to discuss and advance ways of having tools and techniques cooperate on the task of software verification.

Seminar April 22–25, 2025 – <https://www.dagstuhl.de/25172>

2012 ACM Subject Classification Software and its engineering → Formal methods; Theory of computation → Logic and verification

Keywords and phrases Competitions and Benchmarks, Data-Flow Analysis, Deductive Verification, Formal Verification, Model Checking

Digital Object Identifier 10.4230/DagRep.15.4.92

Edited in cooperation with Zsófia Ádám and Paulína Ayaziová

1 Executive Summary

Dirk Beyer (LMU München, DE)

Marieke Huisman (University of Twente – Enschede, NL)

Jan Strejček (Masaryk University – Brno, CZ)

Heike Wehrheim (Universität Oldenburg, DE)

License  Creative Commons BY 4.0 International license

© Dirk Beyer, Marieke Huisman, Jan Strejček, and Heike Wehrheim

The term “software verification” refers to the procedure of deciding the correctness of software with respect to (user-supplied or predefined) specifications. In general, software verification is an undecidable problem. Despite this undecidability, software verification is a very active research field with contributions of researchers from several areas such as theorem proving, deductive verification, static analysis, and automatic verification. The analysis techniques developed in these subareas are often complementary with respect to the type of software and specifications they can efficiently handle.

* Editor / Organizer



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 4.0 International license

Information Exchange in Software Verification, *Dagstuhl Reports*, Vol. 15, Issue 4, pp. 92–111

Editors: Dirk Beyer, Marieke Huisman, Jan Strejček, Heike Wehrheim, Zsófia Ádám, and Paulína Ayaziová



Dagstuhl Reports

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

The objective of this Dagstuhl Seminar was to bring together people working in these different subareas to discuss and advance ways of having tools and techniques cooperate on the task of software verification. Specific focus was given to the exchange of information between tools. In the seminar, we spent one morning session on talks about existing formats for exchange and a second morning session with talks on existing cooperations. The afternoon sessions and the third morning was dedicated to breakout groups on specific topics. In particular, during breakout groups we discussed the following questions:

- Which kind of information can be exchanged?
- How can we make tools provide their (partial) results about program correctness to other tools?
- How can we make tools use (partial) results of other tools during their analysis?
- Which formats are adequate for information exchange?

As an outcome of the discussions, we agreed on new experiments for more advanced exchange of information between tools for deductive verification and software model checking. We have also outlined several new formats for information exchange, in particular an intermediate format for programs and a format for exchanging information about abstraction precision. Action items for further development towards these formats and in other directions have been formulated and agreed by participants.

2 Table of Contents

Executive Summary

Dirk Beyer, Marieke Huisman, Jan Strejček, and Heike Wehrheim 92

Overview of Talks

StaRVOOrS – The Force Awakens Again

Wolfgang Ahrendt 96

FM-Tools: Find, Use, and Conserve Tools for Formal Methods

Dirk Beyer 96

Contract-LIB: A Proposal for a Common Interchange Format for Software System Specification

Gidon Ernst, and Mattias Ulbrich 97

Proofs on Inductive Predicates in Why3

Jean-Christophe Filliâtre 97

K2 and VMT

Alberto Griggio 98

Deductive Validation: Witnesses from Automatic Verifiers for Deductive Verifiers

Matthias Heizmann 98

AutoSV-Annotator: Integrating Deductive and Automatic Software Verification

Marieke Huisman 99

Information Exchange with CoVeriTest and Predicate Maps

Marie-Christine Jakobs 99

SMT-LIB for Exchange of Information Between Verifiers

Martin Jonáš 100

Specification and Verification with Frama-C/MetAcsL

Nikolai Kosmatov 100

Towards Scalable and Distributed Software Verification

Thomas Lemberger 100

From Requirements to Heterogenous Verification: Where Is the Toolchain?

Rosemary Monahan 101

Cooperation via Splitting

Cedric Richter 103

MoXI: An Intermediate Language to Spur Reproducible and Comparable Model

Checking Research

Kristin Yvonne Rozier 103

ACSL: Behavioral Interface Specification Language for C Code

Julien Signoles 104

Correctness Verification Witnesses 2.0

Jan Strejček 104

Paradigm-Spanning Collaboration: The Karlsruhe Java Verification Suite

Mattias Ulbrich 105

Reducers to Aid Information Exchange	
<i>Heike Wehrheim</i>	105
Bridging Hardware and Software Formal Verification	
<i>Zsófia Ádám and Dirk Beyer</i>	106
Working Groups	
Breakout Group: The Goals of the Seminar from the Perspective of Software Model Checking Community	
<i>Jan Strejček</i>	106
Breakout Group: Exchange Between Deductive Verification	
<i>Marieke Huisman</i>	107
Breakout Group: Intermediate Language for Software Verification	
<i>Dirk Beyer</i>	107
Breakout Group: Decomposition of the Verification Task	
<i>Heike Wehrheim</i>	108
Breakout Group: Exchange of Partial Verification Results	
<i>Zsófia Ádám and Thomas Lemberger</i>	109
Breakout Group: Exchange Between Deductive Tools (DV) and Fully Automatic Software Verifiers (SV)	
<i>Marieke Huisman</i>	109
Breakout Group: Information Exchange Between Automatic SW Verifiers	
<i>Jan Strejček</i>	110
Participants	111

3 Overview of Talks

3.1 StaRVOOrS – The Force Awakens Again

Wolfgang Ahrendt (*Chalmers University of Technology – Göteborg, SE*)

License © Creative Commons BY 4.0 International license
© Wolfgang Ahrendt

Joint work of Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, Gerardo Schneider
Main reference Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace, Gerardo Schneider: “Verifying data- and control-oriented properties combining static and runtime verification: theory and tools”, *Formal Methods Syst. Des.*, Vol. 51(1), pp. 200–265, 2017.
URL <https://doi.org/10.1007/S10703-017-0274-Y>

Static verification techniques are used to analyse and prove properties about programs before they are executed. Many of these techniques work directly on the source code and are used to verify data-oriented properties over all possible executions. The analysis is necessarily an over-approximation as the real executions of the program are not available at analysis time. In contrast, runtime verification techniques have been extensively used for control-oriented properties, analysing the current execution path of the program in a fully automatic manner. In this talk, we present a novel approach in which data-oriented and control-oriented properties may be stated in a single formalism amenable to both static and dynamic verification techniques. The specification language we present to achieve this that of ppDATEs, which enhances the control-oriented property language of DATEs, with data-oriented pre/postconditions. For runtime verification of ppDATE specifications, the language is translated into a DATE. We give a formal semantics to ppDATEs, which we use to prove the correctness of our translation from ppDATEs to DATEs. We show how ppDATE specifications can be analysed using a combination of the deductive theorem prover KeY and the runtime verification tool LARVA. Verification is performed in two steps: KeY first partially proves the data-oriented part of the specification, simplifying the specification which is then passed on to LARVA to check at runtime for the remaining parts of the specification including the control-oriented aspects. We show the applicability of our approach on two case studies.

