LLM-Assisted Translation and Bounded Model Checking of Python Code

Shivkumar Shivaji[1], Klaus Havelund[2], Natalia Lobakhina[1], Lucas C. Cordeiro[3,4], and Alessandro Pinto[2]

[1] Generative AI LLC, USA
shiv@revive.cx, nlobakhina@revive.cx
[2] NASA Jet Propulsion Laboratory, California Institute of Technology, USA
klaus.havelund@jpl.nasa.gov, alessandro.pinto@jpl.nasa.gov
[3] University of Manchester, UK
lucas.cordeiro@manchester.ac.uk
[4] Federal University of Amazonas, Brazil

Abstract. Formal verification of Python programs remains challenging due to the language's dynamic nature and rich semantic constructs. We present an approach that combines Large Language Models (LLMs) with Bounded Model Checking (BMC) to verify Python code. Our system uses an LLM to translate Python programs into C code suitable for formal verification using ESBMC, enabling the detection of critical bugs, including arithmetic overflows, array bounds violations, and concurrency errors. We evaluate our approach on a benchmark of 23 carefully designed Python programs with planted bugs representing common verification challenges. The LLM orchestrator successfully detected all planted bugs by iteratively coordinating static, dynamic, and formal verification tools. Our results demonstrate that LLM-assisted translation can make mature C verification tools accessible for Python code analysis, though the current evaluation is scoped to programs of 15–50 lines with bounded loops and statically-sized data structures, a practical limit driven by ESBMC verification times rather than a fundamental BMC constraint.

Keywords: Formal Verification · Bounded Model Checking · Python · Large Language Models · Code Translation

1 Introduction

The formal verification of software systems remains a critical challenge in ensuring the reliability and security of software. Techniques such as model checking and theorem proving have been developed for verifying programs written in statically typed languages such as C and Java, with applications in safety-critical domains including embedded systems and financial software [1,2]. However, formal verification remains challenging even for these languages, and dynamic programming languages, notably Python, introduce additional complexities due to their flexible typing and rich semantic constructs [3].

Python's popularity spans research and commercial applications, particularly in data science and machine learning. The dynamic nature of its semantics presents significant obstacles to existing automated verification tools, which often support only fragmented subsets of the language or simplified code patterns. Features such as metaprogramming, dynamic type mutation, and the wide variety of data structures in Python significantly complicate the construction of unified and scalable formal verification frameworks capable of capturing the full semantics of the language [37].

Recent progress in Large Language Models (LLMs), developed from extensive multilingual code corpora, has opened new directions for addressing longstanding verification challenges. These models demonstrate advanced competence in code synthesis and syntactic transformation across programming languages, making it feasible to automatically translate Python programs into rigorously verifiable representations such as C [4].

This study introduces a framework that uses LLMs to orchestrate bug detection in Python code through translation to C and bounded model checking. Our approach targets two primary use cases: (1) detecting bugs in production Python code such as arithmetic overflows, array bounds violations, and concurrency errors, and (2) enabling engineers to write design models in Python rather than dedicated formal specification languages like TLA+, applying formal verification tools to the translated C code.

Like all functional correctness verification approaches, assertions must be provided to specify desired properties—our contribution is automating the translation and verification process around these specifications. Since the Python-to-C translation is performed by an LLM without formal proof of semantic equivalence, we focus on bug hunting rather than formal verification guaranties for the original Python code. By using AST analysis to guide selective verification and intelligently configuring ESBMC (Efficient SMT-Based Context-Bounded Model Checker [12]) parameters, we demonstrate effective bug detection on small to medium Python functions (15-50 lines of code). An experimental evaluation of 23 Python programs achieves 100% bug detection for planted errors, including overflows, bound violations, and deadlocks.

The remainder of this paper is organized as follows. Section 2 provides background on formal verification, BMC, LLMs in program verification, and related code translation tools. Section 3 describes the methodology and architecture of our EVA (Enhanced Verification Agent) system. Section 4 details the implementation. Section 5 presents our experimental evaluation and results. Section 6 discusses future work.

2 Background

2.1 Formal Verification and Model Checking

Formal verification mathematically proves that a program meets its specified properties by modeling the program and its desired behaviors and rigorously

checking all possible executions. Two foundational paradigms have emerged: model checking, which relies on state space exploration against temporal logic specifications [6,7], and theorem proving, which requires constructing formal proofs for these specifications. Model checkers include tools such as SPIN [39], NuSMV [40], and CBMC [10] for software verification, while theorem provers include PVS [41], Isabelle/HOL [42], Rocq [43], and Lean [44] for interactive proof development.

Model checking systematically tests whether a program, represented as a transition system, satisfies properties such as safety (e.g. absence of buffer overflows) and liveness (e.g. termination or progress). Formally, for a Kripke structure M and temporal logic property φ, verification asks whether $M \models \varphi$ [7,8].

A fundamental challenge is the exponential growth of possible system states (the "state explosion" problem) as program complexity increases [8,9]. Bounded Model Checking (BMC) mitigates this by restricting analysis to execution traces of a given length k. BMC translates the verification problem into a SAT or SMT query:

$$Initial(s_0) \ \wedge \ \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \ \neg \Phi_k$$

where T is the transition relation and Φ_k is the bounded encoding of the property over the finite trace $s_0, \ldots, s_k$. If this formula is satisfiable, the model yields a counterexample trace of length at most k. For example, if the property is the invariant $G\,p$ ("p holds globally"), then its bounded encoding is

$$\Phi_k \ = \ \bigwedge_{i=0}^{k} p(s_i)$$

so the negated property becomes

$$\neg\Phi_k \ = \ \bigvee_{i=0}^{k} \neg p(s_i)$$

meaning that the counterexample contains some state in which p is violated. BMC is particularly effective at detecting shallow bugs and corner-case errors that may be difficult to uncover through testing alone.

Modern verification tools leverage BMC for practical software analysis. CBMC applies BMC to C/C++ programs, targeting assertion violations, pointer errors, and numerical overflows [10,11]. ESBMC extends the method to multi-threaded and embedded systems, improving scalability and expressivity in industrial use cases [12,13]. JBMC adapts similar techniques to Java bytecode, maintaining support for object-oriented program structures [14-16].

However, although BMC is widely adopted and effective for systematic bug-finding and shallow counterexample generation, it remains inherently incomplete: bug-freedom is only proven up to the bound k, and deeper properties may go unverified. Resource constraints and formula complexity further limit scalability

as system size and concurrency grow. These unresolved challenges motivate advances in hybrid verification approaches and LLM-assisted verification, forming the basis for subsequent research in this work.

2.2 LLMs in Program Verification and Synthesis

Recent advances in large language models have significantly impacted formal verification by automating traditionally labor-intensive tasks such as specification generation, invariant detection, and formal proof construction [17]. LLMs trained on extensive code and mathematical corpora are capable of understanding and generating formal languages, bridging the gap between informal code and rigorous formal methods. Figure 1 illustrates this hybrid architecture where LLMs automate the generation of formal artifacts while symbolic engines provide rigorous verification guarantees through an iterative refinement loop.

