CS 560 Project Report: Using Large Language Models to Formally Verify Correctness of Rust Code

Vidush Singhal, Bishal Basak Papan, Jack Roscoe, Faaiz Memon

December 2025

1 Introduction

Formal verification is the process of verifying correctness properties of code via specifications which describe some properties about programs. For instance, in Dafny, we can use ensures and requires constraints to highlight properties about the input to the program and the return value of the program. We can also try to reason about invariants for loops and iterative constructs. These are crucial for proving correctness properties of the program. There has been a plethora of work using large language models to infer specifications for programs. In this work, we focus on AutoVerus which infers invariants for programs when the user gives the Verus program with some requires and ensures constraints.

Formal verification overall is a tough process. Programmers cannot reason about the correctness of the program formally in many cases. They also need to understand the formal verification language and there is usually a steep learning curve when it comes to learning these languages. We aim to make the burden of verifying these programs lesser on the programmer.

Existing research works like AutoVerus [3] allow the users to provide method contracts in the form of preconditions and postconditions with the Verus codes to generate correct specifications using LLMs. Their evaluation results show the effectiveness of LLMs to infer correct loop invariants and specification functions given that the users have provided correct method contracts. However, the main challenge to use AutoVerus is that these method contracts are not automatically generated, rather human provided. Writing correct preconditions and postconditions is itself a difficult task and without the complete method contracts, AutoVerus cannot generate a complete specification to verify most of the Verus functions.

To resolve this limitation and taking off the burden of writing the correct precondition and postcondition, we want to investigate if we can improve AutoVerus by giving a plain Rust code to the LLM along with more reasoning contexts obtained from running symbolic execution tools on that code.

2 Overview

Figure 1 shows the design of our system. Instead of starting with the Verus code, we modified AutoVerus to use the Rust code directly. On a high level, from the Rust code we prompt the LLM to generate Verus code using a one-shot learning approach. We then follow the standard pipleline of AutoVerus to take the Verus code and generate the verifiable Verus code with all the right specifications.

In addition, we wanted to include symbolic execution tools in our pipeline in order to provide the LLM with more context regarding the programs. For instance, some symbolic execution tools can provide the LLM with crucial static analysis to help it generate better specifications. We used two

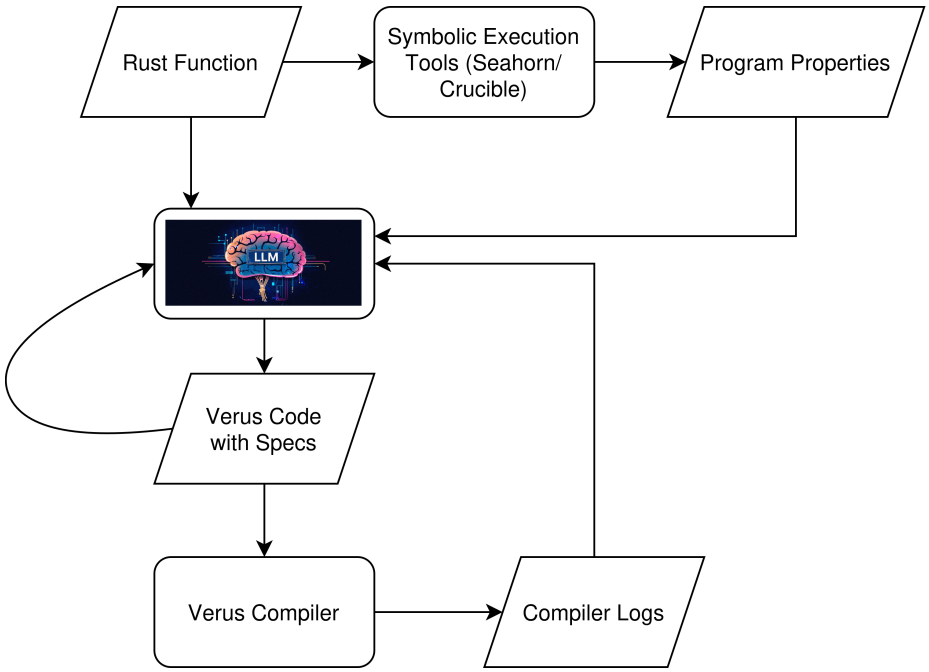

Figure 1: Design of Our System

static analysis tools in our pipeline: Seahorn [2] and Crucible [1]. Seahorn takes as input a Rust file and generates Horn constraints. These Horn constraints represent SMT properties of the program. The SMT properties can be solved via a solver and provide satisfiability or not of the program. Fig. 2 shows an example of the output generated by running SeaHorn on a simple Rust function. As shown in Fig. 2b, the output contains SMT queries in an IR (Intermediate Representation) format.

