西安电子科技大学
XIDIAN UNIVERSITY

AcslBench: A Verified C/ACSL Corpus and Benchmark with Compositional Call Chains for Formal Specification Synthesis

Junjie Hu · Cheng Wen · Bin Yu · Jialun Cao · Dugang Liu · Weidi Sun · Haokun Li · Shengchao Qin · Cong Tian
Xidian University · HKUST · Shenzhen University · Peking University

Automated Software Engineering 2026 | huggingface.co/datasets/noBuggie/AcslBench

Background: Formal Verification and Specification Synthesis

Formal verification proves code against a machine-checkable specification

seL4 (OS kernel) | Linux kernel libs | Aerospace controllers | .....

ACSL — what a C specification is made of (numbers match the figure)

1. **Function contract** — requires / ensures / assigns
2. **Loop invariant** — partial-sum property of the loop
3. **Ranking function** — loop variant ⇒ termination
4. **Caller contract** — composes with callee contracts

All checked automatically by Frama-C/WP — the verifier is the ground truth.

The dilemma

✓ **LLMs write C well**
correct, compilable code

✗ **LLMs write ACSL poorly**
wrong annotation ⇒ unverifiable program

→ **What is missing: high-quality verified C/ACSL data at scale.**

```
1  /*@
2  requires \valid(buf);                                  1. Function Contract
3  requires \valid(buf+(0 .. num-1));
4  ensures \result == \sum(0, num-1, \lambda int k; buf[k])%256;
5  assigns \nothing; */
6  uint08 CheckSumAdd08(const uint08 *buf, uint32 num) {
7      uint32 sum = 0;
8      /*@
9      loop invariant 0 <= i <= num;                      2. Loop Invariant
10     loop invariant i == 0 ||
11         sum == \sum(0, i-1, \lambda int k; buf[k]);
12     loop assigns i, sum;
13     loop variant num - i; */                           3. Ranking Function
14   for (uint32 i = 0; i < num; i++) {
15       sum = sum + buf[i];
16     }
17     return (uint08)(sum%255);
18 }
19
20 /*@
21 requires pIp->rdLen == 19;                            4. Function Contract
22 requires pIp->pbuff[17] != pIp->frm;
23 requires pIp->pbuff[0] != 0xEB || pIp->pbuff[1] != 0x90;
24 ensures pIp->bComSuc == FALSE32;
25 ensures pIp->cntLenRdS == 0;
26 ensures pIp->cntUpdataS == 0;
27 ensures pIp->cntUpdata0 == 0;
28 ensures pIp->cntHeadS == \old(pIp->cntHeadS) + 1;
29 ensures pIp->totalHeadS == \old(pIp->totalHeadS) + 1;
30 ensures pIp->cntHead0 == \old(pIp->cntHead0) + 1;
31 ensures pIp->totalHead0 == \old(pIp->totalHead0) + 1;
32 ensures pIp->pbuff[17] == pIp->frm;
33 ensures pIp->pbuff[0] != 0xEB || pIp->pbuff[1] != 0x90;
34 assigns pIp->bComSuc, pIp->cntLenRdS, pIp->cntUpdataS,
35 pIp->cntUpdata0, pIp->cntHeadS, pIp->totalHeadS,
36 pIp->cntHead0, pIp->totalHead0, pIp->frm; */
37 void Fg333saCheckFun(Fg333saCheck *pIp) {
38   uint08 chksum;
39   Fg333saCheck *p = (Fg333saCheck *)pIp;
40   p->bComSuc = FALSE32;
41   if (p->rdLen == 19) {
42     p->cntLenRdS = 0;
43     if (p->pbuff[17] != (p->frm)) {
44       p->cntUpdataS = 0;
45       p->cntUpdata0 = 0;
46       p->frm = p->pbuff[17];
47       if ((p->pbuff[0] == 0xEB) && (p->pbuff[1] == 0x90)) {
48         p->cntHeadS = 0;
49         p->cntHead0 = 0;
50         chksum = CheckSumAdd08(&p->pbuff[0], 18);
51         ...
```