3.2 FM-Tools: Find, Use, and Conserve Tools for Formal Methods

Dirk Beyer (*LMU Munich, DE*)

License © Creative Commons BY 4.0 International license
© Dirk Beyer

Main reference Dirk Beyer: “FM-Tools: Find, Use, and Conserve Tools for Formal Methods”. 2024
URL https://www.sosy-lab.org/research/pub/2024-Podolski65.Find_Use_and_Conserve_Tools_for_Formal_Methods.pdf

The research area of formal methods has made enormous progress in the last 20 years, and many tools exist to apply formal methods to practical problems. Unfortunately, many of these tools are difficult to find and install, and often they are not executable due to missing installation requirements. The findability and wide adoption of tools, and the reproducibility of research results, could be improved if all major tools for formal methods were conserved and documented in a central repository of tools for formal methods (cf. FAIR principles).

This paper describes a solution to this problem: Collect and maintain essential data about tools for formal methods in a central repository, called FM-Tools, which is available at <https://gitlab.com/sosy-lab/benchmarking/fm-tools>. The repository contains

metadata, such as which tools are available, which versions are advertised for each tool, and what command-line arguments to use for default usage. The actual tool executables are stored in tool archives at Zenodo, and for technically deep documentation, references point to archived publications or project web sites. Two communities, which are concerned with software verification and testing, already adopted the FM-Tools repository for their comparative evaluations.

3.3 Contract-LIB: A Proposal for a Common Interchange Format for Software System Specification

Gidon Ernst (LMU Munich, DE), Mattias Ulbrich (KIT – Karlsruher Institut für Technologie, DE)

License © Creative Commons BY 4.0 International license
 © Gidon Ernst, and Mattias Ulbrich
Joint work of Gidon Ernst, Wolfram Pfeifer, Mattias Ulbrich
Main reference Gidon Ernst, Wolfram Pfeifer, Mattias Ulbrich: “Contract-LIB: A Proposal for a Common Interchange Format for Software System Specification”, in Proc. of the Leveraging Applications of Formal Methods, Verification and Validation. Specification and Verification - 12th International Symposium, ISoLA 2024, Crete, Greece, October 27–31, 2024, Proceedings, Part III, Lecture Notes in Computer Science, Vol. 15221, pp. 79–105, Springer, 2024.
URL https://doi.org/10.1007/978-3-031-75380-0_6

Interoperability between deductive program verification tools is a well-recognized long-standing challenge. In this paper we propose a solution for a well-delineated aspect of this challenge, namely the exchange of abstract contracts for possibly stateful interfaces that represent modularity boundaries. Interoperability across tools, specification paradigms, and programming languages is achieved by focusing on abstract implementation-independent behavioral models. The approach, called Contract-LIB in reminiscence of the widely-successful SMT-LIB format, aims to standardize the language over which such contracts are formulated and provides clear guidance on its integration with established methods to connect high-level specifications with code-level data structures. We demonstrate the ideas with examples, define syntax and semantics, and discuss the rationale behind key design decisions.

3.4 Proofs on Inductive Predicates in Why3

Jean-Christophe Filliâtre (CNRS – Gif-sur-Yvette, FR)

License © Creative Commons BY 4.0 International license
 © Jean-Christophe Filliâtre
Joint work of Jean-Christophe Filliâtre, Andrei Paskevich, Henri Soudubray
URL <https://www.why3.org/>

This talk presents an extension of the Why3 tool to enable proofs by induction on instances of inductive predicates. It consists of a new pattern matching construction to analyze the form of such a derivation, on the one hand, and a new notion of variant to justify the termination of a recursive function that proceeds according to the size of a derivation, on the other hand. We show how this extension can be implemented conservatively, with almost no changes to Why3’s verification condition generator.

3.5 K2 and VMT

Alberto Griggio (Bruno Kessler Foundation – Trento, IT)

License © Creative Commons BY 4.0 International license
© Alberto Griggio

Joint work of Alberto Griggio, Alessandro Cimatti, Stefano Tonetta, Martin Jonáš
Main reference Alberto Griggio, Martin Jonáš: “Kratos2: An SMT-Based Model Checker for Imperative Programs”, in Proc. of the Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III, Lecture Notes in Computer Science, Vol. 13966, pp. 423–436, Springer, 2023.

URL https://doi.org/10.1007/978-3-031-37709-9_20

Main reference Alessandro Cimatti, Alberto Griggio, Stefano Tonetta: “The VMT-LIB Language and Tools”, in Proc. of the 20th Internal Workshop on Satisfiability Modulo Theories co-located with the 11th International Joint Conference on Automated Reasoning (IJCAR 2022) part of the 8th Federated Logic Conference (FLoC 2022), Haifa, Israel, August 11-12, 2022, CEUR Workshop Proceedings, Vol. 3185, pp. 80–89, CEUR-WS.org, 2022.

URL <https://ceur-ws.org/Vol-3185/extended9547.pdf>

In this talk we present two languages for the representation of formal verification problems of infinite-state systems, called VMT and K2. VMT stands for Verification Modulo Theories, and is an extension of SMT-LIB for the representation of symbolic transition systems and properties over them, expressed either as invariants (must hold for all reachable states) or as more general properties in Linear Temporal Logic (LTL). VMT was designed for ease of use and to facilitate interoperability among model checking tools operating over symbolic transition systems, possibly using background theories for representing infinite-state systems. More information is available at <https://vmt-lib.fbk.eu/>. Also the K2 language is conceptually an extension of SMT-LIB; unlike VMT, however, K2 allows to represent imperative programs with multiple procedures. The language, which is the input format of the Kratos2 symbolic model checker, has been used successfully in different research and technology transfer projects with industrial partners as a simple but powerful intermediate language for verification. More information is available at <https://kratos.fbk.eu/>.

