Improving Auto-Formalization to UCLID5 with LLMs and Formal Methods

Anirudh Chaudhary

Acknowledgement

First and foremost, I would like to thank my advisor, Professor Sanjit Seshia, for his incredible vision, patience, wisdom, and help on my projects throughout these past two years. Next, I would like to thank my mentor Federico Mora for sticking with me through countless meetings and helping me shape this work. I also thank Dr. Elizabeth Polgreen for her invaluable insights, ideas, and perspectives when considering the complexities of this project and helping with the evaluation. Finally, I would like to thank everyone else involved in helping me with the evaluation of this work: Kai-Chun Chang, Kevin Cheang, Pei-Wei Chen, Karim Elmaaroufi, Adwait Godbole, Aniruddha Joshi, Shaokai Lin, Alex Sanchez, Amar Shah, Victoria Tuck, Justin Wong, Beyazit Yalcinkaya, and Leiqi Ye.

Improving Auto-Formalization to UCLID5 with LLMs and Formal Methods
by Anirudh Chaudhary

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Sanjit Seshia
Research Advisor

[illegible]

(05/16/2025)

* * * * * * *

Professor Matei Zaharia
Second Reader

[illegible]

(05/16/2025)

Improving Auto-Formalization to UCLID5 with LLMs and Formal Methods

by

Anirudh Chaudhary

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Masters

in

Electrical Engineering and Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Sanjit Seshia, Chair
Professor Matei Zaharia

Spring 2025

Improving Auto-Formalization to UCLID5 with LLMs and Formal Methods

Abstract

Improving Auto-Formalization to UCLID5 with LLMs and Formal Methods

by

Anirudh Chaudhary

Masters in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Sanjit Seshia, Chair

One of the uses for large language models (LLMs) is code generation, and a growing number of tools aim to reduce developer effort by automating routine programming tasks. However, LLMs frequently generate code that appears correct but fails to meet functional requirements. This thesis addresses the problem of automating the generation of formal verification models from natural language, with a particular focus on the UCLID5 formal modeling and verification language. We propose a new pipeline to improve the semantic correctness of generated code by incorporating formal verification through variable- and line-coverage–based specifications. For each natural language task in our dataset, we generate four versions of code: a baseline from previous work, specification-guided generation with bounded model checking (BMC), generation with smoke testing, and a combination of both. We evaluate these variants through a user study in which UCLID5 users and developers assess output quality.

In this work, we implement two additional techniques–specification generation and automated repair–to assist in the code generation process. We find that while the use of the additional techniques, namely specification generation and automated repair, did not significantly increase user preference compared to the baseline, our results reveal important nuances. Our feedback-driven repair process led to either improvement or no regression in 93% of cases, highlighting its promise as a low-risk refinement layer. However, we find no clear correlation between a model's assertion pass rate and user preference, suggesting that current specification strategies may not yet align with human-centered notions of code quality. These findings suggest that while formal methods may not yet enhance perceived output quality, they offer a principled foundation for improving semantic alignment in code generation. Future work should focus on creating and using structural specifications to better capture user intent.

To my family and friends

Thank you for your support throughout this entire journey.

Contents

List of Figures

Acknowledgments

First and foremost, I would like to thank my advisor, Professor Sanjit Seshia, for his incredible vision, patience, wisdom, and help on my projects throughout these past two years. Next, I would like to thank my mentor Federico Mora for sticking with me through countless meetings and helping me shape this work. I also thank Dr. Elizabeth Polgreen for her invaluable insights, ideas, and perspectives when considering the complexities of this project and helping with the evaluation. Finally, I would like to thank everyone else involved in helping me with the evaluation of this work: Kai-Chun Chang, Kevin Cheang, Pei-Wei Chen, Karim Elmaaroufi, Adwait Godbole, Aniruddha Joshi, Shaokai Lin, Alex Sanchez, Amar Shah, Victoria Tuck, Justin Wong, Beyazit Yalcinkaya, and Leiqi Ye.

Chapter 1

Introduction

Large language models (LLMs) have shown impressive capabilities in code synthesis across a range of programming languages and domains [8]. However, ensuring the correctness and reliability of LLM-generated code remains a significant challenge, particularly in formal verification contexts where even small errors can render a system unsafe or unusable. This work investigates the integration of formal methods—specifically bounded model checking, smoke testing, and specification generation—into the LLM-driven code generation process. We discuss these formal methods more in depth in section 1.4. Using UCLID5 [20, 17, 18], a modeling and verification tool for transition systems, we develop and evaluate four variations of a code generation pipeline: a baseline from previous work, pipelines with smoke testing or bounded model checking, and a combined version that incorporates both. We assess these pipelines through user studies, specifically measuring the effectiveness of these methods by analyzing assertion pass rates and unreachable code. Our goal is to understand how these formal techniques impact code correctness, user preference, and the overall stability of the generation process. Through this evaluation, we provide insight into the limitations of current specification-generation strategies and offer guidance for building more reliable formal-method–aware code synthesis pipelines.

1.1 Motivation

Eudoxus [16] is a system that generates code in a very low-resource verification language (UCLID5) by leveraging an SMT-based repair loop. Eudoxus uses a methodology called Synthetic Programming Elicitation and Compilation (SPEAC) to produce syntactically correct code. While Eudoxus 1.0 excels at generating syntactically correct code, our analysis reveals that it struggles with semantic correctness across numerous tasks. This discrepancy becomes evident when examining the UCLID5 output for specific problem specifications. As shown in Figure 1.1, we analyze code generated by Eudoxus 1.0 for an example from our training set. The problem describes a traffic control system with four roads, each having a traffic light that can display four distinct states: green, yellow, red, and amber. Addition-

Consider a street junction with 4 incoming roads. Each of the 4 roads has a traffic light, which are L1, L2, L3, L4.
The traffic light has 4 states: green, yellow, red, and amber which happen in that order.

Write a model of this junction with 4 traffic lights, showing a reasonable controller C that switches the green traffic lights in the following order: L1 and L3 together, then L2, then L4.

Use the variable names L1, L2, L3, L4, green, yellow, red, amber

(a)

```
module TrafficLights {
  ...
  init {
    L1 = false; L2 = false; L3 = false; L4 = false;
    green = true; yellow = false; red = false; amber = false;
  }
  next {
    if (L1 || L3) {
      green' = true; yellow' = false;
      red' = false; amber' = false;
      if (L1) {
        L1' = false; L3' = true;
      } else {
        L3' = false; L2' = true;
      }} ...}}
...
```

(b)

Figure 1.1: Partial task description from Lee and Seshia [11] (a) and partial output of Eudoxus 1.0 in UCLID5 (b). We interpret `L1, L2, L3, L4` to represent traffic lights. The `init` and `next` blocks do not capture the required complexity specified in the problem description.

ally, the problem specifies a particular transition sequence where L1 and L3 change states together, followed by L2, and then L4. The Eudoxus 1.0 generated model fails to capture these requirements in two critical ways: First, the implementation represents each traffic light as a Boolean variable, limiting them to only two states instead of the four specified in the problem. This simplification fundamentally misrepresents the problem description. Second, the transition dynamics coded in the next block show a misunderstanding of the required behavior. The code checks if L1 or L3 are true, then makes changes to incorrectly defined variables labeled as green, yellow, red, and amber. This implementation completely misses the pattern specified in the original problem. We believe that developing formal specifications based on the problem description would allow us to verify the generated code against the actual requirements. This approach would clearly identify semantic inconsistencies that currently go undetected, potentially leading to more reliable code generation in future iterations of the system.

1.2 Thesis Contributions

This thesis makes the following contributions to the study of improving semantic correctness in LLM-based code generation through formal methods:

- **A formal methods–augmented generation pipeline:** We design and implement a code generation pipeline that incorporates formal verification into the LLM workflow. The pipeline supports multiple verification-backed configurations, enabling comparative evaluation of their impact on code quality.
- **Coverage-based specification generation and feedback integration:** We develop methods for automatically synthesizing two types of coverage-based specifications—variable coverage and line coverage—which are used to validate candidate programs using bounded model checking (BMC). Verification feedback is then used to guide iterative repair.
- **Empirical analysis of repair efficacy:** We demonstrate that LLMs are often capable of addressing verification failures when provided with feedback from formal methods. However, we also show that high assertion pass rates do not necessarily correlate with user preference, suggesting a mismatch between machine-verifiable specifications and human-perceived correctness.
- **A user study evaluating formal verification in practice:** We conduct a user study involving 12 participants and 33 natural language programming tasks. Participants evaluate multiple versions of code outputs across different configurations of the formal method–enhanced pipeline.
- **Quantitative and qualitative assessment of user preference:** We analyze user ratings to assess the impact of formal methods on perceived output quality. The results show that while specification-driven verification alone does not improve preference, feedback-guided repair yields net improvement or no degradation in 93% of evaluated cases.
- **A framework for isolating the role of specifications vs repair:** Through our controlled pipeline and multi-version generation strategy, we isolate the effects of specification quality and feedback repair. This enables a deeper understanding of where the bottlenecks lie in aligning verification techniques with user-centric code evaluation.

1.3 Related Work

Large Language Models for Code Generation

The emergence of large language models (LLMs) has significantly advanced automated code generation by enabling models to synthesize code from natural language specifications. Recent surveys, such as Jiang et al. [10], provide a comprehensive overview of the capabilities and limitations of LLMs in this domain. These models have demonstrated strong performance in tasks such as function synthesis, bug fixing, and docstring generation. State-of-the-art code generation models have evolved rapidly in recent years. CodeLlama [19], a code-specialized variant of the Llama architecture, demonstrates exceptional reasoning capabilities across multiple programming languages. Similarly, StarCoder [12] and its successor StarCoder2 [14] were trained on permissively licensed code repositories to produce high-quality code completions. Codex [6], the foundation for GitHub Copilot, revolutionized code assistance by generating contextually relevant code segments based on natural language prompts or partial implementations. More recent models like CodeT5+ [22] extend the T5 architecture with code-specific pre-training objectives to enhance generative performance. Despite these advances, LLM-generated code faces persistent challenges. Systematic studies by Chen et al. [7] and Liu et al. [13] document common failure modes, including hallucinated API calls, incorrect algorithm implementations, failure to handle edge cases, and security vulnerabilities. These limitations, coupled with concerns about hallucinated outputs, lack of semantic alignment, and brittleness in edge cases, motivate the need for integrating correctness guarantees into the generation process.

Integrating Formal Methods into LLM-Based Code Generation

To improve the reliability and trustworthiness of generated code, recent work has focused on coupling LLMs with formal methods. LLMLift [4] introduces formally-verified code transpilation using LLMs. They use verification oracles to guaranteee correctness of the generated code. We use this approach during the compilation step of our pipeline. VeCoGen [21] is a prominent example that leverages formal verification in the code generation loop. It accepts an ACSL (ANSI/ISO C Specification Language) specification, a natural language task description, and a suite of test cases, then iteratively produces and verifies candidate C programs until they meet both the functional and formal correctness constraints. Similarly, AlphaVerus [1] introduces a self-improving framework for verified code generation without requiring fine-tuning or human supervision. It does so by leveraging higher-resource language models and verification feedback to bootstrap more accurate and verifiable program synthesis. These approaches highlight the growing interest in aligning LLM outputs with formal guarantees through verification-aware feedback loops.

Another line of research integrates LLMs with symbolic reasoning and formal verification to infer and validate program specifications. Lue et al. [23] propose an iterative pipeline wherein candidate invariants generated by LLMs are filtered and refined using bounded model checking (BMC) and SMT-based validation. Their system feeds verification results back into the model, improving its ability to synthesize correct invariants over time. Additionally, tools like ESBMC [15], an SMT-based context-bounded model checker, have been used to automatically verify safety properties in LLM-generated code. ESBMC supports multiple languages and can verify both pre-specified and user-defined assertions, making it a practical tool for detecting runtime errors and verifying compliance with critical properties. The challenge of translating natural language requirements into formal specifications remains significant.

