

From Formal Methods to Fuzzing and Back

Alastair F. Donaldson  

Department of Computing, Imperial College London, UK

Abstract

For the first twelve years of my career in computer science I mainly worked on *formal methods*. I had been drawn to the field as an undergraduate by its promise of a way to build software with strong, mathematically based correctness guarantees. But after many years of struggling to make formal methods scale, I was seduced by randomised testing—also known as *fuzzing*. Fuzzing is much less principled than formal methods, but it *just works*: it can be readily applied to very large systems, and it is extremely effective at finding real bugs that developers care about and fix. In this personal account I trace my path from formal methods to fuzzing, by way of the GPUVerify project on verifying GPU kernels and the GraphicsFuzz project on randomised testing of GPU compilers. I argue that the *oracle problem* provides an intellectual link between fuzzing and formal methods, describe my continuing, targeted uses of formal methods in the domain of GPU programming language design, and discuss work at the boundary between formal methods and fuzzing. Finally, I reflect on whether generative AI might be the missing ingredient that allows formal methods to scale, and set out my plans for large-scale formal methods via the Keystone project, through which my collaborators and I aim to find out how much of that promise can be realised in practice.

2012 ACM Subject Classification Software and its engineering → Formal software verification

Keywords and phrases Formal methods, software testing, fuzzing, model checking, SMT, proof assistants, agentic AI

Category Invited Talk

1 Introduction

I first encountered *formal methods* in the penultimate year of my Bachelor’s degree at the University of Glasgow, through a course entitled “Professional Software Development”. Taught by the late Professor Ray Welland, the course mainly focused on software engineering processes, covering topics such as requirements elicitation, project management, legacy systems and re-engineering. While interesting and practical, the material did not feel like core computer science: little about it was mathematical or precise. But at the very end of the course, Professor Welland gave two lectures about the Z notation [16], and I got a glimpse into something that seemed much more satisfying: the idea of using a mathematical formalism to precisely specify properties of software. Then, in my final year, I took “Modelling Reactive Systems”, a course on model checking taught by Professor (and now Dame) Muffy Calder. This was even better! We learned how to describe a distributed protocol in a formalism very close to source code (Promela), express desired properties of the protocol in logic (LTL), feed both to the SPIN model checker [49] and—quick as a flash—get an execution trace demonstrating that the protocol did not conform to the specification (indicating a bug in one or the other). I was hooked. Upon graduating, I embarked on a PhD in formal methods.

I worked primarily on formal methods from 2003 to 2015 as a PhD student, postdoc and early career faculty member. But although I learned a lot and had a lot of fun in the process, there was always an elephant in the room: the real-world impact of formal methods was very limited. Despite islands of success (examples including Microsoft’s Static Driver Verifier [4], the seL4 verified OS kernel [55] and the CompCert verified compiler [62]), in general it turned out to be difficult to make formal methods *scale*. The lack of formal methods impact would occasionally be called out explicitly. I recall a fellow PhD student at Glasgow bluntly telling me: “you know, nobody is *ever* going to use this stuff”. When I was a postdoc at Oxford, a

colleague’s advice was: “if you want to have impact, change field” (this colleague also worked on formal methods, and was at peace with the idea of not having impact). As an early career faculty member at Imperial, one of my PhD students was chided by another student (in jest, but almost accurately) about there being nearly as many co-authors on one of our papers as there were lines of code in the program the paper sought to formally verify!

From around 2014 I started experimenting with randomised testing, also known as *fuzzing*. I was quickly seduced by the fact that fuzzing *just works*. I felt, first-hand, the thrill of being able to find real bugs in industry-scale software projects spanning millions of lines of code—bugs that developers quickly confirmed and fixed. This was the kind of impact I had been yearning for, and over the last decade my work has mainly focused on software testing and fuzzing, including founding a company, GraphicsFuzz Ltd., that was acquired by Google.

Still, despite the *practical* satisfaction that fuzzing brings, I find formal methods more *fundamentally interesting*. Fuzzing takes a “suck it and see” approach, where the worth of an idea is judged by its ability to find bugs. But there is nothing fundamental about finding bugs: whether one finds bugs in a system under test (SUT) is as much a function of how reliable the SUT is (including the extent to which the SUT has already been hardened by previous fuzzing campaigns) as it is a function of how smart and rigorous the current fuzzing strategy is. And ultimately, fuzzing provides no guarantees; Dijkstra famously stated the obvious fact: “Program testing can be used to show the presence of bugs, but never to show their absence!” [34]. So, I never fully left formal methods: I kept working on problems where scale is not a blocker, such as using formal methods to improve the rigour of GPU programming language semantics, as well as working at the intersection of the two fields, using formal methods as a basis for randomised testing, and bringing fuzzing techniques to bear on formal methods tools.

But now, in 2026, it seems that formal methods might finally have their day, thanks to the artificial intelligence revolution. Large language models (LLMs) are shockingly good at producing the mathematical artifacts—such as invariants and proofs—that formal methods demand and that humans struggle to grapple with. Furthermore, they can produce these directly in formal notation, e.g. writing proofs for proof assistants such as Lean and Rocq. And, better still, the outputs of LLMs do not need to be trusted when used in many formal methods contexts: they can be independently and rigorously checked by (non-AI-based) proof checkers—a marriage made in heaven! While LLM capabilities have ballooned month-on-month over the last four years, my experience is that the frontier models of late 2025 (for me, Claude Opus 4.5) marked a step change in capability with respect to formal proof; enough of a step change that formal methods projects that would traditionally have required heroic efforts that few would dare undertake might soon become mainstream. My first direct taste of this came via a project where I used Lean (with which I had no prior experience) to prove a result connecting randomised generation of structured data with the languages accepted by Turing machines [79]. Despite my lack of prior experience, AI enabled me to make very rapid progress on what turned out to be a sizeable proof [37]. Inspired by this, I have recently embarked on *Keystone* [40], a collaboration between Imperial College London and EPFL, funded by the UK’s Advanced Research and Invention Agency (ARIA), on using AI and proof assistants to construct formally verified software, with a particular focus on creating formally verified infrastructure for serving LLMs.

This article is a personal account and reflection on my experience working in and moving between the formal methods and software testing fields. The examples I give, and the formal methods and fuzzing techniques I discuss, are thus heavily influenced by my own experience. Throughout, I consider formal methods applied to *software*, though some of the points made

apply equally well to hardware verification, and some of the remarks about scaling proofs may be relevant to the use of proof assistants in mathematics.

In the rest of this article I elaborate on:

- the scalability challenges that have hindered the practical impact of formal methods (Section 2);
- my experience wrestling with these challenges in the GPUVerify project on formal verification for GPU kernels (Section 3);
- my move from formal methods to fuzzing, which led to the GraphicsFuzz project, with some thoughts on why the *oracle problem* provides an intellectual link between the two fields (Section 4);
- my continued work on “small scale” formal methods, in the context of GPU programming language semantics (Section 5);
- my work at the intersection of formal methods and fuzzing (Section 6);
- the opportunities that the AI revolution brings for formal methods, and the Keystone project through which my collaborators and I plan to explore them (Section 7).

If your background brings to mind different examples or experiences that either support or run counter to the points I make in this article, I would love to hear from you.

2 Challenges of Scale

Why have formal methods struggled to have practical impact? In my experience the answer almost always comes down to *scale*, but “scale” covers several distinct problems. Broadly, I identify four main challenges of scale: the scale of proofs, the scale at which automated verification algorithms can operate, the scale of the tooling needed to handle real-world software, and the scale of teams that work on formal methods projects.

I describe these problems of scale as they stood before the recent advances in AI. Some of them look rather different now, and I return to them in Section 7.