3.6 Deductive Validation: Witnesses from Automatic Verifiers for Deductive Verifiers

Matthias Heizmann (Universität Stuttgart, DE)

License © Creative Commons BY 4.0 International license
© Matthias Heizmann

Joint work of Matthias Heizmann, Dominik Klumpp, Frank Schüssele, Marian Lingsch-Rosenfeld
Main reference Matthias Heizmann, Dominik Klumpp, Marian Lingsch Rosenfeld, Frank Schüssele: “Correctness Witnesses with Function Contracts”, CoRR, Vol. abs/2501.12313, 2025.

URL <https://doi.org/10.48550/ARXIV.2501.12313>

We discuss the possibility to use deductive verification tools for validating correctness witnesses. Perhaps, surprisingly, the annotations provided by automatic verifiers are often not sufficient for deductive verifiers. In this talk we identify five challenges: inductiveness, function contracts, expressiveness of expressions, gotos, and undefined behavior.

3.7 AutoSV-Annotator: Integrating Deductive and Automatic Software Verification

Marieke Huisman (University of Twente – Enschede, NL)

License © Creative Commons BY 4.0 International license

© Marieke Huisman

Joint work of Lukas Armbrorst, Dirk Beyer, Marieke Huisman, Marian Lingsch-Rosenfeld

Software model checking and deductive software verification have complementary strengths and weaknesses: software model checkers are more straight-forward to use, as they analyze the program without user input; but they do not yet support complicated data structures and expressive specifications. In contrast, deductive verifiers can verify expressive specifications and complex data structures modularly, but they require the user to specify the program behavior in detail, which is a time-consuming process. Due to their differing nature, the two approaches usually remain separate. However, for industrial usage, one requires both: ease of use as well as expressiveness. Therefore, we present AutoSV-Annotator, a toolchain that integrates the two approaches for C programs. The toolchain allows a user to iteratively refine the deductive annotations in a C program, calling a model checker to supplement the annotations at each iteration, guided by the already existing annotations. We show that our tool is able to annotate and prove many tasks from the SV-Benchmarks set. Our results show that the two strategies can indeed benefit from each other.

3.8 Information Exchange with CoVeriTest and Predicate Maps

Marie-Christine Jakobs (LMU Munich, DE)

License © Creative Commons BY 4.0 International license

© Marie-Christine Jakobs

Joint work of Dirk Beyer, Marie-Christine Jakobs


Main reference Marie-Christine Jakobs: “Reusing Predicate Precision in Value Analysis”, in Proc. of the Integrated Formal Methods - 17th International Conference, IFM 2022, Lugano, Switzerland, June 7-10, 2022, Proceedings, Lecture Notes in Computer Science, Vol. 13274, pp. 63–85, Springer, 2022.

URL https://doi.org/10.1007/978-3-031-07727-2_5

In this talk, I present two application areas for information exchange, test-case generation and cooperation in verification. I start to describe how I exchange information for test-case generation with CoVeriTest, a cooperative test-case generator based on the configurable program analysis framework CPAchecker. After a brief reminder on test-case generation with verifiers, I present the circular composition used by CPAchecker and give ideas on how information exchanged on test goals but also the explored state space. Thereafter, I continue discussing general application scenarios and concrete approaches for exchanging predicate maps, CPAchecker’s format to specify the abstraction level of CPAchecker’s predicate analysis, i.e., which predicates to use where. I give an overview of producer and consumers of predicate maps and provide an idea how to exchange information between predicate and value analysis. On the one hand, I sketch how to extract information from predicate maps for value analysis, in particular to derive the relevant variables to track. On the other hand, I indicate how to use information from the value analysis to generate predicate maps. I conclude the talk with the lessons learnt from the presented exchange approaches.

3.9 SMT-LIB for Exchange of Information Between Verifiers


Martin Jonáš (Masaryk University – Brno, CZ)

License  Creative Commons BY 4.0 International license
 © Martin Jonáš
URL <https://smt-lib.org/>

SMT-LIB is a widely-used format for specification of first-order formulas over various logical theories, used by a majority of current state-of-the-art SMT solvers. In this talk, I provide a high-level overview of the format with focus on the exchange of information between software verifiers. The talk also introduces several concepts, such as annotation mechanism, that are used by various software-verification languages based on SMT.

3.10 Specification and Verification with Frama-C/MetAcsl


Nikolai Kosmatov (Thales Research & Technology – Palaiseau, FR)

License  Creative Commons BY 4.0 International license
 © Nikolai Kosmatov
Joint work of Adel Djoudi, Martin Hana, Nikolai Kosmatov
Main reference Adel Djoudi, Martin Hana, Nikolai Kosmatov: “Formal Verification of a JavaCard Virtual Machine with Frama-C”, in Proc. of the Formal Methods - 24th International Symposium, FM 2021, Virtual Event, November 20-26, 2021, Proceedings, Lecture Notes in Computer Science, Vol. 13047, pp. 427–444, Springer, 2021.
URL https://doi.org/10.1007/978-3-030-90870-6_23

In the Frama-C ecosystem, tool collaboration is often based on ACSL annotations. Various kinds of more complex properties are translated into basic ACSL annotations that can then be verified using different tools. Examples of properties include relational properties, temporal logic properties, test objectives, etc. In this talk we show how global (high-level) properties are specified and verified using the MetAcsl plugin of Frama-C. Such properties include security properties (isolation, confidentiality, integrity) and are highly relevant for security certification. This approach was successfully used by Thales to formally verify its JavaCard Virtual Machine for the highest level of security certification (EAL7) of Common Criteria.