1.4 Background

Eudoxus 1.0

Eudoxus [16] is a system that generates code in a very low-resource verification language (UCLID5) by leveraging an SMT-based repair loop. Eudoxus first has a large language model (LLM) produce code in a Python-derived pseudo-language, then uses a MaxSMT solver (Z3) to automatically fix any violations of the target language's rules. If the solver cannot fully repair the program, Eudoxus inserts holes and asks the LLM to fill in the missing pieces, repeating as needed. The combination of deduction repair and LLM synthesis dramatically improved syntactic correctness, with Eudoxus producing parsable UCLID5 code 84.8% of the time, versus only around 12% for GPT-4 with naive prompting. For the rest of this paper, the original Eudoxus pipeline will be referenced as Eudoxus 1.0.

Bounded Model Checking

Bounded Model Checking (BMC) is a formal verification technique that systematically explores the behavior of a system over a finite number of steps. It works by unrolling the transition relation of a program or model up to a given bound k, checking whether any property violations occur within those k steps. These properties—such as safety assertions—are encoded into a logical formula, which is then checked for satisfiability using an SMT solver.

One of the key strengths of BMC is its ability to produce concrete counterexamples when properties are violated, making it particularly useful for debugging. However, BMC is inherently incomplete for unbounded systems, as it cannot guarantee correctness beyond the specified bound.

In this work, we use BMC within the code generation pipeline to evaluate whether LLM-generated UCLID5 modules satisfy their associated specifications. When violations are detected, we collect counterexamples and use them to guide feedback-driven repairs to the model.

Smoke Testing

Smoke testing is a lightweight verification technique used to quickly identify obvious failures in a system. In the context of program synthesis and code generation, smoke tests typically validate basic execution behaviors—such as variable initialization, type consistency, and termination—without requiring formal specifications or deep semantic understanding.

In our work, we explore smoke testing as both the primary verification technique and final check on the LLM-generated UCLID5 modules.

Specification Generation

Specifications define the expected behavior of a system and form the foundation of any formal verification effort. In this work, we focus on generating assertion-based invariants, which are logical conditions that should hold during the execution of the system. These specifications are necessary for both bounded model checking and feedback-based repair to function effectively.

Our approach to specification generation is coverage-based: for every variable declared in a UCLID5 module, we generate at least one corresponding assertion. The primary goal is to ensure that all declared program variables are constrained by at least one logical condition, thereby promoting coverage of the model's state space.

UCLID5

UCLID5 is a tool for the multi-modal formal modeling, verification, and synthesis of systems [20, 17, 18]. UCLID5 is an evolution of the earlier UCLID verification system [5]. It supports a variety of data types and logical constructs, including bit-vectors, booleans, and reals. UCLID5 allows users to define modules with state variables, transition relations, and assertions, and then verify the correctness of these models through bounded model checking or symbolic simulation. In this work, UCLID5 serves as the target language for our LLM-generated code, and acts as the verification backend for evaluating specification satisfaction and program correctness.

SMT-Based Techniques

Satisfiability Modulo Theories (SMT) refers to the problem of determining whether a logical formula is satisfiable with respect to a background theory, such as arithmetic, bit-vectors, or arrays [3]. SMT solvers extend propositional SAT solving by incorporating these richer theories, enabling reasoning about real-world program properties. In formal verification, SMT-based techniques are commonly used to encode transition systems, properties, and counterexamples, allowing tools like UCLID5 to check complex correctness conditions efficiently. Our pipeline leverages SMT solvers to perform bounded model checking.

1.5 Contributions to the Thesis

The work reported in this thesis is joint with Federico Mora's work on Eudoxus [16]. The work used for specification generation came from many discussions in the Learn and Verify Group, specifically with Elizabeth Polgreen and Federico Mora. The smoke testing feature that we used as part of our pipeline is thanks to work done by Alex Sanchez. The evaluation section was iterated and finalized after discussions with Elizabeth Polgreen, Adwait Godbole, Pei-Wei Chen, and Alex Sanchez. Finally, this work was advised by Professor Sanjit Seshia.

Chapter 2

Eudoxus 2.0

2.1 Approach

Our work builds upon the foundation established by Eudoxus 1.0 [16], which primarily focused on ensuring syntactic correctness of generated UCLID5 modules. While the original implementation demonstrated some awareness of semantic correctness, we identified opportunities to enhance the quality of generated models by integrating formal verification techniques into the development pipeline. The motivation for this enhancement stemmed from our observation that syntactically correct code often fails to capture the intended behavior specified in problem statements. By incorporating formal methods, we could systematically identify and address semantic inconsistencies that previously went undetected. To improve the semantic accuracy of the generated UCLID5 modules, we added a comprehensive semantic pipeline consisting of four key stages (also pictured in Figure 2.1):

- **Specification generation:** automatically deriving formal specifications from problem descriptions.
- **Formal Method Invocation:** verifying the generated code using bmc and/or smoke testing to find unreachable lines.
- **Error Parsing:** parsing and simplifying verification results concisely for feedback generation.
- **Feedback generation + Integration:** translating verification results into actionable feedback and integrating it for repair.

This pipeline creates a verification loop that allows us to identify semantic errors in the generated code and use this information to guide improvements in future iterations.

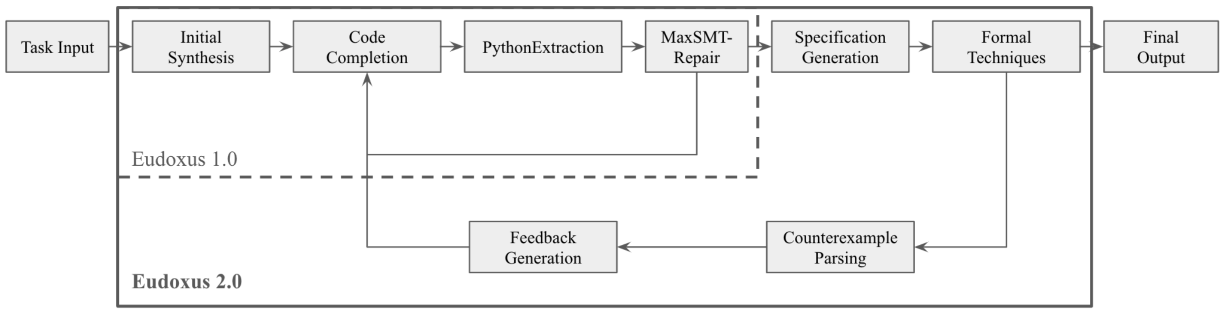


Figure 2.1: Eudoxus Pipeline Overview

Independent Specification Generation

In the original Eudoxus 1.0 pipeline, the large language model (LLM) is prompted to complete a predefined `Module` class (see Figure 2.2) and contains a specification block. In contrast, Eudoxus 2.0 decouples the specification synthesis process from code generation. During the initial prototype synthesis, there are two LLM calls, one responsible for generating code aligning to the task, and the other generating a list of specifications for the input task. Since the specification generation is decoupled from the code generation, the Module class in the initial synthesis does not have a specification block defined. All future code completion prompts use an augmented Module class definition, like the one pictured in Figure 2.2.

This work focuses on generating coverage-based specifications with the primary goal of ensuring that all defined variables in a module are being semantically checked. Specifically, for each iteration of the synthesis-repair pipeline, we ensure that invariants are generated for every variable declared in that version of the module. If, after the repair stage, the model removes or modifies the specification block in a way that omits these invariants, we regenerate them by re-invoking the LLM with targeted prompts. The resulting invariants are inserted into the specification block using a designated placeholder (i.e. a “hole”).

Invariant generation is performed using constrained decoding techniques [24], ensuring that the output adheres to a structured schema. Given the most recent version of the module, current code implementation, and the task description, the LLM is instructed to generate invariants for each defined variable. We define an `InvPair` object as a tuple consisting of a variable name and its corresponding invariant. The model returns an `InvPairList` object containing invariant expressions for all relevant variables. While each invariant must reference its associated variable, the LLM is allowed to incorporate dependencies on other variables within the module, enabling the specification of relational constraints when appropriate.

The generated `InvPairList` is parsed and injected into the module's specification block during each iteration of the pipeline. Figure 2.3 illustrates a representative example of a completed `specification` block following this process.

```
class Module:
    """An abstract class to
    represent a UCLID5 module."""

    def types(self):
        """(Optional)
        Defines the type declarations.
        For example, the following implementation
        defines a 8-bit type called T:
        ```
        def types(self):
            self.T = BitVector(8)
        ```
        """
        pass
    ...
    def specification(self):
        """(Optional) Defines the specification in
        terms of invariant properties.

        Returns:
            bool: True if the specification is satisfied,
            False otherwise.

        For example, the following implementation
        defines two invariants:
        ```
        def specification(self):
            return self.x < 10 and self.y > 0
        ```
        """
        pass
```

(a) Eudoxus 1.0

```
class Module:
    """An abstract class to
    represent a UCLID5 module."""

    def types(self):
        """(Optional) Defines the type declarations.
        For example, the following implementation
        defines a 8-bit type called T:
        ```
        def types(self):
            self.T = BitVector(8)
        ```
        """
        pass
    ...
    def specification(self):
        """(Required) Defines the specification
        in terms of invariant properties.

        Returns:
            bool: True if the specification is satisfied,
            False otherwise.

        For example, the following implementation
        defines two invariants:
        ```
        def specification(self):
            return self.x < 10 and self.y > 0

        DO NOT JUST RETURN THE VARIABLE TYPES!
        """
        raise NotImplementedError
```

(b) Eudoxus 2.0

Figure 2.2: Partial module stencils from Eudoxus 1.0 (a) and Eudoxus 2.0 (b). The main difference to note are in the ‘specification‘ block. Eudoxus 2.0 makes this block required, instructs the LLM to not return variable types, and has a raise NotImplementedError line.

Preliminary experiments revealed that this approach significantly increased the likelihood that the LLM produced well-formed specifications in the desired location. As a result, this structure was adopted as the default strategy throughout the pipeline.
```