Autoformalization is the process by which LLMs translate natural language descriptions, informal comments, or code into formal specifications for theorem provers or model checkers [18-20]. This allows formal capturing of intended program properties, critical for automation. Invariant generation, vital in program

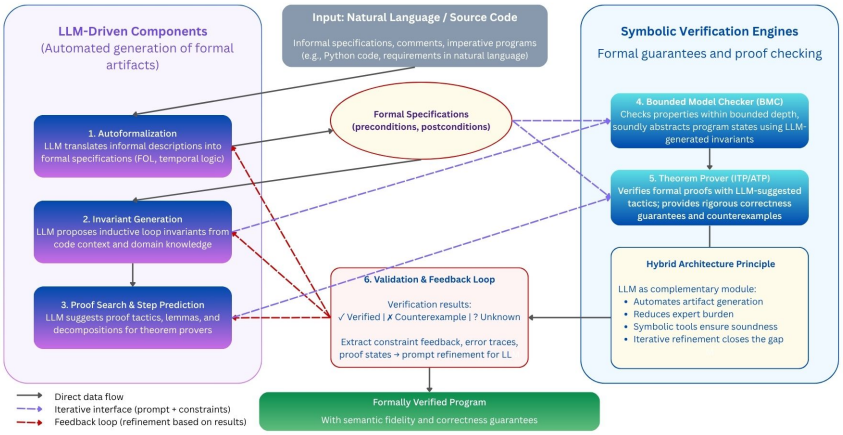

Fig. 1. An example of a hybrid LLM-symbolic verification architecture [17], in which LLMs automate the generation of formal artifacts (specifications, invariants, proof tactics) while symbolic engines provide rigorous verification guarantees through an iterative refinement loop. This general pattern from the literature motivates our EVA system, which adapts it specifically for Python bug detection via LLM-orchestrated tool selection and Python-to-C translation.

verification, benefits from the' ability of LLMs to propose inductive invariants

from code context and known properties, e.g., allowing bounded model checkers to reduce verification complexity by soundly abstracting program states [21,22].

Proof-step prediction and automated proof search also leverage LLMs to suggest next proof steps, reducing expert burden and increasing the degree of automation in theorem-proving pipelines [23-26]. These techniques integrate LLM outputs with symbolic reasoning engines for robust hybrid verification systems.

Further, formal specification extraction from code enhances traceability and correctness guarantees by enabling automatic annotation and assertion generation directly from source, a feature critical for continuous verification environments [27-28]. Recent work has also explored using LLMs trained on formal verification tool outputs (such as ESBMC) for rapid vulnerability detection through learned pattern recognition [22], demonstrating the potential of combining machine learning with formal methods for practical bug detection at scale.

Our work follows this evolving landscape by exploiting LLMs for automated Python-to-C translation to enable bug detection with bounded model checking tools [3,29]. This approach integrates formal verification tools with AI-driven translation for increased bug detection coverage, though without formal guaranties of semantic equivalence between Python and C code.

2.3 Code Translation and Verification Pipelines

Several transpilation systems have emerged to convert code between languages with varying degrees of formal support:

- TransCoder [30,31] is a neural code translation model capable of zero-shot[5] translation across multiple programming languages, substantially improving transfer between Python, Java, and C++ by learning from large multilingual corpora. However, it does not inherently guarantee semantic preservation or produce verifiable output.
- CoQ-of-Python [35] aims to transpile Python code into Rocq formal specifications, focusing on correctness proofs via dependent types. Its approach offers strong theoretical guarantees but faces fundamental scalability challenges: interactive theorem proving remains difficult even for simple programs in statically-typed languages, requiring substantial expert effort to construct proofs. Python's dynamic features and rich semantics compound these inherent difficulties of theorem proving.
- LLMLift [29] combines LLM-based transpilation with formal verification checkpoints, translating between general-purpose languages (C, C++, Java) and domain-specific verification languages. While it enables migration of legacy systems with correctness assurances, the integration remains complex and requires significant engineering effort.
- ESBMC-Python is a bounded model checker for Python programs that transforms Python code into an intermediate representation, which in turn

[5] Zero-shot translation is the ability of a LLM to translate between two languages it has never seen paired together during training.

is converted into formulae evaluated with SMT solvers [34]. It represents the first BMC-based Python-code verifier, demonstrating effectiveness on Ethereum Consensus Specification. As a direct verifier rather than a transpiler, ESBMC-Python is conceptually closer to tools like VeriFast: it operates directly on Python's intermediate representation without cross-language translation, requiring a custom BMC implementation tailored to Python semantics.

– PyVeritas [3] integrates LLM-based Python-to-C transpilation with bounded model checking via CBMC and MaxSAT-based fault localization, automatically producing C code suitable for verification with back-mapping to Python source. Like all functional correctness verification approaches, PyVeritas requires assertions to specify desired properties. It targets Python programs with numeric computations and array manipulations, demonstrating effectiveness in detecting arithmetic overflows, array bounds violations, and assertion failures.

Direct verification tools such as VeriFast [32,33] have demonstrated success for statically-typed languages like C, C++, and Java through annotation-based verification using separation logic. While VeriFast internally translates programs to verification conditions checked by SMT solvers (as do most verification tools), this internal translation differs from the cross-language transpilation approaches discussed above. While ESBMC-Python demonstrates that direct BMC can be extended to Python's intermediate representation, supporting Python's full dynamic feature set in a direct verifier remains challenging; this motivates translation-based strategies that reuse mature C verification infrastructure.

Our approach differs from ESBMC-Python and PyVeritas in key ways: ESBMC-Python performs direct bounded model checking on Python's intermediate representation without cross-language translation (it's a direct verifier, not a transpiler), requiring custom implementation of BMC for Python semantics. PyVeritas, like our work, uses LLM-based Python-to-C translation with CBMC for verification. Our contribution is the orchestrated multi-tool approach: we combine LLM orchestration for tool selection (using AST analysis to identify which verification tools to apply), adaptive parameter configuration (ESBMC bounds and check selection), and integration of complementary analysis tools (static analyzers, dynamic testing, runtime deadlock detection) coordinated by an LLM. This allows our system to handle diverse bug categories (arithmetic, bounds, concurrency) by selecting appropriate tools rather than applying a single verification technique to all code.

Our contribution is the orchestrated multi-tool pipeline: the LLM analyzes code characteristics via AST analysis to select appropriate verification tools, configures ESBMC parameters (bounds, checks) based on code patterns, and iteratively refines the approach. This enables effective bug hunting even in the absence of formal translation guarantees, as validated by our experimental results.

3 Methodology

3.1 Overall Architecture

The core of our system is a multi-agent architecture where an LLM orchestrates verification processes. Our tool, the Enhanced Verification Agent (EVA), coordinates analysis tools and adapts strategies based on analysis and verification results. The LLM orchestrator is pluggable; our implementation uses Claude Sonnet 4.5, though other capable models could be substituted. The architecture, illustrated in Figure 2, unifies four distinct classes of analysis tools to address separate verification goals. When a user submits Python code to EVA, the LLM orchestrator initializes and iteratively selects and coordinates appropriate tools, guided by code features and prior analysis outcomes.

