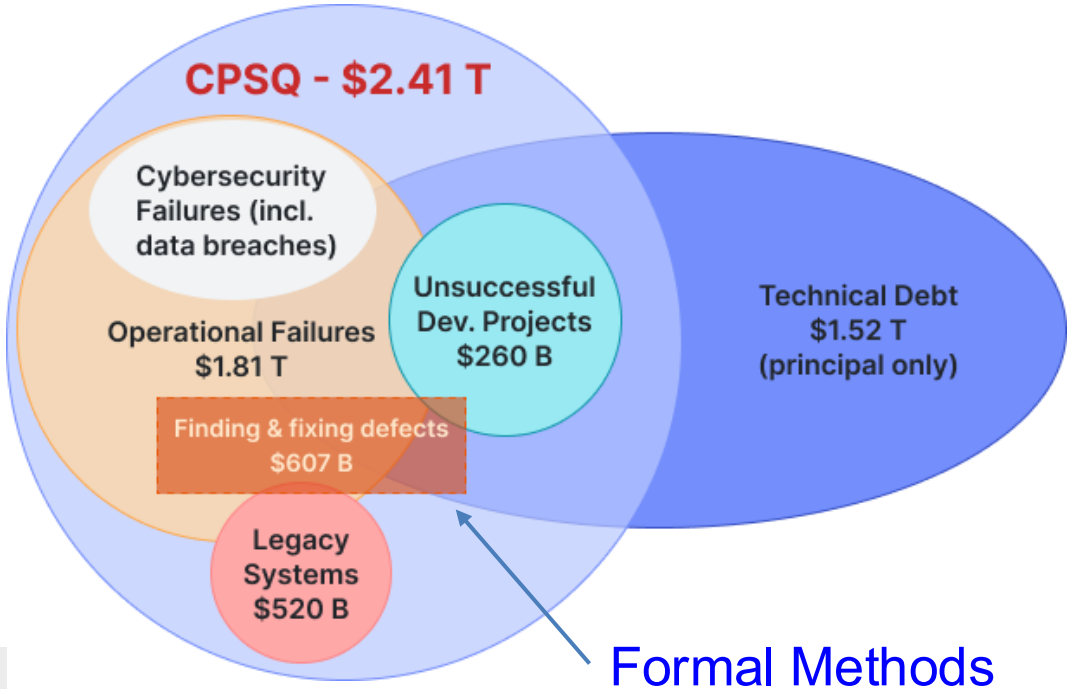
Intl. Joint Conf. On Theory and Practice of Software (ETAPS)
7th Workshop on Cooperative Software Verification (COOP 2026)

Cooperative AI-Assisted Formal Verification: Integrating LLMs with Model Checking for Scalable Software Assurance

Lucas C. Cordeiro
Department of Computer Science
lucas.cordeiro@manchester.ac.uk
https://ssvlab.github.io/lucasccordeiro/

The $2.41 Trillion Problem

Poor software quality cost US companies **$2.41 trillion in 2022**, while the **accumulated software Technical Debt** (TD) has grown to **~$1.52 trillion**



→ 50%+ of project costs go to debugging, not building

→ AI-generated code accelerates development, but also amplifies hidden defects

→ Current workflows prioritize speed over correctness

CISQ Consortium for Information & Software Quality™

2025 Quality Transformation Report: Closing the gap between speed, quality, and trust in the AI era

Tricentis

From AI-Assisted to Agentic Coding

*“Google and Microsoft both report that **25–30% of their new code is AI-generated**. Microsoft's CTO predicts that **95% of all code will be AI-generated by 2030 [1]”***

Human-Driven
Manual debugging
Root cause analysis

AI-Assisted
Code suggestions
Test generation

Agentic Coding
Autonomous debugging/coding
Self-validation

From writing code to delegating outcomes to AI agents, but **who verifies that code**?

[1] https://www.techspot.com/news/107411-microsoft-cto-predicts-ai-generate-95-percent-code.html

How Secure is AI-Generated Code?

Category	Avg Prop. Viol. per Line	Rank	$\mathcal{VS}$	Rank	$\mathcal{VF}$	$\mathcal{VU}$ (Timeout) 500s	Avg Prop. Viol. per File
GPT-4o-mini	**0.0165**	3	4.23%	2	57.14%	36.77%	3.40
Llama2-13B	0.0234	2	12.36%	**1**	**51.30%**	31.78%	3.62
Mistral-7B	0.0254	7	8.36%	4	62.08%	25.88%	**3.07**
CodeLlama-13B	0.0260	**1**	**15.48%**	3	52.71%	29.52%	4.13
Falcon-180B	0.0291	8	6.48%	5	62.07%	28.67%	3.38
GPT-3.5-turbo	0.0295	6	7.29%	7	65.07%	26.09%	4.42
Gemini Pro 1.0	0.0305	5	9.49%	6	63.91%	24.13%	4.70
Gemma-7B	0.0437	4	11.62%	8	67.01%	16.30%	4.20

Legend:

$\mathcal{VS}$:0.0234 Verification Success; $\mathcal{VF}$: Verification Failed; $\mathcal{VU}$: Verification Unknown (Timeout).

Best performance in a category is highlighted with bold and/or Rank.

LLMs generate code quickly, but lack guarantees

Tihanyi, N., Bisztray, T., Ferrag, M., Jain, R., Cordeiro, L.: *How secure is AI-generated code: a large-scale comparison of large language models*. Empir. Softw. Eng. 30(2): 47 (2025)

Why Formal Methods?

The White House Makes It Clear (Feb 2024):

"While formal methods have been studied for decades, their deployment remains limited; further innovation in approaches…is vital to accelerate broad adoption."

Formal methods can:

- ✓ Prove software correctness
- ✓ Guarantee safety/security properties
- ✓ Find bugs exhaustively

But they require:

- Expert knowledge
- Manual effort
- Specialized training

DO-178C incorporates formal methods through its DO-333 supplement

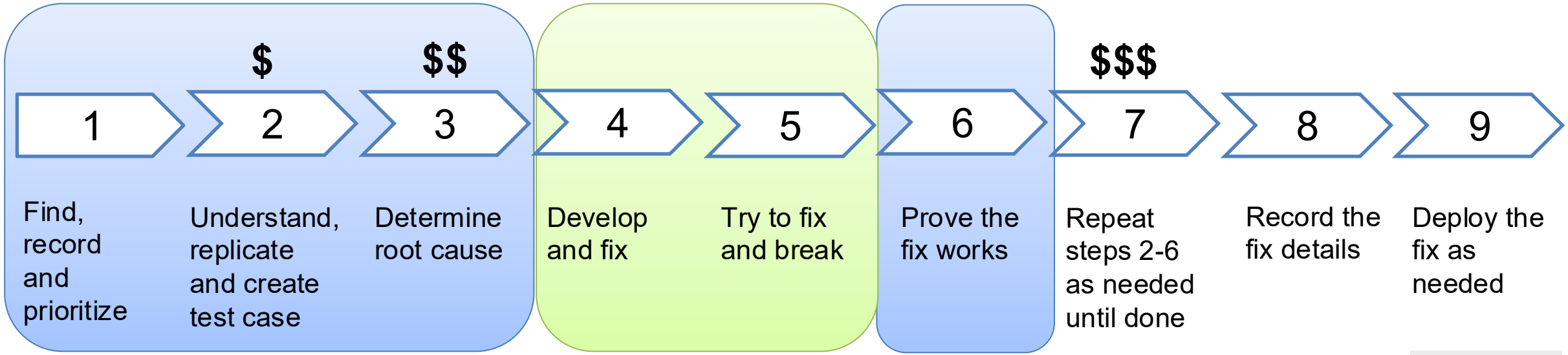
Can cooperation between **LLM** and **verification** give us both **speed** and **correctness**?

AI is evolving from assisting developers to acting autonomously

Traditional Debugging/Verification (Human-Driven)

Agentic AI (plan, act, evaluate, and iterate)

- Plan multi-step solutions
- Act using tools (tests, compilers, verifiers)
- Evaluate results
- Iterate autonomously toward goals



CISQ
Consortium for Information & Software Quality™

Objective of this talk

Discuss how **LLMs and formal methods** can work together to establish a foundation **for building trustworthy software and AI systems**

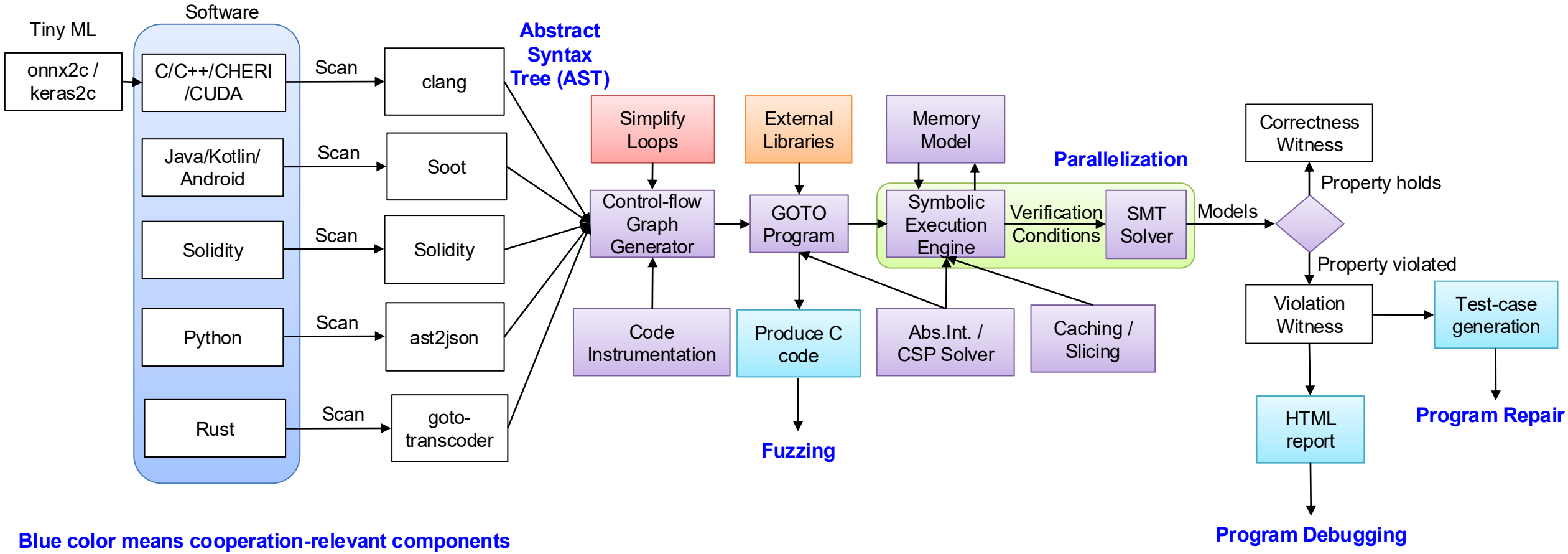
1) Introduce a **logic-based verification platform** to **find and fix software defects**

2) Explain how **testing, verification, and LLMs** can be combined to build **trustworthy software and AI systems**

3) Develop an **automated reasoning system** for **safeguarding software and AI systems** against vulnerabilities

Why ESBMC as the cooperative platform?