```
def specification(self):
    #L1 == green -> L3 == green in prop logic looks like:
    self.L1_inv = ??
    #(L1 == red && L3 == red) -> L2 == amber in prop logic looks like:
    self.L2_inv = ??
    #L3 == green -> L1 == green in prop logic looks like:
    self.L3_inv = ??
    #(L2 == red) -> L4 != green in prop logic looks like:
    self.L4_inv = ??
    # return expression should combine all previous invariants together
    return ??
```

Figure 2.3: Specification Block Creation

Bounded Model Checking

With generated specifications that represent the semantic requirements of a correct implementation, we employ a formal verification technique called bounded model checking (BMC) to identify property violations.
Our verification process implements a three-tiered approach: First, we verify the initialization conditions by running BMC with 0 iterations, effectively checking that the initial state satisfies all invariants and assertions. This step catches immediate violations in the setup of the system before examining any transitions. Next, we extend the verification to 3 iterations to evaluate how the state evolves through the next block. This intermediate check identifies common patterns of transition relation errors while maintaining computational efficiency. Finally, we leverage the LLM's understanding of the problem domain by prompting it to suggest an appropriate bound that will adequately test the system dynamics. This adaptive approach ensures we allocate sufficient verification resources to complex behavioral properties while avoiding unnecessary computation for simpler systems. This progressive verification strategy enables us to identify semantic errors at different levels of system behavior, from initialization to complex dynamic properties, providing targeted feedback for improvement.

Error Parsing / Feedback

Following the bounded model checking phase, we encounter one of two possible outcomes. In the first scenario, the generated UCLID5 module successfully passes all three verification checks, indicating that the implementation satisfies the specified properties. When this occurs, we proceed to the smoke testing stage.
In the second scenario, where verification fails, UCLID5 produces counterexamples that demonstrate specific violations of the specified properties. These counterexamples are valuable but require interpretation to be useful for improving the generated code. Our system

transforms these raw counterexamples into actionable feedback that can guide corrections in subsequent iterations.

A significant challenge in this process is managing the complexity of counterexample traces, particularly for failed assertions at larger bound values where traces can become prohibitively lengthy. To address this issue, we implement a filtering mechanism that captures the most relevant information by tracking the final three steps for each unique failed invariant.

This focused approach to error processing serves two key purposes: it simplifies the number of semantic failures, and it provides targeted guidance for corrective actions. By distilling complex verification results into clear, actionable feedback, we create a more efficient iterative improvement cycle for the code generation system. We generate feedback in the form of function-specific fixes. We query our LLM for these fixes, providing the original task description, current Python model, and counterexample message. The feedback generated in this stage directly informs the next round of code generation, creating a verification-guided improvement loop that progressively enhances the semantic correctness of the generated UCLID5 modules.

Smoke Testing

Successful verification through bounded model checking does not guarantee correctness of the generated model, as this outcome may result from insufficiently rigorous specifications. To address this limitation, we incorporate smoke testing as an additional validation measure, specifically targeting unreachable code sections that may indicate semantic deficiencies.

While UCLID5 provides built-in smoke testing capabilities, we extended this functionality to better serve our specific requirements. Our implementation addresses a critical challenge in the verification pipeline: maintaining traceability between the UCLID5 representation and its Python counterpart.

Given our observation that LLMs demonstrate superior performance with Python, we aim to perform as much processing as possible within the Python version of the model. However, when smoke testing identifies unreachable code sections in UCLID5, we need a reliable mechanism to map these findings back to the corresponding elements in the Python module.

Our solution leverages the abstract syntax tree (AST) representation that underlies both the Python and UCLID5 versions of the model. During the code generation process, we assign unique identifiers to each AST node—whether representing statements, conditionals, or variables. When rendering the code in either language, we include these identifiers as comments alongside the corresponding lines.

This approach creates a bidirectional mapping between the two representations. When smoke testing identifies unreachable sections in the UCLID5 model, we collect the identifiers for all affected lines. These identifiers then serve as precise targets for the fix synthesis step, enabling focused corrections without disrupting functioning components of the model.

Best Model Distinction

Establishing an objective evaluation framework for determining the optimal model in what is inherently a subjective domain presents significant challenges. We acknowledge that ultimate model quality assessment depends on user-specific requirements. Nevertheless, we developed a structured heuristic approach with clearly defined evaluation criteria to guide model selection throughout the iterative process.
Our evaluation framework begins with a straightforward initialization: if no best model has been previously identified, the first generated model automatically assumes this designation, providing a baseline for subsequent comparisons. For all subsequent iterations, we evaluate new models against the current best model using three key metrics, applied hierarchically:

1. **Compilation Status:** Our primary requirement is that the UCLID5 program successfully compiles. A model that compiles is inherently preferable to one that fails this basic criterion, as compilation represents the minimum threshold for potential correctness.

2. **Failed assertions:** When bounded model checking is enabled, we track the number of failed assertions. Our empirical observations indicate that the quantity of generated specifications typically remains consistent across iterations, making the number of satisfied assertions a reliable comparative metric for semantic correctness.

3. **Unreachable code:** When smoke testing is active, we consider the number of warnings generated due to unreachable code sections. Models with fewer unreachable sections are preferred, as this typically indicates more coherent logic and better alignment with the intended functionality.

When both bounded model checking and smoke testing are active, we prioritize assertion failures over unreachable code warnings. This prioritization stems from our finding that dynamic analysis with concrete counterexamples identifies more critical semantic issues compared to static detection of unreachable code. Failed assertions directly highlight violations of specified behavioral properties, while unreachable code may sometimes represent redundancy rather than functional errors. This multi-tiered evaluation approach provides a systematic method for tracking improvement across iterations while maintaining focus on the most critical aspects of model correctness.

Constrained Decoding for Fixes

Since initial models rarely pass verification checks, we developed a systematic approach for iterative model refinement based on execution feedback. This mechanism translates verification results into targeted fixes that address specific deficiencies identified during bounded model checking or smoke testing. We leverage an LLM with constrained decoding to synthesize precise fixes for the model. The LLM receives a carefully crafted input comprising of

