Explainable Verification (Public)

September 24, 2025

Abstract

Summary in English: The cost of software failure is staggering. A rigorous approach to improving software quality and reducing bugs is sound program analysis based on abstract interpretation. In this project, we will push the state of the art in the abstract interpretation of multi-threaded programs written in complex mainstream programming languages, such as C. The goal of this project is to increase trust in the verification process through explainability. To trust the analysis specification, we will explain its mathematical foundations and essence; to increase trust in the implementation, we will enable tools to explain their results to other tools for validation; to gain the trust of the end users, we will enable tools to also explain their reasoning in human-readable form. To achieve these goals, the project will combine research in sound program analysis with compositional methods from programming language and category theory.

Summary in Estonian: Tarkvaratõrgete põhjustatud kulu on jahmatav. Range viis tarkvara kvaliteedi parandamiseks ning vigade vähendamiseks on korrektne programmianalüüs abstraktse interpretatsiooni abil. Selles projektis parendame abstraktse interpretatsiooni meetodeid keerulistes peavoolu programmeerimiskeeltes, nagu C, kirjutatud mitmelõimeliste programmide analüüsimiseks. Selle projekti eesmärk on verifitseerimisprotsessi usaldusväärsust seletatavuse kaudu suurendada. Analüüsi spetsifikatsiooni usaldamiseks selgitame selle matemaatilisi aluseid ja olemust; analüsaatori usalduse suurendamiseks võimaldame selle tulemuste selgitamist eesmärgiga, et need teiste tööriistade poolt valideerida; lõppkasutajate usalduse võitmiseks võimaldame analüsaatoritel selgitada oma arutluskäiku ka inimloetaval kujul. Nende eesmärkide saavutamiseks ühendab projekt korrektse programmianalüüsi meetodite uurimise kompositsiooniliste meetoditega programmeerimiskeelte- ja kategooriateooriast.

1 Scientific Background

The cost of software failure is staggering, estimated to be $2.41 trillion in 2022 [68]. Moving fast is tempting, but in the long run, quality assurance pays off [117]. An exemplary tale is that of the European avionics giant, Airbus, which is now reaping the rewards of a focus on quality [7]. Airbus is a leading backer [34, 70, 107] of a rigorous approach to software verification called **abstract interpretation** [35], which provides a unified view of **static program analysis**. When abstractly interpreting a program, one evaluates it using *abstract values* that over-approximate the set of *concrete values* that the program may see in actual execution. The theory spurred the development of various *numeric abstract domains*, capturing relationships between program variables [36, 79], and practical tools, such as Astreé [37], capable of verifying control software in Airbus aircraft [70].

Given the success of abstract interpretation in providing strong safety guarantees, the wider adoption of rigorous verification approaches would "dramatically reduce software vulnerability" [26]. Thus, one may wonder why abstract interpretation is not more widely used beyond the safety-critical domain? Unlike most source code analysis tools, abstract interpreters aim to be **sound** [105], meaning that they aim to never miss a bug (*no false negatives*). This means a sound tool must assume the system can behave in any possible way that has not been ruled out. This tends to result in *many false positives* that overwhelm users. Sound tools, thus, require much more sophisticated and computationally expensive analyses. Applying these methods on real-world programs is currently difficult, as they require human intervention and deep specialist knowledge [40]. As the sophistication increases, there is also an increased risk of the tool itself being incorrect. This is particularly true for *generative artificial intelligence*. While it can potentially help improving code [123], an industry white paper reports disconcerting trends for maintainability [52] and Stack Overflow, a popular Q&A site for programmers, has banned its use [108]. The impact of these tools remains to be seen, but it is clear that as code becomes easier to generate, it is increasingly important to precisely specify, clearly understand, and thoroughly verify its correctness.

In this project, we will respond to these concerns through a *mixture of foundational and applied research*, building upon and combining the individual *expertise of each project member and partner*, so as to be able to successfully answer our **main research question**:

How can we increase trust in the results of automated software verification tools, while at the same time making rigorous verification approaches more easily usable for developers?

Before explaining how we will answer this question, we first review the relevant **scientific background**.

Abstract interpretation based program analyses have traditionally been specified using **constraint systems** [35, 106] over abstract domains. The system speaks about *unknowns* representing points of interest in the program, while *constraints* formalise how the abstract values stored at each unknown are related. The system is solved by a *fixpoint solver* [45, 53, 62, 71]. If the domain is a lattice of finite height, the solver will eventually converge to a solution; otherwise, solving can be accelerated by applying **widening** and **narrowing** to deal with non-Noetherian analysis domains [9, 10, 14]

In the presence of function calls, the unknowns are **context-dependent** and the system is potentially infinite. The solver then starts with an initial query to some unknown of interest and explores the constraint system only as much as needed to determine the value of the queried unknown, computing a *partial solution* of the system. Challenges arise when one targets **multi-threaded** programs written in complex mainstream languages, such as C, as reasoning about the correctness of one thread is now dependent on how other threads behave. Considering all interleavings of threads does not scale; instead, **thread-modular** variants of abstract interpretation have been developed [78, 80, 82, 111, 112], including by us [100, 101], in which one can analyse each thread in isolation, and then combine the results for a global analysis. In loc. cit. we develop a "local" trace semantics to better justify the analyses in the challenging, but common setting of shared-variable concurrency. When threads communicate via shared globals, this results in "non-local" flows in the analysis—while we analyse a thread based on its local control flow, it can be influenced by other threads, and vice versa, through the globals. To deal with non-local flow in the computation of partial solutions of an infinite system over non-Noetherian domains, our analyses will be based on **side-effecting constraint systems** [12, 120].

In such systems, if an expression e constrains an unknown, the evaluation of e can also affect other unknowns using side-effects.

To increase trust in tools built on such methods and to make them more easily usable, the common theme in the proposed research is **explainability**—by *better explaining their mathematical foundations and essence*, by *better explaining the verification results of one tool to other tools*, and by *better explaining the verification results to developers.*

For **explaining the mathematical foundations and essence** of such tools, we will build on methods from **programming language** and **category theory**. These fields are inherently *compositional*, meaning that definitions, properties, and their proofs are naturally built from smaller building blocks, with larger ones naturally following from smaller ones. They provide us a useful starting point for explaining the foundations of sound program analysis tools in a mathematically natural and compositional way. In particular, we will explore existing and develop new **type-and-effect systems** [17, 66, 74, 124] as a compositional typing-based means to structure programs' correctness specifications and the abstractions they use, and their correctness proofs. This approach is particularly suitable because both the programs we verify (concurrent, multi-threaded) and the methods we use (side-effecting constraints and effectful solvers) contain side-effects. To relate such typing-based foundations back to the (concrete) semantics of the programs in question and the respective program analyses, we will employ the **(graded) monadic denotational semantics** of effectful programs [66, 75, 76, 81, 88]. We describe other, more specific related methods in individual work packages.

