GenAIDE · INVITED TALK

The University of Manchester

Verifying What Generative AI Builds: Formal Methods for the AI Partner in Engineering

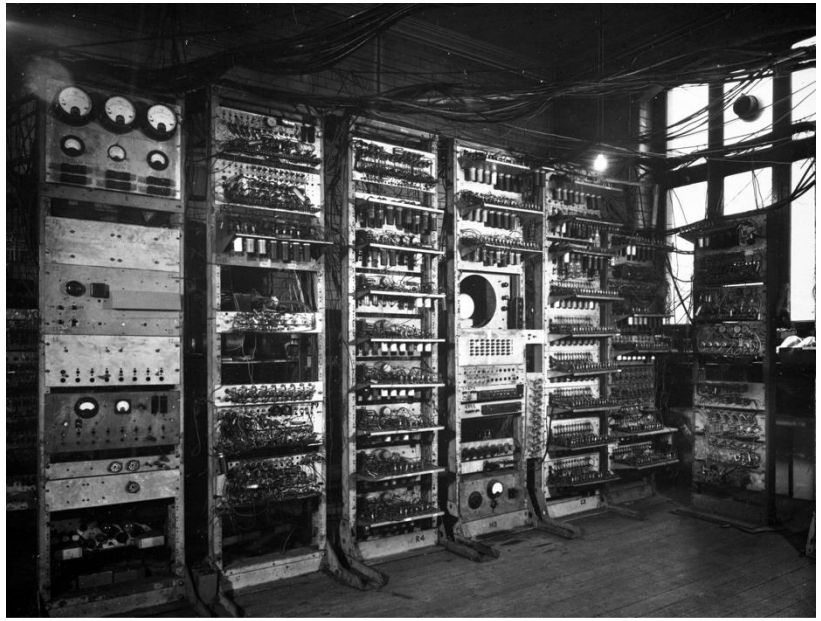
Prof. Lucas C. Cordeiro

Professor of Computer Science · Area Lead, Systems & Software Security (S3)
Director, Arm Centre of Excellence · Department of Computer Science
lucas.cordeiro@manchester.ac.uk

GenAIDE · Generative AI for Industrial Design Engineering · MSCA Industrial Doctoral Network, Horizon Europe Grant Agreement No. 101226927 · The University of Manchester is a consortium beneficiary

Manchester, 1948: the first stored program ran here

The first electronic stored-program computer, and the man who asked if programs are right.



The Manchester Baby, 21 June 1948

First electronic stored-program computer

Alan Turing

At Manchester from September 1948 until his death in 1954

Why do I mention the baby computer?

A stored program is data: it can be written, changed and generated by another program.

So the question of whether a program is right arrived with the first one.

A year later, Turing asked it in print.

> Once programs became data, someone had to ask whether they were right.

1949: Turing asks whether a program is right

The University of Manchester

> “How can one check a routine in the sense of making sure that it is right?”
>
> *A. M. Turing, “Checking a Large Routine”, Cambridge, June 1949*

1949 · Turing

Answers it: assertions at points in the routine, and a separate argument that it stops.

1967–69 · Floyd, Hoare

Make the same idea a method: assertions and invariants become a logic.

Now · generative AI

Code is written faster than anyone can read it, so the question returns with no human in the loop.

Almost eighty years on, Manchester is still working on the question Turing asked.

Four Turing Awards built the ground this talk stands on

Automated program reasoning: a sixty-year research programme.

1978

Robert W. Floyd

Founding automatic program verification as a subfield

How do you even state that a program is correct?

1980

C. A. R. Hoare

Fundamental contributions to the definition and design of programming languages

Hoare logic: preconditions, postconditions, invariants.

1996

Amir Pnueli

Introducing temporal logic into computing science; program and system verification

How do you state a property that must hold over time?

2007

Clarke, Emerson and Sifakis

Developing model checking into a highly effective verification technology

How do you check it automatically? ESBMC is a direct descendant.

Every award was for making correctness machine-checkable; that is still open, and generative AI makes it urgent.

A good field to join: the foundations are settled, the automation is not.

The program: ten lines of code, wired to a battery

A single-cell charger: the software reads two registers and writes one bit.

THE PLANT

USB cable
in or out, at any tick

battery cell
gains one unit while the charger is enabled

THE REGISTERS

`status  VBUS`
is a supply plugged in?

`adc`
what the cell holds, 0..CAPACITY

`ctrl  CHG_EN`
let current flow into the cell

THE SOFTWARE

charger_tick()
runs on the timer interrupt, once a second

ten lines of C
the only part we can change

What the harness leaves open

The cable at every tick, the ADC reading, how long the device runs.

What stays concrete

The cell: enabled means one unit more, and never more than CAPACITY.

> The code is ten lines. The hardware it is wired to is the other half of the specification.

turing-awards-esbmc/charger.h and charger.c; the harness that plays the hardware is 1-floyd.c.

One program, four ways to say what "correct" means

The same battery charger, specified four ways, one per award, each checked by ESBMC.

1978 **Floyd**	State it at a point. `__ESBMC_assert(battery <= CAPACITY, "never overcharged");`	**Proved for every run**

1980
Hoare
	State it compositionally, and inductively.
	`__ESBMC_requires / __ESBMC_ensures`
	`__ESBMC_loop_invariant(battery <= CAPACITY);`
		Proved where BMC gives up

1996 **Pnueli**	State what must hold over time. `G( pressed → F (charge > min) )`	**No violating trace within the bound**
2007 **Clarke, Emerson & Sifakis**	Write nothing new; the machine decides. *every verdict above, reached automatically*	**A proof, or a counterexample**

Three languages for saying what correct means; one machine that checks all three.

ROADMAP

Six questions, in the order you asked them

1. **Who we are**
   Systems & Software Security at Manchester
2. **What we build**
   ESBMC and cooperative verification
3. **Why it matters here**
   Generated code is fast, not correct
4. **How we do it**
   Three cooperation patterns
5. **Open challenges**
   What we think GenAIDE should attack
6. **Our doctoral candidates**
   Nine candidates; two are GenAIDE's own

Then, where the overlap with GenAIDE's work packages is concrete enough to act on.

> Our claim: the AI Partner in Engineering needs a proof partner alongside it.

A national cyber-security centre

"We develop state-of-the-art algorithms, methods and protocols to address security and privacy in networked and distributed system environments, and tools to build verifiable, trustworthy software and AI/ML systems."

One of 21 universities recognised by the NCSC and EPSRC as an Academic Centre of Excellence in Cyber Security Research, announced 21 March 2024.

NCSC = UK National Cyber Security Centre.

