

Theoretical Advances and Emerging Applications in Abstract Interpretation

Arie Gurfinkel^{*1}, Isabella Mastroeni^{*2}, Antoine Miné^{*3},
Peter Müller^{*4}, and Anna Becchi^{†5}

1 University of Waterloo, CA. arie.gurfinkel@uwaterloo.ca

2 University of Verona, IT. isabella.mastroeni@univr.it

3 Sorbonne University – Paris, FR. antoine.mine@lip6.fr

4 ETH Zürich, CH. peter.mueller@inf.ethz.ch

5 Bruno Kessler Foundation – Trento, IT. abecchi@fbk.eu

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 23281 “Theoretical Advances and Emerging Applications in Abstract Interpretation.”

Abstract Interpretation (AI) is a theory of the approximation of program semantics. Since its introduction in the 70s, it led to insights into theoretical research in semantics, a rich and robust mathematical framework to discuss about semantic approximation and program analysis, and the design of effective program analysis tools that are now routinely used in this industry. The seminar brought together academic and industrial partners to assess the state of the art in AI as well as discuss its future. It considered its foundational aspects, connections with other formal methods, emergent applications, user needs in program verification, tool design and evaluation, as well as educational aspects and community management. Its goal was to collect new ideas and new perspectives on all these aspects of AI in order to pave the way for new applications.

Seminar July 9–14, 2023 – <https://www.dagstuhl.de/23281>

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Software and its engineering → Completeness; Software and its engineering → Correctness; Software and its engineering → Formal methods; Software and its engineering; Software and its engineering → Software functional properties; Software and its engineering → Software safety; Software and its engineering → Software verification and validation

Keywords and phrases abstract domains, abstract interpretation, program semantics, program verification, static program analysis

Digital Object Identifier 10.4230/DagRep.13.7.66


1 Executive Summary

Antoine Miné (Sorbonne University – Paris, FR)

Arie Gurfinkel (University of Waterloo, CA)

Isabella Mastroeni (University of Verona, IT)

Peter Müller (ETH Zürich, CH)

License  Creative Commons BY 4.0 International license
© Antoine Miné, Arie Gurfinkel, Isabella Mastroeni, and Peter Müller

Abstract Interpretation (AI) is a theory of the approximation of possible program behaviors. Since its introduction in the late 70s, it has evolved into a very general theory to describe and compare formal semantics of programs and systems. As a more practical aspect, it provides a

* Editor / Organizer

† Editorial Assistant / Collector



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

Theoretical Advances and Emerging Applications in Abstract Interpretation, *Dagstuhl Reports*, Vol. 13, Issue 7, pp. 66–95

Editors: Arie Gurfinkel, Isabella Mastroeni, Antoine Miné, Peter Müller, and Anna Becchi



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

formal framework to design effective static analyzers that are automatic, sound, and efficient. The last two decades have seen the emergence of practical tools now used routinely in the industry, starting with the embedded critical software industry and now being also applied to more mainstream software.

Despite its strength, designing new static analyses with AI remains a challenging task, involving both theoretical research and engineering efforts. The limited diffusion of AI knowledge in Universities, in Engineering Schools, and in the industry may hinder the development and more widespread use of AI-based tools. Moreover, the early focus on the verification of run-time errors in embedded software, while it provided a simpler and more controlled context than the general problem of verifying consumer software, and resulted in industrial successes, also could give the false impression that AI is only suitable for this task, while its theory is in fact much more general. Finally, the AI community has little interactions with that of other formal methods.

Based on these early observations, we set out to organize a Dagstuhl seminar on “Theoretical Advances and Emerging Applications in Abstract Interpretation” to discuss the state of the art in AI, identify its key challenges, and plan for its future. The seminar brought together 37 international experts in static analysis, from 10 countries, and from both academic and industrial background, covering a wide spectrum from pure Abstract Interpretation (AI) theory, to tool providers, to industrial users.

To provide a structure to the discussion, the seminar was organized as a series of topical days focusing on identified aspects (but not excluding other topics) of the state of the art and perceived challenges in AI: static safety verification, tools and applications, verification beyond safety analysis, and education. We proposed three invited talks and an invited tutorial of extended length (1h) and from key people in the community to bootstrap discussions on a variety of topics, ranging from theoretical AI to industrial tools, and providing historical perspectives. These were complemented with 23 shorter talks proposed spontaneously throughout the seminar. These talks, from 10min to 30min long, discussed a large variety of topics, including new research results, theoretical advances, experience reports, technical realizations, and reports on teaching activities. A significant number of presentations discussed connections of AI with other formal methods, including SMT solving, types, program logic.

During the seminar, we also organized several breakout sessions, where smaller working groups discussed a selection of topics: soundness requirements for static analysis tools, expressive domains, community infrastructure, education, connections with other formal methods, connections with machine learning, and tackling the verification of hyper-properties. As a conclusion of these working groups and the overall seminar, several tasks were started and a number of action items were proposed to advance further:

- The group on soundness requirements proposed a first list of requirements that an analyzer should fulfill. It identified the need to discuss these findings with members of the soundness manifesto, which raised awareness on the lack of proper requirements.
- The group on community infrastructure proposed a series of practical actions to build the community on AI. It suggested the creation of a Special Interest Group in AI to coordinate the efforts.
- The group on teaching started an on-going list (to be completed) of educational materials on AI. It presented the need to develop materials targeting undergraduate students and practitioners, to make efforts to better share available teaching resources, and to provide introductory courses on AI on MOOC platforms.
- The group on the interaction between AI and other formal methods presented relevant connections with deductive verification, dynamic techniques, and model checking. In particular, it suggested the organization of a seminar on AI and deductive verification.

In general, this seminar expressed the interest to continue the discussion on the future of AI in further seminars focusing on specific challenges and opportunities as uncovered during the seminar, and adapting the list of participants in consequence.

2 Table of Contents

Executive Summary

Antoine Miné, Arie Gurfinkel, Isabella Mastroeni, and Peter Müller 66

Overview of Talks

Code Reuse Vulnerabilities in Modern Web Applications
Musard Balliu 71

Cooperative Verification
Dirk Beyer 71

Interactive Abstract Interpretation
Bor-Yuh Evan Chang 72

Formal Verification of Avionics Software
David Delmas 73

Calculating Equational Laws over ADTs
Gidon Ernst 73

Teaching Abstract Interpretation with LiSA
Pietro Ferrara 74

Alpha from Below over Quantified First-order Formulas
Eden Frenkel 74

Fast Approximations of Quantifier Elimination
Isabel Garcia-Contreras, Arie Gurfinkel, Hari Govind V K, and Sharon Shoham Buchbinder 75

Uniform Interpolation for Efficient Domain Reduction
Isabel Garcia-Contreras, Arie Gurfinkel, and Jorge Navas 75

An incomplete journey in Completeness
Roberto Giacobazzi 76

Abstract interpretation based under approximations and Sufficient Incorrectness Logic
Roberta Gori 76

On the fly verification with (incremental) interactive abstract interpretation
Manuel Hermenegildo 77

Automated Reasoning for Privacy
Temesghen Kahsai 78

Abstract Interpretation in Industry – Practical Experience with Astrée
Daniel Kästner 79

SSA Translation Is an Abstract Interpretation, and its Application to Machine Code Analysis
Matthieu Lemerre 79

A Multilanguage Static Analysis of Python/C Programs with Mopsa
Raphaël Monat and Antoine Miné 80

Crab: A library for building abstract-interpretation-based analyses
Jorge Navas 80

| | |
|--|----|
| Calculational Design of Program Logics by Abstract Interpretation | |
| <i>Patrick Cousot</i> | 81 |
| Mentorship for Formal Methods | |
| <i>Ruzica Piskac</i> | 81 |
| Abstract Interpretation-based Program (Analysis) Logics | |
| <i>Francesco Ranzato</i> | 82 |
| VeriCode: Correct Translation of Abstract Specifications to C-Code | |
| <i>Gerhard Schellhorn</i> | 82 |
| Data Race Repair using Static Analysis Summaries | |
| <i>Ilya Sergey</i> | 83 |
| Property-Directed Reachability as Abstract Interpretation in the Monotone Theory | |
| <i>Sharon Shoham Buchbinder</i> | 83 |
| Exploiting Pointer Analysis in Memory Models for Deductive Verification | |
| <i>Mihaela Sighireanu</i> | 84 |
| (Un-)Realizability of Condition Synthesis as CHC-SAT | |
| <i>Yakir Vizel</i> | 84 |
| Dataflow Refinement Type Inference | |
| <i>Thomas Wies</i> | 85 |
| Timing Analysis by Abstract Interpretation | |
| <i>Reinhard Wilhelm</i> | 85 |
| Working groups | |
| Soundness requirements, transparency of assumptions | |
| <i>Bor-Yuh Evan Chang and Raphaël Monat</i> | 86 |
| Expressive Domains | |
| <i>Gidon Ernst</i> | 87 |
| Education: Teaching Abstract Interpretation to the Masses | |
| <i>Pietro Ferrara</i> | 88 |
| Community Infrastructure – Interest Group in Static Analysis | |
| <i>Pietro Ferrara</i> | 89 |
| Abstract Interpretation and Other Formal Methods | |
| <i>Arie Gurfinkel</i> | 90 |
| Tools and Applications for Abstract Interpretation | |
| <i>Falk Howar</i> | 91 |
| Hyperproperties verification | |
| <i>Isabella Mastroeni</i> | 92 |
| AI for AI | |
| <i>Antoine Miné</i> | 93 |
| Participants | 95 |

3 Overview of Talks

3.1 Code Reuse Vulnerabilities in Modern Web Applications

Musard Balliu (KTH Royal Institute of Technology – Stockholm, SE)

License © Creative Commons BY 4.0 International license
© Musard Balliu

Joint work of Mikhail Shcherbakov, Musard Balliu, Cristian-Alexandru Staicu

Main reference Mikhail Shcherbakov, Musard Balliu, Cristian-Alexandru Staicu: “Silent Spring: Prototype Pollution Leads to Remote Code Execution in Node.js”, in Proc. of the 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023, pp. 5521–5538, USENIX Association, 2023.

