Formal Methods Engineering

Anonymous Author(s)

Abstract

Formal methods (FM) are experiencing a major resurgence in computer science. Developed and researched for decades, FM tools are now on the verge of large-scale industrial adoption. Besides such notable advances, this renewed interest poses major challenges for mainstream adoption: insufficient tool quality; fragmented, inconsistent infrastructure; and unclear cost-benefits.

We propose *Formal Methods Engineering (FME)*, an initiative to boosting the mainstream adoption of Formal Methods to overcome these challenges. Despite its utmost importance, related research is seldom considered in its own right; spans many fields, venues, and labels; making such research hard to frame and publish. FME aims at consolidating these research streams in a single research area. We identify three key areas of FME: (1) *solidifying FM tools*, (2) *consolidating FM tools and theory*, and (3) *best utilizing FM tools*. In each area, we survey existing work and formulate open problems.

1 Introduction

Formal Methods (FM)—studied for decades—are experiencing a major resurgence in computer science. While industrial use was primarily in hardware, Formal Methods now have software applications in large tech companies including Amazon [1], Facebook [12], and Microsoft [2]. Formal methods are booming, expanding their traditional domains to smart contracts [25], AI safety [31], zero-knowledge proofs [22], formalized mathematics [17] and more. Unfortunately, however, this renewed interest also exposes major challenges for their mainstream adoption.

Pressing challenges include insufficient tool quality, poor usability, fragmented and inconsistent infrastructure, and unclear cost-benefits [21]. Strikingly, many of these challenges are not only raised by practitioners but also shared among FM researchers.[1]

How can we overcome these challenges? We argue that a new initiative is necessary: *Formal Methods Engineering* (FME), an engineering field aimed at driving Formal Methods to mainstream use. We believe Formal Methods to be in a situation similar to that of software in the software crisis of the 1960s: critical for the future, yet not practical enough to be mainstream. As a response to the software crisis, the field of software engineering (SE) has emerged with the goal of systematizing software. Analogously, the response to FM's major challenges is Formal Methods Engineering aimed at systematizing Formal Methods. Despite the utmost importance of systematizing FM, related research is seldom considered in its own right; often challenging to frame and publish; related theories, techniques, and tools span many fields and venues.

Methodologically, FME aims at marrying the holistic, user-centric, and pragmatic approach of SE with the theoretical rigor of FM (c.f., Figure 1). As a key difference to FM and in accordance with SE, FME research is to be evaluated on real-world systems; less on benchmarks to guarantee impact and avoid any level of indirection. FME

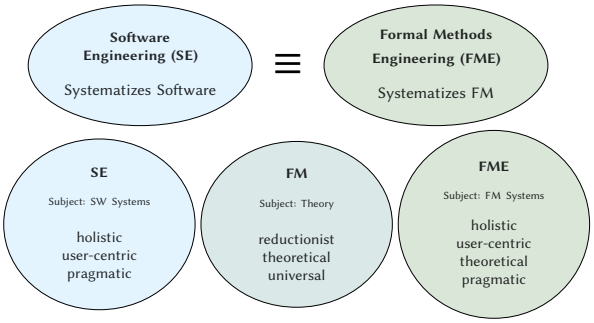

Figure 1: Taxonomy of Formal Methods Engineering (FME). Top: Analogy to Software Engineering (SE). Bottom: FME combines the strengths of SE and Formal Methods (FM).

values theory and novelty but in contrast to FM is more pragmatic, similar to the software engineering community.[2]

This paper categorizes FME research in three key areas that we believe to be crucial for mainstream formal methods. For each area, we outline its open problems, sketch solutions, and outline goals for the next decade.

Solidifying FM Tools Formal methods are traditionally applied highly safety and security-critical domains. Tool quality is hence crucial. We identify four aspects: reliability, stability, performance, and usability. Reliability has been addressed for various FM tools, while stability, performance, and usability are largely open.

Consolidating FM Tools and Theory Decades of FM research have led to a myriad of approaches, techniques, and tools. Unwanted side effects include redundancy in theory, fragmented tools, and poor interoperability. We argue for consolidating tools and theory to prevent unnecessary reinvention, and enable progress.