13 academics · 6 research areas

Lucas Cordeiro (Area Lead)	Richard Banach	Alex Creswell
Daniel Dresner	Boris Fouotsa	Sebastian Köhler
Bernardo Magri	**Edoardo Manino**	Mustafa Mustafa
Gaven Smith	**Zhipeng Wang**	**Dominik Winterer**
Ning Zhang		

Cryptography, code assurance and AI security; one group, one corridor.

What Manchester brings to the AI Partner in Engineering

GenAIDE trains 14 Doctoral Candidates to *"evolve computer-aided engineering from being mere support tools to becoming proactive AI partners within hybrid engineering teams."*

CORDIS objective text, Grant Agreement 101226927

Automated reasoning at industrial scale

Proves the absence of runtime errors, or returns a concrete counterexample. Eight languages.

Generation and proof, made to cooperate

Three published systems: FuSeBMC, then ibmc and SpecVerify, where a model proposes and a proof engine disposes.

A doctoral programme already on this topic

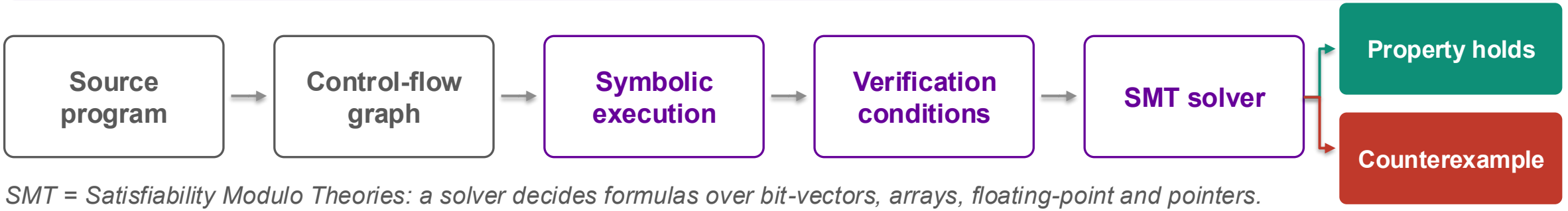
Nine doctoral candidates, two of them GenAIDE's own, plus four recent completions.

We are not a design-optimisation group; we are the assurance layer such a network needs.

> GenAIDE builds the AI Partner in Engineering. We have already built its proof partners.

ESBMC: one intermediate representation language

ESBMC is an open-source, context-bounded model checker that **detects or proves the absence of runtime errors** in single- and multi-threaded C, C++, CUDA, CHERI, Kotlin, Python, Rust and Solidity.



SMT = Satisfiability Modulo Theories: a solver decides formulas over bit-vectors, arrays, floating-point and pointers.

k-induction adds an inductive step, so a property can be proved for loops of any length.

Open source

Apache 2.0: no licence negotiation to trial it.

Two useful answers

A proof, or an input that makes the program fail.

Funded to continue

Arm, EPSRC, Ethereum Foundation, Rust Foundation, EU H2020 ELEGANT, UKRI Soteria.

> The output is evidence a machine can re-check, not a confidence score.

How it works, in one example


```
def main() -> None:
    a = [nondet_int(),
         nondet_int()]
    i: int = nondet_int()
    x: int = nondet_int()
    if x == 0:
        a[i] = 0
    else:
        a[i + 2] = 1
    assert a[i + 1] == 1
```

nondet_int() makes i, x and a symbolic: no test suite covers every combination.

```
g1 = x1 == 0
a1 = a0 WITH [i0 := 0]
a2 = a0
a3 = a2 WITH [2+i0 := 1]
a4 = g1 ? a1 : a3
t1 = a4[1+i0] == 1
```

All paths become one logical formula; none is missed.

Ask the SMT solver:

$$C \wedge \neg P$$

satisfiable → a bug, with the exact i, x and array contents that trigger it

unsatisfiable → no such execution exists within the bound

C = the program's constraints. P = the property you want.

Unfold loops to a bound	Slicing, constant propagation, caching	Convert to single-assignment form	Extract constraints C and property P	Check satisfiability of C ∧ ¬P

BMC is complete only up to the bound; k-induction and loop invariants make the proof unbounded.

k-induction: from bounded search to unbounded proof

Base case B(k)

BMC to depth k: finds real bugs

Forward condition F(k)

All loops unrolled within k: a proof

Inductive step I(k)

Holds for k steps, so for the next: a proof

Four live runs

Bug	**found, k = 11**
Bounded loop	**F(k) proves, k = 21**
Unbounded loop	**UNKNOWN**
+ one invariant	**I(k) proves, k = 2**

SV-COMP ReachSafety runs --k-induction.

Plain k-induction stalls on the unbounded loop; one invariant closes the proof.

Judged by blind competition, not by press release

The University of Manchester

Test-Comp 2026

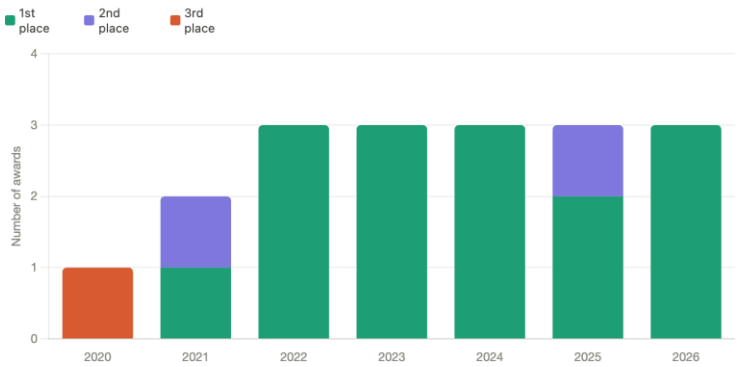
FuSeBMC: 1st in C.Overall, C.Cover-Error and C.Cover-Branches.

SV-COMP 2026

ESBMC-kind: 2nd in C.ReachSafety, ahead of Symbiotic.

Rust Foundation, Aug 2025

Selected for the Rust standard library verification initiative, with Flux, Kani and VeriFast.



FuSeBMC at Test-Comp, 2020–2026

Fourteen years of the same discipline

- Benchmarks change yearly, so scores aren't comparable year to year.
- The method is constant: public tasks, a blind run, organisers who are not us.
- GenAIDE should hold its AI-generated artefacts to that standard: a shared benchmark, a blind run, a published table.

Annual, blind, third-party benchmarking, the same evidence any competitor is judged on.

Production code, not benchmarks: AWS, NVIDIA, Google

2
CVEs in the AWS SDK for C++, fixed in release 1.11.862

15 of 18
reported defects fixed upstream by AWS, NVIDIA, Google and vLLM maintainers

240
public, re-runnable verification harnesses across six repositories

AWS SDK for C++

Base64 decoder: two out-of-bounds defects, CVE-2026-19642/19643, fixed in 1.11.862.