URL <https://www.usenix.org/conference/usenixsecurity23/presentation/shcherbakov>

We study code reuse vulnerabilities in modern web application. Prototype pollution is a dangerous vulnerability affecting prototype-based languages like JavaScript and the Node.js platform. It refers to the ability of an attacker to inject properties into an object’s root prototype at runtime and subsequently trigger the execution of legitimate code gadgets that access these properties on the object’s prototype, leading to attacks such as Denial of Service (DoS), privilege escalation, and Remote Code Execution (RCE).

In this work, we set out to study the problem in a holistic way, from the detection of prototype pollution to detection of gadgets, with the goal of finding end-to-end exploits beyond DoS, in full-fledged Node.js applications. We build a multi-staged framework that uses multi-label static taint analysis to identify prototype pollution in Node.js libraries and applications, as well as a hybrid approach to detect universal gadgets, notably, by analyzing the Node.js source code. We implement our framework on top of GitHub’s static analysis framework CodeQL to find 11 universal gadgets in core Node.js APIs, leading to code execution. Furthermore, we use our methodology in a study of 15 popular Node.js applications to identify prototype pollutions and gadgets. We manually exploit eight RCE vulnerabilities in three high-profile applications such as NPM CLI, Parse Server, and Rocket.Chat.

3.2 Cooperative Verification

Dirk Beyer (LMU München, DE)

License © Creative Commons BY 4.0 International license
© Dirk Beyer

Joint work of Dirk Beyer, Heike Wehrheim

Main reference Dirk Beyer, Heike Wehrheim: “Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework”, in Proc. of the Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles – 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I, Lecture Notes in Computer Science, Vol. 12476, pp. 143–167, Springer, 2020.

URL https://doi.org/10.1007/978-3-030-61362-4_8

Cooperative verification is an approach in which several verifiers help each other solving the verification problem by sharing artifacts about the verification process. There are many verification tools available, and they have different strengths. While the tools continuously increase their individual capabilities, the potential of cooperation is largely unused. The problem is that in order to use verifiers ‘off-the-shelf’, we need clear interfaces to invoke the tools and to pass information. Part of the interfacing problem is to define standard artifacts to be exchanged between verifiers. We explain a few recent approaches for cooperative combinations that are based on verification witnesses as exchange format, including witness

validation, component-based CEGAR, and exchanging invariants between automatic and interactive verifiers. We also give a brief overview of CoVeriTeam, a tool for composing verification systems from existing off-the-shelf components.

References

- 1 Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. *J. Autom. Reasoning* 60(3), 299–335 (2018). <https://doi.org/10.1007/s10817-017-9432-6>
- 2 Beyer, D., Haltermann, J., Lemberger, T., Wehrheim, H.: Decomposing Software Verification into Off-the-Shelf Components: An Application to CEGAR. In: *Proc. ICSE*. pp. 536–548. ACM (2022). <https://doi.org/10.1145/3510003.3510064>
- 3 Beyer, D., Kanav, S.: CoVeriTeam: On-demand composition of cooperative verification systems. In: *Proc. TACAS*. pp. 561–579. LNCS 13243, Springer (2022). https://doi.org/10.1007/978-3-030-99524-9_31
- 4 Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: *Proc. CAV*. pp. 184–190. LNCS 6806, Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
- 5 Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: *Proc. SEFM*. p. 111–128. LNCS 13550, Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7
- 6 Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. In: *Proc. ISoLA* (1). pp. 143–167. LNCS 12476, Springer (2020). https://doi.org/10.1007/978-3-030-61362-4_8

3.3 Interactive Abstract Interpretation

Bor-Yuh Evan Chang (University of Colorado – Boulder, US)

License © Creative Commons BY 4.0 International license
© Bor-Yuh Evan Chang

Joint work of Benno Stein, Bor-Yuh Evan Chang, Manu Sridharan, David Flores
Main reference Benno Stein, Bor-Yuh Evan Chang, Manu Sridharan: “Demanded abstract interpretation”, in *Proc. of the PLDI ’21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pp. 282–295, ACM, 2021.

URL <https://doi.org/10.1145/3453483.3454044>

We consider the problem of making expressive static analyzers interactive. Formal static analysis is seeing increasingly widespread adoption as a tool for verification and bug-finding, but even with powerful cloud infrastructure it can take minutes or hours to get batch analysis results after a code change. While existing techniques offer some demand-driven or incremental aspects for certain classes of analysis, the fundamental challenge we tackle is doing both for arbitrary abstract interpreters. Our technique, demanded abstract interpretation, lifts program syntax and analysis state to a dynamically evolving graph structure, in which program edits, client-issued queries, and evaluation of abstract semantics are all treated uniformly.

3.4 Formal Verification of Avionics Software

David Delmas (Airbus S.A.S. – Toulouse, FR)

- License** © Creative Commons BY 4.0 International license
© David Delmas
- Joint work of** David Delmas, Antoine Miné, Abdelraouf Ouadjaout, Famantanantsoa Randimbivololona, Abderrahmane Brahmi
- Main reference** David Delmas, Abdelraouf Ouadjaout, Antoine Miné: “Static Analysis of Endian Portability by Abstract Interpretation”, in Proc. of the Static Analysis – 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings, Lecture Notes in Computer Science, Vol. 12913, pp. 102–123, Springer, 2021.
URL https://doi.org/10.1007/978-3-030-88806-0_5
- Main reference** Abdellatif Atki, Abderrahmane Brahmi, David Delmas, Mohamed Habib Essoussi, Famantanantsoa Randimbivololona, Thomas Marie: “Formalise to automate: deployment of a safe and cost-efficient process for avionics software”.In ERTSS: Proc. of the 9th European Congress on Embedded Real Time Software and Systems, Jan 2018, Toulouse, France.
URL <https://www.di.ens.fr/delmas/erts18/>
- Main reference** David Delmas: “Static analysis of program portability by abstract interpretation”, 2022.
URL <https://theses.hal.science/tel-04028096>

The size and complexity of avionics software have grown exponentially from one aircraft generation to the next in the past 4 decades. Traditional software development processes leveraging informal verification techniques fail to scale within reasonable costs. In particular, verification is liable for a steadily growing share of the overall development costs. The 2015 current status was about 70.

To address this issue, Airbus have been transforming internal development processes since 2016. Internal domain-specific languages have been developed to enable the formalization of design artifacts, and automate part of verification activities. Automation is enabled by the interoperation of tools relying on sound formal techniques. For instance, Frama-C/WP and SMT-solvers are used to automate unit verification with deductive methods. Most so-called Unit Proofs are automatic, assuming high-level memory and numerical models, as well as some preconditions. Such assumptions are verified by other tools, such as the Astrée static analyzer, which leverages Abstract Interpretation to prove the absence of run-time errors and check assumed non-aliasing properties. We rely on the CompCert formally verified compiler to enable that most formal verification activities may be conducted on source code.

Beyond safety properties and currently established processes, we also develop internally static analyses by Abstract Interpretation to automate regression verification and portability verification. In particular, our portability analysis is able to prove without false alarms the portability of low-level C avionics software up to 1 million lines of C across platforms with opposite byte-orders (endianness).

3.5 Calculating Equational Laws over ADTs

Gidon Ernst (LMU München, DE)


- License** © Creative Commons BY 4.0 International license
© Gidon Ernst
- Joint work of** Gidon Ernst, Grigory Fedyukovich, Robin Sögtrop

We motivate the use of program transformation, traditionally used for the optimization of functional programs, as basic building blocks for lemma synthesis for recursive functions over algebraic data types. The key idea is to calculate at the function level instead of at the formula level, which allows one to work with intermediate syntactic functions. An important step therefore is to complement existing techniques (fixpoint fusion, deaccumulation) with

capabilities to recognize when two functions are identical resp. share commonalities wrt. preconditions. The approach, work in progress, discovers typical distributive laws of list/tree functions like length, append, remove, which are similar in shape to those lemmas often required for data refinement proofs.

3.6 Teaching Abstract Interpretation with LiSA

Pietro Ferrara (University of Venice, IT)

License  Creative Commons BY 4.0 International license
© Pietro Ferrara

Joint work of Pietro Ferrara, Luca Negrini, Vincenzo Arceri

LiSA (Library for Static Analysis – <https://github.com/lisa-analyzer/lisa>) is a Java library that implements the most common components of abstract interpretation-based static analyses. In this talk, we report our experience when adopting LiSA during courses at the master level focused on the theory of abstract interpretation. LiSA allowed students to implement, execute, and practice the theoretical concepts formalized throughout the course. However, it turned out that proper implementation of non-trivial abstract domains is beyond the capabilities of standard CS master students.

