Keeping the Human in the Loop: A Principled Approach to Verified AI-Generated Code

Vadim Zaliva
lord@crocodile.org

Abstract

AI-assisted program synthesis is already commonplace; the challenge is to make it trustworthy by formally verifying its results. The industrial adoption of formal verification has long been hindered by two factors: the difficulty of producing formal specifications and the cost of constructing and maintaining proofs. AI is now actively discussed for both code generation and proof, but this discussion often overlooks the key problem: where is the human in this loop, and in what role? No matter how many adversarial or collaborating AI agents are used to generate and prove the code, the question remains how to ensure that the result matches the human intent. We outline a workflow for AI-generated code that addresses both obstacles, the difficulty of specification and the cost of proof, by grounding code generation and proof construction in a shared, human-reviewed intermediate specification. While AI accelerates formalisation, code generation, and proof development, trust is preserved through human validation of the intermediate specification and mechanised proof checking of the final result. This is a position paper, giving rise to several directions for future research, which we highlight.

1 Where Does the Human Go?

AI code generation is now prevalent, and randomised trials find that it makes developers measurably more productive [1–3]. Yet people are uncomfortable leaving the whole process to AI unattended, and with reason: trust in AI-generated code has been shown to be misplaced [4]. The answer widely discussed [5–8] is *formal verification*: proving that the generated code meets its specification (Section 5 surveys this work). Formal verification itself has two laborious steps, producing the formal specification and constructing the proof, and AI is now actively pursued for both. This is the setting we consider in this paper: AI-assisted code generation with formal proofs of correctness.

To generate code with the help of AI, a human must first codify their *intent*: what they want the AI to achieve. A specification. These *informal specifications* ("prompts") are now often given to AI coding agents by users in natural language. However, they are not precise enough for formal verification, which requires *formal specifications* (e.g. in the language of the Lean or Rocq proof assistants) to verify code against.

It is tempting to use AI to derive a formal specification from an informal one. However, if one uses an informal human-language specification as ground truth and then relies on it both to generate code and to verify it, the ambiguities of human language, specification gaps, and AI hallucinations open a door to misinterpretation of the user's intent when generating formal specs, which then propagates further to code and proofs.

Most of the work on AI for code generation and for proof concentrates on the AI side of the loop: on better models, on multiple agents that collaborate or check each other adversarially, and on feeding the proof assistant's verdicts back into generation. What is usually left implicit is the other side: where the human sits in this loop, and in what role. The agents can be no more faithful to the intent

Preprint.

than the input they start from, and none of them can tell whether that input is what the human meant. The question is therefore not whether to keep a human in the loop, but where to put them and what to make them responsible for. This is the position of this paper: the place and role of human involvement is the key design decision in AI-assisted verified code generation, and the rest of our proposal follows from it.

We will consider the following actors: 1) *Humans*, the ultimate source of truth: their decisions are authoritative and must be obeyed. An informal statement of intent, however, is ambiguous and incomplete and so cannot serve as such a decision; the decision is the specification the human approves once its ambiguities and gaps have been resolved (Section 2). 2) *AI*, inherently untrusted: its output must be verified. 3) An *automatic proof verifier* (such as a proof assistant), assumed to be reliable. Naturally, we want to offload most work to AI, but reduced human involvement means more specification gaps to be filled by AI, creating more opportunities to misinterpret human intent.

2 Verified Trust Chain with a Human in the Loop

Our answer is to ground both code generation and proof construction in a shared, human-reviewed *intermediate specification*.

A specification, whether informal or formal, differs from a sketch of the code: it constrains the space of acceptable solutions rather than proposing one. It is declarative, and an implementation may satisfy it with any of several imperative solutions that differ in performance, complexity, and other non-functional properties. In this sense a specification is a contract: it fixes what counts as correct and leaves the rest to the implementation.

As argued in the introduction, leaving it to AI alone to derive the formal specification from the informal one risks misinterpreting the intent. AI can still be useful to analyse, distil, and formalise specs written in human language by translating them into a less ambiguous form, but humans still need to review the result and confirm that it is exactly what they meant. One possible solution is to use an *intermediate specification language*: a semi-formal notation, stricter and less ambiguous than informal prose, but still readable by humans for understanding and review. The reviewer confirms intent against a text that leaves little ambiguity to resolve, yet, unlike a fully formal specification, one that remains accessible without formal-methods expertise.

The proposed AI-assisted development workflow is shown in Figure 1.

