Report from Dagstuhl Seminar 25421

Sound Static Program Analysis in Modern Software Engineering

Pietro Ferrara[*1], Liana Hadarean[*2], Jorge A. Navas[*3], Caterina Urban[*4], and Greta Dolcetti[†5]

1 **University of Venice, IT.** pietro.ferrara@unive.it
2 **Amazon Web Services – Seattle, US.** hadarean@amazon.com
3 **Certora – Seattle, US.** jorge@certora.com
4 **Inria & ENS Paris, FR.** caterina.urban@inria.fr
5 **University of Venice, IT.** greta.dolcetti@unive.it

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 25421 "Sound Static Program Analysis in Modern Software Engineering".

Sound Static Program Analysis (SSPA) has historically been effective in proving the absence of runtime errors and security vulnerabilities, notably in safety-critical embedded software. However, it has seen limited adoption in desktop applications until its revival for Web application security (e.g., to prove the absence of SQL injection vulnerabilities). Modern software development, characterized by architectures like microservices, serverless computing, and the increasing use of scripting languages, presents challenges to SSPA due to the integration of multiple languages and complex semantics, while also posing new problems related to soundness, precision, and scalability stemming from the machine learning revolution in code development. Despite these new opportunities for SSPA to offer structured feedback and address serious flaws often overlooked by the shallow analyses currently favored by the industry, there has not been a significant resurgence in its industrial application. This Dagstuhl Seminar aimed to bridge the SSPA and software engineering communities to extend existing theories to these new trends, foster integration with modern practices like DevOps, and discuss the formal methods challenges arising from contemporary software architectures.

Seminar October 12–17, 2025 – https://www.dagstuhl.de/25421

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Software and its engineering → Formal methods; Software and its engineering → Formal software verification; Theory of computation → Program analysis; Software and its engineering → Software verification

Keywords and phrases Abstract interpretation, Formal methods, Software engineering, Software verification, Sound static program analysis

Digital Object Identifier 10.4230/DagRep.15.10.37

[*] Editor / Organizer
[†] Editorial Assistant / Collector

Sound Static Program Analysis in Modern Software Engineering, *Dagstuhl Reports*, Vol. 15, Issue 10, pp. 37–74
Editors: Pietro Ferrara, Liana Hadarean, Jorge A. Navas, Caterina Urban, and Greta Dolcetti

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

1 Executive Summary

Pietro Ferrara (University of Venice, IT)
Liana Hadarean (Amazon Web Services – Seattle, US)
Jorge A. Navas (Certora – Seattle, US)
Caterina Urban (INRIA & ENS Paris, FR)

Motivation

Sound static program analysis (SSPA) applies formal methods to prove the absence of software defects, including runtime errors and security vulnerabilities. Although its foundational theories – most notably abstract interpretation – were developed nearly half a century ago, their practical adoption has been uneven. SSPA has become indispensable in safety-critical embedded systems, where software failures can have catastrophic consequences, yet it has historically seen limited uptake in desktop and Web applications. This divide has begun to narrow with the growing prominence of Web security threats, such as SQL injection attacks, and the rapid expansion of the Internet of Things (IoT). In today's software ecosystem – dominated by microservices, serverless architectures, and dynamically typed scripting languages such as Python – SSPA faces a renewed opportunity. Beyond bug prevention, it can offer structured, actionable feedback to a much broader community of developers, including practitioners without expertise in formal verification.

Despite these opportunities, SSPA faces significant challenges regarding the shift toward machine-learning-driven code generation and the complexity of multi-language architectures. Currently, many industrial players rely on shallow, syntactic analyses that leave systems vulnerable to serious flaws. To address this gap, this Dagstuhl Seminar "Sound Static Program Analysis in Modern Software Engineering" (25421) aimed at bridging the SSPA and software engineering scientific communities to adapt formal theories to modern trends. A primary focus was the integration of sound analysis into standard software development lifecycles and DevOps practices.

Summary of Seminar Activities

The Dagstuhl Seminar "Sound Static Program Analysis in Modern Software Engineering" (25421) was attended by 36 researchers, including both senior and junior participants from academia and industry, including graduate students, faculty members, and industry experts. At the beginning of the seminar, each participant briefly introduced themselves and outlined their research interests in a two-minute presentation.

The program interleaved 26 talks (Section 3), two main breakout discussion sessions identifying and discussing open scientific problems (Section 4), and a "speed-dating" session in which participants were divided into *Theory* and *Practice* groups and engaged in a series of short, one-to-one discussions with members of the other group, rotating partners every 10 minutes.

Two invited talks provided perspectives from different points of view at the beginning of the seminar. In particular, Patrick Cousot discussed "Eternal Problems Never or Hardly Solved in Static Analysis by Abstract Interpretation" (Section 3.5), while Davide Taibi tackled "Static Analysis in the Cloud-Native Era" (Section 3.24). Most participants then

illustrated their scientific progress in talks divided into sessions on security, data science programs, machine learning, concurrent software, program verification, heap analysis, and the precision of static analyzers. All together, the talks provided a deep, up-to-date, and some controversial views on the state of the art in SSPA and its application in software engineering practices. This was highly beneficial for sparking discussion of open scientific problems and potential industrial applications. Section 3 reports the abstracts of all the talks.

The breakout discussion sessions were organized into two main steps. During the first breakout session, the participants were randomly split into four distinct groups. Each group then had to identify the most compelling open problems in applying SSA to the software engineering lifecycle. A plenary discussion followed, during which participants identified the four open problems they considered most relevant. The outcome of this discussion is reported in Section 4.1. In a subsequent discussion session, four groups (one for each open problem) were formed, and participants chose which group to join. Each group had a leader and reported the identified problems in a later plenary session. At the end of this discussion, the four most relevant open problems were the standardization of SSA components (Section 4.2), the explainability of SSA results (Section 4.3), the interaction between LLMs and SSA (Section 4.4), and how to push the adoption of SSA at the earlier phases of the software engineering process (Section 4.5).

Last but not least, various social activities took place every evening after dinner. These spanned from organized tournaments to informal board games. Section 5 reports all the results that were tracked during the seminar.

Conclusion

We consider the seminar a success. Altogether, it laid the basis for several tasks and follow-up:

- First of all, each of the four identified open problems represents a fundamental challenge for our community, and each group identified several actionable tasks that we expect the scientific community will target in future work;
- Most of the talks presented either preliminary results not yet published or assessed results with several future perspectives. The lively discussions that took place during the seminar will help the authors to improve the work and continue it in various directions, and other participants will take inspiration for their future work;
- Finally, the friendly atmosphere helped establish new connections through various social activities and informal networking. We expect this will open the door to new collaborations, hopefully leading to novel scientific publications and projects.

2 Table of Contents

Executive Summary
Pietro Ferrara, Liana Hadarean, Jorge A. Navas, and Caterina Urban 38

Overview of Talks

A Journey through LiSA and its Frontends
Vincenzo Arceri 42

Shape Analysis by Abstract Interpretation of Non-Blocking Concurrent Programs
Valentin Barbazo 43

Syntactically Convex Model-Based Projection for Linear Rational Arithmetic
Anna Becchi and Arie Gurfinkel 43

Distributed Summary Synthesis: A Divide-and-Conquer Approach for Distributed Program Analysis
Dirk Beyer 44

Eternal Problems Never or Hardly Solved in Static Analysis by Abstract Interpretation
Patrick Cousot 45

Scaling up Roundoff Analysis of Functional Data Structure Programs
Eva Darulova 45

Perspectives on the Static Analysis of Synchronous Data-Flow Languages
Charles De Haro 46

Path Optimal Symbolic Execution
Giovanni Denaro 46

Optional Type Systems: Current Approaches and Ongoing Efforts
Werner Dietl 46

Pyra: A High-level Linter for Data Science Software Code Smells
Greta Dolcetti 47

RubberDuckBench: A Benchmark for AI Coding Assistants
Elizabeth Dinella 48

Securing Software Supply Chains with Security-Enhanced SBOMs
Musard Balliu 48

Security Analysis at Scale with the Sigma Engine
Isabel Garcia-Contreras 48

Automatic Verification of Replicated Data Types
Elisa Gonzalez Boix 49

Directed Program Analysis: from Suspicion to Witnesses
Kihong Heo 49

Static Analysis and Verification in The Ciao Playground
Manuel Hermenegildo 50

Developing Cost-Effective Combination of Static Analysis Techniques
Minseok Jeon 50

Cost of Soundness in Mixed-Precision Tuning
Debasmita Lohar 51

Formally Checking the Stability of (Small) Decision Tree Models with Intervals
Antoine Miné 51

Try-Mopsa: Relational Static Analysis in Your Pocket
Raphaël Monat 52

Static Analysis through Trust Boundaries
Naïm Moussaoui Remil 52

Formally Verifying Solana Protocols with the Certora Prover
Jorge A. Navas 54

Towards Interactive Abstract Interpretation for Multithreaded Programs
Michael Schwarz and Helmut Seidl 55

Static Analysis in the Cloud-Native Era
Davide Taibi 55

Tai-e: Sound Static Analysis Framework for Modern Software Engineering
Tian Tan 56

From Detection to Quantification: A New Era of Declarative Program Analysis
Jingbo Wang 56

Working Groups

Open Scientific Problems
Pietro Ferrara, Liana Hadarean, Jorge Navas, and Caterina Urban 57

Standardization of Static Analysis Components and Interfaces 59

Explainability in Static Analysis 66

Combination of LLMs and Static Analysis 68

Shifting Abstract Interpretation Left: Using Abstraction Interpretation Earlier in the Development Process 69

Social Activities 71

Participants 74

3 Overview of Talks

3.1 A Journey through LiSA and its Frontends

Vincenzo Arceri (University of Parma, IT)

Joint work of Vincenzo Arceri, Luca Negrini, Luca Olivieri, Pietro Ferrara, Agostino Cortesi
Main reference Luca Negrini, Pietro Ferrara, Vincenzo Arceri, Agostino Cortesi: "LiSA: A Generic Framework for Multilanguage Static Analysis", pp. 19–42, Springer, 2023.
URL https://doi.org//10.1007/978-981-19-9601-6_2

LiSA (Library for Static Analysis)[1, 2] is an open-source Java library that provides a comprehensive infrastructure for building static analyzers based on abstract interpretation. We present some of LiSA's core components, such as the parametric combination of heap and value domains and the CFG nodes-to-symbolic expressions rewriting system. We then present two LiSA frontends, GoLiSA[3] and EVMLiSA[4], which are static analyzers for Go and EVM bytecode, respectively. We discuss the main challenges encountered during the development of both LiSA and its frontends. Finally, we outline the next steps and future directions of the project, such as the integration of string-to-code statement analyses with abstract interpretation in LiSA[5].