As automated verification of real-world programs remains a demanding challenge, the research community has turned to **cooperative verification**, where different tools focus on what they do best, and exchange information and **explain their results to other tools** to jointly verify programs [22, 23]. This is particularly important with the advent of large language models, allowing powerful but heuristic tools to generate invariants that can be validated by sound logic-based tools. The challenge in allowing tools to communicate is to find a common language for expressing invariants and counterexamples, as different tools might represent, e.g., the heap and thread scheduling information differently. It is, however, possible to speak of such things indirectly, e.g., if invariants involve pointer variables or by introducing ghost variables to represent the scheduler state. **Witnesses** are such method-agnostic proof objects that help other analysers validate analysis results. They are used at the International Contest on Software Verification, SV-COMP, to validate counter-examples [21, 24], and correctness invariants [25], though only for single-threaded programs.

Regarding **explaining program analysis results to users**, when a tool identifies a flaw in the program, it is possible to produce a counterexample execution trace that is useful for debugging the program and understanding the flaw. For instance, this has been critical to the success of model checking [32]. In contrast, when a sound analyser verifies the absence of errors in a program, it does not produce an equivalent human-readable artefact to explain this verdict. The challenge we shall be working on in this project is to expose how an automated verifier proves that a property holds along *all* possible executions of the program in a way that is interpretable by humans [16].

The project's findings will be implemented in and evaluated using Goblint [122], a sound program analysis tool actively developed by our group, in collaboration with TU Munich.

2 Objectives, hypotheses, methods

As stated above, our **goal** is to *increase trust in the results of automated software verification tools*, while at the same time *making rigorous verification approaches more easily usable* for developers. Our **hypothesis** is that we can achieve this, if:

O1 We understand the mathematical foundations underpinning the tools and can compositionally prove their correctness.

O2 We can produce machine-checkable witnesses that certify the program analysis results and have other tools validate them.

O3 We can explain the results of the tools to developers in an understandable way.

To achieve these **objectives**, we will combine the full range of programming language research and associated mathematical methods through a synthesis of theoretical, applied, and impact-focussed research, structured into the following **work packages (WPs)**.

WP1: Mathematical Foundations

While sound static program analysis has been successful, reasoning about its correctness is often non-compositional and cumbersome [83, 100], the mathematical foundations are at times unclear, and some aspects have not been studied in depth at all. We will address these concerns using a variety of compositional methods from programming language and category theory.

WP1.1 Modular Proofs for Thread-Modular Analyses, via Effects When justifying thread-modular analyses, the traditional approach is to use induction on the steps of an interleaving semantics [80]. As Mukherjee et al. [83] point out, this makes the proofs tedious as every abstraction needs to repeat history-based reasoning—instead they advocate for a thread-modular concrete semantics that can be conveniently abstracted, at the price of sacrificing precision. We have recently presented a thread-local concrete semantics that is more precise, but again requires history-based arguments for each abstraction [100]. We will investigate methods and abstractions to achieve modularity for such proofs, while retaining precision. Motivated by their compositional nature, and by the recent work on using single-threaded programs' analysis solutions for effect-typing [59], we will investigate the use of type-and-effect systems for compositionally structuring thread-modular analyses of concurrent programs, including their correctness proofs. We will also draw inspiration from Ahman et al's [6] work on typing-based reasoning about monotonic properties of state, where the main application is exactly to simplify history-based arguments. We then aim to connect all this back to the concrete thread-local semantics and thread-modular analysis using a monadic semantics.

WP1.2 Foundations of Side-Effecting Constraint Systems To better understand the practical applications and usefulness of side-effecting constraint systems, we will investigate their more precise mathematical essence. While at the moment solvers pass the meanings of side-effects to constraints as functional arguments, we will explore expressing such constraints more abstractly in terms monads and related structures [15, 72, 81]. We expect that this will shed new light both to the generation of such constraints from control-flow graphs (as a monadic semantics) and to the solvers of such constraints (in terms of their mathematical foundations and how to better structure their correctness proofs). A promising direction is to represent the constraints using monads for algebraic

effects [18, 88], because then effect handlers [2, 64, 89] would allow us to supply and update the meanings of side-effects compositionally. By distinguishing between programs', constraints', and solvers' effects, it will be easier to compositionally reason about allowed interactions and rule out unwanted ones. We will also investigate side-effecting constraint systems' algebraic essence, regarding in which categories and for which functors are their solutions (initial) algebras for [15], and how both their monadic nature and their partial solution aspect fit into the algebraic picture. A good understanding of their algebraic nature will offer new insights into compositional proof methods, such as deriving corresponding (co)induction principles [48, 54].

WP1.3 Interactive and Incremental Analysis Past work on incremental abstract interpretation has focussed either on single-threaded programs [109, 110] or accumulative approaches for shared global variables [104]. These methods were designed for analysing large codebases incrementally. Our more recent efforts [43] have concentrated on the incremental nature of interactive program analysis. When editing in an IDE, shared globals that over-approximate the entire editing history are insufficient. Since shared variables tend to influence significant portions of the program, restarting all globals results in only marginal performance gains over a complete re-analysis. For a holistic approach, we will investigate both the mathematical foundations and the algorithmics involved in achieving truly interactive and incremental analysis. We will build on the results of WP1.1-1.2 on a monadic, effectful understanding of program analysis to explore combining them with mathematical methods for modelling programs' interactive and incremental behaviour, such as update monads [3] and lenses [4], and more generally the areas of bidirected transformations [1] and optics [87]. For algorithms, we will explore ways to limit the reach of recomputation by restarted globals and will implement other algorithmic improvements to constraint system solving, on the one hand, guided by the mathematics we develop, and on the other hand, guiding the kinds of mathematics we need to develop to be able to reason about such algorithms [50, 51].

WP2: Automated Software Verification Algorithms

Engineering a sound static program analysis tool is a complex task. This includes already just defining and implementing the algorithms, before even thinking of verifying their correctness. Our goal is to make progress on both fronts. We will investigate symbolic algorithms for more precise and scalable thread-modular analysis, we will develop novel means for different verifiers to cooperate, and we will also work on machine-checked proofs.

WP2.1 Verified Solvers for Sound Program Analysis. Making program analysis tools more dependable is an active area of research, e.g., novel testing methods have been developed for [29, 67] and there have also been attempts at developing fully certified analysers [19, 27, 33, 39, 46, 63]. However, none of these target mainstream languages in their full complexity, and support side-effecting constraint systems and the techniques (like widening and narrowing) used in our tools. Instead, inspired by these works, and building on Seidl's work [56] on the formal verification of simplified versions of our solvers, we will work towards a better and formally verified feature-complete solver (e.g., based on the Top-Down [14, 71] or RLD solver [13, 103]) that also accommodates side-effecting constraints and other techniques crucial in our tools. As extrinsic, direct-style formal proofs quickly become as complex as pen-and-paper ones, if not worse, then using the monadic insights from WP1 we will explore solver verification in an intrinsic style, e.g., based on Ahman's past work on Dijkstra monads [5, 74] in F* [115], in which specifications

are expressed compositionally in a type-and-effect system and proofs are intertwined with code. We will explore using mainstream proof assistants, such as Coq [114] or Agda [113], and ones tailored for verifying effectful programs, such as F*. Franceschino et al. [46] present an abstract interpreter for a toy language in F*, and highlight the benefits of the intrinsic style.