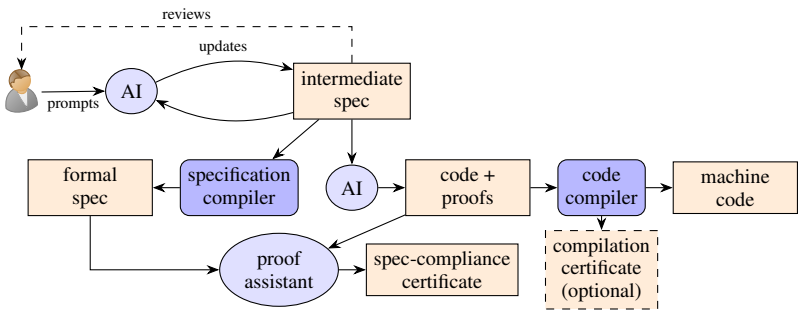
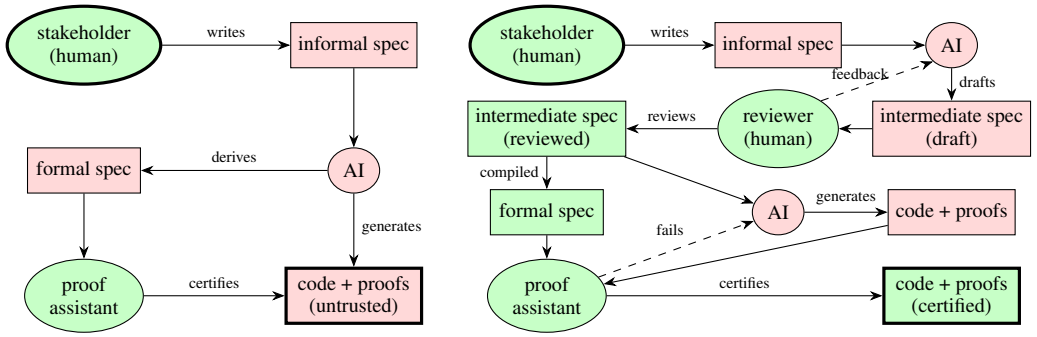


Figure 1: An AI-assisted workflow for specification, code generation, and verification.

In this workflow, a human guides an AI through natural-language prompts to iteratively produce and refine an intermediate specification, reviewing it after each round until it captures the intended behaviour. Once approved, the intermediate specification serves two roles: AI uses it to generate code, while a specification compiler derives a formal specification from it. A proof assistant then verifies the generated code against the formal specification and issues a spec-compliance certificate. Finally, a *code compiler* translates the code to machine code: either a verified compiler such as CompCert [9] or CakeML [10], whose correctness theorem, proven once for all programs, guarantees that the machine code preserves the behaviour of the verified code, or a certifying compiler [9], which emits a compilation certificate of the same fact for each run. AI is used at two points: specification refinement, and the generation of code and proofs (detailed in Section 4). In each case the result is checked, the first by a human, the second by a proof assistant. If implemented, this workflow

gives formal correctness guarantees from the *intermediate specification* down to machine code; the specification itself, and the human review that endorsed it, are trusted rather than proven.

This gives our approach a clear *chain of trust*. It does not start at the informal specification, which, although written by the human, is not authoritative, since it may be self-contradictory, ambiguous, or incomplete, nor at the AI, which is untrusted by assumption. It starts at the intermediate specification: the contractual artefact that bridges intent and formal semantics. AI helps to write it, but the veracity of the whole process is grounded in the human review of the intermediate specification. From there the chain continues to the formal specification, which is compiled from the intermediate one, and to the code and the proofs, which are checked against the formal specification. Figure 2 contrasts this with the naïve scheme in which an informal specification is used both to generate the code and to derive the formal specification.



(a) The naïve scheme.

(b) Our approach.

Figure 2: Trust in the naïve scheme (a) and in our approach (b). Red: untrusted; green: trusted; dashed: the two loops.

The picture is inspired by secure information flow [11]: *trusted* and *untrusted* are integrity *labels*. An artefact is trusted only if it derives from trusted inputs processed by a trusted module; anything an untrusted step produces keeps the untrusted label until a trusted party *endorses* it, much as a sanitiser does for tainted input. In the naïve scheme the human writes an informal spec, and their involvement ends there. An untrusted AI derives from it both the formal specification and the code and proofs. The proof assistant itself is trusted, but it certifies the code and proofs against a formal specification that is itself untrusted, so the result stays untrusted: nothing ties the formal specification to the human intent, its ambiguities pass down the chain unchecked, and no number of AI agents, adversarial or collaborating, changes that. In our approach the involvement of the human specifying the problem is confined to the intermediate specification, which they review. An untrusted AI is still involved below it, but the final artefacts, the code and proofs, are trusted: the AI is contained by the rigid formal specification on one side and the proof assistant check on the other. The two artefacts the AI produces on our side thus have a dual nature, shown by the paired boxes: untrusted as generated, and trusted once endorsed, the intermediate specification by human review and the code and proofs by the proof assistant's certificate. In both places an untrusted step is *contained*: its output is endorsed by what follows it, human review in the first case, proof assistant verification in the second, before anything downstream depends on it. This containment is what lets the chain of trust hold. The dashed arrows in Figure 2 are two loops: the human's feedback to the drafting AI, and a failed check returned to the generating AI. Both are run by a *harness*, the conventional program around the AI that feeds it the result, invokes it again, and lets nothing leave the loop without the human's approval or the proof assistant's certificate.