References

1 Pietro Ferrara, Luca Negrini, Vincenzo Arceri, Agostino Cortesi. Static analysis for dummies: experiencing LiSA. SOAP@PLDI 2021: Proceedings of the 10th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, Virtual Event, Canada, 22 June, 2021, pages 1–6, ACM, 2021. 10.1145/3460946.3464316

2 Luca Negrini, Pietro Ferrara, Vincenzo Arceri, Agostino Cortesi. LiSA: A Generic Framework for Multilanguage Static Analysis. In: *Challenges of Software Verification*. Ed. by Vincenzo Arceri, Agostino Cortesi, Pietro Ferrara, Martina Olliaro. Intelligent Systems Reference Library, volume 238, pages 19–42, Springer, 2023. 10.1007/978-981-19-9601-6_2

3 Luca Olivieri, Luca Negrini, Vincenzo Arceri, Fabio Tagliaferro, Pietro Ferrara, Agostino Cortesi, Fausto Spoto. Information Flow Analysis for Detecting Non-Determinism in Blockchain. 37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States. LIPIcs, volume 263, pages 23:1–23:25, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. 10.4230/LIPIcs.ECOOP.2023.23

4 Vincenzo Arceri, Saverio Mattia Merenda, Luca Negrini, Luca Olivieri, Enea Zaffanella. EVMLiSA: Sound Static Control-Flow Graph Construction for EVM Bytecode. *Blockchain: Research and Applications*, pages 100384, 2025. 10.1016/j.bcra.2025.100384

5 Vincenzo Arceri, Isabella Mastroeni. A sound abstract interpreter for dynamic code. SAC '20: The 35th ACM/SIGAPP Symposium on Applied Computing, online event, [Brno, Czech Republic], March 30 – April 3, 2020, pages 1979–1988, ACM, 2020. 10.1145/3341105.3373964

3.2 Shape Analysis by Abstract Interpretation of Non-Blocking Concurrent Programs

Valentin Barbazo (ENS, PSL University – Paris, FR)

Joint work of Valentin Barbazo, Xavier Rival

While concurrency can achieve significant performance improvements, it also introduces an additional layer of complexity in program design. Synchronization mechanisms are required to prevent data races on shared resources, which could otherwise lead to inconsistent results or data corruption. For instance, pointer updates by one thread may render some memory regions unreachable to others, thereby invalidating their assumptions about shared data. In this talk I present ongoing work toward a shape analysis for concurrent programs based on non-blocking synchronization mechanisms. Our goal is to prove the preservation of structural properties of dynamically allocated data structures shared across multiple threads. Our approach builds on previous work describing how to embed Rely-Guarantee reasoning in the Abstract Interpretation framework [1]. I describe our current progress on the design of an effective thread-modular shape analysis leveraging existing separation-logic-based memory abstract domains to precisely capture and account for thread interactions.

References

1 Antoine Miné. "Relational Thread-Modular Static Value Analysis by Abstract Interpretation". In: *Verification, Model Checking, and Abstract Interpretation.* Ed. by Kenneth L. McMillan and Xavier Rival. Springer Berlin Heidelberg, 2014. DOI: 10.1007/978-3-642-54013-4_3

3.3 Syntactically Convex Model-Based Projection for Linear Rational Arithmetic

Anna Becchi (USI – Lugano, CH) and Arie Gurfinkel (University of Waterloo, CA)

Joint work of Anna Becchi, Arie Gurfinkel, Grigory Fedyukovich, Lev Nachmanson

Quantifier elimination (QE) is a key task in formal verification algorithms. An important feature of QE is the ability to return partial results when interrupted early: a notable example is Model-Based Projection (MBP), a technique used to return an under-approximation of QE. Another desirable feature of QE is that the output is returned in a concise representation. In the theory of Linear Rational Arithmetic (LRA), existing QE methods often fail to preserve syntactic convexity, that is, they return a disjunction even for a conjunctive input, or they return a large non-minimal representation. This can disadvantage the QE client. To address these issues, we introduce the new concept of Bidirectional Model-Based Projection (BMBP). Given a model, BMBP returns both an over- and an under-approximation of QE. We define a new QE algorithm for LRA (BMBP-QE) that (i) is an anytime algorithm capable of providing two complementary partial results – a conjunctive over-approximation and a disjunctive under-approximation – when interrupted early, (ii) returns a minimal conjunction when the input is conjunctive, and (iii) applies to arbitrary LRA formulae. Given the twofold nature of the partial results, BMBP has the potential to enable new strategies in verification

algorithms currently relying on MBP. BMBP also benefits QE clients by returning a minimal representation for conjunctive inputs. We show that our algorithm does not introduce significant overhead compared to QE methods using MBP. In fact, BMBP-QE outperforms SMT-based QE algorithms, offering improvements in both runtime and result size.

3.4 Distributed Summary Synthesis: A Divide-and-Conquer Approach for Distributed Program Analysis

Dirk Beyer (LMU München, DE)

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. This hinders the integration of software model checking into the development process (continuous integration). 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 all blocks can be analyzed in parallel, as independent verification problems. Whenever new information (as a postcondition or violation condition) is available from other blocks, the verification can decide to restart with this new information. We formulate our approach as configurable program analysis and implement it for the verification of C programs in the widely used verifier CPAchecker. A large experimental evaluation shows the potential of our new approach: The distribution of the workload to several processing units works well, and there is a significant reduction of the response time when using multiple processing units. There are even cases in which the new approach beats the highly-tuned, existing single-threaded predicate abstraction.

References

1 Dirk Beyer, Matthias Kettl, and Thomas Lemberger. Decomposing Software Verification using Distributed Summary Synthesis. Proc. ACM Softw. Eng., 1(FSE), 2024. ACM. doi:10.1145/3660766 `https://doi.org/10.1145/3660766`

3.5 Eternal Problems Never or Hardly Solved in Static Analysis by Abstract Interpretation

Patrick Cousot (New York University, US)

I tried to remember the history of the discovery of the principles of abstract interpretation and collect problems that remain challenging, some over 50 years, like definition of the semantics of programming languages, efficiency, soundness, scalability, study of the discovery of inductive arguments in inductive proofs, understanding of passage to the limit in infinite domains, study of widening, narrowing, and their duals, absence of tool for the calculational design of abstract semantics, logics, or interpreters by abstraction of a semantics, automation of the design of transformers, exploitation of parallelism in fixpoint computations (asynchronous iterations), necessary non-monotony of widening, separation, modularity, analysis of complex data structures, absence of precise specification of libraries, absence of education in static analysis, incompetence of programmers in the use of static analyzers, explainability, hesitation of hierarchies in front of innovation, short terms goals, ignorance of safety and security, belief that artificial intelligence will solve all problems, proliferation of ad-hoc unprincipled research, deficiencies of reviewers in conferences if not journals, to cite a few.

3.6 Scaling up Roundoff Analysis of Functional Data Structure Programs

Eva Darulova (Uppsala University, SE)

Joint work of Anastasia Isychev, Eva Darulova
Main reference Anastasia Isychev, Eva Darulova: "Scaling up Roundoff Analysis of Functional Data Structure Programs", in Proc. of the Static Analysis – 30th International Symposium, SAS 2023, Cascais, Portugal, October 22-24, 2023, Proceedings, Lecture Notes in Computer Science, Vol. 14284, pp. 371–402, Springer, 2023.
URL https://doi.org//10.1007/978-3-031-44245-2_17

Floating-point arithmetic is counter-intuitive due to inherent rounding errors that potentially occur at every arithmetic operation. A selection of automated tools now exists to ensure correctness of floating-point programs by computing guaranteed bounds on rounding errors at the end of a computation, but these tools effectively consider only straight-line programs over scalar variables. Much of numerical codes, however, use data structures such as lists, arrays or matrices and loops over these. To analyze such programs today, all data structure operations need to be unrolled, manually or by the analyzer, reducing the analysis to straight-line code, ultimately limiting the analyzers' scalability. We present the first rounding error analysis for numerical programs written over vectors and matrices that leverages the data structure information to speed up the analysis. We facilitate this with our functional domain-specific input language that we design based on a new set of numerical benchmarks that we collect from a variety of domains. Our DSL explicitly carries semantic information that is useful for avoiding duplicate and thus unnecessary analysis steps, as well as enabling abstractions for further speed-ups. Compared to unrolling-based approaches in state-of-the-art tools, our analysis retains adequate accuracy and is able to analyze more benchmarks or is significantly faster, and particularly scales better for larger programs.

3.7 Perspectives on the Static Analysis of Synchronous Data-Flow Languages

Charles De Haro (ENS, PSL University – Paris, FR)

Joint work of Charles De Haro, Marc Pouzet, Xavier Rival

This presentation explores theoretical perspectives on static analyses of synchronous data-flow languages, such as Lustre and SCADE, which model programs as sets of equations describing relationships between input and output streams. We discuss the benefits of analyzing the source program directly, rather than compiling away high-level constructs of these languages – such as automata or clocks – and highlight the advantages of using synchronous observers, which allow safety properties to be specified directly as programs. Additionally, we present scheduling techniques that avoid unnecessary computations when solving equations, while preserving the soundness of the semantics.

3.8 Path Optimal Symbolic Execution

Giovanni Denaro (University of Milano-Bicocca, IT)

Joint work of Giovanni Denaro, Pietro Braione, Luca Guglielmo

Symbolic execution is at the core of many techniques for program analysis and test generation. Traditional symbolic execution of programs with numeric inputs enjoys the property of forking as many analysis traces as the number of analyzed program paths, a property that in this talk we refer to as path optimality. On the contrary, current approaches for symbolic execution of heap-manipulating programs fail to satisfy this property, thereby incurring crucial path explosion effects. In the talk, we present Path Optimal Symbolic Execution, a novel symbolic execution algorithm that originally accomplishes path optimality against heap-manipulating programs.

3.9 Optional Type Systems: Current Approaches and Ongoing Efforts

Werner Dietl (University of Waterloo, CA)

The Java type system gives useful guarantees about software, but there are many properties that cannot be expressed. One pervasive property that cannot be expressed is whether a reference can be null or not. The resulting null pointer exceptions are the bane of programmers and have been called the "billion dollar mistake". They happen even if you think hard about your code and test it thoroughly. There are many causes for null pointer exceptions, including object initialization, missing map keys, confusing contracts, and missing checks. There are many other properties that cannot be expressed in the Java type system that lead to severe runtime errors.

Optional type systems allow developers to improve the quality of their software by encoding additional properties as type systems and enforcing these properties at compile time. We will discuss a type system that prevents null pointer exceptions at compile time and the general framework it builds upon. These optional type systems have found hundreds of bugs in millions of lines of well-tested code. We will discuss alternative tools and interoperability, in particular with Kotlin, and the JSpecify effort to standardize static analysis annotations across the Java ecosystem. Finally, we will briefly discuss whole-program type inference and the interaction of optional type systems with deductive verification.

3.10 Pyra: A High-level Linter for Data Science Software Code Smells

Greta Dolcetti (University of Venice, IT)

Joint work of Greta Dolcetti, Vincenzo Arceri, Antonella Mensi, Enea Zaffanella, Caterina Urban, Agostino Cortesi
Main reference Greta Dolcetti, Vincenzo Arceri, Antonella Mensi, Enea Zaffanella, Caterina Urban, Agostino Cortesi: "Introducing Pyra: A High-Level Linter for Data Science Software", in Proc. of the Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track and Demo Track, pp. 449–453, Springer Nature Switzerland, 2026.
URL https://doi.org/10.1007/978-3-032-06129-4_29

The development of data science software is prone to a variety of mistakes that can easily compromise the validity of results. While existing tools mainly target low-level programming errors, higher-level issues specific to the data science pipeline remain less systematically addressed.

