Assessing Postcondition Completeness in Auto-Generated Function Contracts for Software Verification

Caio Vinícius Pereira Silveira
DECOM, CEFET-MG
Belo Horizonte, Brazil
caiovpsilveira@gmail.com

Mateus Felipe Tymburibá Ferreira
DECOM, CEFET-MG
Belo Horizonte, Brazil
mateustymbu@cefetmg.br

Abstract

The precision of specifications plays a crucial role in system verification, with correctness and completeness being complementary properties required for soundness. Previous approaches for automated software verification focused mainly on evaluating correctness, overlooking or only partially addressing the specification completeness. Recent work has renewed interest in completeness evaluation, but existing approaches remain critically dependent on test suites, effectively shifting the challenge of assessing the coverage of the specification to the coverage of the tests. To overcome this limitation, we introduce a novel method for checking postcondition completeness that can evaluate whether a specification is the Strongest Postcondition, requiring only the candidate specification for a deterministic program. To assess its feasibility, we evaluated our completeness check across representative LLM-based specification generation approaches, demonstrating its effectiveness, reliability, and potential for automated verification.

Keywords

Specification Completeness, Strongest Postcondition, Specification Coverage, Program Verification, Formal Specification, Large Language Models.

1 Introduction

Formal verification establishes program correctness through formal methods, providing strong guarantees about functional behavior. However, writing the required contracts is a demanding task, especially for developers unfamiliar with formal verification [15]. Keeping requirements, specifications, and test suites synchronized as systems evolve also introduces substantial overhead [21]. These challenges have motivated extensive work on automated specification generation, including template-based generation [18], dynamic invariant mining [17], strongest-postcondition predicate transformers [34], neural networks [42], and fuzzing [29]. More recently, large language models (LLMs) have been explored for specification inference [27, 38, 39, 41], alongside growing efforts to verify AI-generated code to prevent bugs or vulnerabilities [37]. Despite the diversity of existing approaches, nearly all have focused on producing specifications that can be verified correct, while overlooking or only partially addressing the specification coverage (also called *completeness*).

Consider the example shown in Listing 1. This specification partially captures the behavior of *abs*, as it requires the result to be non-negative, but not equal to the absolute value of *x*. Low specification coverage can hide functional bugs because required behaviors may remain unchecked by the verifier. It can also limit the information available to callers, preventing the verifier from proving the desired properties. Thus, ensuring a complete specification is crucial for automated software verification, as incomplete specifications can reduce both bug-finding effectiveness and proof scalability.

```
1  //@ requires x != Integer.MIN_VALUE; // neg overflow
2  //@ ensures \result >= 0;
3  public static int abs(int x) {
4      return x > 0 ? x : -x;
5  }
```

Listing 1: Correct but incomplete specification of the *abs* function. Any implementation that returns a non-negative value, such as one that always returns 0, satisfies it.

Endres et al. [16] formalized correctness and completeness as complementary specification-quality metrics. Their completeness evaluation relies on mutated implementations and a test suite assumed to cover all functional behavior, making the assessment dependent on both mutant selection and test adequacy. However, generating tests with exhaustive structural and functional coverage is inherently complex and computationally expensive, as illustrated by approaches based on evolutionary algorithms [11] and mutation testing [1]. Consequently, this approach may be impractical and unreliable, as it effectively shifts the challenge of assessing specification completeness to the completeness of the test suite itself.

To address this limitation, we introduce a novel, self-contained, lightweight, and reliable method to check postcondition completeness. Our approach uses Strongest Postcondition semantics and requires only the specification of a deterministic program. By eliminating the need for oracle implementations or test suites, our method represents an important step towards autonomous specification generation.

To assess the feasibility of our check, we evaluated three LLM-based generation approaches: a conversational repair loop, the SpecGen workflow [27], and a conversational loop augmented with a closed-loop completeness feedback. Our contributions are as follows.

- We present a novel postcondition completeness checking method that is reliable, lightweight, and self-contained, requiring only the candidate specification of a deterministic program (Section 3).
- We assessed the FormalBench [24] mutation-based completeness evaluation on known complete and incomplete specifications, and identified that false positives arise from insufficient mutant coverage and that false negatives occur even with Equivalent Mutant Suppression [23] (Section 6.1).
- We extend the evaluation of SpecGen [27], showing that its mutation repair can trade completeness for correctness (Section 6.2).
- We evaluated an LLM conversational technique enriched with our completeness check and observed an effectiveness of up to

50.0% in improving completeness, contributing up to 10.7% of the total complete successes (Section 6.3).

2 Related Work and Motivation

Previous studies on automated function contract generation [18, 27, 34, 38] focused on generating correct specifications and on techniques to increase the success rate. However, until recently, there was no widely recognized method to assess the coverage of specifications. This led prior studies to rely on human evaluation [27] or heuristics such as length and keyword presence [34]. Other lines of work focused on the generation of loop invariants [39, 41] which were required to prove an already available top-level specification. For this task, the generated loop invariants only need to convey the information necessary to prove the desired properties, making the evaluation more straightforward.

Traditional testing techniques, such as unit testing, end-to-end testing, and regression testing, struggle to exhaustively cover all possible scenarios. Formal methods address this limitation by using static mathematical reasoning to verify program behavior over all execution paths [12]. However, formal methods are not a silver bullet in system verification, and the weakness of such approaches lies precisely in the verification of incomplete or incorrectly stated specifications [7]. If the desired behavior is not properly stated, the specification might have gaps where functional bugs are allowed. Unlike runtime errors, such as nullptr dereference or arithmetic overflows, functional bugs are much harder to catch with static analyzers, as they are valid programs that do not violate any of the language rules [40]. This highlights the need for complete specifications for software verification.

From a scalability perspective, a weak postcondition can be problematic as it can be too imprecise to convey necessary information for its callers. An example is shown in Listing 2: the *abs* postcondition is insufficient to prove that the caller avoids division by zero. Note that even though this might be visible in the body, verifiers (OpenJML, Frama-C/WP, Dafny) use only the specification when verifying the caller, ensuring a modular approach to verification [8, 13].

```
1  // Correct, but weak postcondition:
2  //@ requires x != Integer.MIN_VALUE; // neg overflow
3  //@ ensures \result >= 0;
4  public static int abs(int x) { ... }
5
6  public static int foo(int x) {
7      if (x <= 0) return 0;
8      int abs = abs(x);
9      // Verifier cannot prove non-division by 0.
10     // abs postcondition is insufficient
11     return 1 / abs;
12 }
```

Listing 2: Proof failure in *foo* due to a weak spec of *abs*. Postcondition does not convey that the result cannot be 0 for x > 0.

Endres et al. [16] explored this gap and proposed *correctness* and *completeness* metrics to evaluate the quality of software specification contracts. Their evaluation resorted to runtime assertions rather than formal model checkers, but the concepts can be extended to formal contract verification.

Correctness is a metric that evaluates whether the specification and implementation are in accordance. The idea was that if the tests cover all valid inputs (as constrained by the precondition), one could measure correctness by evaluating whether the postcondition holds after each execution. In contrast, formal verifiers such as OpenJML, Frama-C/WP and Dafny use Formal Methods to reason about every constrained input, verifying that all possible executions satisfy the contract. The specification should also constrain unexpected inputs, an aspect that was overlooked in Endres et al. evaluation, but later addressed in [33] and [28].