The chain also depends on the correctness of the specification compiler, the link between the intermediate and the formal specification (Section 3). Whichever way that correctness is established, the adequacy of the intermediate specification's formal semantics to its human reading remains in the trusted base.

The consequence of resting the chain on human review is that the intermediate specification may not be as simple as the informal prompts people envision for code generation. Producing verified code under our approach will require some human effort and skill. Engineers will not normally be writing code or proofs, but they will be managing (owning) the intermediate specification.

Take the AI out of the scheme and put a team of engineers in its place, and what remains is a familiar process with familiar roles. The human stating the intent is the *product owner* of Scrum [12] or the *on-site customer* of extreme programming [13]. The intermediate specification is the requirements artefact: the requirements specification and use-case model of the Rational Unified Process [14], or the *acceptance criteria* of a user story, which behaviour-driven development [15–17] already writes in a constrained language that the product owner approves and a tool executes. Reviewing the intermediate specification is the requirements sign-off.

3 Specification Development

Section 2 rests the chain of trust on the intermediate specification. This section looks at how it is produced, what it is written in, and how it is compiled into the formal specification.

The AI's part in producing the intermediate specification is twofold. It helps to elaborate the informal specification by pointing out gaps, inconsistencies, and ambiguities in it. It can also offer an intermediate specification rendering of the informal one, filling some of the gaps and making assumptions of its own, for the human to review. Through this iterative process the human addresses the issues raised, accepts or corrects the assumptions made, and refines the intermediate specification until it is complete and unambiguous. The AI's help here is advisory and ultimately untrusted; the human is the final arbiter of the intermediate specification and its correctness.

That completeness is relative to the decisions taken so far: software projects usually start from a vague specification, "the idea", and detail it as development proceeds and decisions are made. The intermediate specification language and its tooling must support this incremental development over the life of a project rather than one-shot authoring.

The cycle needs tool support. The human has to see the gaps and assumptions the AI raises, answer them, and know at any point what the specification still leaves open, what has been endorsed, and what has changed since. A simple text editor does not provide that. Supporting the cycle is thus an HCI problem as much as a language one.

The intermediate specification language has two readers: the human specifying the problem, who need not be a formal-methods expert, and the specification compiler, which needs a formal semantics. These are compatible: a semi-formal notation that a human reads without formal-methods training can still be given a formal semantics. Constrained natural languages for requirements, such as EARS [18], are a precedent rather than a solution. EARS restricts each requirement to one of a few sentence templates built from a fixed set of keywords, but its requirements are written to be read by people, whereas an intermediate specification must also carry enough semantics to be compiled into a formal one.

The intermediate specification does not have to be text. We take one point from UML [19] (which is in many other ways far from what we want): a specification notation can combine diagrams with text. A diagram with a defined semantics compiles as a sentence does, though writing and reviewing it takes tooling, an IDE rather than a text editor.

Unlike the AI components of the workflow, the specification compiler is deterministic, conventional software. It can be *verified* once against a formal semantics of the intermediate specification language, or made *certifying*, emitting for each run a proof, checked by the same proof assistant, that the generated formal specification matches that semantics. Either way, the chain of trust depends on one property: the formal specification must *refine* the intermediate one, so that code proven against the former satisfies the latter. The compiler may over-constrain but never weaken, and it takes no input from the code or the AI, so that the human's review of the intermediate specification determines the formal one.

Since the specification is declarative (Section 2), what it leaves open about the implementation, such as the representation of the program's state, is not a gap but a choice deferred to the code, and any choice that satisfies the specification is acceptable. The compiler *parametrises* such choices: the formal specification becomes an interface with laws, the generated code chooses an instantiation, and the proof assistant checks it.

What it leaves open about the functionality is a gap: the specification is incomplete. Suppose, for example, that it describes a function returning a user's account balance but does not say what happens

when the user does not exist. Whether the call fails or returns a zero balance is not an implementation detail but part of the specification, and the human has not reviewed it. The compiler must not make that choice. It flags the gap and may propose a resolution, which enters the intermediate specification only once the human has reviewed and approved it. Detecting gaps is heuristic; a gap the compiler misses is caught later as a counterexample (Section 4).

4 Proof and Code Generation

The workflow of Section 2 leaves open how the code and its proofs are produced below the intermediate specification. Both are generated with the help of AI, and both are checked; we now look at that part of the picture more closely.