ESBMC supports multiple front-ends, solvers, and integrations with fuzzing engines, LLMs, and SMT solvers

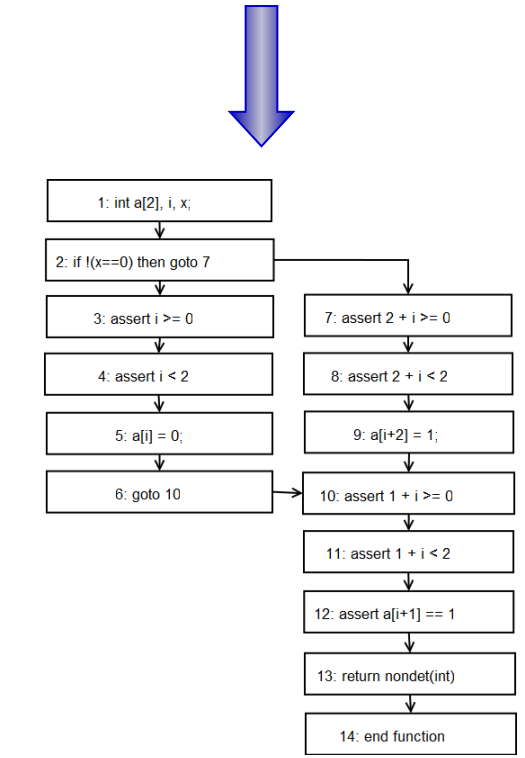


Blue color means cooperation-relevant components

Software BMC

- program modeled as a state transition system

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



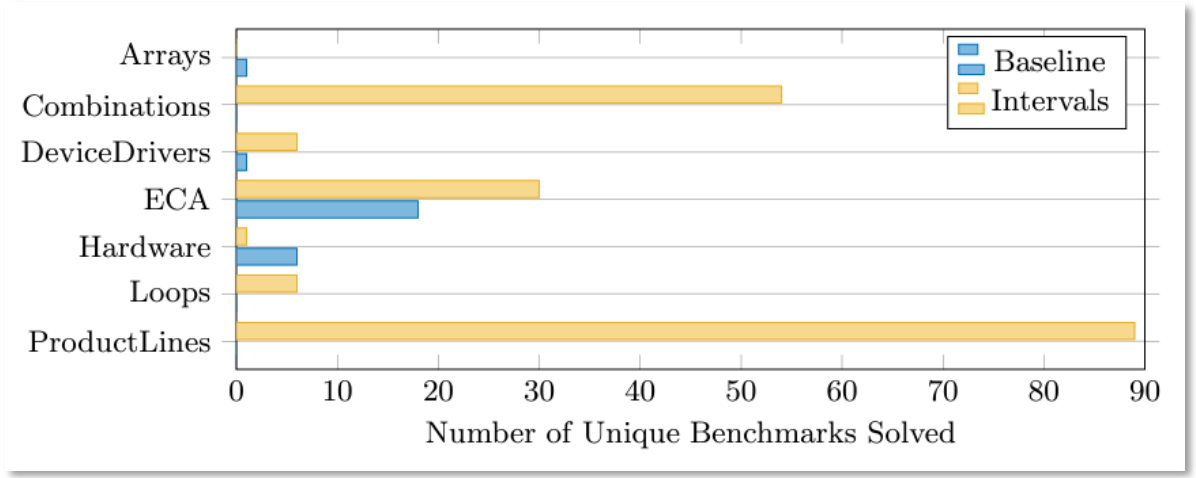
Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes

```
int main()
{
  int x;
  if (x >= 5 && x <= 10)
  {
    assert(x >= 0);
    assert(x <= 100);
  }
  return 0;
}
```

```
[PROGRESS] Starting Bounded Model Checking
Symex completed in: 0.002s (16 assignments)
Caching time: 0.000s (removed 0 assertions)
Slicing time: 0.001s (removed 11 assignments)
Generated 2 VCC(s), 0 remaining after simplification (5 assignments)
BMC program time: 0.003s

VERIFICATION SUCCESSFUL
```

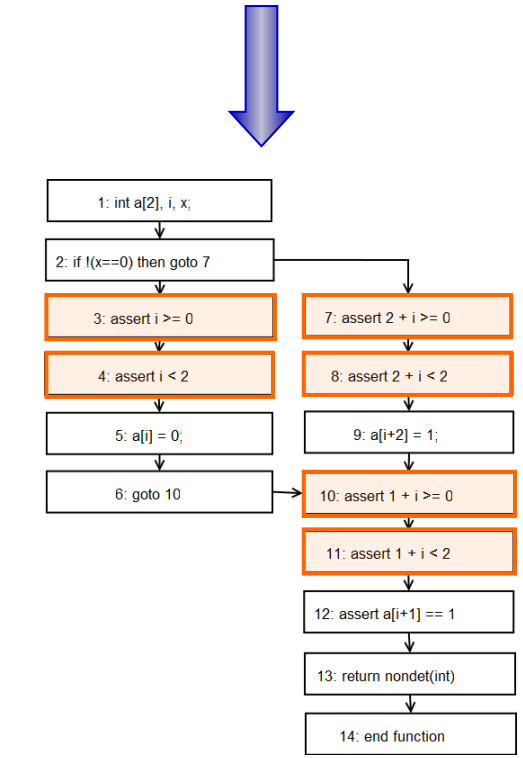


Menezes, R., Manino, E., Shmarov, F., Aldughaim, M., de Freitas, R., Lucas C. Cordeiro: Interval Analysis in Industrial-Scale BMC Software Verifiers: A Case Study.

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```

array bounds

conditional assertion



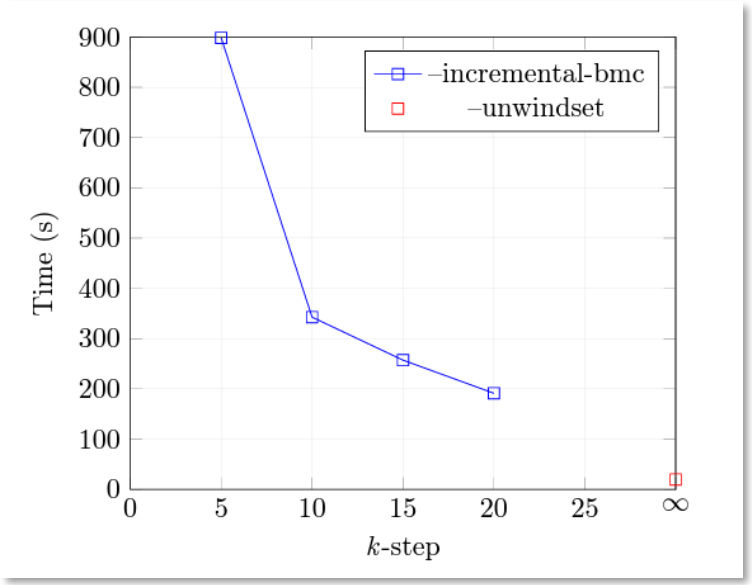
Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes
- program unfolded up to given bounds

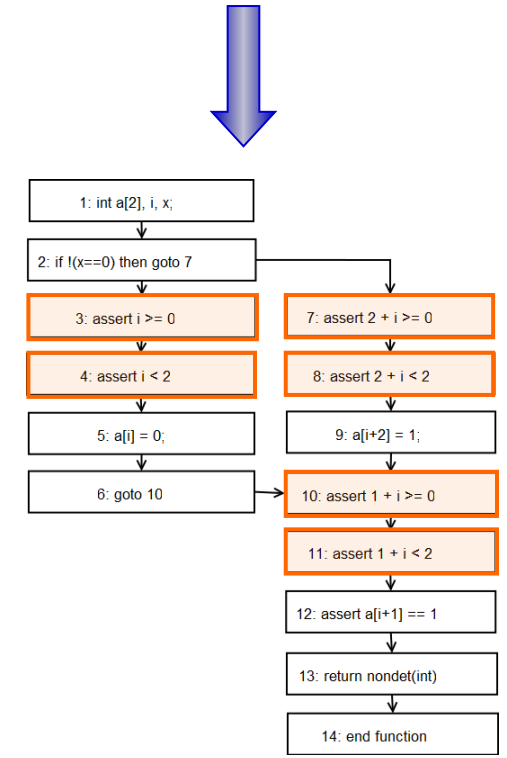
```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```

array bounds

conditional assertion



Wu T., Xiong, S., Manino, E., Stockwell, G., Cordeiro, L.: Verifying components of Arm(R) Confidential Computing Architecture with ESBMC. SAS 2024



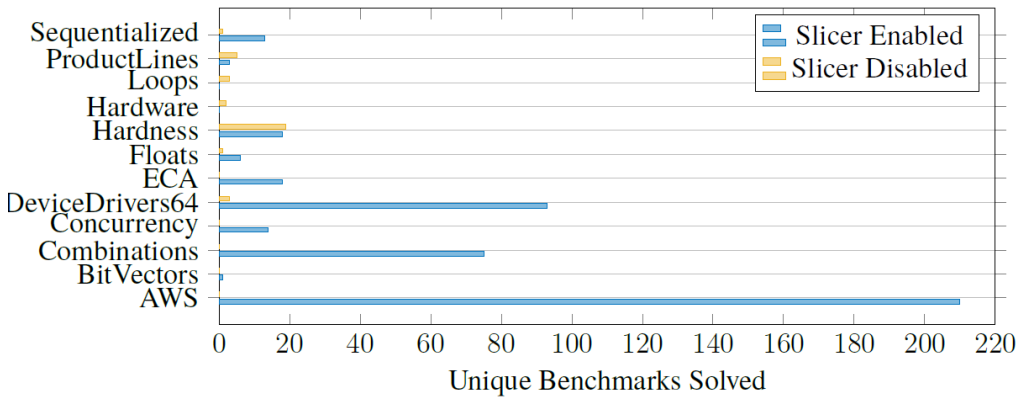
Software BMC

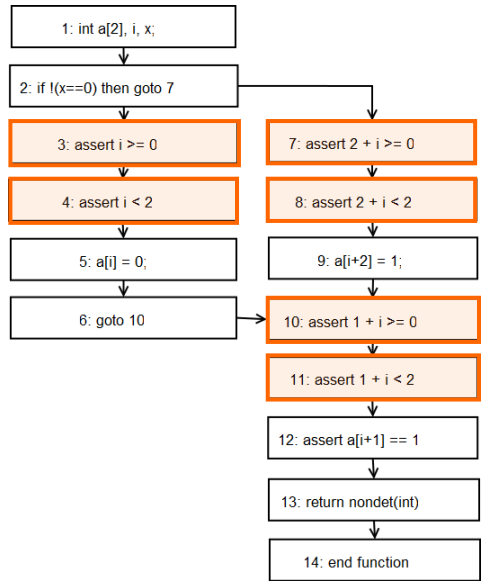
- program modeled as a state transition system
- added assumptions/safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```

array bounds

conditional assertion



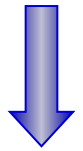


Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

  } crucial
- front-end converts unrolled and **optimized program into SSA**

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



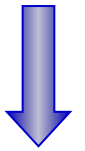
$g_1 = x_1 == 0$
$a_1 = a_0$ WITH $[i_0:=0]$
$a_2 = a_0$
$a_3 = a_2$ WITH $[2+i_0:=1]$
$a_4 = g_1 \ ? \ a_1 : a_3$
$t_1 = a_4 [1+i_0] == 1$

Software BMC

- program modeled as a state transition system
- added assumptions/safety properties as extra nodes
- program unfolded up to given bounds
- unfolded program optimized to reduce blow-up
  - constant propagation/slicing
  - forward substitutions/caching
  - unreachable code/pointer analysis

  } crucial
- front-end converts unrolled and **optimized program into SSA**
- extraction of *constraints C* and *properties P*
  - specific to selected SMT solver, uses theories
- satisfiability check of $C \wedge \neg P$