3.11 Towards Scalable and Distributed Software Verification

Thomas Lemberger (LMU Munich, DE)

License  Creative Commons BY 4.0 International license
 © Thomas Lemberger
Joint work of Dirk Beyer, Matthias Kettl, Thomas Lemberger
Main reference Dirk Beyer, Matthias Kettl, Thomas Lemberger: “Decomposing Software Verification using Distributed Summary Synthesis”, Proc. ACM Softw. Eng., Vol. 1(FSE), pp. 1307–1329, 2024.
URL <https://doi.org/10.1145/3660766>

There are many approaches for automated software verification, but they are either imprecise, do not scale well to large systems, or do not sufficiently leverage parallelization. We propose an approach to decompose one large verification task into multiple smaller, connected verification tasks, based on blocks in the program control flow. For each block, summaries (block contracts) are computed – based on independent, distributed, continuous refinement by communication between the blocks. The approach iteratively synthesizes preconditions to

assume at the block entry (computed from postconditions received from block predecessors, i.e., which program states reach this block) and violation conditions to check at the block exit (computed from violation conditions received from block successors, i.e., which program states lead to a specification violation). This separation of concerns leads to an architecture in which blocks can be analyzed in parallel, as separate verification problems. Whenever new information is available from other blocks, the verification can decide to restart with this new information. We realize our approach on the basis of configurable program analysis and implement it for the verification of C programs in the widely used verifier CPAchecker.

3.12 From Requirements to Heterogenous Verification: Where Is the Toolchain?

Rosemary Monahan (Maynooth University, IE)

License © Creative Commons BY 4.0 International license
© Rosemary Monahan

Joint work of Marie Farrell, Matt Luckcuck, Rosemary Monahan, Conor Reynolds, Oisín Sheridan

Main reference Marie Farrell, Matt Luckcuck, Rosemary Monahan, Conor Reynolds, Oisín Sheridan: “Adventures in FRET and Specification”, CoRR, Vol. abs/2503.24040, 2025.

URL <https://doi.org/10.48550/ARXIV.2503.24040>

Knowing which properties to specify and subsequently verify is one of the biggest bottlenecks in the use of Formal Methods [1]. Requirements are often written in natural-language, which can be ambiguous and is not generally amenable to formal verification. As a result, requirements elicitation and formalisation becomes even more important for understanding what to specify and verify.

Tools such as NASA’s Formal Requirements Elicitation Tool (FRET) have been developed to bridge the gap between natural-language requirements and the logics normally used for formal specification [2]. FRET provides a structured natural-language, called FRETISH, from which temporal logic specifications and other verification conditions can be derived. FRET has been used in many use cases to support both elicitation and formalisation of requirements. The case studies that we report on are an aircraft engine software controller, a mechanical lung ventilator, a rover carrying out an inspection task and an algorithm for autonomously grasping spent rocket stages. We discuss how the FRETISH requirements were elicited and subsequently used for specifying properties to be verified in various formal methods (model-checking, theorem proving and runtime verification approaches) encouraging a framework for interoperability.

The overarching aim of each of these case studies was different. Specifically, in the aircraft engine controller we focused our efforts on eliciting and formalising requirements in conversation with an industrial partner (Collins Aerospace) [3]. In the mechanical lung ventilator we formalised requirements that were supplied in the ABZ case study documentation with a view to developing a formal model of the system in Event-B [4]. For the inspection rover [5], the integration of various verification artefacts into an assurance case were explored, with a hazard analysis used to define the requirements that were formalised in FRET and subsequently verified against models of the system using CoCoSim and Event-B. In the autonomous grasping case study [6], the authors modelled the code of the system using Dafny, and generated a suite of runtime monitors using ROSMonitoring that were able to identify requirement violations at run time.

Tools like FRET are invaluable when communicating with engineers and developers both in academia and industry. This was evidenced by the aircraft engine controller and autonomous grasping case studies. In both, FRET’s diagrammatic and natural-language semantics helped case study providers to consider their system from different perspectives. This undoubtedly

provided a bridge for these developers, providing traceability from natural-language to formalised requirements and verification conditions. This approach to demonstrating formal methods in a user-friendly way that leverages existing requirements engineering approaches encouraged our collaborators to consider using similar formal tools in the future.

Throughout these case studies, we used FRET not only as an elicitation/formalisation tool but also as a way to manage the requirements that we were defining. We used the support that FRET provides for defining parent-child relationships between requirements, which allowed us to group related requirements together. Currently this parent-child link in FRET is informal in the sense that the user can assign a parent to a requirement without any formal/automatic checks that the requirements are related in some way. In future FRET developments it may be useful to formalise this relationship to allow the user to gradually refine requirements and provide a way to verify the traceability that we have documented between them.

In these case studies, some of the integration was via existing tool-supported translations (FRET to CoCoSpec) but the others were manual translations (FRET to Event-B, Dafny and ROSMonitoring). In the future, we hope that we can provide a more logically-founded and systematic approach to integrating these formal methods. Mathematical frameworks like UTP [7] or institution theory [8] may have roles to play here. However, less formal approaches will also be beneficial including the definition of workflows to combine and use various methods alongside one another.

This work was partially supported by the Royal Academy of Engineering and EPSRC grant EP/Y001532/1, as well as Maynooth University’s Hume Doctoral Award. We thank our co-authors and collaborators on each of the case studies: Stylianos Basagiannis, Georgios Giantamidis, Vassilios Tsachouridis, Hamza Bourbough, Anastasia Mavridou, Irfan Sljivo, Guillaume Brat, Louise Dennis, Michael Fisher, Nikos Mavrakis, Angelo Ferrando, Yang Gao and Clare Dixon.

References

- 1 Kristin Yvonne Rozier. *Specification: The biggest bottleneck in formal methods and autonomy*. In *Verified Software. Theories, Tools, and Experiments: 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17–18, 2016, Revised Selected Papers 8*, pages 8–26. Springer, 2016.
- 2 Dimitra Giannakopoulou, Thomas Pressburger, Anastasia Mavridou, and Johann Schumann. *Automated formalization of structured natural language requirements*. *Information and Software Technology*, 137:106590, 2021.
- 3 Marie Farrell, Matt Luckcuck, Oisín Sheridan, and Rosemary Monahan. *FRETting About Requirements: Formalised Requirements for an Aircraft Engine Controller*. In *Requirements Engineering: Foundation for Software Quality*, pages 96–111. Springer, 2022.
- 4 Marie Farrell, Matt Luckcuck, Rosemary Monahan, Conor Reynolds, and Oisín Sheridan. *Fretting and formal modelling: A mechanical lung ventilator*. In *International Conference on Rigorous State Based Methods*, 2024.
- 5 Hamza Bourbough, Marie Farrell, Anastasia Mavridou, Irfan Sljivo, Guillaume Brat, Louise A. Dennis, and Michael Fisher. *Integrating Formal Verification and Assurance: An Inspection Rover Case Study*. In *NASA Formal Methods*, pages 53–71. Springer, 2021.
- 6 Marie Farrell, Nikos Mavrakis, Angelo Ferrando, Clare Dixon, and Yang Gao. *Formal Modelling and Runtime Verification of Autonomous Grasping for Active Debris Removal*. *Frontiers in Robotics and AI*, 2022.
- 7 Charles Anthony Richard Hoare. *Unified theories of programming*. In *Mathematical methods in program development*, pages 313–367. Springer, 1997.
- 8 Joseph A. Goguen and Rod M. Burstall. *Institutions: Abstract model theory for specification and programming*. *Journal of the ACM*, 39(1):95–146, 1992.