We present Pyra, a high-level linter designed to detect code smells in data science workflows. Pyra relies on static analysis by abstract interpretation, assigning ad hoc abstract datatypes to program variables and checking their consistency when interacting with data science libraries. By adopting a descriptive type system, Pyra captures suspicious patterns without requiring annotations or code modifications, simply by analyzing Python code.

Pyra currently implements 16 checkers across four categories: misleading visualizations, misleading results (e.g., data leakage), challenges for reproducibility (e.g., missing random seeds), and general issues (e.g., unintended consequences of in-place operations).

By bridging programming and domain-specific best practices, Pyra supports more robust, reproducible, and trustworthy data science pipelines.

References

1 Dolcetti, G., Arceri, V., Mensi, A., Zaffanella, E., Urban, C., Cortesi, A. (2026). Introducing Pyra: A High-Level Linter for Data Science Software. In: Dutra, I., et al. Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track and Demo Track. ECML PKDD 2025. Lecture Notes in Computer Science(), vol 16022. Springer, Cham. https://doi.org/10.1007/978-3-032-06129-4_29

2 Greta Dolcetti, Agostino Cortesi, Caterina Urban, and Enea Zaffanella. 2024. Towards a High Level Linter for Data Science. In Proceedings of the 10th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (NSAD '24). Association for Computing Machinery, New York, NY, USA, 18–25. https://doi.org/10.1145/3689609.3689996

3.11 RubberDuckBench: A Benchmark for AI Coding Assistants

Elizabeth Dinella (Bryn Mawr College, US)

Joint work of Elizabeth Dinella, Fatma Ayad, Ferida Mohammed, Petros Maniatis, Satish Chandra

Programmers are turning to AI coding assistants to answer questions about their code. Benchmarks are needed to soundly evaluate these systems and understand their performance. To enable such a study, we curate a benchmark of real-world contextualized questions derived from Github pull request comments. Out of this work, we present RubberDuckBench: a multilingual benchmark of questions about code, along with detailed rubrics for evaluating answers. We evaluate a diverse set of 20 LLMs (proprietary & open-source) on answering these questions. We find that even state of the art models fail to give consistent, correct responses across the benchmark. Grok 4 (69.29%), Claude Opus 4 (68.5%), and GPT-5 (67.8%) perform best overall, but do not exhibit pairwise significant superiority over the next 9 best performing models. Most models obtain points through partial credit, with the best performing models only answering at most 2 questions completely correctly across all trials. Furthermore, models often hallucinate with lies in 58.3% of responses on average. Cost analysis reveals no correlation between expense (API pricing or parameter count) and performance. We intend this benchmark to be a target for future research in trustworthy and correct AI coding assistants.

3.12 Securing Software Supply Chains with Security-Enhanced SBOMs

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

Joint work of Eric Cornelissen, Musard Balliu

This talk will discuss an enhancement of software bill of materials (SBOMs) with security capabilities to specify and enforce security policies for applications and their software supply chains with the goal of preventing malware.

References

1 Eric Cornelissen and Musard Balliu. *NodeShield: Runtime Enforcement of Security-Enhanced SBOMs for Node.js.* ACM Conference on Computer and Communications Security (CCS 2025), Taipei, Taiwan, 2025

3.13 Security Analysis at Scale with the Sigma Engine

Isabel Garcia-Contreras (Black Duck – Calgary, CA)

We present how we do static analysis in Black Duck using Sigma, a fast static analyzer for web/mobile security. We focus on how to make it easier for customers to analyze their code and then understand and prioritize analysis findings. We demo Sigma on an open-source project.

3.14 Automatic Verification of Replicated Data Types

Elisa Gonzalez Boix (VU – Brussels, BE)

Joint work of Kevin De Porre, Carla Ferreira, Elisa Gonzalez Boix
Main reference Kevin De Porre, Carla Ferreira, Elisa Gonzalez Boix: "VeriFx: Correct Replicated Data Types for the Masses", in Proc. of the 37th European Conference on Object-Oriented Programming, ECOOP 2023, Seattle, Washington, United States, July 17-21, 2023, LIPIcs, Vol. 263, pp. 9:1–9:45, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023.
URL https://doi.org//10.4230/LIPICS.ECOOP.2023.9

Distributed systems replicate data to improve availability, scalability, and fault tolerance. Ensuring that replicas remain eventually consistent is difficult. Current practices advocate for the use of Replicated Data Types (RDTs) which guarantee convergence out-of-the-box, e.g. CRDTs. However, programming distributed systems using these RDTs is non-trivial due to the lack of appropriate abstractions for replica discovery, update propagation, etc.

In this presentation, we introduce the VeriFx language, a high-level functional object-oriented programming language specially designed to empower developers with automated verification features. VeriFx lets programmers implement RDTs atop functional collections and express correctness properties that are verified automatically. For each proof, VeriFx derives the necessary proof obligations, which are encoded into first-order logic and discharged automatically by leveraging SMT solving. If a property does not hold, VeriFx returns a high-level counterexample. Verified RDTs can be transpiled to mainstream languages, e.g. Scala. VeriFx provides libraries that implement the execution model and define the correctness properties of well-known RDT families. We used those libraries to verify 51 CRDTs and reproduce a study on the correctness of Operational Transformation functions.

3.15 Directed Program Analysis: from Suspicion to Witnesses

Kihong Heo (KAIST – Daejeon, KR)

Joint work of Tae Eun Kim, Jaeseung Choi, Kihong Heo, Sang Kil Cha
Main reference Tae Eun Kim, Jaeseung Choi, Kihong Heo, Sang Kil Cha: "DAFL: Directed Grey-box Fuzzing guided by Data Dependency", in Proc. of the 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023, pp. 4931–4948, USENIX Association, 2023.
URL https://www.usenix.org/conference/usenixsecurity23/presentation/kim-tae-eun

Static program analysis for software bug detection has steadily advanced over many decades and has become a core technology in modern software development. However, the difficulty of inspecting whether analyzer-reported alarms remains a persistent challenge for many developers. In this talk, we present our journey toward addressing this issue and realizing a "fully automated bug detection tool" including buggy input generation. The key lies in "directed input generation". This technique aims to automatically generate inputs that reach suspicious program points (e.g., static analysis alarms, recently changed code, etc). We will discuss how we combine static analysis with fuzzing and program synthesis to efficiently generate bug-triggering inputs for such points. Also, we share the remaining challenges to improve the current state-of-the-art.

3.16 Static Analysis and Verification in The Ciao Playground

Manuel Hermenegildo (IMDEA Software Institute – Madrid, ES)

Joint work of Manuel Hermenegildo, José Francisco Morales, Pedro López, Daniel Jurjo, Marco Ciccalè, Daniela Ferreiro, Marco Perez, Louis Rustenholz
URL http://play.ciao-lang.org

We presented and demoed the Static Analysis and Verification functionality currently supported within the Ciao playground (https://play.ciao-lang.org). The playground is a browser-based program development environment which integrates the CiaoPP abstract interpreter and verifier as an embedded tool. In contrast with the more traditional ways of using CiaoPP (within Visual Studio Code, Emacs, etc.), the playground does not require any installation, while retaining almost all the capabilities of the full, native version of the systems involved. This makes is a very convenient tool for experimentation and teaching. The whole system runs locally on the student's browser, compiled to web assembly code, and does not need any server infrastructure. This has advantages from the point of view of scalability, low maintenance cost, security, privacy, etc. We presented the playground in action, in different analysis and verification including interactive, on-the-fly, assertion checking as the program is developed, covering properties such as types/shapes, sharing/aliasing, numeric properties, determinism, data sizes, computational cost, etc. We also showed the different configuration options available and the different kinds of information that can be obtained from the analyzer. We demonstrated as well the abstract debugger, that allows advancing step by step in the source program editor while watching the evolution of the abstract values as the fixpoints are computed. Finally, we presented how documents with embedded instances of the analyzer can also be developed within the playground, which allows easily developing interactive tutorials for static analysis tasks, from analyzing and verifying programs to developing new abstract domains.

3.17 Developing Cost-Effective Combination of Static Analysis Techniques

Minseok Jeon (DGIST Institute of Science & Technology – Daegu, KR)

In this talk, I will introduce an issue in the current trend of developing static analysis techniques and present an efficient approach to address it. To develop cost-effective static analyzers, various static analysis techniques have been proposed. For optimal performance, these techniques should be combined to leverage the benefits of each approach. In the literature, however, the techniques have largely been developed independently, without considering their potential combinations. We discovered that combining independently developed static analysis techniques often results in suboptimal performance, even when each technique is individually optimal. However, developing combinations of techniques while accounting for all interactions between them is a huge burden. To address this issue, I introduce an approach that safely and effectively reduces the burden of developing such combinations.

3.18 Cost of Soundness in Mixed-Precision Tuning

Debasmita Lohar (KIT – Karlsruher Institut für Technologie, DE)

Joint work of Debasmita Lohar, Anastasia Isychev
Main reference Anastasia Isychev, Debasmita Lohar: "Cost of Soundness in Mixed-Precision Tuning", Proc. ACM Program. Lang., Vol. 9(OOPSLA2), Association for Computing Machinery, 2025.
URL https://doi.org//10.1145/3763137

Mixed-precision tuning is a key optimization technique for numerical programs, where some variables and operations are assigned lower precisions to improve overall performance. It is particularly relevant in applications with frequent numerical computation on resource-constrained hardware, such as embedded systems, scientific computing, and machine learning. The main challenge lies in the trade-off: while reduced precision improves efficiency, it introduces rounding errors that can compromise accuracy.

In this work, we present the first comprehensive evaluation of state-of-the-art tools for mixed-precision tuning. These tools typically fall into two categories: sound static analyses, which offer formal guarantees but are often considered overly conservative, and dynamic methods, which explore a broader optimization space but can violate error bounds, sometimes by several orders of magnitude. Through an extensive comparison on the FPBench benchmark suite, we quantify the trade-offs between performance and soundness. Our results show that sound tools, when enhanced with techniques such as regime inference, can match or even outperform dynamic ones, while still preserving correctness guarantees.

Yet, extending sound guarantees to larger, real-world programs remains a challenge. A promising direction could be to combine static analyses with heuristic guidance – potentially informed by large language models that can help navigate the search space – to improve scalability and enable integration into real-world software engineering workflows.

3.19 Formally Checking the Stability of (Small) Decision Tree Models with Intervals

Antoine Miné (Sorbonne University – Paris, FR)

Joint work of Waly Fall, Antoine Miné

Decision Trees are a popular kind of predictive models used in machine learning. Previous work in explainable AI by Hurault and Marques-Silva proposed logic-based criteria and algorithms to provide local explanations for model decisions (so-called abductive and contrastive explanations), but efficient explanation synthesis requires models to be stable, a global property difficult to check. In this work, we develop an interval-based method able to formally prove (small-scale) decision tree models to be stable. When they are not stable, we are able to generate counter-examples, compute a quantitative measure of stability, and even provide fixes to make the model stable. We also consider monotony, a more restrictive property than stability, and prove formally monotony as well as infer, when possible, all orders that make the model monotone. We present preliminary experimental results on a few small (4-7 features, 50KB-2MB model size) models trained using Gradient Boosting with XGBoost and LightGBM, from data from OpenML, Kaggle, and UC Irvine. Our preliminary results indicate that, even though the training algorithms are advertised as

computing monotone models, the output is not always monotone, and not even stable. In some cases, almost-stable models can be made stable while keeping (or improving) their size and precision. We also show that one-hot encoding is useful to ensure monotony and stability of trained models. This is joint work with Waly Fall.