```
int main() {
  int a[2], i, x;
  if (x==0)
    a[i]=0;
  else
    a[i+2]=1;
  assert(a[i+1]==1);
}
```



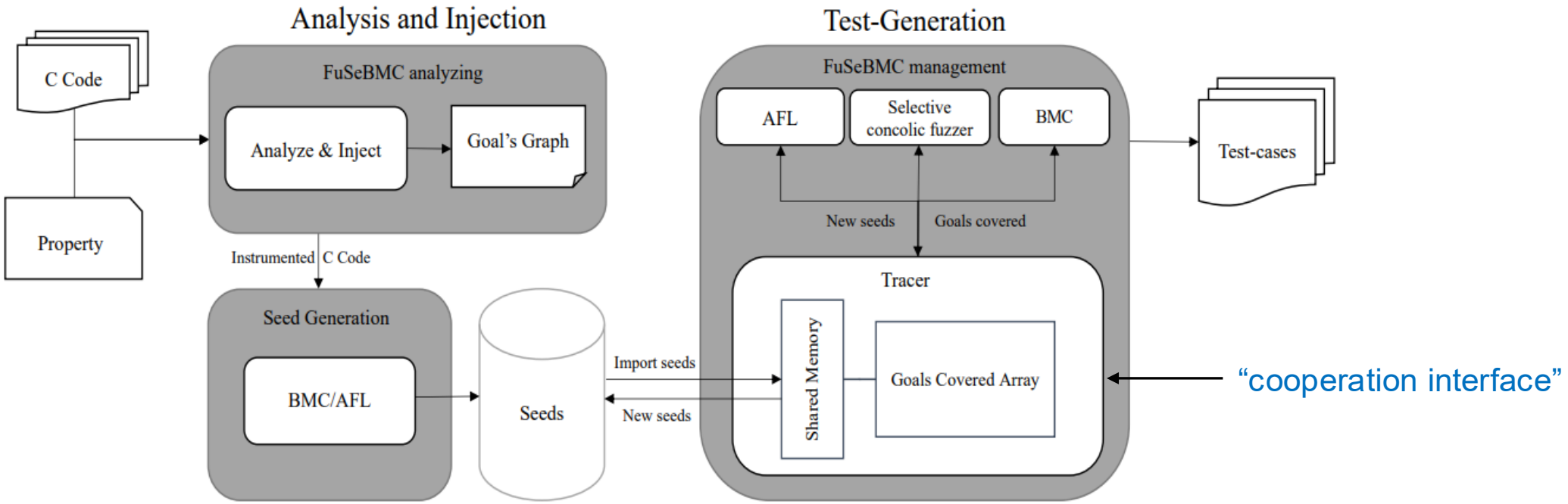
$$C := \begin{bmatrix} g_1 := (x_1 = 0) \\ \wedge\, a_1 := store(a_0, i_0, 0) \\ \wedge\, a_2 := a_0 \\ \wedge\, a_3 := store(a_2, 2 + i_0, 1) \\ \wedge\, a_4 := ite(g_1, a_1, a_3) \end{bmatrix}$$

$$P := \begin{bmatrix} i_0 \geq 0 \wedge i_0 < 2 \\ \wedge\, 2 + i_0 \geq 0 \wedge 2 + i_0 < 2 \\ \wedge\, 1 + i_0 \geq 0 \wedge 1 + i_0 < 2 \\ \wedge\, select(a_4, i_0 + 1) = 1 \end{bmatrix}$$

Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-Based Bounded Model Checking for Embedded ANSI-C Software. IEEE Trans. Software Eng. 38(4): 957-974 (2012)

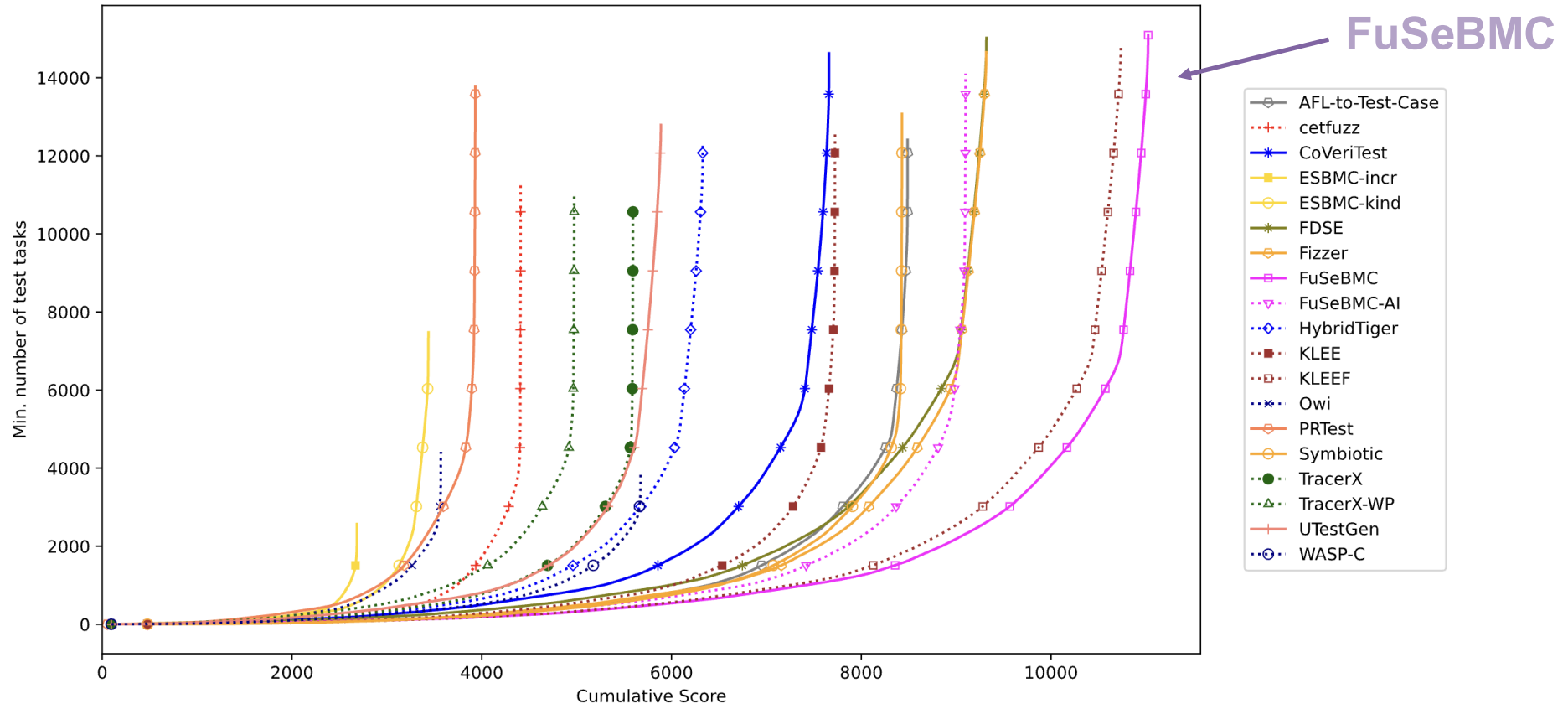
FuSeBMC is a Cooperative Verifier

- **BMC** handles **reachability**, **AFL** explores **coverage**, and the **concolic fuzzer** generates **path-sensitive seeds**
- The tracer is the **cooperation mechanism**, the interface through which engines **share information**



Alshmrany, K., Aldughaim, M., Bhayat, A., Cordeiro, L.: FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing - (Competition Contribution). FASE 2022: 336-340

Cooperation type: black-box portfolio with shared coordinator



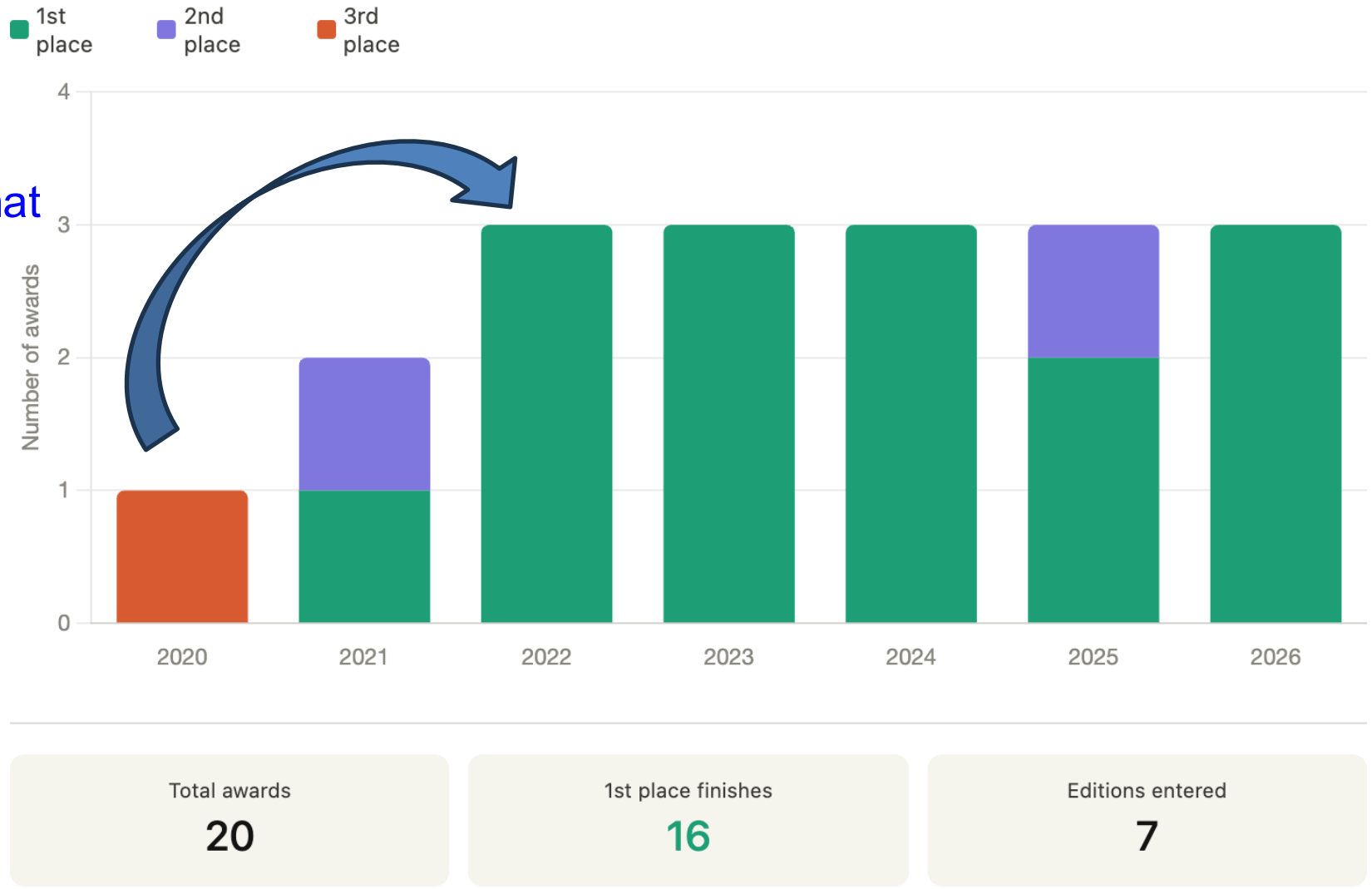
FuSeBMC achieved 3 awards: 1st place in `Cover-Error`, 1st place in `Cover-Branches`, and 1st place in `Overall`

Alshmrany, K., Aldughaim, M., Bhayat, A., Cordeiro, L.: FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing - (Competition Contribution). FASE 2022: 336-340

https://test-comp.sosy-lab.org/2026/

This is a portfolio-style cooperation with a coordinator, where engines run and share coverage information

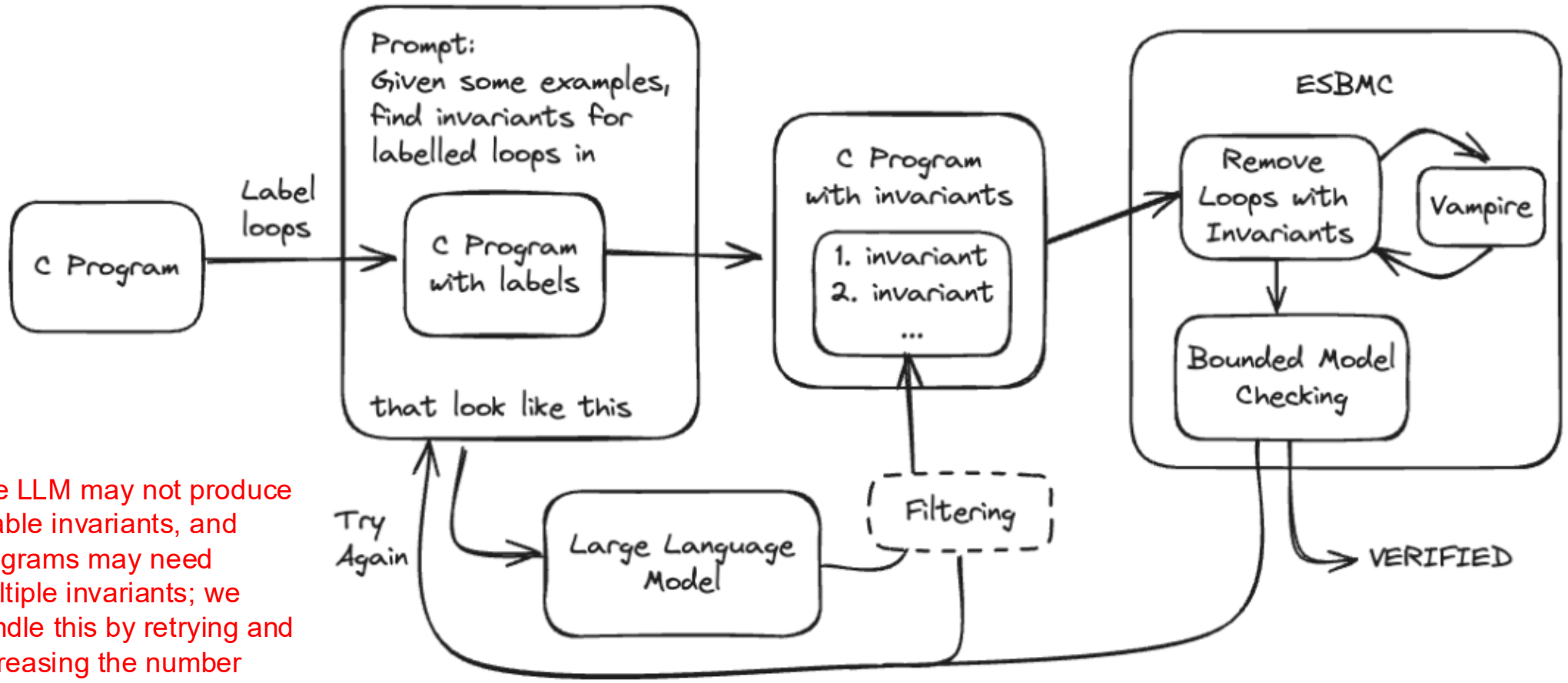
iterative improvement that reinforces the cooperative verification



Research Question

Can we use **LLMs** to **speed up program verification**, either by **proving correctness** or by **finding more software vulnerabilities** than state-of-the-art methods?

The ESBMC ibmc Tool



The LLM may not produce usable invariants, and programs may need multiple invariants; we handle this by retrying and increasing the number requested every n attempts

Is this Program Correct?