3.13 Cooperation via Splitting

Cedric Richter (Carl von Ossietzky Universität Oldenburg, DE)

License © Creative Commons BY 4.0 International license
 © Cedric Richter
Joint work of Cedric Richter, Marek Chalupa, Marie-Christine Jakobs, Heike Wehrheim
Main reference Cedric Richter, Marek Chalupa, Marie-Christine Jakobs, Heike Wehrheim: “Cooperative Software Verification via Dynamic Program Splitting”, in Proc. of the 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, pp. 2087–2099, IEEE, 2025.
URL <https://doi.org/10.1109/ICSE55347.2025.00092>

The goal of cooperative software verification is to share the complexity of the verification task between verifiers. While there exists many approaches to cooperation between software verifiers, they often require specialized interfaces for communication. This hinders the application of off-the-shelf verification tools in a cooperative setting which do not support the required interfaces for information exchange. One idea to circumvent the problem is by encoding the verification progress directly into the program. Still, a verifier that communicates its verification progress is required.

In this talk and in the accompanying paper, we propose dynamic program splitting as a way to record the verification progress of off-the-shelf verification tools. Dynamic program splitting starts by running a given verifier. If the verifier cannot solve the task within a given time, dynamic program splitting then splits the task into multiple parts. The verifier is then executed on each subtask. This process continues on the unverified parts of the verification task, continuously splitting subtasks that are not yet solved. In the end, all unsolved parts are merged together to form a residual verification task. This task can directly be processed by a second verifier. The key advantage of our approach is that off-the-shelf verifiers can be directly be used without requiring specialized interfaces, as the main communication is performed on the task itself. Our evaluation shows that being able to use off-the-shelf tools can be a key advantage in cooperative verification. We also report on further experimental results and discuss open problems.

3.14 MoXI: An Intermediate Language to Spur Reproducible and Comparable Model Checking Research

Kristin Yvonne Rozier (Iowa State University – Ames, US)

License © Creative Commons BY 4.0 International license
 © Kristin Yvonne Rozier
Joint work of Kristin Yvonne Rozier, Rohit Dureja, Ahmed Irfan, Chris Johannsen, Karthik Nukala, Natarajan Shankar, Cesare Tinelli, Moshe Y. Vardi
Main reference Kristin Yvonne Rozier, Rohit Dureja, Ahmed Irfan, Chris Johannsen, Karthik Nukala, Natarajan Shankar, Cesare Tinelli, Moshe Y. Vardi: “MoXI: An Intermediate Language for Symbolic Model Checking”, in Proc. of the Model Checking Software - 30th International Symposium, SPIN 2024, Luxembourg City, Luxembourg, April 8-9, 2024, Proceedings, Lecture Notes in Computer Science, Vol. 14624, pp. 26–46, Springer, 2024.
URL https://doi.org/10.1007/978-3-031-66149-5_2
Main reference Chris Johannsen, Karthik Nukala, Rohit Dureja, Ahmed Irfan, Natarajan Shankar, Cesare Tinelli, Moshe Y. Vardi, Kristin Yvonne Rozier: “The MoXI Model Exchange Tool Suite”, in Proc. of the Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I, Lecture Notes in Computer Science, Vol. 14681, pp. 203–218, Springer, 2024.
URL https://doi.org/10.1007/978-3-031-65627-9_10

As symbolic model checking algorithms have grown more powerful and popular, their implementations have grown increasingly closed-source and specialized. This move away from standardization has stunted symbolic model checking research: there was not a way to build upon or adapt closed-source tools to compare with new algorithms; there was not a way to compare state-of-the-art back-end model checking algorithms because their implementations reasoned over models in different input languages; and replication of previous research results proved challenging due to hidden implementation details. After extensive input from the model checking research community, we have the intermediate language MoXI, the Model eXchange Interlingua, built to stimulate symbolic

model checking research. Publicly available MoXI translators enable comparability with previous research results. Adding a new back-end model checking algorithm and comparing it to previously published algorithms is now as easy as creating a MoXI translator to/from the new algorithm. One can now compare any number of back-end implementations over inputs in any number of different model or specification languages by compiling through MoXI. Accordingly, the public collection of translators through MoXI is quickly growing. We will highlight the state of the art in reproducible and comparable model checking research and point to community resources to continue the groundswell.

The GitHub organization provides full artifacts: <https://github.com/ModelChecker>.

3.15 ACSL: Behavioral Interface Specification Language for C Code

Julien Signoles (Université Paris-Saclay, CEA, List, FR)

License © Creative Commons BY 4.0 International license
© Julien Signoles

Main reference Louis Gauthier, Virgile Prevosto, Julien Signoles: “A Semantics of Structures, Unions, and Underspecified Terms for Formal Specification”, in Proc. of the 2024 IEEE/ACM 12th International Conference on Formal Methods in Software Engineering (FormaliSE), Lisbon, Portugal, April 14-15, 2024, pp. 100–110, ACM, 2024.

URL <https://doi.org/10.1145/3644033.3644380>

ACSL is a Behavioral Interface Specification Language for C Code. It is independent from any tool, even if Frama-C, a framework for analyses of C code, provides a reference implementation. ACSL is meant to express precisely and unambiguously the expected behavior of a piece of C code. It is close to other contract-based languages, e.g., JML for Java, even if there are differences that mainly comes from the underlying programming language. Typically, ACSL must support specification of (low-level) memory properties, which are critical for C code. This short talk introduces the main constructs of ACSL, illustrated by a few small examples.