WP2.2: Precise and Effective Thread-Modular Abstractions. In the past, we have made progress on thread-modular analysis [100, 101] and Goblint's performance on SV-COMP benchmarks is impressive. When analysing real-world C programs, however, we fail to capture which thread actions may happen in parallel when they rely on heap-allocated control structures to synchronise their actions. Based on studying the failures of SV-COMP tools, we have formed the working hypothesis that symbolic reasoning techniques, analogous to how we reason symbolically about heap-based locking patterns [102], could be adapted to reason about time-based patterns. We will develop both the required theory, and implement and evaluate these ideas in Goblint. We also expect to benefit from the theoretical developments in WP1.1, not only in order to better conduct the proofs, but also to guide the design of novel abstractions and algorithms. We will also investigate if we can practically apply the use of graded effects as an additional check on the analysis [59].

WP2.3 Witnesses and Cooperative Verification. Currently, no witness format exists for exchanging multi-threaded correctness invariants. Expressing invariants over all possible interleavings is challenging, and thus thread-modular or rely-guarantee invariants are more suitable. We have shown how to integrate invariants involving pointer variables [98]. We have also shared ideas on how witnesses may reason about thread scheduling [93], and researchers working on the Ultimate Automizer tool have extended our proposal to support ghost variables [20]. We are collaborating with them (together with TU Munich) to develop methods for expressing and validating more refined multi-threaded invariants. The use of ghost variables provides a complete proof method, but when the scheduler state is encoded in the program state, this does not provide sufficient structure for a thread-modular analyser to benefit from the witness. We will be able to show that exchange is possible in principle, but for exchange of more intricate correctness arguments, we will benefit from work, in WP1.1, on the essence of thread-modular reasoning. We will also work on witnesses for data race detection: we aim to extend the witness format towards exchanging information about multiple property violations, similar to the standardised analysis results interchange format, SARIF [44], to build CoOpeRace, a specialised instance of the cooperative verification platform CoVeriTeam [22] for race detection.

WP3: Explainable Verification in Practice

The usability aspects of sound static analysers deserve more research attention, as empirical studies suggest that poor explainability of analysis results is a serious obstacle preventing the wider adoption of static analysis tools [30, 42, 60, 85, 116].

WP3.1: Explainable Abstract Interpretation. Apinis and Vojdani [11] provide a framework for explaining the results of abstract interpretation using the general approach to deriving meta-analyses developed by Cousot et al. [38]. Using our approach has the benefit that the explanations are guaranteed to be semantically consistent with the result of the analysis. Meta-analyses in general are an active area of research, with a potentially

high impact on the usability of verification tools. E.g., there are meta-analyses that focus on quantifying precision loss [28, 49]. We will contribute to this research as follows. First, the framework of Apinis and Vojdani [11] was instantiated for a single-threaded analysis. We will extend it to explain thread-modular analysis [100, 101] in terms of underlying rely-guarantee reasoning [61]. Another important direction is to use our symbolic representation of the analysis computation, and identify which slices of the systems are relevant to being able to compute a given property. Then, we can investigate whether applying results from integer and linear optimisation can provide human-readable explanations, analogous to how generalised Farkas certificates [31] are used in probabilistic model checking [47].

WP3.2: Practical Research on Verification User Interfaces. Here we aim to enhance the usability of sound static analysis for developers by seamlessly integrating it into their workflow. While WP1.3 develops the algorithms for interactive analysis, there is also the need to research how verification tools should be presented in IDEs. Holter has started the development of an IDE integration, GobPie [57], for Goblint, using MagPie Bridge [73]. This allows program analysis results to be presented in the IDE, and updated as the program is edited, but it gives a static view of the analysis. Recent years have seen work on how to display analysis results in a more dynamic manner, giving a similar experience to conventional debuggers: VisuFlow [90], Multiverse Debugging [86, 118], and Symbolic Debugging [65]. We will explore how abstract reachability graphs of Saan [92] can be traversed through the Debug Adapter Protocol [77], so as to experience context-sensitive analysis through a debugger interface. We will then extend this method to provide a meaningful debugging experience for our thread-modular analyses. This can be further improved by the methods for extracting symbolic explanations from WP3.1. Finally, we will devise methods to pinpoint the root causes of the warnings generated by thread-modular analysis. We can draw inspiration from methods for single-threaded programs, such as dependency-based alarm diagnosis [91], the repositioning approach [84], and responsibility analysis [41].

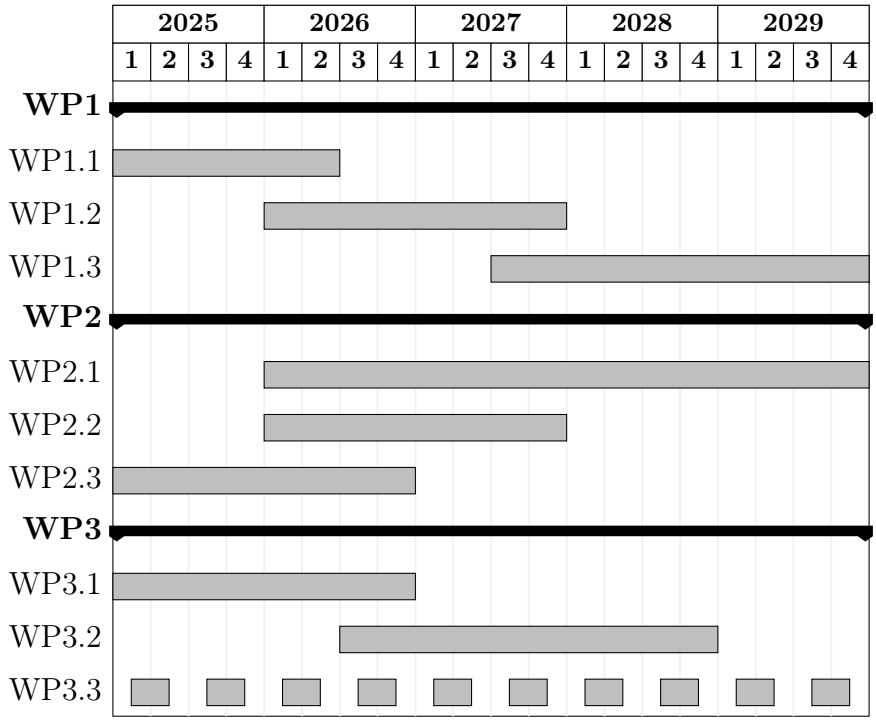
WP3.3: Improving Benchmarking and Evaluation Processes. To assess what tools can do in practice and incentivise the continuous improvement of these tools, the community has established verification contests [8]. We regularly compete at SV-COMP with the Goblint analyser, evaluating our approach [94], auto-tuning [95], memory safety analyses [96] and witness validation [97]. As we integrate the novel methods we develop in this project into our analyser, we will continue to compete annually in SV-COMP. We will also investigate how to evaluate tools on more realistic benchmarks, and aim to improve the benchmarking process that the community uses. In particular, how to extract representative kernels of verification challenges from real-world programs that can be submitted to SV-COMP, and recognising the importance of circling back to the original program [69], how to continuously evaluate research-based tools in real-world scenarios. Starting with the concurrent programs suite of Hong and Ryu [58], we aim to set up a framework—The Continuous Race—for live benchmarking data race verifiers on *real-world* programs using continuous integration workflows. There are interesting research challenges in designing an evaluation process that adapts the ideas from SV-COMP to a more realistic setting using the cooperative race detection artefacts from WP2.3.