```
class TrafficLights(Module):
   def locals(self):
       #TODO: Declare the light states before using them in the 'locals'
function. It seems there is a missing declaration for the type 'Light,
which should be defined as a set of states (green, yellow, red, amber).
       ??
   ...
```

Figure 2.4: Example fix with TODO and ?? for Eudoxus 1.0 "Complete" Step

the original task description, current Python implementation, and identifiers for unreachable code sections. Our prompt directs the model to produce a structured `SuggestionList` where each suggestion maps to a specific function block and includes a detailed description of the required fix. After parsing this structured output, we insert the fix descriptions into their corresponding blocks within the Python version of the model. This annotated model then reenters the syntax repair loop for implementation of the suggested changes. To optimize the effectiveness of the syntax repair process, we implemented two key enhancements based on empirical observations. First, we prepend each fix description with a "TODO" marker, which significantly improves the LLM's attention to these specific sections requiring modification. Second, we append a syntactic "hole" (represented as ??) to each fix description, leveraging the syntax repair loop's built-in mechanism for identifying and resolving incomplete code segments. Figure 2.4 illustrates one such fix structure, demonstrating how verification feedback translates into structured fix suggestions and ultimately into code modifications. This approach creates a closed-loop system where verification results directly inform targeted improvements to the model.

Four Methodologys

This section is for the reader's convenience to clearly see the difference between the four versions that were tested.

- **Plain:** The 'plain' methodology refers to the pipeline that was used in the original Eudoxus paper- without any formal verification integration.
- **All:** The 'all' methodology refers to the pipeline that generates specifications, invokes bounded model checking, and runs smoke testing.
- **BMC:** The 'bmc' methodology refers to the pipeline that generates specifications and invokes bounded model checking.
- **Smoke:** The 'smoke' methodology refers to the pipeline that only runs smoke testing.

2.2 Implementation

Much of the experimental setup for Eudoxus 2.0 remained the same as Eudoxus 1.0. We use tree-sitter to parse partial programs, Z3 to solve MAX-SMT queries, and make LLM calls through the OpenAI Python API. We use two models in our pipeline for code generation and model feedback. Code generation is done with gpt-3.5-turbo-0125 while feedback and specifications are generated with gpt-4-turbo-2024-04-09. Our MAX-SMT queries are solved locally on a 2.6 GHz 6-Core Intel Core i7 processor with 16 GB of RAM. We allow the syntactic repair pipeline to run for at max 5 iterations like in Eudoxus 1.0, and allow the feedback loop to run for at max 5 iterations.

Chapter 3

Evaluation

In this section, we discuss the experimental setup and results for Eudoxus 2.0, the formal method backed version of the SPEAC prototype for UCLID5. We seek to evaluate the performance of Eudoxus 2.0 across the following research questions:

- **RQ1:** Which verification-guided code generation approach demonstrates highest user preference across the four implemented methodologies?
- **RQ2:** To what extent do participants desire code generated from each implementation variant?
- **RQ3:** What relationship exists between the number of satisfied formal assertions and user-assigned quality ratings?
- **RQ4:** Which structural components are most frequently identified as semantically deficient in LLM-generated UCLID5 modules?
- **RQ5:** How effectively does BMC or smoke testing improve model quality, as measured by assertions passed?
- **RQ6:** What relationship exists between the prevalence of unreachable code segments and user-assigned quality ratings?

Benchmarks

Our dataset consists of 33 natural language task descriptions. These natural language descriptions are pulled from three textbooks. We pull 21 examples from ”Principles of Model Checking” by C. Briar and J.P. Katoen [2], 3 examples from ”Logic in Computer Science: Modelling and Reasoning about Systems” by M. Ruth and M. Ryan [9], and 9 examples from ”Introduction to Embedded Systems 2ND ED” by E Lee and S. Seshia [11].
For each example in our dataset and each version with toggled features, we run the described

pipeline. While iterating through the pipeline, we keep track of three pieces of information: a log of the outputs, meta-data csv, and final output. The folders with all of this information are in the Github.

3.1 Experiment Setup

We evaluated the desirability of code generated from our pipelines by conducting a user study of UCLID5 users and developers ranging from different levels of familiarity with UCLID5.

User Study

In our experimental evaluation, we sought to quantify user preferences regarding various verification-guided code generation features. We implemented a tournament-style comparison framework in which participants selected preferred outputs and indicated their relative desire for the generated code. While this methodology was used in our final experimental design, we conducted a preliminary user study that was subsequently revised based on participant feedback. A comprehensive discussion of the initial experiment is provided in the Appendix for reference.

Tournament Structure

To implement the comparative evaluation, we anonymized the output from each system variant by assigning unique identifiers, maintaining the mapping between these identifiers and their corresponding implementation versions. We then constructed tournament brackets for systematic comparison. As an illustrative example, for a given task with four outputs—baseline (A), comprehensive (B), bounded model checking (C), and smoke testing (D)—the first round might pair A versus B and C versus D. Participants viewed only the anonymized identifiers rather than implementation details. For each pairing, participants indicated their preference, with an option to select neither if they found both implementations equally satisfactory or unsatisfactory. Winners from the first round advanced to a final comparison, where participants selected their ultimate preference, determining the tournament winner. To mitigate potential bias from specific pairings, we implemented three distinct bracket variations: (1) A versus B and C versus D, (2) A versus C and B versus D, and (3) A versus D and B versus C. The 33 example tasks were distributed equally across these variations, with 11 examples per configuration. For statistical robustness, we conducted each example evaluation twice, resulting in 66 total evaluations.

Code Desirability