References

- 1 P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto. ANSI/ISO C Specification Language Reference Manual. <https://frama-c.com/download/acsl.pdf>
- 2 A. Blanchard. Introduction to C program proof with Frama-C and its WP plugin. <https://allan-blanchard.fr/publis/frama-c-wp-tutorial-en.pdf>
- 3 A. Blanchard, C. Marché, V. Prevosto. Formally Expressing what a Program Should Do: The ACSL Language. In Guide to Software Verification with Frama-C: Core Components, Usages and Applications, Springer Cham, 2024

3.16 Correctness Verification Witnesses 2.0

Jan Strejček (Masaryk University – Brno, CZ)

License © Creative Commons BY 4.0 International license
© Jan Strejček

Joint work of Paulína Ayaziová, Dirk Beyer, Marian Lingsch Rosenfeld, Martin Spiessl, Jan Strejček
Main reference Paulína Ayaziová, Dirk Beyer, Marian Lingsch Rosenfeld, Martin Spiessl, Jan Strejček: “Software Verification Witnesses 2.0”, in Proc. of the Model Checking Software - 30th International Symposium, SPIN 2024, Luxembourg City, Luxembourg, April 8-9, 2024, Proceedings, Lecture Notes in Computer Science, Vol. 14624, pp. 184–203, Springer, 2024.

URL https://doi.org/10.1007/978-3-031-66149-5_11

Verification witnesses are now widely accepted objects used not only to confirm or refute verification results, but also for general exchange of information among various tools for program verification. The original format for witnesses is based on GraphML, and it has some known issues including a semantics based on control-flow automata, limited tool support of some format features, and a large

size of witness files. The version 2.0 of the witness format is based on YAML and overcomes the above-mentioned issues. The presentation focuses on correctness witnesses in format version 2.0 and it also mentions some extensions planned for version 2.1.

3.17 Paradigm-Spanning Collaboration: The Karlsruhe Java Verification Suite

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

License © Creative Commons BY 4.0 International license
© Mattias Ulbrich

Joint work of Jonas Klamroth, Florian Lanzinger, Wolfram Pfeifer, Mattias Ulbrich

Main reference Jonas Klamroth, Florian Lanzinger, Wolfram Pfeifer, Mattias Ulbrich: “The Karlsruhe Java Verification Suite”, in Proc. of the The Logic of Software. A Tasting Menu of Formal Methods - Essays Dedicated to Reiner Hähnle on the Occasion of His 60th Birthday, Lecture Notes in Computer Science, Vol. 13360, pp. 290–312, Springer, 2022.

URL https://doi.org/10.1007/978-3-031-08166-8_14

Deductive verification of sophisticated properties of sophisticated data structures is a difficult task. Proof endeavours in such contexts still have many proof obligation which do not require the expressiveness of a deductive verifier.

In the talk I presented the Karlsruhe Java Verification Suite, a collection of verification tool around the deductive verification engine KeY. The bounded model checker JJBMC extends the checker JBMC by JML specifications and the Property Checker uses the Java Checker Framework to capture object properties and provides a lightweight method to annotate formal guarantees throughout an application.

The tools cooperate via JML and Java assertions. We show that if every assertion in a program is handled by at least one sound verification tool all other tools can change assertions into helpful assumptions.

3.18 Reducers to Aid Information Exchange

Heike Wehrheim (Universität Oldenburg, DE)

License © Creative Commons BY 4.0 International license
© Heike Wehrheim

Joint work of Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, Heike Wehrheim

Main reference Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, Heike Wehrheim: “Reducer-based construction of conditional verifiers”, in Proc. of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 1182–1193, ACM, 2018.

URL <https://doi.org/10.1145/3180155.3180259>

Despite recent advances, software verification remains challenging. To solve hard verification tasks, we need to leverage not just one but several different verifiers employing different technologies. To this end, we need to exchange information between verifiers. Conditional model checking was proposed as a solution to exactly this problem: The idea is to let the first verifier output a condition which describes the state space that it successfully verified and to instruct the second verifier to verify the yet unverified state space using this condition. However, most verifiers do not understand conditions as input.

In this talk and the accompanying paper, we propose the usage of an off-the-shelf construction of a conditional verifier from a given traditional verifier and a reducer. The reducer takes as input the program to be verified and the condition, and outputs a residual program whose paths cover the unverified state space described by the condition. This program can then be given to the second verifier for completing the verification. We report on results of some experiments and discuss open questions.

3.19 Bridging Hardware and Software Formal Verification

Zsófia Ádám (Budapest University of Technology & Economics, HU), Dirk Beyer (LMU Munich, DE)

License © Creative Commons BY 4.0 International license

© Zsófia Ádám and Dirk Beyer

Joint work of Zsófia Ádám, Dirk Beyer, Po-Chun Chien, Nian-Ze Lee, Nils Sirrenberg
Main reference Zsófia Ádám, Dirk Beyer, Po-Chun Chien, Nian-Ze Lee, Nils Sirrenberg: “Btor2-Cert: A Certifying Hardware-Verification Framework Using Software Analyzers”, in Proc. of the Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III, Lecture Notes in Computer Science, Vol. 14572, pp. 129–149, Springer, 2024.
URL https://doi.org/10.1007/978-3-031-57256-2_7

Computing systems demand meticulous verification to ensure their correct functionality. Formal methods have been successfully applied in real-world scenarios and delivered correctness guarantees with mathematical rigor. The challenges to formal methods have escalated due to increasing interactions among diverse components, e.g., software programs, hardware circuits, and cyber-physical devices, therefore, a cross-disciplinary approach is necessary among separate research communities. Although the research communities for formal methods share similar concepts and techniques, their alignment with distinct computational models creates gaps between the communities. In this talk, we introduce our attempts to bridge the gap between software and hardware analysis, including transferring verification approaches and translating verification tasks. By systematically cross-applying their techniques and extracting valuable insights into analyzing heterogeneous computing systems, we aim to unify verification approaches for comprehensive system-level verification.

4 Working Groups

4.1 Breakout Group: The Goals of the Seminar from the Perspective of Software Model Checking Community

Jan Strejček (Masaryk University – Brno, CZ)

License © Creative Commons BY 4.0 International license

© Jan Strejček

Joint work of Zsófia Ádám, Paulína Ayaziová, Marek Chalupa, Daniel Dietsch, Matthias Heizmann, Marie-Christine Jakobs, Thomas Lemberger, Cedric Richter, Jan Strejček, Heike Wehrheim

The breakout group consisted mainly of developers of automatic software verification tools. The group formulated the following answers to questions given by organizers.