3.20 Try-Mopsa: Relational Static Analysis in Your Pocket

Raphaël Monat (INRIA Lille, FR)

Main reference Raphaël Monat: "Try-Mopsa: Relational Static Analysis in Your Pocket", CoRR, Vol. abs/2509.13128, 2025.
URL https://doi.org//10.48550/ARXIV.2509.13128

Static analyzers are complex pieces of software with large dependencies. They can be difficult to install, which hinders adoption and creates barriers for students learning static analysis. This work introduces Try-Mopsa: a scaled-down version of the Mopsa static analysis platform, compiled into JavaScript to run purely as a client-side application in web browsers. Try-Mopsa provides a responsive interface that works on both desktop and mobile devices. Try-Mopsa features all the core components of Mopsa. In particular, it supports relational numerical domains. We present the interface, changes and adaptations required to have a pure JavaScript version of Mopsa. We envision Try-Mopsa as a convenient platform for onboarding or teaching purposes.

3.21 Static Analysis through Trust Boundaries

Naïm Moussaoui Remil (ENS, PSL University – Paris, FR)

Joint work of Naïm Moussaoui Remil, Caterina Urban

Background

Classical static analysis methods used to treat variables behavior uniformly. Yet, in practice, this can be unnecessarily restrictive. Indeed, not all programs variables play the same role. Some variables are *untrusted*: their value may be externally-controlled, for instance by user-input, an operating system scheduler, or a third-party library. By contrast, other variables are trusted: their values stem from non-deterministic choices, internal computations, or library calls under the control of the programmer. Recent works have studied safety properties while distinguishing trusted variables untrusted variables. Girol et al. [1] proposed Robust Reachability which refine the classical notion of reachability. Robust reachability holds if the untrusted variable can make a bug reachable whatever the values of the trusted variables. Also, Parolini et al. [2] define a static analysis for *safety* Non-Exploitability proving that the untrusted variables cannot trigger or silence a safety bug. We go beyond safety properties, and contribute with static analysis for (generalization of) liveness and CTL properties withing this setting.

Presentation Proposal

First, we will present an abstract-interpretation based static analysis that proves that for every possible (sequence of) untrusted input(s), there exists at least one terminating execution. We call this property Termination Resilience, it generalizes termination on program with untrusted and trusted inputs. We also propose a static analysis Robust Non-Termination which is defined as the negation of Termination Resilience i.e. there exists a choice for the untrusted inputs such that the program always diverge. Next, we will present abstract-interpretation based static analysis for inferring minimal sets of inputs variables, along with a sufficient preconditions, ensuring a CTL (computational tree logic) property. This static analysis can be interpreted in multiple way – for example, as a partition of the inputs variables in untrusted and trusted variables that guarante the robustness or the resilience of a CTL property.

Termination Resilience

When the control of some untrusted variables alone is sufficient to enforce divergence, we say that the program is vulnerable to Robust Non-Termination, which represents a serious concern in practice. Non-termination that may be avoided through trusted choices may be acceptable, as it cannot be reliably exploited by an adversary; it reflects internal system behavior rather than external influence. From a security perspective, this distinction is crucial: divergence caused by untrusted inputs constitutes a potential denial-of-service vulnerability, whereas divergence caused by trusted variables does not. From a software engineering standpoint, the distinction is equally important as it hints at a principled way to triage non-termination alarms, helping developers prioritize the more critical cases.

This distinction motivates the property of Termination Resilience; for every possible (sequence of) untrusted inputs, there exists at least one terminating execution. We have derived a sound static analysis for both Termination Resilience and Robust-Non Termination by building upon Urban and Miné's decision tree abstract domain [4, 5], enhancing it with novel and non-trivial transformer operators to effectively manage this mixed settings. When unable to prove that a program always satisfies Termination Resilience, our static analysis automatically infers *sufficient preconditions* that ensure Termination Resilience.

Inference of Minimal Sets of Variables Ensuring a CTL Property

We have extended an abstract-interpretation–based static analysis for CTL properties (using the decision tree abstract domain) with an abstraction refinement process. This abstract refinement process infers minimal sets of variables, along with sufficient preconditions, to ensure a CTL property. One interpretation of this analysis is the inference of partitions of the input variables into untrusted and trusted sets for a given CTL property. For instance, if the CTL property defines the reachability of a bug, our analysis infers minimal sets of input variables with sufficient preconditions that ensure the bug is reachable. Hence, our analysis automatically infers sets of untrusted input variables with sufficient preconditions that ensure Robust Reachability holds.

This work has been published in the 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning [3].

References

1 Guillaume Girol, Benjamin Farinier, Sébastien Bardin. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In: *Computer Aided Verification – 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I.* Ed. by Alexandra Silva and K. Rustan M. Leino. Lecture Notes in Computer Science, volume 12759, pages 669–693, Springer, 2021. 10.1007/978-3-030-81685-8_32

2 Francesco Parolini, Antoine Miné. Sound Abstract Nonexploitability Analysis. In: *Verification, Model Checking, and Abstract Interpretation – 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II.* Ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Lecture Notes in Computer Science, volume 14500, pages 314–337, Springer, 2024. 10.1007/978-3-031-50521-8_15

3 Naïm Moussaoui Remil, Caterina Urban, Antoine Miné. Automatic detection of vulnerable variables for CTL properties of programs. In: *25th Conference on Logic for Programming, Artificial Intelligence and Reasoning.* Volume 100, pages 116–126, EasyChair, 2024.

4 Caterina Urban, Antoine Miné. A decision tree abstract domain for proving conditional termination. International Static Analysis Symposium, pages 302–318, Springer, 2014.

5 Caterina Urban, Samuel Ueltschi, Peter Müller. Abstract Interpretation of CTL Properties. SAS 2018, pages 402–422, 2018.

3.22 Formally Verifying Solana Protocols with the Certora Prover

Jorge A. Navas (Certora – Seattle, US)

Joint work of Jorge A. Navas, Arie Gurfinkel

The Solana Certora Prover (SCP) is a formal verification tool designed to check the correctness of Solana smart contracts. SCP takes as input both a Solana program and its specification, both written in Rust, and determines whether the implementation satisfies its specification.

In this seminar, I will demonstrate how SCP can be applied to prove fundamental correctness properties of a tokenized vault protocol, including solvency, absence of shares dilution, and fee assessment. I will also discuss the main technical challenges involved in generating efficient and sound SMT-based verification conditions, and how Abstract Interpretation techniques are used to address these challenges.

3.23 Towards Interactive Abstract Interpretation for Multithreaded Programs

Michael Schwarz (National University of Singapore, SG) and Helmut Seidl (TU München – Garching, DE)

Joint work of Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, Helmut Seidl
Main reference Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, Helmut Seidl: "Interactive abstract interpretation: reanalyzing multithreaded C programs for cheap", Int. J. Softw. Tools Technol. Transf., Vol. 26(6), pp. 647–667, 2024.
URL https://doi.org//10.1007/S10009-024-00768-9

To put sound program analysis at the fingertips of working class developers, we propose a framework for interactive abstract interpretation of multithreaded C code leveraging mixed-flow sensitivity. Abstract interpretation provides sound analysis results, but can be quite costly in general. To achieve quick response times, we incrementalize the analysis infrastructure, including post-processing, without necessitating any modifications to the analysis specifications themselves. The integration of update rules enables precise incremental analysis of intricate program properties – including concurrency deficiencies such as data-races and deadlocks. A prototype of the framework has been implemented in the static analyzer Goblint, and integrated into IDEs leveraging `MagpieBridge`. We evaluate our implementation w.r.t. the yard sticks of response time and from-scratch-consistency. We provide examples of program development highlighting the usability of our approach. Lastly, we discuss promising new directions along the lines of on-demand precision increases and incremental result refinement as an interaction between the analyzer and a developer, making the analysis not just incremental but *interactive*.

3.24 Static Analysis in the Cloud-Native Era

Davide Taibi (University of Southern Denmark – Odense, DK)

Modern cloud-native systems are built as polyglot: multiple languages, frameworks, containers, and deployment descriptors. This makes their actual architecture hard to see and even harder to verify against intended blueprints.

This talk presents a static, multi-artifact analysis approach that reconstructs a system's architecture by mining source code, build files, Dockerfiles, Helm/Kubernetes manifests, API contracts, and developers' comunication. In particular, the reconstruction enables to detect recurring cloud-native antipatterns such as shared databases or cyclic service dependencies and to have a overview on the system evolution

3.25 Tai-e: Sound Static Analysis Framework for Modern Software Engineering

Tian Tan (Nanjing University, CN)

Main reference Tian Tan, Yue Li: "Tai-e: A Developer-Friendly Static Analysis Framework for Java by Harnessing the Good Designs of Classics", in Proc. of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023, Seattle, WA, USA, July 17-21, 2023, pp. 1093–1105, ACM, 2023.
URL https://doi.org//10.1145/3597926.3598120
Main reference Yufei Liang, Teng Zhang, Ganlin Li, Tian Tan, Chang Xu, Chun Cao, Xiaoxing Ma, Yue Li: "Pointer Analysis for Database-Backed Applications", Proc. ACM Program. Lang., Vol. 9(PLDI), pp. 1417–1441, 2025.
URL https://doi.org//10.1145/3729307
Main reference Menglong Chen, Tian Tan, Minxue Pan, Yue Li: "PacDroid: A Pointer-Analysis-Centric Framework for Security Vulnerabilities in Android Apps", in Proc. of the 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 – May 6, 2025, pp. 2803–2815, IEEE, 2025.
URL https://doi.org//10.1109/ICSE55347.2025.00208
Main reference Teng Zhang, Yufei Liang, Ganlin Li, Tian Tan, Chang Xu, Yue Li: "Bridge the Islands: Pointer Analysis for Microservice Systems", Proc. ACM Softw. Eng., Vol. 2(ISSTA), pp. 504–526, 2025.
URL https://doi.org//10.1145/3728896

Static analysis framework plays a crucial role in determining the capabilities of static analysis to adapt and integrate with modern software engineering practices. Recognizing the limitations of existing frameworks, we have developed Tai-e, a new static analysis framework for Java, aimed at meeting the demands of contemporary software development. Our study shows that Tai-e is easier to learn and use, offering a more extensible and efficient pointer analysis system than other frameworks. This provides a solid foundation for the development and integration of new analysis techniques.

A distinctive feature of modern software engineering today is the reliance on various frameworks, such as microservice, database, and mobile frameworks, rather than building software from scratch. While these frameworks greatly enhance productivity, their complexity poses challenges to the soundness of static analysis. To address this, we leveraged Tai-e's extensible pointer analysis system to develop a series of innovative techniques and tools for effectively analyzing these framework-based applications. Our evaluation demonstrates that our techniques achieve significantly better soundness compared to state-of-the-art methods for these types of applications.

In this talk, I will introduce Tai-e and our techniques for analyzing modern framework-based software.

3.26 From Detection to Quantification: A New Era of Declarative Program Analysis

Jingbo Wang (Purdue University – West Lafayette, US)

Declarative program analysis has long been a powerful tool for detecting data races, tracking information flows, and uncovering security vulnerabilities such as side-channel leaks. However, designing analysis rules that are both accurate and efficient remains a significant challenge, even for domain experts. To improve scalability, many prior works favor efficiency by relaxing completeness, leading to rules that are sound but incomplete. While this trade-off improves performance, it also produces a flood of false alarms, overwhelming developers and reducing the practical usefulness of the analysis.