Best Utilizing FM Tools Given the available FM tools, a key challenge is to *best utilize* them. We identify three aspects: (a) unifying approaches, (b) validation portfolios, and (c) a theory for software reliability . Points (a) and (b) are practical approaches aimed at advancing FM application while point (c) aims at a more principled understanding of software reliability.

Before diving into either of the three, we first define Formal Methods. We thereby follow a lightweight interpretation of FM established by the 2020 FM expert survey [21].

Definition (Garavel et al. 2020)**.** Formal Methods are *all those techniques introduced for defensive programming.*

Defensive programming is software development under the assumption that things will go wrong. This FM definition is broad

[1] As revealed by the 2020 Expert Survey on Formal Methods conducted with 130 high-profile experts of the field including three Turing and 13 CAV award winners.

[2] Unlike in FM, theory without real-world impact is not a contribution; novelty is one of several criteria, not the dominant one.

and includes various tools for improving software reliability including static analysis, systematic testing, symbolic execution, type-checking, library interfaces, program assertions, loop invariants alongside the proof-producing methods and specification languages.

2 Solidifying FM Tools

Tool quality is key for the success of Formal Methods. We identify four aspects: (a) reliability, (b) stability, (c) performance, and (d) usability. This section addresses each of these four aspects.

Reliability is, thus far, the most addressed aspect. Influenced by Cadar and Donaldson's ICSE 2016 position paper "Analysing the Program Analyser",[3] various Formal Methods tools have been systematically tested, including program analyzers [19, 29], symbolic execution engines [30], type-checkers [14, 46], Datalog engines [36, 38], software model checkers [32, 57], program [8, 26, 58] and eBPF [49] verifiers, SMT [3, 9, 24, 37, 41, 42, 45, 48, 50–53, 55, 56], CHC [48], and LTL solvers [13]. Despite such substantial progress, several families of Formal Method tools remain untested. These include interactive theorem provers such as Coq, Isabelle, and Lean; automated theorem provers besides SMT solvers such as Vampire; refinement and dependent type systems such as F*, LiquidHaskell, and Flux; checkers for formal specification languages such as TLA+, Alloy, and PlusCal; program verifiers other than Dafny such as Viper, Boogie, Why3, and FramaC; SQL verifiers such as Cosette, Proof of SQL; Zero-knowledge provers such as snarkjs, halo2, and zksync; and provers for cryptographic protocols such as Tamarin. What most of these tools have in common is that their input languages are both expressive and rigid, making it challenging to generate syntactically and semantically valid (and meaningful) inputs. To test the remaining FM tools, we need flexible fuzzing methodologies that can be applied to a variety of tools at once and be adjusted to changes in the input format. We believe LLM-based and traditional fuzzing may be effectively combined to realize such tools. LLMs can help with the rigidity of the input languages by preemptively fixing errors while traditional fuzzing techniques guarantee out-of-distribution inputs. Besides, another important problem is automating fuzzing campaigns which, while effective and leading to hundreds of bug findings, are ad-hoc and not fully automated Researchers conduct fuzzing campaigns to evaluate their tools and often stop shortly after their research has been published. However, to ensure the detection of critical bugs, we need continuous, and fully automated fuzzing campaigns of FM tools similar to Google's OSS fuzz [23] for open-source tools. This yields several unique challenges: the limited resources of CI/CD pipelines; automated bug reduction and de-duplication; finding a balance between testing on existing inputs to avoid regressions and new ones to validate new functionality. We believe that portfolio approaches with fuzzers similar to the portfolio approaches that exist for SAT and SMT solvers may be suitable for realizing continuous FM fuzzing. Besides bug finding, tools for localizing bugs in and suggesting fixes in Formal Methods would relieve tool developers (often small teams). Leveraging advances in ML-based bug fixing agents to FM as well as component-based testing are promising avenues.

[3] An earlier research stream existed before Cadar and Donaldson's paper. Most notably the work on fuzzing SMT solvers by Biere et al. (2003).

Besides reliability, instability is a longstanding open problem. Small changes to input or source code cause FM tools to be slow, hang, or be incomplete on previously solved inputs. This issue has been recently observed in tools such as Amazon Zelkova and program verifiers such as Dafny, most of which are backed by SMT solvers. Existing research can be subdivided into instability wrt. input and instability wrt. source code changes in the tools. Research on input instability has focused on quantifying issues in Dafny [60] and mitigating them via trigger selection [34], and query restructuring [16]. Recent work moreover targets SMT-level instability via normalization [34], ensuring robustness to variable renaming and assertion or term reordering. Tool instability work has mainly focused on detection in SMT solvers [5, 6, 45] and model checkers [59], but mitigation strategies remain unaddressed. In addition, determining the extent and severity of instability is an open problem. Beyond diagnosis, identifying practical input subsets on which stability would benefit users is crucial; especially when algorithms are (necessarily) incomplete.