DateTime parsers: two more integer overflows (PR #3896).

NVIDIA OpenSMA

Out-of-bounds array write, proven reachable from a single MCTP network packet. NVIDIA confirmed and fixed it.

22 modules of C++ server-management firmware.

vLLM

Eight findings: CLI flags that pass every validator, then crash the engine. Five fixed upstream.

Config validation and KV-cache integer arithmetic (PRs #43794, #44070, #44207).

Also under the checker: Chromium Dashboard, AWS Neuron NKI and boto3, six repositories in all.

> The same checker that wins the competitions finds real defects in code that ships.

vLLM: verifying the engine that serves the models

A malformed flag passes argparse and every validator, then kills the engine at init.

31
verification targets over the config chain and KV-cache arithmetic

8
live findings, filed as 6 upstream issues

5 of 6
issues fixed upstream; one still open

`--block-size 0`	ZeroDivisionError at engine init	**fixed**
`--hash-block-size -k`	infinite loop in request_block_hasher	**fixed**
`--max-logprobs <neg>`	accepted in silence, confusing error later	**fixed**
`--num-gpu-blocks-override 0`	bare AssertionError at engine init	**open**

Every finding is a public issue; status re-checked 2026-08-27 in source. Harness for row 1: turing-awards-esbmc/8-vllm-block-size.py

> The stack that serves the models is ordinary software, and it breaks in ordinary ways.

AI writes code fast; correctness does not follow

Category	Avg Prop. Viol. per Line	Rank	$\mathcal{VS}$	Rank	$\mathcal{VF}$	$\mathcal{VU}$ (Timeout)	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.

FormAI-v2: 310,531 AI-generated C programs, nine LLMs. VS = success, VF = failed, VU = timeout.

4.23% – 15.48%

of AI-generated C programs verified successfully, across the nine LLMs tested

The ranking is not the point.

The best model passed about one time in six.

It is not a prompt problem.

Ordinary C, checked against ordinary memory-safety and arithmetic properties.

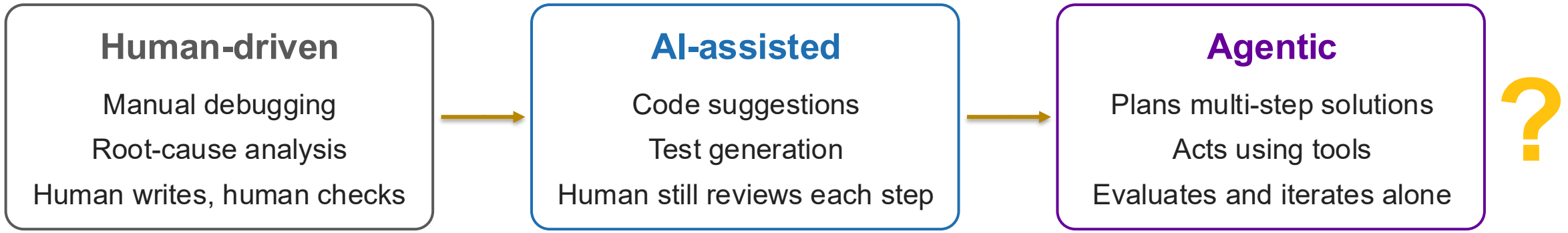
It gets worse as we delegate more.

Agentic systems revise code with no human reading each step.

If AI writes more of the artefact, something must check it, and it cannot be another LLM.

Tihanyi, Bisztray, Ferrag, Jain & Cordeiro, "How secure is AI-generated code: a large-scale comparison of large language models", Empirical Software Engineering 30(2):47, 2025, DOI 10.1007/s10664-024-10590-1.

The AI Partner in Engineering will not wait for a reviewer



The GenAIDE proposal: an AI partner that can "pro-actively make suggestions for design choices, preferences and strategies, explores and validates own concepts, and communicates them to the teams."

The question this talk exists to ask: who verifies that?

> Once you ask an AI for a result rather than a step, the tools have to prove it is right.

Cooperation, not replacement: the fast and the sound

Language models

- Fast, fluent, general
- Guess what humans find hard: loop invariants, contracts, specifications, test inputs
- **No guarantee whatsoever**

The interface

is the research problem

What is passed, in what form, and what happens when it is wrong

Proof engines

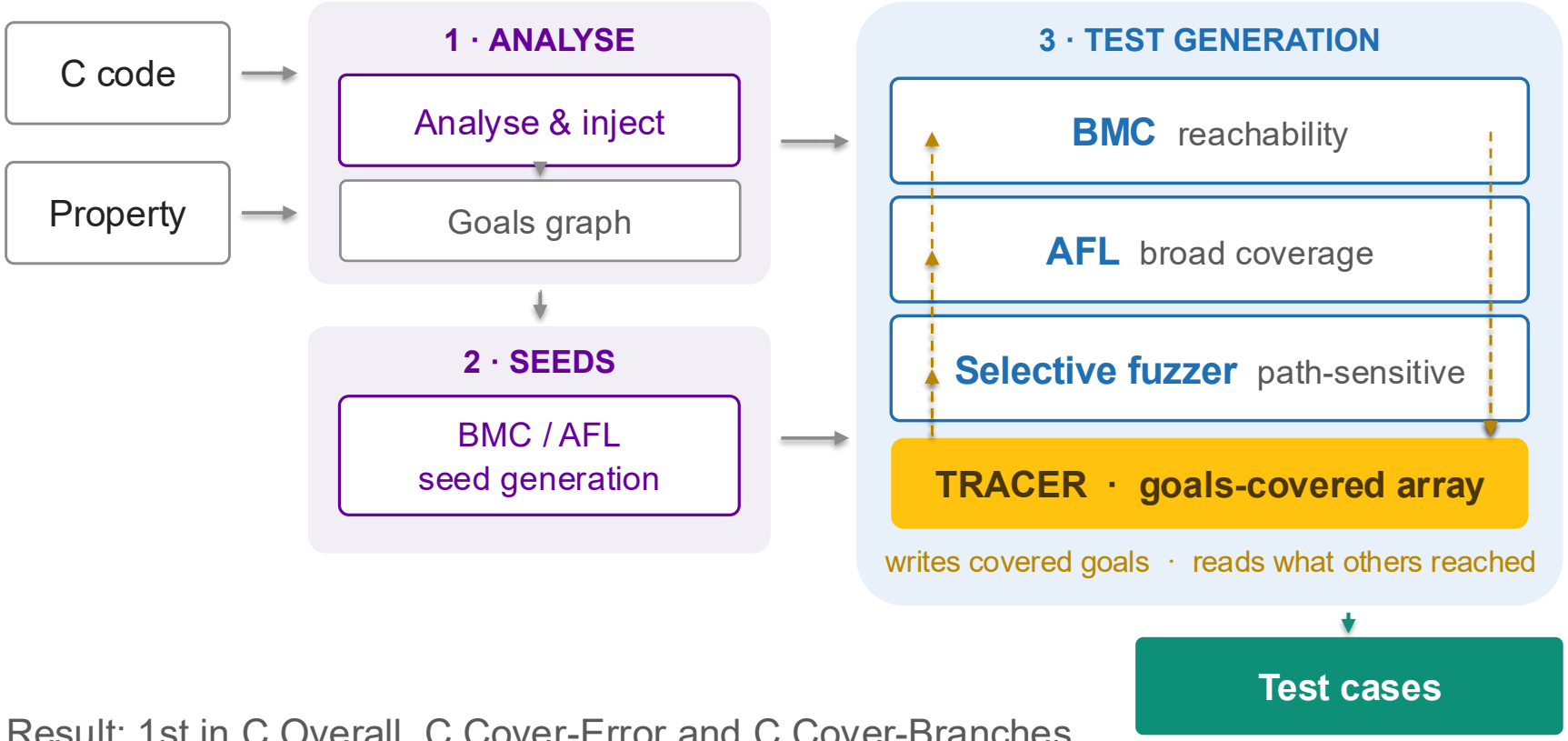
- Exhaustive within their bound
- Verdicts are machine-checkable; a counterexample is executable
- **Expensive; specification**