```
def main() -> None:
    x: int = 0
    y: int = 50
    while x < 100:
        x = x + 1
        if x > 50:
            y = y + 1
    assert y == 100
```

```
def main() -> None:
  x: int = 0
  y: int = 50
  x = x + 1
  if x > 50:
      y = y + 1
  ... // 99 other loop unrollings
  assert y == 100
```

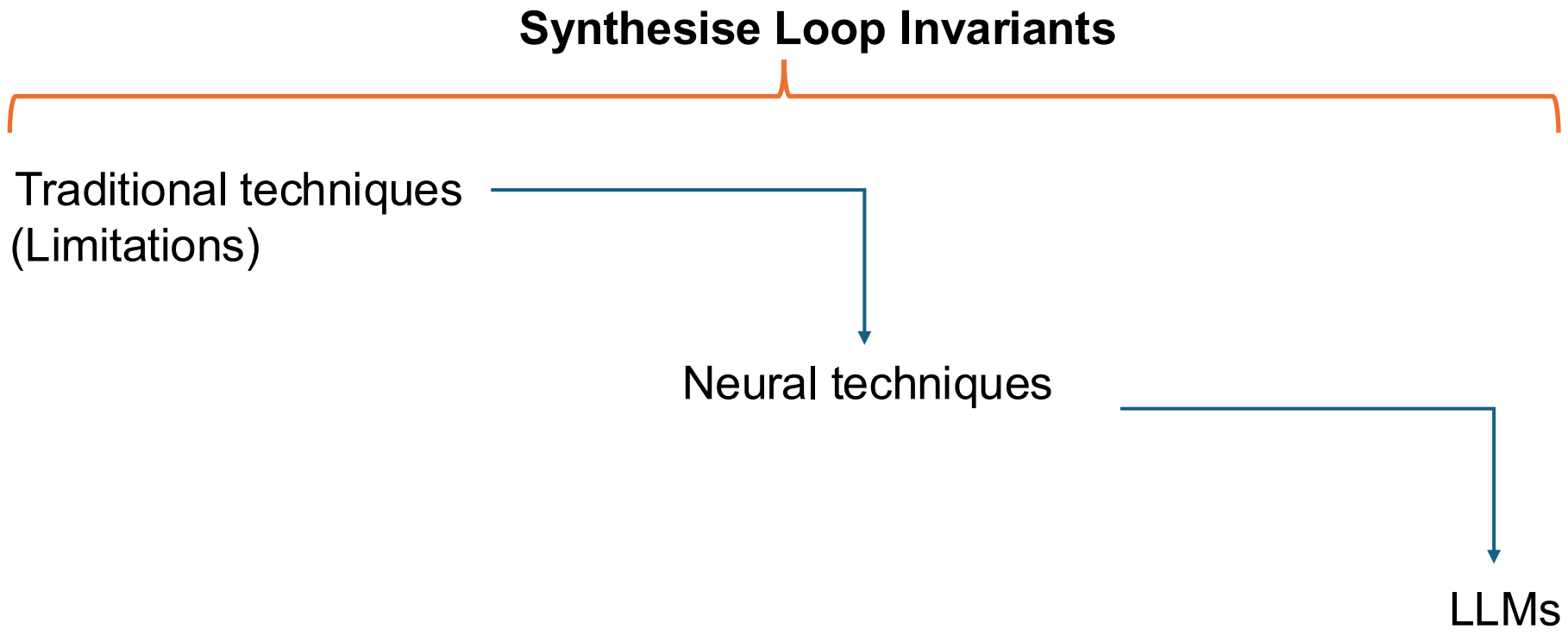
BMC can detect bugs, but requires assistance in proving correctness

100 loop iterations = 100× larger formula

BMC struggles with loops that can't be statically bound or that have a large bound

Loop Invariants

- **Inductive loop invariant:** A logical assertion over program variables that (1) holds before the loop is entered, and (2) is preserved by every execution of the loop body
- **Base Case:** The invariant holds before the first iteration of the loop
- **Step Case:** The invariant holds for every iteration of the loop



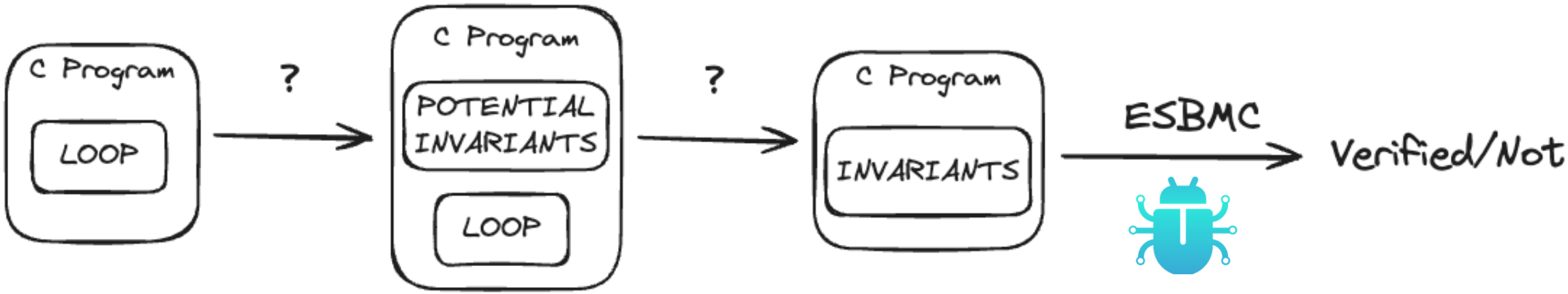
Replace Loop with Loop Invariants