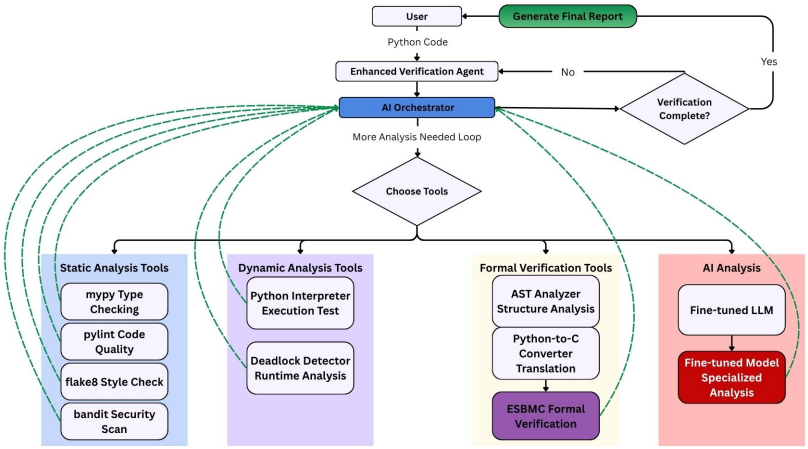

Fig. 2. Complete System Architecture. The LLM orchestrator coordinates verification by routing analysis through four tool categories based on code characteristics: Static Analysis, Dynamic Analysis, Formal Verification, and AI Analysis. The system iterates up to 10 times, synthesizing results until verification objectives are met, then generates a final report.

Static Analysis tools examine code structure without execution. This category includes mypy [45] for type checking and type inference validation, pylint [46] for code quality assessment and anti-pattern detection, flake8 [47] for style consistency verification, and bandit [48] for security vulnerability identification. These tools provide rapid initial feedback on common issues and help narrow the scope for slower analyses.

Dynamic Analysis tools evaluate runtime behavior through partial or complete code execution. The Python Interpreter performs execution-based testing to detect runtime errors, boundary condition violations, and edge case failures that static analysis cannot identify. Dynamic testing executes the Python code once with the concrete inputs already present in the submitted code (i.e., the values hardcoded or initialized in the program as submitted by the user) to detect runtime errors. The Deadlock Detector is a custom runtime monitor implemented as part of EVA that instruments Python `threading.Lock` operations, executes the threaded code, tracks lock acquisition order, detects circular wait conditions (deadlock), and uses timeouts to catch actual deadlocks.

Formal Verification tools provide rigorous verification of program properties on the translated C code. The AST Analyzer examines the abstract syntax tree structure to identify verification-relevant patterns and code characteristics. The Python-to-C Converter translates Python code into C code that preserves the properties under verification (arithmetic behavior, array accesses, assertions) to the extent achievable by LLM-based translation. ESBMC conducts bounded model checking on the translated C code to check properties such as arithmetic overflow absence, array bounds compliance, assertion violations (Python assert statements become C `assert()` or `__ESBMC_assert()`—the latter is an ESBMC intrinsic that behaves like `assert()` but is recognized directly by ESBMC's symbolic execution engine, enabling richer counterexample generation without triggering C runtime abort semantics), and deadlock freedom. Since full semantic equivalence between Python and C cannot be formally guaranteed, findings are interpreted as evidence of bugs in the original Python code rather than formal proofs of correctness.

AI-powered analysis tools leverage learning to support specialized verification tasks. Fine-tuning refers to additional supervised training of a pre-trained model on a domain-specific dataset; in our case a DeepSeek Coder 6.7B model is further trained on ESBMC-verified examples to recognise bug patterns without symbolic execution. The fine-tuned LLM performs deep pattern recognition for complex bug categories, including arithmetic overflows, buffer violations, and concurrency errors. This approach draws inspiration from recent work on using LLMs trained on formal verification tool outputs for vulnerability detection [22]. When formal verification proves too resource-intensive or inconclusive, the fine-tuned model conducts specialised analysis, providing probabilistic assessments of code correctness based on learned patterns from verified codebases.

The AI Orchestrator intelligently coordinates verification by maintaining a conversation history that tracks: (1) which tools have been invoked and their results, (2) what issues have been identified, and (3) what verification remains. At each iteration (Algorithm 1), the LLM analyzes this history and selects the next tool based on code characteristics: mypy for type-annotated code, the Deadlock

Algorithm 1 LLM-Guided Multi-Tool Verification

Input: Python program P
Output: Verification report R
Initialize message history H with:
(1) verification strategy instructions (the LLM system prompt encoding tool selection logic), and
(2) program P
for $i = 1$ to *MAX_ITERATIONS* **do**
  $(T, C) \leftarrow$ LLM_Plan(H) ▷ T: requested tool invocations, C: commentary
  if $T = \emptyset$ **then**
    return FinalReport(C, H)
  end if
  for each tool invocation $t \in T$ **do**
    $r \leftarrow$ ExecuteTool(t, P)
    Append r to H
  end for
end for
return FinalReport("Max iterations reached", H)

Detector for threading, ESBMC with overflow checks for arithmetic operations, and ESBMC with bounds checking for array accesses. This prevents redundant tool invocations (the conversation history shows what has been done) and enables progressive refinement: early iterations use fast tools (AST analysis, static analysis) to understand code structure, guiding later expensive formal verification (ESBMC). The orchestrator decides when verification is complete by synthesizing findings: if ESBMC proves a property, verification succeeds; if ESBMC finds a counterexample, the bug is reported; if fast tools find no issues after appropriate coverage, verification concludes.

3.2 Translation of Python to C for Formal Verification

LLM-Based Translation for ESBMC Compatibility A core challenge for the formal verification of Python is the mismatch between Python's dynamic behavior and the requirements of model checkers such as ESBMC, which expect statically typed C code. Our approach uses the LLM orchestrator to translate Python into C while preserving semantics relevant for verification. The LLM handles this translation by understanding Python semantics and generating equivalent C code that addresses key incompatibilities:

- Python's dynamic typing is mapped to explicit static C types, guided by type hints where available. The LLM infers appropriate C types based on variable usage patterns and context.
- Python's reference semantics and automatic memory management are translated to explicit pointer operations and manual memory allocation in C where necessary.

– High-level Python structures (lists, dictionaries, sets) are mapped to appropriate C representations (arrays, structs) with bounded sizes suitable for BMC. The LLM determines reasonable bounds based on code analysis.

The LLM-generated C code is designed for bounded model checking of specific properties (overflow, bounds violations, deadlocks). We do not claim full semantic equivalence—the LLM may introduce translation errors, so our approach is best characterized as bug hunting rather than formal verification of the Python code. Exception-handling mechanisms are converted to explicit error codes and conditional checks. The LLM translation targets Python programs amenable to BMC: those with bounded loops, properties expressible as assertions, and behavior suitable for bounded analysis. Complex metaprogramming and dynamic code generation remain challenging.

The translation is performed by the LLM orchestrator through its tool-use capability. When the orchestrator determines that formal verification is needed, it invokes a Python-to-C conversion tool that leverages the LLM's code understanding and generation abilities. The LLM is instructed via its system prompt to perform the translation following these steps: First, analyzing the Python code structure to identify type hints, infer variable types from usage patterns, detect memory access patterns, and understand the semantics of Python constructs. Second, generating equivalent C code where function definitions preserve signatures (using type hints for parameter/return types), nondeterministic value generation is translated to appropriate ESBMC C declarations (benchmark programs use `esbmc.nondet_*()` calls—e.g., `esbmc.nondet_int()`—to declare symbolic inputs that range over all possible values, enabling ESBMC to exhaustively check all execution paths up to bound k; these are translated to the corresponding ESBMC C nondeterminism intrinsics), Python data structures are represented as bounded C equivalents (lists as arrays with size tracking), and assertions are converted to `__ESBMC_assert()` or `assert()` statements. Third, the LLM is prompted to instrument the code for verification by adding explicit bounds checks for array accesses, overflow detection for arithmetic operations when needed, and converting exception handling to return codes with conditional checks.