Proof scale

Interactive theorem provers such as Isabelle [66], Lean [30] and Rocq [73] (formerly Coq) sit at one end of the formal methods spectrum—the “highly manual” end. Their big advantage is that, if you are willing to put the manual work in, you can express the argument needed to prove a program correct in as intricate a fashion as is required. The only limits are (a) how elaborate an argument must be communicated to the system for a proof to go through, and (b) how much time the user is able to invest in the process of discovering and communicating a detailed argument.

The problem is that the argument that must be communicated to the system can be *very* elaborate. For example, the authors of the seL4 formally verified kernel reported that while the kernel comprises 8,700 lines of C, its refinement proofs comprise around 165,000 lines of Isabelle, and the complete development, including frameworks, libraries and generated proofs, comes to some 200,000 lines [55]. CompCert tells a similar story: of the development’s 42,000 lines of Coq, around 6,000 are the compiler itself and around 4,000 are the semantics of the source, intermediate and target languages, while the remaining 32,000 are the correctness proof, so that there are more than five lines of proof for every line of compiler code [62]. And an empirical study of the seL4 proofs, together with the largest software verification developments in the Archive of Formal Proofs, suggests that the size of a proof grows quadratically with the size of the statement being proved [64].

Another problem is that when working with a proof assistant one never knows how close one is to the end, and there are always twists and turns. For example, a leaf lemma may turn out not to be provable without some extra piece of machinery that then needs to be threaded through every theorem that depends (directly or indirectly) on the lemma, or a subtle flaw in a definition or theorem statement can turn out to make a result untrue as stated, requiring major proof surgery. A related problem is that the effort required to update a proof when changes are made to the associated source code can be substantial: the seL4 team reported that adding *reply capabilities* to the kernel required substantial updates to existing proofs, with re-verification taking around a person-year [55]. DeMillo, Lipton and Perlis anticipated this kind of problem decades earlier in their 1979 critique of formal verification [33, p. 278]:

“The discontinuous nature of programming sounds the death knell for verification. A sufficiently fanatical researcher might be willing to devote two or three years to verifying a significant piece of software if he could be assured that the software would remain stable. But real-life programs need to be maintained and modified. There is no reason to believe that verifying a modified program is any easier than verifying the original the first time around.”

Algorithmic scale

At a different point on the formal methods spectrum are verification techniques that aim to be fully or highly automatic, such as model checking. Here the limits of scale are algorithmic. For software with unbounded state (integers, dynamically allocated memory, recursion), most interesting verification problems are undecidable, by reduction from the halting problem. Even when restricted to finite-state systems, where the problem is decidable, the state space of a system typically grows exponentially in the number of its components, so that exhaustive exploration quickly becomes infeasible: this is the infamous *state explosion problem* [26].

Key categories of techniques that respond to the problem of algorithmic scale include *abstraction* [24, 5, 48], which shrinks the state space of a system by discarding detail, *pruning techniques* (e.g. partial order reduction [46] and symmetry reduction [25, 42]), which reduce the algorithmic work required by avoiding the need to explore parts of the state space that are provably equivalent to others, and *symbolic methods*, which represent sets of states implicitly (typically using BDDs [18], SAT formulas [14] or SMT formulas [3, 28]).

While these techniques have dramatically improved the scalability of algorithmic verification techniques, they all come with limitations and trade-offs. This is unavoidable since the underlying problems are intractable or undecidable in general. Abstraction leads to over-approximate verification techniques that are sound but may report false alarms. Abstraction refinement, which uses spurious counterexamples to recover just enough detail to rule them out [23], is great when it works. However, the refinement loop can diverge, generating ever more refinements without converging on a proof [53]. State-space pruning techniques can achieve dramatic reductions on systems with the right structure, but they do not fundamentally solve the problem of scale. While symmetry reduction can give a factorial saving when applied to a distributed protocol over a fully connected topology, the saving shrinks as the degree of symmetry falls. A ring topology, for instance, yields only a linear reduction, and for systems with no symmetry at all, symmetry reduction offers no benefit. Partial-order reduction collapses commuting interleavings in a concurrent system, but the reduced space is still typically exponential in the number of concurrent components. And while symbolic methods allow verification to scale by orders of magnitude compared with explicit-state representations, the underlying techniques can be brittle (for instance, the size

of a BDD can vary dramatically depending on how variables are ordered [17]), making them highly sensitive to how things are modelled.

The brittle nature of symbolic techniques is a particular problem in so-called *auto-active* verification [61], which sits between fully automatic model checking and interactive theorem proving. In an auto-active verifier such as Dafny [60], Why3 [43], Viper [65] or F* [72], the user annotates the program with specifications (pre- and post-conditions, loop invariants, assertions and lemmas) and the tool generates verification conditions from the annotated program and discharges them automatically by calling an SMT solver. While leveraging SMT solvers can improve automation (compared with interactive theorem proving), the brittle nature of solvers leads to fragility. Verification conditions often involve quantifiers, and SMT solvers handle quantified formulas by heuristic instantiation (e.g. via *E-matching* against syntactic trigger patterns [31]), which is incomplete and the behaviour of which depends on the precise shape of the query. Because the user has only indirect control over that shape, through annotations whose effect on the solver is opaque, a failed proof attempt offers little guidance as to what to change. As a result, SMT-based program verifiers suffer from *instability*, whereby a semantically irrelevant change to a query, such as renaming identifiers or reordering assertions, can cause a previously successful proof to fail or time out [81]. Amazon Web Services, which has used Dafny heavily and contributed to its development, has identified verification instability as a factor that hindered the adoption of Dafny, reporting that production projects at AWS see previously-verified code stop verifying after small, seemingly unrelated edits [71], and has published guidance on structuring Dafny code to reduce the risk [74]; engineers on AWS’s formally verified authorization engine report that brittleness grew with the codebase to the point of delaying deployments [19].

Tool scale

Getting practitioners interested in a technique requires giving them something they can try out on their own code, in their own domain. Yet formal methods are often defined and evaluated in a simplified setting such as a core calculus or toy language. Making them work for real-world languages—with gnarly features such as preprocessor macros, templates, foreign-function interfaces, inline assembly and a rich infrastructure of libraries managed by build systems—is very difficult from a practical point of view. DeMillo, Lipton and Perlis make this point vividly, characterising real production systems through the programmer’s jargon of “[p]atches, ad hoc constructions, band-aids and tourniquets, bells and whistles, glue, spit and polish” [33, p. 277]. Scaling formal methods to real languages requires substantial tooling work that formal methods researchers may not want to do or know how to do. Furthermore, undertaking such tooling work may be high risk from a career advancement point of view given that efforts towards building tools are not valued as highly in academic computer science as I believe they should be.

Various industry reports on applying static analysis techniques to large codebases (varying in the extent to which the associated methods would be deemed “formal”) bear out the need for attention to tooling issues. The team behind the Coverity static analyser found that parsing the many dialects of C accepted by their customers’ compilers, and fitting into customers’ build processes, were major obstacles that had to be overcome before any analysis could take place [9, pp. 69–70]; reports from Google and Facebook similarly emphasise the importance of integration into developer workflows [70, 35].

The availability of open-source program analysis frameworks has helped a great deal in this regard, e.g. the Clang/LLVM project [57] provides industrial-strength front ends for C-family languages; the CProver framework that underpins CBMC [27], and configurable program

analysis [12] as realised in the CPAchecker framework [13], give verification researchers shared infrastructure to build on; and intermediate verification languages such as Boogie [7] play an analogous role in the development of verification tools as compiler intermediate representations play in the development of compilers, avoiding the need to build a verification tool entirely from scratch. GPUVerify, which I discuss in Section 3, relied on both Clang/LLVM and Boogie. But while such frameworks lower the barrier to building formal methods tools, their main benefit is to take away the pain of parsing and type-checking a real language. Developers of a tool built on such a framework still need to decide how to handle the weird and wonderful features that the front end faithfully passes on.