Work plan: WP (co-)leads, Gantt chart, WP interactions

We assign the following **leads, co-leads, and partners** to individual work packages:

	Description	**Leads & Project Partners** & Co-leads
WP1	**Mathematical Foundations**	**Ahman & Kammar**
WP1.1	Modular Proofs	Nester
WP1.2	Foundations of Side-Effecting	Apinis
WP1.3	Incremental Analysis	Saan
WP2	**Verification Algorithms**	**Apinis & Seidl**
WP2.1	Verified Solvers	Ahman
WP2.2	Thread-Modular Reasoning	Vojdani
WP2.3	Witness Validation	Saan
WP3	**Explainable Verification**	**Vojdani & Lam**
WP3.1	Explainable Analysis	Apinis
WP3.2	Verification UIs	Holter
WP3.3	Evaluation	Saan

The **durations and order of work packages** is visualised in the following **Gantt chart**:



We also foresee a significant amount of **interaction between work packages**. First, *later work packages will naturally learn from and build on the results of earlier ones.* But in addition, there will also be *interaction between overlapping work packages.* In particular, the results of WP1 will be used to inform and guide the more applied work in WP2,3, whereas WP2,3 will give additional input to WP1 regarding problems faced in practice. This in turn means that WP1.1 and WP1.2 could be revisited even after their planned durations, or WP1.3 started earlier, based on the

findings and encountered challenges in the more applied WP2-3. In addition, all WPs, especially the more applied ones, will feed into the evaluation and benchmarking WP3.3, whose duration in the Gantt chart is partitioned to coincide with submitting to and participating at SV-COMPs.

3 Impact

3.1 Expected Results, Scientific Impact, and Future Research

The potential scientific impact of the project is significant, as the project addresses a critical and timely issue in software development in enhancing software quality and reducing costly errors [26, 117]. As a result, we will have better algorithms and more dependable tools for program analysis (WP2, WP3). We will have better understanding of our approaches mathematical foundations (WP1), and our tools will be easier for users to adopt (WP3).

Scientific impact. We will advance sound program analysis research by refining abstractions of concurrent programs and verifying real-world applications. If successful, the project's novel focus on usability and explainability in program analysis will result in the popularization of sound static analysis techniques. The project's other major focus on mathematical foundations will explain the essence of such analyses and why we can trust them, including how to best prove them correct, and it will connect sound program analysis to other major computer science areas, namely, type systems and category theory.

International collaboration & knowledge transfer. Our demonstrated collaboration with TU Munich and our active participation at SV-COMP has allowed the young researchers in our group to exchange knowledge with the elite of the European verification community. We will continue it in this project. Our dedication to contributing real-world benchmarks to SV-COMP will encourage other researchers and tool developers to address real-world software issues. The collaboration with our project partners, in particular the planned visits in both directions, will give our group access to invaluable know-how and expertise in their respective fields. Furthermore, Ohad Kammar has offered to deliver a lecture course on quasi-Borel spaces and the semantics of (statistical) probabilistic programming during his planned visits to Tartu. We will advertise this course also outside our group and institution, so that the whole Estonian scientific community could benefit from it at a time when probabilistic methods are becoming increasingly important with the emergence of ever more artificial intelligence tools.

Applicability. The open-source nature of our tools facilitates knowledge sharing and adoption by the broader community. We adhere to high software engineering standards, using continuous integration to ensure that the tools build, enabling others to use them. As a result, people are beginning to use Goblint as a baseline representative of the state-of-the-art in sound static race detection, e.g., an application of Goblint for ensuring the safety of OCaml bindings to C libraries was presented by Edwin Török of XenServer [119].

3.2 Importance of the project outside academia

By investigating the foundations and developing tools for improving software quality, including focusing on how to make it easier for developers to adopt such tools, the project will contribute towards a more reliable digital infrastructure. We have already emphasized that software errors have staggering economic costs. They also have serious societal implications because weaknesses in software can be exploited by hackers. As over two-thirds of vulnerabilities are due to programming errors, sound static analysis is important for improving cyber security [26].

Implementation plan and technology transfer. In Munich, the PI participated in the Artemis IA project MBAT [55], working on the analysis of automotive code [99]. Taking into account the lessons learned, our current proposal is designed with clear pathways from theoretical results to practical tools, with dedicated work packages for overcoming

the obstacles to industrial application, including explainability (WP3.1) and usability (WP3.2). Meanwhile, in WP3.3, we aim to improve the community processes for the evaluation of tools such that the transfer gap between research and industry is shortened. Beyond that, we will be proactive in engaging with industry partners. The PI presented Goblint at the Industry Day of the CHESS project [121], and given resources, we could participate and help organise industry events in Estonia. This also includes active participation in Meetups of the Estonian IT community, with interest in functional programming, where partnerships can be formed. We also plan to participate in one larger EU project with industrial partners. As a first attempt at this, Dietmar Pfahl, the professor of software systems here at Tartu, included us in a proposal for Horizon Cluster 4, where we will consider the use of AI for the heuristic parts of the unassuming process.

Teaching plan and knowledge transfer. The project will also ensure the preservation and enable the growth of programming language and software verification research and expertise at the University of Tartu and in Estonia more broadly. Our anticipated results and the knowledge we acquire and develop will have a direct impact on the education of future IT professionals in Estonia. Our group's involvement in mandatory undergraduate programming courses ensures that a considerable proportion of students entering the IT workforce will be equipped with the knowledge and skills acquired through our research. This will lead to better-trained professionals capable of developing high-quality software, and it will allow them to then also bring these good practices to their future employers, which include both private companies and government institutions. We will also release open source tools and develop educational resources based on our research to equip users with the necessary knowledge for effectively using analysers and understanding program correctness principles.

References

[1] Faris Abou-Saleh, James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. *Introduction to Bidirectional Transformations*, pages 1–28. Springer International Publishing, Cham, 2018. ISBN 978-3-319-79108-1. doi:10.1007/978-3-319-79108-1_1.

[2] Danel Ahman. Handling fibred algebraic effects. *Proc. ACM Program. Lang.*, 2 (POPL):7:1–7:29, 2018. doi:10.1145/3158095.

[3] Danel Ahman and Tarmo Uustalu. Update monads: Cointerpreting directed containers. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs, TYPES 2013, April 22-26, 2013, Toulouse, France*, volume 26 of *LIPIcs*, pages 1–23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2013. doi:10.4230/LIPICS.TYPES.2013.1.