3.7 Alpha from Below over Quantified First-order Formulas

Eden Frenkel (Tel Aviv University, IL)

License  Creative Commons BY 4.0 International license
© Eden Frenkel

Joint work of Eden Frenkel, Sharon Shoham Buchbinder, Oded Padon, Tej Chajed

Often, verification of infinite-state and other complex systems, such as the Paxos consensus protocol, must work for unbounded domains. Quantified first-order logic allows us to reason over these unbounded domains, describe the desired behavior of these systems, and specify correctness properties. This is done by constructing formulas that describe the initial states of the system and its possible transitions, as well as formalizing the notion of illegal or unwanted states. A proof for the safety of a system then becomes an unreachability proof for the bad states, which can be provided via another formula, a safety invariant, which holds on the initial states, is invariant with respect to the transitions, and doesn't hold on the bad states.

In this work we tackle invariant inference through the framework of abstract interpretation. We propose an algorithm that computes the strongest over-approximation of reachable states by iteratively sampling counter-examples to induction, but which is highly infeasible due to the magnitude of the abstract domain and SMT solver limitations. We proceed by presenting techniques based on syntactic subsumption that manage to avoid redundant formulas and explore an identically expressive search space exponentially more efficiently, and generalize to the abstract domain of first-order formulas.

3.8 Fast Approximations of Quantifier Elimination

Isabel Garcia-Contreras (University of Waterloo, CA), Arie Gurfinkel (University of Waterloo, CA), Hari Govind V K, and Sharon Shoham Buchbinder (Tel Aviv University, IL)

License © Creative Commons BY 4.0 International license

© Isabel Garcia-Contreras, Arie Gurfinkel, Hari Govind V K, and Sharon Shoham Buchbinder

Main reference Isabel Garcia-Contreras, Hari Govind V. K., Sharon Shoham, Arie Gurfinkel: “Fast Approximations of Quantifier Elimination”, CoRR, Vol. abs/2306.10009, 2023.

URL <https://doi.org/10.48550/ARXIV.2306.10009>

Quantifier elimination (qelim) is used in many automated reasoning tasks including program synthesis, exist-forall solving, quantified SMT, Model Checking, and solving Constrained Horn Clauses (CHCs). Exact qelim is computationally expensive. Hence, it is often approximated. For example, Z3 uses “light” pre-processing to reduce the number of quantified variables. CHC-solver Spacer uses model-based projection (MBP) to under-approximate qelim relative to a given model, and over-approximations of qelim can be used as abstractions. In this talk, we present the QEL framework for fast approximations of qelim. QEL provides a uniform interface for both quantifier reduction and model-based projection. QEL builds on the egraph data structure – the core of the EUF decision procedure in SMT – by casting quantifier reduction as a problem of choosing ground (i.e., variable-free) representatives for equivalence classes. We have used QEL to implement MBP for the theories of Arrays and Algebraic Data Types (ADTs). We integrated QEL and our new MBP in Z3 and evaluated it within several tasks that rely on quantifier approximations, outperforming state-of-the-art.

3.9 Uniform Interpolation for Efficient Domain Reduction

Isabel Garcia-Contreras (University of Waterloo, CA), Arie Gurfinkel (University of Waterloo, CA), and Jorge Navas (Certora – Seattle, US)

License © Creative Commons BY 4.0 International license

© Isabel Garcia-Contreras, Arie Gurfinkel, and Jorge Navas

Handling precise abstract values over large sets of program variables is costly. A solution is to split such “monolithic” values (over the whole set of program variables) into several subvalues over smaller subsets of variables. Applying abstract transformers separately to each subvalue is then more efficient than to the monolithic one, but typically incurs precision loss. To address this, one can transfer information between subvalues, a process known as reduction. Reduction is done iteratively by refining a subvalue using information from other subvalues until no values are further refined. Information transfer can be stopped at any time, guaranteeing soundness. Convergence is not ensured. In this talk, we study termination and precision properties of reduction from the perspective of logic. We define the notion of refutational equivalence between a monolithic and split value as a practical way to understand when a split domain is “precise enough”. Our main result is that if the theories used in the abstract domain admit and are closed uniform interpolation, reduction can be done in one step – by computing the uniform interpolant of the formula – guaranteeing refutational equivalence. For theories (or their combination) that do not admit uniform interpolation, or are not closed under uniform interpolation, we show that if a uniform interpolant is found, the reduction procedure can be immediately stopped, guaranteeing the best precision for the partitioned domain.

3.10 An incomplete journey in Completeness


Roberto Giacobazzi (University of Verona, IT)

License  Creative Commons BY 4.0 International license
© Roberto Giacobazzi

In this talk I will present some results concerning the bridge between the theory of computation and abstract interpretation. The two theories differ profoundly in the way programs are interpreted: In the standard theory of computation program equivalence is extensional, we have compositionally and referential transparency. Abstract interpretation instead is deeply intensional, non compositional (in the sense that by composition we may lose precision). However, it is possible to view the theory of abstract interpretation from the perspective of the theory of computation. In this case a different notion of program equivalence is considered – the one defined by the equivalence of the abstract interpretations, and push the theory of abstract interpretation to its limits by studying the properties of abstraction that make an abstract interpretation closer to the standard notion of interpretation as originally defined by A. Turing. Interestingly, a analogous of Rice Theorem can be stated for abstract interpretation and a number of results can be obtained for classes of programs for which an abstract interpreter is precise (complete).

3.11 Abstract interpretation based under approximations and Sufficient Incorrectness Logic

Roberta Gori (University of Pisa, IT)

License  Creative Commons BY 4.0 International license
© Roberta Gori

Joint work of Flavio Ascari, Roberto Bruni, Roberta Gori, Francesco Logozzo
Main reference Flavio Ascari, Roberto Bruni, Roberta Gori: “Limits and difficulties in the design of under-approximation abstract domains”, in Proc. of the Foundations of Software Science and Computation Structures – 25th International Conference, FOSSACS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Lecture Notes in Computer Science, Vol. 13242, pp. 21–39, Springer, 2022.

URL https://doi.org/10.1007/978-3-030-99253-8_2

To address bug finding rather than correctness, Incorrectness Logic has been recently proposed by O’Hearn: it is based on under-approximations, thus it only reports *true* alarms. In principle, Abstract Interpretation techniques can handle under-approximations as well as over-approximations, but, in practice, few attempts were developed for the former, notwithstanding the wide literature on the latter. We aim to answer the following open question raised by O’Hearn: which role can Abstract Interpretation play in the development of under-approximate tools for bug catching? Our findings clarify, for the first time, why over- and under-approximation analysers exhibited such a different development and outline the limits of under-approximation Abstract Interpretation based analyses. Our key argument is the practical difficulty to design an effective under-approximation abstract domain able to deal with common program statements. For this reasons we investigate logics for underapproximations. We introduce Sufficient Incorrectness Logic (SIL), a new under-approximating, triple-based program logic to reason about program errors. SIL is designed to set apart the initial states leading to errors. We formally compare SIL to existing triple-based program logics.

3.12 On the fly verification with (incremental) interactive abstract interpretation

Manuel Hermenegildo (IMDEA Software Institute – Pozuelo de Alarcón, ES & UPM – Madrid, ES)

License © Creative Commons BY 4.0 International license
© Manuel Hermenegildo

Joint work of Manuel Hermenegildo, Isabel Garcia-Contreras, José F. Morales, Pedro López-García, Louis Rustenholz, Daniela Ferreira, Daniel Jurjo

Main reference Miguel A. Sanchez-Ordaz, Isabel Garcia-Contreras, Victor Perez-Carrasco, José F. Morales, Pedro López-García, Manuel V. Hermenegildo: “VeriFly: On-the-fly Assertion Checking via Incrementality”, CoRR, Vol. abs/2106.07045, 2021.

URL <https://arxiv.org/abs/2106.07045>

We demonstrated how the integration of the Ciao abstract interpretation framework within different IDEs takes advantage of our efficient and incremental fixpoint to achieve effective verification on-the-fly, as the program is developed. We also demonstrated an embedding of this framework within the browser, and how it can be used to build interactive tutorials for teaching abstract interpretation.

Further reading

The work and demo presented builds on several specific components of the Ciao abstract interpretation framework:


- **The “top-down” algorithm:** (a.k.a. the PLAI algorithm) is the fundamental component of our approach: see [4] and [5]. The latter is a step by step tutorial on how the algorithm was derived, the reasons for the different optimizations, etc. and is probably the best single reference for the original “top-down algorithm.” This first algorithm already achieves precision and efficiency through the use of memo tables; inferring call-answer pairs (aka summaries, precondition-postcondition pairs, etc.) which can be several per each procedure / block / program point (a.k.a. multivariance, path/context/call-site sensitivity, cloning, etc.); abstraction of the paths and recursions/loops in the program as graphs / regular trees; detection of SCCs; dependency tracking between memo table entries for accelerating the fixpoint (keeping track of what has to be recomputed when something changes, which is later instrumental for incremental analysis); handles procedures (projection, extension), including (mutual) recursion; etc. It is also generic in the sense that it takes care on one hand of abstracting the control: paths, dynamic CFG, procedure call and return (projection, extension), including (mutual) recursion, etc. Then, multiple ‘abstract domains’ are available as plug-ins to abstract the data: recursive heap data structures, pointer aliasing, numerical domains, etc.
- **Incremental Analysis:** After developing the PLAI algorithm we developed and benchmarked the incremental version. It was first described in [2], which was later published in longer form as [3], and an improved incremental algorithm was later described in [6]. Finally, we recently extended the ’96 algorithm to deal at the same time with modules (coarse-grained incrementality) and fine-grained incrementality within each module [1].
- **Interactive verification:** The final component is the integration of the analyzer/verifier-optimizer CiaoPP in the IDE. The latest version of this integration (“VeriFly” [8, 7]), put in practice the idea of interactive verification with abstract interpretation.

