Dominik Winterer
ETH Zurich, Switzerland

Research Statement

November 2024

Software is pervasive in our everyday lives. Unfortunately, software bugs are *still* regularly causing tremendous harm.[1] *I work towards a future where critical software bugs are extremely rare.* Of key importance to achieve this goal are Formal Methods, mathematically rigorous techniques to prevent dangerous bugs in software. Despite decades of research with substantial breakthroughs, Formal Methods have not yet been widely adopted. One major reason is that Formal Methods tools are often *themselves* incorrect, unstable, slow, or not user-friendly. **My career goal is to develop Formal Methods Engineering (FME)—a dedicated discipline for making Formal Methods more correct, stable, performant, and usable.** FME entails three major research challenges: *(C1) Making SMT solvers—the foundation of many Formal Methods tools—much more solid, stable and performant, (C2) Unifying testing and verification, and (C3) Specification Engineering*. Similar to software engineering for software, FME transforms the use of Formal Methods from an ad hoc activity to a structured approach.

I am a researcher in software engineering and programming languages. My research yields widely applicable, practical, and ambitious methodologies. Through my Ph.D. research, I made a significant contribution towards developing Formal Methods Engineering. I focused on solidifying Satisfiability Modulo Theory (SMT) solvers—one of the most powerful classes of Formal Methods. SMT solvers are foundational for software research and industry and are almost always at the heart of their client applications which are often security/safety-critical [2, 4, 9, 8]. Moreover, SMT has the potential to serve as an intermediate representation for judging the reasoning abilities of LLMs. Incorrect, incomplete, or slow SMT solvers can have severe undesirable effects. Hence, it is crucial that SMT solvers be correct and performant. Before my Ph.D. research body, there was uncertainty about SMT solvers: Unfounded trust, false assumptions, and many hidden soundness bugs—the most severe category of bugs. *My research found 1,800+ bugs in SMT solvers, leading to 1,300+ bug fixes and 350+ soundness bug fixes.* Moreover, I validated SMT solvers to ensure that they no longer have bugs on simple formulas.

Ph.D. Research: Solidifying Modern SMT Solvers

My Ph.D. research was guided via three consecutive research questions: (1) Do SMT solvers *even have* critical bugs? (2) How to effectively test SMT solvers? (3) Have we tested enough?

Do SMT solvers *even have* critical bugs? As of early 2019, there was little reason to doubt the correctness of SMT solvers. GitHub issues of leading SMT solvers contained no soundness bugs. In the SMT solver competition, they were also rare. Moreover, research papers only found very few bugs in unstable solvers. This was indicating that besides a few known issues, SMT solvers should be trusted. *How to put this trust in SMT solvers to the test?* There are two key obstacles: (O1) fabricating complex formulas to thoroughly stress-test the SMT solvers and (O2) validating the SMT solver's result. O1 is challenging because SMT solvers have large codebases and an expressive input format while O2 is challenging because of the trade-off between formula complexity and control over its satisfiability. I invented *Semantic Fusion*, an approach to solve both challenges [12]. There is a large collection of benchmarks (400k), complex, real-world SMT formulas on which SMT solvers are well-exercised. Semantic Fusion fabricates new formulas from these benchmarks on which SMT solvers are *not* well-exercised, hence potentially returning incorrect results. To create these new formulas, we fuse formula pairs of known satisfiability in many different ways, resulting in a large space of formulas closely resembling the original benchmarks but not quite. The key here is that we fabricate the new formulas such that we also know their satisfiability by construction. We first combine the formulas, resulting in a conjunction/disjunction. Then, we slice off a piece of each formula and combine the two formulas in an interesting manner, resulting in a fused formula that we can use to stress-test SMT solvers. *Do SMT solvers* even have *critical bugs? Yes! They do, and*

[1]The most recent example is the CrowdStrike IT outage in July 2024, causing financial damage of 10+ billion USD.

quite a few. We found 76 bugs in the Z3 and CVC4, the most powerful and popular SMT solvers with this idea. Strikingly, we found soundness bugs, the most critical bug category, that can invalidate the results of verification tools. Many of these bugs were undetected for years and some were shockingly simple. *The work had an eye-opening effect on the community and won a Distinguished Paper Award at PLDI '20.*