[4] Danel Ahman and Tarmo Uustalu. Coalgebraic update lenses. In Bart Jacobs, Alexandra Silva, and Sam Staton, editors, *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics, MFPS 2014, Ithaca, NY, USA, June 12-15, 2014*, volume 308 of *Electronic Notes in Theoretical Computer Science*, pages 25–48. Elsevier, 2014. doi:10.1016/J.ENTCS.2014.10.003.

[5] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martínez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 515–529. ACM, 2017. doi:10.1145/3009837.3009878.

[6] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness: foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL):65:1–65:30, 2018. doi:10.1145/3158153.

[7] Liz Alderman. Airbus Pulls Further Ahead of Boeing in Global Plane Rivalry. *The New York Times*, February 2024. ISSN 0362-4331.

[8] Jade Alglave, Alastair F. Donaldson, Daniel Kroening, and Michael Tautschnig. Making Software Verification Tools Really Work. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 28–42, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-24372-1. doi:10.1007/978-3-642-24372-1_3.

[9] Gianluca Amato, Francesca Scozzari, Helmut Seidl, Kalmer Apinis, and Vesal Vojdani. Efficiently intertwining widening and narrowing. *Science of Computer Programming*, 120:1–24, 2016. doi:10.1016/j.scico.2015.12.005.

[10] Kalmer Apinis. *Frameworks for analyzing multi-threaded C.* PhD thesis, Institut für Informatik, Technische Universität München, June 2014.

[11] Kalmer Apinis and Vesal Vojdani. Context-Sensitive Meta-Constraint Systems for Explainable Program Analysis. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 453–472, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-30820-8. doi:10.1007/978-3-031-30820-8_27.

[12] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *Programming Languages and Systems*, pages 157–172. Springer, Berlin, Heidelberg, December 2012. doi:10.1007/978-3-642-35182-2_12.

[13] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 377–386, New York, NY, USA, June 2013. Association for Computing Machinery. ISBN 978-1-4503-2014-6. doi:10.1145/2491956.2462190.

[14] Kalmer Apinis, Helmut Seidl, and Vesal Vojdani. Enhancing Top-Down Solving with Widening and Narrowing. In Christian W. Probst, Chris Hankin, and René Rydhof Hansen, editors, *Semantics, Logics, and Calculi*, number 9560 in Lecture Notes in Computer Science, pages 272–288. Springer International Publishing, 2016. ISBN 978-3-319-27809-4 978-3-319-27810-0. doi:10.1007/978-3-319-27810-0_14.

[15] Steve Awodey. *Category Theory*. Oxford University Press, Inc., USA, 2nd edition, 2010. ISBN 0199237182.

[16] Christel Baier and Holger Hermanns. From Verification to Explanation (Track Introduction). In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Tools and Trends*, pages 1–7, Cham, 2021. Springer International Publishing. ISBN 978-3-030-83723-5. doi:10.1007/978-3-030-83723-5_1.

[17] Andrej Bauer and Matija Pretnar. An effect system for algebraic effects and handlers. In Reiko Heckel and Stefan Milius, editors, *Algebra and Coalgebra in Computer Science*, pages 1–16, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40206-7.

[18] Andrej Bauer and Matija Pretnar. Programming with algebraic effects and handlers. *J. Log. Algebr. Meth. Program.*, 84(1):108–123, 2015.

[19] Benedikt Becker and Claude Marché. Ghost Code in Action: Automated Verification of a Symbolic Interpreter. In Supratik Chakraborty and Jorge A. Navas, editors, *Verified Software. Theories, Tools, and Experiments*, pages 107–123, Cham, 2020. Springer International Publishing. ISBN 978-3-030-41600-3. doi:10.1007/978-3-030-41600-3_8.

[20] Manuel Bentele, Dominik Klumpp, and Frank Schüssele. Concurrency Correctness Witnesses with Ghosts, July 2023. Talk presented at the 1st Workshop on Verification Witnesses and Their Validation (VeWit 2023).

[21] Dirk Beyer and Karlheinz Friedberger. Violation Witnesses and Result Validation for Multi-Threaded Programs. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, Lecture Notes in Computer Science, pages 449–470, Cham, 2020. Springer International Publishing. ISBN 978-3-030-61362-4. doi:10.1007/978-3-030-61362-4_26.

[22] Dirk Beyer and Sudeep Kanav. CoVeriTeam: On-Demand Composition of Cooperative Verification Systems. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 561–579, Cham,

2022. Springer International Publishing. ISBN 978-3-030-99524-9. doi:10.1007/978-3-030-99524-9_31.

[23] Dirk Beyer and Heike Wehrheim. Verification Artifacts in Cooperative Verification: Survey and Unifying Component Framework. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, pages 143–167, Cham, 2020. Springer International Publishing. ISBN 978-3-030-61362-4. doi:10.1007/978-3-030-61362-4_8.

[24] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness validation and stepwise testification across software verifiers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 721–733, New York, NY, USA, August 2015. Association for Computing Machinery. ISBN 978-1-4503-3675-8. doi:10.1145/2786805.2786867.

[25] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: exchanging verification results between verifiers. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 326–337, New York, NY, USA, November 2016. Association for Computing Machinery. ISBN 978-1-4503-4218-6. doi:10.1145/2950290.2950351.

[26] Paul E Black, Lee Badger, Barbara Guttman, and Elizabeth Fong. Dramatically reducing software vulnerabilities: Report to the White House Office of Science and Technology Policy. Technical Report NIST IR 8151, National Institute of Standards and Technology, Gaithersburg, MD, November 2016.

[27] David Cachera and David Pichardie. A Certified Denotational Abstract Interpreter. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 9–24, Berlin, Heidelberg, 2010. Springer. ISBN 978-3-642-14052-5. doi:10.1007/978-3-642-14052-5_3.

[28] Marco Campion, Mila Dalla Preda, and Roberto Giacobazzi. Partial (In)Completeness in abstract interpretation: limiting the imprecision in program analysis. *Proceedings of the ACM on Programming Languages*, 6(POPL):59:1–59:31, January 2022. doi:10.1145/3498721.

[29] Ignacio Casso, José F. Morales, P. López-García, and Manuel V. Hermenegildo. Testing Your (Static Analysis) Truths. In Maribel Fernández, editor, *Logic-Based Program Synthesis and Transformation*, pages 271–292, Cham, 2021. Springer International Publishing. ISBN 978-3-030-68446-4. doi:10.1007/978-3-030-68446-4_14.

[30] Maria Christakis and Christian Bird. What Developers Want and Need from Program Analysis: An Empirical Study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 332–343, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3845-5. doi:10.1145/2970276.2970347.

[31] T. D. Chuong and V. Jeyakumar. A generalized Farkas lemma with a numerical certificate and linear semi-infinite programs with SDP duals. *Linear Algebra and its Applications*, 515:38–52, February 2017. ISSN 0024-3795. doi:10.1016/j.laa.2016.11.008.

[32] Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, November 2009. ISSN 0001-0782. doi:10.1145/1592761.1592781.

[33] Arthur Correnson and Dominic Steinhöfel. Engineering a Formally Verified Automated Bug Finder. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 1165–1176, New York, NY, USA, November 2023. Association for Computing Machinery. ISBN 9798400703270. doi:10.1145/3611643.3616290.