```
while x < 100:
    x += 1
    if x > 50:
        y += 1
```

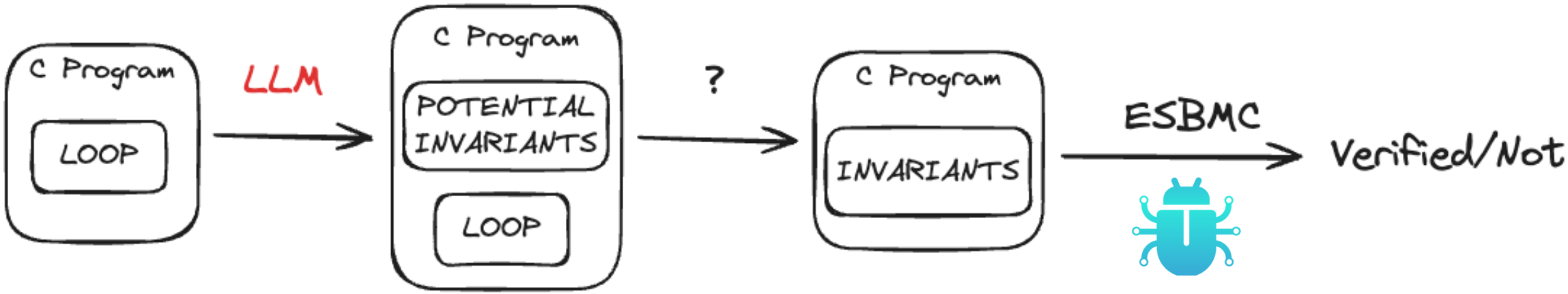
```
def main():
    x: int = 0
    y: int = 50
    # The loop keeps x between 0 and 100
    __ESBMC_assume(0 <= x and x <= 100)
    # If x is 50 or less then y is 50
    __ESBMC_assume(x <= 50 ==> y == 50)
    # If x is greater than 50 then y = x
    __ESBMC_assume(x > 50 ==> y == x)
    # At the end of the loop x is not < 100
    __ESBMC_assume(x >= 100)

    assert(y == 100)
```

Replace Loop with Loop Invariants

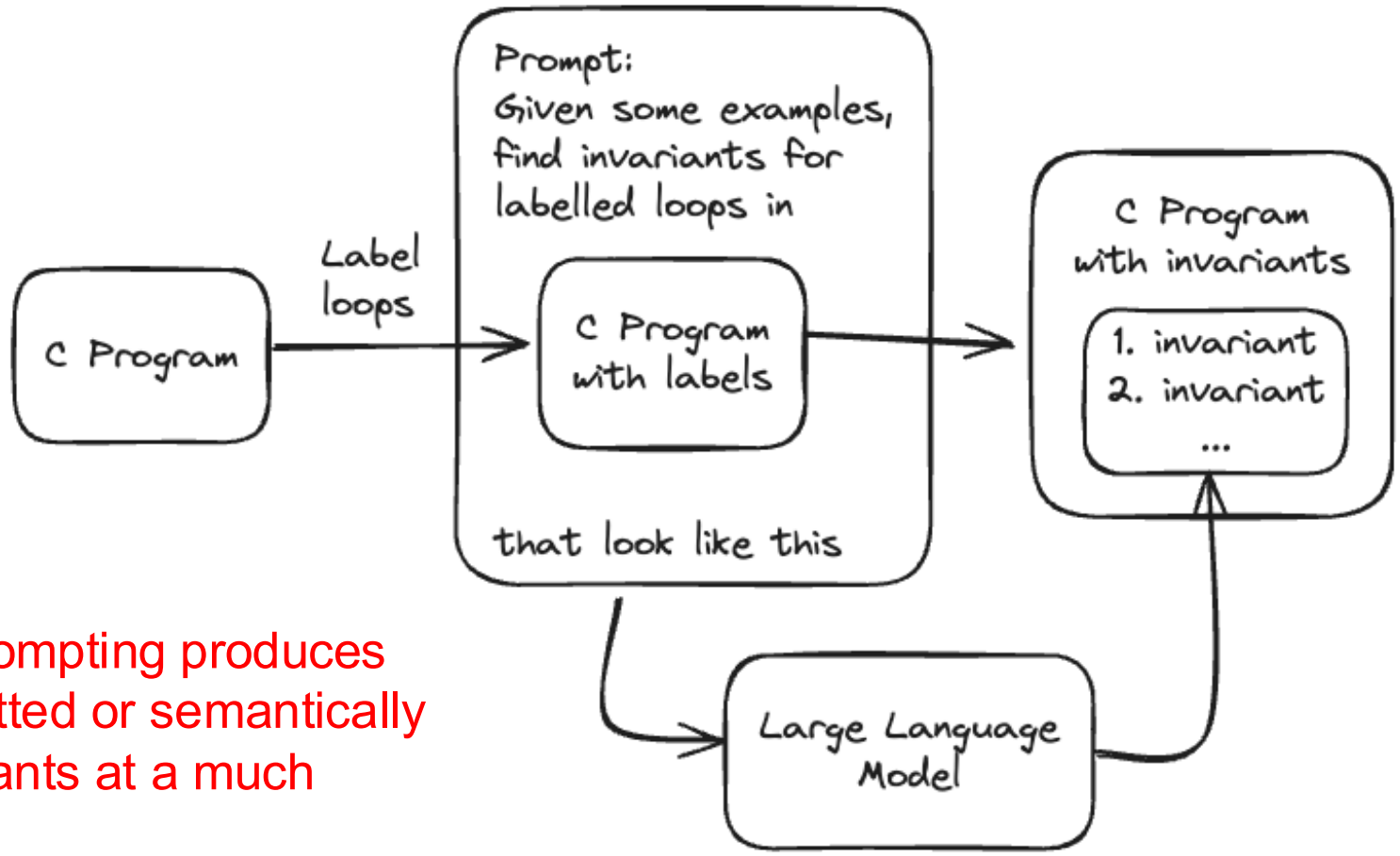


Generating Potential Invariants



Generate C expressions of the form *__invariant(...)* that are likely to be inductive for the given loop

Generating Potential Invariants



Zero-shot prompting produces poorly formatted or semantically wrong invariants at a much higher rate

LLMs Can Get it Wrong!

```
__invariant(c > 1 && c < 2 ==> i == 0);
__invariant(...);
__invariant(
This is the invariant
__invariant(...); for
the c program.
```

```
int x;
int y;
__invariant(x > max);
```

How to get better answers? (Language Server Protocol)

1. Constrain the prompt (no explanation as it can be confusing)
2. Filter the answers (simple regex filtering for now)

In non-filtered experiments, 67–75% of iterations are wasted due to syntactically incorrect invariants

Prompt Engineering for Invariant Generation

```
<Examples with explanation>

Based on these examples provided above can you
generate  a C invariant for the following code,