The design rule we keep arriving at

Let the model propose anything. Let nothing through that the proof engine has not confirmed. Then a wrong guess costs time, never correctness.

> LLMs are fast but unsound. Verifiers are sound but slow. A careful interface gives you both.

FuSeBMC: a portfolio with a shared coordinator



Result: 1st in C.Overall, C.Cover-Error and C.Cover-Branches at Test-Comp 2026.

Three engines

Each strong where the others are weak.

One coordinator

The Tracer records covered goals in shared memory.

It feeds back

New coverage becomes a seed for the next round.

> No engine talks to another: they cooperate only through a shared record of covered goals.

The bound problem: 100 unrollings for one assertion

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

Correct, but only after 100 unrollings.

An inductive invariant replaces the loop

Holds on entry, preserved by every iteration:

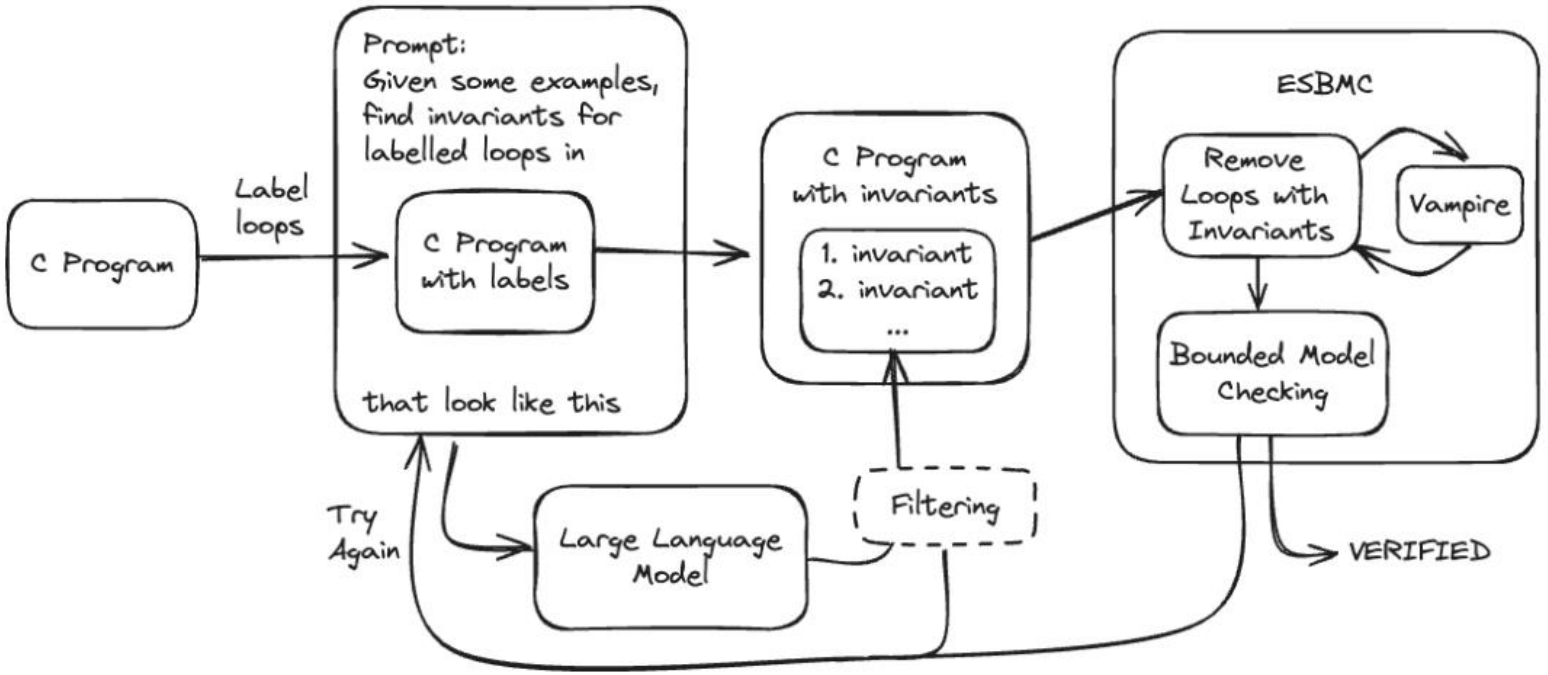
```
__ESBMC_loop_invariant(0 <= x and x <= 100)
__ESBMC_loop_invariant(not (x <= 50) or y == 50)
__ESBMC_loop_invariant(not (x > 50) or y == x)
```

The bound disappears.

A wrong guess costs time, never correctness.

> Invariant synthesis is the classic bottleneck in automated verification, and a good fit for LLMs.

ibmc: the LLM proposes, two proof engines dispose



Vampire checks that the invariant is inductive before ESBMC uses it.

Sound, not complete

A proof is always correct. The LLM costs time, never correctness.

Zero-shot is not enough

67–75% of iterations wasted.

Guide, do not filter

Post-hoc filtering limits flexibility.

Balance the prompt

Too much explanation distracts.

Soundness is preserved by construction: nothing the model says is trusted, only checked.

Pirzada, Reger, Bhayat & Cordeiro, "LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling", ASE 2024, pp. 1395–1407. DOI 10.1145/3691620.3695512.