Writing formal proofs by hand is laborious and difficult, while current automatic proving techniques such as SAT and SMT solving are limited in their capabilities. Even with the current state of AI models, it has been shown that many formal proofs can be generated automatically by AI [8, 20, 21], replacing costly labour that previously required highly specialised proof engineers.

How much of this can be automated depends in part on the language the code is written in. Mainstream languages are designed for humans writing and reading programs, not for formal verification: underspecified semantics, undefined behaviour, and no built-in place for proof-level information such as invariants and assumptions. Here the division of responsibilities pays off again: what the human owns is the intermediate specification, not the code, so the code can be written in a "proof-friendly" language designed for AI to generate and a proof assistant to check, with a concise, human-readable projection for the occasions when humans do need to examine it. The workflow does not prescribe such a language, and we do not pursue its design here.

Starting from Figure 1, we zoom in on proof generation and verification, shown in Figure 3. From the *intermediate specification*, AI generates the code. Using the *formal specification* (also derived from the *intermediate specification*), a proof synthesis engine produces a proof script; the proof assistant verifies the script and thereby that the code meets the *formal specification*.

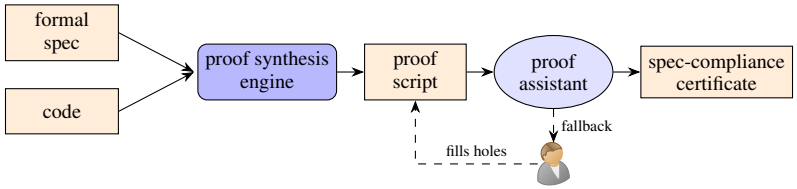


Figure 3: AI-assisted proof generation and verification.

The proof synthesis engine may not be able to automate every part of the proof. In this case there are two options. 1) It can leave the parts it cannot prove as explicit *holes* in the proof script, which a proof engineer fills by proving them interactively with the proof assistant. Once no holes remain, the proof assistant issues the final *certificate*. 2) An obligation the engine cannot discharge statically can instead be compiled into a runtime assertion. The proof assistant then certifies a weaker statement: the code meets its specification on every execution in which no assertion fires. This falls short of the guarantee of Section 2; the certificate states the weaker claim, and whether it is acceptable is the human's decision.

A failed obligation is either a hole or a fault. It is a hole if the engine can neither prove nor refute it, and the two options above apply. It is a fault if the engine refutes it with a counterexample, and the counterexample is against either the code or the specification. Against the code, a different program could still satisfy the specification in the case the counterexample exhibits, so the counterexample goes back to the generating AI and no human is involved. Against the specification, it exposes a gap in the intermediate specification the compiler did not see, and closing it requires the human to change the intermediate specification and review it again.

The proof synthesis engine uses AI as well as various heuristics and techniques such as SAT and SMT solving. However, the resulting proof script is verified by a proof assistant which satisfies the de Bruijn criterion [22], ensuring that the proof is sound and that the certificate it issues can be rechecked independently.

So far we have presented a more or less traditional approach: code synthesis followed by automatic proof generation. Generating the two together, so that the proof obligations guide the code as it is written, is a possibility we leave to Section 6.

5 Related Work

The combination of AI and formal verification is an active and rapidly developing area, and what follows is a current snapshot of it. We group related work by the part of the workflow it addresses: proof synthesis, verified code generation, the step from intent to specification, and spec-driven development tooling.

Proof synthesis. Neural provers generate proofs for proof assistants, on Isabelle/HOL and Lean benchmarks [20, 23], on competition mathematics [24, 25], up to problems of the 2024 International Mathematical Olympiad [26], and increasingly on software: Selene and AutoReal target the seL4 proofs [27, 28], and AutoRocq discharges the proof obligations of C programs in Rocq [29]. In all of these the prover is given the theorem statement; Selene names the specification as the other labour-intensive half of verification and addresses only the proofs.

Verified code generation. In verification-aware languages the LLM writes the code, the proof annotations, or both, and the verifier checks the result: Dafny [30, 31], Verus [7, 32], F* [33], and C with ACSL [34]. The formal specification is either given as input [32, 34] or produced by the AI, together with the code from the informal description [30], which is the naïve scheme of Figure 2, or from existing code [35]. Clover replaces review of the specification by consistency checks among docstring, annotations, and code, and names human oversight as the obstacle to scale [31]; AlphaVerus makes the same argument [7]. The experience reports are the exception: in agentic proof-oriented programming the human's part reduces to describing the problem, reviewing the generated specifications, and occasionally supplying a key invariant [8], and in the machine-generated proof of a CertiCoq compiler pass the human gave high-level guidance and reviewed the generated statements and proofs without writing any [21].