In this talk, I will present our approach to bridging qualitative and quantitative program analysis. Rather than simply reporting whether a potential issue exists, our framework estimates the probability that a reported issue – such as an information leak – corresponds to a real vulnerability. This probabilistic perspective enables the analysis to prioritize high-risk findings and significantly reduce false alarms. Our solution is built on probabilistic logic programming, extending the widely used Soufflé Datalog engine. I will demonstrate how this framework achieves efficiency, accuracy, and incremental adaptability, making it well-suited for evolving codebases and complex analysis domains.

4 Working Groups

4.1 Open Scientific Problems

Pietro Ferrara (University of Venice, IT)

On the first day, participants were randomly split into four groups. Each group identified a leader and was supervised by one of the organizers. Each group discussed and then identified the three most compelling open scientific problems in applying static program analysis to modern software.

4.1.1 Group 1

Leader: Helmut Seidl, Organizer: Pietro Ferrara, Participants: Vincenzo Arceri, Musard Balliu, Werner Dietl, Arie Gurfinkel, Debasmita Lohar, Naïm Moussaoui Remil, Tian Tan

The discussion was somewhat controversial. First of all, formalizing a sound semantics for a real-world programming language is highly complex and time-consuming, given the wide variety of constructs, primitives, and technologies in each programming language. On the other hand, the best approach to explaining formal methods is to apply them to minimal languages, highlighting the main features tackled by the formalization. So far, it seems there is no solution to reconcile these two opposing needs. In addition, modern software comprises large libraries, and modular reasoning is needed to analyze them. In such a context, several relevant open problems need to be solved: how can we model these libraries? How can we verify their implementation meets their specification? Can we automatically derive the specification?

Open problems

1.1 Language workbench
1.2 Tiny languages
1.3 Modularity

4.1.2 Group 2

Leader: Michael Schwarz, Organizer: Caterina Urban, Participants: Roberto Bagnara, Patrick Cousot, Isabel Garcia-Contreras, Elisa Gonzalez Boix, Kihong Heo, Antoine Miné, Michael Schwarz

The first open problem concerns systems that comprise different languages and paradigms, and in particular, how these systems can be effectively statically analyzed without duplicating the effort of the scientific community. Other communities, particularly those focused on SMTs, have already successfully tackled similar problems and might inspire our community.

Another problem concerns the explanability and reproducibility of warnings. In particular, a warning by itself is not enough in production, since the software developer or project manager might not be able to understand the cause of the warning.

Finally, the last aspect concerns the application of static analysis at the early (aka, left) phases of the development lifecycle (e.g., during development) rather than at the end (e.g., before deployment). This requires incremental and/or interactive analysis and would help developers produce more robust code.

Open problems

2.1 Multilanguage and multiparadigm systems
2.2 Explanability of warnings
2.3 Shifting static analysis left

4.1.3 Group 3

Leader: Greta Dolcetti, Organizer: Jorge A. Navas, Participants: Valentin Barbazo, Eva Darulova, Charles De Haro, David Delmas, Minseok Jeon, Tim King, Valentina Lenarduzzi

The discussion began with the adoption of large language models (LLMs) in static analyzers. This is a recent, emerging scientific trend, but it is not yet clear how these can be adopted to help configure and use static analyzers. To explain their warnings? In combination or in replacement of static analysis? In addition, several properties (e.g., hallucinations) of LLMs have not yet been formalized, and their sustainability should be considered. A second open problem was identified in the lack of non-linear abstract domains. While there are very few examples, it seems that specific domains (such as avionics and machine learning algorithms) would need much more generic and expressive numerical domains in this context. Finally, two more open problems were discussed, one concerning shared data structures in concurrent or distributed systems and how components interact through them, and another one regarding differential analysis and portability.

Open problems

3.1 Integration of LLM into static program analysis
3.2 Non-linear abstract domains
3.3 Shared data structures in concurrent/distributed systems
3.4 Differential analysis and portability

4.1.4 Group 4

Leader: Raphaël Monat, Organizer: Liana Hadarean, Participants: Anna Becchi, Dirk Beyer, Giovanni Denaro, Elizabeth Dinella, Guido Salvaneschi, Jingbo Wang

Three main topics were identified from the beginning. Building on the idea of witnesses adopted by competitions for software verification tools, the group identified the need for the certification of programs through witnesses that are cheaper than complete analysis but still prove that the programs respect given properties. A second topic concerned how different analysis engines (potentially implemented with different programming languages and technologies) can cooperate, and how this can be generalized. Last but not least, how to communicate the results of an analysis and its justifications to non-expert users is still obscure and non-standard, as well as making the analysis assumptions clear and explicit to make the analysis more transparent.

Open problems

4.1 Certification of programs for sound AI use.
4.2 Cooperative verification engines
4.3 User interface and communication with non-expert users

4.1.5 Plenary Discussion

All the open problems identified by the various groups were grouped into homogeneous topics. Then each topic was voted on by the participants, and the four most voted on were chosen. Each participant was free to vote on many topics. The final results were as follows:

A Standardized multilanguage static analysis (open problems: 1.3, 2.1, 4.2), 18 votes.
B Explainability (open problems: 4.1, 4.3, 2.2), 20 votes.
C LLMs and static program analysis (open problems: 3.1), 15 votes.
D Real-world huge vs. academic tiny languages (open problems: 1.1, 1.2), 8 votes.
E Shifting static analysis left (open problems: 2.3), 13 votes.
F Non-linear abstract domains (open problems: 3.2), 7 votes.
G Shared data structures in concurrent/distributed systems (open problems: 3.3), 9 votes.
H Differential analysis and portability (open problems: 3.4), 4 votes.

Therefore, topics A, B, C, and E were chosen. Similar to the first selection phase, a leader and an organizer were chosen for each topic, but participants were free to join the topic they preferred. The outcomes of each discussion group are presented in Sections 4.2, 4.3, 4.4, and 4.5.

4.2 Standardization of Static Analysis Components and Interfaces

Discussion Leader: Raphaël Monat, Inria & University of Lille

We discussed how to standardize static analysis components and their interfaces, following the standardization successes in neighboring communities such as SMT-LIB [5] in the last decade. These challenges have been identified for more than 20 years by [6, 7] in their "Verifying compiler challenge" and "Verified software initiative".

This standardization can be achieved at different granularities, e.g at the whole analyzer level, or for given abstract domains (components of the analyzer).

Standardizing inputs and outputs at the analyzer level would be a good first step. It would allow to share and compare results, and even interoperate between analyses. Standardizing abstract domains would be more difficult, as their interfaces currently vary a lot depending on the kind of abstraction. From a technical point of view, such an interface should be supported by the wide variety of implementation languages (C/C++, Java, OCaml, Prolog) used by the community. Thanks to standardization, maintenance efforts of components could be shared by the whole community (rather than each group having to re-implement and maintain classical components).

We identified several efforts paving the way towards a kind of standardization:

- the Apron [12] library for relational abstract domains offers a de-facto interface supported by a wide variety of relational libraries, including PPL [13] Elina [14] and PPLite [16]. Some additional strong points underlined by participants where the fact that this interface is available through multiple language bindings (C/Java/OCaml), and that it is actively maintained.
- external fixpoint solvers such as the top-down solver [8, 19] could also be used by multiple tools. They are currently used in Goblint [33, 34] and Salto [32].
- correctness and violation witness [11] used in the software verification competition. A new format version 2.0 [20] was introduced in 2024. They allow the encoding of invariants and contracts – expressed through side-effect-free C expressions, although the format could be extended to e.g. ACSL [28] – or error paths. The format was extended to non-termination [27] in 2025. There is ongoing work to extend the format to handle more properties: memory and termination, as well as concurrency [15].
- the Static Analysis Results Interchange Format (SARIF) [29] format is an industrial format integrated within CI processes. It is extremely general and does not specify how different analyses of a same language could communicate.
- recent intermediate verification-oriented languages have been proposed, by [30] in the field of software verification, [10] in the field of deductive verification, and by [17] in model checking.
- JavaSMT [9] and PySMT [18] offer features of SMT solvers for software development in Java and Python, as an alternative to the SMT-LIB format.
- modular analyses platforms:
  - Crab [21] is a modular C++ library exposing reusable components such as abstract domains (with integration for PPL [13], Apron [12], and Elina [14]), fixpoint solvers, inter-procedural algorithms, and a language-agnostic Intermediate Representation (IR) enabling effective static analysis across diverse contexts. First, it serves as the foundation for stand-alone static analyzers or verifiers such as Clam [31], a LLVM-based abstract interpreter that translates LLVM bitcode to Crab IR, and Prevail [23], a verifier for eBPF programs. Second, it is used as an evaluation framework by other researchers where new abstract domains or algorithms have been implemented (see e.g., [14, 24, 25, 26]). Third, it integrates within broader verification infrastructures: Clam is part of the verification framework SeaHorn [22], enabling a combination of Abstract Interpretation with Model-Checking techniques. This versatility facilitates the standardization of static analyses by offering common building blocks that can be reused across tools and research projects.
  - Mopsa [36, 35] is a modular and open static analysis platform written in OCaml. Its goal is to encourage the research and education in abstract interpretation by providing a fully-featured and extensible open-source platform and usable analyses built with it. It supports the analysis of multiple languages (C [44, 42], Python [45], OCaml [47]), and

even multi-language analyses [46]. These analyses share the usage of some core abstract domains defining a semantic kernel. Mopsa has been used to develop new kind of analyses including non-exploitability of bugs [39], sufficient precondition inference [40], as well as patch and endianness portability analyses in industrial settings [38, 37]. In terms of usability, Mopsa provides transparency by reporting the proofs it attempted, supports abstract debugging [41] and is available in lightweight web-based version for demonstration purposes [43]. Mopsa has recently been used as a building block by researchers within the software engineering community [50, 49, 48].

- CiaoPP [1, 2, 3] is both an abstract interpretation-based pre-processor and a library of components for building abstract interpretation-based tools. As a tool CiaoPP can perform interactive program analysis, verification, certification (abstraction-carrying code), and optimization (abstract partial evaluation, slicing, program parallelization). The analysis process can be followed interactively on the program source. Different source languages can be supported by translation to a Horn clause-based IR. The tool is highly configurable in order to make use of all the different library components. Regarding the library, it includes a good number of abstract domains for shapes/types, variable (pointer) sharing and other variable instantiation properties, numerical properties, non-failure, determinacy, bounds on computational cost, bounds on data sizes, etc. There are also facilities for defining and combining domains and interfaces with tools like PPL. The library also includes several different analysis fixpoint algorithms (including incremental and modular), abstract partial evaluators, parallelization modules, etc. An almost complete version of CiaoPP can be run interactively on the browser [4].