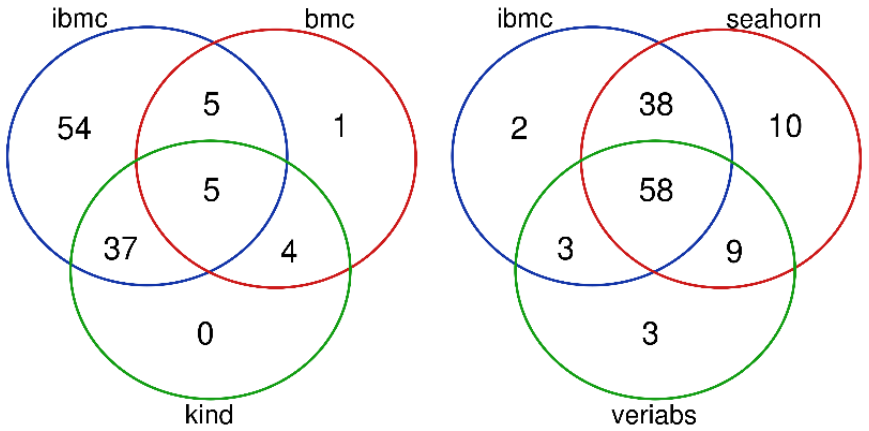
101 programs proved, none of them wrongly

Benchmark programs solved

16	46	101	115
plain BMC	k-induction	ibmc	SeaHorn

We do not win outright: 2 of ibmc's 101 are reached by nothing else.

Which programs each engine reaches



Sound by construction

Every one of the 101 is a real proof. The LLM cannot make the verdict wrong, only slower.

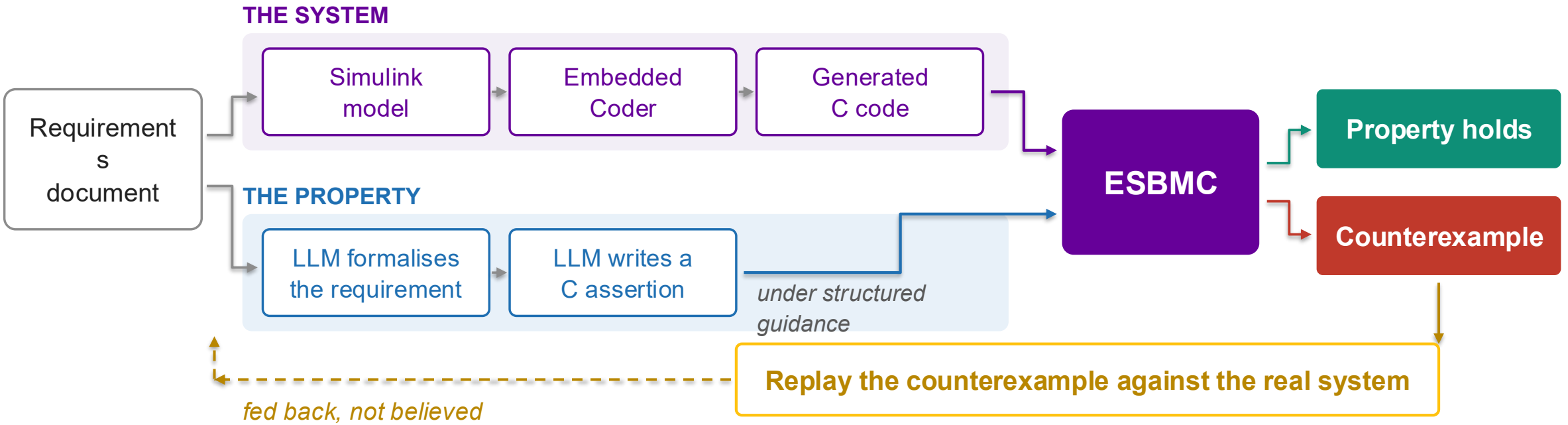
Complementary, not redundant

Different engines miss different programs: the argument for a portfolio.

LEMUR's invariants through our ESBMC verified 6 more; ours verified 10.

> The interesting result is not the total; it is that different engines miss different programs.

SpecVerify: from prose to a checked property



Their toolchain, not ours

Simulink and generated C: what GenAIDE's partners already use.

Two phases

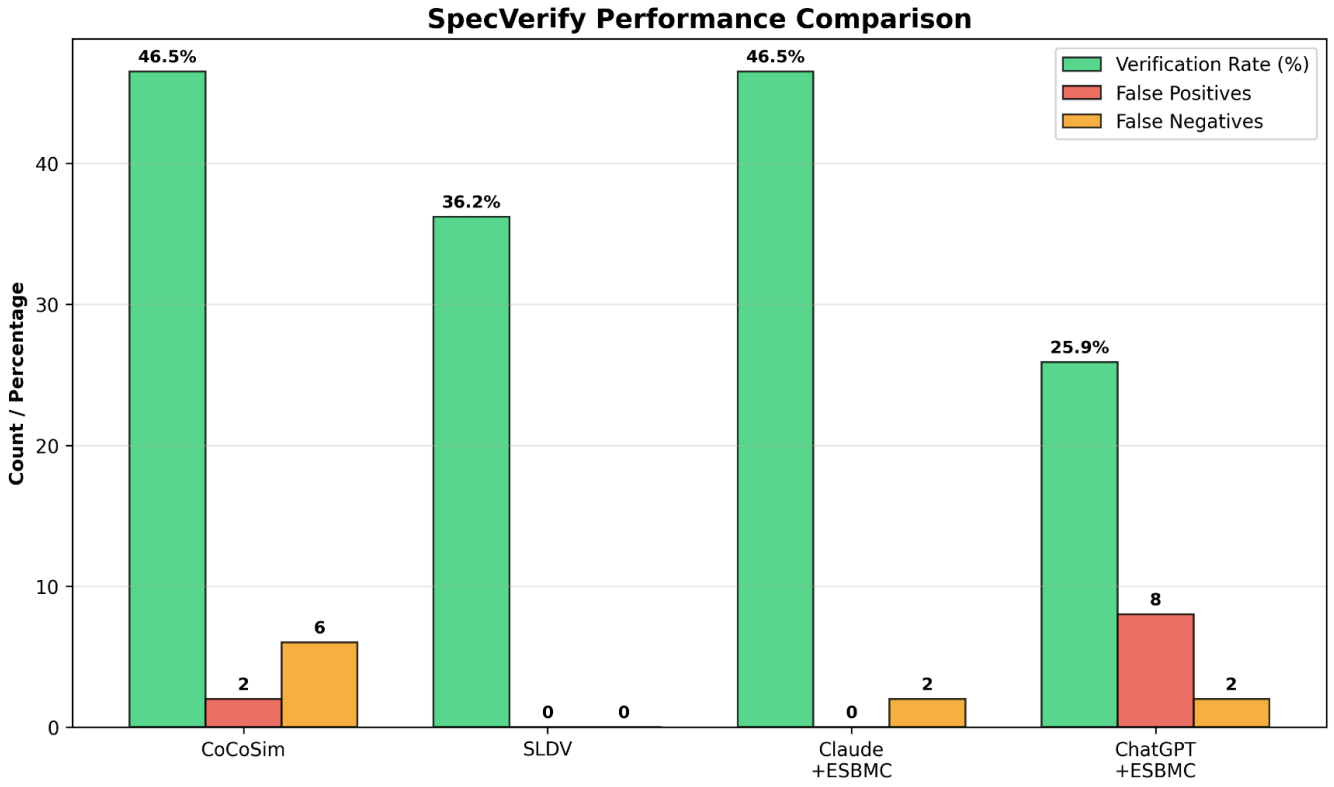
Formalise the requirement, then turn it into a C assertion.