Completeness is a metric that evaluates how much functional behavior is covered by the specification. Endres et al. [16] measured this by evaluating the postconditions against a test suite with mutated implementations. The idea is that if there is a program mutated to differ with respect to each functional behavior of the original implementation, a complete specification would flag all mutants as incorrect, as it describes all expected (un-mutated) behaviors. Based on our literature review, the use of mutation to evaluate specification coverage was first introduced by Ammann and Black [2]. Despite this early work, this method has received limited attention until recently. In addition to Endres et al. [16], this method has also been used by Richter and Wehrheim [33], VeriAct [28], FormalBench [24], and AutoReSpec [5].

Despite its widespread use in recent work, mutation-based completeness evaluation remains limited by the selected mutants. Generated mutants are not guaranteed to cover all possible behaviors, since their effectiveness depends heavily on the mutation operators and target task [1, 2, 22, 36]. This dependence makes mutation-based coverage difficult to generalize across different environments. Mutants are also not equally useful or representative [22], which can lead to inaccurate coverage scores. The presence of equivalent mutants also affects the accuracy and can lead to false negatives, representing a recognized obstacle to mutation adoption [3, 9, 23]. Overall, constructing tests that capture all relevant functional behavior without redundancy remains a central challenge of mutation testing and other test-suite based approaches [9]. To address these challenges, we explored an alternative solution using formal model checkers and Strongest Postcondition semantics.

3 Assessing the Specification Quality

To evaluate the quality of auto-generated function contracts, we need to assess both postconditions and preconditions. This section presents the methodology used to evaluate these complementary aspects.

3.1 Evaluating Postcondition Completeness

This section presents our main contribution, a methodology for evaluating postcondition completeness requiring only the specification of a deterministic program. We also describe an empirical evaluation of the proposed approach, used as a proof of concept.

3.1.1 Description. Verification completeness measures how much of a program's behavior is covered by the verification process [4]. A specification can be formally expressed as a precondition P and a postcondition Q, which define the behaviors that a program C

must exhibit. We say that the specification is consistent with the program if the Hoare triple $\{P\}C\{Q\}$ can be proven to hold [20].

The Strongest Postcondition predicate transformer, $SP(P, C)$, characterizes exactly the set of final states reachable by executing the program C from an initial state satisfying P [14]. It is the most refined specification, which contains all and only the reachable final states. Therefore, it is not an over-approximation of the program behavior.

$$SP(P, C) = \{\, s \mid s \text{ is reachable from } (P, C) \,\}$$

In specification inference, the program is used only to infer and validate a candidate specification. Once $\{P\}C\{Q\}$ has been verified, we know that the specification is satisfied by the program. However, this does not guarantee that Q precisely characterizes the program's behavior, as it may be an over-approximation that loosely accepts states that are unreachable from (P, C). A direct way to assess completeness is to compute the reachable states $C(i)$ for every input $i \in P$ and verify that $\forall i \in P, C(i) = Q(i)$. If this holds, then Q corresponds to $SP(P, C)$.

Computing the exact set of reachable states can be difficult using static analysis alone. Instead, we explored a key property of deterministic programs: the same input always produces the same output. Therefore, the SP of a deterministic function associates each valid input with a single reachable output state. Once $\{P\}C\{Q\}$ has been verified, Property 1 is true. Figure 1 illustrates this relationship.

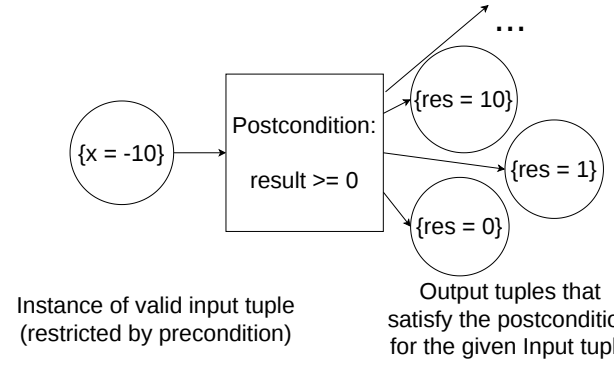
$$\forall i \in P \rightarrow \left|Q_{(i)}\right| = 1 \iff Q \text{ is } SP(P, C) \wedge C \text{ is deterministic} \quad (1)$$

Determining whether a function is deterministic can also be addressed using formal techniques. For example, Mudduluru et al. [30] introduced a type-system approach to express determinism. As determinism is a requirement for the proof to succeed, knowing in advance that the specification describes deterministic behavior allows us to better interpret failed proofs, which must then indicate an incomplete specification or a limitation of the verifier.

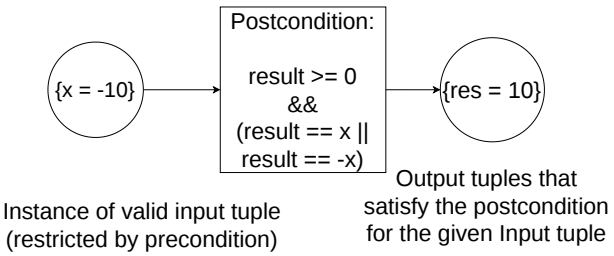
Heap allocation is a common source of nondeterminism. However, if allocated objects are not used in heap-exposing ways, such as object equality comparisons on fresh allocations or using the address as unique keys for operations [13], then this does not affect our approach. We view this form of nondeterminism as an implementation detail rather than a property of the specification.

It is important to note that, due to the challenges of deductive verification, a failed proof does not necessarily imply that the specification is incomplete. A specification formulation may be difficult for the verifier to reason about, often requiring it to be rewritten in ways that are more verifier-friendly [10, 12]. In contrast, Bounded Model Checking (BMC) techniques could be used to search for counterexamples. When a counterexample is found, it provides evidence that the specification is incomplete.

The main advantage of this approach is that, assuming a sound verifier, it cannot produce false positives, i.e., classify an incomplete specification as complete. This follows from the rigor of formal verification: rather than relying on a finite set of tests or generated mutants, the check establishes the one-to-one input-output mapping for all valid inputs. In contrast, mutation-based approaches may produce false positives when their mutation operators fail to expose behaviors omitted by the specification, or false negatives when they generate semantically equivalent mutants.



(a) For deterministic programs, any postcondition that is not the Strongest Postcondition maps at least one input to multiple output states.



(b) For deterministic programs, the Strongest Postcondition maps each input to a single output state.

Figure 1: For deterministic programs, a correct specification is the Strongest Postcondition if and only if every valid input tuple maps to a single output tuple.

3.1.2 Proof-of-Concept Implementation. Our implementation uses OpenJML, a deductive verifier, to reason about all possible inputs, addressing the limitations identified in [16, 33]. As a proof of concept, we implemented the completeness check through an auxiliary method rather than extending the verifier functionality. The method invokes the target function twice from identical initial states and attempts to prove that, for all valid inputs, both executions produce the same observable outputs. The algorithm is shown in Listing 1. The assertions in the auxiliary method encode the left-hand side of Property 1. Listing 3 shows the check applied to the weak *abs* specification discussed earlier.