From intent to specification. Autoformalisation translates informal statements into formal ones, in mathematics [36] and for programs, where intent becomes postconditions [37, 38]. Lahiri argues that this intent gap is the reliability bottleneck of AI coding [6], and that there is no algorithmic way to check that a formal specification matches the intent [38], so these systems are evaluated by automated proxies. Several systems put a human in the loop: nl2spec lets the user edit and approve sub-translations of a requirement before the LTL formula is accepted [39]; Req2LTL and AeroReq2LTL insert a structured or templated intermediate representation that a human can inspect before it is translated to LTL [40, 41]; AWS's policy formaliser offers an optional vetting phase in which inspecting the generated policy model is likened to code review [42]; and Gordon and Matskevich give a subset of English a formal semantics inside Lean, with certificates that record how each word was read [43]. These are the closest to our intermediate specification, and they stop where our chain of trust starts: the reviewed artefact is their output, and no code is generated from it and proven against it.

Spec-driven development. Spec-driven development with AI assistance is already appearing in industrial tooling. Amazon's Kiro IDE [44] turns a natural-language prompt into a requirements document in EARS form, which the developer reviews before the tool derives a design, a task breakdown, and code from it; the reviewing role this gives the requirements document is close to the one we ask of the intermediate specification. GitHub's Spec Kit and structured spec-driven engineering supply the agent with a written specification instead of a prompt [45, 46], and Monperrus argues that specification review should replace code review as the quality gate [47]. In all of these the chain stops at the specification: the code is checked against it by tests and inspection, not by proof, so the human's sign-off vouches for the specification and for nothing below it. Closest to our proposal is the Atlas Computing toolchain roadmap, in which the user evaluates and improves a formal specification before tools generate an implementation and a proof checked by a small trusted kernel [5], with a prototype IDE for reviewing mechanised specifications against their requirements [48]. Atlas formalises in the opposite order. It produces the formal specification first and renders it back into prose for the human to evaluate. We refine in one direction, from informal to intermediate

to formal. Between the informal statement and the intermediate specification, the human fills the omissions and resolves the ambiguities and conflicts. Between the intermediate and the formal specification, the compiler chooses the mathematical abstractions and adds the technical detail the human does not care about. Formalising first makes that choice before anyone has reviewed the intent, and the rendering back into prose then has to undo it. The concepts the human thinks in have already been replaced by mathematical ones, so the person asked to endorse the rendering may not recognise the problem in it, and nothing checks that the rendering is faithful. In our scheme the human reviews the artefact itself, and the translation below it can be verified.

6 Future Research Directions

We have proposed an end-to-end workflow in which AI produces both the code and its proofs, with the human in the loop, and a chain of trust runs from the intermediate specification the human approved down to the machine code, with every link below that approval machine-checked. This is a high-level outline. It sketches a solution rather than giving one, and we have not tried to answer every question it raises. Much remains to be worked out, and the rest of this section lists the research directions we see for that work.

Intermediate specification language design. What notation can the human specifying the problem read and review, and still carry enough semantics to compile? Constrained natural language is one end of the range and a controlled subset of a formal specification language the other; where between them the intermediate language should sit is open. So is whether it should be textual at all: which parts of a specification are better drawn than written, and which diagram notations can be given a semantics precise enough to compile. Another open question is extensibility: the language needs a way to define domain vocabulary, give it a semantics, and reuse the definitions across projects in the same domain. The human's review must scale as well: rather than one monolithic document for the whole project, the specification should be a set of sub-specifications, each written and reviewed on its own, trusted from then on, and only then connected to the others. What the language needs for this, interfaces between sub-specifications and separate compilation, is open. Hints for code generation and proof synthesis, which are no part of the specification's semantics, could be embedded in the language, like inline assembly or comments in code, or kept as a separate artefact linked to the specification.

Interactive specification development. The human's review is not checked mechanically, so the tooling around it decides how strong the chain is. Can the tool help the human see what the specification implies, for instance by generating examples and counterexamples from it? Specifications are developed incrementally, so what can the workflow do with one that is still incomplete: can code be generated and checked against the parts that are settled? And when the specification changes, can the human review only the change rather than the whole?

The specification compiler. Whether to make the compiler *verified*, proven correct once, or *certifying*, checking each run by translation validation. Whether a separate compiler is needed at all: if the semantics of the intermediate language is given as a shallow embedding in the proof assistant, the compiler and the semantics coincide, and the question becomes what that embedding looks like. How implementation choices are left open in the formal specification, as interfaces with laws or otherwise. How the compiler detects the functionality gaps it must flag and chooses the resolutions it proposes; these heuristics need not be proven correct, since the human reviews what they produce, whereas the translation must. Whether the compiler can use the proof assistant to check the specification itself, such as its consistency, and report the problems it finds back to the human.

The proof synthesis engine. How to combine AI with heuristics and SAT/SMT solving, and whether specially trained models are needed or general ones suffice. What to do with the obligations the engine cannot discharge: when a proof engineer should fill the remaining holes, when a runtime check is an acceptable substitute, and how to tell a counterexample against the code from one against the specification. A further possibility is generating the code and its proof together, so that the proof obligations guide the code.