References

- 1 I. Garcia-Contreras, J. F. Morales, and M. V. Hermenegildo. Incremental and Modular Context-sensitive Analysis. *Theory and Practice of Logic Programming*, 21(2):196–243, January 2021. <https://arxiv.org/abs/1804.01839>
- 2 M. V. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Logic Programs. In *International Conference on Logic Programming*, pages 797–811. MIT Press, June 1995. https://cliplab.org/papers/incanal-iclp95_bitmap.pdf
- 3 M. V. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000. <https://cliplab.org/papers/incanal-toplas.pdf>
- 4 K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989. <https://cliplab.org/papers/abs-int-naclp89.pdf>
- 5 K. Muthukumar and M. Hermenegildo. Deriving A Fixpoint Computation Algorithm for Top-down Abstract Interpretation of Logic Programs. Technical Report ACT-DC-153-90, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, April 1990. <http://cliplab.org/papers/mcctr-fixpt.pdf>
- 6 G. Puebla and M. V. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium (SAS 1996)*, number 1145 in Lecture Notes in Computer Science, pages 270–284. Springer-Verlag, September 1996. https://cliplab.org/papers/inc-fixp-sas_bitmap.pdf
- 7 M. A. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, and M.V. Hermenegildo. VeriFly: On-the-fly Assertion Checking with CiaoPP. In *6th Workshop on Formal Integrated Development Environment (F-IDE 2021, part of NASA NFM'21)*, Electronic Proceedings in Theoretical Computer Science (EPTCS), pages 1–5. Open Publishing Association (OPA), May 2021. Co-located with ETAPS 2021. https://cister-labs.pt/f-ide2021/images/preprints/F-IDE_2021_paper_7.pdf
- 8 M.A. Sanchez-Ordaz, I. Garcia-Contreras, V. Perez-Carrasco, J. F. Morales, P. Lopez-Garcia, and M. V. Hermenegildo. Verify: On-the-fly Assertion Checking via Incrementality. *Theory and Practice of Logic Programming*, 21(6):768–784, September 2021. Special Issue on ICLP'21. <http://arxiv.org/abs/2106.07045>

3.13 Automated Reasoning for Privacy

Temesghen Kahsai (Amazon Lab 126, US)

License  Creative Commons BY 4.0 International license
© Temesghen Kahsai

Automated reasoning techniques offer an exceptional avenue to attain the utmost assurance in safeguarding data privacy. In this presentation, we will explore our adaptations of these techniques in addressing vital inquiries concerning code and cloud infrastructure, aimed at identifying potentially harmful misconfigurations. We will delve into the deployment of highly scalable static analysis for sensitive data tracking, the utilization of a provably correct differential privacy library to guarantee the safety of shared aggregated data, and the implementation of diverse pseudo-anonymization methods to safeguard sensitive information.

3.14 Abstract Interpretation in Industry – Practical Experience with Astrée

Daniel Kästner (*AbsInt – Saarbrücken, DE*)

License © Creative Commons BY 4.0 International license
© Daniel Kästner

The presentation gives a brief overview of the verification goals addressed by the Astrée analyzer, and then focuses on the development history from the ready-for-market academic version from 2009 till today’s state. Enhancements are grouped into different categories, usability, compliance to formal requirements, new capabilities, domain-specific extensions, precision improvement and optimizations for scalability. The current state is briefly summarized by experimental results on automotive integration analysis projects, taken from our 2023 SAE conference paper. We then briefly summarize “selling points” of verification tools to industry users with an emphasis on the role of safety norms: they define the minimum state of the art for system development and play a fundamental role in the adoption of tools in industrial development processes. We give examples how abstract interpretation is addressed in DO-178C and ISO 26262, observe that it is entirely missing in others, and conclude that it is under-represented in today’s safety norms. The presentation ends with listing some verification challenges we see upcoming due to current market trends.

References

- 1 D. Kästner, C. Mallon, L. Mauborgne, S. Schank, S. Wilhelm, C. Ferdinand. *Automatic Sound Static Analysis for Integration Verification of AUTOSAR Software*. SAE Technical Paper 2023-01-0591, SAE World Congress 2023, Detroit, April 2023. DOI: <https://doi.org/10.4271/2023-01-0591>

3.15 SSA Translation Is an Abstract Interpretation, and its Application to Machine Code Analysis

Matthieu Lemerre (*CEA LIST – Gif-sur-Yvette, FR*)

License © Creative Commons BY 4.0 International license
© Matthieu Lemerre

Joint work of Matthieu Lemerre, Olivier Nicole, Xavier Rival, Sébastien Bardin

Main reference Matthieu Lemerre: “SSA Translation Is an Abstract Interpretation”, Proc. ACM Program. Lang., Vol. 7(POPL), pp. 1895–1924, 2023.

URL <https://doi.org/10.1145/3571258>

Conversion to Static Single Assignment (SSA) form is usually viewed as a syntactic transformation algorithm that gives unique names to program variables, and reconciles these names using “phi” functions based on a notion of domination. We instead propose a semantic approach, where SSA translation is performed using a simple dataflow analysis. Based on a new technique to use cyclic terms in abstract domains, we propose a Symbolic Expression abstract domain that performs a Global Value Numbering analysis, upon which we build our SSA translation. This implies a shift in perspective, as global value numbering becomes a prerequisite of SSA translation, instead of depending on SSA.

One application to performing SSA Translation by Abstract Interpretation is that SSA optimizations passes can be implemented as a combination of abstract domains, allowing to perform several optimizations simultaneously to solve the usual phase ordering problem and avoiding tedious maintenance of SSA invariants.

Our main motivation for this research is an analyser for machine code which uses SSA as its main intermediate representation. Machine code is too low-level to allow SSA translation without a prior semantic analysis, while SSA is an intermediate representation that makes static analysis easier than direct analysis of machine code. Viewing SSA translation as a semantic analysis solves this chicken-and-egg problem, allowing to simultaneously decompile machine code to SSA and use the SSA representation to perform the other semantic analyses (value analysis, memory analysis, and control-flow analysis). We illustrate the use of such an analysis on an embedded OS kernel where we prove security properties directly from its executable.

References

- 1 *Matthieu Lemerre*: SSA Translation Is an Abstract Interpretation. In Principle of Programming Languages (POPL), 2023.
- 2 *O. Nicole, M. Lemerre, S. Bardin, X. Rival*: No Crash, No Exploit : Automated Verification of Embedded Kernels . In Real-time systems and applications (RTAS), 2021

3.16 A Multilanguage Static Analysis of Python/C Programs with Mopsa

Raphaël Monat (INRIA Lille, FR) and Antoine Miné (Sorbonne University – Paris, FR)

License © Creative Commons BY 4.0 International license
© Raphaël Monat and Antoine Miné

Joint work of Ouadjaout, Abdelraouf; Miné, Antoine

Main reference Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné: A Multilanguage Static Analysis of Python Programs with Native C Extensions. SAS 2021: 323-345

URL https://link.springer.com/chapter/10.1007/978-3-030-88806-0_16

Mopsa is a conservative static analysis platform, independent of language and abstraction choices. Developers are free to add arbitrary abstractions (numeric, pointer, memory, etc.) and syntax iterators for new languages. Mopsa encourages the development of independent abstractions which can cooperate or be combined to improve precision. In this talk, we will show how Mopsa analyses Python programs calling C libraries. It analyses directly and fully automatically both the Python and the C source codes. It reports runtime errors that may happen in Python, in C, and at the interface. We implemented our analysis in a modular fashion: it reuses off-the-shelf C and Python analyses written in the same analyzer. Our analyzer can tackle tests of real-world libraries a few thousand lines of C and Python long.

This talk is based on our SAS'21 paper.

3.17 Crab: A library for building abstract-interpretation-based analyses

Jorge Navas (Certora – Seattle, US)

License © Creative Commons BY 4.0 International license
© Jorge Navas

Crab is an open-source library that helps to develop static analyses based on the theory of Abstract Interpretation. It provides a variety of software components such as inter-procedural forward and backward analyses, fixpoint solvers and a rich set of abstract domains, including interfaces with numerical abstract domain libraries such as Apron, Elina or PPLite.

This tutorial focuses on demonstrating how to use Crab from the perspective of two different kinds of users: (1) the ones who want to quick prototype new abstractions by combining existing abstract domains, and (2) those who want to develop new static analyses using Crab as a black-box. We show several interactive demos on how to derive new abstractions based on the combination of an array domain with relational numerical domains to prove memory safety of eBPF programs and how to implement a new LLVM-based static analyzer to infer upper-bounds for dynamic memory allocation in less than 100 lines of C++ code.