Our implementation of the check is sound due to the characteristics of the verification setup and the following assumptions:

(1) Method bodies are not considered when proving callers, even if visible. OpenJML (as well as Frama-C/WP and Dafny) reasons about called methods using their specifications [13]. By introducing a clone method, the verifier treats f and f_{clone} as potentially different implementations. If the specification over-approximates the behavior for any valid input, the two calls

Algorithm 1 Pseudocode for automating our completeness check for deterministic programs with OpenJML.

Require: Specification S
Require: function signature SIG : type f(type p_1, type p_2, ...)

Create a clone method with the same spec S, signature SIG, replacing the function name: $f \rightarrow f_{clone}$
Create a void model method M with the same parameters as SIG
In M, assume the precondition of S

Begin M implementation:
parameters_to_call_f = []
parameters_to_call_f_clone = []

for each parameter p_i in SIG **do**
    if p_i is in the frame condition of S **then**
        p_i_dc1 ← deep_copy(p_i)
        p_i_dc2 ← deep_copy(p_i)
        parameters_to_call_f ← p_i_dc1
        parameters_to_call_f_clone ← p_i_dc2
    else
        $parameters\_to\_call\_f \leftarrow p_i$
        $parameters\_to\_call\_f_{clone} \leftarrow p_i$
    end if
end for

if return type is not void **then**
    $r_1 \leftarrow f$(∗parameters_to_call_f)
    $r_2 \leftarrow f_{clone}$(∗parameters_to_call_f_clone)

    if the return value is of reference type and is a fresh heap allocation **then**
        assert $*r_1$ and $*r_2$ are equal
    else
        assert r_1 and r_2 are equal
    end if
else
    f(∗parameters_to_call_f)
    f_{clone}(∗parameters_to_call_f_clone)
end if

for each parameter p_i in the frame condition of S **do**
    if p_i is of reference type and it's a fresh heap allocation **then**
        assert $*p_i$_dc1 and $*p_i$_dc2 are equal
    else
        assert p_i_dc1 and p_i_dc2 are equal
    end if
end for

End M implementation.
Verify M. If all assertions are verified, then S corresponds to the Strongest Postcondition and specifies a deterministic program.

```
1  public class Test {
2      //@ requires x != Integer.MIN_VALUE;
3      //@ assigns \nothing;
4      //@ ensures \result >= 0;
5      public static int abs(int x) {
6          return x > 0 ? x : -x;
7      }
8
9      //@ requires x != Integer.MIN_VALUE;
10     //@ assigns \nothing;
11     //@ ensures \result >= 0;
12     public static int abs_clone(int x) {
13         return x > 0 ? x : -x;
14     }
15
16 //@ requires x != Integer.MIN_VALUE;
17 //@ model public static void test_completeness(int x) {
18 //@     int ret1 = abs(x);
19 //@     int ret2 = abs_clone(x);
20 //@     assert ret1 == ret2;
21 //@ }
22 }
```

Listing 3: Completeness check of weak Abs specification. As the postcondition is incomplete (and so, not the strongest postcondition), the assertion at line 20 cannot be verified.

may therefore return different values, preventing the equality assertions from being proven.

(2) OpenJML requires all side effects to be explicitly captured by frame conditions. Therefore, comparing only the parameters listed in the frame condition (lines 29–35) is sufficient to capture all method outputs.

(3) Specifications are verified for *total correctness*, requiring the implementation to terminate for all valid inputs. Under *partial correctness*, contracts are vacuously satisfied for non-terminating executions [20]. Since JML cannot specify which inputs cause non-termination, partial correctness would prevent us from characterizing a one-to-one input-output mapping.

(4) The deep copies (lines 7–11) ensure that the two calls receive an identical initial state, despite sequential execution. To simplify verification, copies are restricted to the parameters listed in the frame condition. Although copying all parameters would be semantically equivalent, we observed cases where additional copies made the verification task more difficult for the verifier. This step assumes that there is no parameter aliasing in the parameter list, as otherwise the deep copy would incorrectly duplicate aliased objects. This assumption holds for our dataset and could be eliminated with additional checks. These limitations are artifacts of our proof-of-concept encoding and could be cleanly addressed in a built-in verifier implementation.

3.2 Evaluating Precondition Quality

The Strongest Postcondition is a predicate defined relative to both the precondition and postcondition, since it needs to describe behaviors only for the allowed inputs. This makes the precondition another aspect that must be considered in automatic contract inference (as also discussed in [33]). If the precondition *underconstrains* the input space, the postcondition must also generalize the program behavior for unintended inputs, which may make it more difficult to be achieved and verified. On the other hand, if it *overconstrains*

the input space, it may exclude desired behaviors and reject valid callers.

Richter and Wehrheim [33] reported that LLM-inferred preconditions were generally aligned with developer intent. We observed similar behavior in initial generation attempts, but also saw cases where, after struggling to verify the postcondition, the LLM excluded failing behaviors by overconstraining the precondition. Listing 4 shows one example. The function explicitly handles negative inputs, but the generated precondition excludes them. The comments were added by the LLM.

```
/*@
@ // Restrict to non-negative inputs to avoid reasoning
@ // about sign/negation overflow and to
@ // keep verification tractable.
@ requires x >= 0 && y >= 0;
@ // ensure the repeated-addition result fits into an int
@ // (prevents overflow of the algorithm)
@ requires (long)x * (long)y <= Integer.MAX_VALUE;
@ assigns \nothing;
@ ensures \result == x * y;
@*/
public static int mul(int x, int y);
```

Listing 4: Example of overconstrained precondition, generated by GPT-5 Mini. It undesirably excludes negative inputs.

Automatically assessing precondition quality is beyond the scope of this work. *Feasibility checks* [13] could potentially identify overconstrained preconditions by detecting unreachable code. However, systematically evaluating this approach is left for future work.

We evaluate precondition quality by checking logical equivalence against the ground-truth dataset. Given two Hoare triples $(PRE_1, C_1, POST_1)$ and $(PRE_2, C_2, POST_2)$ with $C_1 \equiv C_2$, equivalence can be checked by proving $PRE_1 \iff PRE_2$ and then $POST_1 \iff POST_2$ assuming either precondition holds [6, 35].

4 Dataset

Our dataset contains 50 Java programs, each consisting of a single static method. The programs implement common computer science tasks and were sourced or adapted mainly from *SpecGenBench* [27], but also from *CloverBench* [35], programming course material, and problems from competition platforms [25]. We classified them by loop nesting depth. The classification groups 18 loop-free programs, classified as easy; 23 programs with single-level loops, classified as medium; and 9 programs with double-nested loops, classified as hard.

We curated the dataset to avoid redundancy and ensure compatibility with our verification evaluation. *SpecGenBench* [27] contains 120 test programs. Among these, we identified 35 near-duplicates with minor syntactic or semantic variations, such as replacing a single expression with multiple conditionals, using alternative loop structures, or reversing the intended semantics. We excluded these programs to avoid overrepresenting repeated variants of the same task.

We also excluded programs for which a complete specification required predicates that are not directly supported by OpenJML, such as the *multiset* predicate for permutations and *sum* or *product* predicates for loop accumulation [13]. Evaluating whether LLMs can generate more complex predicates, which often require inductive properties expressed in auxiliary methods [19], is outside the scope of this work.