The LLM prioritizes verification requirements over optimization, generating C code in which the properties under verification are explicit and verifiable. This approach can handle more complex Python constructs than rule-based translation, as the LLM can reason about semantics and adapt the translation strategy based on the specific verification goals identified by AST analysis.

Iterative Refinement for Verification Tractability The initial LLM-generated C code may be too complex for ESBMC to analyze within practical time limits. When ESBMC times out or exceeds resource constraints, the system provides feedback to the LLM orchestrator, which can regenerate simpler C code in subsequent iterations. The LLM applies simplification strategies such as adding `__ESBMC_assume()` constraints to bound nondeterministic values, simplifying loop structures, reducing input domains, or abstracting auxiliary functions while

preserving properties under verification. The automated retry logic first attempts reduced unwind bounds and disabled expensive checks before asking the LLM to regenerate code.

We recall that we do not formally verify semantic equivalence between Python and C code—this would require proof techniques beyond BMC. Our goal is bug-finding through deeper analysis than testing alone, not proof of correctness. The system includes dynamic execution of Python code as one verification tool, which can detect runtime errors and provide confidence in basic functionality before expensive formal verification is attempted on the translated C code.

Spurious Faults and Translation Quality A natural concern is whether translation errors introduced by the LLM could produce *spurious* ESBMC reports—counterexamples that reflect bugs in the C translation rather than bugs in the original Python code. In principle this is possible: if the LLM mistranslates a Python construct into semantically different C code, ESBMC may find a violation that does not correspond to any real Python execution.

EVA mitigates this risk in two ways. First, the Python interpreter is always invoked as a dynamic analysis tool: after ESBMC reports a counterexample, the orchestrator can attempt to reproduce the failure by executing the original Python code with the concrete input values from the counterexample. If the Python execution does not fail, the result is flagged as a potential translation artefact rather than a confirmed bug. Second, the iterative refinement loop allows the LLM to regenerate the C translation when ESBMC produces results that appear inconsistent with the Python execution outcome, reducing the likelihood of persistent spurious reports.

In our benchmark of 23 programs (15–50 lines, bounded loops, statically-sized data structures), we observed no spurious counterexamples: every ESBMC violation was reproducible in the original Python code or corresponded to an arithmetic overflow that Python's arbitrary-precision integers silently mask. The small program size is a contributing factor—shorter, structurally simple programs are less likely to expose LLM translation errors than large codebases with complex control flow. Spurious faults remain a known limitation for larger programs, and cross-validating ESBMC counterexamples against Python execution is the recommended mitigation strategy.

Figure 3 shows the orchestrator's decision logic. Figure 4 illustrates the complete sequence from user submission through iterative tool selection to final report. When ESBMC times out, the system's retry logic automatically attempts simplification: reducing the unwind bound (halved from the default of 10), disabling expensive checks (overflow, memory-leak), and suggesting to the LLM to add `__ESBMC_assume()` constraints or simplify loops in subsequent iterations.

4 Implementation

The Enhanced Verification Agent is implemented in approximately 2000 lines of Python 3.11+ code using the Anthropic Claude API (claude-sonnet-4.5) as the

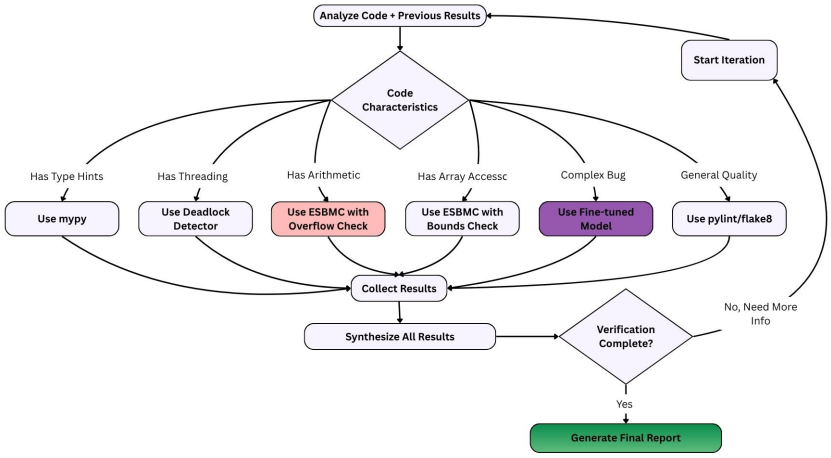

Fig. 3. Decision Logic Flowchart showing tool routing based on code characteristics. This flowchart represents the decision logic explicitly encoded in the LLM system prompt: the prompt instructs the orchestrator to apply mypy for type-annotated code, the Deadlock Detector for threading code, ESBMC overflow checks for arithmetic operations, and ESBMC bounds checks for array accesses. The flowchart was derived directly from these system prompt instructions.

central orchestrator. Tool versions: mypy 1.8+, pylint 3.0+, flake8 7.0+, bandit 1.7+, ESBMC 7.4, Python 3.11+.

Tool outputs follow a unified JSON schema. The orchestrator deduplicates issues and applies a priority hierarchy: formal proofs override all findings, runtime failures override static warnings, and static analysis provides baseline assessments. An SQLite database tracks verified properties and tool history, enabling incremental verification that skips re-checking proven properties.

The implementation is containerized with Docker for reproducible environments. All code, configuration files, and instructions for running the system are available at: https://github.com/esbmc/esbmc-python-cpp/tree/main/agent (see `README.md` for setup and usage details). The full benchmark of 23 Python programs used in the evaluation is included in the repository under `agent/benchmark/` to enable reproduction and comparison in future studies.

5 Experimental Evaluation

We evaluated the Enhanced Verification Agent on a diverse benchmark of Python programs to assess its effectiveness in automated formal verification. Our experiments address three research questions: (1) Can the orchestrated multi-tool agent successfully detect bugs in Python programs using formal verification? (2) How