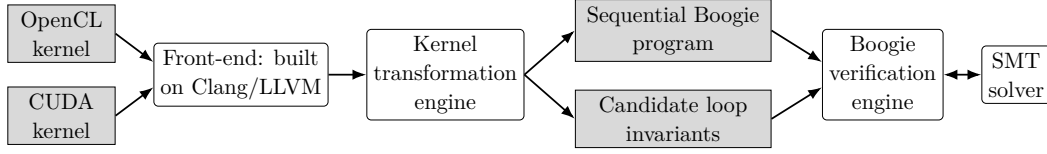
Team scale

To bring a formal method to bear on a project, one first needs to build a team whose members have mastery of the formal method, and gaining mastery of a formal method can be demanding. One of Hall’s “Seven Myths of Formal Methods” is “Formal methods require highly trained mathematicians” [47, Myth 4]. However, Hall’s argument that this is a myth mainly relates to writing specifications: “The fact is that the mathematics for specification is easy. Once it is recognized that the practice of formal methods is most concerned with writing specifications, the mathematical difficulties become much less significant. You can develop specifications themselves with very straightforward mathematics that any practicing engineer should know” [47, p. 16]. My experience is both that writing specifications is more challenging than Hall states, and that using formal methods tools tends to involve a lot more than specification writing: in particular, many formal methods involve writing proofs, which can be a major challenge. The steep learning curve associated with interactive theorem provers is well known [54, 69].

Beyond the challenge of building a team, there is the problem of scaling formal methods work *across* a team. Agile software development revolves around small, independent increments that can be delivered quickly. Agile *formal methods* work is harder to achieve, because a small change to a codebase may necessitate a substantial change to a proof, or may derail an automated verification technique by upsetting the conditions for which its heuristics have been carefully tuned. An experience report by members of the seL4 project acknowledges this challenge: “Ideally [distribution of proof sub-tasks to team members] is achieved by a compositional calculus that allows clean, small interfaces between sub-tasks, which are defined once and remain stable. In reality, this is rarely the case. Full compositionality often comes with a high price in other aspects of complexity, while interfaces are rarely small or stable.” [15, p. 40] Nevertheless, with a suitably designed proof structure, the seL4 team were able to divide proof work among team members: “the FM [formal methods] team managed to design refinement calculi for each of the major proof steps that allowed the work to be divided up among members of the team” [2, p. 1008].

3 GPUVerify: My Formal Methods Push

My most sustained attempt to make formal verification work in practice was GPUVerify, a static verification tool for GPU kernels written in OpenCL and CUDA, developed with a team of PhD students, postdocs and collaborators at Imperial College London and Microsoft Research [11, 6]. A GPU kernel is a function executed simultaneously by thousands of threads, and a recurring source of bugs in such kernels is the *data race*, where two threads access a common memory location, at least one of the accesses is a write, and the accesses are not ordered by inter-thread synchronisation. Our hope with GPUVerify was that by



■ **Figure 1** The components comprising the GPUVerify tool for verifying data race-freedom of GPU kernels, adapted from [36]

targeting a niche-but-important problem—proving race-freedom for GPU kernels, rather than proving arbitrary functional properties for arbitrary software—we could build a highly automatic formal methods tool [6]. In terms of the challenges of Section 2, we addressed tool scale by building on existing infrastructure, as I describe below; the tightly scoped problem meant that a relatively small team sufficed; and our focus on automation was a deliberate attempt to sidestep proof scale. That left algorithmic scale, and in particular the fragility of SMT-based verification, as the challenge we could not avoid.

GPUVerify: a brief overview

The main insight behind GPUVerify is that if execution barriers are the only means of synchronisation (a property of many GPU kernels, and one that was almost ubiquitous when we started the project in 2011), a race-free kernel behaves deterministically. Thus, as long as races are detected, it suffices to analyse a single, canonical thread schedule. This allows a massively parallel kernel to be rewritten as a sequential program. On its own this does not scale, because the sequential program would encode the work of thousands of threads and would thus be infeasibly large. GPUVerify therefore applies a *two-thread abstraction*: since a data race is a conflict between exactly two threads, it suffices to check an arbitrary pair of distinct threads, with the effects of all other threads over-approximated by treating shared memory abstractly. Combined with predicated execution to handle thread-dependent control flow, and instrumentation to log and check memory accesses, GPUVerify turns a kernel into a sequential program, expressed in the Boogie intermediate verification language [7]. Correctness of this sequential program implies that the original kernel is free from data races (and from barrier divergence, a related defect). Verifying the sequential program requires loop invariants, which GPUVerify infers using the Houdini algorithm [45]: a large set of *candidate* invariants is guessed, using domain-specific rules that capture memory access idioms we observed in real kernels, and Houdini repeatedly discards candidates that cannot be proven until the survivors form an inductive invariant. The components that comprise GPUVerify are illustrated by Figure 1.

Standing on the shoulders of giants

GPUVerify would not have been possible without heavy re-use of existing infrastructure. We benefitted from OpenCL and CUDA support in the Clang/LLVM framework, which handled the parsing and lowering of kernels to LLVM IR, sparing us from dealing with the full syntactic complexity of the source languages (C++ templates in CUDA code being a case in point). Our own translator then mapped LLVM IR into Boogie, and Boogie, in turn, handled verification condition generation and invariant inference, delegating to the Z3 [32] and CVC4 [8] SMT solvers. This allowed us to apply GPUVerify directly to developer-written kernels, and we evaluated the tool on all the publicly available GPU benchmark suites we could get our hands on at the time, discovering several real data races along the way. An

industrial collaborator, Rightware (a Finnish company specialising in benchmarking), used GPUVerify to verify race-freedom across its Basemark CL benchmark suite, finding a defect in the process [6].

GPUVerify: limits of scale

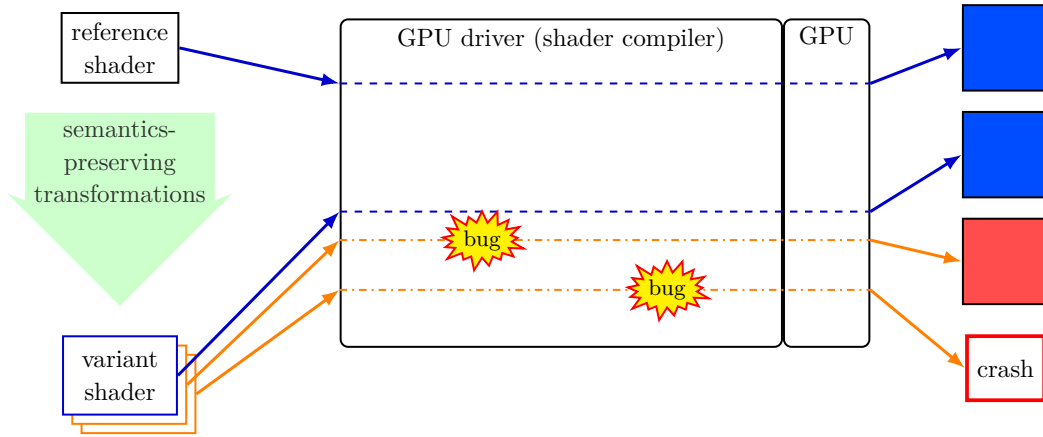
Despite its relative success on a number of benchmark suites, GPUVerify also ran into some of the problems of scale described in Section 2, principally the problems associated with auto-active verifiers discussed at the end of “Algorithmic scale”.