When prompting the LLM for Verus code with specifications, we also included the Horn constraints for the Rust program. We send the horn constraints as a string in the hope that it will be able to infer some of the specifications automatically.

2.1 Counterexample Generation

Another feature we added to AutoVerus was counterexample generation. While we were learning Verus, we often noticed it was difficult to tell exactly why an invariant or ensures clause we wrote was failing. Verus only shows what clause has an issue, but it does not tell you what makes it fail. In other words, Verus did not give concrete values that caused verification to fail. It may not always be the case the Verus errors have a concrete counterexample (for example, some errors are due to not introducing terms into the prover state, or other things of that nature), but many faulty clauses likely do have counterexamples. We reasoned that if counterexamples are useful for a human programmer, they may also be useful for an LLM. Our focus was on generating counterexample arguments for individual functions. 3b shows an example Verus function and the generated counterexamples.

In addition to integration with Symbolic Execution tools, we have also kept the ability for the users to provide some specifications (preconditions and postconditions) as comments in the Rust codes. We expect that these annotations might help the LLM to reason better about the function to verify.

We also integrated the compiler output from Verus, such as the errors obtained when the input Verus program could not be verified. We believe this will help the LLM to understand exactly what it needs and what is missing in the Verus program so that it can generate correct specifications for the

```
fn countdown(n: i32)
     -> i32 {
    let mut steps =
        0;
    let mut x = n;

    while x > 0 {
        x -= 1;
        steps += 1;
    }

    steps
}

fn main(n: i32) ->
    i32 {
    let _ = countdown
        (n);
    0
}
```

(a) Example Rust code for `countdown`.

```
(rule (main@start main@%n_0))
(rule (=> (and (main@start main@%n_0)
        true
        (= main@%_0_0 main@%loop.bound_0)
        (= main@%_1_0 (= main@%_0_0 0))
        main@%_1_0)
    (main@empty.loop main@%_0_0 main@%n_0)))
(rule (=> (and (main@bb1.i main@%x.0.i_0 main@%steps.0.i_0 main@
    %_0_0)
        true
        (= main@%_3.i_0 (> main@%x.0.i_0 main@%_0_0))
        main@%_3.i_0)
    ...
(rule (=> main@countdown.exit main@bb1))
(rule (=> (and (main@empty.loop main@%_0_0 main@%n_0) true main@
    %nd.loop.cond_0)
    (main@empty.loop.body main@%_0_0 main@%n_0)))
(rule (=> (and (main@empty.loop main@%_0_0 main@%n_0)
        true
        (not main@%nd.loop.cond_0)
        (= main@%steps.0.i_0 0)
        (= main@%x.0.i_0 main@%n_0))
    (main@bb1.i main@%x.0.i_0 main@%steps.0.i_0 main@%_0_0)))
(rule (=> (main@empty.loop.body main@%_0_0 main@%n_0)
    (main@empty.loop main@%_0_0 main@%n_0)))
(query main@bb1)
```

(b) SeaHorn-generated rules, trimmed some constraints.

Figure 2: (a) Rust source and (b) corresponding SeaHorn SMT-LIB2 encoding for `countdown`.

```
fn sum(n: u8, mut input: &[u8]) -> (r: Vec<u8>)
ensures forall |i| 0 <=i< r.len() ==> r[i] == 69 + n
{
    let mut output = Vec::new();
    let mut i = 0;
    while i < n
        invariant i > n
    {
        i += 1;
    }

    if input.len() == 2 && input[0] == 67 {
        output.push(70 + n);
    } else {
        output.push(69 + n);
    }
    output
}
```

(a) Example Verus code for which we want a counterexample.

```
Counterexamples:
Counterexample for clause: ‘i >
     n‘ on line 13, column 15
arg0 = 128
arg1 = []

Counterexample for clause: ‘
    forall |i| 0 <= i < r.len()
     ==> r[i] == 69 + n‘ on
    line 7, column 9
arg0 = 0
arg1 = [67, 0]
```

(b) Generated Counterexample.