3.18 Calculational Design of Program Logics by Abstract Interpretation

Patrick Cousot

License © Creative Commons BY 4.0 International license
© Patrick Cousot

Main reference Submitted

We study transformational program logics for correctness and incorrectness that we extend to explicitly handle both termination and nontermination. We show that the logics are abstract interpretations of the right image transformer for a natural relational semantics covering both finite and infinite executions. This understanding of logics as abstractions of a semantics facilitates their comparisons through their respective abstractions of the semantics (rather than the much more difficult comparison through their formal deductive systems). More importantly, the formalization provides a calculational method for constructively designing the sound and complete formal deductive system by abstraction of the semantics. As an example, we extend Hoare logic to cover all possible behaviors of nondeterministic programs and design a new precondition (in)correctness logics. This logic can be used to prove that false alarms in static analysis are due to the over approximation of nonterminating behaviors by terminating over approximation in the analysis which cannot be done by incorrectness or outcome logic is unreachable in the concrete (although it is reachable in the abstract).

3.19 Mentorship for Formal Methods

Ruzica Piskac (Yale University – New Haven, US)

License © Creative Commons BY 4.0 International license
© Ruzica Piskac

Joint work of Mark Santolucito, Ruzica Piskac


Main reference Mark Santolucito, Ruzica Piskac: “Formal Methods and Computing Identity-based Mentorship for Early Stage Researchers”, in Proc. of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE 2020, Portland, OR, USA, March 11-14, 2020, pp. 135–141, ACM, 2020.

URL <https://doi.org/10.1145/3328778.3366957>

The field of formal methods relies on a large body of background knowledge that can dissuade researchers from engaging with younger students, such as undergraduates or high school students. However, we have found that formal methods can be an excellent entry point to computer science research – especially in the framing of Computing Identity-based Mentorship. In this talk, we report on our experience in using a cascading mentorship model to involve early stage researchers in formal methods, covering the process with these students from recruitment to publication. We present case studies and how we were able to integrate formal methods research with the students’ own interests. We outline some key strategies that have led to success and reflect on strategies that have been, in our experience, inefficient.

3.20 Abstract Interpretation-based Program (Analysis) Logics

Francesco Ranzato (University of Padova, IT)

License  Creative Commons BY 4.0 International license
© Francesco Ranzato

Joint work of Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Francesco Ranzato
Main reference Roberto Bruni, Roberto Giacobazzi, Roberta Gori, Francesco Ranzato: “A Correctness and Incorrectness Program Logic”, J. ACM, Vol. 70(2), pp. 15:1–15:45, 2023.
URL <https://doi.org/10.1145/3582267>

We introduce a program logic, called Local Completeness Logic for an abstract domain A (LCL_A), for proving both the correctness and incorrectness of program specifications. This proof system, which is parametrized by the abstraction A , combines over- and under-approximating reasoning: in a provable triple $\vdash_A [p]c[q]$ for the program c , q is an under-approximation of the strongest post-condition of c on input pre-condition p , such that their abstractions in A coincide. If A is the straightforward abstraction making all program properties equivalent, then the logic LCL_A coincides with O’Hearn’s incorrectness logic. We discuss the pitfalls of this logic LCL_A and why it is hard to make proofs within it. We, therefore, advocate designing a weakening of LCL_A which should be able to prove properties of program analyses rather than program behaviors, in particular for proving that a program analysis is the best possible one in the underlying domain A .

3.21 VeriCode: Correct Translation of Abstract Specifications to C-Code

Gerhard Schellhorn (Universität Augsburg, DE)

License  Creative Commons BY 4.0 International license
© Gerhard Schellhorn

The talk presented the new VeriCode project (funded by German Research Foundation DFG) we just started. The project aims at the generation of efficient code from abstract specification. Such specifications are typical when using higher-order logic and abstract programs to specify functions. They typically use abstract datatypes which are easily translated to functional code. However the resulting code is typically not very efficient and requires garbage collection. The project was motivated by an earlier DFG project called Flashix that produced a verified file system for flash memory. A simple code generator is already implemented that produces C- and Scala-Code, based on the principle that in contrast to a functional implementation data structure should not share. The approach enables destructive updates and allows to avoid garbage collection. The approach still causes too much copying and the talk showed some first optimizations. Compared to a native implementation of a flash filesystem (UBIFS) in C that we used as a blueprint, our code is still some factors slower and the project aims to close the gap. The talk also showed the overall approach of the project, which is not just to implement an efficient code generator, but to verify that it is correct again using specifications of the relevant functionality and semantics. Ultimately this should allow to bootstrap the code generator by again generating code from the specification of its functionality.

3.22 Data Race Repair using Static Analysis Summaries

Ilya Sergey (National University of Singapore, SG)

License © Creative Commons BY 4.0 International license
© Ilya Sergey

Joint work of Andreea Costea, Abhishek Tiwari, Sigmund Chianasta, Kishore R, Abhik Roychoudhury, Ilya Sergey
Main reference Andreea Costea, Abhishek Tiwari, Sigmund Chianasta, Kishore R, Abhik Roychoudhury, Ilya Sergey: “Hippodrome: Data Race Repair Using Static Analysis Summaries”, ACM Trans. Softw. Eng. Methodol., Vol. 32(2), pp. 41:1–41:33, 2023.
URL <https://doi.org/10.1145/3546942>

Implementing bug-free concurrent programs is a challenging task in modern software development. State-of-the-art static analyses find hundreds of concurrency bugs in production code, scaling to large codebases. Yet, fixing these bugs in constantly changing codebases represents a daunting effort for programmers, particularly because a fix in the concurrent code can introduce other bugs in a subtle way.

In this talk, I will show how to harness compositional static analysis for concurrency bug detection, to enable a new Automated Program Repair (APR) technique for data races in large concurrent Java codebases. The key innovation of our work is an algorithm that translates procedure summaries inferred by the analysis tool for the purpose of bug reporting into small local patches that fix concurrency bugs (without introducing new ones). This synergy makes it possible to extend the virtues of compositional static concurrency analysis to APR, making our approach effective (it can detect and fix many more bugs than existing tools for data race repair), scalable (it takes seconds to analyse and suggest fixes for sizeable codebases), and usable (generally, it does not require annotations from the users and can perform continuous automated repair). Our study conducted on popular open-source projects has confirmed that our tool automatically produces concurrency fixes similar to those proposed by the developers in the past.

3.23 Property-Directed Reachability as Abstract Interpretation in the Monotone Theory

Sharon Shoham Buchbinder (Tel Aviv University, IL)

License © Creative Commons BY 4.0 International license
© Sharon Shoham Buchbinder

Joint work of Yotam M. Y. Feldman, Mooly Sagiv, Sharon Shoham Buchbinder, Mooly Sagiv
Main reference Yotam M. Y. Feldman, Mooly Sagiv, Sharon Shoham, James R. Wilcox: “Property-directed reachability as abstract interpretation in the monotone theory”, Proc. ACM Program. Lang., Vol. 6(POPL), pp. 1–31, 2022.
URL <https://doi.org/10.1145/3498676>
Main reference Yotam M. Y. Feldman, Sharon Shoham: “Invariant Inference with Provable Complexity from the Monotone Theory”, in Proc. of the Static Analysis – 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings, Lecture Notes in Computer Science, Vol. 13790, pp. 201–226, Springer, 2022.
URL https://doi.org/10.1007/978-3-031-22308-2_10

Inferring inductive invariants is one of the main challenges of formal verification. One of the latest breakthroughs in invariant inference is property-directed reachability (IC3/PDR). In this talk, we utilize the rich theory of abstract interpretation to shed light on the overapproximation of the reachable states performed by PDR’s frames. Namely, we define an eager version of PDR, called Lambda-PDR, in which all generalizations of counterexamples are used to strengthen a frame, and show that its frames can be formulated as an abstract interpretation algorithm in a logical domain based on Bshouty’s monotone theory. Since the

frames of Lambda-PDR are tighter than the frames of PDR, the same overapproximation, and more, is present in PDR's frames. We demonstrate that this overapproximation can result in an exponential gap compared to exact forward reachability.

3.24 Exploiting Pointer Analysis in Memory Models for Deductive Verification

Mihaela Sighireanu (ENS Paris-Saclay – Gif-sur-Yvette, FR)

License © Creative Commons BY 4.0 International license

© Mihaela Sighireanu

Joint work of Quentin Bouillaguet, François Bobot, Mihaela Sighireanu, Boris Yakobowski

Main reference Quentin Bouillaguet, François Bobot, Mihaela Sighireanu, Boris Yakobowski: “Exploiting Pointer Analysis in Memory Models for Deductive Verification”, in Proc. of the Verification, Model Checking, and Abstract Interpretation – 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings, Lecture Notes in Computer Science, Vol. 11388, pp. 160–182, Springer, 2019.

URL https://doi.org/10.1007/978-3-030-11245-5_8

Cooperation between verification methods is crucial to tackle the challenging problem of software verification. The paper focuses on the verification of C programs using pointers and it formalizes a cooperation between static analyzers doing pointer analysis and a deductive verification tool based on first order logic. We propose a framework based on memory models that captures the partitioning of memory inferred by pointer analyses, and complies with the memory models used to generate verification conditions. The framework guided us to propose a pointer analysis that accommodates to various low-level operations on pointers while providing precise information about memory partitioning to the deductive verification. We implemented this cooperation inside the Frama-C platform and we show its effectiveness in reducing the task of deductive verification on a complex case study.