References

1. M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, 2005. Available at `https://cliplab.org/papers/ciaopp-sas03-journal-scp.pdf`.
2. F. Bueno, P. Lopez-Garcia, J. F. Morales, G. Puebla, and M. V. Hermenegildo. The Ciao Program Preprocessor. Technical Report CLIP-2/2025, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, June 2025. Available at `https://ciao-lang.org/ciao/build/doc/ciaopp.html/`.
3. I. Garcia-Contreras, D. Ferreiro, J. F. Morales, F. Bueno, G. Puebla, P. Lopez-Garcia, and M. V. Hermenegildo. CiaoPP Tutorials. Technical Report CLIP-3/2025, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, January 2025. Available at `https://ciao-lang.org/ciao/build/doc/ciaopp_tutorials.html`.
4. The Ciao Developers. The Ciao Playground. Available at `https://ciao-lang.org/playground/`.
5. Clark Barrett, Pascal Fontaine, Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.
6. C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. In: *Perspectives of Systems Informatics, 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers.* Ed. by Manfred Broy and Alexandre V. Zamulin. Lecture Notes in Computer Science, volume 2890, pages 1–12, Springer, 2003. `10.1007/978-3-540-39866-0_1`
7. Tony Hoare, Jayadev Misra, Gary T. Leavens, Natarajan Shankar. The Verified Software Initiative: A Manifesto. In: *Theories of Programming: The Life and Works of Tony Hoare.* Ed. by Cliff B. Jones and Jayadev Misra. ACM Books, volume 39, pages 81–92, ACM / Morgan & Claypool, 2021. `10.1145/3477355.3477361`

8 Helmut Seidl, Ralf Vogler. Three Improvements to the Top-Down Solver. In: *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018.* Ed. by David Sabel and Peter Thiemann. Pages 21:1–21:14, ACM, 2018. 10.1145/3236950.3236967

9 Daniel Baier, Dirk Beyer, Karlheinz Friedberger. JavaSMT 3: Interacting with SMT Solvers in Java. In: *Computer Aided Verification – 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II.* Ed. by Alexandra Silva and K. Rustan M. Leino. Lecture Notes in Computer Science, volume 12760, pages 195–208, Springer, 2021. 10.1007/978-3-030-81688-9_9

10 Gidon Ernst, Paula Herber, Marieke Huisman, Mattias Ulbrich. SpecifyThis Bridging Gaps Between Program Specification Paradigms: Track Introduction. In: *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.* Ed. by Tiziana Margaria and Bernhard Steffen. Lecture Notes in Computer Science, volume 15221, pages 3–7, Springer, 2024. 10.1007/978-3-031-75380-0_1

11 Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, Michael Tautschnig. Verification Witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4): 57:1–57:69, 2022. 10.1145/3477579

12 Bertrand Jeannet, Antoine Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 – July 2, 2009. Proceedings.* Ed. by Ahmed Bouajjani and Oded Maler. Lecture Notes in Computer Science, volume 5643, pages 661–667, Springer, 2009. 10.1007/978-3-642-02658-4_52

13 Roberto Bagnara, Patricia M. Hill, Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2): 3–21, 2008. 10.1016/j.scico.2007.08.001

14 Gagandeep Singh, Markus Püschel, Martin T. Vechev. Fast polyhedra abstract domain. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017.* Ed. by Giuseppe Castagna and Andrew D. Gordon. Pages 46–59, ACM, 2017. 10.1145/3009837.3009885

15 Julian Erhard, Manuel Bentele, Matthias Heizmann, Dominik Klumpp, Simmo Saan, Frank Schüssele, Michael Schwarz, Helmut Seidl, Sarah Tilscher, Vesal Vojdani. Correctness Witnesses for Concurrent Programs: Bridging the Semantic Divide with Ghosts. In: *Verification, Model Checking, and Abstract Interpretation – 26th International Conference, VMCAI 2025, Denver, CO, USA, January 20-21, 2025, Proceedings, Part I.* Ed. by Shankaranarayanan Krishna, Sriram Sankaranarayanan, and Ashutosh Trivedi. Lecture Notes in Computer Science, volume 15529, pages 74–100, Springer, 2025. 10.1007/978-3-031-82700-6_4

16 Anna Becchi, Enea Zaffanella. PPLite: Zero-overhead encoding of NNC polyhedra. *Inf. Comput.*, 275: 104620, 2020. 10.1016/j.ic.2020.104620

17 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: *Model Checking Software – 30th International Symposium, SPIN 2024, Luxembourg City, Luxembourg, April 8-9, 2024, Proceedings.* Ed. by Thomas Neele and Anton Wijs. Lecture Notes in Computer Science, volume 14624, pages 26–46, Springer, 2024. 10.1007/978-3-031-66149-5_2

18 Marco Elio Gustavo Gario, Andrea Micheli. PySMT: a Solver-Agnostic Library for Fast Prototyping of SMT-Based Algorithms. In: *Proceedings of the 13th International Workshop on Satisfiability Modulo Theories (SMT)*, 2015.

19 Kalyan Muthukumar, Manuel V. Hermenegildo. Compile-Time Derivation of Variable Dependency Using Abstract Interpretation. *J. Log. Program.*, 13(2&3): 315–347, 1992. 10.1016/0743-1066(92)90035-2

20 Paulína Ayaziová, Dirk Beyer, Marian Lingsch Rosenfeld, Martin Spiessl, Jan Strejček. Software Verification Witnesses 2.0. In: *Model Checking Software – 30th International Symposium, SPIN 2024, Luxembourg City, Luxembourg, April 8-9, 2024, Proceedings.* Ed. by Thomas Neele and Anton Wijs. Lecture Notes in Computer Science, volume 14624, pages 184–203, Springer, 2024. 10.1007/978-3-031-66149-5_11

21 Arie Gurfinkel, Jorge A. Navas. Abstract Interpretation of LLVM with a Region-Based Memory Model. In: *Software Verification – 13th International Conference, VSTTE 2021, New Haven, CT, USA, October 18-19, 2021, and 14th International Workshop, NSV 2021, Los Angeles, CA, USA, July 18-19, 2021, Revised Selected Papers.* Ed. by Roderick Bloem, Rayna Dimitrova, Chuchu Fan, and Natasha Sharygina. Lecture Notes in Computer Science, volume 13124, pages 122–144, Springer, 2021. 10.1007/978-3-030-95561-8_8

22 Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, Jorge A. Navas. The SeaHorn Verification Framework. In: *Computer Aided Verification – 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I.* Ed. by Daniel Kroening and Corina S. Pasareanu. Lecture Notes in Computer Science, volume 9206, pages 343–361, Springer, 2015. 10.1007/978-3-319-21690-4_20

23 Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.* Ed. by Kathryn S. McKinley and Kathleen Fisher. Pages 1069–1084, ACM, 2019. 10.1145/3314221.3314590

24 Maria Christakis, Hasan Ferit Eniser, Holger Hermanns, Jörg Hoffmann, Yugesh Kothari, Jianlin Li, Jorge A. Navas, Valentin Wüstholz. Automated Safety Verification of Programs Invoking Neural Networks. In: *Computer Aided Verification – 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I.* Ed. by Alexandra Silva and K. Rustan M. Leino. Lecture Notes in Computer Science, volume 12759, pages 201–224, Springer, 2021. 10.1007/978-3-030-81685-8_9

25 Vincenzo Arceri, Greta Dolcetti, Enea Zaffanella. Speeding up Static Analysis with the Split Operator. In: *Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis, SOAP 2023, Orlando, FL, USA, 17 June 2023.* Ed. by Pietro Ferrara and Liana Hadarean. Pages 14–19, ACM, 2023. 10.1145/3589250.3596141

26 Guangsheng Fan, Liqian Chen, Banghu Yin, Wenyu Zhang, Peisen Yao, Ji Wang. Program Analysis Combining Generalized Bit-Level and Word-Level Abstractions. *Proc. ACM Softw. Eng.*, 2(ISSTA): 663–685, 2025. 10.1145/3728905

27 Zsófia ádám, Paulína Ayaziová, Levente Bajczi, Dirk Beyer, Marek Jankola, Marian Lingsch-Rosenfeld, Jan Strejček. Non-termination Witnesses and Their Validation. 2025.

28 Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, Virgile Prevosto. ANSI/ISO C Specification Language Version 1.22. 2025.

29 OASIS Open. Static Analysis Results Interchange Format (SARIF) Version 2.1.0. OASIS Standard, April 2020. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html

30 Dirk Beyer, Gidon Ernst, Martin Jonáš, Marian Lingsch-Rosenfeld. SV-LIB: A Standard Exchange Format for Software-Verification Tasks. Technical Report, 2025. https://www.sosy-lab.org/research/pub/2025-TR.SV-LIB_A_Standard_Exchange_Format_for_Software-Verification_Tasks.pdf

31 Jorge Navas, Arie Gurfinkel. Clam: Static Analyzer for LLVM Bitcode based on Abstract Interpretation. Version dev14, 2024. https://github.com/seahorn/clam

32 Pierre Lermu siaux, Benoît Montagu. Detection of Uncaught Exceptions in Functional Programs by Abstract Interpretation. In: *Programming Languages and Systems – 33rd European Symposium on Programming, ESOP 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 II.* Ed. by Stephanie Weirich. Lecture Notes in Computer Science, volume 14577, pages 391–420, Springer, 2024. 10.1007/978-3-031-57267-8_15

33 Simmo Saan, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, Helmut Seidl. Goblint: Abstract Interpretation for Memory Safety and Termination – (Competition Contribution). In: *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.* Ed. by Bernd Finkbeiner and Laura Kovács. Lecture Notes in Computer Science, volume 14572, pages 381–386, Springer, 2024. 10.1007/978-3-031-57256-2_25

34 Simmo Saan, Michael Schwarz, Julian Erhard, Sarah Tilscher, Karoliine Holter, Ralf Vogler, Kalmer Apinis, Vesal Vojdani. Goblint. https://github.com/goblint/analyzer. 10.5281/zenodo.5735006

35 Antoine Miné, Abdelraouf Ouadjaout, Matthieu Journault, Raphaël Monat, Francesco Parolini, Marco Milanese, Jérôme Boillot. Mopsa. Version 1.2, 2025. https://gitlab.com/mopsa/mopsa-analyzer

36 Matthieu Journault, Antoine Miné, Raphaël Monat, Abdelraouf Ouadjaout. Combinations of Reusable Abstract Domains for a Multilingual Static Analyzer. In: *Verified Software. Theories, Tools, and Experiments – 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers.* Ed. by Supratik Chakraborty and Jorge A. Navas. Lecture Notes in Computer Science, volume 12031, pages 1–18, Springer, 2019. 10.1007/978-3-030-41600-3_1

37 David Delmas, Abdelraouf Ouadjaout, Antoine Miné. Static Analysis of Endian Portability by Abstract Interpretation. In: *Static Analysis – 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings.* Ed. by Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi. Lecture Notes in Computer Science, volume 12913, pages 102–123, Springer, 2021. 10.1007/978-3-030-88806-0_5

38 David Delmas, Antoine Miné. Analysis of Software Patches Using Numerical Abstract Interpretation. In: *Static Analysis – 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings.* Ed. by Bor-Yuh Evan Chang. Lecture Notes in Computer Science, volume 11822, pages 225–246, Springer, 2019. 10.1007/978-3-030-32304-2_12

39 Francesco Parolini, Antoine Miné. Sound Abstract Nonexploitability Analysis. In: *Verification, Model Checking, and Abstract Interpretation – 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II.* Ed. by Rayna Dimitrova, Ori Lahav, and Sebastian Wolff. Lecture Notes in Computer Science, volume 14500, pages 314–337, Springer, 2024. 10.1007/978-3-031-50521-8_15

