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 programming language and software engineering researcher**. 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, 5, 10, 9]. Incorrect, incomplete, or slow SMT solvers can have severe undesirable effects, leading to potential disasters. Hence, it is crucial that SMT solvers be correct and performant. My Ph.D. research body made a significant contribution to Formal Methods Engineering. Before, there was uncertainty about SMT solvers: Blind 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.*

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?; and (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 [13]. 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 returning incorrect results. To create these new formulas, we fuse formula pairs in many different ways, resulting in a large space of formulas closely resembling the original benchmarks but not quite. 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 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

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

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 an ACM Distinguished Paper Award at PLDI '20.*

How can we effectively test SMT solvers? Semantic Fusion was a very targeted attack, raising the question of whether we can do something more general. 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 [12]. The key here is also that this mutation should be *type-aware*, to create well-typed formulas to stress-test the SMT solvers. We realized this idea in a testing tool called OpFuzz and conducted a large-scale testing campaign. *We found 1,200+ 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 had limitations. Formulas will stay at roughly the same size, i.e., they cannot grow or shrink. In our work published in OOPSLA '21, we extended OpFuzz to expressions, yielding another 237 bugs [7].

Have we tested enough? With so many bug findings, this is a natural follow-up question. Existing SMT testers cannot answer this question as they are unsystematic, randomly scattering large test formulas. If testers find no bugs, we do not know whether there are truly none or whether we were just out of luck. Perhaps most importantly they miss small bugs. The small scope hypothesis states that *most bugs of software trigger on small inputs* [6]. We could confirm the hypothesis for bugs in SMT solvers. My idea was to exploit it 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; (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 [11]. It works as follows: given a grammar that describes a relevant formula space and a number of tests, it first compiles the grammar into algebraic datatypes and this approach with an off-the-shelf enumerative tester. We conducted a validation campaign of Z3 and cvc5. We found 100+ bugs in the solvers. Perhaps more important than the bugs are the assurances that we gained: SMT solvers no longer have simple bugs. *Did SMT solvers become better, historically?* We stacked up releases from Z3 and cvc5 from the last five years and conducted an experiment. We observed that their correctness increased significantly, and SMT solvers became much better over the years and are tested enough.

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.* 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 FME field.

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. The first step towards this is reducing dead, unused, and rarely-used code of an existing SMT solver's codebase (e.g., cvc5). This is formulated as an optimization problem trading off code size with its completeness. [2] Our first results show possible reductions of at least 20% (30k LoC) for rarely-used life code alone. The second

[2] To the best of my knowledge, this is a novel perspective on software debloating [8] which has so far not been focusing on rarely-used life code. but instead on dead and unused code.

step is a new approach for self-correcting software using parallelism. By fabricating equivalent versions of an input formula, we can tackle unsoundness issues and instabilities of SMT solvers. 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 are have common goal: to make software more reliable. Their unification brings several key advantages. First, it helps bridge the gap between the disconnected communities, freeing them from exclusivity claims. Second, on a practical level, it enables validation portfolios and hybrid approaches. Given a program P, its specification S_P, and a time budget T as inputs, a validation portfolio automatically selects the appropriate tool chains yielding a set of bugs B, and a certificate C to justify the thoroughness of the validation. A first step to an integrated approach is my OOPSLA '24 [11] and other works [4, 3] for making testing more systematic to find dangerous bugs earlier guided by the small cope hypothesis [6]. I will research validation portfolios and hybrid approaches to lead to their integration into mainstream practical toolchains.

Challenge C3: Specification Engineering Specifications are vital in enabling the validation techniques of C2. Hence, lowering human effort in specification engineering is key. Zelkova, AWS's access policy tool [1] used on its cloud service relieved practitioners from manually writing specifications, instead letting practitioners give boolean answers. With 1 billion SMT queries, Zelkova is one of the largest formal methods applications worldwide. Using my expertise in software engineering, I will research similar groundbreaking approaches for specification engineering.

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] Koen Claessen and John Hughes. "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs". In: ICFP '00. 2000, 268–279.

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

[5] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing". In: *PLDI '05*. 2005, pp. 213–223.

[6] 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.

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

[8] 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.

[9] Neha Rungta. "A Billion SMT Queries a Day (Invited Paper)". In: *CAV '22*. 2022, pp. 3–18.

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

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

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

[13] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT Solvers via Semantic Fusion". In: *PLDI '20*. 2020, pp. 718–730.