First, invariant inference was unpredictable. Verifying kernels with loops typically required intricate invariants relating the memory locations that a thread accesses to its thread and block ids: invariants we could not reasonably expect GPU programmers to write. Our domain-specific candidate-generation rules let Houdini find such invariants for many kernels [10]. This often worked out of the box, but when it did not the user was left high and dry, facing a possible data race report that was indistinguishable at first glance from a real bug. Speculating more candidates also made verification slower and its running time harder to predict. Second, we were at the mercy of the quirks of SMT solvers. Much of our engineering effort went into solver-friendly encodings, such as quantifier-free representations of the sets of memory accesses used for race checking. We found ourselves talking to solver developers to find just the right solver options to use, and there was no guarantee that our encodings would remain efficient as the solvers evolved. Performance was excellent for some kernels but miserable for others, switching between Z3 and CVC4 could change the picture dramatically, and even an optimisation that simplified the program to be verified could make some kernels much slower to verify, often for no apparent reason [6]. Third, the two-thread abstraction that made GPUVerify scale also imposed a hard limit on what it could prove. Because the effects of all other threads on shared memory are over-approximated, GPUVerify reports false alarms for *data-dependent* kernels, where race-freedom depends on values that other threads have written to shared memory. We introduced barrier invariants, a richer shared state abstraction, to handle some of these cases [21], but they needed to be written by hand.

In practice, data-dependent kernels were a minority, so the two-thread abstraction was a good trade-off due to its scalability benefits. My overall feeling about GPUVerify is that it scaled well enough, in the sense of handling the majority of kernels from real benchmark suites, but that it owed this to a domain-specific abstraction rather than to a more general advance in verification technology. What ultimately limited the tool was not the size of the kernels but the unpredictability of the automated reasoning on which it relied, for invariant generation and SMT solving alike.

4 GraphicsFuzz: From Formal Methods to Fuzzing

When I gave talks about GPUVerify, a question I was often asked was: “but can you trust the GPU compiler?” Proving that a kernel is free from data races is of limited comfort if the compiler that translates it into machine code introduces bugs of its own. Because GPU kernels are compiled at runtime by the driver of whatever GPU happens to be present, an application developer cannot easily anticipate or work around such bugs. My friend Anton Lokhmotov, then at ARM and a collaborator on the CARP EU project, pointed me to Csmith [78], a tool that generates random C programs and cross-checks C compilers against one another, and which had found hundreds of bugs in GCC and LLVM. “Maybe you should look at doing something similar for OpenCL,” he said. I recruited a talented undergraduate, Christopher Lidbury, who built CLsmith, an OpenCL adaptation of Csmith [63]. It proved



■ **Figure 2** An overview of the GraphicsFuzz approach to testing GPU shader compilers, adapted from [39]

tremendously good at finding bugs: more than 50 in OpenCL compilers, most of them in commercial implementations. Coming from several years working on GPUVerify, this was a revelation: there were no false alarms to explain away and no scalability problems to wrestle with; the tool readily found real bugs that developers cared about and fixed.

The oracle problem for graphics compilers

I next turned to compilers for graphics shading languages such as GLSL, the OpenGL shading language, initially because they posed an intriguing puzzle. Shading languages are deliberately underspecified with respect to floating-point precision, so as to cater for a diverse range of current and future GPUs. As a result there is no single correct image that a shader should render [38]. As someone from a formal methods background I found this fascinating: you clearly cannot formally verify a compiler against such a loose specification, but can you reliably test it? This led me to Weyuker’s classic work on testing “non-testable” programs, whose correct output is not known in advance [76], and to *metamorphic testing* [20], where instead of checking individual outputs, one checks that related inputs yield appropriately related outputs. I was particularly inspired by work on testing compilers via *equivalence modulo inputs* (EMI) [58], which mutates the parts of a program that are not executed on a given input, yielding variants that should behave identically on that input. We had already adapted the EMI idea to OpenCL by injecting code that is dead by construction [63], and for GLSL we generalised it to consider a range of semantics-preserving transformations [38]. Our approach, implemented in a tool originally called GLFuzz and later renamed GraphicsFuzz, would create a set of *variant* shaders from an original *reference* shader. Each variant was produced through the application of a random sequence of semantics-preserving transformations. By construction, a shader and its transformed variants should render visually identical images, with a significantly different image being indicative of a compiler bug. Crucially, because the reference and all variants would be executed on the same device, the compiler under test was only ever cross-checked against itself, so that the problem of differing floating-point behaviour across GPUs became irrelevant. We found that a straightforward image comparison metric was sufficient to tolerate small rounding differences arising due to optimisations, while still flagging up major differences arising from miscompilations. Figure 2 illustrates the GraphicsFuzz approach.

From research to production

In practice, GraphicsFuzz found bugs affecting every GPU and driver configuration we tested, spanning all the major GPU designers [38]. I had never had so much fun finding bugs. Some of them were serious security vulnerabilities: a malicious WebGL page viewed in Chrome on a Samsung Galaxy S6 could capture the contents of other browser tabs; another could trigger a “blue screen of death” under Windows 10, simply by being visited; and further issues led to Apple fixing a memory corruption bug in iOS (CVE-2017-2424) and NVIDIA fixing a machine freeze in its Linux drivers (CVE-2017-6259) [38].

GraphicsFuzz formed the basis of a spin-out company, GraphicsFuzz Ltd., which Google acquired in 2018. At Google, we put the technique into production to improve the Khronos Vulkan Conformance Test Suite (CTS), which Vulkan drivers must pass in order to be deemed conformant, and which is incorporated into the Android Compatibility Test Suite [39]. The idea was simple: whenever GraphicsFuzz finds a bug in a conformant driver, this reveals a gap in the CTS, so we built tooling to automatically find such bugs, reduce the bug-inducing shaders to small, readable examples, and package them as CTS tests, ensuring that future conformant drivers cannot exhibit those bugs. We also used differential coverage analysis to synthesise tests covering driver code that the CTS did not reach, leading to hundreds of new CTS tests.

The appeal of the oracle problem to a formal methods mindset

Although fuzzing is where my work has had the most practical impact, my interest in it is, at heart, that of a formal methods researcher. What makes compiler fuzzing intellectually satisfying to me is the *oracle problem*: it is trivial to generate test inputs at random in a naive fashion, but generating well-formed test programs for which a compiler should produce meaningful results is challenging, and then knowing whether the output the compiler has generated is correct is hard. Attacking the oracle problem calls for the kind of semantic reasoning that formal methods are built on: metamorphic relations are, in effect, partial specifications, the argument that a particular class of program transformations is semantics-preserving can be seen as a small theorem about the programming language of interest, and creating a generator of programs that are free from undefined behaviour requires attention to the details of programming language semantics.

I find the link between formal methods and fuzzing provided by the oracle problem deeply appealing, and some of my favourite randomised testing papers of recent years feature clever oracles. For example, *ternary logic partitioning* [68] introduces an oracle for testing database engines by exploiting the fact that, in SQL’s three-valued logic, every row satisfies exactly one of a predicate p , its negation $\neg p$, or p IS NULL. Thus the rows returned by a query such as `SELECT * FROM t` should be *exactly* the union of those returned by the three partitioned queries that add constraints such as `WHERE c > 5`, `WHERE NOT (c > 5)` and `WHERE (c > 5) IS NULL`, respectively. Any discrepancy between the union of the returned sets and the result returned by the unconstrained query reveals a bug. Importantly, one does not need to know what the correct set of returned results is to know that there is a bug—this is the beauty of metamorphic testing.