For each selected program, we manually constructed and verified a ground-truth (GT) specification with OpenJML, including the expected precondition and postcondition. GT specifications are used to evaluate precondition quality through logical equivalence, as discussed in Section 3.2. We excluded inputs that trigger runtime exceptions by constraining the precondition, rather than using *exceptional_behavior* clauses.

The GT specifications were also used to create a second dataset of correct but intentionally incomplete specifications. We obtained these variants by weakening postconditions while preserving correctness. For example, replacing "$\backslash result \iff predicate$" with "$predicate \rightarrow (\backslash result == true)$", replacing "$\backslash result == value\_expr$" with "$\backslash result >= value\_expr$", or removing some *ensures* clauses from multi-behavior specifications (*also* clauses).

5 Methodology

This section describes our evaluation methodology. We first assess how accurately the FormalBench [24] mutation-based completeness metric identifies known complete and incomplete specifications. We then evaluate how our completeness check can be used to assess LLM-based specification generation techniques.

5.1 Mutation-Based Completeness Evaluation

As discussed in Sections 2 and 3.1.1, mutation-based completeness evaluation may produce false negatives, i.e., classify a complete specification as incomplete, when it generates mutants semantically equivalent to the original program. It may also produce false positives, i.e., classify an incomplete specification as complete, when the generated mutants fail to cover missing behaviors.

We evaluated the accuracy of the *FormalBench* [24] mutation-based evaluation in identifying *complete* specifications by running it on our ground-truth dataset, whose specifications were validated as Strongest Postconditions by our method. This experiment assessed the ability of Equivalent Mutant Suppression (EMS) [23], used by FormalBench, to remove equivalent mutants.

We then evaluated its accuracy in identifying *incomplete* specifications by running it on our incomplete specification dataset, obtained by manually weakening the complete postconditions. This experiment assessed whether the generated mutants adequately cover the behaviors missing from the weakened specifications.

5.2 Specification Inference

We evaluated three generation strategies: (1) a conversational loop, in which the LLM generates a specification and iteratively repairs it when verification fails; (2) the SpecGen workflow [27], which uses fixed mutation operators to repair incorrect specifications; and (3) a conversational loop augmented with closed-loop Completeness Feedback (CF).

The LLM is expected to infer specifications from code artifacts such as data types, variable names, control flow, function calls, and comments. We used our completeness check (Section 3.1) to evaluate generated postconditions, and evaluated precondition quality

by checking logical equivalence against the GT specifications, as described in Section 4.

We included SpecGen to evaluate how its mutation-based repair phase affects specification completeness. SpecGen applies fixed mutations to failed loop invariants or postconditions in an attempt to repair incorrect specifications and make them verifiable. This made it a natural case study for analyzing whether its mutation repair preserves, improves, or degrades completeness.

The evaluated scenarios were:

- The SpecGen workflow, to evaluate the trade-offs introduced by its mutation repair heuristics, which interrupt the LLM repair loop once a mutation candidate is found.
- A Conversational loop, to compare continued LLM-based repair against SpecGen's mutation-based repair workflow.
- A Conversational loop with closed-loop Completeness Feedback (CF), to evaluate whether the LLM could strengthen or rewrite specifications that could not be proven complete with our check.

Figure 2 illustrates the generation flow for these scenarios. The generated specification is first checked against validity rules (Section 5.2.1) and then verified with OpenJML. If verification fails, we generate simple repair instructions from the verifier error, following the approach used by Ma et al. [27].

The SpecGen workflow is built on top of the conversational flow. On each failed attempt, it checks for mutation candidates, namely, loop-invariant or postcondition errors. If a mutation candidate is found, the process branches to SpecGen's mutation-based repair phase. Otherwise, the workflow starts another iteration, until a correct specification is generated or the maximum number of iterations is reached.

In the CF flow, once a correct specification is found, we apply our completeness check. If the specification cannot be proven complete, we provide feedback to the LLM stating that the specification is correct but needs to be strengthened. The CF loop continues until a correct and complete specification is generated or the maximum number of iterations is reached.

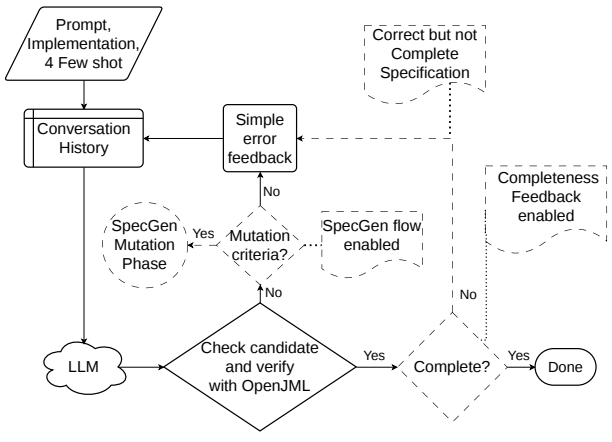


Figure 2: Flowchart of the three specification generation approaches: SpecGen, Conversational and Completeness Feedback (CF).

Our SpecGen setup uses the random mutation-candidate selector heuristic available in the digital artifact [26]. We limited the mutation repair phase to 1000 iterations, since preliminary runs showed that some programs produced more than 3000 repair candidates and took over two hours to complete. The original setup of Ma et al. did not impose this limit, but reported an average of approximately 50 mutation iterations per test [27].

All three approaches use the same initial generation prompt, kept from the SpecGen artifact [26]. The LLM receives the task description, the program, and four fixed few-shot examples. We used a limit of 10 LLM generation iterations and an OpenJML timeout of 200 seconds per invocation.

5.2.1 Generation Rules. We enforced additional rules in the generation phase. If one of these rules is violated, the iteration candidate is rejected, and the LLM receives feedback to fix it.

- Assumptions are forbidden. Preliminary tests showed that GPT-3.5 Turbo sometimes used assumptions instead of proper loop invariants, a behavior encouraged by some SpecGen repair examples [26]. We forbid assumptions because they can make verification unsound if used to bypass proof obligations.
- The specification must contain at least one explicit postcondition. Preliminary tests showed that, when all repair mutation operators failed to produce a correct specification, the SpecGen mutation phase removed the failing specification clause entirely. In one GPT-3.5 Turbo trial, this removed all behaviors from the postcondition in 24 out of 143 tests that reached the mutation phase. An empty postcondition defaults to *ensures true*, which is always correct. We therefore classify empty postconditions as incorrect.
- The specification must contain an explicit frame condition different from *assigns \everything* [32].
- The LLM may modify only comments, not source code.
- The specification must not use exceptional postconditions (*signals* clauses). This keeps the generated specifications aligned with the GT specifications. Inputs that would cause exceptions must instead be excluded by the precondition.

5.2.2 Model Variants. We evaluated three models: GPT-3.5 Turbo (gpt-3.5-turbo-1106, t=0.4), GPT-4.1 Mini (gpt-4.1-mini-2025-04-14, t=0.4), and GPT-5 Mini (gpt-5-mini-2025-08-07, t=1). GPT-3.5 Turbo follows the setup of Ma et al. GPT-4.1 Mini and GPT-5 Mini add more recent non-reasoning and reasoning variants. These models are suitable for well-defined tasks and precise prompts [31] and fit the requirements of our evaluation.