The loop that closes it

The LLM is never the final authority; a property that fails is fed back.

> The specification is the bottleneck in industrial verification, and it is written in prose.

Expert-level agreement, with no false positives



79.31% logical equivalence with expert-verified properties

0 false positives: the verifier never cried wolf

2 previously unreported floating-point errors found

Read the chart carefully

46.5% matches NASA's CoCoSim baseline; it does not beat it.

CoCoSim: 2 false positives, 6 false negatives. Claude+ESBMC: 0 and 2, and no manual variable mapping.

The model choice matters: the same pipeline with a weaker model produced 8 false positives.

Three systems, three cooperation interfaces

System	What cooperates	The interface	What it bought us
FuSeBMC	Model checker, coverage fuzzer, selective fuzzer	A shared-memory array of covered goals	1st in all three C categories, Test-Comp 2026
ibmc	LLM proposes invariants; Vampire and ESBMC check them	Prompt → candidate invariant → inductiveness check → proof	101 programs proved, vs 16 unaided. Sound by construction
SpecVerify	LLM formalises requirements; ESBMC checks the generated C	Prose → assertion → verdict → counterexample replayed against the system	79% agreement with experts, 0 false positives, 2 new errors found

The generative side may be wrong; the checking side may never be skipped.

Cooperation gives you speed and correctness, if the interface is designed on purpose.

Where this plugs into what GenAIDE is already building

GenAIDE builds	The assurance question	What we bring
Proposes and optimises designs	Does it satisfy the constraints that were actually stated?	**Requirements formalisation (SpecVerify)**
Explains its reasoning	Is the explanation faithful, or merely plausible?	**Counterexamples you can execute and re-check**
Generates optimisation algorithms	Does it terminate, stay in bounds, preserve invariants?	**ESBMC over the generated C, C++, Python or Rust**
Hybrid human, AI teams	What must the human trust, and on what evidence?	**Machine-checkable verdicts, not confidence scores**

Our proposition, not a GenAIDE work-package statement.

> Four things GenAIDE builds, four questions of trust, one answer: check, don’t assume.

Six problems we think are worth a doctoral thesis

The University of Manchester

1 What does "correct" mean for a design?

Code gets memory safety by default. A generated geometry has none: someone must build the property library.

2 Verification at the speed of generation

Agents propose hundreds a minute; proof takes seconds to hours. Only incremental checking closes the gap.

3 Explanations that can be falsified

An explanation nobody can check is a liability. Counterexamples are checkable; can explanations be?

4 Guarantees for multi-agent workflows

Verifying each agent says nothing about the workflow. Assume-guarantee reasoning here is largely unsolved.

5 Alignment: whose limit is it?

“Reasonable” cannot be formalised. The model picks a bound silently; a checker forces the question.

6 Evidence that regulators and engineers accept

DO-178C already admits formal methods via DO-333. What is the audit trail for a generated artefact?

Every one of these is a joint problem: neither community solves it alone.

Nine current doctorates, one question, four angles on it

The University of Manchester

One line, from generation at one end to certification at the other.

Generate the specification Requirements and contracts, checked by a prover	**Weiqi Wang · Taohong Zhu**
Generate the proof ingredient Invariants that free a bounded checker from its bound	**Muhammad A. A. Pirzada**
Generate the fix, and the test Repair and test generation, gated on verification	**Yiannis Charalambous**
Secure the generative system itself Attacks on, and defences for, the models themselves	**Adrians Skapars · Fatima Abacha**

Plus Bruno Farias on Python semantics, our two GenAIDE candidates next, and four recent completions.

Not a set of unrelated projects: one research programme, nine doctorates deep.

Making a model produce something a prover will accept

Weiqi Wang

PhD candidate, S3 group

Requirements formalisation for industrial verification

Built SpecVerify: requirements become C assertions.

IEEE RE 2025 · with Marie Farrell and Liping Zhao

Taohong Zhu

PhD candidate, 2023–2026

LLM agents for requirements and repair

ReqInOne: an agent that writes requirements specifications.

IEEE RE 2025 · with Youcheng Sun

Muhammad A. A. Pirzada

PhD candidate, S3 group

LLM-generated invariants for unbounded proofs

Built ibmc, then ConVer: contracts plus invariant synthesis.

ASE 2024 · with Konstantin Korovin

Yiannis Charalambous

PhD candidate, S3 group

Automated program repair with verification and LLMs

An agent proposes a fix; the verifier decides if it is one.

AST 2025 · GeCoIn@ECAI 2025 · ASE 2025

Adrians Skapars

PhD candidate, S3 group

Security of the generative model itself

Exact inversion: recovering the input behind an output.

With Edoardo Manino and Youcheng Sun

Fatima Abacha

PhD researcher, S3 group, 2023–2026

Privacy and robustness in federated learning

Backdoor defences, and synthetic data when real data cannot be shared.

With Mustafa Mustafa

Six of the nine current doctorates, on the seam between proposal and proof.

The best evidence a doctorate worked: the tool

Tong Wu PhD 2026	**Efficient concurrent software verification with BMC and deep learning**	Scheduling and partial-order reduction in ESBMC v7.7. SAS 2024 · TACAS 2025
Rafael Sá Menezes PhD 2025, now Research Associate	**A formal semantics for GOTO in the CProver ecosystem**	The goto-transcoder behind ESBMC's acceptance into the Rust initiative. Rust Foundation, Aug 2025
Mohannad Aldughaim PhD 2025	**Interval analysis and methods in software analysis**	Abstraction that keeps the solver from drowning on large programs. FASE 2023 · CoRR 2024
Kaled M. Alshmrany PhD 2023	**Efficient hybrid fuzzing for vulnerabilities and high coverage**	FuSeBMC, repeat Test-Comp winner since 2021, now maintained by Veribee. FASE 2022 · Test-Comp 2021–2026
Fatimah K. Aljaafari PhD 2023	**Black-box cooperative verification for vulnerabilities in concurrent programs**	EBF couples BMC with fuzzing; it exposed a data race in wolfMQTT. IEEE Access 2022 · TACAS 2023

Five theses, each leaving a tool behind: the training model we would bring to GenAIDE.