Another example of a metamorphic oracle is *semantic fusion* [77], which tests SMT solvers by fusing two formulas whose satisfiability is already known and which are known to agree (both satisfiable or both unsatisfiable). Using an illustrative example from the semantic fusion paper, consider the satisfiable formulas $\varphi_1 = x > 0 \wedge x > 1$ and $\varphi_2 = y < 0 \wedge y < 1$, over disjoint variables. Their conjunction is obviously satisfiable. To fuse them, one introduces

a fresh variable z and a fusion constraint $z = x + y$ and replaces some occurrences of x by $z - y$ and of y by $z - x$, yielding e.g.:

$$\varphi_{\text{fused}} = (x > 0 \wedge z - y > 1) \wedge (z - x < 0 \wedge y < 1).$$

This formula differs significantly from both seed formulas, and it is satisfiable by construction: any pair of models of φ_1 and φ_2 extends to a model of φ_{fused} by setting $z = x + y$. Thus if a solver reports φ_{fused} to be unsatisfiable, the solver exhibits a soundness bug. For unsatisfiable seeds the construction is dual: the formulas are disjoined and the constraint $z = x + y$ is added, so that the fused formula remains unsatisfiable. This can be seen as an example of metamorphic testing because the expected satisfiability of the constructed formula is inherited from that of the seeds.

Metamorphic oracles such as EMI, the semantics-preserving transformations employed by GraphicsFuzz, and ideas like ternary logic partitioning and semantic fusion have the feel of formal methods specifications because they exploit semantic insights. These insights allow fuzzing to go beyond crash bugs, enabling deep functional correctness bugs to be found.

Problems of scale?

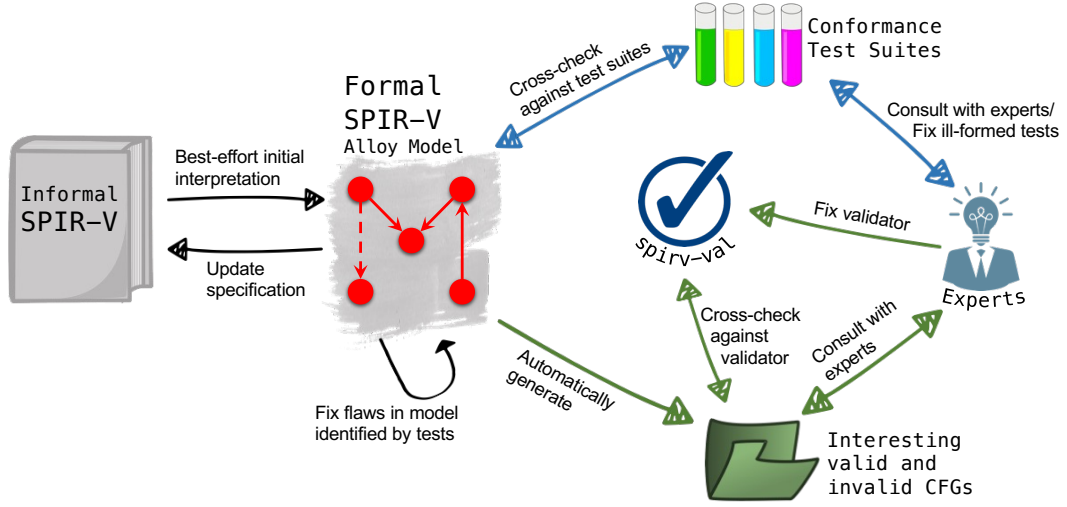
It would be an exaggeration to say that fuzzing is entirely free from problems of scale. A fuzzer's success depends in large part on throughput, so fuzzers must be engineered for efficiency. And the tool-scale problems of Section 2 have an analogue when the system under test takes a rich input format such as a programming language: the generator must then produce valid programs. But by and large fuzzing is not burdened by problems of scale, and importantly, it can be readily applied to very large software projects.

5 Targeted Formal Methods Where Scalability Is Not Required

My change of focus from formal methods to fuzzing did not mean leaving formal methods behind entirely. Instead, doubling down on the GPUVerify ethos of using formal methods to attack niche-but-important problems, I gravitated towards problems where scale is not the blocker. I now briefly describe two recent projects on the semantics of GPU programming languages where a small formal model, developed in close collaboration with the engineers responsible for the language design, can have meaningful impact.

Taking back control in SPIR-V

A project I am particularly proud of concerned SPIR-V, the industry-standard intermediate representation for GPU programming used by Vulkan and OpenCL [56]. SPIR-V is an unstructured, control-flow-graph-based language, but it is equipped with annotations that allow high-level structured control flow, such as loops and selection constructs, to be expressed. The rules governing these annotations had long been a source of confusion and frustration: they were complex, ambiguous, and in places at odds with what their designers had intended. In some cases their complexity had arisen from unsuccessful attempts to align them better with designer intentions. My group at Imperial, working with SPIR-V experts at Google, built a small formal model of SPIR-V control flow in the Alloy modelling language [52], capable of determining whether a given control-flow graph is valid, and of generating examples of valid and invalid control-flow graphs. In the absence of a reliable specification, we started by writing a best-effort formal model, faithfully modelling uncontroversial aspects of the SPIR-V specification, and including tentative fixes for parts of the specification we knew to



■ **Figure 3** The virtuous cycle of validating our Alloy model of SPIR-V against various sources of truth [56]

be problematic. We then iterated on our model by cross-checking it against several sources of truth: the specification text, various SPIR-V test suites, the `spirv-val` validation tool, and the expert judgement of our industrial collaborators. Whenever the model disagreed with one of these sources, the discrepancy revealed a problem in at least one of the model, the tests, the validator, or the specification itself, which we fixed. This process led us to a new concept, *structural dominance*, through which the control flow rules of SPIR-V can be stated accurately and far more simply. Our reformulated rules have been adopted in the SPIR-V specification, and along the way we fixed problems in the conformance tests and in `spirv-val` [56]. Figure 3 illustrates this virtuous cycle.

Uniformity analysis in WGSL

More recently, we have taken a similar approach to the WebGPU Shading Language (WGSL), whose specification mandates a *uniformity analysis* to reject programs that might suffer from barrier divergence. We gave a formal semantics that defines barrier divergence for a core of WGSL, reformulated the analysis as concise formal rules, used the semantics to expose soundness and precision problems in the analysis as specified, leading to changes to the specification that have all been accepted, and mechanised the analysis in Alloy [59].

Neither of these projects required heroic proof effort: in both cases the models are relatively small, and the Alloy Analyzer explores them automatically up to a given bound. And neither of the projects offered particularly good scalability—due to its use of a SAT solver behind the scenes, we found that the Alloy Analyzer would quickly struggle to produce examples for large graphs with intricate constraints, and (as is so often the case with symbolic methods) performance could be rather unpredictable. This turned out not to matter much. Alloy’s design rests on the *small scope hypothesis* [51]: most defects in a model can be exhibited by a small instance, so exhaustive exploration of all instances up to a modest size is a surprisingly effective way of finding them. The same reasoning applies to defects in whatever the model describes, and our experience with SPIR-V and WGSL bore this out: the soundness and precision problems we found in the SPIR-V and WGSL specifications all manifested on modest control-flow graphs and programs, and once the specification issues

had been fixed we were able to push the analysis to graphs and programs well beyond the size at which the last problem had appeared, without finding any more. While this kind of bounded result does not rule out the possibility of larger problematic examples, it gives a more systematic version of the kind of confidence that testing provides.

6 Formal Methods Meets Fuzzing

Having a foot in both the formal methods and fuzzing camps, I am naturally interested in work that sits at the boundary between them, where each can help the other. Formal models can be a source of randomised test inputs for fuzzing, with strong oracles coming for free from the model; and formal methods tools are themselves large, complex pieces of software in which fuzzing can find bugs.