5.2.3 Experimental Settings. The SpecGen artifact [26] invokes OpenJML with assumptions that references are non-null and that any arithmetic operations are in range. The command used by the artifact is:

```
openjml --esc --esc-max-warnings 1 --arithmetic-failure=quiet --nonnull-by-default --timeout 200
```

These options simplify verification by reducing the behaviors that generated contracts must cover. In particular, specifications do not need to account for nullable references or arithmetic overflows, simplifying both correctness and completeness checking (Section 3.2). We confirmed this effect in a preliminary GPT-3.5 Turbo

experiment, as removing these assumptions reduced the overall success rate by 7.6% for SpecGen and 4.8% for Conversational. The data are available in the artifact repository.

Because we aim to evaluate whether LLMs can infer specifications covering multiple behaviors, our evaluation used stricter OpenJML settings:

```
openjml --esc --esc-max-warnings 1 --nullable-by-default
--arithmetic-failure=hard --timeout 200
```

Table 1 summarizes the experimental configurations for the selected generation strategies.

Table 1: Experimental settings.

Model	SpecGen	Conversational	Conversational + CF
GPT-3.5 Turbo	E1	E2	E3
GPT-4.1 Mini	E4	E5	E6
GPT-5 Mini	E7	E8	E9

To account for variation in LLM outputs, each experiment was repeated 10 times per dataset entry and the results were aggregated (50x10). Our evaluation used OpenJML version 21-0.21.

6 Results

This section presents and discusses the empirical results of the experiments described in Section 5.

6.1 Mutation-Based Completeness Evaluation

Table 2 shows the coverage scores obtained for the known *complete* specifications. The metric follows FormalBench [24] and is computed as the number of mutants that fail verification divided by the total number of generated mutants. A failed verification indicates that the mutated behavior is ruled out by the specification and therefore covered by it. Thus, for complete specifications, the expected result is a 100% mutant failure rate, corresponding to a 100% coverage score. The results show that some equivalent mutants remain even after applying Equivalent Mutant Suppression (EMS) [23]. These mutants can lead to false negatives, i.e., reporting a complete specification as incomplete.

Table 2: FormalBench mutation-based coverage scores on known complete specifications. Equivalent mutants that remain even after EMS lead to false negatives.

Dif.	Cov. Score	Avg. number of mutants generated	Avg. num. of equiv. mutants generated
easy	0.95	14.11	0.44
med.	0.92	20.78	1.83
hard	0.90	25.44	2.22

Table 3 shows the coverage scores obtained for the known *incomplete* specifications. FormalBench generated the same mutants used in the complete-specification experiment. The evaluation produced 6 false positives out of 50 cases, i.e., incomplete specifications reported as complete. These false positives occur because the generated mutants fail to cover the behaviors missing from the incomplete specifications.

Except for the easy category, the coverage scores are close to those obtained for the complete specifications in Table 2. This similarity suggests that the mutation-based coverage score does not reliably measure the degree of specification coverage. Instead, the score is strongly influenced by the selected mutants and may therefore give a misleading characterization of completeness.

Table 3: FormalBench mutation-based coverage scores on known incomplete specifications. The scores are similar to those obtained for complete specifications, and 6 out of 50 incomplete specifications were reported as complete.

Difficulty	Coverage Score	False positives reported
easy	0.72	2/18
med.	0.90	4/23
hard	0.88	0/9

6.2 SpecGen Reproducibility and Re-Evaluation

Table 4 compares the SpecGen and conversational workflows. For GPT-3.5 Turbo, we obtained non-incorrect rates of 58.6% for SpecGen (E1) and 51.4% for the conversational workflow (E2). Ma et al. [27] reported corresponding rates of 72.3% and 64.2%. The lower success rates in our re-evaluation can be explained by the removal of near-duplicate programs and by the stricter OpenJML settings, which remove the non-null and no-overflow assumptions used in the original artifact. Nevertheless, the relative result reported by Ma et al. is preserved: on GPT-3.5 Turbo, SpecGen outperforms the conversational workflow for correctness.

For GPT-4.1 Mini and GPT-5 Mini, however, the trend changes. As shown in Table 4, the conversational workflow outperforms SpecGen on both models. This suggests that, for more recent models, allowing the LLM to continue repairing the specification is more effective than prematurely switching to the mutation-based repair phase. The difference is more visible in the detailed categories shown in Table 5. In particular, SpecGen produces fewer verifiable Complete-GT outcomes than the conversational workflow for GPT-4.1 Mini and GPT-5 Mini, while both workflows produce similar completeness rates for GPT-3.5 Turbo. We discuss this degradation in more detail in Section 6.2.1.

Table 4: Performance comparison between the SpecGen workflow and the conversational workflow. The conversational workflow achieved higher non-incorrect rates on GPT-4.1 Mini and GPT-5 Mini.

Model	Overall SpecGen non-incorrect	Overall conversational non-incorrect
GPT-3.5 Turbo: E1 vs E2	293/500 (58.6%)	257/500 (51.4%)
GPT-4.1 Mini: E4 vs E5	407/500 (81.4%)	413/500 (82.6%)
GPT-5 Mini: E7 vs E8	441/500 (88.2%)	475/500 (95.0%)

Table 5: Detailed result categories for all experiments. GPT-4.1 Mini and GPT-5 Mini produced more verifiable Complete-GT specifications with the conversational workflow than with SpecGen.

Experiment	Dif.	Incorrect	At Least Correct	Complete not GT	Complete GT
E1: SpecGen. GPT-3.5 Turbo	easy	16 (8,9%)	19 (10,6%)	54 (30,0%)	91 (50,6%)
	med.	112 (48,7%)	58 (25,2%)	22 (9,6%)	38 (16,5%)
	hard	79 (87,8%)	9 (10,0%)	1 (1,1%)	1 (1,1%)
E2: Convers. GPT-3.5 Turbo	easy	15 (8,3%)	17 (9,4%)	55 (30,6%)	93 (51,7%)
	med.	139 (60,4%)	32 (13,9%)	21 (9,1%)	38 (16,5%)
	hard	89 (98,9%)	0 (0,0%)	0 (0,0%)	1 (1,1%)
E3: Convers. + CF GPT-3.5 Turbo	easy	17 (9,4%)	9 (5,0%)	67 (37,2%)	87 (48,3%)
	med.	137 (59,6%)	35 (15,2%)	20 (8,7%)	38 (16,5%)
	hard	90 (100,0%)	0 (0,0%)	0 (0,0%)	0 (0,0%)
E4: SpecGen GPT-4.1 Mini	easy	0 (0,0%)	18 (10,0%)	30 (16,7%)	132 (73,3%)
	med.	62 (27,0%)	50 (21,7%)	45 (19,6%)	73 (31,7%)
	hard	31 (34,4%)	30 (33,3%)	1 (1,1%)	28 (31,1%)
E5: Convers. GPT-4.1 Mini	easy	0 (0,0%)	7 (3,9%)	32 (17,8%)	141 (78,3%)
	med.	40 (17,4%)	28 (12,2%)	57 (24,8%)	105 (45,7%)
	hard	47 (52,2%)	12 (13,3%)	1 (1,1%)	30 (33,3%)
E6: Convers. + CF GPT-4.1 Mini	easy	0 (0,0%)	0 (0,0%)	32 (17,8%)	148 (82,2%)
	med.	44 (19,1%)	19 (8,3%)	49 (21,3%)	118 (51,3%)
	hard	51 (56,7%)	8 (8,9%)	3 (3,3%)	28 (31,1%)
E7: SpecGen GPT-5 Mini	easy	3 (1,7%)	7 (3,9%)	51 (28,3%)	119 (66,1%)
	med.	34 (14,8%)	39 (17,0%)	40 (17,4%)	117 (50,9%)
	hard	22 (24,4%)	18 (20,0%)	7 (7,8%)	43 (47,8%)
E8: Convers. GPT-5 Mini	easy	3 (1,7%)	5 (2,8%)	48 (26,7%)	124 (68,9%)
	med.	6 (2,6%)	32 (13,9%)	54 (23,5%)	138 (60,0%)
	hard	16 (17,8%)	12 (13,3%)	5 (5,6%)	57 (63,3%)
E9: Convers. + CF GPT-5 Mini	easy	4 (2,2%)	0 (0,0%)	54 (30,0%)	122 (67,8%)
	med.	5 (2,2%)	18 (7,8%)	58 (25,2%)	149 (64,8%)
	hard	12 (13,3%)	4 (4,4%)	13 (14,4%)	61 (67,8%)