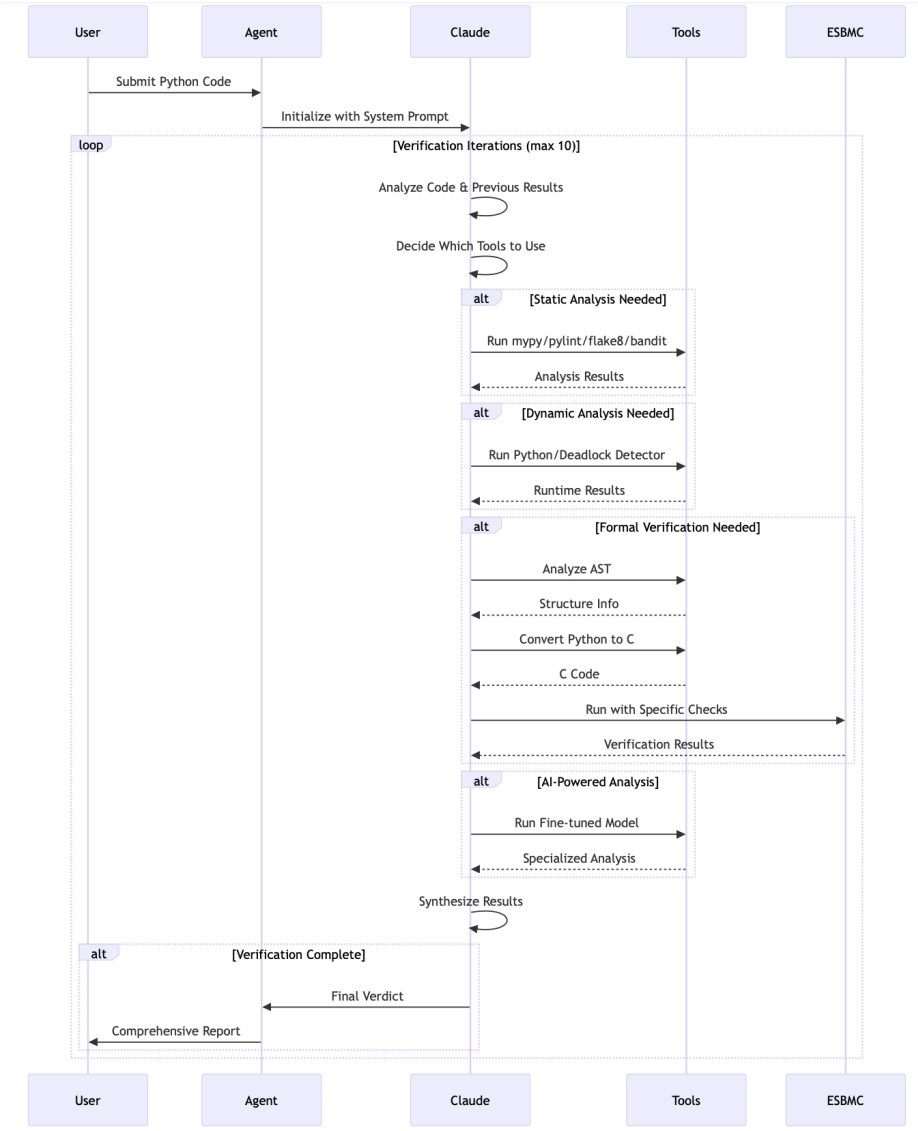

Fig. 4. Sequence Diagram of the Verification Workflow. The user submits Python code to EVA, which initializes the LLM orchestrator (Claude). At each iteration, Claude selects appropriate tools based on code analysis. For formal verification, Claude performs Python-to-C translation (shown as the "Python to C Converter" tool in the diagram) by generating a detailed prompt that describes the translation requirements, invoking itself to generate the C code, and then passing the result to ESBMC for analysis. Claude synthesizes results and decides whether to continue iteration or conclude.

does iterative tool selection improve verification coverage compared to single-tool approaches? (3) Can we accelerate formal verification while maintaining reliability using fine-tuned models?

5.1 Experimental Setup

The benchmark consists of 23 Python programs exhibiting various verification challenges with planted bugs: overflow vulnerabilities (10 programs), bounds violations (8 programs), and race conditions/deadlocks (5 programs). Programs range from 15 to 50 lines of code and represent common bug patterns. Programs use nondeterministic inputs via the ESBMC Python module to enable symbolic execution and bounded model checking.

The system runs with a 10-iteration limit on Macbook Pro M4 Max with 128GB RAM. We also evaluate an optional fine-tuned DeepSeek Coder 6.7B model (with LoRA adapters trained on ESBMC examples, running locally via Apple MLX) that provides rapid pre-screening.

5.2 Verification Results

The orchestrated multi-tool agent successfully detected all planted bugs in the 23 benchmark programs. Table 1 summarizes the results by bug category, with an average of 7.6 iterations required.

Table 1. Verification results by bug category

Bug Category	Programs	Bugs Detected	Avg. Iterations	Detection Rate
Overflow	10	10	7.6	100%
Bounds Violation	8	8	7.4	100%
Race/Deadlock	5	5	7.8	100%
Total	**23**	**23**	**7.6**	**100%**

The orchestrator's iterative tool selection ensured comprehensive analysis, using an average of 6.8 tools per program. The detailed breakdown of detection results by tool type is discussed in Section 6.3, which shows that bugs in this benchmark required formal verification for definitive detection—dynamic execution detected only 3 of 8 bounds violations and 0 of 10 overflow bugs, while static analysis flagged suspicious patterns without confirming actual bugs. All bugs manifested as assertion violations in the formal verification results, with ESBMC providing concrete counterexamples.

Example: Pythagorean Triple Checker. Figure 5 shows a program that checks whether $x^2 + y^2 \neq z^2$ for nondeterministic integers. The agent's iterative process (Figure 6) applied six tools across seven iterations. Initial static analysis (mypy, pylint, flake8, bandit) found no issues. Dynamic analysis with Python interpreter passed. The critical step was formal verification: the agent

```
import random

def main():
    x = random.randint(1, 16383)  # Equivalent to x > 0 && x < 16384
    y = random.randint(1, 16383)  # Equivalent to y > 0 && y < 16384
    z = random.randint(1, 16383)  # Equivalent to z > 0 && z < 16384

    # Assert that the condition is not satisfied
    assert x * x + y * y != z * z, "Assertion failed: x^2 + y^2 == z^2"

    return 0

if __name__ == "__main__":
    main()
```

Fig. 5. Python program for Pythagorean triple verification. The code generates random integers x, y, z in range [1, 16383] and asserts that $x^2 + y^2 \neq z^2$. This represents a verification challenge requiring formal methods to find counterexamples.

identified the assertion as a verification target, used the LLM to translate Python to C, and invoked ESBMC with overflow and bounds checks. ESBMC successfully found the counterexample (e.g., x=6, y=8, z=10 satisfies the Pythagorean relation).

```
============================================================
 EVA - Enhanced Verification Agent | Final Report
============================================================

 [*] Severity Assessment
 ------------------------------------------------------------
  Logical Correctness  : FAILED -- assertion can be violated
  Code Quality         : Minor style issues
  Security             : No issues
  Type Safety          : No issues
  Style Compliance     : No issues

 [*] Recommendations
 ------------------------------------------------------------
  1. Exclude Pythagorean triples if they are not intended.
  2. Replace the assertion with an explicit check if such
     cases are valid.
  3. Document the intended behavior of the code.

 [*] Educational Value
 ------------------------------------------------------------
  This example shows why formal verification can outperform
  random testing, especially when rare mathematical edge
  cases are involved.

 [*] Conclusion
 ------------------------------------------------------------
  A provable bug exists: the assertion fails for Pythagorean
  triples such as (6,8,10). ESBMC detected the counterexample
  even though random testing did not.

============================================================
 Verification Summary
============================================================
  Iterations    : 7
  Verified      : No
  Primary check : overflow
  Generated     : converted_code.c, esbmc_verify.c
============================================================
```

Fig. 6. Verification report generated by EVA for the Pythagorean triple checker, shown as rendered in the terminal during an actual run. The LLM orchestrator provides a natural language explanation of the violation, a severity assessment, and actionable recommendations, making formal verification results accessible to developers.

5.3 Accelerated Verification with Fine-tuned Models

To address research question (3) on accelerating formal verification, we evaluated a fine-tuned DeepSeek Coder 6.7B model (enhanced with LoRA adapters trained on 1000+ ESBMC-verified examples, running locally via Apple MLX) as a rapid pre-screening alternative. This approach builds on prior work using LLMs trained on formal verification tool outputs for vulnerability detection [22], extending the concept from binary classification to bug pattern recognition. The key motivation is speed: ESBMC requires 35-50 seconds per program, while the fine-tuned model completes analysis in 2-10 seconds (depending on code complexity)—up to a 20x speedup. This acceleration comes from learned pattern recognition rather than symbolic execution, trading formal guarantees for probabilistic assessments. The orchestrator (Claude Sonnet 4.5) remains responsible for tool coordination; the fine-tuned local model is invoked as an optional analysis step at any point in the pipeline.