Beyond simply comparing two output files, we ask participants to evaluate the desirability of the generated code itself. Specifically, for each of the two UCLID5 files presented, the user is prompted to indicate how much of the code they would choose to retain. The options provided are: ”I would keep very little or none of this code”, ”I would keep some of this code”, and ”I would keep all of this code”. This question aims to capture a more nuanced, user-centered assessment of the generated code’s perceived quality and utility.

User Rating Score Calculations

For many of the subsequent graphs and results, we explore the relationships between various variables and user ratings in an effort to identify potential correlations. However, computing the user rating scores for each output is not entirely straightforward. While each user was shown a distinct set of outputs in their respective Google Forms, some forms were shared across multiple participants. As a result, certain output files received multiple responses to the question, ”How much of this code would you want to keep?”. Naturally, these participants expressed differing preferences and expectations regarding the code they were shown.

To aggregate these varying responses, we assign a numerical value to each response category: ”little/none” is assigned a score of 1, ”some” receives a score of 3, and ”all” is given a score of 5. The final score for each output file is then computed as the average of all its corresponding response values. For example, if 25.ucl received one response of ”some” and another of ”all,” the resulting score for that output would be 4.

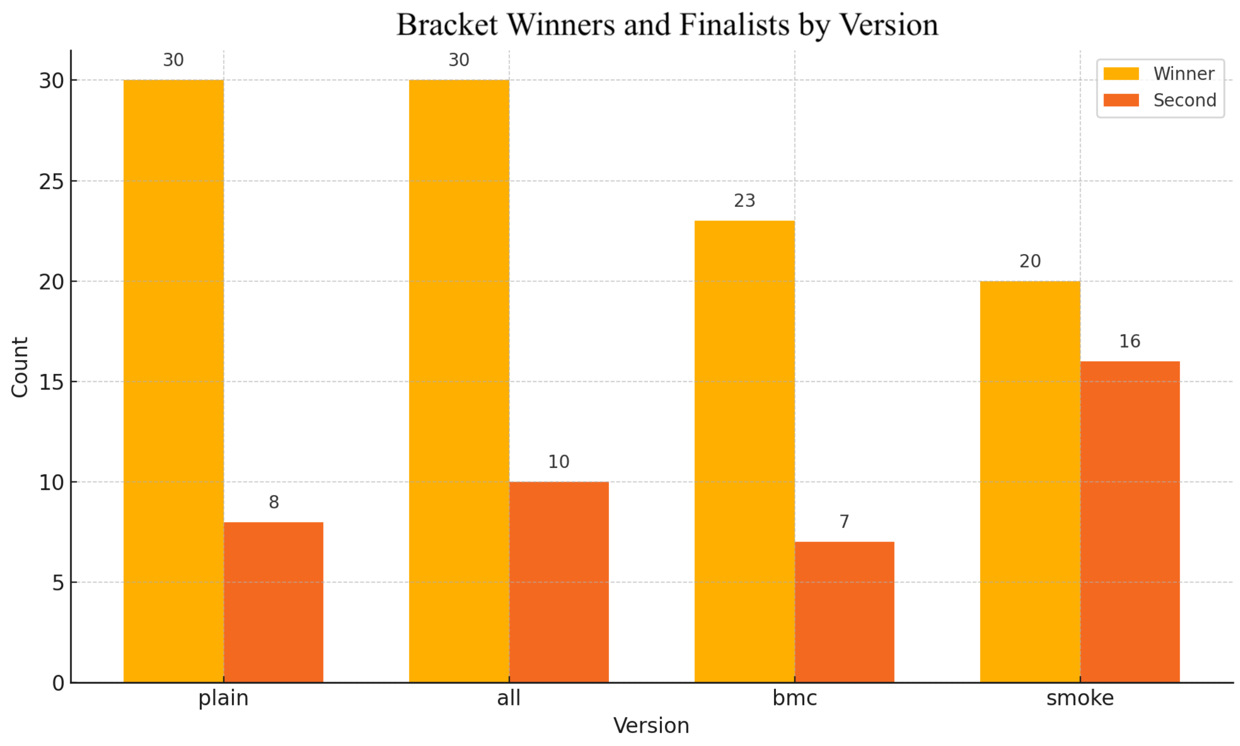


Figure 3.1: Final aggregate winners and finalists split by each version

3.2 Results

In this section, we will look at answer at the data collected for each of the research questions.

RQ1: Method Preference

This graph illustrates user preferences for each of the output versions, aggregated across all bracket comparisons. A few notable trends emerge from this data, particularly when comparing the "all" method to the "plain" method. Interestingly, both versions won an equal number of brackets, suggesting that there may be no substantial difference in user preference between the original Eudoxus output and the versions enhanced with bmc/smoke testing. Although the "all" version reached the final round more frequently than any other, it did so by a narrow margin. **Another noteworthy observation is the underperformance of the "bmc" variant, which suggests that there may be underlying issues in the specification generation process used in this particular pipeline.**

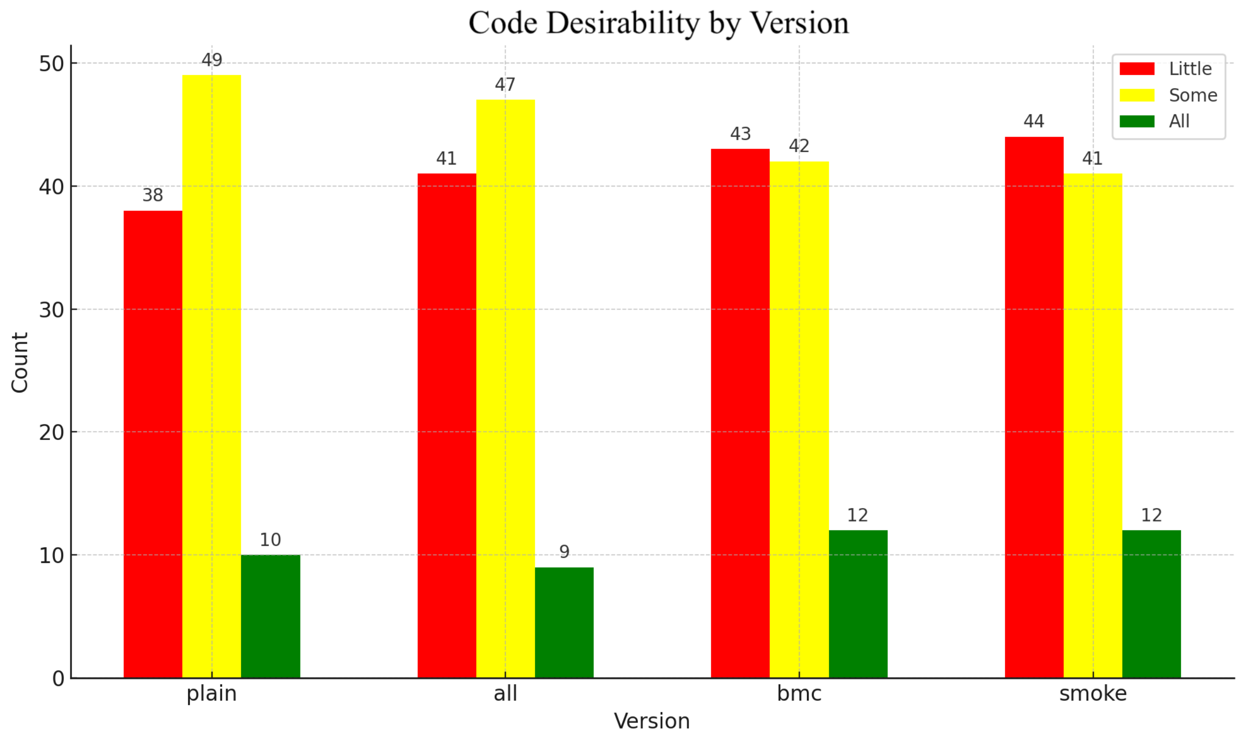


Figure 3.2: User ratings of code desirability across versions, grouped into three categories: All (fully desirable), Some (partially desirable), and Little (largely undesirable).

RQ2: Code Desirability

The graph above presents the aggregate distribution of code desirability ratings for each output version. Overall, the ”plain” version performed the best, with participants indicating a stronger inclination to retain more of its generated code. The ”bmc” version, however, offers a particularly interesting case. Despite being the lowest performer in the preference-based bracket evaluations, participants still expressed a notable willingness to keep code from its outputs. This suggests that relying solely on comparative preference data may overlook important nuances in the quality of individual outputs. In fact, the ”bmc” version received the highest number of ”I would keep all of this code” ratings, while also receiving nearly the highest number of ”I would keep very little or none of this code” responses. **This polarization indicates that the outputs from the ”bmc” pipeline tend to be either highly effective or notably subpar, with little in between.**

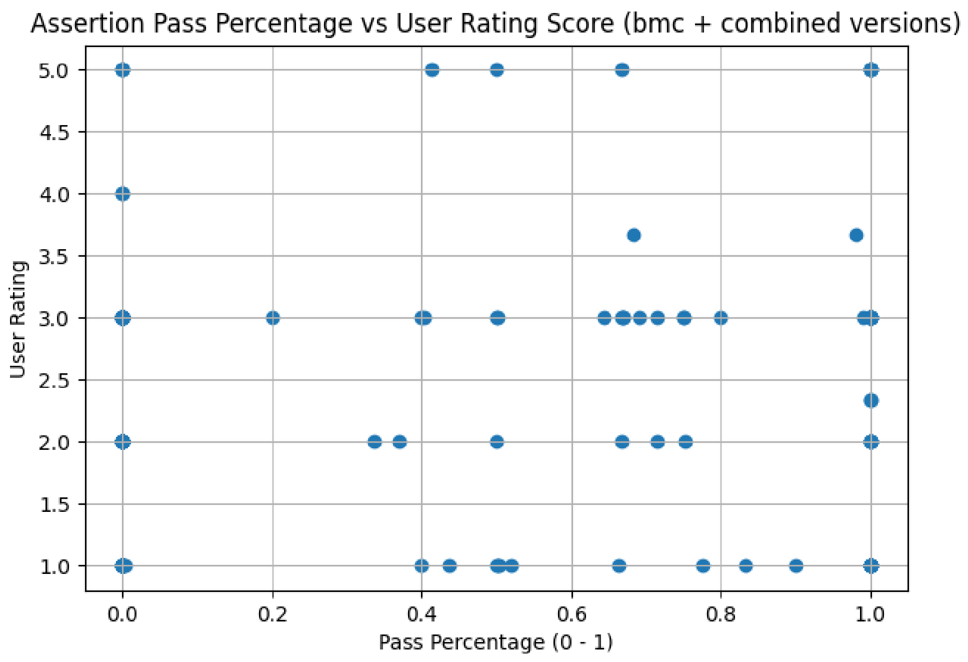


Figure 3.3: Aggregated Assertion Pass Percentage vs User Rating for "bmc" and "all" pipeline outputs. This graph contains pass percentages for any pipeline that incorporates a bounded model checking step.

RQ3: Passed Assertions vs User Rating

For all outputs that underwent bounded model checking (BMC), we selected the best model generated and ran BMC for 100 iterations, recording the number of assertions that passed and failed. The graph above plots the pass percentage of these assertions—defined as the proportion of passed checks out of total checks—against the corresponding user rating for each output. The resulting Pearson correlation coefficient is 0.05, with a p-value of 0.6050. Contrary to our expectations, **we did not observe a statistically significant positive correlation between specification pass percentage and user rating.** This result suggests that **our current approach to generating coverage-based specifications may not align with user priorities or preferences, and highlights the need for further refinement.**

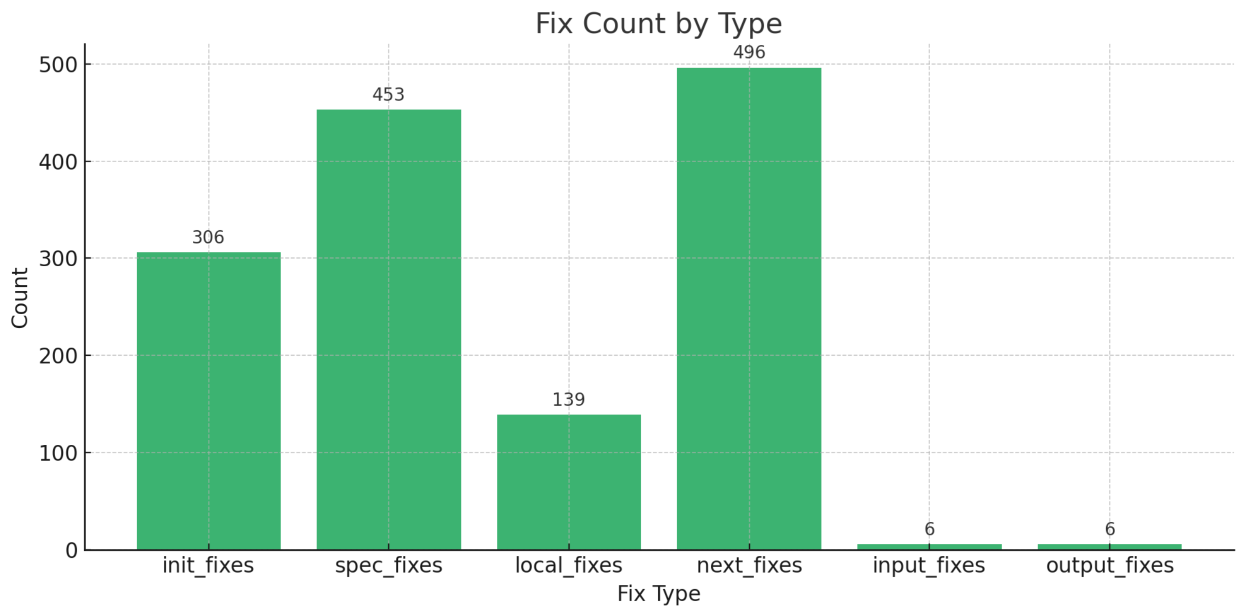


Figure 3.4: Fix Count by Function Block

RQ4A: Semantically Deficient Sections

In the formal-method–backed pipeline, the LLM interprets feedback produced by the formal method and generates a corresponding fix in the form of a (block, description) pair. Here, the block refers to the function or code region targeted for modification, while the description outlines the nature of the proposed change. As part of our analysis, we tracked the frequency of fixes applied to each block throughout the pipeline.

The graph above shows the distribution of fix counts by block. As expected, the next block received the highest number of fixes. This is consistent with intuition, as the next block defines the program's transition relations and core operational logic—areas that are particularly sensitive to formal verification feedback. Notably, the specification block exhibited the second-highest number of fixes. **This suggests that a significant portion of the issues detected by the formal method stem from the way specifications are generated, pointing to specification synthesis as a potential area for improvement.**

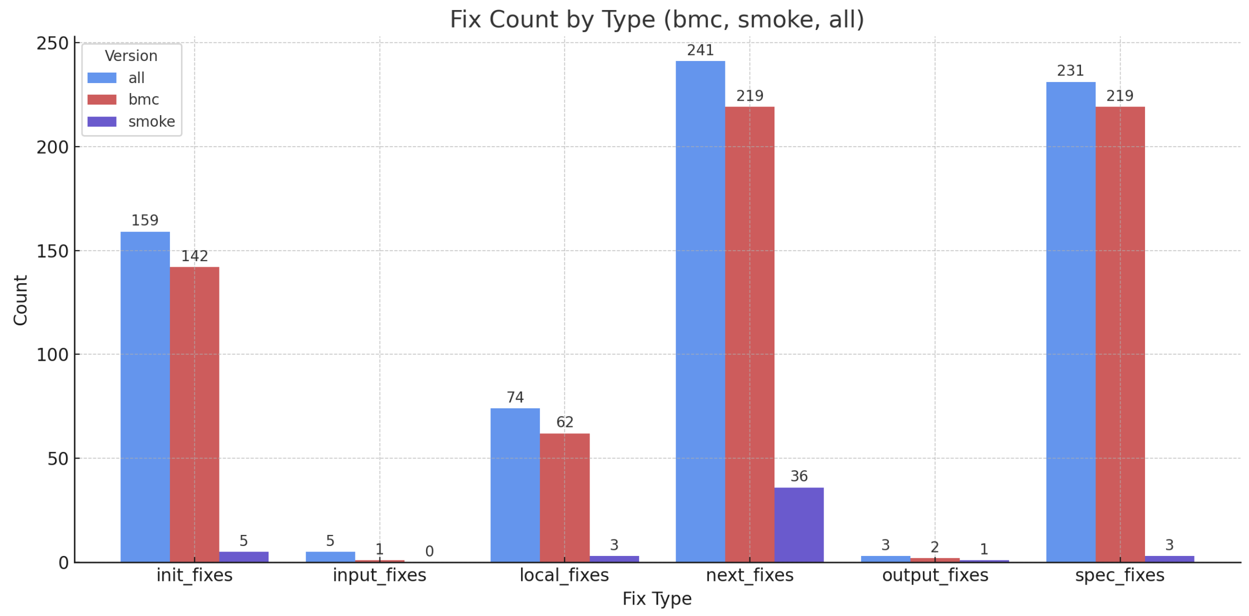


Figure 3.5: Graph of counter LLM-identified faulty sections

RQ4B: Semantically Deficient Sections - Breakdown

To better understand the source of the identified issues, we further dissected the faulty section graph described previously, categorizing the feedback by output version. It becomes immediately apparent that the bmc version is responsible for the majority of the generated feedback. Both the "all" and "bmc" versions exhibit the highest counts across all feedback categories. This trend reinforces the conclusion that the bmc pipeline is the primary contributor to these issues. Given that the "all" version also relies on bounded model checking, its elevated feedback count further supports the claim that **BMC is the key source of error detections and subsequent fix suggestions in the formal-method–backed pipelines.**