What do we want to achieve with the exchange formats? The group wants to support the cooperation between (static and dynamic) verifiers and/or program analyses (that can provide, e.g., points-to information). It should also facilitate incremental verification, validation of verification results, human/AI interaction with verification tools, and support human reasoning about the program and found property violations.

Which objects do we want to exchange? The group came up with the following objects: invariants (including object invariants and transition invariants), useful ghost code, abstract reachability graphs, precision sufficient to prove the task, function contracts/summaries, error conditions, ranking functions, abstract contracts (independent of the implementation), assumptions for verifiers, verification sub-tasks (that were not proven yet), memory-specific information (e.g., shape graphs), information about synchronization between threads (e.g., execution graphs or lock order), sets of specific program paths (e.g., abstract counterexamples), and meta-information about the provenance of the presented objects.

What are suitable formats to use for the exchange? Besides the formats that are already widely used by the community (C programs, witness formats), the group suggested using (E-)ACSL and introducing specific predicates to describe some of the objects identified above. The group

also discussed whether the information retrieval should be static or query-based and whether the information should be exchanged within the program (internally) or aside (externally). Here the group did not agree on answers as each option has its pros and cons.

4.2 Breakout Group: Exchange Between Deductive Verification

Marieke Huisman (University of Twente – Enschede, NL)

License © Creative Commons BY 4.0 International license
© Marieke Huisman

Joint work of Wolfgang Ahrendt, David Cok, Gidon Ernst, Jean-Christophe Filliâtre, Marieke Huisman, Nikolai Kosmatov, Robert Mensing, Rosemary Monahan, Simmo Saan, Julien Signoles, Alexander Stekelenburg, Mattias Ulbrich

This breakout group had participants that are working on deductive verifiers. The discussion concentrated mainly on exchange between deductive verification tools. We started by discussing how interoperability involving deductive verifiers could be good. Concrete examples where heterogeneous verification, where different tools target different properties, validation and cross-checking of verification results, and combinations with automatic tools that could be used first to resolve the basic properties, or to infer specifications.

We then discussed what would be the appropriate level of exchange. One obvious option is to exchange concrete examples, i.e., programs with specifications. Exchanging intermediate verification results would be interesting, but also much more challenging. One important candidate is the exchange of axiomatisations of data structures and logical theories, that are used in specifications, possibly based on format inspired by SMT-LIB.

Finally, we concluded the discussion by identifying areas where already some exchange is happening between deductive verifiers, such as ContractLib, and the language constructs for which exchange could take place, such as well-encapsulated objects, lemmas etc.

In a second session, the breakout group continued by identifying a list of concrete short-term actions that could be taken to experiment with information exchange: sharing axiomatisations, work on a common example together, and organise a hackathon/specathon, to make progress on the examples, share examples of verified programs, and add tools to FM-Tools.

In the long term, we believe that these actions should lead to support for specification exchange, annotation exchange and proof exchange. We also identified a concrete list of semantical issues that we would have to agree upon to enable this.

4.3 Breakout Group: Intermediate Language for Software Verification

Dirk Beyer (LMU Munich, DE)

License © Creative Commons BY 4.0 International license
© Dirk Beyer

Joint work of Dirk Beyer, Gidon Ernst, Alberto Griggio, Martin Jonáš, Viktor Malík, Robert Mensing, Kristine Yvonne Rozier, Jan Strejček, Vesal Vojdani

The state of the art in software verification is to use programming languages like C and Java, possibly annotated using ACSL, JML, and other annotation languages, to describe the verification tasks. The two breakout sessions discussed the need to exchange verification tasks and the motivation to create a new, community-based, intermediate language (IL) that is easy to support and process by tools. The requirements for the language are as follows: The IL should

- (a) have a formal semantics, such that comparative evaluations of verification algorithms does not depend on interpretations, and the same should hold for witness validation,
- (b) support interoperability, such that frontends and backends are exchangeable,
- (c) support verification and witness validation by transformation,

- (d) be independent from the higher-level input language (such as C and Java) and independent from verification technology used in the backend,
- (e) retain more structure than MoXI, VMT, and CHC (control flow, procedures, sequential composition, ...) to express tasks before encoding to verification conditions by a particular approach,
- (f) be based on well-understood programming concepts and consolidate existing languages in that spectrum (e.g., Boogie, Why3, Viper),
- (g) support witness validation that is compilable to a simple first-order check (i.e., SMT) without further inference, for all properties, and
- (h) appeal to use-cases in both software model checking and deductive verification.

The objects to be exchanged by the language shall be (1) programs, (2) properties, and (3) witnesses (counterexamples and invariants). The group suggests that the new IL should be based on a text format (not binary) and use a well-known base syntax (e.g., Lisp-based). As a concrete outcome, it was agreed that the language would be defined before autumn 2025 and that a track for the language shall be established by the SV-COMP competition organizers.

4.4 Breakout Group: Decomposition of the Verification Task

Heike Wehrheim (Universität Oldenburg, DE)

License © Creative Commons BY 4.0 International license
© Heike Wehrheim

Joint work of Ádám, Zsófia; Wolfgang Ahrendt, Paulína Ayaziová, Marek Chalupa, Daniel Dietsch, Matthias Heizmann, Marie-Christine Jakobs, Thomas Lemberger, Cedric Richter, Simmo Saan, Vesal Vojdani, Heike Wehrheim

A verification task is a program and a (correctness) specification. The purpose of decomposition is to divide a verification task into several subtasks which can be given to (different) software verifiers for solving them. The verifier's results then at the end need to be combined. Thus, we do not just need decomposition, but also merging of results (for which we however did not have time in the discussion).

We basically identified two sorts of decomposition:

Based on program structure Decomposition based on program structure might split programs on branches or along paths. We decided to not discuss this longer as there are already a number of proposals around.

Based on partial results The second sort of decomposition we considered is the one based on a partial results, i.e., one verifier has already inspected the verification task and has obtained some partial information about it. This partial information might have come along because of a non-finished (full) proof attempt or as a result of a specific query (e.g., “give me all points-to information”).

