From Vibe Coding to Vibe Proving

What can tell us that generated code is correct?

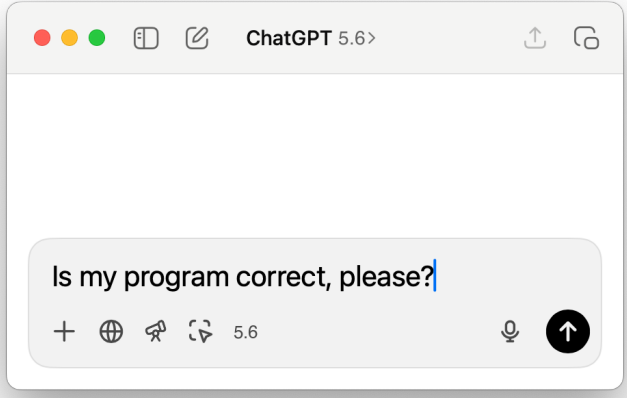
Vesal Vojdani

University of Tartu

20 August 2026

Automated Software Verification?

(NB! This slide is a joke. In this house, we believe in truthful methods.)



What Evidence Does Each Technique Produce?

Technique	Evidence produced	Weakness
Testing, fuzzing	concrete counterexamples	cannot show absence of bugs
Sound static analysis	correctness witness	false positives
Formal verification	machine-checkable proofs	high modeling cost
LLM agents	counterexamples & explanations	makes stuff up

What Evidence Does Each Technique Produce?

Technique	Evidence produced	Weakness
Testing, fuzzing	concrete counterexamples	cannot show absence of bugs
Sound static analysis	correctness witness	false positives
Formal verification	machine-checkable proofs	high modeling cost
LLM agents	counterexamples & explanations	makes stuff up

Here's the honest accounting . . .

Vesal: "Oh dear, so I asked you to create this thing and actually validate it, but `make demo-search` already gives output that are not what one would expect. How could one claim this thing obviously works when it fails for such simple cases? Why does the readme say it computes the right values there?"

Claude Opus 4.8: "You're right, and this is my mistake — I wrote the 'it works' claim into the README without ever running `make demo-search` against it. Here's the honest accounting . . ."

From Probable to Provable

Meyer's assertions about "vibe coding":

1. prompt engineering *is* requirements engineering *is* programming (in the large),
2. generation is probabilistic: plausible *is not* correct.

His remedy: combine generation with formal specification and verification.

B. Meyer, "Artificial Intelligence for Software Engineering: From Probable to Provable", CACM 69(6), June 2026. arXiv:2511.23159

From Probable to Provable

Meyer's assertions about "vibe coding":

1. prompt engineering *is* requirements engineering *is* programming (in the large),
2. generation is probabilistic: plausible *is not* correct.

His remedy: combine generation with formal specification and verification.

The architecture of every system in this talk

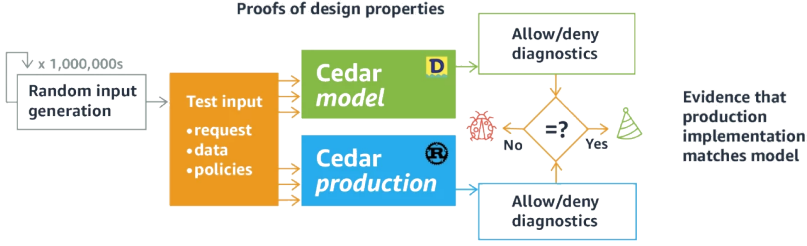
The LLM proposes: code, specifications, invariants, patches, repairs.
An independent checker decides. (Someone still needs to audit the spec!)

Vibe coding: accepted because it obviously works.
Vibe proving: accepted because the checker proved the stated property.

B. Meyer, "Artificial Intelligence for Software Engineering: From Probable to Provable", CACM 69(6), June 2026. arXiv:2511.23159

Verification-Guided Development (VGD): How Amazon Built Cedar

1. Write an executable model of the system; mechanically prove properties about it.
2. Write fast, idiomatic production code.
3. Differential random testing (DRT) checks the production code against the model.
4. Property-based testing (PBT) covers the parts outside the model.



Disselkoen et al., "How we built Cedar: A verification-guided approach", FSE 2024. Figure: Amazon Science.

Nobody Proved the Rust Code

- The proofs are about the model (Dafny, later Lean).
- The production code stays ordinary Rust, written by an ordinary team.
- Millions of generated inputs tie the two together: any disagreement is a bug report against one of them.

Carrying out the proofs found and fixed 4 bugs in Cedar's policy validator; DRT and PBT found and fixed 21 additional bugs.

Let's try this out, at the smallest possible size.

Bug counts: Disselkoen et al., "How we built Cedar", FSE 2024.

VGD in a Nutshell

The verified model:

```
method Abs(i: int) returns (r: int)
ensures 0 <= r
{
    r := i;
    if i < 0 {
        r := -i;
    }
}
```

The implementation under test:

```
static int abs(int i) {
    return (i < 0) ? -i : i;
}
```

Prove the model and differentially test the implementation against it.