<Labelled Input C Program>

Print an invariant for this loop that holds in the
form '__invariant(...);'. They should help prove the
assertions. You can utilise '&&' or '||' if required.
No explanation.  Your answer should be in the form
'__invariant(. . .);'
```

Chain of Thought (CoT) Prompt Engineering

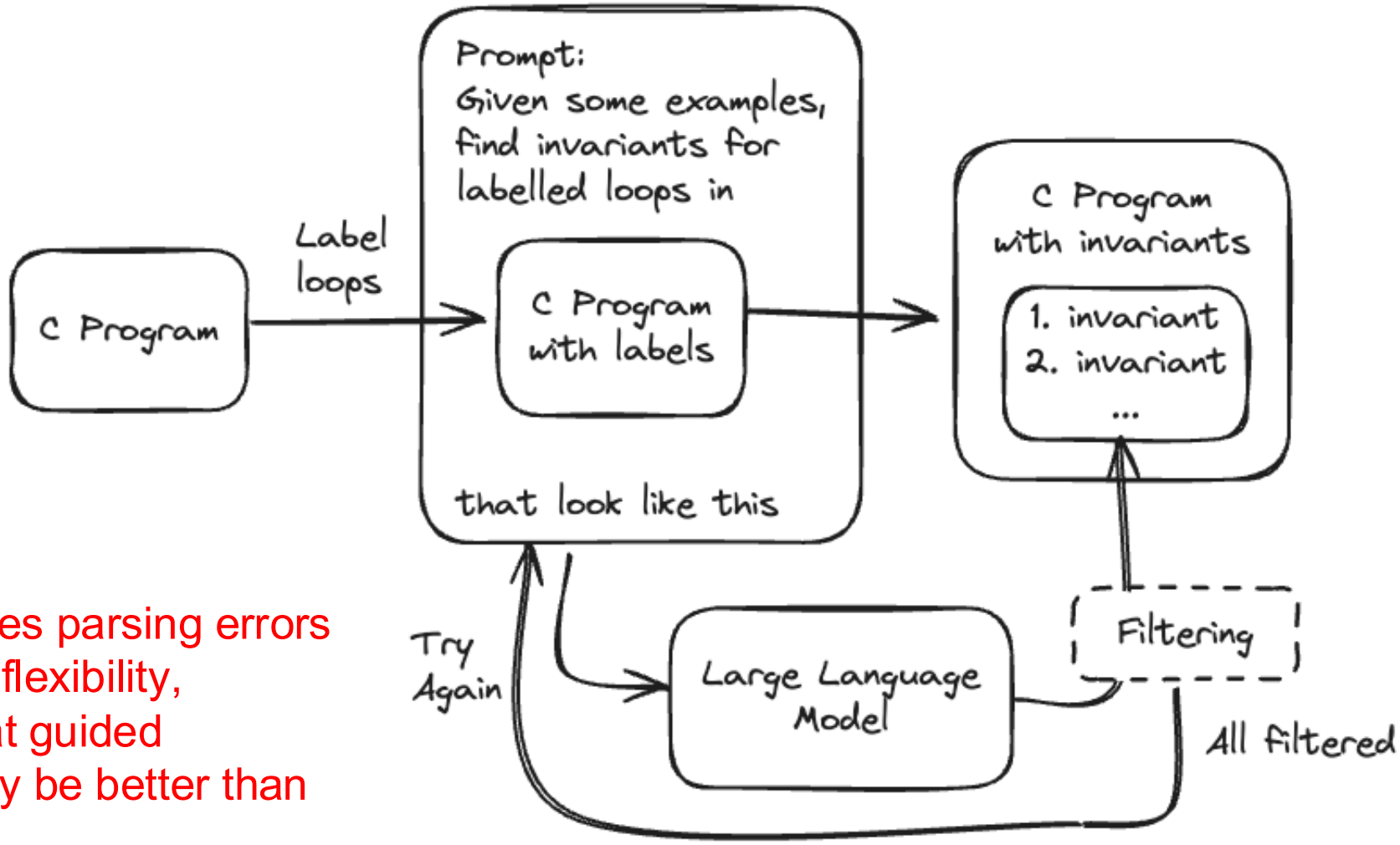
Small number of different prompts that vary by their examples e.g. :

- Single Invariant
- Multiple Invariants

Full prompts (more explanation), constrained prompts (less explanation), and combined prompts

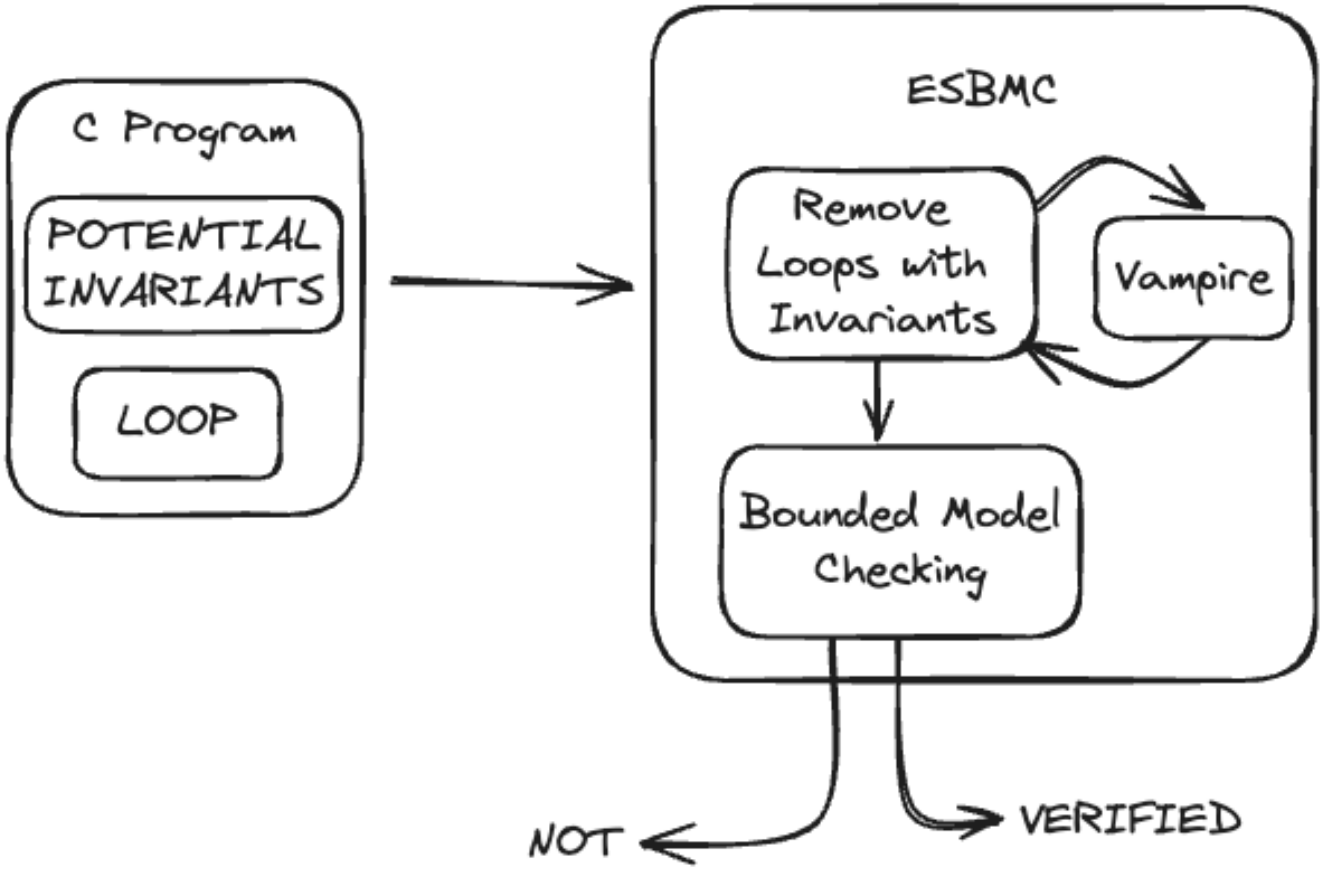
More explanation helps, but can distract, so a balance is best; further tuning may improve results

Generating Potential Invariants



Filtering reduces parsing errors but limits LLM flexibility, suggesting that guided generation may be better than post-filtering

Proving Potential Invariants



cooperation between two different formal methods engines

Sound but not Complete

The approach is Sound:

If the method proves an assertion, then it is correct (for partial correctness); it never proves false assertions

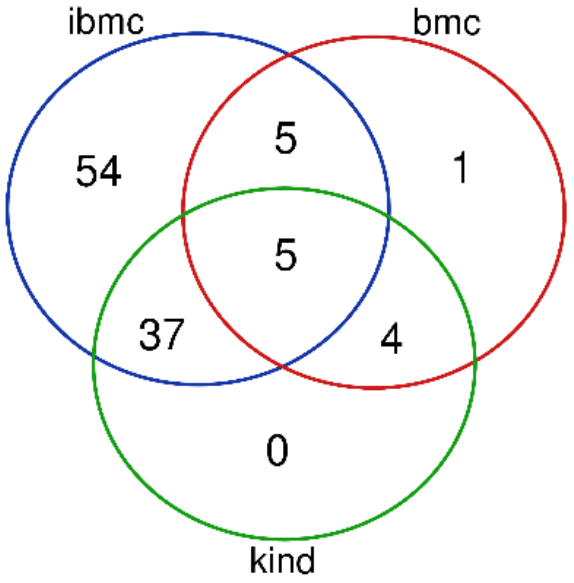
The approach is not Complete:

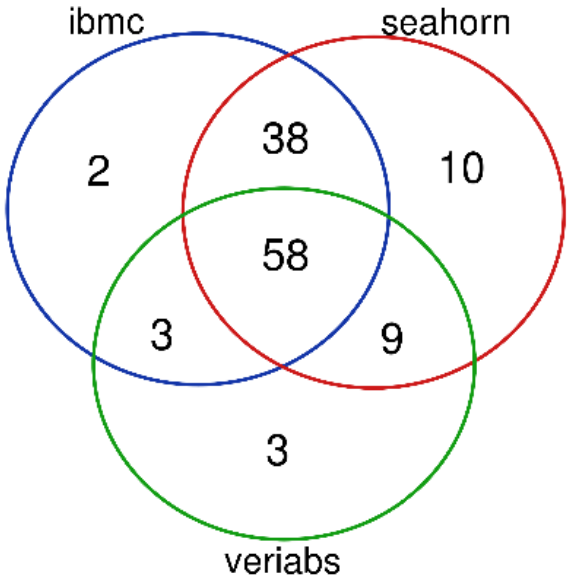
The method may fail to prove correct assertions, e.g., when the LLM does not generate sufficient invariants to capture the loop semantics (such as invariants requiring polynomial reasoning)

Comparing to Other Verifiers

- VeriAbs combines 12 techniques with slicing and strategy selection
- SeaHorn uses abstract interpretation to compute invariants and Horn clauses to verify programs

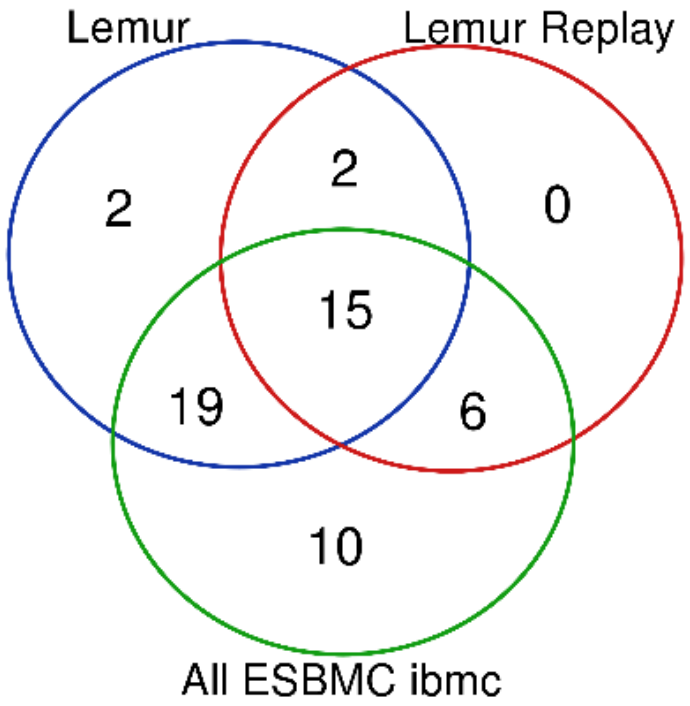
Tool	Solved	Unique
ESBMC bmc	16	0
ESBMC k-induction	46	0
SeaHorn	115	10
VeriAbs	73	2
ESBMC ibmc	101	2





ibmc solves 101 problems, outperforming ESBMC bmc (16) and k-induction (46), and uniquely solves 2 problems