Figure 7 demonstrates the fine-tuned model's operation on a more complex concurrent program, with analysis completing in under 10 seconds using modest resources (2632 prompt tokens, 200 generation tokens at 26k tokens/sec).

5.4 Iteration Analysis

Convergence occurred within 6-10 iterations (median: 8, mean: 7.6) across all 23 programs. For this benchmark of programs with planted bugs, formal verification via ESBMC was necessary for definitive bug detection with concrete counterexamples. Static analysis tools identified suspicious patterns but could not confirm bugs, while dynamic analysis detected some runtime failures but lacked systematic coverage. The LLM orchestrator selectively invoked tools based on code characteristics rather than running all tools in parallel on every program. Notably, the 5 concurrency programs were resolved entirely by the runtime deadlock detector without invoking ESBMC or C translation; the 100% ESBMC invocation rate for arithmetic and bounds programs reflects that all 18 such benchmark programs were specifically designed to require formal verification. In a real codebase, a significant fraction of functions would be resolved by static or dynamic analysis alone. While parallel execution is technically possible, it would increase unnecessary computation (running ESBMC on code without arithmetic operations, deadlock detection on sequential code) and LLM API costs (more tool results to process). The orchestrator's adaptive strategy—using AST analysis to guide tool selection—balances thoroughness with efficiency, applying expensive formal verification only when code patterns warrant it.

Mean verification time per program was approximately 45-60 seconds for the full orchestrated agent. The breakdown: static analysis (2-3s), dynamic analysis (3-5s where applicable), formal verification (35-50s), and orchestration overhead (5-7s). The iterative approach uses an average of 6.8 tools per program.

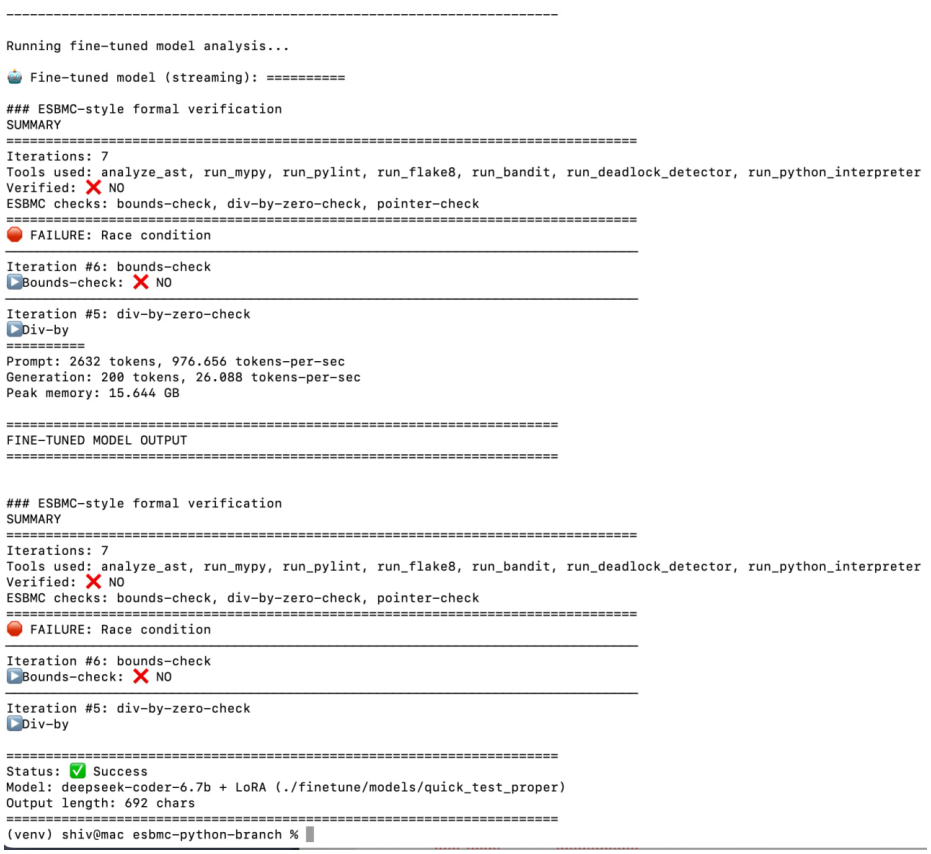

Fig. 7. Fine-tuned DeepSeek Coder 6.7B (LoRA, Apple MLX) analysis output for a concurrent program with race condition (`example_race_condition.py` from the benchmark). The model performs rapid bug pattern recognition by analyzing Python code directly, without Python-to-C translation or symbolic execution. Performance: 2632 prompt tokens, 200 generation tokens at 26k tokens/sec, 15.6GB peak memory. Analysis completes in under 10 seconds, demonstrating the speed advantage over the full ESBMC pipeline (35–50s). The model output (shown in the figure) includes a predicted bug category, a confidence assessment, and a natural-language reasoning trace identifying the problematic lock-acquisition pattern. The EVA orchestrator (Claude Sonnet 4.5) invokes this model as an optional analysis step.

5.5 Bug Detection Effectiveness

The orchestrated agent successfully detected all planted bugs in the 23 benchmark programs. Key findings by bug category:

Overflow vulnerabilities (10 programs): Static analysis (pylint) flagged potential arithmetic issues in 7 programs but could not confirm overflows. Dynamic testing with random inputs detected 0 overflows (as overflow conditions require specific nondeterministic value combinations). ESBMC with overflow checking detected all 10 bugs with concrete counterexamples.

Bounds violations (8 programs): Static analysis flagged suspicious array accesses in 4 programs. Dynamic testing caught 3 bounds violations through runtime exceptions. ESBMC detected all 8 bugs, including 5 that dynamic testing missed.

Race conditions and deadlocks (5 programs): Static analysis detected threading usage but not deadlocks. The runtime deadlock detector identified all 5 bugs through lock instrumentation and circular wait detection, without requiring C translation. ESBMC's deadlock checking was not used for these programs as the Python-based deadlock detector proved more effective for threading code.

These results show that formal verification via ESBMC is essential for arithmetic and bounds checking bugs, while runtime instrumentation handles concurrency bugs effectively. Static and dynamic analysis provide useful preliminary screening but cannot provide definitive bug detection for this benchmark. Note that our approach of selective verification (using AST analysis to decide which blocks need ESBMC) works well for our small benchmark programs (15-50 lines), but scaling to large programs remains challenging—AST patterns alone cannot reliably predict which functions in a large codebase require formal verification without analyzing the entire program.

Benchmark limitations. Our benchmark consists of 23 self-constructed programs with planted bugs, which carries a risk of selection bias: the programs were designed with the EVA pipeline in mind. We acknowledge this limitation. A more rigorous evaluation would use independently constructed benchmarks; the SV-COMP verification competition [38] provides a well-established set of C verification tasks, and adapting a representative subset to Python is a natural direction for future work to demonstrate broader generality.