How can we effectively test SMT solvers? Semantic Fusion is a highly targeted attack, particularly effective on nonlinear and string logic. As fusing two ancestor formulas produces a test formula with a size equal to the combined size of its ancestors, it is crucial to carefully select the ancestor formulas to avoid excessive timeouts during testing. This raises the question of whether we can design a more lightweight technique that lets us leverage all available formulas. Operators are central to the semantics of SMT formulas. If we change them, it affects the solution space of formulas, exercising new control and dataflow in the solver codebase. Hence, mutating operators yields a simple and powerful technique for testing SMT solvers [11]. We realized this idea in a testing tool called OpFuzz and conducted a large-scale testing campaign. *We found 1,200+ unique bugs in Z3 and CVC4.* Most notably, OpFuzz found soundness bugs in almost every theory of SMT. *This came as a great shock to the SMT community*. Despite its unusual effectiveness, OpFuzz also has limitations. Formulas will stay at roughly the same size, i.e., they cannot grow or shrink. In our OOPSLA '21 work, we extended OpFuzz, yielding another 237 bugs [6].

Have we tested enough? With so many bugs being found, this is a natural follow-up question. Existing SMT fuzzers cannot answer this question as they are unsystematic, randomly scattering large test formulas. If fuzzers find no bugs, we do not know whether there are truly none or whether we were just out of luck. Perhaps most importantly, fuzzers miss small-sized bugs. The small scope hypothesis [5] states that *most bugs of software trigger on small inputs*. We confirmed this hypothesis for bugs in SMT solvers. My idea to exploit this insight is to enumerate the smallest inputs from a context-free grammar. There are several unique advantages to this approach. (1) it is systematic (2) if it finds bugs, they are small in size (3) it provides bounded guarantees on formula spaces (4) we can measure the evolution of SMT solvers. We integrate this idea in a tool that we call ET [10]. It works as follows: given a grammar that describes a relevant formula space and a number of designated tests, it first compiles the grammar into algebraic datatype in Haskell. Using this representation, we then take an off-the-shelf enumerative testing library [3]. With ET, we conducted a validation campaign of Z3 and cvc5, the two most widely used and popular solvers. We found 100+ new bugs but perhaps more important than these bugs are the assurances that we gained: SMT solvers no longer have bugs on the smallest formulas anymore. We are working on making ET a part of SMT solver's GitHub CI/CD pipelines.

Did SMT solvers become better, historically? This is an essential question that ET enables us to investigate. To approach it, we tested all stable releases of Z3 and cvc5 on 8 million formulas generated by ET, tracking the number of bugs being triggered. Our results show a clear trend: While early releases of Z3 and cvc5 exhibit bugs in many theories, later releases show progressively fewer bugs and the latest Z3 has no bugs at all. Have we tested enough? Yes, under the assumption of the small scope hypothesis.

Future Direction: Developing Formal Methods Engineering

My long-term vision is to create a future in which bugs in software are an extreme rarity. To bring this future to fruition, *I will devote my career to developing Formal Methods Engineering (FME)—a new, dedicated discipline for making Formal Methods more correct, stable, performant, and usable.* With much more powerful Formal Methods, we can prevent dangerous bugs in software. FME transforms the use of Formal Methods from an ad hoc activity to a structured approach. FME is a pragmatic field connecting the dots between the mostly theoretical Formal Methods research of the Programming Language and Verification communities with the practical engineering reality. Three major challenges characterize the Formal Methods Engineering field. My mission is to make substantial progress on each of them.

Challenge C1: Making SMT solvers much more solid, stable and performant SMT solvers are the foundations for many Formal Methods tools. They are among the most complex software systems, frequently suffering from instability and performance issues. Moreover, even if SMT solvers can detect a soundness bug, i.e., a wrong result occurs, they cannot recover from it. *I will build new fully-functional SMT solvers with small codebases fixing all these issues.* The resulting solver will have fewer bugs, be simpler, and much faster. Instead of building new SMT solvers from scratch, I start with an existing SMT solver's codebase (e.g., cvc5). The first step is reducing dead, unused, and rarely-used code. This is formulated as an optimization problem trading off code size with its completeness.[2] Our first results show promising reductions of at least 50k out of 150k LoC for rarely-used life code alone while maintaining 95% of the completeness of the solver. The second step is a new approach for self-correcting software using parallelism. By fabricating equivalent versions of an input formula, we can correct unsoundness issues and instabilities of SMT solvers. Orthogonal to this, I aim at exploring to make SMT solvers significantly faster using techniques from compiler optimization and performance computing. This direction, if successful, will lead to a new generation of SMT solvers, enabling more mainstream adoption, which will have a substantial impact on academia and industry.