References

[1] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of AI on developer productivity: Evidence from GitHub Copilot. arXiv preprint arXiv:2302.06590, 2023. URL https://arxiv.org/abs/2302.06590.

[2] Elise Paradis, Kate Grey, Quinn Madison, Daye Nam, Andrew Macvean, Vahid Meimand, Nan Zhang, Ben Ferrari-Church, and Satish Chandra. How much does AI impact development speed? an enterprise-based randomized controlled trial. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 618–629. IEEE, April 2025. doi: 10.1109/icse-seip66354.2025.00060. URL http://dx.doi.org/10.1109/icse-seip66354.2025.00060.

[3] Kevin Zheyuan Cui, Mert Demirer, Sonia Jaffe, Leon Musolff, Sida Peng, and Tobias Salz. The effects of generative AI on high-skilled work: Evidence from three field experiments with software developers. *Management Science*, February 2026. ISSN 1526-5501. doi: 10.1287/mnsc.2025.00535. URL http://dx.doi.org/10.1287/mnsc.2025.00535.

[4] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with AI assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 2785–2799. ACM, November 2023. doi: 10.1145/3576915.3623157. URL http://dx.doi.org/10.1145/3576915.3623157.

[5] Shaowei Lin, Evan Miyazono, and Daniel Windham. A toolchain for AI-assisted code specification, synthesis and verification. Report Version 1.2, Atlas Computing, January 2025. URL https://atlascomputing.org/ai-assisted-fv-toolchain.pdf.

[6] Shuvendu K. Lahiri. Intent formalization: A grand challenge for reliable coding in the age of AI agents. arXiv preprint arXiv:2603.17150, 2026. URL https://arxiv.org/abs/2603.17150.

[7] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. AlphaVerus: Bootstrapping formally verified code generation through self-improving translation and treefinement. In Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu, editors, *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 587–615. PMLR, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/aggarwal25a.html.

[8] Eleftherios Ioannidis, Nikhil Swamy, Gabriel Ebner, Matthai Philipose, and Tahina Ramananandro. Proofs promptly: Proof-oriented programming with AI agents (experience report). *Proc. ACM Program. Lang.*, 10(ICFP):1059–1077, 2026. doi: 10.1145/3828709. URL https://doi.org/10.1145/3828709.

[9] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52 (7):107–115, July 2009. ISSN 1557-7317. doi: 10.1145/1538788.1538814. URL http://dx.doi.org/10.1145/1538788.1538814.

[10] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. The verified CakeML compiler backend. *Journal of Functional Programming*, 29, 2019. ISSN 1469-7653. doi: 10.1017/s0956796818000229. URL http://dx.doi.org/10.1017/S0956796818000229.

[11] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003. doi: 10.1109/JSAC.2002.806121.

[12] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, 2001. ISBN 0-13-067634-9.

[13] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000. ISBN 0-201-61641-6.

[14] Philippe Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 3rd edition, 2003. ISBN 0-321-19770-4.

[15] Dan North. Introducing BDD. https://dannorth.net/blog/introducing-bdd/, 2006. Accessed 9 September 2026.

[16] Gojko Adzic. *Specification by Example: How Successful Teams Deliver the Right Software*. Manning, 2011. ISBN 978-1-61729-008-4.

[17] Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387, 2011. doi: 10.1109/SEAA.2011.76.

[18] Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy approach to requirements syntax (EARS). In *2009 17th IEEE International Requirements Engineering Conference*, pages 317–322. IEEE, 2009. doi: 10.1109/RE.2009.9.

[19] Object Management Group. Unified Modeling Language specification, version 2.5.1. Specification formal/17-12-05, Object Management Group, December 2017. URL https://www.omg.org/spec/UML/2.5.1.

[20] 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. URL https://doi.org/10.1145/3611643.3616243.

[21] Zoe Paraskevopoulou. Machine-generated, machine-checked proofs for a verified compiler (experience report). *Proc. ACM Program. Lang.*, 10(ICFP):807–826, 2026. doi: 10.1145/3828700. URL https://doi.org/10.1145/3828700.

[22] Henk Barendregt and Freek Wiedijk. The challenge of computer mathematics. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363 (1835):2351–2375, 2005. doi: 10.1098/rsta.2005.1650.

[23] Kaiyu Yang, Aidan Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan J Prenger, and Animashree Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Advances in Neural Information Processing Systems 36*, NeurIPS 2023, pages 21573–21612. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2023. doi: 10.52202/075280-0944. URL http://dx.doi.org/10.52202/075280-0944.

[24] Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing theorem proving in LLMs through large-scale synthetic data. *CoRR*, abs/2405.14333, 2024. doi: 10.48550/ARXIV.2405.14333. URL https://doi.org/10.48550/arXiv.2405.14333.