6.2.1 SpecGen Mutation Analysis. Table 6 shows the effectiveness of the SpecGen mutation repair phase. The results show that most mutation runs resulted in *Incorrect* or *At-Least-Correct* specifications, while few produced verifiable *Complete-not-GT* or *Complete-GT* outcomes. This suggests that the mutation-based repair rarely contributed to generating verifiable complete specifications.

Table 7 shows the contribution of the SpecGen mutation repair phase within all the results obtained. The results show that the LLM alone produced the majority of *Complete-not-GT* and *Complete-GT*. This suggests that prematurely shifting to the mutation phase may prevent further LLM-based repair, which was more effective than mutation-based repair at producing complete specifications in our experiments.

SpecGen uses four mutation operators: *predicative*, *logical*, *comparative*, and *arithmetic* [27]. If all mutation operators are applied and none produces a verifiable specification, the candidate specification clause is removed. Upon manual inspection, we found that this step often removes desired behaviors that the LLM had identified but failed to express correctly, with the extreme case being the removal of all postconditions, as discussed in Section 5.2.1. These results suggest that SpecGen's mutation-based repair phase often improves verifiability by weakening the specification, which can reduce completeness even when correctness improves.

Table 6: Effectiveness of SpecGen's mutation repair phase, measured by the distribution of outcomes among runs in which the phase was activated. Most mutation-repair runs could not be proven complete and produced either *Incorrect* or *At-Least-Correct* specifications.

Experiment	Dif.	Incorrect	At Least Correct	Complete not GT	Complete GT
E1: SpecGen GPT-3.5 Turbo	easy	5/8 (62.5%)	0/8 (0.0%)	3/8 (37.5%)	0/8 (0.0%)
	med.	56/92 (60.9%)	28/92 (30.4%)	5/92 (5.4%)	3/92 (3.3%)
	hard	45/54 (83.3%)	9/54 (16.7%)	0/54 (0.0%)	0/54 (0.0%)
E4: SpecGen GPT-4.1 Mini	easy	0/9 (0.0%)	8/9 (88.9%)	0/9 (0.0%)	1/9 (11.1%)
	med.	54/83 (65.1%)	27/83 (32.5%)	1/83 (1.2%)	1/83 (1.2%)
	hard	17/37 (45.9%)	16/37 (43.2%)	0/37 (0.0%)	4/37 (10.8%)
E7: SpecGen GPT-5 Mini	easy	1/6 (16.7%)	5/6 (83.3%)	0/6 (0.0%)	0/6 (0.0%)
	med.	28/47 (59.6%)	9/47 (19.1%)	9/47 (19.1%)	1/47 (2.1%)
	hard	19/36 (52.8%)	9/36 (25.0%)	1/36 (2.8%)	7/36 (19.4%)

Table 7: Contribution of SpecGen's mutation repair phase to the final results, measured as the proportion of outcomes attributable to mutation repair among all obtained outcomes. The LLM alone accounted for most *Complete-not-GT* and *Complete-GT* specifications.

Experiment	Dif.	Incorrect	At Least Correct	Complete not GT	Complete GT
E1: SpecGen GPT-3.5 Turbo	easy	5/16 (31.3%)	0/19 (0.0%)	3/54 (5.6%)	0/91 (0.0%)
	med.	56/112 (50.0%)	28/58 (48.3%)	5/22 (22.7%)	3/38 (7.9%)
	hard	45/79 (57.0%)	9/9 (100.0%)	0/1 (0.0%)	0/1 (0.0%)
E4: SpecGen GPT-4.1 Mini	easy	N/A (-)	8/18 (44.4%)	0/30 (0.0%)	1/132 (0.8%)
	med.	54/62 (87.1%)	27/50 (54.0%)	1/45 (2.2%)	1/73 (1.4%)
	hard	17/31 (54.8%)	16/30 (53.3%)	0/1 (0.0%)	4/28 (14.3%)
E7: SpecGen GPT-5 Mini	easy	1/3 (33.3%)	5/7 (71.4%)	0/51 (0.0%)	0/119 (0.0%)
	med.	28/34 (82.4%)	9/39 (23.1%)	9/40 (22.5%)	1/117 (0.9%)
	hard	19/22 (86.4%)	9/18 (50.0%)	1/7 (14.3%)	7/43 (16.3%)

6.3 Impact of Completeness Feedback (CF)

Table 8 shows the effectiveness of the Completeness Feedback. It groups all results obtained by runs where CF occurred. The results show that GPT-3.5 Turbo was usually unable to improve the specification after CF, as most activations remained *At-Least-Correct*. For GPT-4.1 Mini and GPT-5 Mini, CF was more effective, producing *Complete-GT* results in up to 50.0% of cases where feedback occurred. These runs would otherwise have terminated earlier with correct but not provably complete specifications.

Table 9 shows the contribution of the Completeness Feedback within all results obtained[1]. CF accounted for up to 10.7% of all *Complete-GT* outcomes. It also appears frequently among *Complete-not-GT* outcomes, which may suggest that the feedback can induce the LLM to modify the precondition rather than only strengthen the postcondition. Confirming this effect requires further experimentation.

[1]Ideally, every *At-Least-Correct* result would have received the CF. We identified that 6 tests were proven complete during the generation, but could not be verified when reprocessing the logs. We attribute this to the SMT solver nondeterminism, which can be mitigated with the *solver-seed* switch [12]. These are listed as missing feedback in the *At-Least-Correct* column of Table 9.

Table 8: Effectiveness of Completeness Feedback, measured by the outcome distribution among runs in which CF was activated. For GPT-4.1 Mini and GPT-5 Mini, Completeness Feedback produced *Complete-GT* specifications in up to 50.0% of activated runs.