Our two GenAIDE projects: generate, then repair

Two of GenAIDE's Doctoral Candidates are ours; both extend ESBMC.

DC7 Manan Mal

Robust Code Generation for Simulation and Optimization

→ Safe LLM code generation, formally verified

→ A requirements and constraints library

→ Co-develop GenAIDEBench-V

WP2, WP4 · C++, Java, Python, Matlab

DC8 Mohd Ruhul Ameen

LLM-based Simulation/Optimization Software Self-repair

→ Chain-of-thought consistency checking

→ Verification results drive the repair: BMC, k-induction, abstract interpretation, CP/SMT

→ Validate repairs on GenAIDEBench-V

WP3, WP4 · the same four languages

Two halves of one loop: DC7 checks what the model generates; DC8 repairs it.

This is the generate-then-verify loop of the whole talk, written into the workplan.

Four things that need a conversation, not a proposal

The University of Manchester

A	**Secondments, both directions**	A candidate spends three to six months here, learning to verify what they generate. We send someone back.
B	**GenAIDEBench-V, run like a competition**	Already in the workplan, co-developed by our two DCs. Hold it to the SV-COMP standard: public tasks, blind runs.
C	**A pilot on one real toolchain**	Point ESBMC at one partner-nominated component: counterexamples with concrete inputs, or a proof that none exists. Apache-2.0.
D	**Co-supervision and a summer-school session**	A Science & Skills session on verifying generated artefacts, and co-supervision where a candidate's topic touches assurance.

C is the cheapest to start and the fastest to produce something checkable.

IN CLOSING

LLMs are fast but unsound.
Proof engines are sound but slow.
Cooperation gives you both,
if you design the interface on purpose.

The group

Systems & Software Security at Manchester: 13 academics, a national cyber-security centre, and a verification platform used worldwide.

The method

Let the generator propose anything. Let nothing through that a proof engine has not confirmed.

The people

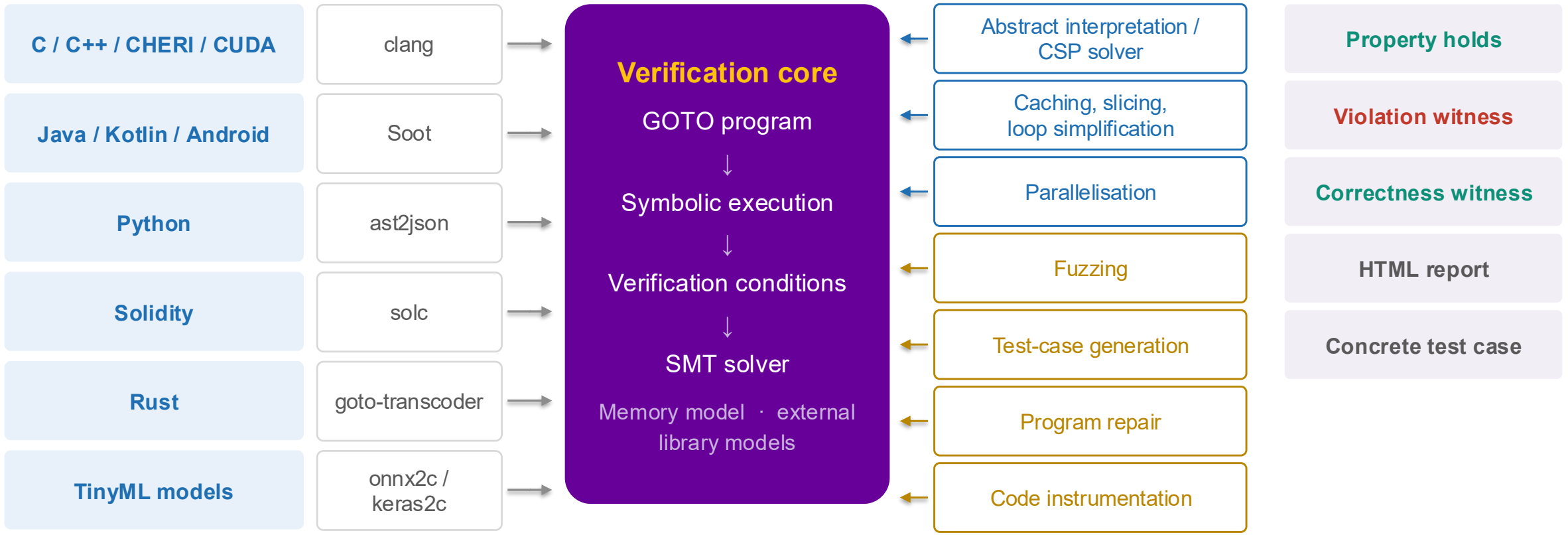
Nine current doctorates on exactly this seam; two are GenAIDE's own Doctoral Candidates.

Prof. Lucas C. Cordeiro
lucas.cordeiro@manchester.ac.uk
cs.manchester.ac.uk/research/expertise/systems-and-software-security/ · github.com/esbmc/esbmc

MANCHESTER 1824
The University of Manchester

Thank you. Backup slides follow: full ESBMC architecture, competition record, and the reference list.

ESBMC in full: eight front-ends, one verification core



Blue: components that cooperate with other analysis engines. Gold: components that consume or produce artefacts for other tools, the surface a new generative component attaches to.

> Every language reduces to one intermediate form, so a new front-end inherits the whole engine.

Roadmap: automated formal verification of requirements

Formalization, completed

SpecVerify: natural-language requirements to checkable assertions, published at IEEE RE 2025.

Decomposition, in progress

System-level properties decomposed into component obligations, driven by counterexample-guided inductive synthesis.

Auditing, planned

Property validation and coverage analysis: evidence that the properties checked were the ones that mattered.

Stage 3 is where an audit trail for AI-generated artefacts would be built.

Competition record: what the numbers do and do not show

The University of Manchester

Itemised from the projects' own award pages: 22 medals at SV-COMP and 18 at Test-Comp, 2012–2026.

Year	SV-COMP (ESBMC)	Test-Comp (FuSeBMC / ESBMC)
2026	2nd ReachSafety	1st Cover-Error · 1st Cover-Branches · 1st Overall
2025	2nd ReachSafety	1st Cover-Error · 2nd Cover-Branches · 1st Overall
2022–24	None	1st Cover-Error and Cover-Branches each year; 1st Overall 2023–24
2020–21	3rd Falsification (2020)	1st Cover-Error (2021) · 3rd Bug Finding (2020)
2012–19	6 category golds; 3rd in Overall in 2012, 2013 and 2018	None

Two things not to say

→ SV-COMP awards medals per meta category only: first places in sub-categories (ReachSafety-XCSP in 2021, MemSafety-Juliet in 2024) are not medals.

