A Review on "AutoVerus: Automated Proof Generation for Rust Code"

Gijung Im

FCAI Lab
Yonsei University

Jan 26, 2026

- **Title:** AutoVerus: Automated Proof Generation for Rust Code
- **Source:** OOPSLA 2025
- **Authors:** Microsoft Research (Weidong Cui, et al.), UIUC, and Columbia Univ.
- **Presenter:** Gijung Im

Table of Contents

Motivation

1. Generative AI in Software Engineering:
   - LLMs excel at code generation (e.g., GitHub Copilot)
   - but lag in generating formal proofs.
2. The Need for Correctness:
   - Developers often distrust AI-generated code.
3. Goal:

   $\rightsquigarrow$ To automatically generate correctness proofs for Rust code,
   allowing developers to enjoy both **productivity** (AI) and **realiablity** (Verification).

Verus: Verified Rust for Low-level Systems

- A static verification tool for Rust that uses an SMT solver (Z3).
- Allows developers to write specifications and proofs directly in Rust-like syntax.
- Key Philosophy:
    1. *Implementation and Proof in the same language.*
       No need to learn spearate languages like Rocq or Dafny.
    2. *Zero-overhead.*
       All “ghost code” (specifications/proofs) are erased at compile time,
       leaving only optimized Rust executables.

The Mode System (spec vs. proof vs. exec)

```
spec fn is_digit(c: u8) -> bool {
    c >= 48 && c <= 57
}

spec fn cnt_dig(seq: Seq<u8>) -> int
    decreases seq.len(),
{
    if seq.len() == 0 {
        0
    } else {
        cnt_dig(seq.drop_last()) +
        if is_digit(seq.last()) {
            1 as int
        } else {
            0 as int
        }
    }
}
```

(a) The spec functions.

```
fn count_digits(text: &Vec<u8>) -> (ret: usize)
    ensures ret == cnt_dig(text@),
{
    let mut count = 0;
    let mut i = 0;
    while i < text.len()
      invariant
        i <= text.len(),
        count <= i,
        count==cnt_dig(text@.subrange(0, i as int)),
    {
      if text[i] >= 48 && text[i] <= 57
         {count += 1;}
      i += 1;
      assert(text@.subrange(0, i - 1 as int)
        == text@.subrange(0, i as int).drop_last());
    }
    assert(text@ == text@.subrange(0, i as int));
    count
}
```

(b) The implementation.

Fig. 1. A Rust function, in gray background, with Verus annotations. The dark-yellow background highlights the proof annotation needed by Verus to prove the specifications highlighted in the light-yellow.

Challenges in Verus Proof Generation

1. *Data Scarcity.*
   - Verus is young (≤ 3 years old), #GitHub repos ≤ 10.
   - Unlike Rocq or Isabelle, there is no massive dataset for training.
2. *Syntax Subtleties.*
   - Verus introduces specific syntax extensions—e.g., the `@` operator, `int` vs. `u64`.
   - LLMs often confuse ***ghost code*** rules with standard Rust rules.
3. *Feedback Ambiguity.*
   - Unlike Interactive Theorem Provers (ITP) that show proof state, SMT solvers essentially return either "Pass" or "Fail".
   - It is difficult to measure progress when verification fails.

Workflow

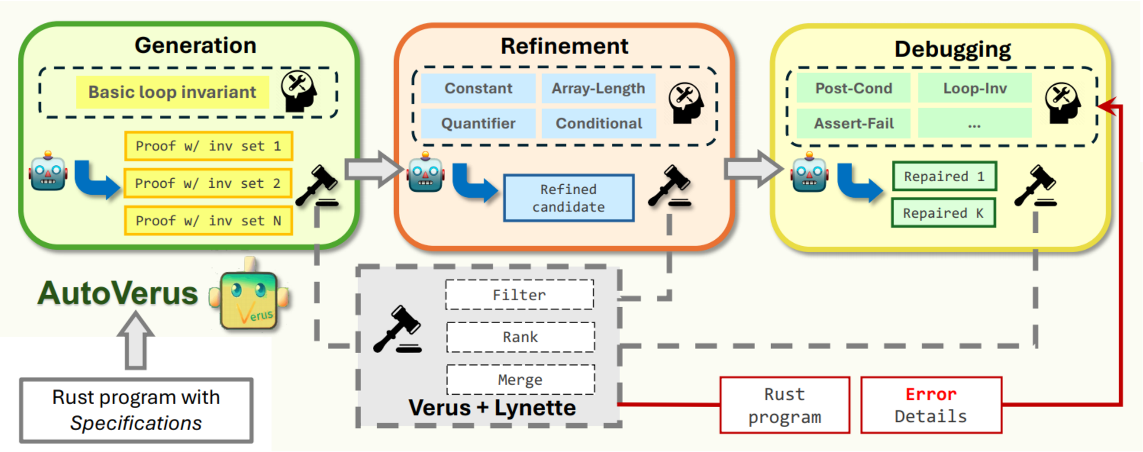

Fig. 2. The workflow of AutoVerus

Phase I–Generation (Preliminary Proof)