Experiment	Dif.	At Least Correct	Complete not GT	Complete GT
E3: Convers.	easy	9/17 (52.9%)	6/17 (35.3%)	2/17 (11.8%)
+ CF	med.	33/33 (100.0%)	0/33 (0.0%)	0/33 (0.0%)
GPT-3.5 Turbo	hard	N/A (-)	N/A (-)	N/A (-)
E6: Convers.	easy	0/10 (0.0%)	5/10 (50.0%)	5/10 (50.0%)
+ CF	med.	18/38 (47.4%)	8/38 (21.1%)	12/38 (31.6%)
GPT-4.1 Mini	hard	8/13 (61.5%)	2/13 (15.4%)	3/13 (23.1%)
E9: Convers.	easy	N/A (-)	N/A (-)	N/A (-)
+ CF	med.	15/35 (42.9%)	6/35 (17.1%)	14/35 (40.0%)
GPT-5 Mini	hard	4/15 (26.7%)	5/15 (33.3%)	6/15 (40.0%)

Table 9: Contribution of Completeness Feedback to the final results, measured as the proportion of outcomes attributable to Completeness Feedback among all obtained outcomes. Completeness Feedback accounted for up to 10.7% of *Complete-GT* specifications and contributed frequently to *Complete not GT* specifications.

Experiment	Dif.	At Least Correct	Complete not GT	Complete GT
E3: Convers.	easy	9/9 (100.0%)	6/67 (9.0%)	2/87 (2.3%)
+ CF	med.	33/35 (94.3%)	0/20 (0.0%)	0/38 (0.0%)
GPT-3.5 Turbo	hard	N/A (-)	N/A (-)	N/A (-)
E6: Convers.	easy	N/A (-)	5/32 (15.6%)	5/148 (3.4%)
+ CF	med.	18/19 (94.7%)	8/49 (16.3%)	12/118 (10.2%)
GPT-4.1 Mini	hard	8/8 (100.0%)	2/3 (66.7%)	3/28 (10.7%)
E9: Convers.	easy	N/A (-)	0/54 (0.0%)	0/122 (0.0%)
+ CF	med.	15/18 (83.3%)	6/58 (10.3%)	14/149 (9.4%)
GPT-5 Mini	hard	4/4 (100.0%)	5/13 (38.5%)	6/61 (9.8%)

7 Threats to Validity

Internal validity: Deductive verifiers may fail to prove valid properties due to the formulation of the specification or the need for additional invariants [10]. Therefore, some specifications classified as *At Least Correct* may in fact be complete but not provable by OpenJML in their current form. To avoid overclaiming, we treat *At Least Correct* as an inconclusive category and do not use it for broad direct comparisons.

Our results also depend on the soundness of OpenJML and its underlying SMT solver. To mitigate this threat, we extensively tested OpenJML on our dataset and reported all issues found during the evaluation, which were later fixed by the developer.

External validity: The size and scope of our dataset limit the generalization of the results. The dataset focuses on Java methods, deterministic behavior, and specifications expressible with the OpenJML constructs used in this study. Therefore, the results of the specification inference may not generalize to larger programs or specifications that require more complex predicates. Due to this, we frame our evaluation as a proof of concept rather than a large-scale benchmark.

We became aware of the FormalBench base dataset [24] only after constructing and verifying our dataset. As discussed in Section 4, our evaluation required manually curating the dataset for fairness. Manually curating the 700 programs on FormalBench would require substantial additional effort. Although our dataset is small, it was sufficient to illustrate the feasibility of our method and to reveal limitations of mutation-based completeness evaluation, as well as the trade-off between correctness and completeness introduced by SpecGen's mutation repair phase.

Conclusion validity: The limited dataset size may also affect conclusion validity, since aggregate results can be sensitive to individual samples. We therefore avoid broad quantitative claims and interpret the results as evidence of feasibility rather than as a definitive benchmark of specification quality.

8 Conclusion and Future Work

We introduce a verification-based postcondition completeness check, which requires only the candidate specification for a deterministic program. This makes it independent of the specification generation process, extending its applicability beyond LLM-inferred contracts. We demonstrated its feasibility through an empirical study of three LLM-based generation workflows, using it both to evaluate generated specifications and to provide closed-loop completeness feedback. This also allowed us to obtain additional insights about the studied methodologies.

Bounded model checking is a natural complement to our approach, as it can help identify incomplete specifications through counterexamples when deductive verification cannot prove completeness. Integrating such checks directly into the verifier functionality would also make completeness checking more practical and easier to adopt.

Precondition quality is also fundamental to the completeness analysis, since the Strongest Postcondition is defined relative to the precondition. Therefore, automatically inferring the appropriate preconditions remains an important direction for automated contract inference.

Artifact Availability

The dataset, code, logs, and other artifacts are available at https://zenodo.org/records/22119557, under the MIT license.

Acknowledgments

We thank all reviewers for their feedback and David R. Cok for his support with OpenJML.

References

[1] Samia Alblwi, Amani Ayad, and Ali Mili. 2024. Mutation Coverage is not Strongly Correlated with Mutation Coverage. In *Proceedings of the 5th ACM/IEEE International Conference on Automation of Software Test (AST 2024)* (Lisbon, Portugal) *(AST '24)*. Association for Computing Machinery, New York, NY, USA, 1–11. doi:10.1145/3644032.3644442

[2] Paul E. Ammann and Paul E. Black. 2000. A Specification-Based Coverage Metric To Evaluate Test Sets. *International Journal of Reliability, Quality and Safety Engineering* 8 (11 2000). doi:10.1142/S0218539301000530

[3] Samuel Amorim, Leonardo Fernandes, Márcio Ribeiro, Rohit Gheyi, Márcio Eduardo Delamaro, Marcio Guimarães, and André Santos. 2024. Reducing manual

efforts in equivalence analysis in mutation testing. *Journal of Software Engineering Research and Development* 13, 3 (2024), 1–17.

[4] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. 2014. Test Code Quality and Its Relation to Issue Handling Performance. *IEEE Transactions on Software Engineering* 40, 11 (2014), 1100–1125. doi:10.1109/TSE.2014.2342227

[5] Ragib Shahariar Ayon and Shibbir Ahmed. 2026. AutoReSpec: A Framework for Generating Specification using Large Language Models. arXiv:2604.03758 [cs.SE] https://arxiv.org/abs/2604.03758

[6] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. *SIGPLAN Not.* 39, 1 (Jan. 2004), 14–25. doi:10.1145/982962.964003

[7] Daniel M Berry. 2002. Formal methods: the very idea: Some thoughts about why they work when they work. *Science of Computer Programming* 42, 1 (2002), 11–27. doi:10.1016/S0167-6423(01)00026-0 Special Issue on Engineering Automation for Computer Based Systems.

[8] Allan Blanchard. 2020. *Introduction to C program proof with Frama-C and its WP plugin.* https://allan-blanchard.fr/frama-c-wp-tutorial.html

[9] Claudinei Brito, Vinicius H. S. Durelli, Rafael S. Durelli, Simone R. S. de Souza, Auri M. R. Vincenzi, and Marcio Eduardo Delamaro. 2020. A preliminary investigation into using machine learning algorithms to identify minimal and equivalent mutants. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 304–313.