Figure 3: (a) Verus source with invalid clauses and (b) corresponding counterexamples fed into LLM.

```rust
#![no_std]
#![no_main]
#![feature(lang_items)]
#![feature(start)]

use core::panic::PanicInfo;

#[panic_handler]
fn panic(_info: &PanicInfo) -> ! {
    loop {}
}
#[lang = "eh_personality"]
extern "C" fn eh_personality() {}

// requires(n >= 0)
// ensures(result == n)
#[no_mangle]
pub extern "C" fn countdown(n: i32)->i32 {
    let mut steps = 0;
    let mut x = n;
    while x > 0 {
        x -= 1;
        steps += 1;
    }
    steps
}
#[no_mangle]
pub extern "C" fn main(n: i32) -> i32 {
    let _ = countdown(n);
    0
}
```

(a) Input: Seahorn Compatible Rust code.

```rust
use vstd::prelude::*;

verus! {

#[verifier::loop_isolation(false)]

fn countdown(n: i32) -> (r: i32)
    requires
        n >= 0,
    ensures
        r == n,
{
    let mut steps = 0;
    let mut x = n;

    while x > 0
        invariant
            x >= 0,
            steps + x == n,
    {
        x -= 1;
        steps += 1;
    }
    steps
}

} // verus!

fn main() {}
// Score: (1, 0)
// Safe: None
```

(b) Output: Verus code with specifications.

Figure 4: Example of the pipeline (a) A simple Rust code in Seahorn compatible format and (b) Generated Verus output through the pipeline for `countdown`, the (x, y) format output represents x function(s) are verified, y functions faced error

.

program. Whenever AutoVerus prompts the LLM and gets back a new Verus program, we compile the generated program with Verus, if it gets verified we exit, however, if we do get verification errors, we store the errors as a string and send it to the LLM in the next prompt, effectively integrating the errors.

Using the pipeline, we can generate a syntactically correct Verus program, as shown in Fig. 4. However, the specifications generated are not correct in most of the cases.

3 Implementation

Our implementation is done on top of AutoVerus. We used OpenAI API 4.0 as AutoVerus to evaluate the modifications that we implemented. For Seahorn integration, from the user provided annotated Rust code, we first converted it to an IR as a bitcode format with the debug related output stripped, the generated the Horn constraints by running seahorn on this IR. In this section, we mainly discuss

how we integrated Crucible to generate the counterexamples, as this was the most challenging part.

Our initial idea for how to implement the counterexamples was similar to how tools such as Dafny were discussed in class. In this case, we have some block of code with a precondition and postcondition. Like in class, we would construct a verification condition which is the logical formula $precondition \rightarrow weakest\_precondtion(postcondition)$. We can then ask an SMT solver for a satisfying assignment to the formula $\neg(precondition \rightarrow weakest\_precondtion(postcondition))$ to obtain falsifying examples. This falsifying example would be in terms of some intermediate function variables, not the function inputs, but it may still be useful for a LLM. To get a falsifying example in terms of the arguments, we would do something akin to symbolic execution, where we build up an SMT expression in terms of the inputs to the function instead of the intermediate values present in the inverted verification condition. Solving this SMT formula should yield a counterexample in terms of the function arguments.

We first tried to modify Verus itself, but this proved difficult, as Verus is an incredibly complicated codebase. The SMT formula Verus was emitting for even a very simple function was incredibly complicated and tracked lots of things like fuel and datatypes. As a result, we decided to implement our own tool outside of Verus. Implementing the verification condition falsification would have required implementing the semantics of lots of different rust expressions in SMT queries. Perhaps doing it on the rust MIR could be possible and the MIR may have less total operations to implement, but we predicted we would have spent most of the time reimplmenting rust semantics, which was not the point of the project.

Instead, we utilized a tool called Crux-MIR which uses the Crucible symbolic execution library to perform symbolic execution on the Rust MIR. Crux-MIR is an easy to use wrapper around crucible which allows writing symbolic test cases. You add some assertions, and pass in symbolic values to the function, and crux-mir will try to find arguments which cause the assertions to fail. Our idea was to transform the Verus code into Rust code with assertions representing the Verus clauses. We could then use crux-mir to find counterexamples for these clauses.

This approach works reasonably well, but has some issues with quantifiers, as you cannot check all types of quantifiers quickly in code. The current approach is limited and generates a loop to check all values, and symbolic execution will only check some iterations of the loop up to a bounded limit. We had an improved method of converting each clause to prenex normal form, and quantifier variables can be instantiated as symbolic variables in crux-mir. This works for universal quantifiers, and for existential quantifiers we can invert the inner condition and check if crux-mir can find an assertion failure to check if the quantifier holds. The problem is a mix of universal and existential quantifiers is still inefficient to check, as we think this trick is only possible in the outer quantifiers.

Ideally, you can specify a logical clause for the symbolic execution engine to reach, rather than just a specific basic block. This may be possible by using the crucible library directly to encode more complex conditions, instead of using crux-mir.

4 Summary of Results

In this section, we give a summary of our evaluation and summary of results

4.1 Test Cases

We used two types of test cases to evaluate our pipeline, such as:

- **T1**: Linear Integer Arithemtic (LIA) Rust functions without nested loops, contains 7 Rust functions.
- **T2**: LIA Rust functions with nested loops, contains 5 Rust functions.