Comparison to LEMUR



LEMUR is similar as it also uses an LLM to suggest invariants.

We could not run LEMUR so replayed their invariants using our extended ESBMC

Using their invariants we verified 6 more programs

Using our invariants we verified 10 more programs

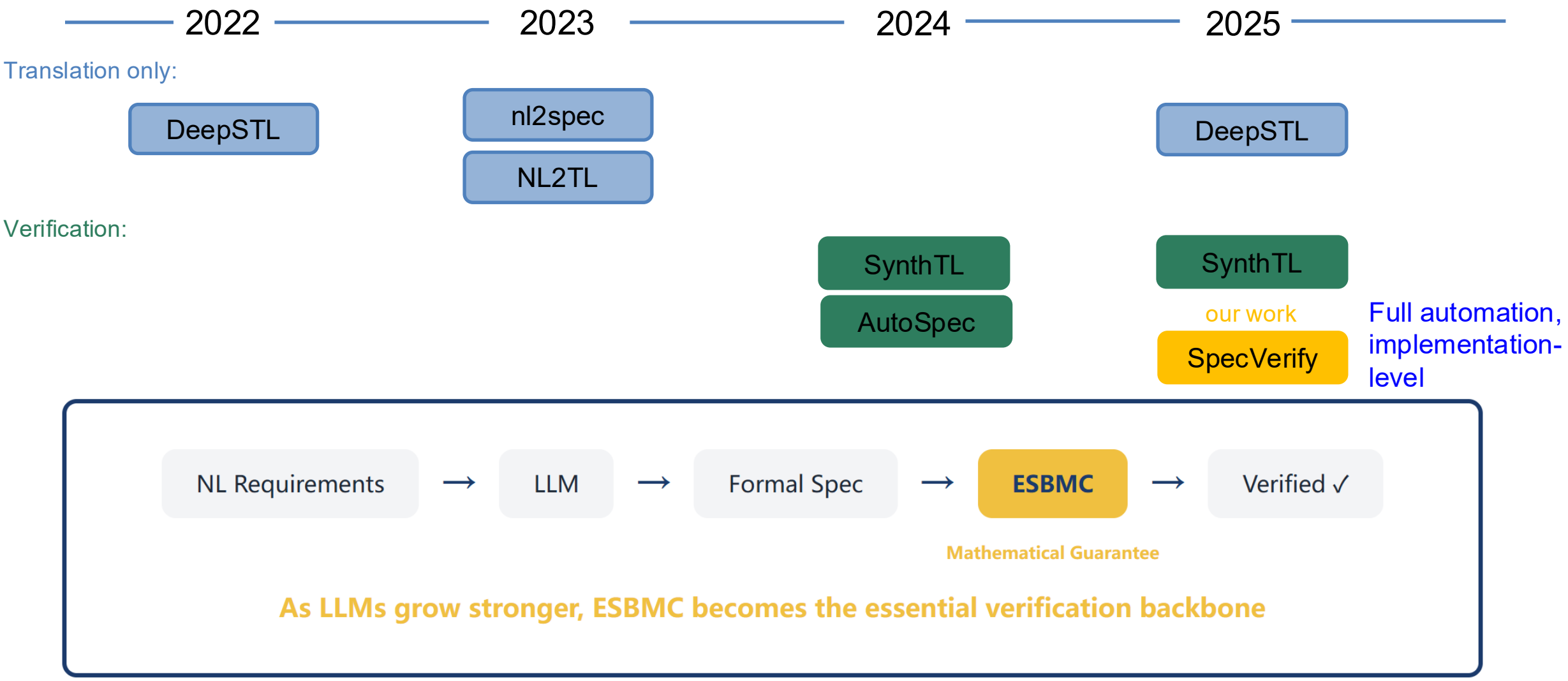
Potential lessons from both sides

Portfolio-style cooperation between ibmc and LEMUR!

Research Question

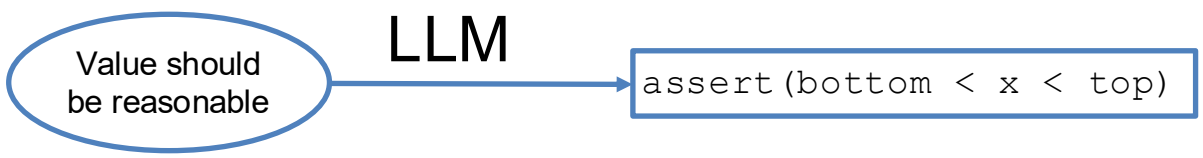
Can we use **LLMs** to **identify** and **formalize** a **system's properties** from its **requirements specification**?

LLM Meets Formal Verification: A Trend



Selected works on LLM-assisted requirement formalization and verification (2022–2025). DeepSTL [He et al., ICSE'22] · nl2spec [Cosler et al., CAV'23] · NL2TL [Chen et al., EMNLP'23] · SynthTL [Mendoza et al., FMCAD'24] · AutoSpec [Wen et al., CAV'24] · Aerospace IP [Ma et al., 2024] · Req2LTL [Ma et al., 2025] · SpecVerify [Wang et al., RE'25]

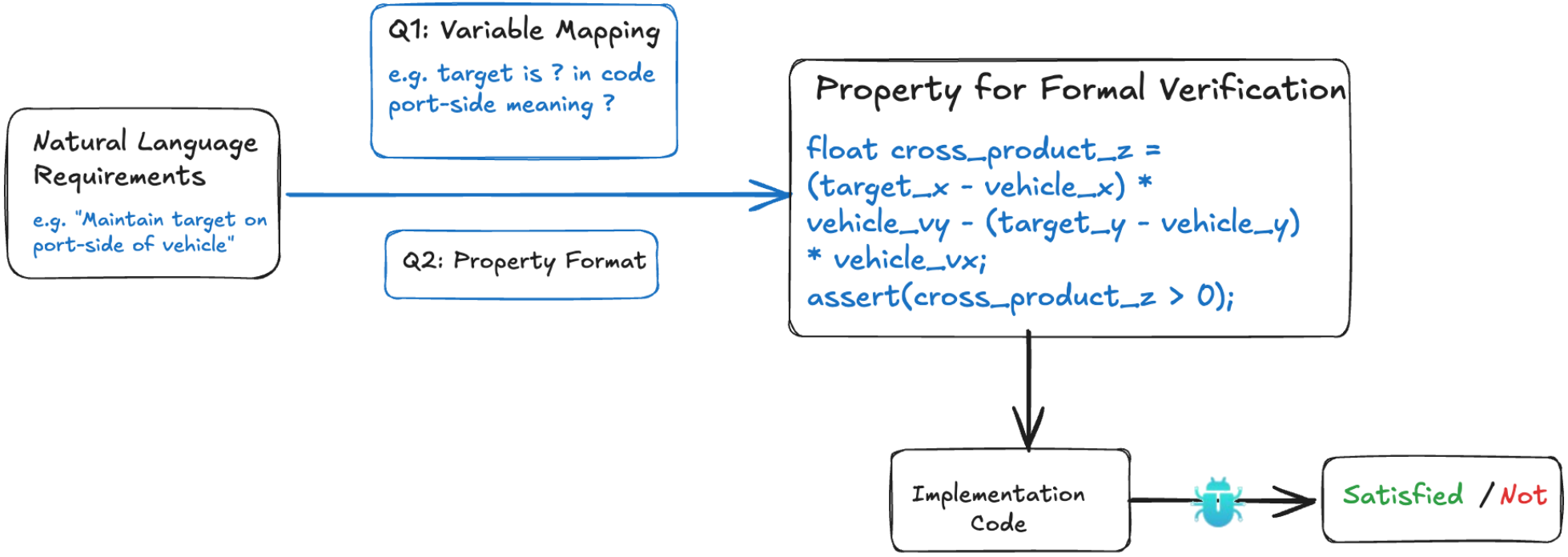
Challenges



- But what is reasonable?
- LLM cannot understand the vague message. Without a precise instrument, the result is “GIGO”
- Good point is that despite LLM being overconfident, BMC is strict: a wrong description at least will trigger a false alarm, and users will notice

- Alignment Problem: Which limit?
- It prevents the fully automated requirement-to-formal verification process
- LLM might pick one silently; however, BMC will force you to decide those limits

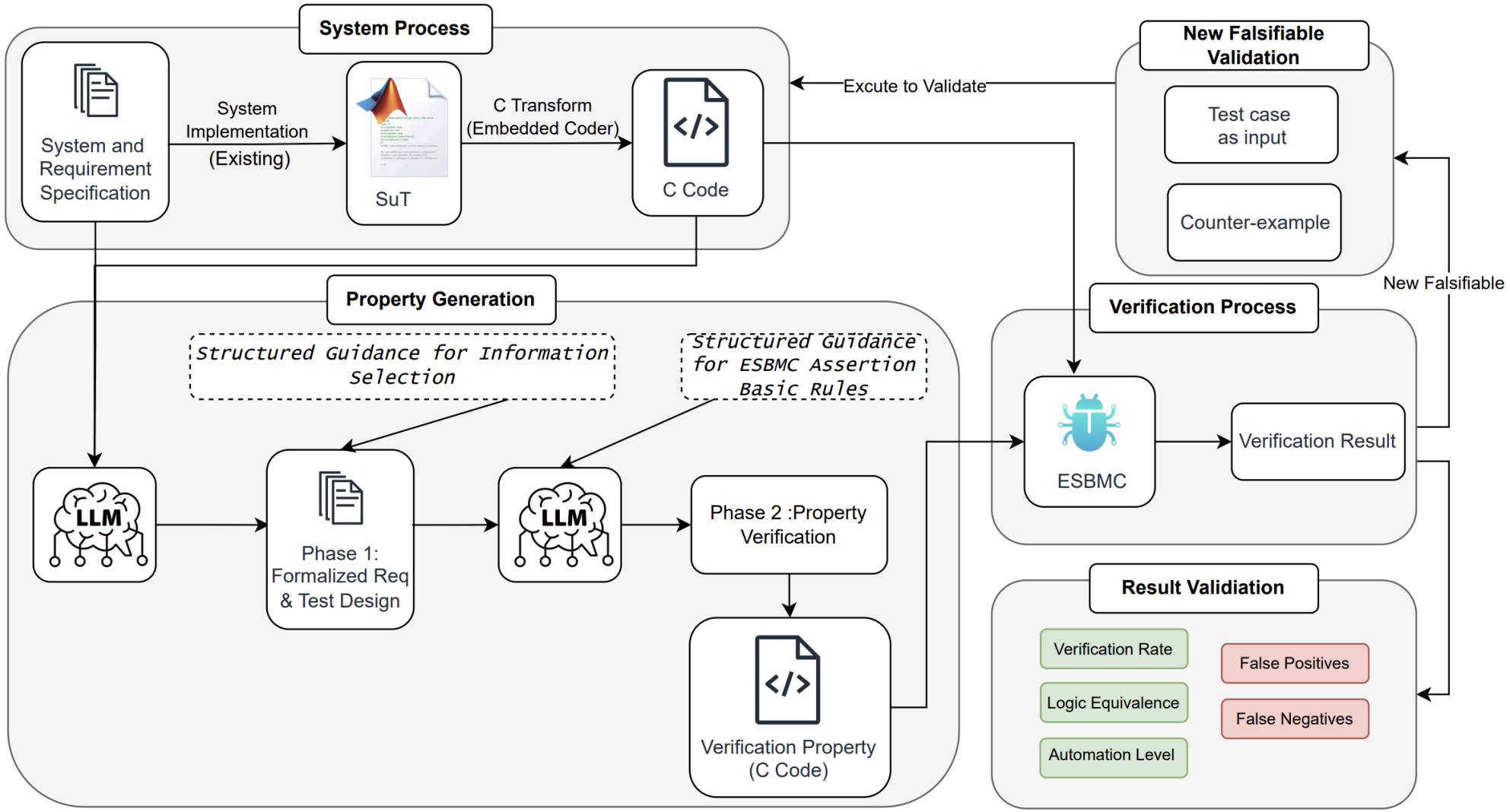
Making verification as easy and fast as AI-assisted development



BMC can verify functional correctness, but requires an expert to write the property

Variable and logic mapping are time-consuming for property generation

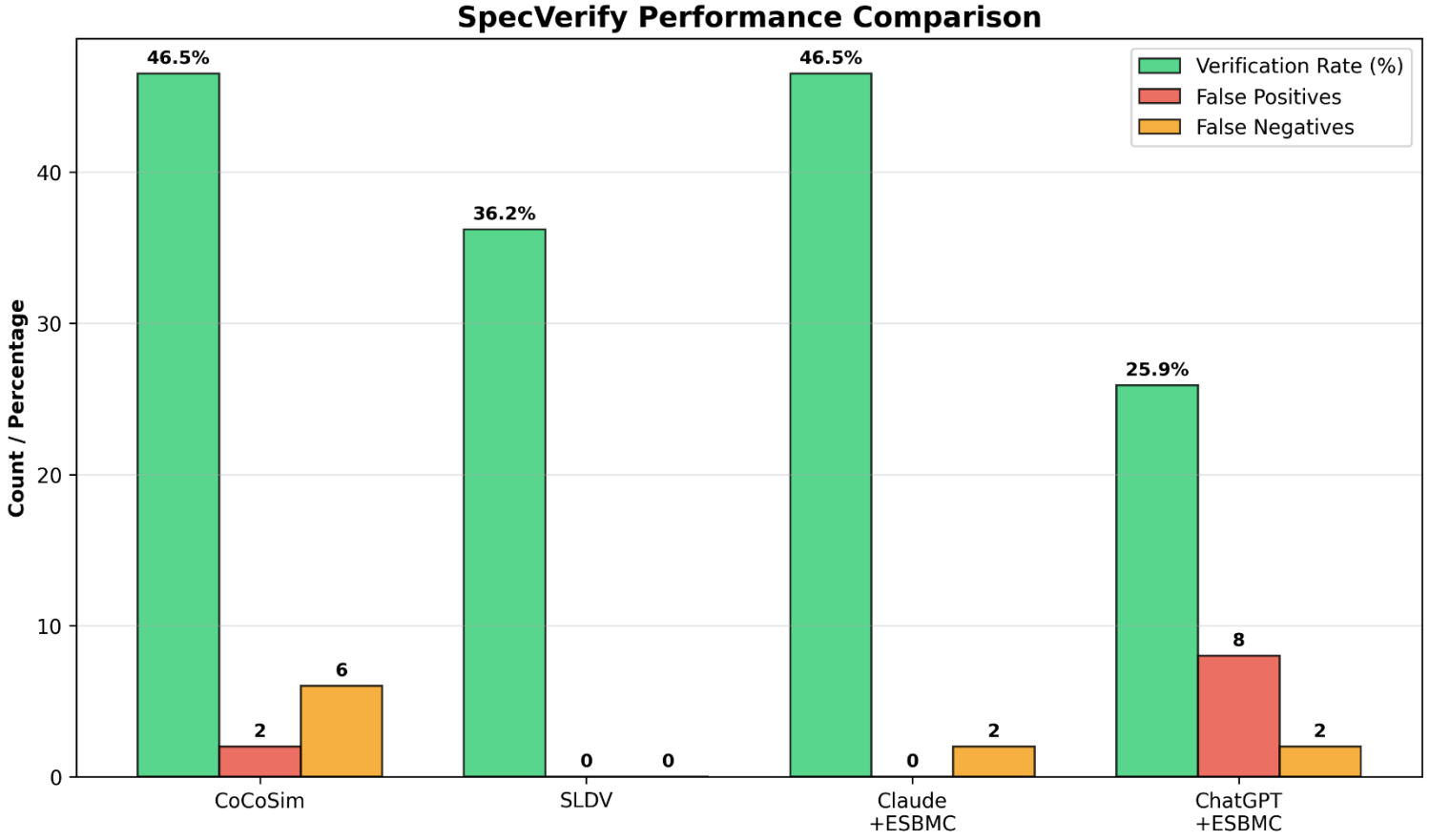
SpecVerify: LLM-Aided Requirements