Challenge C2: Unifying testing and verification Testing and verification have fundamentally the same goal: to make software more reliable. Instead of thinking about testing and verification as two separate sets of techniques, we should rather think of them as one set of Formal Methods. A major research question is: *(a) How to combine the best of testing and verification?* Concretely, given a software system S with several components $C \in S$, how best to validate and analyze each component C with respect to the needed assurances and costs. I aim at developing *validation portfolios* that given a program and its specification, automatically select the appropriate technique for each component C. As an output, this yields a set of bugs and a certificate to justify the thoroughness of the process. Moreover, I aim at hybrid approaches. An instance of a hybrid approach is my OOPSLA '24 work [10] which makes testing more systematic yielding bounded guarantees. I will research validation portfolios and hybrid approaches leading to their integration into mainstream toolchains.

A second fundamental research question is: *(b) How can we build a theory for automated reasoning about the reliability of software systems?* Given a component C with a quality score and system S as a dependency graph with components as its nodes, how can we infer an overall quality score for the entire system S and identify its critical weak spots for more validation? Provided we can do this, what are the appropriate techniques to eliminate the detected weak spots? I aim to formulate a theory clarifying these question for a more principled understanding of the quality of software systems.

Challenge C3: Specification Engineering Specifications are vital for the application of Formal Methods. However, writing accurate specifications is tedious work done by humans. Hence, lowering human effort in engineering specifications is key. One such approach is realized in Zelkova, AWS's access policy tool [1], eliminating the need for manually writing specifications. Instead they let practitioners give boolean answers. Another promising avenue is to use LLMs to infer specifications for software. For the future, I aim to explore methods for engineering specifications.

References

[1] John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. "Semantic-based Automated Reasoning for AWS Access Policies using SMT". In: *FMCAD '18*. 2018, pp. 1–9.

[2] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *OSDI '08*. 2008, pp. 209–224.

[3] Jonas Duregard, Patrick Jansson, and Meng Wang. "FEAT: Functional Enumeration of Algebraic Types". In: *Haskell '12*. 2012, 61–72.

[2]To the best of my knowledge, this is a novel perspective on software debloating [7] which has so far not been focusing on rarely-used life code but instead on dead and unused code.

[4] Patrice Godefroid, Nils Klarlund, and Koushik Sen. “DART: directed automated random testing”. In: *PLDI ’05*. 2005, pp. 213–223.

[5] D. Jackson and C.A. Damon. “Elements of style: analyzing a software design feature with a counterexample detector”. In: *IEEE Transactions on Software Engineering* (1996), pp. 484–495.

[6] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. “Generative Type-Aware Mutation for Testing SMT Solvers”. In: *OOPSLA ’21*. 2021, pp. 1 –19.

[7] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. “RAZOR: A Framework for Post-deployment Software Debloating”. In: *USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1733–1750.

[8] Neha Rungta. “A Billion SMT Queries a Day (Invited Paper)”. In: *CAV ’22*. 2022, pp. 3–18.

[9] Emina Torlak and Rastislav Bodik. “A lightweight symbolic virtual machine for solver-aided host languages”. In: *PLDI ’14*. 2014, pp. 530–541.

[10] Dominik Winterer. “Grammar-based Enumeration of SMT Solvers for Correctness and Performance”. In: *OOPSLA ’24*. 2024.

[11] Dominik Winterer, Chengyu Zhang, and Zhendong Su. “On the Unusal Effectiveness of Type-Aware Operator Mutation”. In: *OOPSLA ’20*. 2020, pp. 1–25.

[12] Dominik Winterer, Chengyu Zhang, and Zhendong Su. “Validating SMT Solvers via Semantic Fusion”. In: *PLDI ’20*. 2020, pp. 718–730.