3.25 (Un-)Realizability of Condition Synthesis as CHC-SAT

Yakir Vizel (Technion – Haifa, IL)

License © Creative Commons BY 4.0 International license

© Yakir Vizel

Joint work of Yakir Vizel, Bat-Chen Rothenberg, Orna Grumberg

Condition synthesis takes a program in which some of the conditions in conditional branches are missing, and a specification, and automatically infers conditions to fill-in the holes such that the program meets the specification.

In this talk, we present COSYN [1], an algorithm for determining the realizability of a condition synthesis problem, with an emphasis on proving unrealizability efficiently. COSYN is based on a reduction of the condition synthesis problem to satisfiability of Constrained Horn Clauses (CHCs). In order to allow this reduction to CHCs, we use the novel concept of a *doomed* initial state, which is an initial state that can reach an error state along *every* run of the program. For a doomed initial state σ , there is no way to make the program safe by forcing σ (via conditions) to follow one computation or another. COSYN encodes the existence of a doomed initial state as CHCs.

COSYN is implemented in SEAHORN using SPACER as the CHC solver and evaluated it on multiple examples. The evaluation shows that COSYN outperforms the state-of-the-art syntax-guided tool CVC5 in proving both realizability and unrealizability. Evaluation also shows that joining forces of COSYN and CVC5 outperforms CVC5 alone, allowing to solve more instances, faster.

References

- 1 B. Rothenberg et al., *Condition Synthesis Realizability via Constrained Horn Clauses*. NASA Formal Methods – 15th International Symposium, Houston, TX, USA, May 16-18, 2023.

3.26 Dataflow Refinement Type Inference

Thomas Wies (New York University, US)

License © Creative Commons BY 4.0 International license
© Thomas Wies

Joint work of Zvonimir Pavlinovic, Yusen Su, Thomas Wies

Main reference Zvonimir Pavlinovic, Yusen Su, Thomas Wies: “Data flow refinement type inference”, Proc. ACM Program. Lang., Vol. 5(POPL), pp. 1–31, 2021.

URL <https://doi.org/10.1145/3434300>

Refinement types enable lightweight verification of functional programs. Algorithms for statically inferring refinement types typically work by reduction to solving systems of constrained Horn clauses extracted from typing derivations. An example is Liquid type inference, which solves the extracted constraints using predicate abstraction. However, the reduction to constraint solving in itself already signifies an abstraction of the program semantics that affects the precision of the overall static analysis. To better understand this issue, we study the type inference problem in its entirety through the lens of abstract interpretation. We propose a new refinement type system that is parametric with the choice of the abstract domain of type refinements as well as the degree to which it tracks context-sensitive control flow information. We then derive an accompanying parametric inference algorithm as an abstract interpretation of a novel data flow semantics of functional programs. We further show that the type system is sound and complete with respect to the constructed abstract semantics. Our theoretical development reveals the key abstraction steps inherent in refinement type inference algorithms. The trade-off between precision and efficiency of these abstraction steps is controlled by the parameters of the type system. Existing refinement type systems and their respective inference algorithms, such as Liquid types, are captured by concrete parameter instantiations.

3.27 Timing Analysis by Abstract Interpretation

Reinhard Wilhelm (Universität des Saarlandes – Saarbrücken, DE)

License © Creative Commons BY 4.0 International license
© Reinhard Wilhelm

Main reference Reinhard Wilhelm: “Real time spent on real time”, Commun. ACM, Vol. 63(10), pp. 54–60, 2020.

URL <https://doi.org/10.1145/3375545>

Hard real-time systems need a proof that they keep their deadlines. This proof is produced by a code-level WCET analysis and a schedulability analysis for a set of tasks to be executed on the same platform. A code-level WCET analysis computes a safe upper bound for all

execution times of a task. The only sound WCET analysis used widely in industry, developed by my group in Saarbrücken, uses several abstract interpretations to safely and efficiently derive such upper bounds for tasks to be executed on single-core platforms. Central for the practicality of our approach are adequate abstractions of the architecture of the execution platform. Thus, the architecture is an integral part of the semantics from which abstract interpretations of real-time programs are derived. This entails a number of peculiarities compared to more conventional abstract interpretations. They are more concerned with the occupancy of platform resources than with the values contained in those resources. The occupancy of resources, e.g. the cache contents or the usage of bus bandwidth influence the timing behavior. Iteration over loops, as part of the fixed-point iteration, needs to consider machine parameters to achieve accuracy. The first iteration of a loop typically loads the cache, and later iterations profit from this cache loading. Therefore, first and non-first iterations may have vastly different execution times. In order to obtain accurate execution times for loops, the iteration of the analysis first needs to stabilize the execution time of the loop body. In my talk I gave some such examples of peculiarities [2]. The main part was concerned with the development history of our WCET-analysis technology [1].

References

- 1 Reinhard Wilhelm. *Real time spent on real time*. Commun. ACM 63(10): 54-60 (2020)
- 2 Jan Reineke and Reinhard Wilhelm. *Static Timing Analysis – What is Special?*. Semantics, Logics, and Calculi 2016: 74-87

4 Working groups

4.1 Soundness requirements, transparency of assumptions

Bor-Yuh Evan Chang (University of Colorado – Boulder, US) and Raphaël Monat (INRIA Lille, FR)

License © Creative Commons BY 4.0 International license
© Bor-Yuh Evan Chang and Raphaël Monat

We held a discussion session around soundness requirements, and the transparency of assumptions made to ensure an analysis or a tool is sound.

The starting point of this discussion was the Soundness manifesto [1]. The goal of this manifesto is to raise awareness and highlight that there likely are – at least small and sometimes intentional – discrepancies between the formalized concrete semantics and the actual implementation. This might be due to the concrete semantics being formalized in a research paper not matching the whole semantics of the targetted language – either to simplify the presentation, make a trade-off in the analysis, or because modern programming languages are large and complex. However, the soundness paper is sometimes misunderstood and used as an excuse to give up on establishing soundness results. Thus, we argue (and believe it to be in line with the original intent of the soundness manifesto) for transparency in soundness claims: researchers should state and make explicit the limitations of their soundness claims, and in particular the assumptions that are being made. The reasonableness of those assumptions should be empirically evaluated whenever possible.

In order to ensure this transparency at the tool level, we suggest tools should report the assumptions they made alongside the properties they have been able – or unable – to prove.

For example, let us consider a C program where an extern function is called. For early works, it might be sufficient to abort the analyses when encountering such cases. Then, a reasonable approach to analyze this program is to assume the function does not have side-effects and that it returned a value abiding by its return type signature. Instead of silently performing this assumption and continuing the analysis, we argue that for transparency, this assumption must be stored and displayed during the analyzer's report.

As another example, consider in Java, a program can modify the fields of an object using reflection. Because perhaps such uses of reflection are rare in the code of interest or that it is considered bad practice, a concrete semantics that models arbitrary field updates at any point using reflection would lead to unrealistically imprecise and pessimistic analyses. Thus, a reasonable approach is to make the assumption there is no reflective field update and exclude it from the concrete semantics. Transparency means checking for the potential reflective field update on the runs of the analysis are reporting it if it is encountered. And transparency means empirically evaluating the prevalence of reflective field update on a representative corpus.

This approach is being prototyped within the Mopsa static analyzer for its C, Python and multilanguage analyses. We believe it may also help developers better understand the outputs of static analyzers.

Thanks to this approach, the assumptions made by a static analyzer are now clearly defined in the implementation. In turn, this simplifies establishing theoretical soundness claims on paper: it is easier to list the current limitations, and soundness assumptions of a given analysis in practice.

Of course, developers of commercial static analyzers that are used to certify safety-critical systems with respect to regulations such as DO-178C are deeply aware of this transparency-based approach. They internally have documents describing the trusted computing base, and the exact soundness theorem, with all potential assumptions. While this is too time-consuming and stringent for proof-of-concepts and early academic software, we believe both that it is an ideal and a must for usable tools.

Our next step is to discuss with the authors of the original soundness manifesto. It was suggested in our plenary recap and discussion to push for an evaluation of the transparency of static analysis tools in selected conferences performing artifact evaluation.