[34] P. Cousot. Avionic software verification by abstract interpretation. In *2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation. Special Workshop Theme: Formal Methods in Avionics, Space and Transport*, Poitiers, France, December 12–14 2007.

[35] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symp. on Principles of Programming Languages (POPL'77)*, pages 238–252, 1977. doi:10.1145/512950.512973.

[36] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '78, pages 84–96, New York, NY, USA, January 1978. Association for Computing Machinery. ISBN 978-1-4503-7348-7. doi:10.1145/512760.512770.

[37] Patrick Cousot, Radhia Cousot, Jerôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ Analyzer. In Mooly Sagiv, editor, *Programming Languages and Systems*, pages 21–30, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-31987-0. doi:10.1007/978-3-540-31987-0_3.

[38] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. A^2I: abstract^2 interpretation. *Proceedings of the ACM on Programming Languages*, 3(POPL):42:1–42:31, January 2019. doi:10.1145/3290355.

[39] Paulo Emílio de Vilhena, François Pottier, and Jacques-Henri Jourdan. Spy game: verifying a local generic solver in Iris. *Proceedings of the ACM on Programming Languages*, 4(POPL):33:1–33:28, January 2020. doi:10.1145/3371101.

[40] David Delmas and Jean Souyris. Astrée: From Research to Industry. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 437–451, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-74061-2. doi:10.1007/978-3-540-74061-2_27.

[41] Chaoqiang Deng and Patrick Cousot. Responsibility Analysis by Abstract Interpretation. In Bor-Yuh Evan Chang, editor, *Static Analysis*, Lecture Notes in Computer Science, pages 368–388, Cham, 2019. Springer International Publishing. ISBN 978-3-030-32304-2. doi:10.1007/978-3-030-32304-2_18.

[42] Lisa Nguyen Quang Do and Eric Bodden. Explaining Static Analysis With Rule Graphs. *IEEE Transactions on Software Engineering*, 48(2):678–690, February 2022. ISSN 1939-3520. doi:10.1109/TSE.2020.2999534. Conference Name: IEEE Transactions on Software Engineering.

[43] Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, and Helmut Seidl. Interactive Abstract Interpretation: Reanalyzing Whole Programs for Cheap, November 2022. arXiv:2209.10445 [cs].

[44] Michael C. Fanning and Laurence J. Golding. Static Analysis Results Interchange Format (SARIF) Version 2.1.0 Plus Errata 01. `https://docs.oasis-open.org/sarif/sarif/v2.1.0/errata01/os/sarif-v2.1.0-errata01-os-complete.html`, 8 2023. OASIS Standard incorporating Approved Errata. Latest stage available at `https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html`.

[45] Christian Fecht. Gena—a tool for generating prolog analyzers from specifications. In *Static Analysis: Second International Symposium, SAS'95 Glasgow, UK, September 25–27, 1995 Proceedings 2*, pages 418–419. Springer, 1995. doi:10.1007/3-540-60360-3_53.

[46] Lucas Franceschino, David Pichardie, and Jean-Pierre Talpin. Verified Functional Programming of an Abstract Interpreter. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, pages 124–143, Cham, 2021. Springer International Publishing. ISBN 978-3-030-88806-0. doi:10.1007/978-3-030-88806-0_6.

[47] Florian Funke, Simon Jantsch, and Christel Baier. Farkas Certificates and Minimal Witnesses for Probabilistic Reachability Constraints. In Armin Biere and David Parker, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 324–345, Cham, 2020. Springer International Publishing. ISBN 978-3-030-45190-5. doi:10.1007/978-3-030-45190-5_18.

[48] Neil Ghani, Patricia Johann, and Clément Fumex. Generic fibrational induction. *Log. Methods Comput. Sci.*, 8(2), 2012. doi:10.2168/LMCS-8(2:12)2012.

[49] Roberto Giacobazzi, Francesco Logozzo, and Francesco Ranzato. Analyzing Program Analyses. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 261–273, New York, NY, USA, January 2015. Association for Computing Machinery. ISBN 978-1-4503-3300-9. doi:10.1145/2676726.2676987.

[50] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 156–166, New York, NY, USA, June 2014. Association for Computing Machinery. ISBN 978-1-4503-2784-8. doi:10.1145/2594291.2594324.

[51] Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 748–766, New York, NY, USA, October 2015. Association for Computing Machinery. ISBN 978-1-4503-3689-5. doi:10.1145/2814270.2814305.

[52] William Harding and Matthew Kloster. Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality (incl 2024 projections) - GitClear. Technical report, GitClear, January 2024.

[53] Manuel V. Hermenegildo, Richard Warren, and Saumya K. Debray. Global flow analysis as a practical compilation tool. *The Journal of Logic Programming*, 13(4): 349–366, 1992. doi:10.1016/0743-1066(92)90053-6.

[54] Claudio Hermida and Bart Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. Comput.*, 145(2):107–152, 1998. doi:10.1006/INCO.1998.2725.

[55] Jens Herrmann et al. MBAT: Combined model-based analysis and testing of embedded systems, 2011 – 2014.

[56] Martin Hofmann, Aleksandr Karbyshev, and Helmut Seidl. Verifying a Local Generic Solver in Coq. In *Static Analysis*, pages 340–355. Springer, Berlin, Heidelberg, September 2010. doi:10.1007/978-3-642-15769-1_21.

[57] Karoliine Holter, Simmo Saan, and Vesal Vojdani. GobPie: A Magpie Bridge for Goblint. Available at https://github.com/goblint/GobPie, 2022.

[58] Jaemin Hong and Sukyoung Ryu. Concrat: An Automatic C-to-Rust Lock API Translator for Concurrent Programs. In *Proceedings of the 45th International Conference on Software Engineering*, ICSE '23, pages 716–728, Melbourne, Victoria, Australia, July 2023. IEEE Press. ISBN 978-1-66545-701-9. doi:10.1109/ICSE48619.2023.00069.

[59] Andrej Ivaskovic, Alan Mycroft, and Dominic Orchard. Data-flow analyses as effects and graded monads. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 15:1–15:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.FSCD.2020.15.

[60] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 672–681, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3.

[61] C. B. Jones. Specification and Design of (Parallel) Programs. *9th IFIP World Computer Congress (Information Processing 83)*, 1983. Publisher: Newcastle University.

[62] Niels Jørgensen. Finding fixpoints in finite function spaces using neededness analysis and chaotic iteration. In *International Static Analysis Symposium*, pages 329–345. Springer, 1994.

[63] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A Formally-Verified C Static Analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 247–259, New York, NY, USA, January 2015. Association for Computing Machinery. ISBN 978-1-4503-3300-9. doi:10.1145/2676726.2676966.

[64] Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In Greg Morrisett and Tarmo Uustalu, editors, *Proc. of 18th ACM SIGPLAN Int. Conf. on Functional Programming, ICFP 2013*, pages 145–158. ACM, 2013. ISBN 978-1-4503-2326-0. doi:10.1145/2500365.2500590.