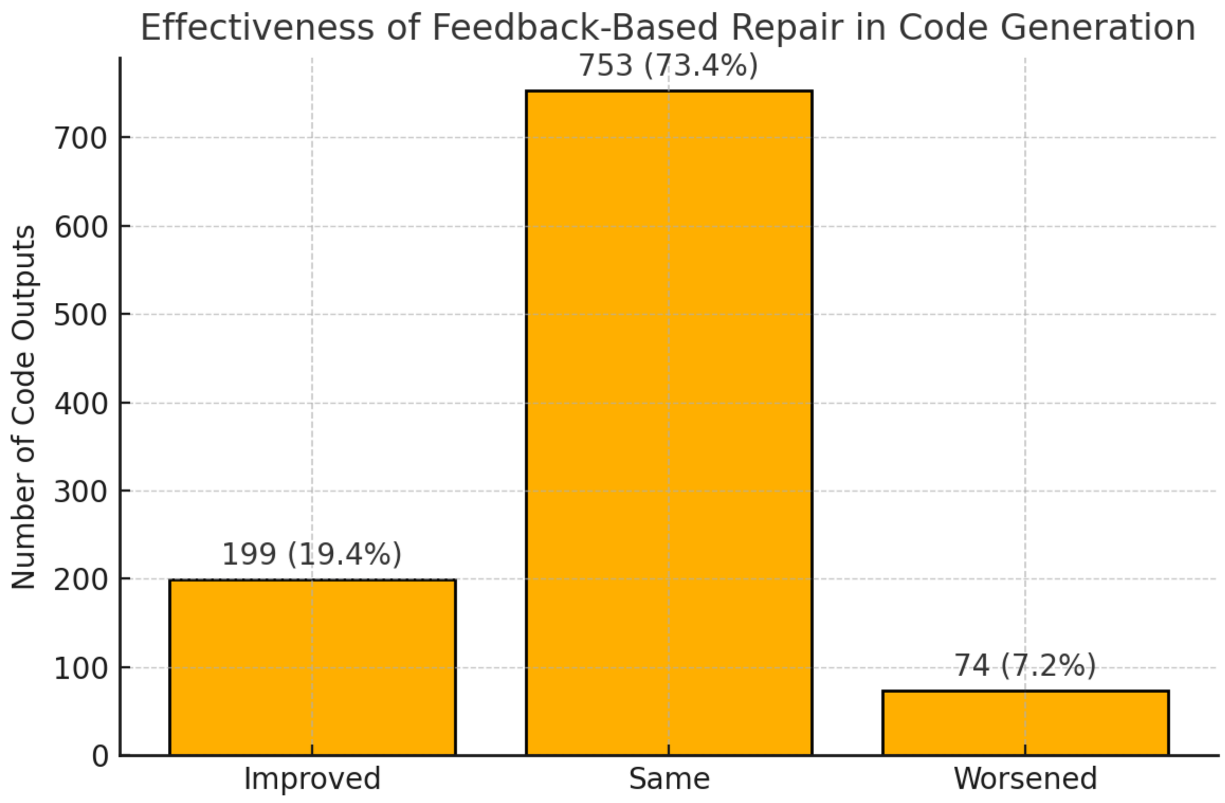


Figure 3.6: Histogram showcasing the delta between iterations in number of failures per assertion.

RQ5: Effectiveness of Feedback

The formal-method–backed pipeline integrates a feedback mechanism designed to modify the program logic in response to failed assertions, with the goal of improving overall correctness. Our evaluation indicates that this feedback component is generally stable and reliable. In 93% of cases, its application resulted in either an improvement or no change in the number of failed assertions. Notably, only 7% of cases exhibited a regression, wherein the number of errors increased after feedback was applied. **These results suggest that the integration of formal-method feedback into the code generation process can be done safely in the vast majority of scenarios, making it a viable component for further development and refinement.**

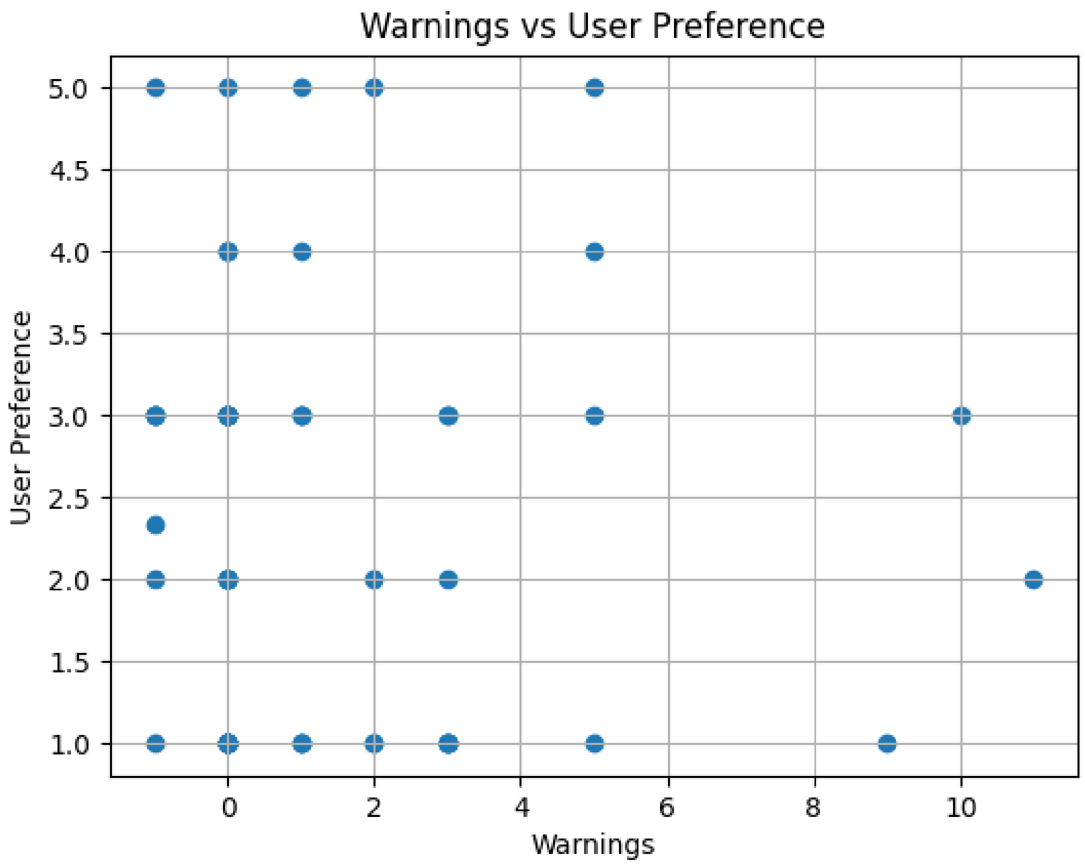


Figure 3.7: Warnings vs User Rating

RQ6: Unreachable Lines vs Rating

We hypothesized that there would be a strong negative correlation between the number of unreachable lines in the generated code and its overall desirability, as rated by participants. Intuitively, the presence of unreachable code might suggest lower quality or lack of coherence, potentially making the output less appealing. However, upon analyzing the data, we observed only a marginal negative correlation. **The computed Pearson correlation coefficient was -0.02, with a p-value of 0.84—indicating no statistically significant relationship between these two variables.**

Threats to Validity

Our experiment gave us valuable insights into the features that were important and effective. Answering each of the research questions provided a new insight into understanding how our new pipeline was perceived by UCLID5 users and developers.
RQ1: We note that there is no substantial difference in user preference between outputs generated with formal methods and no formal methods.
RQ2: On further investigation we see that 'bmc' output preferences are quite polarized. We recognize that these preferences are subjective and running this evaluation on a new set of participants can cause different outcomes.
RQ3: We analyze the assertion pass percentage versus the user preference ratings and find that there is no correlation between the assertion pass percentage for an output file and a participant's preference.
It is worth noting that not all BMC invocations produced counterexamples; a subset failed during execution due to toolchain or model-specific issues. Nevertheless, the majority of runs did yield counterexamples. A detailed breakdown of these cases is provided in the Appendix. Additionally, we include supplementary analyses that explore the relationship between pass percentage and user rating, disaggregated by pipeline type—specifically, the "all" and "bmc" versions (see Figures 5.7 and 5.8 in the Appendix).
RQ4: The error message parsing and feedback generation was a critical component of our work, and we sought to understand what blocks the LLM thoughts the errors were coming from. We noted that most of these errors were identified when running BMC, specifically in the 'next' and 'specification' blocks.
RQ5: With the error parsed and feedback generated, we wanted to understand how effective our repair step is. We found that the feedback component is stable, with only 93% of cases having either an improvement or no change in the number of failed assertions.
However, Figure 3.6 only reflects the situations where BMC executes without failure for two consecutive invocations. There are some scenarios where bounded model checking encounters an error, whether that is a syntax or parser error, which causes bounded model checking to crash. We could not pinpoint the exact cause of those failures, so they are not considered in the "worsened" category. In addition, between iterations of the pipeline, it is possible for the name of the specification to change. We only count assertions that persist with the same name from the previous iteration. These invariant names are created from the variables defined in the model, which generally meant that invariants would not change as long as the variable were named the same. Unfortunately, with coverage-based specifications, we did not have a way to map "similar" types of specifications, only exact name duplicates.
RQ6: Our smoke testing results showed that unreachable lines are not a strong indicator of user preference. Rather, smoke testing should be implemented as a lightweight sanity check on the generated models.

Chapter 4

Conclusion

4.1 Conclusion

This work explored the use of formal methods to improve the quality of code generated by large language models (LLMs), specifically within the Eudoxus pipeline targeting UCLID5 modules. We implemented and evaluated four pipeline variants: a baseline version without formal methods (plain), specification generation with bounded model checking (bmc), smoke testing (smoke), and a combination of specification generation with both bounded model checking and smoke testing (all). The goal was to understand how formal-method–backed components affect user preferences, code desirability, and overall model performance.

Our bracket-style comparison revealed no significant preference for the formal method augmented versions over the baseline. Both the plain and all versions won an equal number of brackets, with the all version reaching the final round slightly more often, suggesting only marginal improvement when integrating formal techniques. Interestingly, while the bmc version performed the worst in overall preference, it showed a polarized desirability pattern—receiving both the highest number of "I would keep all of this code" and one of the highest counts of "I would keep very little or none of this code." This suggests the bmc outputs were either quite strong or significantly flawed.

When we examined formal metrics, we found no meaningful correlation between user preference and model correctness indicators. The percentage of passed assertions from bounded model checking had a Pearson correlation of 0.05 ($p = 0.6050$) with user ratings, while the number of unreachable lines in the code showed a Pearson correlation of -0.02 ($p = 0.84$), indicating no statistically significant relationship. These results suggest that conventional verification metrics alone do not reliably predict user satisfaction or perceived code quality.