Formal methods for fuzzing

A bonus of building the Alloy models described in Section 5 is that the Alloy Analyzer can generate example instances of them, via a SAT solver. In the SPIR-V work, this gave us a ready supply of valid control-flow graphs that are arbitrary in shape, unconstrained by any particular programming style. These were ideal seeds for fuzzing. We exploited this via a simple-but-effective technique that we call *control-flow graph fleshing* [56]. Given a control-flow graph skeleton generated by Alloy, we choose a random path through it, and flesh the skeleton out into an executable program that is forced to follow this path and that records the path it actually takes. If the recorded path differs from the expected one, the compiler is at fault. This gives every test a strong oracle by construction, and fuzzing via fleshing found numerous previously unknown bugs in SPIR-V compilers and translators from several vendors. Similarly, in the WGS� work we used Alloy’s ability to generate examples to stress-test the uniformity analysis implemented in the Chromium web browser. Here the oracle comes from differential testing between the model and Chromium: if they disagree about whether a program passes the analysis, one of them has a bug [59].

Fuzzing for formal methods

Formal methods tools are software, and there is a strong case that formal methods tools should be considered critical software because a defect in a formal methods tool has the potential to undermine the guarantees that the tool claims to provide.¹ It therefore makes sense to use fuzzing to find bugs in formal methods tools!

An early example of this is the use of Csmith, designed to test C compilers, to test the Frama-C static analysis platform [29]. This led to fifty bugs being found and fixed in Frama-C, and in a pleasing twist, Frama-C in turn exposed bugs in Csmith, in the form of generated programs that exhibited undefined behaviour despite Csmith being designed to avoid it. In a similar vein, various researchers (including my collaborators and me) have applied fuzzing to the verifier and compiler of Dafny, a verification-aware programming language [41, 50, 80]. Bugs in a verifying compiler’s code generator are particularly insidious: if the compiler silently generates wrong code, the guarantees established by verification at the source level are lost, and the bug evades both code review and formal verification.

¹ Although I am excited about working on testing formal methods tools such as analysers and verifiers, I am not fully sold on the dangers that bugs in these tools *really* pose; this was the subject of an interesting debate at a recent Dagstuhl seminar [22, §4.1].

In the hardware domain, we have used fuzzing to find bugs in commercial C-to-RTL formal equivalence checkers, tools that are trusted to the point of being used to authorise hardware design sign-off [67]. Our tool, Equifuzz, generates rich SystemC designs that are known, by construction, to be equivalent to trivial hardware designs. An equivalence checker that disagrees has a *completeness* bug: it fails to prove equivalence of designs that really are equivalent. The counterexample associated with a failed proof often contains values indicating the result the equivalence checker (wrongly) believes the SystemC design should compute, and it is often possible to exploit this by asking the equivalence checker to (wrongly) prove that the SystemC design indeed produces the given value, thus flipping the completeness bug into a (more serious) *soundness* bug. Equifuzz found several soundness and completeness bugs in major commercial equivalence checkers, which have since been confirmed and fixed, improving trust in these tools.

7 Artificial Intelligence: Scalable Formal Methods At Last?

As I noted in the introduction, the frontier models of late 2025 crossed a threshold with respect to formal proof. They have also become much more capable at constructing software, especially when used agentially. I now return to the four problems of scale from Section 2 and consider, for each, how much LLMs might change the picture. It is important to note that these potential benefits of scale are only available to practitioners with access to state-of-the-art models, and that scale comes at high compute costs with associated environmental impact. Making AI both accessible and sustainable is a challenge for the AI revolution as a whole, and one that those of us who stand to benefit from it have a responsibility to help address.

Proof scale

Recall that the central burden of interactive theorem proving is in constructing laborious, detailed proofs, and that the central burden of auto-active verification is in finding supporting annotations, such as loop invariants. Both share the property that it is hard for a human to come up with the required artifact, but easy for a machine to check whether a *candidate* artifact fits the bill. This plays to the strengths of LLMs: their stochastic nature makes them well suited to proposing plausible proofs and invariants, but there is no need to trust what they propose, because a proof assistant kernel will reject an incorrect proof, and a verification engine will reject an invariant that does not hold or is not inductive. An LLM can therefore be used as a tireless, fallible generator of candidates, for consideration by a rigorous, non-AI-based checker. Hallucinations and other inaccuracies can, at worst, waste time and resources. Critically, as long as the checker does its job, they cannot lead to an incorrect proof being accepted.

As a result of these advances, within the formal methods and programming languages communities it is becoming increasingly routine to formalise definitions, semantics and results in Lean or Rocq. Just a few years ago, such formalisations would only usually be attempted by particular research groups with the requisite years of expertise.

One thing that a proof checker *cannot* catch is a mistake in the *specification*—the theorem statement and the definitions on which the statement depends. Special care must therefore be taken before letting AI loose on both specifications and proofs, and carefully reviewing and testing specifications (including via fuzzing) is more important than ever in an age of AI-generated proofs.

Algorithmic scale

Here I am less sure that AI changes the picture fundamentally. The limits on automatic verification are limits of intractability and undecidability, and a clever language model cannot make the state space of a system smaller!² That said, AI has the potential to help around the edges of algorithmic scale. For example, AI could absorb some of the brittleness associated with SMT-based verification described in Section 2: an agent could watch for cases where previously successful verification breaks after a tool or configuration update, and then iterate the process of (for example) updating solver options, modifying encodings, adjusting invariants and fine-tuning E-matching triggers, until verification succeeds again. This is exactly the kind of tedious, well-specified task that AI is good at. Furthermore, AI may serve to accelerate research on verification algorithms themselves, because LLMs make it cheap to prototype an idea and, increasingly, to formally verify that the idea is sound.

Tool scale

The opportunity afforded by AI here is simple: LLMs make it much cheaper to create software, or at least initial prototypes of software. Whether they will transform the creation of high-quality, long-lived software remains to be seen. But proof-of-concept tooling of the kind needed to demonstrate a formal method in action—extending a front end to handle the esoteric language features that one particular codebase relies on, bridging a solver-based method to a different solver, wiring a verifier into a build system—is precisely the sort of engineering work that agentic coding tools are now very good at. In Section 2 I noted that formal methods researchers are often unwilling or unable to do the tooling work needed to apply a technique to real code; in the age of LLMs that excuse now seems rather weak.

Team scale

There is potential for AI to help on both sides of the team-scale problem: accelerating the acquisition of sufficient expertise in a formal method, and making it easier to distribute formal methods work across a team.

On the first point, I have personally found LLMs to be enormously useful in learning Lean, quickly allowing me to get to the point where I can do meaningful work in it. It also seems clear that “on-tap” access to AI assistance lowers the level of expertise one needs to use a formal method productively.³

Second, there is growing evidence that LLMs are good at proof repair [44, 75], which is the core of proof maintenance. If proofs can be kept in step with code largely automatically, the agile style of development that I argued in Section 2 is at odds with formal methods may become practical for formal methods too. DeMillo, Lipton and Perlis’s argument that verifying a modified program is as hard as verifying the original would then lose much of its force.

² One might object that proof search is intractable too, and yet AI has shown great promise. My sense is that writing a proof is more like programming than like solving an NP-hard instance, and that this is why LLMs are good at it.

³ As an aside, I do *not* expect AI to do much for the acquisition of deep expertise; digesting a complex formalism properly takes time, and learning often comes from being lost for a while. My hunch is that the ability to reach for an LLM every time one gets stuck will accelerate short-term progress to the detriment of long-term understanding. The likely outcome is fewer people with deep expertise and many more people who know enough to be effective.