[25] Yong Lin, Shange Tang, Bohan Lyu, Ziran Yang, Jui-Hui Chung, Haoyu Zhao, Lai Jiang, Yihan Geng, Jiawei Ge, Jingruo Sun, Jiayun Wu, Jiri Gesi, Ximing Lu, David Acuna, Kaiyu Yang, Hongzhou Lin, Yejin Choi, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover-V2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *CoRR*, abs/2508.03613, 2025. doi: 10.48550/ARXIV.2508.03613. URL https://doi.org/10.48550/arXiv.2508.03613.

[26] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz, Vivek Veeriah, Lei Yu, Oliver Nash, Paul Lezeau, Salvatore Mercuri, Calle Sönne, Bhavik Mehta, Alex Davies, Daniel Zheng, Fabian Pedregosa, Yin Li, Ingrid von Glehn, Mark Rowland, Samuel Albanie, Ameya Velingker, Simon Schmitt, Edward Lockhart, Edward Hughes, Henryk Michalewski, Nicolas Sonnerat, Demis Hassabis, Pushmeet Kohli, and David Silver. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, November 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09833-y. URL http://dx.doi.org/10.1038/s41586-025-09833-y.

[27] Lichen Zhang, Shuai Lu, and Nan Duan. Selene: Pioneering automated proof in software verification. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1776–1789. Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.98. URL http://dx.doi.org/10.18653/v1/2024.acl-long.98.

[28] Jianyu Zhang, Fuyuan Zhang, Jiayi Lu, Jilin Hu, Xiaoyi Yin, Long Zhang, Feng Yang, and Yongwang Zhao. Towards real-world industrial-scale verification: LLM-driven theorem proving on seL4. *CoRR*, abs/2602.08384, 2026. doi: 10.48550/ARXIV.2602.08384. URL https://doi.org/10.48550/arXiv.2602.08384.

[29] Haoxin Tu, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. Agentic verification of software systems. *Proc. ACM Softw. Eng.*, 3(FSE):3558–3581, 2026. doi: 10.1145/3808164. URL https://doi.org/10.1145/3808164.

[30] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards AI-assisted synthesis of verified Dafny methods. *Proc. ACM Softw. Eng.*, 1(FSE):812–835, 2024. doi: 10.1145/3643763. URL https://doi.org/10.1145/3643763.

[31] Chuyue Sun, Ying Sheng, Oded Padon, and Clark W. Barrett. Clover: Closed-loop verifiable code generation. In Guy Avni, Mirco Giacobbe, Taylor T. Johnson, Guy Katz, Anna Lukina, Nina Narodytska, and Christian Schilling, editors, *AI Verification - First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22-23, 2024, Proceedings*, Lecture Notes in Computer Science, pages 134–155. Springer, 2024. doi: 10.1007/978-3-031-65112-0_7. URL https://doi.org/10.1007/978-3-031-65112-0_7.

[32] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu K. Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. AutoVerus: Automated proof generation for Rust code. *Proc. ACM Program. Lang.*, 9(OOPSLA2):3454–3482, 2025. doi: 10.1145/3763174. URL https://doi.org/10.1145/3763174.

[33] Saikat Chakraborty, Gabriel Ebner, Siddharth Bhat, Sarah Fakhoury, Sakina Fatima, Shuvendu Lahiri, and Nikhil Swamy. Towards neural synthesis for SMT-assisted proof-oriented programming. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 1755–1767. IEEE, April 2025. doi: 10.1109/icse55347.2025.00002. URL http://dx.doi.org/10.1109/ICSE55347.2025.00002.

[34] Merlijn Sevenhuijsen, Khashayar Etemadi, and Mattias Nyberg. VeCoGen: Automating generation of formally verified C code with large language models. In *13th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE@ICSE 2025, Ottawa, ON, Canada, April 27-28, 2025*, pages 101–112. IEEE, 2025. doi: 10.1109/FORMALISE66629.2025.00017. URL https://doi.org/10.1109/FormaliSE66629.2025.00017.

[35] Xiaohan Lin, Qingxing Cao, Yinya Huang, Haiming Wang, Jianqiao Lu, Zhengying Liu, Linqi Song, and Xiaodan Liang. FVEL: Interactive formal verification environment with large language models via theorem proving. In *Advances in Neural Information Processing Systems 37*, NeurIPS 2024, pages 54932–54946. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2024. doi: 10.52202/079017-1742. URL http://dx.doi.org/10.52202/079017-1742.

[36] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Advances in Neural Information Processing Systems 35*, NeurIPS 2022, pages 32353–32368. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2022. doi: 10.52202/068431-2344. URL http://dx.doi.org/10.52202/068431-2344.