As sorts of partial results we identified (a) conditions (as in Conditional Model Checking), (b) annotated Control-Flow-Graphs/Automata (this is what e.g. the verifier Goblint could produce) or (c) specific assumptions under which the verification has succeeded (e.g., knowledge about no non-null pointers or predicates about input variables). Potential formats of such assumptions have then been further discussed in another breakout group.

4.5 Breakout Group: Exchange of Partial Verification Results

Zsófia Ádám (Budapest University of Technology & Economics, HU) and Thomas Lemberger (LMU Munich, DE)

License © Creative Commons BY 4.0 International license

© Zsófia Ádám and Thomas Lemberger

Joint work of Zsófia Ádám, Wolfgang Ahrendt, Paulína Ayaziová, Marek Chalupa, Daniel Dietsch, Thomas Lemberger, Heike Wehrheim

Automated and deductive verifiers perform well for different challenges. This motivates different methods of cooperation, which often require the sharing of (partial) verification results. However, no widely adopted exchange format between automated and deductive verifiers exists. The working group on exchange of partial verification results focused on three aspects of a successful exchange format:

- (1) **The type of information that may be exchanged.** Most notably:
 - Local or global assumptions that the verifier assumed;
 - invariants of various types: global invariants, location-bound invariants, loop invariants, object invariants, and resource invariants; and
 - properties that were established through the analysis, such as safety of assertions or nullability.
- (2) **Existing related formats that serve as examples of information exchange in verification.** For example the Software Witness 2.0 format and (E-)ACSL. In this context the working group also discussed various pitfalls in exchange formats: undefined behavior in the expression language and how to precisely define elements and corresponding locations tool-independently.
- (3) **The terminology to use and biases in data representation that may be introduced through certain terminology.** For example, the terms *assumed* (compared to *assert*) and *asserted* (compared to *assume*) express that the information is not instructions for a potential consumer, but that it is a description of the performed work and established truths; independent of the consumer's use case.

Going forward, we propose a wider discussion on the above concepts and ideas to establish a consumer-independent exchange format for partial verification results.

4.6 Breakout Group: Exchange Between Deductive Tools (DV) and Fully Automatic Software Verifiers (SV)

Marieke Huisman (University of Twente – Enschede, NL)

License © Creative Commons BY 4.0 International license

© Marieke Huisman

Joint work of David Cok, Gidon Ernst, Jean-Christophe Filliâtre, Alberto Griggio, Matthias Heizmann, Marieke Huisman, Nikolai Kosmatov, Robert Mensing, Rosemary Monahan, Cedric Richter, Julien Signoles, Alexander Stekelenburg, Mattias Ulbrich

In this breakout group, we discussed how fully automatic software verifiers could be combined with deductive verifiers. We first discussed several ongoing efforts in this direction. There are several ways in which the combination can work: deductive verifiers can prove some parts that are hard for automated software verifiers, software verifiers can prove dedicated properties that are useful for the deductive verifiers. We can also use the combination to speed up the overall verification process.

One particular challenge is that deductive verifiers often uses specifications with quantifiers. Not all software verifiers can reason about those, but there are several tricks to work around this, e.g., by encoding the quantified variable as a non-deterministic variable, but this encoding hinders the exchange.

We agreed on several follow-up actions, in particular to look at some concrete examples and look what could be achieved. We also intend to collect the existing approaches for integration in a joint document, so everybody could study them. Eventually, we plan to organise some online events to discuss the experiences with some of these concrete examples.

4.7 Breakout Group: Information Exchange Between Automatic SW Verifiers

Jan Strejček (Masaryk University – Brno, CZ)

License  Creative Commons BY 4.0 International license

© Jan Strejček

Joint work of Dirk Beyer, Marie-Christine Jakobs, Martin Jonáš, Viktor Malík, Kristin Yvonne Rozier, Simmo Saan, Jan Strejček, Vesal Vojdani

The group went through (a part of) the list of objects for exchange between verification tools identified by Wednesday’s breakout group of the software model checking community (see Section 4.1). Using that list, we suggested further extensions of the witness format 2.0, in particular more general ghost code, **modifies** clause in function contracts, and support of a fragment of ACSL.

In the second part of the session, the group focused on designing a format for exchanging precision (i.e., used predicates and relevant variables) that was used during the verification run. We agreed on the objects that should represent the precision: side-effect-free and terminating C expressions together with their scope (global, local, or a particular function). We also decided to use YAML and adopt some parts of the witness format 2.0. Finally, we have formulated some action items aiming to establish the precision exchange format.

Participants

- Zsófia Ádám
Budapest University of
Technology & Economics, HU
- Wolfgang Ahrendt
Chalmers University of
Technology – Göteborg, SE
- Paulína Ayaziová
Masaryk University – Brno, CZ
- Dirk Beyer
LMU München, DE
- Marek Chalupa
IST Austria –
Klosterneuburg, AT
- David Cok
Safer Software Consulting –
Rochester, US
- Daniel Dietsch
Universität Freiburg, DE & Qt
Group – Espoo, FI
- Gidon Ernst
LMU München, DE
- Jean-Christophe Filliâtre
CNRS – Gif-sur-Yvette, FR
- Alberto Griggio
Bruno Kessler Foundation –
Trento, IT
- Matthias Heizmann
Universität Stuttgart, DE
- Marieke Huisman
University of Twente –
Enschede, NL
- Marie-Christine Jakobs
LMU München, DE
- Martin Jonáš
Masaryk University – Brno, CZ
- Nikolai Kosmatov
Thales Research & Technology –
Palaiseau, FR
- Thomas Lemberger
LMU München, DE
- Viktor Malík
Brno University of Technology,
CZ
- Robert Mensing
University of Twente –
Enschede, NL
- Rosemary Monahan
Maynooth University, IE
- Cedric Richter
Carl von Ossietzky Universität
Oldenburg, DE
- Kristin Yvonne Rozier
Iowa State University –
Ames, US
- Simmo Saan
University of Tartu, EE
- Julien Signoles
Université Paris-Saclay, CEA,
List, FR
- Alexander Stekelenburg
University of Twente –
Enschede, NL
- Jan Strejček
Masaryk University – Brno, CZ
- Mattias Ulbrich
KIT – Karlsruher Institut für
Technologie, DE
- Vesal Vojdani
University of Tartu, EE
- Heike Wehrheim
Universität Oldenburg, DE