Performance gains in Formal Methods tools traditionally rely on heuristics, specialized algorithmic refinements, etc. Orthogonal to these, holistic, system-level performance gains that sped up tools broadly are under-explored. For example, researchers have accelerated SMT solvers by translating formulas into LLVM IR to leverage compiler optimizations [39]; by improving scalability by converting unbounded SMT theories into bounded ones [40], or by tailoring SAT solvers specifically for model checking algorithms [47]. Another crucial problem is to assess the performance of portfolio and other holistic approaches, such as SAT and SMT samplers, in applications. While these approaches have proven effective on benchmarks; even winning competitions, evaluations on benchmarks are a level of indirection to real-world applications. Instead we need evaluations on actual applications to prove the value of such approaches. Portfolio approaches also yield several open software engineering problems: (a) how they fare on out-of-sample inputs when machine learning selectors are used, (b) ensuring consistent performance on updating individual tools, (c) triaging bugs, and (d) optimizing portfolio approaches for low-latency applications.

Poor usability is one of the most frequently attributed barriers to Formal Methods. Recently a researcher even argued for a dedicated conference with a mandatory evaluation of usability.

> *We need a conference on Usable Formal Methods.*
>
> *Every submission to it must come with a verifier,a set of positive/negative examples for it, and, most importantly, an exercise. A review may give a positive score only if it provides a solution to the exercise, which is accepted by the verifier.*
>
> https://bsky.app/profile/ilyasergey.bsky.social/post/3lnxwc5lha22a

While attractive, this would be a significant effort for PC members to solve such exercises. However, user studies of this kind would be highly valuable and of interest to the (empirical) software engineering community. Decades of research in the HCI and empirical software engineering has been investigating usability. Criteria for usability of software are often subjective (depending on user's background) often challenging to be quantified [7]. However, the *ex-negativo* approach, i.e., investigating barriers could be more

fruitful. E.g., a recent work has been investigating the usability barriers in liquid types [20]. We believe more such work should be done and that top tier SE conferences are excellent homes for it.

Goals for 2035. *Within the next decade we envision: (a) flexible approaches to test the remaining untested FM tools, (b) continuous, fully automated fuzzing of all major FM tools, (c) major progress on the instability problem, (d) more holistic approaches for speeding up FM tools, and (e) measurable improvements in FM usability.*

3 Consolidating FM Tools and Theory

Decades of research in Formal Methods have led to substantial progress but also yielded a very high number of approaches, theories, and tools. Unintended side-effects include redundancy in theory, fragmentation of tools, and a lack of interoperability. These issues slow progress and challenge novices entering the field.

A key challenge is the consolidation of tools. There is a significant overlap. For instance, Frama-C, VeriFast, VCC, and Infer are static analyzers for C/C++; Boogie, Why3, and F* are intermediate verification languages; TLA+, Alloy, Event-B, and I/O Automata all realize system modeling and analysis; Lean, Coq, Isabelle, and HOL4 are interactive theorem provers; and Certora, Move Prover, 2vyper, and Solc-verify target smart contracts. While each tool has distinctive strengths, the interoperability between them is limited. Artifacts and results are rarely reusable across different tools. Moreover, tool choices are influenced equally by technical features and more by the academic environment in which users were trained.[4] Studies and dedicated translation tools may be able to mitigate this issue. Another goal is to develop shared input formats similar to SMT-LIB for SMT solvers or OpenTheory for theorem provers.

Redundancy also exists in theory. Type soundness, for example, has been repeatedly proved in Coq, Agda, Lean, and F*, each using its own syntax and semantics. Separation logic appears in many incompatible forms across Viper, Iris, VeriFast, and Bedrock. Algebraic structures such as groups and rings are redefined in every proof assistant. Similarly, secrecy and authentication are formalized differently in ProVerif, Tamarin, CryptoVerif, and F*, hindering comparison and reuse. Just as with tools, we need systematic efforts to consolidate and connect theoretical results. Related research connecting between the different formalisms and theory could prevent reinventing the wheel.