The Keystone project

With collaborators at Imperial College London and EPFL, I recently embarked on Keystone [40], a project which seeks to investigate the impact that AI can have on formal methods at scale. Keystone is funded through a 2026 call from ARIA’s Safeguarded AI programme on using AI-enabled formal methods for cybersecurity [1]. The premise of the call aligns closely with the argument of this paper: it observes that formal methods have historically been too slow and expertise-intensive to apply at scale, that AI is strong at exactly the kind of search-intensive work that traditionally made formal methods hard to apply, and that “speed from AI does not require trust in AI” [1]. Funded teams are asked to build production-grade, security-critical components with machine-checked proofs of their key properties, using AI as a central means of reaching a level of ambition that would otherwise be out of reach.

Our focus in Keystone is on creating LLM inference serving infrastructure with strong, machine-checked correctness guarantees. Inference serving software is now a form of critical infrastructure: the routine dependence of businesses, governments and members of the public on LLMs now means that denial-of-service vulnerabilities or isolation failures between tenants whose queries are being served via the same GPU have the potential to be very serious.

An appealing feature of inference as a verification target is that, unusually for a large software system, it has a clean specification: a transformer computes the next token as a well-defined mathematical function of the model’s weights and the embeddings of the preceding tokens. Showing that a high-performance inference implementation—with optimisations such as paged attention, prefix caching and continuous batching—refines that function is a challenging but well-defined task, and I am excited about the progress we can make thanks to the assistance that AI can provide.

Acknowledgements

I am grateful to Amber Gorzynski, James Lee-Jones, Manuel Rigger and John Wickerson for their feedback on drafts of this paper.

The work that I summarise here was done in collaboration with many Masters and PhD students at Imperial, postdoctoral and faculty colleagues at Imperial, and academic and industrial collaborators internationally. I am grateful to all of them.

The work was supported by numerous grants over the years, including the EU FP7 STREP project CARP (project number 287767), an EPSRC Early Career Fellowship (EP/N026314/1), the EPSRC IRIS Programme Grant (EP/R006865/1), two Amazon Research Awards, gift funding from Google, and a grant from the Advanced Research and Invention Agency (ARIA).

References

- 1 Advanced Research and Invention Agency. Safeguarded AI: Cybersecurity – Call for Proposals. https://aria.org.uk/media/1padxpaf/safeguarded-ai_cybersecurity_solicitation.pdf, May 2026. Version 1.0, 20 May 2026. Accessed 15 October 2026.
- 2 June Andronick, D. Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu. Large-scale formal verification in practice: A process perspective. In Martin Glinz, Gail C. Murphy, and Mauro Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1002–1011. IEEE Computer Society, 2012. doi:10.1109/ICSE.2012.6227120.
- 3 Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *Int. J. Softw. Tools Technol. Transf.*, 11(1):69–83, 2009. doi:10.1007/S10009-008-0091-0.

- 4 Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In Eerke A. Boiten, John Derrick, and Graeme Smith, editors, *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2004. doi:10.1007/978-3-540-24756-2_1.
- 5 Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In Michael Burke and Mary Lou Soffa, editors, *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, pages 203–213. ACM, 2001. doi:10.1145/378795.378846.
- 6 Ethel Bardsley, Adam Betts, Nathan Chong, Peter Collingbourne, Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Daniel Liew, and Shaz Qadeer. Engineering a static verification tool for GPU kernels. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2014. doi:10.1007/978-3-319-08867-9_15.
- 7 Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387. Springer, 2005. doi:10.1007/11804192_17.
- 8 Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011. doi:10.1007/978-3-642-22110-1_14.
- 9 Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallen, Charles-Henri Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, 2010. doi:10.1145/1646353.1646374.
- 10 Adam Betts, Nathan Chong, Pantazis Deligiannis, Alastair F. Donaldson, and Jeroen Ketema. Implementing and evaluating candidate-based invariant generation. *IEEE Trans. Software Eng.*, 44(7):631–650, 2018. doi:10.1109/TSE.2017.2718516.
- 11 Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10:1–10:49, 2015. doi:10.1145/2743017.
- 12 Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007. doi:10.1007/978-3-540-73368-3_51.
- 13 Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011. doi:10.1007/978-3-642-22110-1_16.
- 14 Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam,*

- The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999. doi:10.1007/3-540-49059-0_14.
- 15 Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Cal-culemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2012. doi:10.1007/978-3-642-31374-5_3.
 - 16 Jonathan P. Bowen. The Z notation: Whence the cause and whither the course? In Zhiming Liu and Zili Zhang, editors, *Engineering Trustworthy Software Systems - First International School, SETSS 2014, Chongqing, China, September 8-13, 2014. Tutorial Lectures*, volume 9506 of *Lecture Notes in Computer Science*, pages 103–151. Springer, 2014. doi:10.1007/978-3-319-29628-9_3.
 - 17 Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986. doi:10.1109/TC.1986.1676819.
 - 18 Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Inf. Comput.*, 98(2):142–170, 1992. doi:10.1016/0890-5401(92)90017-A.
 - 19 Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clément Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas G. Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Taejoon Byun, Joshua M. Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. Formally verified cloud-scale authorization. In *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*, pages 2508–2521. IEEE, 2025. doi:10.1109/ICSE55347.2025.00166.
 - 20 T.Y. Chen, S.C. Cheung, and S.M. Yiu. Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998.
 - 21 Nathan Chong, Alastair F. Donaldson, Paul H. J. Kelly, Jeroen Ketema, and Shaz Qadeer. Barrier invariants: a shared state abstraction for the analysis of data-dependent GPU kernels. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 605–622. ACM, 2013. doi:10.1145/2509136.2509517.
 - 22 Maria Christakis, Alastair F. Donaldson, John Regehr, and Thodoris Sotiropoulos. Testing program analyzers and verifiers (Dagstuhl Seminar 25242). *Dagstuhl Reports*, 15(6):69–83, 2025. doi:10.4230/DAGREP.15.6.69.
 - 23 Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000. doi:10.1007/10722167_15.
 - 24 Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. doi:10.1145/186025.186051.
 - 25 Edmund M. Clarke, Somesh Jha, Reinhard Enders, and Thomas Filkorn. Exploiting symmetry in temporal logic model checking. *Formal Methods Syst. Des.*, 9(1/2):77–104, 1996. doi:10.1007/BF00625969.

- 26 Edmund M. Clarke, William Klieber, Milos Nováček, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. doi:10.1007/978-3-642-35746-6_1.
- 27 Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. doi:10.1007/978-3-540-24730-2_15.
- 28 Lucas C. Cordeiro, Bernd Fischer, and João Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *IEEE Trans. Software Eng.*, 38(4):957–974, 2012. doi:10.1109/TSE.2011.59.
- 29 Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In Alwyn Goodloe and Suzette Person, editors, *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings*, volume 7226 of *Lecture Notes in Computer Science*, pages 120–125. Springer, 2012. doi:10.1007/978-3-642-28891-3_12.
- 30 Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction - CADE 28 - 28th International Conference on Automated Deduction, Virtual Event, July 12-15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.
- 31 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Efficient E-matching for SMT solvers. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 183–198. Springer, 2007. doi:10.1007/978-3-540-73595-3_13.
- 32 Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.
- 33 Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, 1979. doi:10.1145/359104.359106.
- 34 Edsger W. Dijkstra. Notes on structured programming. Technical Report 70-WSK-03, Technische Hogeschool Eindhoven, April 1970. EWD249. URL: <https://www.cs.utexas.edu/~EWD/ewd02xx/EWD249.PDF>.
- 35 Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O’Hearn. Scaling static analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019. doi:10.1145/3338112.
- 36 Alastair F. Donaldson. The GPUVerify method: a tutorial overview. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, 70, 2014. doi:10.14279/TUJ.ECEASST.70.986.
- 37 Alastair F. Donaldson. Randomised generation formalisation: Proof that GEN = RE, 2026. URL: <https://github.com/mc-imperial/RandomisedGenerationFormalisation>.
- 38 Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *Proc. ACM Program. Lang.*, 1(OOPSLA):93:1–93:29, 2017. doi:10.1145/3133917.
- 39 Alastair F. Donaldson, Hugues Evrard, and Paul Thomson. Putting randomized compiler testing into production (experience report). In Robert Hirschfeld and Tobias Pape, editors, *34th European Conference on Object-Oriented Programming, ECOOP 2020, Berlin, Germany*