- **Objective:** Generate initial loop invariants.
- **Post-Processing.**

```
def is_correct(p):
    return verus_correct(p) or verus_correct(houdini(p))
    # The Houdini algorithm deletes the unprovable proof annotations from p.

def GenPrelimProof(program):
    pCanidates = LoopInvAgent(program) # proof candidates
    safeCandidates = { p ∈ pCanidates | is_safe(program, p) }
    Sort(safeCandidates, comparison=(V,E)-rank) # V: verified, E: erroneous
    mergeP = program # merged proof
    for p in safeCandidates:
        mergeP = merge(mergeP, p)
        if is_correct(mergeP):
            return mergeP
    return safeCandidates[0] # first element
```

Listing 1: Preliminary proof generation with post-processing

Example–Diffy Benchmark

```
invariant
  N > 0,
  i <= N as usize,
  sum.len() == 1,
  sum[0] <= N + i,
  forall|k: int|0<=k<N ==> b[k]==1,
  b.len() == N,
// (V, E) = (4, 2)
```

(a) Loop invariants of best-so-far P_B.

```
invariant
  N > 0,
  i <= N as usize,
  sum.len() == 1,
  sum[0] <= N + i,
  forall|k: int|0<=k<N ==> b[k]==1,
  N < 1000,
// (V, E) = (3, 2)
```

(b) Loop invariants of a lower-ranked P_i.

```
i = 0;

while (i < N as usize)
  invariant
  N > 0,
  i <= N as usize,
  sum.len() == 1,
  sum[0] <= N + i,
  forall|k: int|0<=k<N ==> b[k]==1,
  b.len() == N,
  N < 1000,
{
  sum.set(0, sum[0] + b[i]);
  i = i + 1;
}
// (V, E) = (5, 0)
// Verified
```

(c) The merged program P' which is *verified*.

Fig. 5. An example of a merged program for a problem from Diffy benchmark.

Phase II–Refinement (Generic Fixes)

- **Objective:** Correct common mistakes before consulting the verifier.
- **4 Specialized Agents.**
  - *Constant-Propagation:* Propagates preconditions into every loop.
  - *Array-Length:* Adds invariants about every array/container sizes (e.g., `len()`).
  - *Quantifier:* Checks whether quantifier-related invariants are used correctly.
  - *Conditional-Loop-Invariant:* Adjusts invariants for specific iterations of a loop.

```
def RefineProof(program):
    best = program
    # from the simplest (i.e., Constant-Propagation) to the more complex ones
    for refineAgent in refineAgentList:
        p = refineAgent(best)
        if is_correct(p):
            return p
        if accept_refine(p):
            best = p
    return best
```

Listing 2: Proof Refinement

Phase III–Debugging (Error-Driven)

- **Objective:** Iteratively fix verification errors reported by Verus.

```
def DebugProof(program):
    for _ in range(MaxIter):
        if is_correct(program):
            return program
        errors = verus_compile(program)
        e = select_error(errors)
        repair_agent = select_repair_agent(e)
        repairedCandidates = repair_agent(program, e)
        for p in repairedCandidates:
            if is_correct(p):
                return p
            if accept_repair(p):
                program = p
                break
    return program
```

Listing 3: Debug Proof Generation

- repair_precond_error
- repair_precond_veclen
- repair_postcond_error
- repair_invfail_front
- repair_invfail_end
- repair_assertion_error
- repair_arithmetic_flow
- repair_mismatch_type
- repair_assertion_error_with_proof_func
- repair_default