Goals for 2035. *Within the next decade, we envision a research stream on the consolidation of Formal Methods tools and theory.*

4 Best Utilizing FM Tools

Given solid and consolidated FM tools, a key question is how to *best utilize* them. We describe three different approaches: (a) unifying methods, (b) validation portfolios, and (c) theory of software reliability as a foundational framework.

unifying methods are tools that tightly integrate testing and verification. Instead of viewing testing and verification as separate, we argue that one should focus on their complementarity and pragmatically integrate them within the same tool infrastructure. A recent blog post by a Galois engineer suggests that testing may even boost the use of more expensive techniques such as verification by delivering intermediate results for customers:

> *Formal methods people like to make fun of testing. But the advantage of testing is that it's easy to run a few tests and find some bugs. If you write a few more tests, you'll probably find a few more bugs. That gives a very attractive cost-benefit curve [...]*
>
> https://www.galois.com/articles/what-works-and-doesnt-selling-formal-methods

There are two ways of mitigating the gap between testing and verification: sacrificing the completeness of verification tools and making automated testing more systematic to yield stronger assurances. Several tools have been proposed for the first [11, 28], and fewer tools exist for the second [15, 18, 51]. However, almost all these tools treat testing and verification as distinct. We argue for tools with tighter integration of the two techniques within tools.

A *validation portfolio (VP)* is a system that, given a software artifact, applies different FM tools under resource constraints and outputs bugs, proofs, and other certificates. The front end of a VP can be chat-based, allowing users to specify needs and constraints; the back end determines a schedule of applicable FM tools subject to constraints. VPs generalize portfolio approaches for SAT/SMT solvers [43, 54] and model checkers [35] via transpilers for the different tool families' input languages. A first application scenario for VPs could be medium-sized C codebases, where the methods include black-box testing, greybox testing (e.g., AFL), symbolic execution (e.g., KLEE), and bounded model checking (e.g., ESBMC). We believe validation portfolios can boost the adoption of FM tools.

A principled way to understand the net benefits of FM tools *a priori*[5] is to devise a *theory of software reliability (TSR)*. Given a software system of interconnected components $\mathcal{C} = \{C_1, \cdots, C_n\}$, and different formal methods $\mathcal{M} = \{M_1, \cdots, M_m\}$, the problem is to map methods to components, maximizing guarantees subject to cost constraints. Guarantees can be measured by code or input coverage metrics. Costs can be measured in time or compute units. The purpose of TSR is (a) to provide a heuristic for informing choices about the net benefits of applying FM tools to a given software system (e.g., to inform validation portfolios), and (b) to provide a theoretical framework for bridging the gap between vastly different Formal Methods (e.g., unifying testing and verification).

Goals for 2035. *Within the next decade, we envision (a) more unifying approaches, (b) validation portfolios, and (c) a theory of software reliability.*

5 Related Work

Formal Methods Engineering is related to several other initiatives. Perhaps the closest is proof engineering (PE) [44], a recent initiative focusing on the engineering challenges of software proof production and maintenance. While both focus on engineering, PE is restricted to interactive theorem provers, whereas FME is not. Lightweight formal methods (LFM) [27] is another related initiative proposed about 25 years ago. Its focus was on *partiality*: instead of full verification of a system, LFM promotes the verification of selected critical components, bounded analyses, and automation. FME and LFM share the goal of boosting the adoption of Formal

[4] A prominent example is the co-existence of Coq and Isabelle; two largely disjoint universes of proof assistants.

[5] I.e., before applying a particular method to the software system at hand.

Methods, but FME focuses on tools rather than methods. Moreover, LFM came earlier; most of today's FM tools are in fact LFM tools. Less directly related, but similar in name, are Formal Engineering Methods (FEM), whose goal was to popularize the use of formal approaches within software engineering. Finally, the work of Krishnamurthi and Nelson (2019) on Formal Methods with *the human in the loop* relates to our proposal on tool quality.