40 Marco Milanese, Antoine Miné. Under-Approximating Memory Abstractions. In: *Static Analysis – 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20-22, 2024, Proceedings.* Ed. by Roberto Giacobazzi and Alessandra Gorla. Lecture Notes in Computer Science, volume 14995, pages 300–326, Springer, 2024. 10.1007/978-3-031-74776-2_12

41 Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné. Easing maintenance of academic static analyzers. *Int. J. Softw. Tools Technol. Transf.*, 26(6): 673–686, 2024. 10.1007/s10009-024-00770-1

42 Matthieu Journault, Antoine Miné, Abdelraouf Ouadjaout. Modular Static Analysis of String Manipulations in C Programs. In: *Static Analysis – 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings.* Ed. by Andreas Podelski. Lecture Notes in Computer Science, volume 11002, pages 243–262, Springer, 2018. 10.1007/978-3-319-99725-4_16

43 Raphaël Monat. Try-Mopsa: Relational Static Analysis in Your Pocket. *ArXiv*, abs/2509.13128, 2025. arxiv.org/abs/2509.13128

44 Abdelraouf Ouadjaout, Antoine Miné. A Library Modeling Language for the Static Analysis of C Programs. In: *Static Analysis – 27th International Symposium, SAS 2020, Virtual Event, November 18-20, 2020, Proceedings.* Ed. by David Pichardie and Mihaela Sighireanu. Lecture Notes in Computer Science, volume 12389, pages 223–247, Springer, 2020. 10.1007/978-3-030-65474-0_11

45 Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné. Static Type Analysis by Abstract Interpretation of Python Programs. In: *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference).* Ed. by Robert Hirschfeld and Tobias Pape. LIPIcs, volume 166, pages 17:1–17:29, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. 10.4230/LIPIcs.ECOOP.2020.17

46 Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In: *Static Analysis – 28th International Symposium, SAS 2021, Chicago, IL, USA, October 17-19, 2021, Proceedings.* Ed. by Cezara Dragoi, Suvam Mukherjee, and Kedar S. Namjoshi. Lecture Notes in Computer Science, volume 12913, pages 323–345, Springer, 2021. 10.1007/978-3-030-88806-0_16

47 Milla Valnet, Raphaël Monat, Antoine Miné. Compositional Static Value Analysis for Higher-Order Numerical Programs (Artifact). *Dagstuhl Artifacts Ser.*, 11(2): 5:1–5:5, 2025. 10.4230/DARTS.11.2.5

48 Zhongyi Wang, Linyu Yang, Mingshuai Chen, Yixuan Bu, Zhiyang Li, Qiuye Wang, Shengchao Qin, Xiao Yi, Jianwei Yin. Parf: Adaptive Parameter Refining for Abstract Interpretation. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 – November 1, 2024.* Ed. by Vladimir Filkov, Baishakhi Ray, and Minghui Zhou. Pages 1082–1093, ACM, 2024. 10.1145/3691620.3695487

49 Markus Fleischmann, David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, Maria Christakis. Constraint-Based Test Oracles for Program Analyzers. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 – November 1, 2024.* Ed. by Vladimir Filkov, Baishakhi Ray, and Minghui Zhou. Pages 344–355, ACM, 2024. 10.1145/3691620.3695035

50 David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, Maria Christakis. Interrogation Testing of Program Analyzers for Soundness and Precision Issues. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 – November 1, 2024.* Ed. by Vladimir Filkov, Baishakhi Ray, and Minghui Zhou. Pages 319–330, ACM, 2024. 10.1145/3691620.3695034

4.3 Explainability in Static Analysis

Discussion Leader: Roberto Bagnara, University of Parma

Attendees

Roberto Bagnara, Valentin Barbazo, Anna Becchi, Patrick Cousot, Kihong Heo, Tim King, Naim Moussaoui-Remil, Jorge Navas.

Agenda

1. Define and shape the topic, and identify possible contributions.
2. Decide on potential next steps.

4.3.1 Preliminary Notes

There is some (limited) literature on the subject. See, e.g., [3] and [1].

4.3.2 Tentative Definition

For a static analysis tool, *explainability* denotes its ability to explain its findings to a given intended audience.

The most typical audience consists of development and QA teams, possibly with different skill sets and needs. This diversity should be considered, even though it was not directly discussed during the meeting.

4.3.3 What Has to Be Explained

Ideally, the tool should, for each of its findings or messages, explain – or at least make it easy for the audience to understand – the following aspects (the list may not be exhaustive):

1. **The bad thing to be avoided and why it is problematic.** Examples: memory leak (undesirable because it can lead to memory exhaustion); misaligned access to memory (undesirable because it is undefined behavior in C and C++).
2. **The specific finding.** Examples: A pointer to allocated memory went out of scope without freeing the associated memory; a pointer cast that may result in a misaligned pointer.
3. **The connection between the finding and the undesired behavior.** Example: If the pointer that went out of scope was the only reference to the allocated memory block, and that block was not deallocated, a memory leak occurs. Similarly, if a cast produces a misaligned pointer that is later dereferenced, undefined behavior occurs.
4. **Whether the finding has a definite or possible nature.** Examples: Is it certain that the deallocation function was not called before the pointer went out of scope? Is it certain that the cast was executed and could indeed result in a misaligned pointer?
5. **How the static analyzer decided to report the finding.** Example: Variable `ptr` received the result of `malloc` at program point p_1 and the returned value is definitely/possibly non-null. There exists a computation path leading to p_2, where `ptr` goes out of scope. The path is definitely/possibly feasible; `ptr` is definitely/possibly not copied to a memory location that survives `ptr`; the memory pointed to by `ptr` is definitely/possibly not deallocated. Hence, the memory block allocated at p_1 is definitely/possibly leaked.

Some of the above aspects may be trivial or redundant for certain audiences; in those cases, a reference to relevant documentation may suffice. Furthermore:

- most tools already provide the basic information (points 1–3);
- high-quality tools typically distinguish well between definite and possible findings (point 4);

Point 5 – explaining why a finding is reported – is particularly challenging and where further research is most needed.

4.3.4 Discussion Highlights

- **Iterated forward + backward analysis** (suggested by Cousot and others): useful for excluding false positives, but does not inherently explain the remaining positives.
- **Model checking / symbolic execution / automatic test-case generation:** can filter out false positives and also produce a witness, arguably the best form of explanation for any audience, at least for those cases where the witness is of manageable size.
- **Tracking dependencies during analysis** (suggested by Bagnara): record dependencies of abstract values alongside the values themselves, allowing users to understand the source of imprecision and refine the analysis through annotations or re-execution.
  Example: Here we have the addition of $a + b$ for which we cannot exclude an arithmetic overflow. Where do the (too large) intervals for a and b come from? They come from function parameters x_1 and x_2 and widening at program point p_1. If we present this information to the user, they may add assertions about the possible values of x_1 and x_2. We rerun the analysis (iterated forward + backward) and maybe the problem goes away. If it does not go away, maybe the issue is real or maybe the problem is in the widening at program point p_1. What can the user do with the information that "maybe the problem is the widening at program point p_1"? Adding an annotation at program point p_1 saying "tool, try harder here"?
- **Widening-based idea** (Cousot): record the delta used during widening. When an error is reached again, retry with a different widening delta.
- **Statistical prioritization** (Heo): use statistical analysis of invariants to estimate whether a warning is more or less likely to be a true positive, effectively measuring analysis precision.
- **Meta-analysis / "analysis of the analysis"** (Cousot): track dependencies and probabilistic data to record the analyzer's internal reasoning, recalling the ideas of A^2I (i.e., analysis of the analysis) [2].

4.3.5 Possible Action Items

1. Review the available literature.
2. Reflect further on the discussed approaches.
3. Consider the possible next steps (e.g., establishing a working group for the purpose of writing a paper, or organizing a new Dagstuhl Seminar on the topic).

References

1 Marcus Nachtigall, Lisa Nguyen Quang Do, Eric Bodden. Explaining Static Analysis – A Perspective. In: *34th IEEE/ACM International Conference on Automated Software Engineering – Workshop (ASEW)*, pages 29–32, 2019. 10.1109/ASEW.2019.00023

2 Patrick Cousot, Roberto Giacobazzi, Francesco Ranzato. A^2I: Abstract^2 Interpretation. *Proc. ACM Program. Lang.*, 3(POPL): 42:1–42:31, January 2019. 10.1145/3290355

3 Eric Bodden, Lisa Nguyen Quang Do. Explainable Static Analysis. In: *Software Engineering und Software Management 2018*, pages 205–208. Gesellschaft für Informatik, Bonn, 2018. ISBN 978-3-88579-673-2.

4.4 Combination of LLMs and Static Analysis

Discussion Leader: Greta Dolcetti, Ca' Foscari University of Venice

The adoption of Large Language Models (LLMs) and static analysis together is reshaping how developers interact with code analysis tools [1]. This emerging field offers compelling opportunities to enhance verification workflows, democratize complex analysis techniques, and bridge the gap between technical analysis outputs and developer understanding.

Our discussion group explored how to enhance the developer and user experience through natural language and how hybrid approaches can combine the strengths of these two areas.

4.4.1 Enhancing Developer and User Experience Through Natural Language

Traditional static analysis tools often present barriers to adoption through complex configuration requirements and technical jargon. LLMs might transform this landscape by serving as intuitive natural language interfaces. LLMs might help contextualize findings within specific code contexts. They can transform cryptic error messages into clear explanations [2] that relate directly to the developer's code, making static analysis more accessible to teams with varying levels of expertise. However, LLMs demonstrate notable limitations in certain critical analysis areas where traditional static analysis methods remain superior, highlighting the complementary rather than competitive nature of these technologies. Another promising application lies in automating the creation of custom analysis rules. LLMs can interpret human-readable specifications and generate corresponding analysis rules[3], dramatically reducing the manual effort and difficulty traditionally required for custom static analysis configurations, especially domain-specific requirements or legacy codebases requiring tailored analysis approaches. The integration becomes even more powerful when combined with synthesis techniques. Using frameworks like Syntax-Guided Synthesis (SyGus) [4], LLMs can generate analysis rules for specific patterns that organizations want to detect, then incorporate these rules into feedback systems that allow for human intervention and verification. This creates a collaborative workflow where LLMs handle the initial heavy lifting while human expertise ensures correctness and relevance. While manual verification remains necessary to ensure query correctness, the automated generation significantly accelerates the development cycle.

4.4.2 Hybrid Approaches: Combining Strengths

The most effective applications often emerge from hybrid approaches that leverage both LLM capabilities and traditional static analysis strengths. In computationally expensive analyses such as floating-point analysis, LLMs can provide initial hints to prune the search space, with traditional tools validating the results. This combination addresses scalability challenges while maintaining the correctness guarantees that static analysis provides. Multi-model strategies offer another promising direction. Running multiple LLMs on the same repository and checking for agreement increases confidence in results, as different models may excel in different contexts. This ensemble approach can improve overall accuracy while providing insights into the reliability of specific findings. Fine-tuning presents additional opportunities for improvement. Domain-specific training can enhance LLM performance in static analysis tasks, though this approach requires significant domain expertise and carefully curated training data. LLMs also show promise in addressing one of static analysis's persistent challenges: false positive management [5]. They can assist in triaging potential false positives, grouping findings with similar root causes, and generating counterexamples that help analysts understand analysis results. This capability can significantly reduce the manual effort required to derive actionable insights from static analysis outputs.