References

- 1 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, Dimitrios Vardoulakis: In defense of soundness: a manifesto. *Commun. ACM* 58(2): 44-46 (2015)

4.2 Expressive Domains

Gidon Ernst (LMU München, DE)


License © Creative Commons BY 4.0 International license
© Gidon Ernst

We discussed abstract domains for data structures and domains that can capture complex properties. We observed that a large variety of techniques have been described in the literature, e.g., for the data structures sequences/strings, sets, multisets, arrays, pointer-structures, and trees/general ADTs, and which e.g., can express properties like sortedness or initialization. However, while for numerical domains, we have nice open-source libraries that

can easily be embedded into larger use-cases, it was noted that this is hardly the case for such expressive domains despite the fact that there are many implementations. We then discussed the challenges in the design of both the domains itself as well as a uniform interface that could capture their features and commonalities: The first challenge is to define a common interface in the first place. Since domains are usually designed to tackle specific properties, the operations they offer are tuned to their respective use cases and therefore not uniform. As an example, it seemed unclear whether it is possible to represent the features offered by domains as part of a CHC solver. As possible future work, a first step towards usable off-the-shelf libraries was to investigate the design of a suitable API. A second challenge is the combined use of abstract domains, for example when nesting domains for element types inside container types. There is no canonical choice to do so many, and which is appropriate strongly depends on the application. This kind of integration often relies on the tight integration of the data structures representing the abstract domains at the implementation level. This kind of compositionality, however, is hard to achieve general because domains – even for the same purpose – may rely on rather different techniques. A related problem is how to provide on-line transfer from and to symbolic representations. We identified as a question for the efficient combination of domains, whether some of the higher-level domains benefit from some specialized operators of the underlying abstract domains. The third challenge is that it is hard to design abstract domains that are robust. A lot of knowledge goes into selecting the right abstraction for a given (sub-)problem and choosing the an inappropriate one incurs high computational cost. Unfortunately, this is hard to determine automatically, and moreover may rely on user-provided partial specifications as guidance.

4.3 Education: Teaching Abstract Interpretation to the Masses

Pietro Ferrara (University of Venice, IT)

License  Creative Commons BY 4.0 International license
© Pietro Ferrara

During this working group, the discussion was divided into two main themes: teaching to graduate students, and to undergraduates or, generally speaking, practitioners. In the first case, a lot of good material is available. A non-comprehensive list (that should be further integrated) to the best of the participants' knowledge is the following one: books [1, 2]; slides [3, 4, 5, 6, 7, 8]; tools [9, 10].

For undergrads and/or practitioners there is less material available. After an open discussion with all the seminar participants, we were able to assemble the following list: [11, 12, 13].

Generally speaking, the first idea would be to augment undergraduate courses on the implementation of interpreters with the idea of abstraction (e.g., [7]). Another proposal was to design a generic interface to interact (ranging from the application to the extension) with static analyzers through a unique library, avoiding having analyzers that require being bound to specific programming languages. This is a rather standard approach adopted by our communities that over the years achieved better visibility and popularity (e.g., the machine learning community with libraries such as Scikit-learn and Pytorch, or the theorem-proving community with libraries such as SMT-lib).

Overall, the participants agreed that the community is missing a place where educational material about static analysis can be shared, as well as introduction courses in popular MOOC platforms.

References

- 1 Patrick Cousot: Principles of Abstract Interpretation, <https://mitpress.mit.edu/9780262044905/principles-of-abstract-interpretation/>
- 2 Rival and Yi: Introduction to Static Analysis, <https://mitpress.mit.edu/9780262043410/introduction-to-static-analysis/>
- 3 Patrick Cousot's course at MIT: <http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/>
- 4 Feret, Giet, Rival course at ENS Paris: <https://www.di.ens.fr/~rival/semverif-2023/>
- 5 Miné, Urban, Feret, Rival in Paris (MPRI): <https://www-apr.lip6.fr/~mine/enseignement/mpri/current/>
- 6 Ferrara (old - 2012) course at ETH: <https://ethz.ch/content/dam/ethz/special-interest/infk/chair-program-method/pm/documents/Education/Courses/SS2012/SPA/Lectures.zip>
- 7 Evan course at the University of Colorado Boulder: <https://csci3155.cs.colorado.edu/f22/> and <https://github.com/csci3155/>
- 8 Jan Midtgaard: Abstract Interpretation (2015 Winter School) <https://janmidtgaard.dk/aiws15/>
- 9 LiSA (Ca' Foscari University of Venice): <https://github.com/lisa-analyzer/lisa>
- 10 MOPSA (Sorbonne Université): <https://gitlab.com/mopsa/mopsa-analyzer>
- 11 Anders Møller and Michael I. Schwartzbach: Static Program Analysis <https://cs.au.dk/~amoeller/spa/>
- 12 Manuel's older intro to AI, and AI for (C)LP (a tutorial from the early 90's. but we still use it sometimes) https://cliplab.org/logalg/slides/B_ai.pdf
- 13 Manuel: Some tutorials on PLAI/CiaoPP using our analyzers embedded in web pages (work in progress) https://ciao-lang.org/ciao/build/doc/ciaopp_tutorials.html/

4.4 Community Infrastructure – Interest Group in Static Analysis

Pietro Ferrara (University of Venice, IT)

License © Creative Commons BY 4.0 International license
© Pietro Ferrara

The main outcome of this working group was a proposal to build up a community infrastructure to facilitate the promotion of events in our community, stimulate interactions, and provide better visibility of the main outcomes in our field. In particular, currently, we have various groups on social networks (such as Facebook and LinkedIn) that are mostly silent. However, every year we have a series of regular events that help the networking activities of our community, such as conferences like SAS and VMCAI, workshops like SOAP, and events organized by various institutions such as the “Dependable and Secure Software Systems” workshop at ETH or the Challenges of Software Verification Symposium at Ca' Foscari University of Venice (just to name a few the participants were aware of, but such a list should be quite expanded).

The final proposal of the working group was to: (i) build up a website that contains information about events and materials about static analysis (taking inspiration from <https://microservices.community>), (ii) open a mailing list about announcements regarding scientific activities and opportunities in static analysis, and (iii) establish an Interest Group in Static Analysis (IGSA) with an advisory board that supervise the overall process. Potentially this might become an ACM Special Interest Group, an IFIP Working Group, or something else, based on the success of the initiative.

4.5 Abstract Interpretation and Other Formal Methods

Arie Gurfinkel (University of Waterloo, CA)

License  Creative Commons BY 4.0 International license
© Arie Gurfinkel

This working group discussed potential new connections between Abstract Interpretation and other Formal Methods. Based on the interests and expertise of the participants, the other formal methods that the group focused on were: Deductive Verification, Dynamic Techniques, Deduction, Model Checking, and Practical and Industrial applications.

For deductive verification, there is a desire for Abstract Interpretation to infer complex specifications, especially in the context of memory analysis. Currently, most developed Abstract Interpretation techniques are geared towards numeric domains that are not sufficiently structurally complex (i.e., no quantifiers, no memory separation, etc.). Deductive verification also requires very reliable tools that reliably provide an answer since such tools involve a direct and active interaction with a user.

For dynamic techniques, there is a desire for better abstract domains for strings. Perhaps based on the deep connection between automata and regular languages. There is also a potential in using dynamic analysis to infer context for Abstract Interpretation-based static analysis. This, for example, is already used in debloating projects such OCCAM at SRI. The group felt that this combination might also create new challenges to maintaining soundness of the analysis and/or articulating the soundness conditions clearly.

For deduction, it was discussed that the main challenge is to identify an insight for why a proof works or fails. In this context, a complex but hard-to-understand specification, such as an automatically generated inductive invariant, is not very helpful. Perhaps the results of Abstract Interpretation can be presented in a more readable form by using some pre-defined set of predicates.

For model checking, there is an interest in building Abstract Interpretation models that explain core model checking algorithms such as IC3 and PDR. There is recent work on this subject, but the group agrees that better understanding is required to capture the many nuances of these algorithms. There is also a potential application in using Abstract Interpretation to infer temporal specifications.

Group members from industry, were interested in Abstract Interpretation for probabilistic programs, with application to differential privacy. There is also interest in Abstract Interpretation in aid of property-based testing.

Overall, the working group concluded that there is a lot of interest in both applications of Abstract Interpretation in different domains, and in research on challenges that are posed by these domains. The group members from Deductive Verification have been most vocal, and, perhaps the combination of Abstract Interpretation and Deductive Verification is most suited for a follow up seminar.

4.6 Tools and Applications for Abstract Interpretation

Falk Howar (TU Dortmund, DE)

License © Creative Commons BY 4.0 International license
© Falk Howar

We had a discussion on challenges that developers of research tools must address to enable industrial collaborations and on possible working directions for the research community to foster the development of robust and scalable research tools. Among the attendees were researchers, academic tools developers, as well as developers of commercial tools and industrial researchers from companies that use formal methods tools in their product development, presenting both, academic perspectives, and insights from industry.

The initial focus of the discussion was on the potential and challenges of using research tools effectively in real-world scenarios and started with the observation that it is often difficult to use research tools in industrial collaborations.

Lessons learned by academic developers included the following points:

Usefulness: It is important to have findings and to make these findings accessible to project partners, who typically are not experts in a particular formal method or a particular research tool.

Robustness: It is important that tools can be used in industrial settings. This often requires operation in an automated analysis pipeline, e.g., a build system. In such a setting, an analysis will not be run manually executed one target but is executed as part of a build process.

Participants from companies complemented these observations with the following points:

Helpful features: To support robustness, certain features help applicability that are (usually) not important in an academic context: since industrial codebases in many cases cannot be extended with annotations for a tool, it is important to support declarative configurations and external annotations. Usefulness can be increased by support for problem extraction (i.e., interpretation / mapping of results to code) and by providing information on relationships between alarms (i.e., root causes).

Project formats: Analyzing an industrial codebase is not an easy task, especially when done by someone who is not a contributor to that codebase. Typically, success hinges on support from developers, access to architecture documentation, and on communication between maintainers and researchers. Project formats should accommodate these success factors.

Licenses: Copyleft licenses are a big obstacle in industry if code is under such a license and the industrial partner must look at the code (e.g., to understand how a tool work).