[10] Lea Salome Brugger, Xavier Denis, and Peter Müller. 2026. Toward Practical Deductive Verification: Insights from a Qualitative Survey in Industry and Academia. arXiv:2510.20514 [cs.SE] https://arxiv.org/abs/2510.20514

[11] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.

[12] David R. Cok. 2025. *User guide to specification and verification with the Java Modeling Language and OpenJML. DRAFT September 27, 2025.* https://www.openjml.org/documentation/OpenJMLUserGuide.pdf

[13] David R. Cok, Gary T. Leavens, and Mattias Ulbrich. 2025. Java Modeling Language (JML). Reference Manual. 2ed. DRAFT September 27, 2025. https://www.openjml.org/documentation/JML_Reference_Manual.pdf

[14] Edsger W. Dijkstra and Carel S. Schölten. 1990. *The strongest postcondition.* Springer New York, New York, NY, 209–215. doi:10.1007/978-1-4612-3228-5_12

[15] Arnaud Ebalard, Patricia Mouy, and Ryad Benadjila. 2019. Journey to a RTE-free X.509 parser. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*. 29–49. https://www.sstic.org/media/SSTIC2019/SSTIC-actes/journey-to-a-rte-free-x509-parser/SSTIC2019-Article-journey-to-a-rte-free-x509-parser-ebalard_mouy_benadjila_3cUxSCv.pdf

[16] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. 2024. Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? *Proc. ACM Softw. Eng.* 1, FSE, Article 84 (July 2024), 24 pages. doi:10.1145/3660791

[17] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1 (2007), 35–45. doi:10.1016/j.scico.2007.01.015 Special issue on Experimental Software and Toolkits.

[18] Cormac Flanagan and K. Rustan M. Leino. 2001. Houdini, an Annotation Assistant for ESC/Java. In *FME 2001: Formal Methods for Increasing Software Productivity*, José Nuno Oliveira and Pamela Zave (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 500–517.

[19] Jens Gerlach. 2025. ACSL by Example: Version 32.0.0 for Frama-C 32.0 (Germanium). *Fraunhofer FOKUS* (2025). https://github.com/fraunhoferfokus/acsl-by-example

[20] Mike Gordon. 2016. *Background reading on Hoare Logic.* Learning guide / lecture notes. University of Cambridge, Computer Laboratory. MJCG February 2, 2016, available at
urlhttps://www.cl.cam.ac.uk/archive/mjcg/HL/Notes/Notes.pdf.

[21] Dalton N. Jorge, Patrícia D. L. Machado, Everton L. G. Alves, and Wilkerson L. Andrade. 2018. Integrating requirements specification and model-based testing in agile development. In *2018 IEEE 26th International Requirements Engineering Conference (RE)*. IEEE, 336–346.

[22] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing mutants to guide mutation testing. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1743–1754. doi:10.1145/3510003.3510187

[23] Benjamin Kushigian, Samuel J. Kaufman, Ryan Featherman, Hannah Potter, Ardi Madadi, and René Just. 2024. Equivalent Mutants in the Wild: Identifying and Efficiently Suppressing Equivalent Mutants for Java Programs. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 654–665. doi:10.1145/3650212.3680310

[24] Thanh Le-Cong, Bach Le, and Toby Murray. 2025. Can LLMs Reason About Program Semantics? A Comprehensive Evaluation of LLMs on Formal Specification Inference. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 21991–22014. doi:10.18653/v1/2025.acl-long.1068

[25] Chang Liu. 2019. fluency03 LeetCode Java problems GitHub repository. https://github.com/fluency03/leetcode-java

[26] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. Github repository artifact - SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. https://github.com/Lezhi-Ma/SpecGen-Artifact

[27] Lezhi Ma, Shangqing Liu, Yi Li, Xiaofei Xie, and Lei Bu. 2025. SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 16–28. doi:10.1109/ICSE55347.2025.00129

[28] Md Rakib Hossain Misu, Iris Ma, and Cristina V. Lopes. 2026. VeriAct: Beyond Verifiability – Agentic Synthesis of Correct and Complete Formal Specifications. arXiv:2604.00280 [cs.SE] https://arxiv.org/abs/2604.00280

[29] Facundo Molina, Marcelo d'Amorim, and Nazareno Aguirre. 2022. Fuzzing class specifications. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1008–1020. doi:10.1145/3510003.3510120

[30] Rashmi Mudduluru, Jason Waataja, Suzanne Millstein, and Michael Ernst. 2021. Verifying Determinism in Sequential Programs. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 37–49. doi:10.1109/ICSE43902.2021.00017

[31] OpenAI. 2025. OpenAI Models — Model Reference Documentation. https://platform.openai.com/docs/models. Accessed: 2025-11-29.

[32] OpenJML. 2026. OpenJML Frame Conditions Tutorial. https://www.openjml.org/tutorial/FrameConditions. Accessed: 2026-02-15.

[33] Cedric Richter and Heike Wehrheim. 2025. Beyond Postconditions: Can Large Language Models infer Formal Contracts for Automatic Software Verification? arXiv:2510.12702 [cs.SE] https://arxiv.org/abs/2510.12702

[34] John L. Singleton, Gary T. Leavens, Hridesh Rajan, and David Cok. 2018. An algorithm and tool to infer practical postconditions. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 313–314. doi:10.1145/3183440.3194986

[35] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. 2024. Clover: Closed-Loop Verifiable Code Generation. In *AI Verification*, Guy Avni, Mirco Giacobbe, Taylor T. Johnson, Guy Katz, Anna Lukina, Nina Narodytska, and Christian Schilling (Eds.). Springer Nature Switzerland, Cham, 134–155.

[36] Zhao Tian, Junjie Chen, Qihao Zhu, Junjie Yang, and Lingming Zhang. 2023. Learning to Construct Better Mutation Faults. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 64, 13 pages. doi:10.1145/3551349.3556949

[37] Norbert Tihanyi, Tamas Bisztray, Mohamed Amine Ferrag, Ridhi Jain, and Lucas C. Cordeiro. 2025. How secure is AI-generated code: a large-scale comparison of large language models. *Empirical Software Engineering* 30, 47 (2025). doi:10.1007/s10664-024-10590-1

[38] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. 2024. Enchanting Program Specification Synthesis by Large Language Models Using Static Analysis and Program Verification. In *Computer Aided Verification*, Arie Gurfinkel and Vijay Ganesh (Eds.). Springer Nature Switzerland, Cham, 302–328.

[39] Guangyuan Wu, Weining Cao, Yuan Yao, Hengfeng Wei, Taolue Chen, and Xiaoxing Ma. 2024. LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering* (Sacramento, CA, USA) *(ASE '24)*. Association for Computing Machinery, New York, NY, USA, 406–417. doi:10.1145/3691620.3695014

[40] Yiheng Xiong, Mengqian Xu, Ting Su, Jingling Sun, Jue Wang, He Wen, Geguang Pu, Jifeng He, and Zhendong Su. 2023. An Empirical Study of Functional Bugs in Android Apps. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1319–1331. doi:10.1145/3597926.3598138

[41] 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. 2025. Autoverus: Automated Proof Generation for Rust Code. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (2025).

[42] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. 2020. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 106–120.