[65] Nat Karmios, Sacha-Élie Ayoun, and Philippa Gardner. Symbolic Debugging with Gillian. In *Proceedings of the 1st ACM International Workshop on Future Debugging Techniques*, DEBT 2023, pages 1–2, New York, NY, USA, July 2023. Association for Computing Machinery. ISBN 9798400702457. doi:10.1145/3605155.3605861.

[66] Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 633–646. ACM, 2014. doi:10.1145/2535838.2535846.

[67] Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially testing soundness and precision of program analyzers. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 239–250, New York, NY, USA, July 2019. Association for Computing Machinery. ISBN 978-1-4503-6224-5. doi:10.1145/3293882.3330553.

[68] Herb Krasner. Cost of Poor Software Quality in the U.S.: A 2022 Report, 2022.

[69] Daniel Kroening and Michael Tautschnig. Automating Software Analysis at Large Scale. In Petr Hliněný, Zdeněk Dvořák, Jiří Jaroš, Jan Kofroň, Jan Kořenek, Petr Matula, and Karel Pala, editors, *Mathematical and Engineering Methods in Computer Science*, Lecture Notes in Computer Science, pages 30–39, Cham, 2014. Springer International Publishing. ISBN 978-3-319-14896-0. doi:10.1007/978-3-319-14896-0_3.

[70] Daniel Kästner, Reinhard Wilhelm, and Christian Ferdinand. Abstract Interpretation in Industry – Experience and Lessons Learned. In Manuel V. Hermenegildo and José F. Morales, editors, *Static Analysis*, pages 10–27, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-44245-2. doi:10.1007/978-3-031-44245-2_2.

[71] Baudouin Le Charlier and Pascal Van Hentenryck. A Universal Top-Down Fixpoint Algorithm. Technical Report 92–22, Institute of Computer Science, University of Namur, Belgium, 1992.

[72] P. B. Levy. *Call-By-Push-Value: A Functional/Imperative Synthesis*, volume 2 of *Sem. Structures in Computation*. Springer, 2004.

[73] Linghui Luo, Julian Dolby, and Eric Bodden. MagpieBridge: A General Approach to Integrating Static Analyses into IDEs and Editors (Tool Insights Paper). In Alastair F. Donaldson, editor, *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 21:1–21:25, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-111-5. doi:10.4230/LIPIcs.ECOOP.2019.21. ISSN: 1868-8969.

[74] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Catalin Hritcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP):104:1–104:29, 2019. doi:10.1145/3341708.

[75] Dylan McDermott and Tarmo Uustalu. Flexibly graded monads and graded algebras. In Ekaterina Komendantskaya, editor, *Mathematics of Program Construction - 14th International Conference, MPC 2022, Tbilisi, Georgia, September 26-28, 2022,*

Proceedings, volume 13544 of *Lecture Notes in Computer Science*, pages 102–128. Springer, 2022. doi:10.1007/978-3-031-16912-0_4.

[76] P.-A. Melliès. Parametric Monads and Enriched Adjunctions. Manuscript. https://www.irif.fr/∼mellies/tensorial-logic/8-parametric-monads-and-enriched-adjunctions.pdf, 2012.

[77] Microsoft. Debug Adapter Protocol. Available at https://microsoft.github.io/debug-adapter-protocol/.

[78] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2014. doi:10.1007/978-3-642-54013-4_3.

[79] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, March 2006. ISSN 1573-0557. doi:10.1007/s10990-006-8609-1.

[80] Antoine Miné. Static Analysis of Run-Time Errors in Embedded Real-Time Parallel C Programs. *Logical Methods in Computer Science*, Volume 8, Issue 1, March 2012. ISSN 1860-5974. doi:10.2168/LMCS-8(1:26)2012. Publisher: Episciences.org.

[81] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

[82] Raphaël Monat and Antoine Miné. Precise Thread-Modular Abstract Interpretation of Concurrent Programs Using Relational Interference Abstractions. In Ahmed Bouajjani and David Monniaux, editors, *Verification, Model Checking, and Abstract Interpretation*, Lecture Notes in Computer Science, pages 386–404, Cham, 2017. Springer International Publishing. ISBN 978-3-319-52234-0. doi:10.1007/978-3-319-52234-0_21.

[83] Suvam Mukherjee, Oded Padon, Sharon Shoham, Deepak D'Souza, and Noam Rinetzky. Thread-Local Semantics and Its Efficient Sequential Abstractions for Race-Free Programs. In Francesco Ranzato, editor, *Static Analysis*, pages 253–276, Cham, 2017. Springer International Publishing. ISBN 978-3-319-66706-5. doi:10.1007/978-3-319-66706-5_13.

[84] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. Repositioning of static analysis alarms. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 187–197, New York, NY, USA, July 2018. Association for Computing Machinery. ISBN 978-1-4503-5699-2. doi:10.1145/3213846.3213850.

[85] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. Explaining Static Analysis - A Perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 29–32, November 2019. doi:10.1109/ASEW.2019.00023. ISSN: 2151-0830.

[86] Matthias Pasquier, Ciprian Teodorov, Frédéric Jouault, Matthias Brun, Luka Le Roux, and Loïc Lagadec. Temporal Breakpoints for Multiverse Debugging.

In *Proceedings of the 16th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2023, pages 125–137, New York, NY, USA, October 2023. Association for Computing Machinery. ISBN 9798400703966. doi:10.1145/3623476.3623526.

[87] Matthew Pickering, Jeremy Gibbons, and Nicolas Wu. Profunctor optics: Modular data accessors. *Art Sci. Eng. Program.*, 1(2):7, 2017. doi:10.22152/PROGRAMMING-JOURNAL.ORG/2017/1/7.

[88] Gordon D. Plotkin and John Power. Notions of computation determine monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. doi:10.1007/3-540-45931-6_24.

[89] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Logical Methods in Computer Science*, 9(4:23), 2013.

[90] Lisa Nguyen Quang Do, Stefan Krüger, Patrick Hill, Karim Ali, and Eric Bodden. VisuFlow: A Debugging Environment for Static Analyses. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 89–92, May 2018. ISSN: 2574-1934.

[91] Xavier Rival. Abstract Dependences for Alarm Diagnosis. In Kwangkeun Yi, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 347–363, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-32247-4. doi:10.1007/11575467_23.

[92] Simmo Saan. Witness Generation for Data-flow Analysis. Master's thesis, Tartu Ülikool, 2020.

[93] Simmo Saan and Julian Erhard. Beyond Automaton-Based Witnesses and Location Invariants, April 2023. Talk presented at the 4th Workshop on Cooperative Software Verification (COOP 2023).

[94] Simmo Saan, Michael Schwarz, Kalmer Apinis, Julian Erhard, Helmut Seidl, Ralf Vogler, and Vesal Vojdani. Goblint: Thread-Modular Abstract Interpretation Using Side-Effecting Constraints (Competition Contribution). In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 438–442, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72013-1. doi:10.1007/978-3-030-72013-1_28.

[95] Simmo Saan, Michael Schwarz, Julian Erhard, Manuel Pietsch, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. Goblint: Autotuning Thread-Modular Abstract Interpretation (Competition Contribution). In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 547–552, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-30820-8. doi:10.1007/978-3-031-30820-8_34.