- (Virtual Conference), November 15–17, 2020, volume 166 of *LIPICs*, pages 22:1–22:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.EC00P.2020.22.
- 40 Alastair F. Donaldson, Marios Kogias, John Wickerson, Thomas Bourgeat, Nate Foster, and Clément Pit-Claudel. Keystone: verifying the AI inference stack. <https://aria.org.uk/opportunity-spaces/trust-everything-everywhere/safeguarded-ai/funded-projects?tabId=applications&cardId=keystone>, 2026. Advanced Research and Invention Agency (ARIA), accessed 13 October 2026.
 - 41 Alastair F. Donaldson, Dilan Sheth, Jean-Baptiste Tristan, and Alex Usher. Randomised testing of the compiler for a verification-aware programming language. In *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27–31, 2024*, pages 407–418. IEEE, 2024. doi:10.1109/ICST60714.2024.00044.
 - 42 E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods Syst. Des.*, 9(1/2):105–131, 1996. doi:10.1007/BF00625970.
 - 43 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 – where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16–24, 2013. Proceedings*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013. doi:10.1007/978-3-642-37036-6_8.
 - 44 Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3–9, 2023*, pages 1229–1241. ACM, 2023. doi:10.1145/3611643.3616243.
 - 45 Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12–16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 500–517. Springer, 2001. doi:10.1007/3-540-45251-6_29.
 - 46 Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. doi:10.1007/3-540-60761-7.
 - 47 Anthony Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, 1990. doi:10.1109/52.57887.
 - 48 Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In John Launchbury and John C. Mitchell, editors, *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16–18, 2002*, pages 58–70. ACM, 2002. doi:10.1145/503272.503279.
 - 49 Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2004.
 - 50 Ahmed Irfan, Sorawee Porncharoenwase, Zvonimir Rakamaric, Neha Rungta, and Emina Torlak. Testing Dafny (experience paper). In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 556–567. ACM, 2022. doi:10.1145/3533767.3534382.
 - 51 Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge, MA, revised edition, 2012.
 - 52 Daniel Jackson. Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76, 2019. doi:10.1145/3338843.
 - 53 Ranjit Jhala and Kenneth L. McMillan. A practical and complete approach to predicate refinement. In Holger Hermanns and Jens Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as*

- Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings*, volume 3920 of *Lecture Notes in Computer Science*, pages 459–473. Springer, 2006. doi:10.1007/11691372_33.
- 54 Sára Juhošová, Andy Zaidman, and Jesper Cockx. Pinpointing the learning obstacles of an interactive theorem prover. In *33rd IEEE/ACM International Conference on Program Comprehension, ICPC@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*, pages 159–170. IEEE, 2025. doi:10.1109/ICPC66645.2025.00024.
 - 55 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neeffe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.
 - 56 Vasileios Klimis, Jack Clark, Alan Baker, David Neto, John Wickerson, and Alastair F. Donaldson. Taking back control in an intermediate representation for GPU computing. *Proc. ACM Program. Lang.*, 7(POPL):1740–1769, 2023. doi:10.1145/3571253.
 - 57 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.
 - 58 Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In Michael F. P. O’Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 216–226. ACM, 2014. doi:10.1145/2594291.2594334.
 - 59 James Lee-Jones, John Wickerson, and Alastair F. Donaldson. Uniformity analysis in the WebGPU shading language. *Proc. ACM Program. Lang.*, 10(PLDI):2109–2133, 2026. doi:10.1145/3808331.
 - 60 K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.
 - 61 K. Rustan M. Leino and Michał Moskal. Usable auto-active verification, November 2010. URL: <https://leino.science/papers/krm1212.pdf>.
 - 62 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.
 - 63 Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 65–76. ACM, 2015. doi:10.1145/2737924.2737986.
 - 64 Daniel Matichuk, Toby C. Murray, June Andronick, D. Ross Jeffery, Gerwin Klein, and Mark Staples. Empirical study towards a leading indicator for cost of formal software verification. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 722–732. IEEE Computer Society, 2015. doi:10.1109/ICSE.2015.85.
 - 65 Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, volume 9583 of *Lecture Notes in Computer Science*, pages 41–62. Springer, 2016. doi:10.1007/978-3-662-49122-5_2.

- 66 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.
- 67 Michalis Pardalos, Alastair F. Donaldson, Emiliano Morini, Laura Pozzi, and John Wickerson. Who checks the checkers? Automatically finding bugs in C-to-RTL formal equivalence checkers. In *DVCon Europe*, 2024. doi:10.30420/566438006.
- 68 Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.*, 4(OOPSLA):211:1–211:30, 2020. doi:10.1145/3428279.
- 69 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *Found. Trends Program. Lang.*, 5(2-3):102–281, 2019. doi:10.1561/25000000045.
- 70 Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Commun. ACM*, 61(4):58–66, 2018. doi:10.1145/3188720.
- 71 Robin Salkeld. Proof, but at what cost? Talk at the 22nd Annual High Confidence Software and Systems Conference (HCSS 2022), May 2022. Abstract and recording available at <https://sos-vo.org/node/83866>.
- 72 Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and monadic effects in F^* . In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 256–270. ACM, 2016. doi:10.1145/2837614.2837655.
- 73 The Rocq Development Team. The Rocq prover, 2026. URL: <https://doi.org/10.5281/zenodo.1003420>.
- 74 Aaron Tomb and Jean-Baptiste Tristan. Avoiding verification brittleness in Dafny. Dafny Blog, <https://dafny.org/blog/2023/12/01/avoiding-verification-brittleness/>, December 2023. Accessed 13 October 2026.
- 75 Evan Wang, Simon Chess, Daniel Lee, Siyuan Ge, Ajit Mallavarapu, and Vasily Ilin. Learning to repair lean proofs from compiler feedback. *CoRR*, abs/2602.02990, 2026. arXiv:2602.02990, doi:10.48550/ARXIV.2602.02990.
- 76 Elaine J. Weyuker. On testing non-testable programs. *Comput. J.*, 25(4):465–470, 1982. doi:10.1093/COMJNL/25.4.465.
- 77 Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 718–730. ACM, 2020. doi:10.1145/3385412.3385985.
- 78 Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 283–294. ACM, 2011. doi:10.1145/1993498.1993532.
- 79 Pingshi Yu, Chengsong Tan, Nicolas Wu, and Alastair F. Donaldson. Complexity theory of randomised testing. *CoRR*, abs/2607.11811, 2026. arXiv:2607.11811, doi:10.48550/ARXIV.2607.11811.
- 80 Chengyu Zhang and Zhendong Su. SMT2Test: From SMT formulas to effective test cases. *Proc. ACM Program. Lang.*, 8(OOPSLA2):222–245, 2024. doi:10.1145/3689719.
- 81 Yi Zhou, Jay Bosamiya, Yoshiki Takashima, Jessica Li, Marijn Heule, and Bryan Parno. Mariposa: Measuring SMT instability in automated program verification. In Alexander Nadel and Kristin Yvonne Rozier, editors, *Formal Methods in Computer-Aided Design, FMCAD*

2023, Ames, IA, USA, October 24-27, 2023, pages 178–188. IEEE, 2023. doi:10.34727/2023/
ISBN.978-3-85448-060-0_26.