5.6 LLM-Only Baseline Comparison

To assess the added value of the EVA pipeline over direct LLM-based bug analysis, we conducted a baseline experiment in which Claude Sonnet 4.6 was queried directly on all 23 benchmark programs, without any translation or formal verification toolchain. The prompt described the semantics of `esbmc.nondet_*()` and asked the model to identify correctness issues and provide a concrete triggering input. No hints about the type of bug were given.

The LLM-only baseline detected 18 of 23 bugs (78%), with all 5 misses occurring in the overflow category. Three missed programs (*Checksum*, *Balance accumulation*, *Modular arithmetic*) contain no explicit `assert` statement—the

Table 2. Bug detection across all 23 benchmark programs: LLM-only baseline vs. EVA. ✓= correctly identified; × = missed.

Category	Program	LLM-Only	EVA
Overflow	Product of range overflow	×	✓
	Circle circumference overflow	✓	✓
	Multiplication overflow	✓	✓
	Pixel calculation overflow	✓	✓
	Factorial overflow	✓	✓
	Checksum overflow	×	✓
	Balance accumulation overflow	×	✓
	Permutation calculation overflow	✓	✓
	Hash function overflow	✓	✓
	Power function overflow	✓	✓
	Modular arithmetic overflow	×	✓
	Pythagorean theorem overflow	✓	✓
	Subtotal	**8/12 (67%)**	**12/12**
Bounds/Other	List comprehension (class-based)	×	✓
	Off-by-one in loop	✓	✓
	Average with zero items	✓	✓
	Modulo by zero	✓	✓
	Dynamic index bounds violation	✓	✓
	Unbounded array access	✓	✓
	Subtotal	**5/6 (83%)**	**6/6**
Concurrency	Check-then-act race condition	✓	✓
	Shared list race condition	✓	✓
	Counter race condition	✓	✓
	Classic two-lock deadlock	✓	✓
	Resource ordering deadlock	✓	✓
	Subtotal	**5/5 (100%)**	**5/5**
Total		**18/23 (78%)**	**23/23 (100%)**

LLM correctly reasoned that Python's arbitrary-precision integers prevent overflow and found no property to violate, whereas ESBMC's `-overflow-check` flag treats any arithmetic overflow in the translated C code as a violation by construction. The remaining two misses (*Product of range*, *List comprehension*) involve non-obvious control flow that the LLM incorrectly concluded was safe. On concurrency bugs, the LLM matched EVA perfectly (5/5), as race conditions and deadlocks are semantically visible from lock-acquisition patterns without requiring formal analysis.

Beyond detection rate, the LLM-only baseline provides no formal guarantee: its verdicts are probabilistic assessments without verified counterexamples and may include false positives. EVA produces verified counterexamples via ESBMC's symbolic execution. These results confirm that the Python-to-C translation and bounded model checking pipeline adds concrete value over direct LLM analysis, particularly for overflow bugs without explicit assertions and for subtle conditions hidden behind guard clauses.

6 Future Work

While our current system demonstrates effective bug detection for standalone Python programs, several promising directions remain for future research.

Expanding the evaluation to independently constructed benchmarks is an important next step. The SV-COMP verification competition [38] provides a large, well-established set of C verification tasks; adapting a representative subset to Python would enable comparison with established verification tools and reduce selection bias inherent in self-constructed benchmarks.

Future work should train more powerful fine-tuned models on larger datasets (10,000+ examples) to approach ESBMC-comparable accuracy while maintaining speed advantages. The ideal workflow combines fine-tuned model pre-screening (rapid triage) with selective ESBMC verification (formal proofs for critical sections).

Our benchmark consists of standalone programs with localized bugs. Real-world production systems present greater challenges: server-based architectures with distributed components, asynchronous communication patterns, complex state management across multiple services, and emergent behaviors from component interactions. Extending our verification framework to handle multi-file projects, inter-service dependencies, and distributed concurrency requires advances in modular verification, compositional reasoning, and scalable state space exploration. Future work should investigate verification techniques for microservices, REST APIs, message queues, and distributed databases, where bugs manifest through intricate timing dependencies and cross-service invariant violations.

Practical adoption requires seamless integration into continuous integration / deployment pipelines, incremental verification for code changes, and developer-friendly error reporting that maps formal verification counterexamples back to Python source with actionable remediation suggestions.

Acknowledgments The research by Klaus Havelund and Alessandro Pinto was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. © 2026.

References

1. Amusuo, P.C., Cochell, O., Le Lievre, T., Patil, P.V., Machiry, A., Davis, J.C.: An Evaluation of the Use of Bounded Model Checking for Memory Safety Verification of Embedded Network Stacks. arXiv preprint arXiv:2503.13762v1 (2025). https://arxiv.org/html/2503.13762v1

2. Masoudi, T.: Investigating the Role of Formal Verification in Software Development: From Automatic Specification Generation to Usability of Verification Languages. In: FSE Companion '25: Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering, pp. 1273–1276. ACM, New York, NY, USA (2025). https://doi.org/10.1145/3696630.3731465
3. Orvalho, P., Kwiatkowska, M.: PyVeritas: On Verifying Python via LLM-Based Transpilation and Bounded Model Checking for C. arXiv preprint arXiv:2508.08171 (2025). https://arxiv.org/abs/2508.08171
4. Siavash, N., Moin, A.: LLM-Powered Quantum Code Transpilation. arXiv preprint arXiv:2507.12480 (2025). https://arxiv.org/abs/2507.12480
5. Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21(4), 181–185 (1985)
6. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing 2(3), 117–126 (1987)
7. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic Model Checking without BDDs. In: Cleaveland, W.R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 1999. Lecture Notes in Computer Science, vol. 1579, pp. 193–207. Springer, Berlin, Heidelberg (1999)
8. Clarke, E.M., Klieber, W., Nováček, M., Zuliani, P.: Model Checking and the State Explosion Problem. In: Meyer, B., Nordio, M. (eds.) Tools for Practical Software Verification. LASER 2011. Lecture Notes in Computer Science, vol. 7682, pp. 1–30. Springer, Berlin, Heidelberg (2012)
9. Kheireddine, A., Renault, E., Baarir, S.: Towards better heuristics for solving bounded model checking problems. Constraints 28, 45–66 (2023)
10. Kroening, D., Tautschnig, M.: CBMC – C Bounded Model Checker. In: Ábrahám, E., Havelund, K. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2014. Lecture Notes in Computer Science, vol. 8413, pp. 389–391. Springer, Berlin, Heidelberg (2014)
11. Kroening, D., Schrammel, P.: CBMC: The C Bounded Model Checker. arXiv preprint arXiv:2302.02384 (2023)
12. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. IEEE Transactions on Software Engineering 38(4), 957–974 (2012)
13. Menezes, R.S., Aldughaim, M., Farias, B., Li, X., Manino, E., Shmarov, F., Song, K., Brauße, F., Gadelha, M.R., Tihanyi, N., Korovin, K., Cordeiro, L.C.: ESBMC v7.4: Harnessing the Power of Intervals. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2024, LNCS, vol. 14572, pp. 376–380. Springer, Cham (2024)
14. Cordeiro, L., Kesseli, P., Kroening, D., Schrammel, P., Trtík, M.: JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification, CAV 2018, LNCS, vol. 10981, pp. 183–190. Springer, Cham (2018)
15. Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: Bounded Model Checking for Java Bytecode. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2019, LNCS, vol. 11429, pp. 219–223. Springer, Cham (2019)
16. Brenguier, R., Cordeiro, L.C., Kroening, D., Schrammel, P.: JBMC: A Bounded Model Checking Tool for Java Bytecode. arXiv preprint arXiv:2302.02381 (2023)
17. Jiang, Y., Li, W., Liu, S., Wu, T., Song, K., Deng, S., Wu, Z., Yue, Y., Deng, Y., Zheng, W., Yang, Z., Liu, Z.: The Fusion of Large Language Models and Formal