6 Conclusion

We propose *Formal Methods Engineering (FME)*, an initiative to boost FM's mainstream adoption; consolidating existing research that has several labels, spans many communities and venues. We envision three key challenges of FME: (1) solidfiying FM tools, (2) consolidation of FM tools and theory, and (3) best utilizing FM tools. For each of these, we outlined existing work and open challenges. FME complements the existing novelty, theory driven Formal Method communities providing little incentives for research on these vital challenges. FME enables pragmatic, user-centric, open-minded applied formal methods work with the high impact standards of the SE community. We hope to attract researchers and practitioners from adjacent communities, especially those from Formal Methods, contribute their best work at top-tier software engineering venues.

References

[1] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. 2018. Semantic-based Automated Reasoning for AWS Access Policies using SMT. In *FMCAD '18*.

[2] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. 2004. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *IFM '04*.

[3] Murphy Berzish, Yunhui Zheng, and Vijay Ganesh. 2017. Z3str3: A String Solver with Theory-aware Branching. In *FMCAD '17*.

[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded model checking. (2003).

[5] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *CAV '18*.

[6] Mauro Bringolf, Dominik Winterer, and Zhendong Su. 2022. Finding and Understanding Incompleteness Bugs in SMT Solvers. In *ASE '22*.

[7] John Brooke. 1996. *SUS: a "quick and dirty" usability scale.* Technical Report.

[8] Alexandra Bugariu, Arshavir Ter-Gabrielyan, and Peter Müller. 2023. Identifying Overly Restrictive Matching Patterns in SMT-based Program Verifiers (Extended Version). (2023).

[9] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically testing implementations of numerical abstract domains. In *ASE '18*.

[10] Cristian Cadar and Alastair Donaldson. 2016. Analysing the Program Analyser. In *ICSE '16*.

[11] Cristiano Calcagno, Dino Distefano, Joxan Jaffar, and Peter W. O'Hearn. 2015. Moving fast with software verification. In *NASA Formal Methods Symposium (NFM) (LNCS)*. 3–11.

[12] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2015. Moving Fast with Software Verification. In *NFM '15*.

[13] Luiz Carvalho, Renzo Degiovanni, Maxime Cordy, Nazareno Aguirre, Yves Le Traon, and Mike Papadakis. 2024. SpecBCFuzz: Fuzzing LTL Solvers with Boundary Conditions. In *ICSE '24*.

[14] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. 2022. Finding Typing Compiler Bugs. In *PLDI '22*.

[15] Koen Claessen and John Hughes. 2000. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *ICFP '00*.

[16] Joseph W. Cutler, Emina Torlak, and Michael W. Hicks. 2023. Improving the Stability of Type Soundness Proofs in Dafny. In *Dafny Workshop '23*.

[17] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language (System Description). (2021).

[18] Jonas Duregard, Patrick Jansson, and Meng Wang. 2012. FEAT: Functional Enumeration of Algebraic Types. In *Haskell '12*.

[19] Markus Fleischmann, David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Constraint-Based Test Oracles for Program Analyzers. In *ASE '24*.

[20] Catarina Gamboa, Abigail Reese, Alcides Fonseca, and Jonathan Aldrich. 2025. Usability Barriers for Liquid Types. (2025).

[21] Hubert Garavel, Maurice H. Ter Beek, and Jaco Van de Pol. 2020. The 2020 Expert Survey on Formal Methods. In *FMICS '20*.

[22] Oded Goldreich and Hugo Krawczyk. 1990. On the Composition of Zero-Knowledge Proof Systems. In *ICALP '90*.

[23] Google. 2017. OSS-Fuzz: Continuous Fuzzing for Open Source Software.

[24] Hichem Rami Ait El Hara, Guillaume Bury, and Steven de Oliveira. 2022. Alt-Ergo-Fuzz: A fuzzer for the Alt-Ergo SMT solver. In *JFLA '22*.

[25] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, and Grigore Roşu. 2018. KEVM: A Complete Formal Semantics of the Ethereum Virtual Machine. In *CSF '18*.

[26] Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamari'c, Neha Rungta, and Emina Torlak. 2022. Testing Dafny (experience paper). In *ISSTA '22*.

[27] Daniel Jackson. 2001. Lightweight Formal Methods. In *FME '01*.

[28] Bart Jacobs and Frank Piessens. 2011. The VeriFast program verifier. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (LNCS, Vol. 6605)*. 41–55.

[29] David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Interrogation Testing of Program Analyzers for Soundness and Precision Issues. In *ASE '24*.

