LLM-Assisted Translation and Bounded Model Checking of Python Code

The EVA Verification Agent

AISoLA 2025 — AI Assisted Programming (AIAP) Track

Shivkumar Shivaji Klaus Havelund Natalia Lobakhina
Lucas C. Cordeiro Alessandro Pinto

Generative AI LLC

NASA JPL / Caltech

University of Manchester

UFAM

AIAP — AI Assisted Programming Track, AISoLA 2025

Outline

Why Verify Python?

- Python is widely used in **data science and machine learning**, spanning research and commercial applications
- Mature C/Java verification tools **do not transfer easily**:
  - Dynamic typing, metaprogramming, runtime mutation
  - Heterogeneous data structures (lists, dicts, sets)
- Existing approaches are fragmented:
  - Direct verifiers (ESBMC-Python) need custom semantics
  - Theorem-prover translation (Coq-of-Python) needs expert effort
- **Industrial pain point**: silent runtime bugs (overflow on large ints, off-by-one, deadlocks) escape testing

Real-World Bugs This Targets

- **Arithmetic overflow** — silently masked by Python's arbitrary-precision integers, but real once the same logic runs against fixed-width semantics
- **Array bounds violations** on lists / arrays
- **Race conditions and deadlocks** in threaded Python code
- **Logic bugs** that surface only for rare input combinations (planted-bug benchmark categories from the paper)

Two industrial use cases

1. Bug-finding in production Python code
2. Letting engineers write **design models in Python** (instead of TLA$^+$) and verifying them

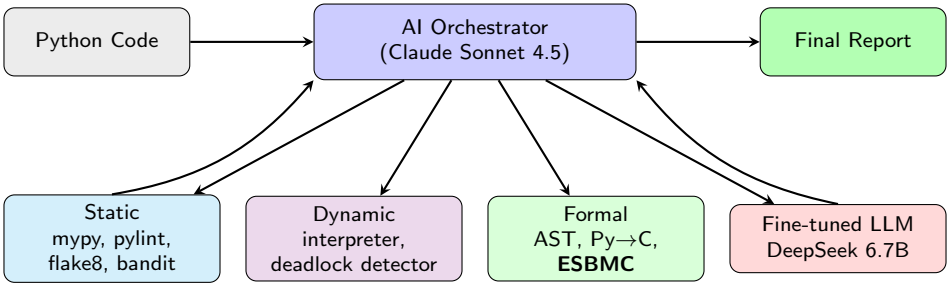
Core Idea — LLM-Orchestrated Verification

- **Reuse mature C verification infrastructure** (ESBMC) by translating Python→C with an LLM
- Wrap translation + BMC in an agent (**EVA — Enhanced Verification Agent**)
- LLM (Claude Sonnet 4.5) acts as an **orchestrator**: picks tools, configures bounds/checks, refines on feedback
- **Bug-hunting, not full equivalence proof**: counterexamples are validated against the original Python interpreter

Honest scoping

No formal proof of Python↔C semantic equivalence. Spurious warnings are mitigated by replaying counterexamples through the Python interpreter.

System Architecture



Four tool classes; orchestrator iterates up to 10 times until verification objectives are met.

The Four Tool Classes

- **Static analysis** — mypy, pylint, flake8, bandit
  Cheap first pass: types, style, security smells
- **Dynamic analysis** — Python interpreter + custom deadlock detector
  Runtime exceptions, lock-acquisition order, circular waits
- **Formal verification** — AST analyzer + Python→C + **ESBMC**
  Overflow, bounds, assertion violations, deadlock freedom
- **AI analysis** — fine-tuned DeepSeek Coder 6.7B (LoRA, Apple MLX)
  Rapid bug pattern recognition without symbolic execution

End-to-End Workflow

1. Engineer submits Python program (with `assert` specifications)
2. Orchestrator inspects the AST and **routes by code characteristics**:
   - Has type hints? $\rightarrow$ mypy
   - Has threading? $\rightarrow$ deadlock detector
   - Has arithmetic? $\rightarrow$ ESBMC `--overflow-check`
   - Has array access? $\rightarrow$ ESBMC bounds check
3. LLM translates Python $\rightarrow$ C (preserving types, bounds, asserts)
4. ESBMC runs BMC on the translated C
5. **Counterexample replayed via the Python interpreter** to flag translation artefacts
6. If timeout: retry with **halved unwind, disabled expensive checks, simplified loops**
7. Final natural-language report with severity, recommendations, generated C

Python→C Translation Strategy

The LLM is instructed to bridge specific incompatibilities:

- Dynamic typing → explicit static C types (using hints & usage patterns)
- Reference semantics → explicit pointers and manual allocation
- Lists / dicts / sets → **bounded** arrays / structs with size tracking
- `esbmc.nondet_*()` → ESBMC C nondeterminism intrinsics
- `assert` → `assert()` or `__ESBMC_assert()`
- Exceptions → explicit return codes / conditional checks

Iterative refinement

On ESBMC timeout, the orchestrator adds `__ESBMC_assume()` bounds, simplifies loops, or asks the LLM to regenerate simpler C.