Wang, W., Farrell, M., Cordeiro, L. C., & Zhao, L. (2025, September). *Supporting Software Formal Verification with Large Language Models: An Experimental Study*. In 2025 IEEE 33rd International Requirements Engineering Conference (RE) (pp. 423-431).

Weiqi Wang
University of Manchester
PhD student in S3 group

Comparing with Human-Expert Verification Result



Key Achievements:

- 79.31% logical equivalence with expert-verified properties
- **28% better** than SLDV
- 46.5% verification rate (matches CoCoSim baseline): 300s (TO)
- Superior accuracy: 2 false negatives, 0 false positives

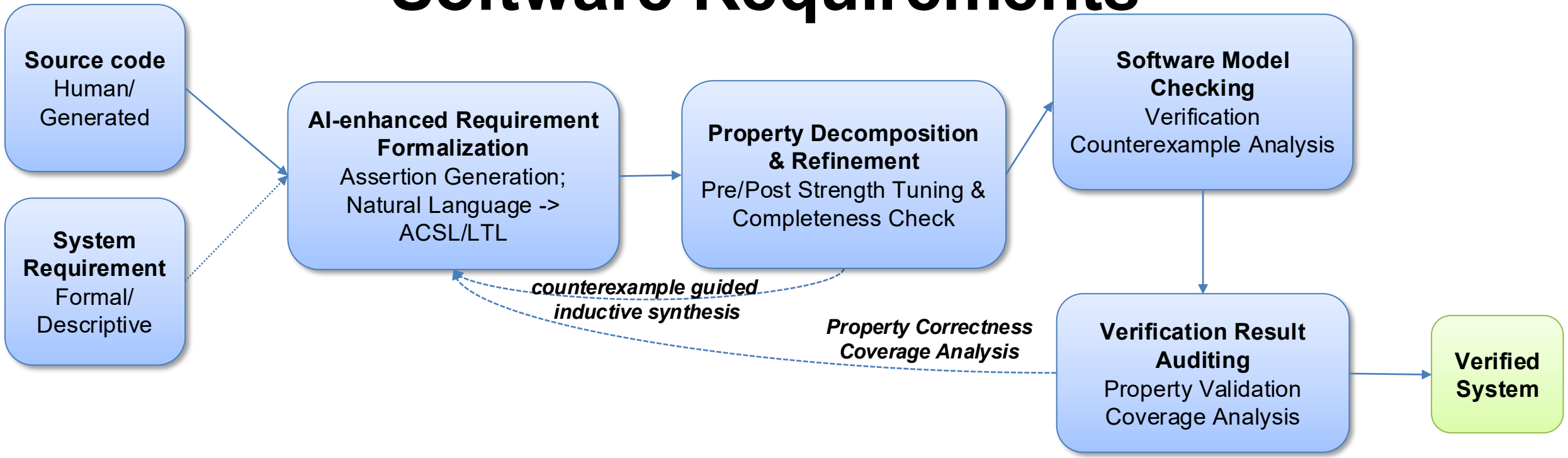
Key Advantage:

- Full automation: no manual variable mapping
- Found 2 new floating-point errors
- Direct implementation-level verification

Cyber-physical systems (Lockheed Martin)

Wang, W., Farrell, M., Cordeiro, L. C., & Zhao, L. *Supporting Software Formal Verification with Large Language Models: An Experimental Study*. In RE 2025, pp. 423-431.

Roadmap: Automated Formal Verification of Software Requirements



1 Formalization — Completed

Automated requirements formalization (SpecVerify)

2 Decomposition — In Progress

System-level property decomposition with refinement

3 Auditing — Planned

Verification result auditing and quality assurance

Cooperative AI-Assisted Formal Verification: Key Takeaways

LLMs are fast but unsound. Formal verifiers are sound but slow. Cooperation gives you both if the interface is designed carefully

System	Cooperation Type	Interface	Result
FuSeBMC	Black-box portfolio with shared coordinator	Tracer + shared memory	**20 awards · 16× 1st place**
ibmc	Sequential LLM-oracle + formal proof	Prompt → invariant → ESBMC / Vampire	**101 verified · sound no false positives**
SpecVerify	Iterative feedback loop with human-in-the-loop	NL → LLM → assertion → ESBMC → counterexample	**79% logical equiv. 0 false positives**

Acknowledgements