[30] Timotej Kapus and Cristian Cadar. 2017. Automatic testing of symbolic execution engines via program generation and differential testing. In *ASE '17*.

[31] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *CAV '17*.

[32] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In *ISSTA '19*.

[33] Shriram Krishnamurthi and Tim Nelson. 2019. The Human in Formal Methods. In *FME '19*.

[34] K. Rustan M. Leino and Cl'ement Pit-Claudel. 2016. Trigger Selection Strategies to Stabilize Program Verifiers. In *CAV '16*.

[35] Zhengyang Lu, Po-Chun Chien, Nian-Ze Lee, Arie Gurfinkel, and Vijay Ganesh. 2025. Btor2-Select: Machine Learning Based Algorithm Selection for Hardware Model Checking. In *CAV '25*, Vol. 15931. 296–311.

[36] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic testing of Datalog engines. In *ESEC/FSE '21*.

[37] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *ESEC/FSE '20*.

[38] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *ISSTA '23*.

[39] Benjamin Mikek and Qirun Zhang. 2023. Speeding up SMT Solving via Compiler Optimization. In *ESEC/FSE '23*.

[40] Benjamin Mikek and Qirun Zhang. 2024. SMT Theory Arbitrage: Approximating Unbounded Constraints using Bounded Theories. In *PLDI '24*.

[41] Aina Niemetz, Mathias Preiner, and Clark Barrett. 2022. Murxla: A Modular and Highly Extensible API Fuzzer for SMT Solvers. In *CAV '22*.

[42] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. In *OOPSLA '21*.

[43] Nikhil Pimpalkhare, Federico Mora, Elizabeth Polgreen, and Sanjit A. Seshia. 2021. MedleySolver: Online SMT Algorithm Selection. In *24th International Conference on Theory and Applications of Satisfiability Testing (SAT) (Lecture Notes in Computer Science, Vol. 12831)*. Springer, 453–470.

[44] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. (2019).

[45] Joseph Scott, Federico Mora, and Vijay Ganesh. 2020. BanditFuzz: Fuzzing SMT Solvers with Reinforcement Learning. In *CAV '20*.

[46] Thodoris Sotiropoulos, Stefanos Chaliasos, and Zhendong Su. 2024. API-Driven Program Synthesis for Testing Static Typing Implementations. (2024).

[47] Yuheng Su, Qiusong Yang, Yiwei Ci, Yingcheng Li, Tianjun Bu, and Ziyu Huang. 2025. Deeply Optimizing the SAT Solver for the IC3 Algorithm. In *CAV '25*.

[48] Anzhela Sukhanova and Valentyn Sobol. 2023. HornFuzz: Fuzzing CHC solvers. In *EASE '23*.

[49] Hao Sun and Zhendong Su. 2024. Validating the eBPF Verifier via State Embedding. In *OSDI '24*.

[50] Maolin Sun, Yibiao Yang, Ming Wen, Yongcong Wang, Yuming Zhou, and Hai Jin. 2023. Validating SMT Solvers via Skeleton Enumeration Empowered by Historical Bug-Triggering Inputs. In *ICSE '23*.

[51] Dominik Winterer. 2024. Grammar-based Enumeration of SMT Solvers for Correctness and Performance. In *OOPSLA '24*.

[52] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusal Effectiveness of Type-Aware Operator Mutation. In *OOPSLA '20*.

[53] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI '20*.
[54] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2008. SATzilla: portfolio-based algorithm selection for SAT. *J. Artif. Int. Res.* 32, 1 (June 2008), 565–606.
[55] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *ISSTA '21*.
[56] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Skeletal Approximation Enumeration for SMT Solver Testing. In *ESEC/FSE '21*.
[57] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and understanding bugs in software model checkers. In *ESEC/FSE '19*.
[58] Chengyu Zhang and Zhendong Su. 2024. SMT2Test: From SMT Formulas to Effective Test Cases. In *OOPSLA '24*.
[59] Chengyu Zhang, Minquan Sun, Jianwen Li, Ting Su, and Geguang Pu. 2021. Feedback-Guided Circuit Structure Mutation for Testing Hardware Model Checkers. In *ICCAD '21*.
[60] Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. 2023. Mariposa: Measuring SMT Instability in Automated Program Verification. In *FMCAD '23*.