Our analysis of repair feedback further highlighted that most fixes were concentrated in the next block, as expected given its role in defining system behavior. The specification block, however, received the second-highest number of repairs, underscoring issues with the specification generation process. The 'bmc' and 'all' versions were responsible for the bulk of these fixes, confirming that bounded model checking, while powerful, introduces instability if not paired with high-quality specifications.

Nevertheless, the repair process proved to be relatively stable. In 93% of cases, feedback-driven fixes either improved or maintained assertion outcomes, while only 7% resulted in regression. This suggests that formal-method–based feedback mechanisms can be safely integrated into generation pipelines in most cases, making them a reliable foundation for iterative improvement.

Ultimately, this study demonstrates that while formal methods can contribute valuable insights and structural rigor to LLM-based code generation, their effectiveness hinges on the quality of the specifications they are guided by. Simply achieving coverage is not sufficient—semantics matter. This insight motivates several directions for future work.

4.2 Future Work

Evaluation

This study primarily focused on user preferences as a proxy for evaluating the quality of generated code. While this provides valuable insight into subjective usability and readability, future iterations of this work should incorporate more rigorous, objective evaluation metrics. These might include expert annotation of correctness, formal validation against intended behavior, or comparison against known gold-standard outputs. Such an approach would offer a deeper understanding of how each pipeline version performs in practice, though it would require a significantly more involved evaluation process.

Specification Generation

Our current approach to specification generation was rooted in coverage-based methods, ensuring that each variable had a corresponding assertion. However, our findings suggest that this approach is insufficient. Many generated specifications failed to produce meaningful verification constraints or improve user-rated outcomes. To address this, we propose shifting toward more semantically rich specifications—such as those based in Linear Temporal Logic (LTL)—which can express how variables evolve over time and how system states interact. This would allow us to better align specifications with real behavioral expectations and reduce the generation of spurious counterexamples.

Bibliography

[1] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. *AlphaVerus: Bootstrapping Formally Verified Code Generation through Self-Improving Translation and Treefinement.* 2024. arXiv: `2412.06176 [cs.LG]`. URL: `https://arxiv.org/abs/2412.06176`.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series).* The MIT Press, 2008. ISBN: 026202649X.

[3] Clark Barrett et al. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability.* Ed. by Armin Biere et al. Second. IOS Press, 2021. Chap. 33, pp. 1267–1329.

[4] Sahil Bhatia et al. "Verified Code Transpilation with LLMs". In: *Advances in Neural Information Processing Systems.* Ed. by A. Globerson et al. Vol. 37. Curran Associates, Inc., 2024, pp. 41394–41424. URL: `https://proceedings.neurips.cc/paper_files/paper/2024/file/48bb60a0c0aebb4142bf314bd1a5c6a0-Paper-Conference.pdf`.

[5] Randal E. Bryant, Shuvendu K. Lahiri, and Sanjit A. Seshia. "Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions". In: *CAV02.* Ed. by E. Brinksma and K. G. Larsen. LNCS 2404. July 2002, pp. 78–92.

[6] Mark Chen et al. *Evaluating Large Language Models Trained on Code.* 2021. arXiv: `2107.03374 [cs.LG]`. URL: `https://arxiv.org/abs/2107.03374`.

[7] QiHong Chen et al. *A Deep Dive Into Large Language Model Code Generation Mistakes: What and Why?* 2025. arXiv: `2411.01414 [cs.SE]`. URL: `https://arxiv.org/abs/2411.01414`.

[8] Tristan Coignion, Clément Quinton, and Romain Rouvoy. "A Performance Study of LLM-Generated Code on Leetcode". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering.* EASE 2024. ACM, June 2024, pp. 79–89. DOI: `10.1145/3661167.3661221`. URL: `http://dx.doi.org/10.1145/3661167.3661221`.

[9] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems.* USA: Cambridge University Press, 2004. ISBN: 052154310X.

[10] Juyong Jiang et al. *A Survey on Large Language Models for Code Generation.* 2024. arXiv: `2406.00515 [cs.CL]`. URL: `https://arxiv.org/abs/2406.00515`.

[11] Edward A. Lee and Sanjit A. Seshia. "An introductory textbook on cyber-physical systems". In: *WESE*. ACM, 2010, p. 1.

[12] Raymond Li et al. *StarCoder: may the source be with you!* 2023. arXiv: 2305.06161 [cs.CL]. URL: https://arxiv.org/abs/2305.06161.

[13] Fang Liu et al. *Exploring and Evaluating Hallucinations in LLM-Powered Code Generation*. 2024. arXiv: 2404.00971 [cs.SE]. URL: https://arxiv.org/abs/2404.00971.

[14] Anton Lozhkov et al. *StarCoder 2 and The Stack v2: The Next Generation*. 2024. arXiv: 2402.19173 [cs.SE]. URL: https://arxiv.org/abs/2402.19173.

[15] Rafael Menezes et al. "ESBMC 7.4: Harnessing the Power of Intervals". In: 30th *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'24)*. Vol. 14572. Lecture Notes in Computer Science. Springer, 2024, pp. 376–380. DOI: https://doi.org/10.1007/978-3-031-57256-2_24.

[16] Federico Mora et al. *Synthetic Programming Elicitation for Text-to-Code in Very Low-Resource Programming and Formal Languages*. 2024. arXiv: 2406.03636 [cs.PL]. URL: https://arxiv.org/abs/2406.03636.

[17] Elizabeth Polgreen et al. *UCLID5: Multi-Modal Formal Modeling, Verification, and Synthesis*. 2022. arXiv: 2208.03699 [cs.LO]. URL: https://arxiv.org/abs/2208.03699.

[18] Elizabeth Polgreen et al. "UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis". In: *CAV (1)*. Vol. 13371. Lecture Notes in Computer Science. Springer, 2022, pp. 538–551.

[19] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL]. URL: https://arxiv.org/abs/2308.12950.

[20] Sanjit A. Seshia and Pramod Subramanyan. "UCLID5: Integrating Modeling, Verification, Synthesis and Learning". In: *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*. 2018, pp. 1–10. DOI: 10.1109/MEMCOD.2018.8556946.

[21] Merlijn Sevenhuijsen, Khashayar Etemadi, and Mattias Nyberg. *VeCoGen: Automating Generation of Formally Verified C Code with Large Language Models*. 2025. arXiv: 2411.19275 [cs.SE]. URL: https://arxiv.org/abs/2411.19275.

[22] Yue Wang et al. *CodeT5+: Open Code Large Language Models for Code Understanding and Generation*. 2023. arXiv: 2305.07922 [cs.CL]. URL: https://arxiv.org/abs/2305.07922.

[23] Guangyuan Wu et al. "LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 406–417. ISBN: 9798400712487. DOI: 10.1145/3691620.3695014. URL: https://doi.org/10.1145/3691620.3695014.

[24] Yanxuan Zhang et al. “Language (Model) is Not Enough: Aligning LMs with Constrained Decoding”. In: *International Conference on Learning Representations (ICLR)*. 2023.