Example — Pythagorean Triple Checker

```
import random

def main():
    x = random.randint(1, 16383)    # x in [1, 16383]
    y = random.randint(1, 16383)
    z = random.randint(1, 16383)

    # Engineer's spec: no Pythagorean triple should exist
    assert x*x + y*y != z*z, "Assertion failed: x^2 + y^2 == z^2"
    return 0
```

Random testing misses it. EVA in 7 iterations:

- Static + dynamic: no issue found
- Translates to C, invokes ESBMC with overflow + bounds checks
- ESBMC counterexample: `x=6, y=8, z=10`
- Report classifies as *logical correctness FAILED*, suggests fixing the spec

Benchmark and Results

23 self-constructed Python programs (15–50 LOC) with planted bugs:

Bug Category	**Programs**	**Detected**	**Avg. Iter.**
Overflow	10	10	7.6
Bounds violation	8	8	7.4
Race / Deadlock	5	5	7.8
Total	**23**	**23 (100%)**	**7.6**

- Average **6.8 tools per program** — adaptive, not exhaustive
- Mean wall-clock per program: **45–60 s**
- No spurious counterexamples observed in this benchmark

EVA vs. LLM-Only Baseline

Baseline: ask Claude Sonnet 4.6 directly to find bugs (no translation, no BMC).

Category	**LLM-Only**	**EVA**
Overflow (12)	8/12 (67%)	**12/12**
Bounds / Other (6)	5/6 (83%)	**6/6**
Concurrency (5)	5/5 (100%)	5/5
Total (23)	18/23 (78%)	**23/23 (100%)**

- LLM-only **misses overflow** hidden behind guard clauses or absent assertions
- LLM-only gives **no counterexample** — only probabilistic verdicts
- EVA produces **concrete failing inputs** engineers can run

Where Each Tool Earns Its Keep

- **Overflow (10)**: pylint flagged 7, dynamic testing 0, **ESBMC 10/10**
  - Overflows need specific nondet input combinations
- **Bounds (8)**: dynamic testing 3, **ESBMC 8/8** (5 missed by tests)
- **Concurrency (5)**: deadlock detector **5/5** without invoking ESBMC
  - Lock-instrumentation + circular-wait detection beats BMC here

Engineering takeaway

No single technique wins. The orchestrator's value is **routing each bug class to the right tool** based on code features, not running everything in parallel.

Accelerating with a Fine-Tuned Local Model

- Fine-tuned **DeepSeek Coder 6.7B** (LoRA, Apple MLX) on 1000+ ESBMC-verified examples
- Pattern recognition without symbolic execution
- **ESBMC**: 35–50 s per program
- **Fine-tuned model**: 2–10 s per program (**up to 20× speedup**)
- Used as *rapid pre-screening*; ESBMC remains the source of truth for formal claims

Practical pipeline

Local fine-tuned triage → selective ESBMC for critical sections.
Trades formal guarantees for speed where speed matters.

Implementation and Deployment

- $\sim$2,000 lines of Python 3.11+, Anthropic Claude API as the orchestrator
- **Containerized with Docker** for reproducible environments
- Tools emit a **unified JSON schema**; orchestrator deduplicates issues with a priority hierarchy (formal $>$ runtime $>$ static)
- **SQLite** tracks verified properties and tool history — enables **incremental verification** that skips re-checking proven properties
- Counterexamples are concrete input values that can be replayed in the Python interpreter
- Targeted use cases (per paper): bug-finding on production Python, and design-model checking as a Python alternative to TLA$^+$
- Repository: `github.com/esbmc/esbmc-python-cpp`

Note: seamless CI/CD integration is identified as future work.

Trade-Offs and Limitations

Strengths

- Reuses mature C verifier (ESBMC)
- 100% detection on benchmark
- Concrete counterexamples
- Engineer-friendly final reports
- Pluggable LLM backend

Limitations

- Bug-hunting, **not** equivalence proof
- 15–50 LOC, bounded loops, statically-sized data
- Self-constructed benchmark (selection bias)
- Spurious-counterexample risk grows with code size
- Distributed / multi-file systems not yet supported

Future Work and Implications

Future directions

- Adapt SV-COMP tasks to Python for unbiased benchmarking
- Train fine-tuned models on **10,000+** examples for ESBMC-comparable accuracy
- Multi-file projects, microservices, distributed concurrency
- Tighter CI integration with **incremental** verification on diffs

What this means for engineering teams

- Lowers the barrier to applying mature C verifiers to Python code
- LLM handles translation; ESBMC delivers **concrete counterexamples on the translated C**, then they are cross-checked against the Python interpreter
- Practical bug-hunting today; *not* a proof of Python correctness

Key Takeaways

1. **LLM + BMC** makes mature C verifiers accessible to Python developers
2. **Orchestration matters more than any single tool** — route bugs to the technique that solves them
3. **100% detection** on a 23-program benchmark covering overflow, bounds, and concurrency bugs
4. **Concrete counterexamples** (not probabilistic verdicts) make findings actionable
5. **Honest about scope**: bug-hunting (not equivalence proof), small programs (15–50 LOC), self-constructed benchmark — broader productivity impact remains to be evaluated

Questions?

`github.com/esbmc/esbmc-python-cpp`