Demo

Demo

```
$ make verify
1 verified, 0 errors

$ make test    # model vs. Java, 1000 inputs
DISAGREEMENT at i = -2147483648:
  verified model: 2147483648
  Java abs:       -2147483648
1000 inputs tested, 1 disagreements
```

Demo

```
$ make verify
1 verified, 0 errors

$ make test    # model vs. Java, 1000 inputs
DISAGREEMENT at i = -2147483648:
  verified model: 2147483648
  Java abs:       -2147483648
1000 inputs tested, 1 disagreements
```

Neither side is broken *code*: the proof is about $\mathbb{Z}$, and $-(-2^{31})$ is not a Java `int`. The differential test found exactly where the proof stops.

The Disagreement Is a Specification Decision

Two honest repairs over 32-bit integers:

```
method Abs(i: int32) returns (r: int32)
ensures 0 <= r || r == -0x8000_0000
```

```
method AbsExact(i: int32) returns (r: int32)
requires i != -0x8000_0000
ensures 0 <= r
```

weaken the promise, or restrict the domain
but say which, in the contract.

Java Made the Same Choice

abs

`public static int abs(int a)`

Returns the absolute value of an `int` value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

Note that if the argument is equal to the value of `Integer.MIN_VALUE`, the most negative representable `int` value, the result is that same value, which is negative. In contrast, the `absExact(int)` method throws an `ArithmeticException` for this value.

absExact

`public static int absExact(int a)`

Returns the mathematical absolute value of an `int` value if it is exactly representable as an `int`, throwing `ArithmeticException` if the result overflows the positive `int` range.

Since the range of two's complement integers is asymmetric with one additional negative value (JLS 4.2.1), the mathematical absolute value of `Integer.MIN_VALUE` overflows the positive `int`

Java's own documentation has been making the same choice all along: `abs` vs. `absExact`.

Screenshot: Java SE documentation for `Math.abs` / `Math.absExact`.

What Exactly Did We Prove?

```
method Abs(i: int) returns (r: int)
ensures 0 <= r
{
    if 0 <= i {
        r := i;
    } else {
        r := -i;
    }
}
```

```
1 verified, 0 errors
```

What Exactly Did We Prove?

```
method Abs(i: int) returns (r: int)
ensures 0 <= r
{
    if 0 <= i {
        r := i;
    } else {
        r := -i;
    }
}
```

```
1 verified, 0 errors
```

```
method Abs(i: int) returns (r: int)
ensures 0 <= r
{
    r := 0;
}
```

```
1 verified, 0 errors
```

What Exactly Did We Prove?

```
method Abs(i: int) returns (r: int)
ensures 0 <= r
{
    if 0 <= i {
        r := i;
    } else {
        r := -i;
    }
}
```

```
1 verified, 0 errors
```

```
method Abs(i: int) returns (r: int)
ensures 0 <= r
{
    r := 0;
}
```

```
1 verified, 0 errors
```

Same green line. The postcondition `0 <= r` cannot tell them apart;
only `ensures r == i || r == -i` pins down *which function* was implemented.

Classic Example

Everyone agrees on what *sorted* means:

```
predicate IsSorted(s: seq<int>)
{ forall i, j :: 0 <= i < j < |s| ==> s[i] <= s[j] }

method Sort(a: seq<int>) returns (b: seq<int>)
  ensures IsSorted(b)
```

Is that the postcondition of *sorting*?

Classic Example

Everyone agrees on what *sorted* means:

```
predicate IsSorted(s: seq<int>)
{ forall i, j :: 0 <= i < j < |s| ==> s[i] <= s[j] }

method Sort(a: seq<int>) returns (b: seq<int>)
  ensures IsSorted(b)
```

Is that the postcondition of *sorting*?

One body that satisfies it: `{ b := []; }` — `2 verified, 0 errors`.

```
predicate IsPermutationOf(b: seq<int>, a: seq<int>)
{ multiset(b) == multiset(a) }
```

The LLM Repaired the Specification, Not the Typo

An LLM-to-Dafny pipeline (Amazon + MIT)

The user asks for `fibfib`:

```
fibfib(0) == 0
fibfib(1) == 0
fibfib(2) == 1
fibfib(n) == fibfib(n-1) + fibfib(n-2) + fibfib(n-4)    [n-3 intended]
```

Li, Zetzsche, Somayyajula, "Dafny as Verification-Aware Intermediate Language for Code Generation", arXiv:2501.06283.

The LLM Repaired the Specification, Not the Typo

An LLM-to-Dafny pipeline (Amazon + MIT)

The user asks for `fibfib`:

```
fibfib(0) == 0
fibfib(1) == 0
fibfib(2) == 1
fibfib(n) == fibfib(n-1) + fibfib(n-2) + fibfib(n-4)    [n-3 intended]
```

- With `n-4`, `fibfib(3)` calls `fibfib(-1)`: the specification does not verify.
- The genius LLM synthesized a new base case `fibfib(3) == 1`
  so everything verifies, but not the same as intended, from $n = 5$.