Chapter 5

Appendix

5.1 Initial User Study

Experiment Setup

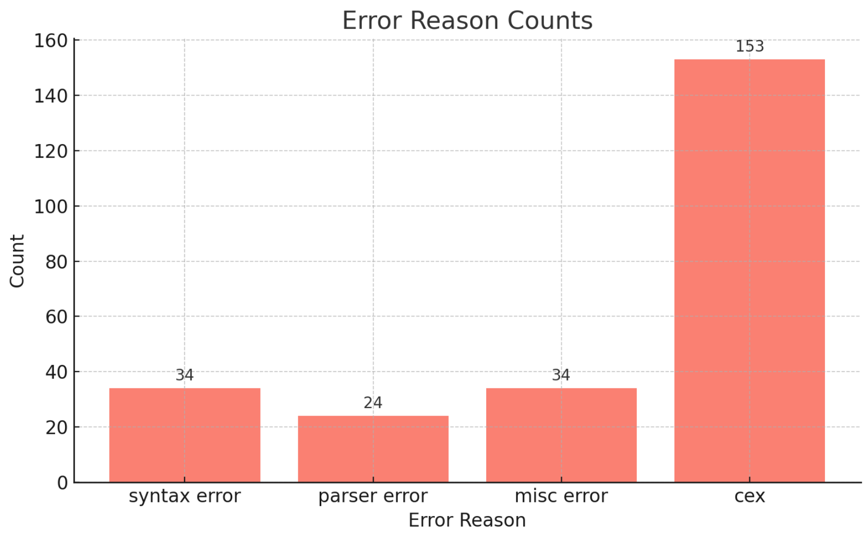
To conduct the experiment, we first anonymized the model outputs by assigning each output a unique identifier, maintaining a mapping between these identifiers and their corresponding versions.

In our initial user study, participants were asked to rank their top three choices from among five available outputs. However, user feedback indicated that this approach posed significant challenges. Specifically, participants found it difficult to evaluate outputs accurately without a clear rubric, making it hard to distinguish and qualify differences between them.

In response to this feedback, we revised the study design by reducing the number of outputs from five to four and implementing a more structured comparison system. This modification aimed to alleviate cognitive load and enable participants to make more informed and precise comparisons between outputs.

5.2 BMC Failure Cases

In every invocation of bounded model checking, our goal is to generate counterexamples if the logic is incorrect. The counterexamples are parsed and used to generate feedback. Sometimes in the invocation of bounded model checking, we found syntax and parsing errors. The graph below shows the breakdown of the errors that we encountered during this experiment. Fixing these issues is also part of the next study.



5.3 Prompts

Generate Specifications:

To generate specifications, we used a combination of prompt engineering and constrained decoding. We provide both the task semantics and current code and ask for an invariant for each variable.

Generate BMC Error / Smoke Testing Fixes:

To generate the fixes from the BMC counter examples, we provide our LLM the original task description, generated python model, and counterexample message. We query the LLM, asking for it to generate a fix for the problematic function in our current model. We add a section guiding the LLM towards either the 'next' block or 'init' block based on where most of the errors occurred.

LLM Generated BMC bound: We query an LLM to generate a bmc bound. We provide the LLM the natural language task description and the current python model. We ask the LLM to return the number of iterations we should run the 'next' block to best ensure the correctness of the code.

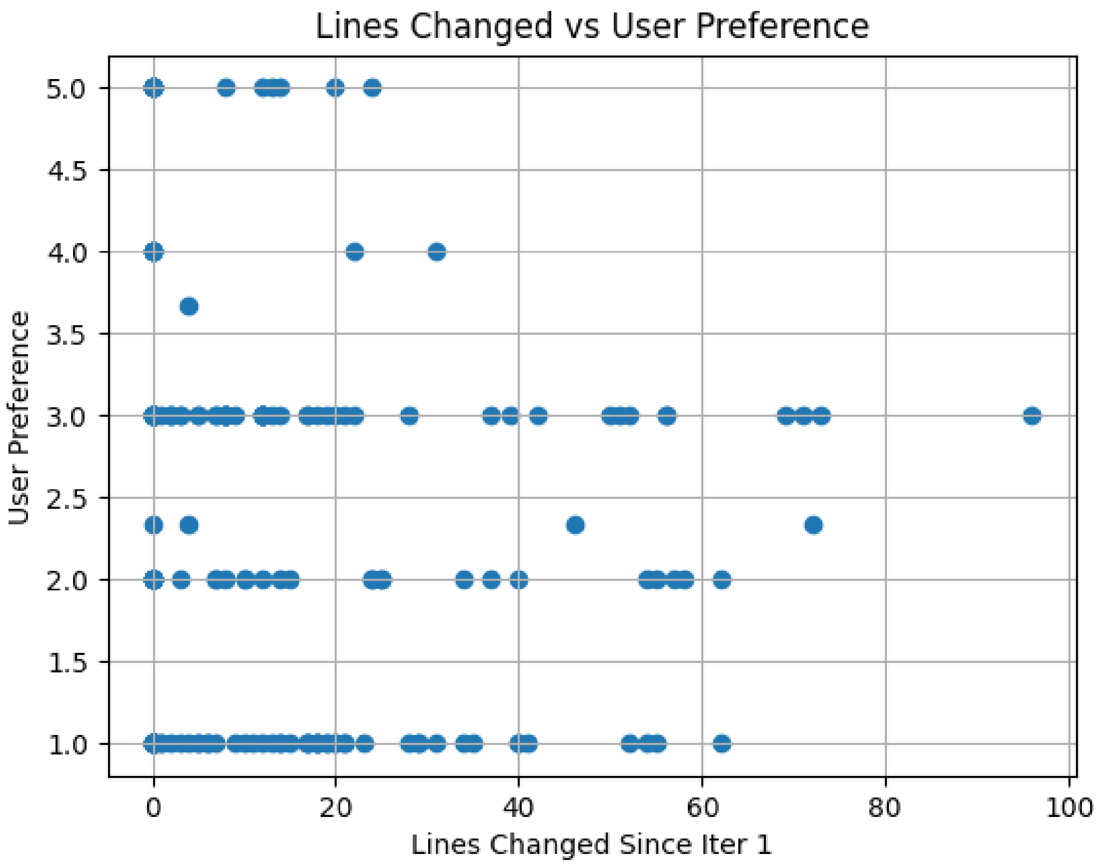


Figure 5.1: Graph showing the relationship between lines changed from initial code completion and repair to final output.

5.4 Additional Research Questions

RQ: Changed Lines vs User Rating

Is there a correlation between the number of changed lines and user rating?
We found that an increase in changed lines generally lead to a lower participant preference score. **This implies that the code generated in the first few iterations is probably more aligned to the user's interests.**
Pearson correlation: -0.07, p-value: 0.2960

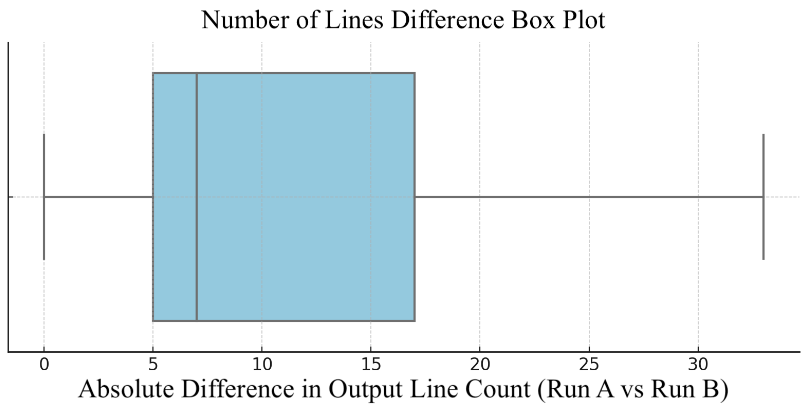


Figure 5.2: A box plot showing the variability in the number of lines between two different pipeline runs with the exact same parameters

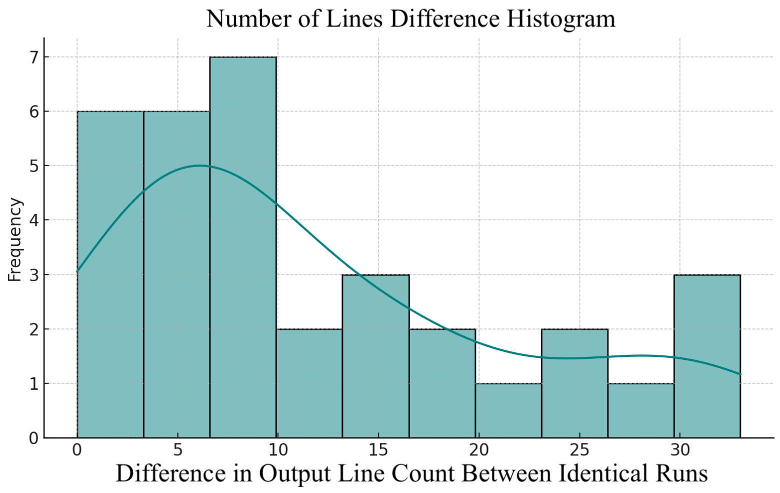


Figure 5.3: Histogram showcasing the variability in the user rating

RQ: Output Variabilty (Lines)

How much variation (# lines) is there between outputs created with the same parameters?

It is clear and evident that large language model outputs will not be consistent. We constrained that variability through constrained decoding; however, there were only a few cases where the model outputs were the same. The two graphs above depict the absolute value difference in lines from the output of the first and second run of the model with the same parameters.

We list the specific stats here: Mean: 11.85, Median: 7, Standard Deviation: 9.77, Min: 0, Max: 33.

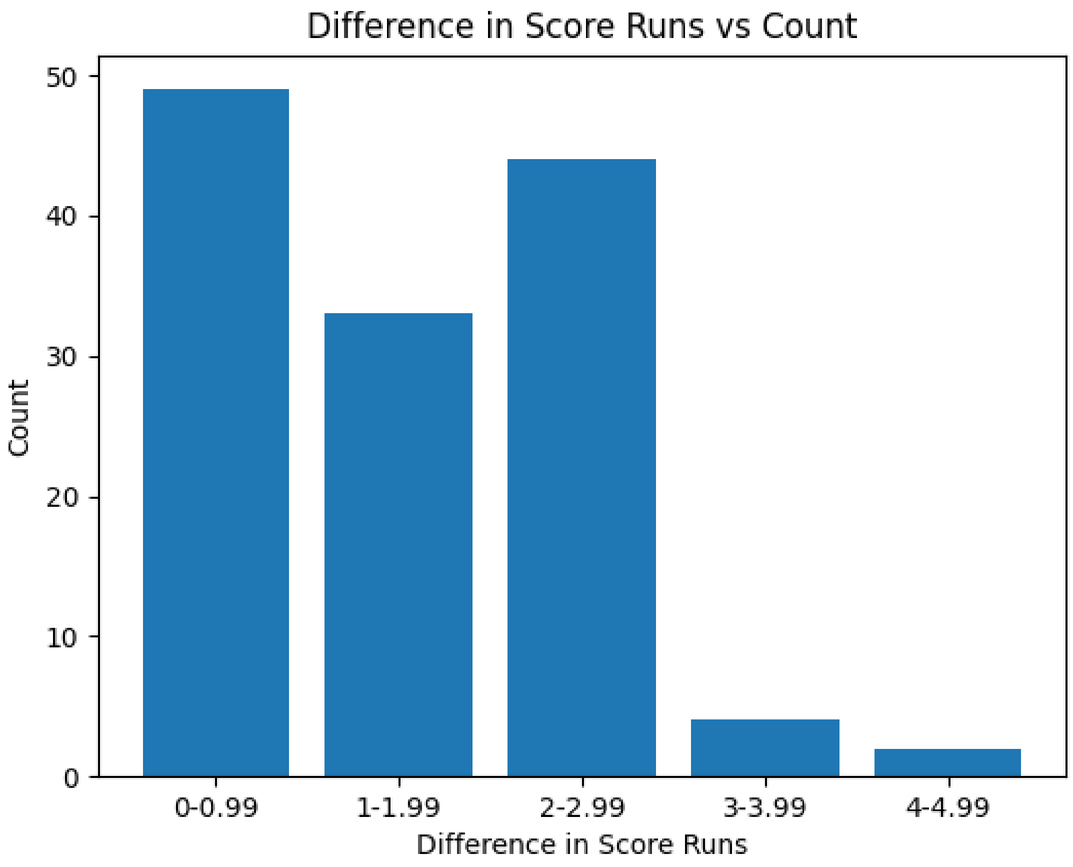


Figure 5.4: Histogram of user variability between two different pipeline runs with the exact same parameters

RQ: Output Variabilty (User Rating)

How much variation (user rating) is there between outputs created with the same parameters?
We knew that running the same command would yield different outputs and wanted to see what the difference in user ratings for these outputs across the two runs was. We see that for the most part the ratings are concentrated on the left hand side, implying that the ratings generally stay in the same category. **However, there are a considerable amount of differences between files that fall in the 2-2.99 category. A difference of 2 is an entire category difference which is notable.**

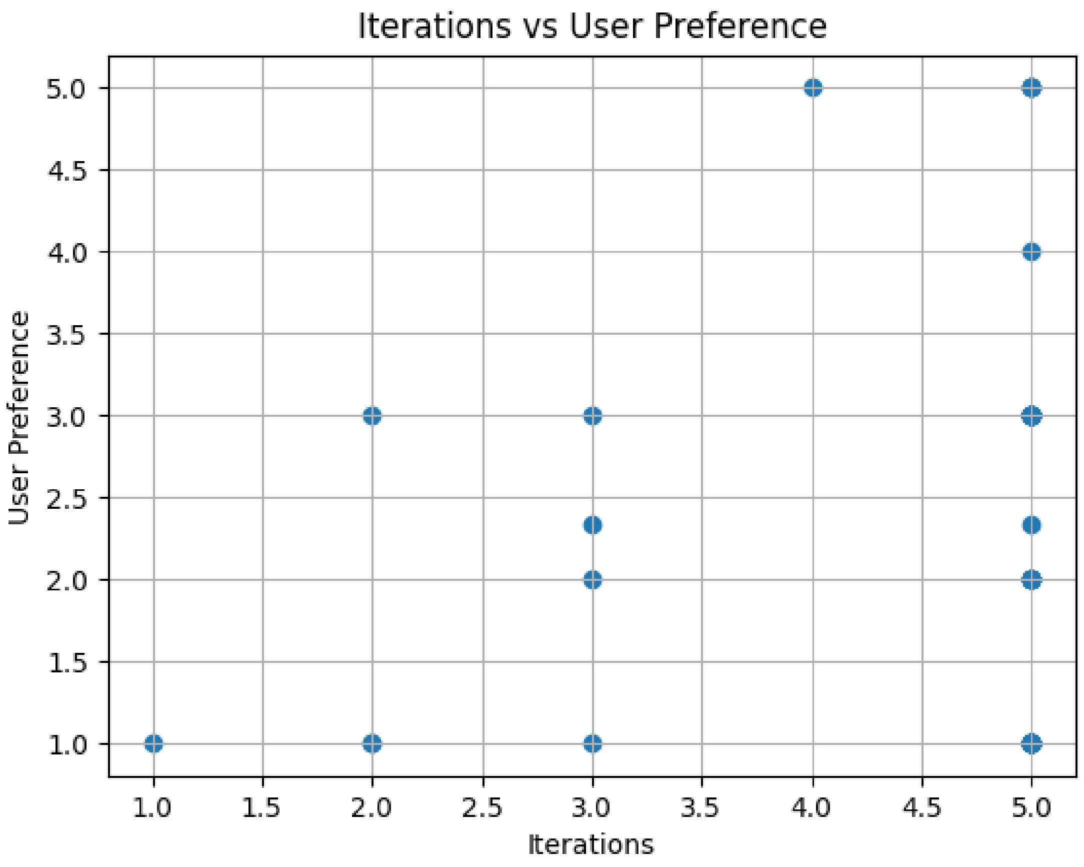


Figure 5.5: Iterations vs User Rating

RQ: Iterations vs Rating

Is there a correlation between number of iterations run and user rating?
In this section, we plotted to see if there was a correlation between the number of iterations the feedback loop ran for and user preference for the code. We calculated a Pearson correlation score of 0.13 with p-value 0.31. We find a slight positive, but not statistically significant, correlation.

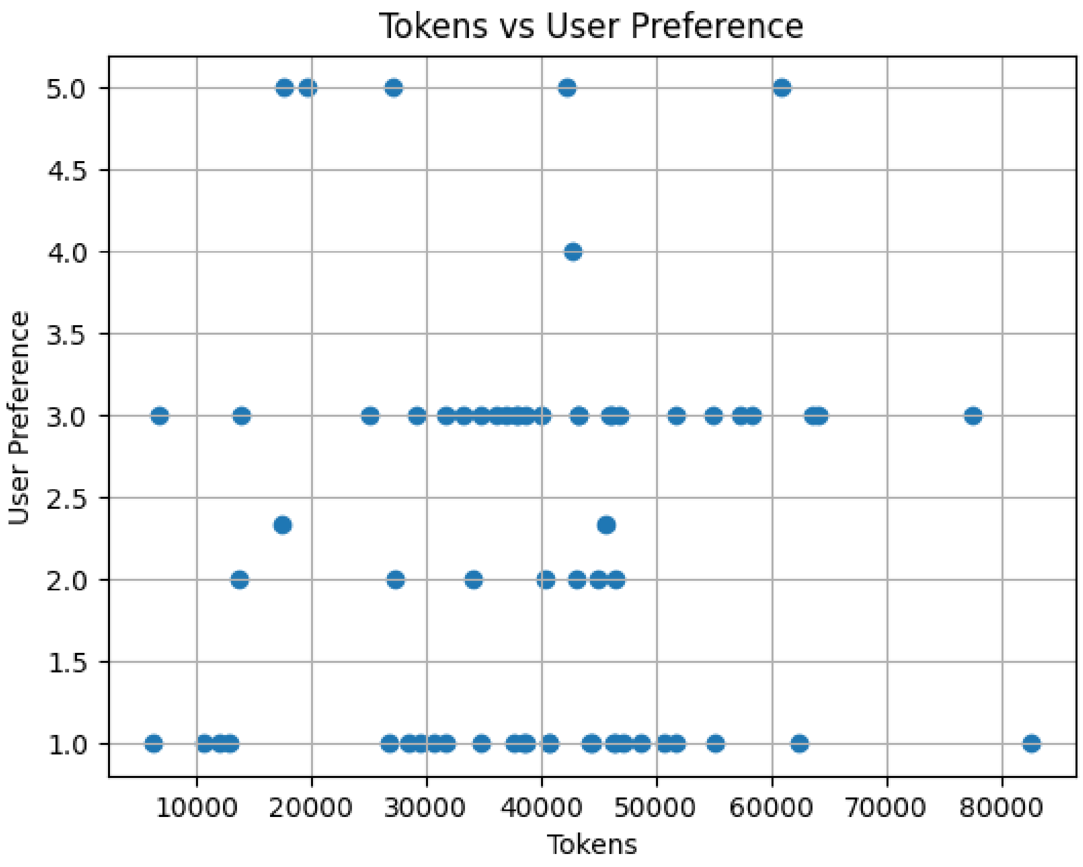


Figure 5.6: Tokens vs User Rating

RQ: Tokens vs Rating

Is there a correlation between number of tokens used and user rating?
In this section, we plotted to see if there was a correlation between the number of tokens used and user preference for the code. We calculated a Pearson correlation score of 0.01 with a p-value of 0.93. **So, there is no correlation and it is not statistically significant.**

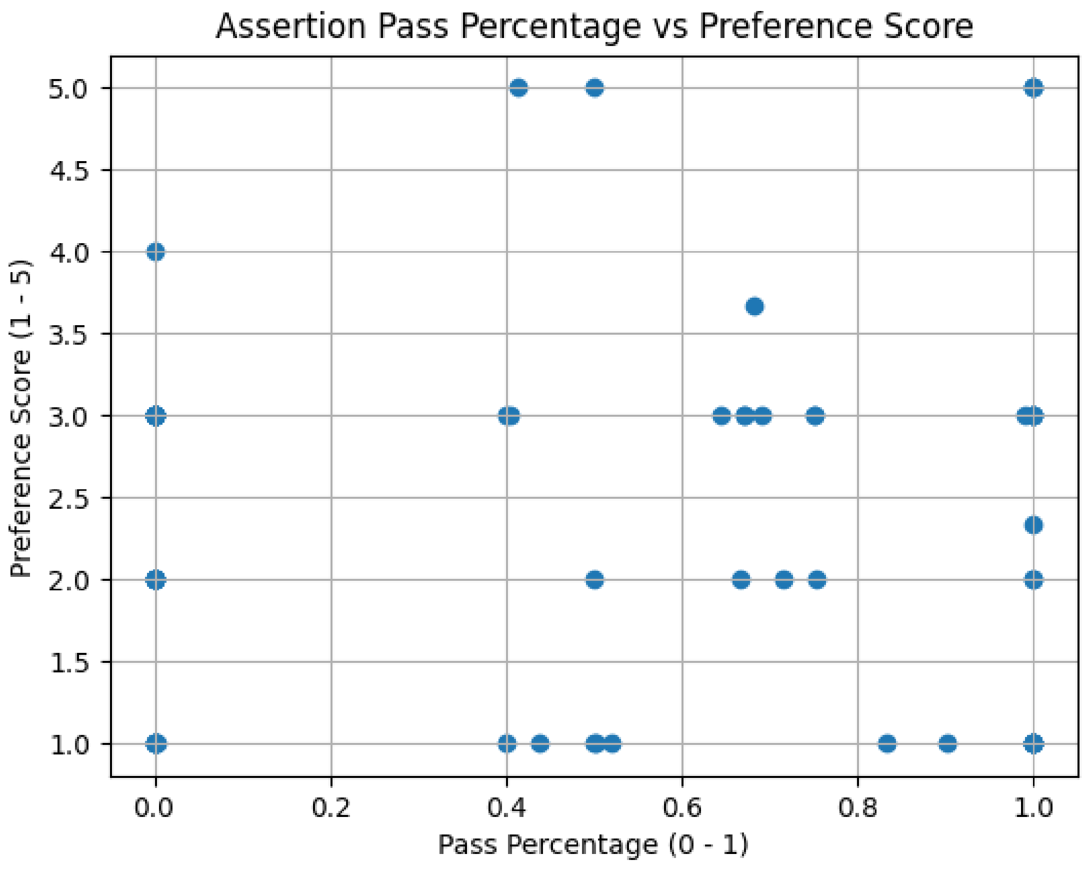


Figure 5.7: Assertion Pass Percentage vs User Rating

RQ: Passed Assertions vs User Rating (bmc)

On all of the final "bmc" outputs, we took the best model generated and ran bounded model checking on it for 100 iterations, noting how many passed and failed. The graph above shows the "pass-percentage" of the models vs user rating. Our Pearson correlation value is 0.02 with p-value of 0.86. **The results imply that there is almost no correlation between the number of assertions passed and user preference. This means that there are persistent problems in the specification or logic.**

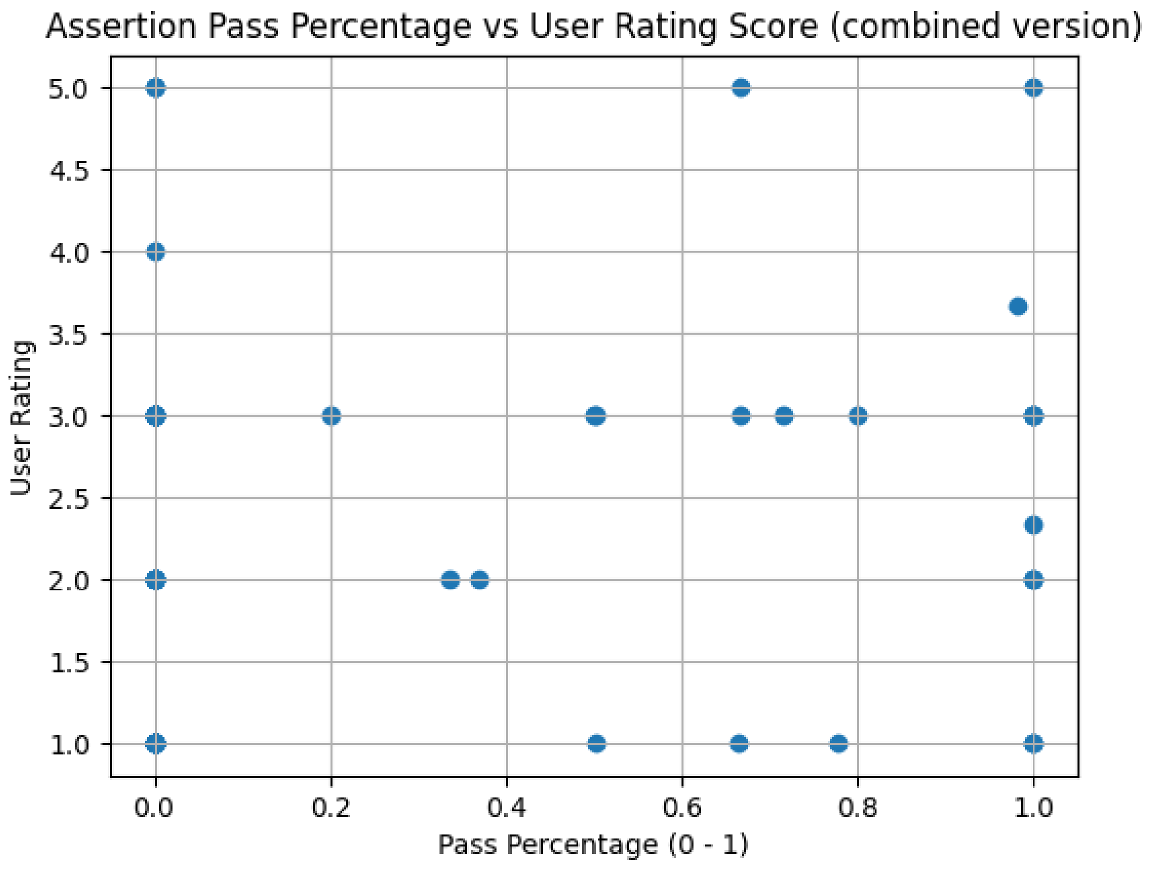


Figure 5.8: Assertion Pass Percentage vs User Rating for models generated by the pipeline with both specification generation with bmc and smoke testing.

RQ: Passed Assertions vs User Rating (combined)

On all of the final ”all” outputs, we took the best model generated and ran bounded model checking on it for 100 iterations, noting how many passed and failed. The graph above shows the ”pass-percentage” of the models vs user rating. Our Pearson correlation value is 0.07 with p-value of 0.568. **We find the correlation to be higher in this version versus just the bmc version-implying that the addition of smoke testing might be helping with the model logic.**