Fig. 6. AutoVerus repair agent list

Fibonacci Sequence (Iteration 1 & 2)

```
fn fibonacci(n: usize) -> (ret: Vec<i32>)
  requires
    fibo_fits_i32(n as int),
    n >= 2,
  ensures
    forall|i: int|2 <= i < n ==>
        #[trigger] ret@[i] == fibo(i),
    ret@.len() == n,
{
  let mut fib = Vec::new();
  fib.push(0);
  fib.push(1);
  let mut i = 2;
+1// Assert the invariant right before the loop
+1 assert(fib.len() == 2);
  while i < n
    invariant
      0 <= 2 <= i <= n,
      fib.len() == i,
      fibo_fits_i32(n as int),
      forall|j: int| 0 <= j < i ==>
        # [trigger] fib[j] == fibo(j),
-1  fib.len() == n;
  {
+2// Assert the addition won't overflow an i32
+2 assert(fib[i-1]as int + fib[i-2]as int
+2          < 0x8000_0000) by {
+2  assert(fib[i - 1] == fibo(i - 1));
+2  assert(fib[i - 2] == fibo(i - 2));
+2  assert(fibo(i-1)+fibo(i-2)==fibo(i));
+2  lemma_fibo_monotonic(i, n);
+2 };
    let next_fib = fib[i - 1] + fib[i - 2];
    fib.push(next_fib);
    i += 1;
  }
  fib
}
+2 proof fn lemma_fibo_monotonic(..) {..}
```

(a) Repair the “the loop invariant not satisfied before loop” and “arithmetic overflow” errors.

Fibonacci Sequence (Iteration 3)

```
spec fn fibo(n: int) -> nat
    decreases n,
{
    if n <= 0 { 0 }
    else if n == 1 { 1 }
    else { fibo(n - 2) + fibo(n - 1) }
}
spec fn fibo_fits_i32(n: int) -> bool {
    fibo(n) < 0x8000_0000
}
```

(b) The spec functions.

```
proof fn lemma_fibo_monotonic(n:int, m:int)
  requires n <= m,
  ensures fibo(n) <= fibo(m),
  decreases m - n
{
  if n < m {
    lemma_fibo_monotonic(n, m - 1);
+   assert(fibo(n) <= fibo(m - 1));
+   assert(fibo(m - 1) <= fibo(m));
  }
}
```

(c) Repair the post-condition not satisfied error.

Fibonacci Sequence (Iteration 4 & 5)

```
proof fn lemma_fibo_monotonic(n:int, m:int)
  ...
{
  if n < m {
    lemma_fibo_monotonic(n, m - 1);
    assert(fibo(n) <= fibo(m - 1));
-   assert(fibo(m - 1) <= fibo(m));
+   if m > 1 { // Avoiding the case m == 1 to
     prevent negative indexing in fibo(m-2)
+     assert(fibo(m-2)+fibo(m-1)==fibo(m));
+     assert(fibo(m - 1) <= fibo(m));
+   }
  }
}
```

(d) Repair the assertion failure error.

Verus-Bench

- A curated suit of $150 (= 78 + 38 + 11 + 23)$ non-trivial proof tasks.
  - *MBPP:* 78 tasks (Python → Rust)
  - *Diffy:* 38 tasks (C benchmarks)
  - *CloverBench:* 11 tasks
  - *Misc:* 23 tasks (Algorithms)
- Excludes trivial tasks that Verus can verify without annotations.
- *Comparison Baseline:* GPT-4o with sophisticated prompting (repeated invocations).

Table 1. Summary of Verus-Bench

Benchmark Sources	CloverBench	Diffy	MBPP	Misc	Total
# of Proof Tasks	11	38	78	23	150
Executable LOC	175	951	1,333	390	2,849
Specification LOC	80	265	700	207	1,252

Results

- **Success Rate:** AutoVerus $= 91.3\%$ vs. Baseline $= 44.7\%$.
- **Efficiency:**
  - Solved $> 50\%$ of tasks within 30[sec] or 3 LLM calls.
  - Baseline required significantly more time and calls.
- **Cost:** Total cost for the entire benchmark was only $\leq \$37$.

Summar & Significance

Summary.

- *Synergy of Techniques:* Combines <u>LLM creativity</u> with <u>formal rigor</u>.
- *Workflow Mimicry:* Imitated the “Draft-Refine-Debug” loop used by human experts.
- *Overcoming Data Scarcity:* Demonstrated that <u>In-Context Learning + Static Analysis</u> can solve problems in few resouces languages without fine-tuning.

Significance.

- *Democratizing Verification:* Lowers the barrier to entry for verifying Rust systems.
- *Practicality:* Works on standard hardware with low cost and high speed.

Future Work

Future Work.

1. Scaling to large-scale system verification with complex dependencies.
2. Automating specification generation.

Thank you for listening!