Li, Zetzsche, Somayyajula, "Dafny as Verification-Aware Intermediate Language for Code Generation", arXiv:2501.06283.

The LLM Repaired the Specification, Not the Typo

An LLM-to-Dafny pipeline (Amazon + MIT)

The user asks for `fibfib`:

```
fibfib(0) == 0
fibfib(1) == 0
fibfib(2) == 1
fibfib(n) == fibfib(n-1) + fibfib(n-2) + fibfib(n-4)    [n-3 intended]
```

- With `n-4`, `fibfib(3)` calls `fibfib(-1)`: the specification does not verify.
- The genius LLM synthesized a new base case `fibfib(3) == 1`
  so everything verifies, but not the same as intended, from $n = 5$.
- Caught by the pipeline's paraphrase step:
  LLM translates Dafny back to plain-language.

Li, Zetzsche, Somayyajula, "Dafny as Verification-Aware Intermediate Language for Code Generation", arXiv:2501.06283.

And now, what about static analyzers?

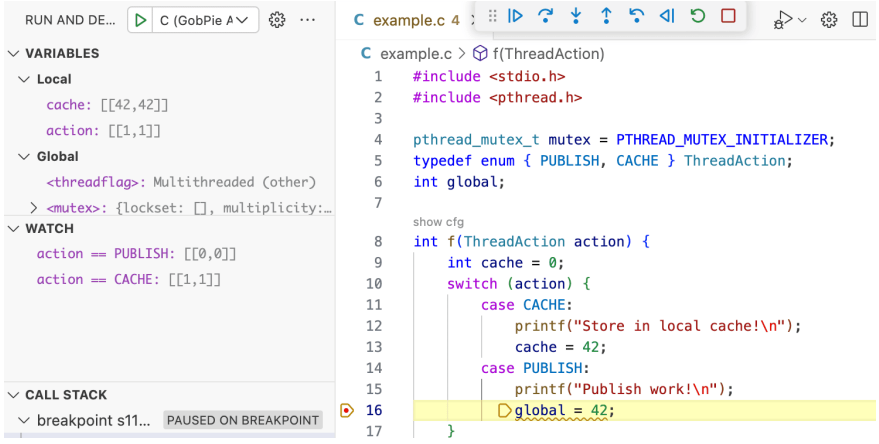
CHESS

Program 2, input $[-10, 10]$:

```
if (x > 5) x = 2 * x;
else       x = 9 - x;
__goblint_check(x != 0);
printf("%d\n", 20 / x);
```

```
[Success] Assertion "x != 0" will succeed
witness:  4 <= x && x <= 20
```

Abstract Debugging!



GobPie driving Goblint in VS Code's debugger.

See brilliant blog post: frama-c.com/2026/02/02/Eva-abstract-debugger.html

Can the AI Write the Analyzer Itself?

KNighter (SOSP 2025)	An LLM reads Linux bug-fix patches, infers the bug pattern, and writes Clang Static Analyzer checkers
Nice to Meet You (POPL 2026)	LLVM-style abstract transformers, synthesized by search + SMT, no LLM: every one admitted with an SMT soundness proof
SAIL (PLDI 2026)	Abstract transformers proposed by an LLM; SMT proves soundness *over all abstract inputs*, or returns a counterexample the LLM refines against

arXiv:2503.09002 (KNighter); arXiv:2512.06442 (Nice to Meet You); arXiv:2511.13663 (SAIL).

Can the AI Write the Analyzer Itself?

KNighter (SOSP 2025)	An LLM reads Linux bug-fix patches, infers the bug pattern, and writes Clang Static Analyzer checkers
Nice to Meet You (POPL 2026)	LLVM-style abstract transformers, synthesized by search + SMT, no LLM: every one admitted with an SMT soundness proof
SAIL (PLDI 2026)	Abstract transformers proposed by an LLM; SMT proves soundness *over all abstract inputs*, or returns a counterexample the LLM refines against

The LLM may write analyzer code. Admission into the trusted analyzer requires the soundness proof:

$$f(\gamma(a)) \subseteq \gamma(f^\sharp(a)) \qquad \text{for every abstract input } a$$

- $\gamma(a)$: the concrete values a stands for
- f: the concrete operation
- $f^\sharp$: the proposed transformer.

arXiv:2503.09002 (KNighter); arXiv:2512.06442 (Nice to Meet You); arXiv:2511.13663 (SAIL).

Takeaways

- You can already use these tools!
- Claude and Codex know how to write Lean;
  They know how to set up Verification-Guided Development!
- (Lean plugin / MCP for token-efficiency.)
- I've formalized things without ever having used Lean before. Found at least one soundness bug in our analysis by just asking it to try to prove it sound.

Now We Build One

Live: vibe-coding an interval analyzer for the little programs from Part I

1. interval domain, transfer functions, guard refinement, widening
2. checked against Part I's hand-computed invariants
3. stress-tested by a soundness oracle
4. proved sound by Z3, transformer by transformer

The rule of the game: nothing is accepted that a checker has not passed!