Methods for Trustworthy AI Agents: A Roadmap. arXiv preprint arXiv:2412.06512 (2024)
18. Wu, Y., Jiang, A. Q., Li, W., Rabe, M., Staats, C., Jamnik, M., Szegedy, C.: Autoformalization with Large Language Models. In: Advances in Neural Information Processing Systems 35 (NeurIPS 2022), pp. 32353–32368 (2022)
19. Jiang, A. Q., Welleck, S., Zhou, J. P., Lacroix, T., Liu, J., Li, W., Jamnik, M., Lample, G., Wu, Y.: Draft, Sketch, and Prove: Guiding Formal Theorem Provers with Informal Proofs. In: The Eleventh International Conference on Learning Representations (ICLR 2023) (2023)
20. Zhou, J. P., Staats, C., Li, W., Szegedy, C., Weinberger, K. Q., Wu, Y.: Don't Trust: Verify – Grounding LLM Quantitative Reasoning with Autoformalization. In: The Twelfth International Conference on Learning Representations (ICLR 2024)
21. Wu, G., Cao, W., Yao, Y., Wei, H., Chen, T., Ma, X.: LLM Meets Bounded Model Checking: Neuro-symbolic Loop Invariant Inference. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE 2024), pp. 406–417 (2024)
22. Ferrag, M.A., Battah, A., Tihanyi, N., Jain, R., Maimuţ, D., Alwahedi, F., Lestable, T., Thandi, N.S., Mechri, A., Debbah, M., Cordeiro, L.C.: SecureFalcon: Are We There Yet in Automated Software Vulnerability Detection With LLMs? IEEE Transactions on Software Engineering 51(4), 1248–1265 (2025). https://doi.org/10.1109/TSE.2025.3548168
23. Chakraborty, S., Lahiri, S., Fakhoury, S., Lal, A., Musuvathi, M., Rastogi, A., Senthilnathan, A., Sharma, R., Swamy, N.: Ranking LLM-Generated Loop Invariants for Program Verification. In: Findings of the Association for Computational Linguistics: EMNLP 2023, pp. 9164–9175. Association for Computational Linguistics, Singapore (2023)
24. Huang, S., Wang, Z., Li, Y., Song, P., George, R. J., Anandkumar, A.: LeanProgress: Guiding Search for Neural Theorem Proving via Proof Progress Prediction. arXiv preprint arXiv:2502.17925 (2025)
25. Song, P., Yang, K., Anandkumar, A.: Lean Copilot: Large Language Models as Copilots for Theorem Proving in Lean. arXiv preprint arXiv:2404.12534 (2024)
26. Xin, H., Guo, D., Shao, Z., Ren, Z., Zhu, Q., Liu, B., Ruan, C., Li, W., Liang, X.: DeepSeek-Prover: Advancing Theorem Proving in LLMs through Large-Scale Synthetic Data. arXiv preprint arXiv:2405.14333 (2024)
27. Wu, Z., Huang, S., Zhou, Z., Ying, H., Yuan, Z., Zhang, W., Lin, D., Chen, K.: InternLM2.5-StepProver: Advancing Automated Theorem Proving via Critic-Guided Search. arXiv preprint arXiv:2410.15700 (2024)
28. Ma, L., Liu, S., Li, Y., Xie, X., Bu, L.: SpecGen: Automated Generation of Formal Program Specifications via Large Language Models. arXiv preprint arXiv:2401.08807 (2024)
29. Li, H., Dong, Z., Wang, S., Zhang, H., Shen, L., Peng, X., She, D.: Extracting Formal Specifications from Documents Using LLMs for Automated Testing. arXiv preprint arXiv:2504.01294 (2025)
30. Bhatia, S., Qiu, J., Hasabnis, N., Cheung, A., Seshia, S.A.: Verified Code Transpilation with LLMs. In: Advances in Neural Information Processing Systems 37 (NeurIPS 2024), pp. 41394–41424 (2024)
31. Roziere, B., Lachaux, M.-A., Chanussot, L., Lample, G.: Unsupervised Translation of Programming Languages. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M.F., Lin, H. (eds.) Advances in Neural Information Processing Systems 33 (NeurIPS 2020), pp. 20601–20611 (2020)

32. Szafraniec, M., Roziere, B., Leather, H., Charton, F., Labatut, P., Synnaeve, G.: Code Translation with Compiler Representations. arXiv preprint arXiv:2207.03578 (2022)
33. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods Symposium, NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
34. Mommen, N., Jacobs, B.: Verification of C++ Programs with VeriFast. arXiv preprint arXiv:2212.13754 (2022)
35. Farias, B., Menezes, R., de Lima Filho, E.B., Sun, Y., Cordeiro, L.: ESBMC-Python: A Bounded Model Checker for Python Programs. In: Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024), pp. 1836–1840. ACM (2024)
36. Mohamed Amine Ferrag, Ammar Battah, Norbert Tihanyi, Ridhi Jain, Diana Maimut, Fatima Alwahedi, Thierry Lestable, Narinderjit Singh Thandi, Abdechakour Mechri, Mérouane Debbah, Lucas C. Cordeiro: SecureFalcon: Are We There Yet in Automated Software Vulnerability Detection With LLMs? IEEE Trans. Software Eng. 51(4): 1248-1265 (2025)
37. Formal Land. Translation of Python code to Rocq. https://formal.land/blog/2024/05/10/translation-of-python-code (2024)
38. Beyer, D.: Competition on Software Verification (SV-COMP 2024). In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2024. Lecture Notes in Computer Science, vol. 14572, pp. 2–24. Springer, Cham (2024). https://sv-comp.sosy-lab.org/
39. Holzmann, G.J.: The Model Checker SPIN. IEEE Transactions on Software Engineering 23(5), 279–295 (1997)
40. Cimatti, A., Clarke, E., Giunchiglia, F., Giunchiglia, E., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Berlin, Heidelberg (2002)
41. Owre, S., Rushby, J.M., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) CADE-11. LNCS, vol. 607, pp. 748–752. Springer, Berlin, Heidelberg (1992)
42. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer, Berlin, Heidelberg (2002)
43. The Coq Development Team: The Coq Proof Assistant, Reference Manual. https://coq.inria.fr/ (2024)
44. Moura, L., Ullrich, S.: The Lean 4 Theorem Prover and Programming Language. In: Platzer, A., Sutcliffe, G. (eds.) CADE 28. LNCS, vol. 12699, pp. 625–635. Springer, Cham (2021)
45. Lehtosalo, J. et al.: Mypy: Optional Static Typing for Python. https://mypy-lang.org/ (2012)
46. PyCQA: Pylint – Code Analysis for Python. https://pylint.readthedocs.io/ (2001)
47. Cordasco, I. et al.: flake8: Your Tool for Style Guide Enforcement. https://flake8.pycqa.org/ (2010)
48. PyCQA: Bandit – A Security Linter for Python. https://bandit.readthedocs.io/ (2014)