[96] Simmo Saan, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. Goblint: Abstract Interpretation

for Memory Safety and Termination (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, 2024. To appear.

[97] Simmo Saan, Julian Erhard, Michael Schwarz, Stanimir Bozhilov, Karoliine Holter, Sarah Tilscher, Vesal Vojdani, and Helmut Seidl. Goblint Validator: Correctness Witness Validation by Abstract Interpretation (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems*, 2024. To appear.

[98] Simmo Saan, Michael Schwarz, Julian Erhard, Helmut Seidl, Sarah Tilscher, and Vesal Vojdani. Correctness Witness Validation by Abstract Interpretation. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 74–97, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-50524-9. doi:10.1007/978-3-031-50524-9_4.

[99] Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, Peter Lammich, and Markus Müller-Olm. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 93–104, New York, NY, USA, January 2011. Association for Computing Machinery. ISBN 978-1-4503-0490-0. doi:10.1145/1926385.1926398.

[100] Michael Schwarz, Simmo Saan, Helmut Seidl, Kalmer Apinis, Julian Erhard, and Vesal Vojdani. Improving Thread-Modular Abstract Interpretation. In Cezara Drăgoi, Suvam Mukherjee, and Kedar Namjoshi, editors, *Static Analysis*, Lecture Notes in Computer Science, pages 359–383, Cham, 2021. Springer International Publishing. ISBN 978-3-030-88806-0. doi:10.1007/978-3-030-88806-0_18.

[101] Michael Schwarz, Simmo Saan, Helmut Seidl, Julian Erhard, and Vesal Vojdani. Clustered Relational Thread-Modular Abstract Interpretation with Local Traces. In Thomas Wies, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 28–58, Cham, 2023. Springer Nature Switzerland. ISBN 978-3-031-30044-8. doi:10.1007/978-3-031-30044-8_2.

[102] Helmut Seidl, Vesal Vojdani, and Varmo Vene. A Smooth Combination of Linear and Herbrand Equalities for Polynomial Time Must-Alias Analysis. In Ana Cavalcanti and Dennis R. Dams, editors, *FM 2009: Formal Methods*, Lecture Notes in Computer Science, pages 644–659, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-05089-3. doi:10.1007/978-3-642-05089-3_41.

[103] Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. *Compiler Design: Analysis and Transformation*. Springer Science & Business Media, 2012. ISBN 978-3-642-17548-0. doi:10.1007/978-3-642-17548-0.

[104] Helmut Seidl, Julian Erhard, and Ralf Vogler. Incremental Abstract Interpretation. In Alessandra Di Pierro, Pasquale Malacaria, and Rajagopal Nagarajan, editors, *From Lambda Calculus to Cybersecurity Through Program Analysis: Essays Dedicated to Chris Hankin on the Occasion of His Retirement*, pages 132–148. Springer International Publishing, Cham, 2020. ISBN 978-3-030-41103-9. doi:10.1007/978-3-030-41103-9_5.

[105] Ilya Sergey. What does it mean for a program analysis to be sound? https://blog.sigplan.org/2019/08/07/what-does-it-mean-for-a-program-analysis-to-be-sound/, 2019.

[106] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Application*, pages 189–233. Prentice-Hall, 1981.

[107] Jean Souyris. Industrial experience of abstract interpretation-based static analyzers. In Renè Jacquart, editor, *Building the Information Society*, pages 393–400, Boston, MA, 2004. Springer US. ISBN 978-1-4020-8157-6.

[108] Stack Exchange Inc. What is this site's policy on content generated by generative artificial intelligence tools? - Help Center. https://stackoverflow.com/help/ai-policy, 2024.

[109] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Demanded abstract interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, pages 282–295, New York, NY, USA, June 2021. Association for Computing Machinery. ISBN 978-1-4503-8391-2. doi:10.1145/3453483.3454044.

[110] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. Interactive Abstract Interpretation with Demanded Summarization. *ACM Transactions on Programming Languages and Systems*, February 2024. ISSN 0164-0925. doi:10.1145/3648441.

[111] Thibault Suzanne and Antoine Miné. From array domains to abstract interpretation under store-buffer-based memory models. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *Lecture Notes in Computer Science*, pages 469–488. Springer, 2016. doi:10.1007/978-3-662-53413-7_23.

[112] Thibault Suzanne and Antoine Miné. Relational thread-modular abstract interpretation under relaxed memory models. In Sukyoung Ryu, editor, *Programming Languages and Systems - 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings*, volume 11275 of *Lecture Notes in Computer Science*, pages 109–128. Springer, 2018. doi:10.1007/978-3-030-02768-1_6.

[113] Agda Development Team. The Agda Proof Assistant. https://wiki.portal.chalmers.se/agda/pmwiki.php, 20.03.2024.

[114] Coq Development Team. The Coq Proof Assistant. https://coq.inria.fr, 20.03.2024.

[115] F* Development Team. The F* Proof Assistant. https://fstar-lang.org, 20.03.2024.

[116] Daniil Tiganov, Lisa Nguyen Quang Do, and Karim Ali. Designing UIs for static-analysis tools. *Communications of the ACM*, 65(2):52–58, January 2022. ISSN 0001-0782. doi:10.1145/3486600.

[117] Adam Tornhill and Markus Borg. Code red: the business impact of code quality - a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt*, TechDebt '22, pages 11–20, New York, NY, USA, August 2022. Association for Computing Machinery. ISBN 978-1-4503-9304-1. doi:10.1145/3524843.3528091.

[118] Carmen Torres Lopez, Robbert Gurdeep Singh, Stefan Marr, Elisa Gonzalez Boix, and Christophe Scholliers. Multiverse Debugging: Non-Deterministic Debugging for Non-Deterministic Programs (Brave New Idea Paper). In *DROPS-IDN/v2/document/10.4230/LIPIcs.ECOOP.2019.27*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ECOOP.2019.27.

[119] Edwin Török. Targeted Static Analysis for OCaml C Stubs: eliminating gremlins from the code, July 2023. arXiv:2307.14909 [cs].

[120] Vesal Vojdani. *Static Data Race Analysis of Heap-Manipulating C Programs*. PhD thesis, University of Tartu, 2010.

[121] Vesal Vojdani. Goblint & GobPie: Towards Usable Data Race Verification. https://formela.fi.muni.cz/events/chess-industrial-day-2023, December 2023.

[122] Vesal Vojdani, Kalmer Apinis, Vootele Rõtov, Helmut Seidl, Varmo Vene, and Ralf Vogler. Static race detection for device drivers: the Goblint approach. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, pages 391–402, New York, NY, USA, August 2016. Association for Computing Machinery. ISBN 978-1-4503-3845-5. doi:10.1145/2970276.2970337.

[123] Nalin Wadhwa, Jui Pradhan, Atharv Sonwane, Surya Prakash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. Frustrated with Code Quality Issues? LLMs can Help! September 2023. doi:10.48550/arXiv.2309.12938.

[124] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, jan 2003. ISSN 1529-3785. doi:10.1145/601775.601776.