References

1 VN Ignatyev, NV Shimchik, DD Panov, AA Mitrofanov. Large language models in source code static analysis. 2024 Ivannikov Memorial Workshop (IVMEM), pages 28–35, IEEE, 2024.

2 John S. Y. Lee, Fengkai Liu, Tianyuan Cai. Code Debugging with LLM-Generated Explanations of Programming Error Messages. 2024 IEEE 13th International Conference on Engineering Education (ICEED), pages 1–5, IEEE, 2024. 10.1109/ICEED62316.2024.10923833

3 Chenyuan Yang, Zijie Zhao, Zichen Xie, Haoyu Li, Lingming Zhang. KNighter: Transforming Static Analysis with LLM-Synthesized Checkers. Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, 2025.

4 Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, Abhishek Udupa. Syntax-guided synthesis. 2013 Formal Methods in Computer-Aided Design, pages 1–8, IEEE, 2013. 10.1109/FMCAD.2013.6679385

5 Xueying Du, Kai Yu, Chong Wang, Yi Zou, Wentai Deng, Zuoyu Ou, Xin Peng, Lingming Zhang, Yiling Lou. Minimizing False Positives in Static Bug Detection via LLM-Enhanced Path Feasibility Analysis. *ArXiv*, abs/2506.10322, 2025.

4.5 Shifting Abstract Interpretation Left: Using Abstraction Interpretation Earlier in the Development Process

Discussion Leader: Michael Schwarz, National University of Singapore

Currently, abstract interpretation is often used only after a program has been developed. However, performing static analysis during development could provide significant benefits, ranging from easing development to detecting bugs much earlier or even preventing them altogether.

Our discussion group explored how to "shift left" abstract interpretation, the prerequisites for doing so, and promising future directions. One of the identified key prerequisites is to make abstract interpreters *incremental*, i.e., able to efficiently re-analyze a program after small changes. While there is existing work [3, 1, 2, 4], new execution models prevalent in, e.g., data science notebooks pose novel incremental analysis problems, as the effects of previous versions may linger in the environment. This poses challenges at all levels, ranging from defining a version-aware concrete semantics to designing suitable abstract domains. We also discussed why it may be an advantage to maintain results for several old program versions: It may help recover precision if a catastrophic loss of precision occurs for an intermediate version, or allow an incremental analysis to start from the version most semantically similar to the one to be analyzed. We also observed a relationship between incremental analysis and relational analyses relating multiple program versions. It may, for instance, be helpful to alert a user if, after a change intended to be purely cosmetic, the equivalence of the old and new versions can no longer be established.

Beyond incremental analysis, we discussed how to involve developers more directly in the analysis process. For example, they could supply candidate invariants during development. While one may not want to take such invariants at face value, they can guide the analysis in a sound way, e.g., by acting as widening thresholds [5]. Furthermore, it may be useful to provide immediate feedback to developers, e.g., when a call to a function is the first one for which the analysis cannot prove that the argument is non-null, or when a function argument has a value far outside the range of previous calls. While such situations may not be bugs

in themselves, they can serve as useful hints to the programmer that something may be amiss and may help avoid the introduction of bugs in the first place. Similarly, the analysis may flag instances where precision degrades significantly, prompting the user to write code more amenable to static analysis and thus likely also easier for humans to understand. Such interactions may help shape code in a way similar to style guides, except that they are tied to a more semantic property, namely whether the analyzer can reason about the code precisely.

We also discussed designing a specification language for developers to express their own properties of interest. To reach widespread adoption, such a specification language may follow the example of the popular semantic patching tool `Coccinelle` [6] by leveraging familiar syntax to lower the barrier to entry, perhaps by building on existing mechanisms such as `CodeQL` [7].

One challenge facing work on left-shifting abstract interpretation is the question of how to evaluate such techniques, especially when they involve interactions with users. We concluded that such evaluations may have to involve user studies, which can perhaps best be conducted by collaborating with other communities, such as the HCI community, as few researchers have experience both in sound static analysis and in conducting user studies.

References

1 Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, Helmut Seidl. Interactive abstract interpretation: reanalyzing multithreaded C programs for cheap. *International Journal on Software Tools for Technology Transfer*, 26(6): 647–667, 2024. 10.1007/s10009-024-00768-9

2 Benno Stein, Bor-Yuh Evan Chang, Manu Sridharan. Demanded abstract interpretation. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 282–295, ACM, 2021. 10.1145/3453483.3454044

3 Isabel Garcia-Contreras, José F. Morales, Manuel V. Hermenegildo. Incremental and Modular Context-sensitive Analysis. *Theory and Practice of Logic Programming*, 21(2): 211–243, 2021. 10.1017/S1471068420000496

4 Mamy Razafintsialonina, David Bühler, Antoine Miné, Valentin Perrelle, Julien Signoles. Reusing Caches and Invariants for Efficient and Sound Incremental Static Analysis (Extended Version). In: *39th European Conference on Object-Oriented Programming (ECOOP 2025)*, Bergen, Norway, 2025.

5 Nicolas Halbwachs, Yann-Erick Proy, Patrick Roumanoff. Verification of Real-Time Systems using Linear Relation Analysis. *Formal Methods in System Design*, 11(2): 157–185, 1997. 10.1023/A:1008678014487

6 Yoann Padioleau, Julia Lawall, René Rydhof Hansen, Gilles Muller. Documenting and automating collateral evolutions in linux device drivers. In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 247–260, ACM, 2008. 10.1145/1352592.1352618

7 Oege De Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, Julian Tibble. Keynote address: QL for source code analysis. In: *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16, IEEE, 2007.

5 Social Activities

Pietro Ferrara (University of Venice, IT), Liana Hadarean (Amazon Web Services – Seattle, US), Jorge Navas (Certora – Seattle, US), and Caterina Urban (INRIA & ENS Paris, FR)

During the first two days of the seminar, we organized a table football (Figure 1) and a tennis table (Figure 2) tournament. We formed teams of two participants for both tournaments; eight teams participated in each tournament (Table 1 reports the teams). We organized single-elimination tournaments starting with the quarter-finals. Since everybody wanted to play several games, we also organized semi-finals and finals for teams that had lost earlier matches, resulting in a total ordering in the final standings for all the teams. Teams were "randomly" paired using ChatGPT. The results of the two tournaments are reported by Tables 1 and 2. The competition was tough, and, as reported by Figure 1, one table football player got injured after the tournament. We hope that the Dagstuhl staff was able to fix it before the next seminar started.

In the late evenings, we played several board games (Figure 3), but unfortunately, we neither organized tournaments nor kept track of the results.

Figure 1 Table football tournament.

Figure 2 Tennis table tournament.

Figure 3 Board games.

Table 1 Teams.

Fussbal		Tennis table	
Name	Players	Name	Players
LiSA	Arceri, Ferrara	Seahorn	Navas, Wang/Gurfinkel
Function	Urban, Moussaoui	LiSA	Ferrara, Dolcetti
MOPSA	Monat, Mine	MOPSA	Mine', Monat
MemCAD	Barbazo, De Haro	YoYak	Minseok, Kihong
SeaHorn	Delmas, Becchi	Function	Urban, Moussaoui
GIDA	Denaro, Taibi	MemCAD	Barbazo, De Haro
PPL	Bagnara, Dolcetti	Llm	Dinella, Tan
Last minute tea (LMT)	Garcia, Dinella	Aws	Hadarean, Balliu

Table 2 Table football tournament.

	Quarter-finals			**Semi-finals**	
	MemCAD-PPL	12-10	1st-4th	MemCAD-Function	7-10
	Function-MOPSA	10-8	1st-4th	GIDA-LiSA	10-8
	GIDA-LMT	10-5	5th-8th	PPL-MOPSA	10-4
	SeaHorn-LiSA	6-10	5th-8th	LMT-SeaHorn	12-14
	Finals			**Standing**	
			1)	GIDA	
1st-2nd	GIDA-Function	10-6	2)	Function	
3rd-4th	MemCAD-LiSA	3-10	3)	LiSA	
5th-6th	PPL-SeaHorn	10-5	4)	MemCAD	
7th-8th	MOPSA-LMT	10-8	5)	PPL	
			6)	SeaHorn	
			7)	MOPSA	
			8)	LMT	

Table 3 Tennis table tournament.

Quarter-finals			**Semi-finals**		
	Seahorn-Llm	13-21	1st-4th	Llm-Aws	11-21
	Aws-YoYak	21-14	1st-4th	MOPSA-MemCAD	15-21
	LiSA-MOPSA	14-21	5th-8th	Seahorn-YoYak	21-6
	Function-MemCAD	12-21	5th-8th	LiSA-Function	21-19
Finals			**Standing**		
			1)	MemCAD	
1st-2nd	Aws-Memcad	20-22	2)	Aws	
3rd-4th	Llm-MOPSA	21-19	3)	Llm	
5th-6th	Seahorn-LiSA	21-12	4)	MOPSA	
7th-8th	YoYak-Function	21-22	5)	Seahorn	
			6)	LiSA	
			7)	Function	
			8)	Yoyak	

Participants

- Vincenzo Arceri
University of Parma, IT
- Roberto Bagnara
University of Parma, IT
- Musard Balliu
KTH Royal Institute of Technology – Stockholm, SE
- Valentin Barbazo
ENS, PSL University – Paris, FR
- Anna Becchi
USI – Lugano, CH
- Dirk Beyer
LMU München, DE
- Patrick Cousot
New York University, US
- Eva Darulova
Uppsala University, SE
- Charles De Haro
ENS, PSL University – Paris, FR
- David Delmas
Airbus – Toulouse, FR
- Giovanni Denaro
University of Milano-Bicocca, IT
- Werner Dietl
University of Waterloo, CA
- Elizabeth Dinella
Bryn Mawr College, US
- Greta Dolcetti
University of Venice, IT
- Pietro Ferrara
University of Venice, IT
- Isabel Garcia-Contreras
Black Duck – Calgary, CA
- Elisa Gonzalez Boix
VU – Brussels, BE
- Arie Gurfinkel
University of Waterloo, CA
- Liana Hadarean
Amazon Web Services – Seattle, US
- Kihong Heo
KAIST – Daejeon, KR
- Manuel Hermenegildo
IMDEA Software Institute – Madrid, ES
- Minseok Jeon
DGIST Institute of Science & Technology – Daegu, KR
- Tim King
Amazon Web Services – Santa Clara, US
- Valentina Lenarduzzi
University of Oulu, FI
- Debasmita Lohar
KIT – Karlsruher Institut für Technologie, DE
- Antoine Miné
Sorbonne University – Paris, FR
- Raphaël Monat
INRIA Lille, FR
- Naïm Moussaoui Remil
ENS, PSL University – Paris, FR
- Jorge Navas
Certora – Seattle, US
- Guido Salvaneschi
Universität St. Gallen, CH
- Michael Schwarz
National University of Singapore, SG
- Helmut Seidl
TU München – Garching, DE
- Davide Taibi
University of Southern Denmark – Odense, DK
- Tian Tan
Nanjing University, CN
participant Caterina Urban
INRIA & ENS Paris, FR
- Jingbo Wang
Purdue University – West Lafayette, US