Eventual Payoff: It is important for industrial partners to understand if and what the eventual payoff can be when using a technique or tool. As a concrete successful example, the technique of unit proving was mentioned. In the experience of one participant, maintainability and conformance to low level specs seems to be much easier with unit proving than with testing once it is established in the development process.

After the initial collection of lessons learned, discussion evolved around dealing with theoretical limitations of techniques and (potential) usability requirements.

Dealing with Limitations: Commercial tool developers (AbsInt and Astree tools) reported that in their experience mostly scalability and precision are bottlenecks and that proofs tend to work on slices, but do not work on whole programs. It was discussed if and to what extent a target can be modified in order to scale or work around limitations (e.g., loop unrolling, transformations at the IR level). Consensus was that, while it is usually not


possible to change a customer’s code, transformations typically happen during analysis (with some limitations in safety-critical domains). It was deemed important that the result/verdict of an analysis can be explained to the user and in terms of the original code.

Usability: There was no consensus on or shared understanding of usability requirements. Most participants agreed that a user interface is not a strong requirement for a research tool, but that usability is still important in the sense that users need to be able to understand the findings of a tool (e.g., based on CFGs).

The discussion concluded with plans for actions the research community could take to recognize and facilitate the development of robust research tools. Case studies were identified as a relevant format: It shows potential users how a tool can be used on a realistic example and forces developers to invest in robustness and applicability. Examples of existing case studies and publicly available systems included the effort to verify the Linux kernel, medical systems (e.g., infusion pumps, a pacemaker), and a wheel-break system. Participants agreed that it will be important to promote case study papers in academic conferences. Published case studies should include a public repository with the problem, corresponding artifacts (e.g., documentation), and an experience report.

4.7 Hyperproperties verification

Isabella Mastroeni (University of Verona, IT)

License  Creative Commons BY 4.0 International license
© Isabella Mastroeni

Hyperproperties are intended as sets of sets of traces, for a more general point of view, we can see them as sets of properties. A standard example of hyperproperty is non-interference, a 2-safety property that requires to compare pairs of executions. The question for stimulating the discussion was about the right implication formalization for hyperproperties. In the literature we mainly find two different approaches: (1) The first one allows to add new elements in the hyper set, but all the elements in the stronger property must be in the weaker one, more formally, let A, B, C, D, E sets, then $\{A, B, C\}$ implies $\{A, B, C, D, E\}$; (2) the second one consists in allowing to approximate the inner elements without adding any new element, formally let A, B, C, A', B', C' be sets such that $A \subseteq A'$, $B \subseteq B'$ and $C \subseteq C'$, then $\{A, B, C\}$ implies $\{A', B', C'\}$ but $\{A, B, C\}$ does not imply $\{A', B', C', D'\}$ where D' does not contain any of A, B, C . Hence, which of these orders properly capture the approximation order between hyperproperties.

There are several approaches to noninterference that can help in understanding the issue, in particular those based on hyper analysis of programs. But also there are other relevant formal approaches to hyper properties verification, such as the hyper Hoare logic, useful for understanding the issue.

By reasoning on both these formal approaches, the discussion ended up understanding that probably there is no a right or better order, in particular (2) is necessary for computing fixpoints programs/properties in an hyper levels both in the analysis and in the logic, while (1) is used for deciding whether a property imply another one, again both in the analysis and in the logic (consequence rule). Hence, the conclusion was that (1) is the approximation order while (2) is the computational order, which maybe must co-exists when dealing with hyper property verification. We believe that this final observation, even if it may appear quite “simple”, may represent anyway an important first step for really understanding the

connection between standard approaches to static analysis and the emerging hyper property verification issue. All participants agree that surely further understanding of the problem is necessary.

4.8 AI for AI

Antoine Miné (Sorbonne University – Paris, FR)

License © Creative Commons BY 4.0 International license
© Antoine Miné

A small working group entitled “AI for AI” discussed existing and possible future interactions between Abstract Interpretation (AI) and Machine Learning (ML). Both directions, AI to help ML and ML to help AI, were considered.

We started with the observation that, with the progress and increasing use of ML systems, ensuring their correctness also became an increasing concern. Since 2015, the field of formal verification techniques applied to ML systems has been growing steadily. A large amount of work targets the verification of trained neural networks, but other models are considered as well (e.g., support vector machines, ensemble trees). Moreover, many kinds of formal methods have been adapted to work on ML systems, including: SMT solving, constraint solving, optimisation, model-checking, Abstract Interpretation. Notable works on Abstract Interpretation for ML models include the developments of dedicated numeric abstract domains able to prove local robustness properties and fairness properties. A more detailed review of recent work can be found in Urban et al. 2021 [1]. However, given the pace of the research in this area, such reviews become outdated very quickly.

We then discussed what we perceived as some of the current limitations of AI methods for ML and possible area of improvements.

- The verification of robustness has been successful for models of moderate size, and for local robustness against perturbation and adversarial attacks. Future challenge concern more global robustness properties. Numeric abstractions and partitioning techniques from AI could be key to achieve scalability.
- ML research has recently focused on interpretability and explainability of ML models. This constitutes an interesting opportunity to apply formal verification and obtain sound and automatic guarantees on these properties. We postulate that AI-specific techniques, such as relational abstractions, might help in evaluating out inputs influence the outcome of a ML model.
- In general, however, a key difficulty for formal verification (including Abstract Interpretation) is the lack of formal specification of the properties to prove. This is an area that must be improved in order to unlock further progress.
- Most works in ML ignore the effect of floating-point rounding errors. AI techniques exist to soundly verify floating-point programs and to evaluate the effect of rounding errors formally. Applying such techniques to ML models could prove useful.

The working group also considered how ML could help AI. An important aspect is to keep the formal guarantees of the combined AI and ML method by maintaining soundness, even in the absence of soundness guarantees for the ML part.

- One possible use of ML in AI is the automatic parameterization of static analyzers to optimize the efficiency and precision of the analysis within the parameter space (e.g., choice of abstract domains, level of relationality and sensitivity, etc.).

- A specific aspect of AI is the use of widening operators to automatically infer complex invariants. The heuristic nature of widenings makes them an interesting target for ML techniques: the heuristics “guessing” candidate invariants do not need strong soundness guarantees, given that candidate invariants can be checked with a classic, sound stability tests. This idea could be extended to the inference of contract pre/post-conditions to unlock efficient modular analyses.

As concluding remark, we observed that early research in formal verification of ML was mainly conducted by formal method researchers, adapting on ML models the techniques they were familiar with for the verification of software. Given the substantial differences between ML models and programs, and the need to extract and formalize the properties we are interested to verify on ML models, we believe that progress in this field requires a tighter collaboration of experts in ML and experts in AI.

References

- 1 C. Urban and A. Miné. A review of formal methods applied to machine Learning. Technical report, Computing Research Repository (arXiv) (CoRR), Apr. 2021. <http://www-apr.lip6.fr/~mine/publi/article-urban-mine-mlfm2021.pdf>.

Participants

- Vincenzo Arceri
University of Parma, IT
- Musard Balliu
KTH Royal Institute of
Technology – Stockholm, SE
- Anna Becchi
Bruno Kessler Foundation –
Trento, IT
- Dirk Beyer
LMU München, DE
- Bor-Yuh Evan Chang
University of Colorado –
Boulder, US
- Patrick Cousot
New York University, US
- David Delmas
Airbus S.A.S. – Toulouse, FR
- Gidon Ernst
LMU München, DE
- Pietro Ferrara
University of Venice, IT
- Eden Frenkel
Tel Aviv University, IL
- Isabel Garcia-Contreras
University of Waterloo, CA
- Roberto Giacobazzi
University of Verona, IT
- Roberta Gori
University of Pisa, IT
- Arie Gurfinkel
University of Waterloo, CA
- Reiner Hähnle
TU Darmstadt, DE
- Ben Hermann
TU Dortmund, DE
- Manuel Hermenegildo
IMDEA Software Institute –
Pozuelo de Alarcón, ES & UPM –
Madrid, ES
- Falk Howar
TU Dortmund, DE
- Daniel Kästner
AbsInt – Saarbrücken, DE
- Temesghen Kahsai
Amazon Lab 126, US
- Matthieu Lemerre
CEA LIST – Gif-sur-Yvette, FR
- Isabella Mastroeni
University of Verona, IT
- Antoine Miné
Sorbonne University – Paris, FR
- Raphaël Monat
INRIA Lille, FR
- Peter Müller
ETH Zürich, CH
- Jorge Navas
Certora – Seattle, US
- Marie Pelleau
Université Côte d’Azur –
Sophia Antipolis, FR
- Ruzica Piskac
Yale University – New Haven, US
- Francesco Ranzato
University of Padova, IT
- Gerhard Schellhorn
Universität Augsburg, DE
- Ilya Sergey
National University of
Singapore, SG
- Sharon Shoham Buchbinder
Tel Aviv University, IL
- Mihaela Sighireanu
ENS Paris-Saclay –
Gif-sur-Yvette, FR
- Yakir Vizel
Technion – Haifa, IL
- Thomas Wies
New York University, US
- Reinhard Wilhelm
Universität des Saarlandes –
Saarbrücken, DE
- Enea Zaffanella
University of Parma, IT