[37] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proc. ACM Softw. Eng.*, 1(FSE):1889–1912, 2024. doi: 10.1145/3660791. URL https://doi.org/10.1145/3660791.

[38] Shuvendu K. Lahiri. Evaluating LLM-driven user-intent formalization for verification-aware languages. In Nina Narodytska and Philipp Rümmer, editors, *Formal Methods in Computer-Aided Design, FMCAD 2024, Prague, Czech Republic, October 15-18, 2024*, pages 142–147. IEEE, 2024. doi: 10.34727/2024/ISBN.978-3-85448-065-5_19. URL https://doi.org/10.34727/2024/isbn.978-3-85448-065-5_19.

[39] Matthias Cosler, Christopher Hahn, Daniel Mendoza, Frederik Schmitt, and Caroline Trippel. nl2spec: Interactively translating unstructured natural language to temporal logics with large language models. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, Lecture Notes in Computer Science, pages 383–396. Springer, 2023. doi: 10.1007/978-3-031-37703-7_18. URL https://doi.org/10.1007/978-3-031-37703-7_18.

[40] Zhi Ma, Cheng Wen, Zhexin Su, Xiao Liang, Cong Tian, Shengchao Qin, and Mengfei Yang. Bridging natural language and formal specification–automated translation of software requirements to LTL via hierarchical semantics decomposition using LLMs. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1208–1220. IEEE, 2025. doi: 10.1109/ASE63991.2025.00104. URL https://doi.org/10.1109/ASE63991.2025.00104.

[41] Zhi Ma, Xiao Liang, Cheng Wen, Rui Chen, Bin Gu, Shengchao Qin, Cong Tian, and Mengfei Yang. Automated ltl specification generation from industrial aerospace requirements. In Augusto Sampaio and Marielle Stoelinga, editors, *Formal Methods*, pages 670–690, Cham, 2026. Springer Nature Switzerland. ISBN 978-3-032-26220-2. doi: 10.1007/978-3-032-26220-2_33.

[42] Chenyang An, Sam Bayless, Stefano Buliani, Darion Cassel, Byron Cook, Duncan Clough, Rémi Delmas, Nafi Diallo, Ferhat Erata, Nick Feng, Dimitra Giannakopoulou, Aman Goel, Aditya Gokhale, Joe Hendrix, Victor Heorhiadi, Marc Hudak, Dejan Jovanovic, Andrew M. Kent, Benjamin Kiesl-Reiter, Jeffrey J. Kuna, Nadia Labai, Joseph Lilien, Divya Raghunathan, Zvonimir Rakamaric, Niloofar Razavi, Michael Tautschnig, Ali Torkamani, Nathaniel Weir, Michael W. Whalen, and Jianan Yao. A neurosymbolic approach to natural language formalization and verification. In Eva Darulova, Anthony W. Lin, and Philipp Rümmer, editors, *Computer Aided Verification - 38th International Conference, CAV 2026, Lisbon, Portugal, July 26-29, 2026, Proceedings, Part II*, volume 16683 of *Lecture Notes in Computer Science*, pages 601–617. Springer, 2026. doi: 10.1007/978-3-032-32526-6_28. URL https://doi.org/10.1007/978-3-032-32526-6_28.

[43] Colin S. Gordon and Sergey Matskevich. Trustworthy formal natural language specifications. In Tijs van der Storm and Robert Hirschfeld, editors, *Proceedings of the 2023 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2023, Cascais, Portugal, October 25-27, 2023*, pages 50–70. ACM, 2023. doi: 10.1145/3622758.3622890. URL https://doi.org/10.1145/3622758.3622890.

[44] Kiro. Feature Specs. Kiro documentation, Amazon Web Services, https://kiro.dev/docs/specs/feature-specs/, 2026. Page updated 4 August 2026; accessed 11 September 2026.

[45] GitHub. Spec Kit. https://github.com/github/spec-kit, 2025. Accessed 10 September 2026.

[46] Shuzhao Feng, Boqi Chen, Brett H. Meyer, and Gunter Mussbacher. LLM-assisted repository-level generation with structured spec-driven engineering. In *Proceedings of the 34th ACM International Conference on the Foundations of Software Engineering, FSE Companion 2026, Concordia University, Montreal, QC, Canada, July 5-9, 2026*, pages 1257–1261. ACM, 2026. ISBN 979-8-4007-2636-1. doi: 10.1145/3803437.3805567. URL https://doi.org/10.1145/3803437.3805567.

[47] Martin Monperrus. Bootstrapping coding agents: The specification is the program. *IEEE Softw.*, 43(4):19–22, 2026. doi: 10.1109/MS.2026.3687051. URL https://doi.org/10.1109/MS.2026.3687051.

[48] Atlas Computing. formal-specification-ide. https://github.com/atlas-computing-org/formal-specification-ide, 2025. Accessed 10 September 2026.