We also have manually annotated versions for the test cases in **T1** and **T2**. The reason for two different test sets based on the existence of nested loops is that the complexity to generate counter examples grows exponentially in Crucible based on the depth of nested loops.

4.2 Results

For evaluation, we ran the pipeline on the test functions on both the test sets. For **T2**, we were not able to complete the evalution using Crucible-generated counter examples. So, we show the evaluation results on the test cases in **T1**. The results are shown in Table 1, where the pair $(1, 0)$ means 1 function was verified, 0 faced errors.

Test	**AutoVerus (One Shot)**	**AutoVerus + Seahorn**	**AutoVerus + Crucible**	**AutoVerus + Seahorn + Crucible**	**AutoVerus + Seahorn+ Crucible + Annotations**
1	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)
2	(1,0)	(1,0)	(1,0)	(1,0)	(1,0)
3	(0,1)	(0,1)	(0,1)	(0,1)	-
4	(0,1)	(0,1)	(0,1)	(0,1)	-
5	(0,1)	(0,1)	(0,1)	(0,1)	-
6	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)
7	(0,1)	(0,1)	(0,1)	(0,1)	(0,1)

Table 1: Results for different AutoVerus configurations across test cases in **T1**. The results are displayed in the format used in Fig. 4.

From the results, it is evident that our modifications did not work as expected in almost all of the test cases. However, for a trivial program (test 2), both AutoVerus pipeline and our modifications generated the correct specifications. The "−" symbol in some of the cells indicate that due to the issues in the annotations, the pipeline was not able to generate syntactically correct Verus codes.

Even though we don't add a table containing the evaluation result on **T1** here, we get a similar result there, even after the added contexts, our pipeline is giving same result as the AutoVerus pipeline.

5 Reflection

We think that our approach was too "loose" and broad and we should have narrowed the scope of our project. Getting to the Verus code directly from the rust code does not contain enough context and details for the LLM in order to do a good job at generating the right code even though we provided the LLM with more context such as verification errors, output from symbolic execution tools etc. A better approach would have been to start extend Autoverus's current approach with more complex examples or come up with metrics such as how much more effective is the inference when we give the LLM more context from additional tools. So we could have asked research questions such as, how much did we reduce inference time? Did we use lesser tokens? Did we use lesser energy? Although its also tough to quantize these quantitative measures, the direction to extend rather than invent a new tool that takes Rust would have been a better approach on hindsight.

With regards to counterexample generation, we underestimated how long it would take to implement. There was a lot of work related to parsing and transforming everything, which meant we finished very late and did not have adequate time to implement some features such as improved quantifier checking.

Another issue we had was we never were able to implement tools to try and improve the output of the LLM; we only were able to implement tools that fed in extra context. I think this is in part because fixing the output is a harder problem. In addition, we didn't have API keys for a while, which meant we weren't able to test Autoverus that much and see what exactly the common failure modes of the LLM are. How it fails would inform us what kind of corrections we would have to implement, and without this information it was difficult to get started on this task.

Another area to explore is to take other formal verification languages and try to infer specifications for that language. For instance, Dafny. This is a complementary idea but we believe that a lot of the core concepts would remain the same in this approach. However, it also depends on the LLM, how well it is trained on prior inputs and how much Dafny, Rust, Verus code it has seen.

This bring us to the limitations of the LLM itself. The LLM needs to be well trained on input/output examples for a lot of these languages in order to be good at generating new code. For this reason, we need to make sure that we try our approach on newer models. Therefore, we need to also do a new study to test our approach with a newer LLM model.

6 Teamwork

Our tasks were divided among the members as follows:

- Vidush: Idea formalization, Generating SMT queries using Seahorn
- Jack: Generating counterexamples using Crucible
- Bishal: Integration of Crucible and Seahorn to the AutoVerus pipeline
- Faaiz: Integration of Crucible and Seahorn to the AutoVerus pipeline

7 Course Topics

The primary course topics we used are the lectures about formal verification using non interactive provers. We used some of the information about how these work when planning our counterexample generation. The overall concepts of first order logic were also used, as well as some specifics about how to convert to clausal normal form for making quantifiers more efficient (though we only needed prenex normal form).

8 Artifacts

The full implementation including source code and examples used in the project can be found here: parent-github-repository. We also forked the AutoVerus repo and the fork exists here: modified-AutoVerus. The complete docker image can be found here: docker_image.

References

[1] GaloisInc. Crucible. `https://github.com/GaloisInc/crucible`.

[2] Arie Gurfinkel, Temesghen Kahsai, and Jorge A. Navas. Seahorn: A framework for verifying c programs competition contribution. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, page 447–450, Berlin, Heidelberg, 2015. Springer-Verlag.

[3] 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. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2), 2025.