→ The 2026 gold in Java.Overall belongs to JBMC, not ESBMC; ESBMC-kind took one silver, in C.ReachSafety. Affiliations differ; see the speaker notes.

Only the itemised 40 is checkable line by line; that is the number to quote.

References and sources

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.
Alshmrany, K., Aldughaim, M., Bhayat, A., Cordeiro, L.	FuSeBMC v4: Smart Seed Generation for Hybrid Fuzzing (Competition Contribution). FASE 2022, pp. 336–340. Journal version: Formal Aspects of Computing 36(2):12, 2024.
Pirzada, M. A. A., Reger, G., Bhayat, A., Cordeiro, L.	LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling. ASE 2024, pp. 1395–1407. DOI 10.1145/3691620.3695512.
Wang, W., Farrell, M., Cordeiro, L., Zhao, L.	Supporting Software Formal Verification with Large Language Models: An Experimental Study. IEEE RE 2025, pp. 423–431. DOI 10.1109/RE63999.2025.00049.
Tihanyi, N., Bisztray, T., Ferrag, M. A., Jain, R., Cordeiro, L.	How secure is AI-generated code: a large-scale comparison of large language models. Empirical Software Engineering 30(2):47, 2025. DOI 10.1007/s10664-024-10590-1.
Wu, T., Xiong, S., Manino, E., Stockwell, G., Cordeiro, L.	Verifying Components of Arm® Confidential Computing Architecture with ESBMC. SAS 2024, pp. 451–462.
Charalambous, Y., Coelho, C. N., Lamb, L. C., Cordeiro, L.	UnitTenX: Generating Tests for Legacy Packages with AI Agents Powered by Formal Verification. GeCoIn @ ECAI 2025.
Zhu, T., Cordeiro, L., Sun, Y.	ReqInOne: A Large Language Model-Based Agent for Software Requirements Specification Generation. IEEE RE 2025.

Non-bibliographic sources

GenAIDE: cordis.europa.eu/project/id/101226927 · genaide.eu · honda-ri.de/genaide · S3 group: cs.manchester.ac.uk/research/expertise/systems-and-software-security/ · People: research.manchester.ac.uk/en/persons/ · ESBMC: github.com/esbmc/esbmc · Competitions: sv-comp.sosy-lab.org/2026 · test-comp.sosy-lab.org/2026 · Rust: rustfoundation.org, 7 Aug 2025 · ACE-CSR: ncsc.gov.uk. All accessed 2026-08-27.

Turn "repeat until" into "repeat k times"

The worked example has no loop, so unfolding leaves it unchanged. Here is the step on a program that has one: the loop from Pattern B.

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

The loop as written runs an unknown number of times.

```
if x < 100:
    x = x + 1
    if x > 50: y = y + 1
    if x < 100:
        x = x + 1
        if x > 50: y = y + 1
        assert x >= 100   # unwinding assertion
```

Unfolded to bound k = 2: two nested copies, plus a check that k was enough.

- If the unwinding assertion can fail, k was too small: ESBMC reports a violated unwinding assertion rather than call the program safe. --incremental-bmc and --k-induction raise k automatically.
- This loop needs k = 100 to be sure. Unfolding 100 copies is exactly the cost k-induction and LLM-generated invariants exist to avoid.

> Unfolding is cheap when k is small and expensive when it isn't.

Shrink the formula before the solver ever sees it

The University of Manchester

Constant propagation

```
n = 4
r = n * 2
# becomes: r = 8
```

A value known at compile time replaces every place it is used.

Slicing

```
report = build_report()
assert x > 0
# report is dropped:
# it never reaches the property
```

Remove code that cannot affect the property being checked before solving.

Caching

```
p = f(x) + 1
q = f(x) - 1
# f(x) computed once,
# reused for q
```

An expression computed more than once is computed only once.

None of these fire on the ten-line worked example; it is already minimal. They matter at the size ESBMC actually verifies: thousands of lines, where the unoptimised formula would be too large for a solver to finish.

> Optimisation never changes what's true. It changes whether the solver finishes.

Every line becomes one equation, for both branches at once

SSA = Static Single Assignment: every variable is written exactly once. Back to the worked example.

Python source	SSA form
a = [nondet_int(), nondet_int()]	a0 = a fresh, unconstrained array
i: int = nondet_int()	i0 = a fresh, unconstrained integer
x: int = nondet_int()	x1 = a fresh, unconstrained integer
if x == 0:	g1 = x1 == 0
a[i] = 0 (if-branch)	a1 = a0 WITH [i0 := 0]
else: a[i + 2] = 1 (else-branch)	a2 = a0; a3 = a2 WITH [2+i0 := 1]
(the two branches merge)	a4 = g1 ? a1 : a3
assert a[i + 1] == 1	t1 = a4[1+i0] == 1

a4 = g1 ? a1 : a3 is the whole trick: both branches live in one formula, so no path is discarded before the solver sees it.

> This SSA form is the same maths, whichever language the program started in.

What the program does, versus what you hoped was true

C, the program's meaning

```
g1 = (x1 == 0)  ∧
a1 = a0 WITH [i0 := 0]  ∧
a2 = a0  ∧
a3 = a2 WITH [2+i0 := 1]  ∧
a4 = g1 ? a1 : a3
```

every SSA equality, conjoined

P, the property

```
t1
```

the assert's condition, and nothing else

Why check ¬P, not P?

- An SMT solver natively answers one question: is this formula satisfiable?
- To prove P always holds, ask the solver to satisfy its opposite, ¬P.
- If it can't, P holds everywhere; if it can, the satisfying assignment is a counterexample to P.

C is everything the program could do. P is the one thing you hoped was always true.

One satisfying assignment is all it takes

The solver returns satisfiable and hands back a concrete assignment. Trace it through the worked example:

`i = 0,  x = 0`	x == 0, so the if-branch runs
`a[i] = 0`	sets a[0] = 0, no effect on the property
`a[1] = 0  # solver's pick`	the property never told the program to set a[1]
`assert a[i + 1] == 1`	reads a[1] == 1 → asserts 0 == 1 → false

Why a passing test proves nothing here

- Nothing in the program constrains a[1], so whether a test fails depends on whatever a[1] happened to contain.
- The solver did not get lucky: it searched the SSA formula, proved an assignment exists that makes the property fail, and reported one.

This is the counterexample the main talk promised: concrete i, x, and array contents.