Figure 1. Verified aerospace controller code (Frama-C/WP)

The C/ACSL Data Gap

Ecosystem	Verified corpus	Scale
Lean	LeanNavigator (2025)	4.7M theorems
Rust/Verus	VeruSyn (2026)	6.9M programs
C/ACSL	**all prior sets combined**	**< 700**

Sources: arXiv:2503.04772; arXiv:2602.04910; Table 1 below

Dataset	#Programs	Role	Funcs/Prog
SyGuS	133	Benchmark	1
OOPSLA13	46	Benchmark	1
SV-COMP (portion)	21	Benchmark	1–2
FramaC-Problems	51	Benchmark	1–2
Verker	48	Benchmark	1–2
WP Examples	295	Tutorial	1–2
Live-FMBench	630	Benchmark	2
AcslBench Corpus	**504,587**	**Corpus**	**1**
AcslBench Benchmark	**495**	**Benchmark**	**2–10**

Table 1: Scale and function complexity of C/ACSL datasets

Lean and Verus accumulated millions of verified samples. C/ACSL had nothing comparable — by three to four orders of magnitude.

Existing Fragile data and evaluation

1. Every sample must pair executable C code with verifier-checkable specifications

A wrong annotation is not a tolerable noisy label — it can make the program unverifiable, or worse, silently specify unintended behavior.

2. Existing evaluations stop at single functions and local loops

Realistic verification requires composing contracts across function calls — assuming callee guarantees and propagating them through the caller.

Verifier-in-the-Loop Corpus Construction

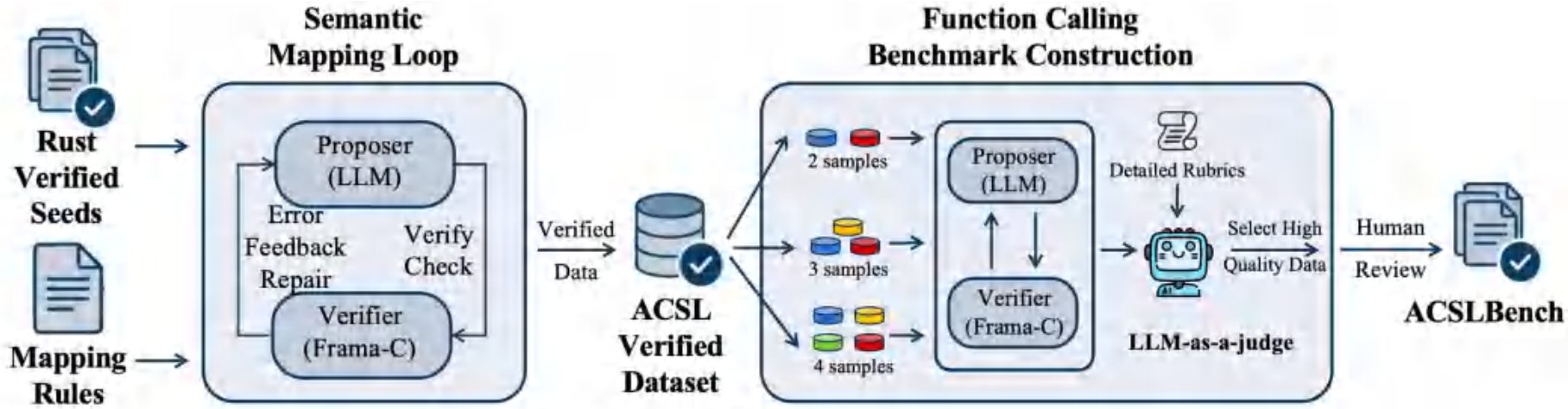


Figure 2: Overview of the AcslBench construction pipeline

Seeds — ~0.7M verified Rust/Verus programs from VeruSyn (6.9M)

Translation — LLM maps seeds into C + ACSL, preserving verification intent

Verification — Frama-C/WP discharges every proof obligation

Repair — failed candidates return with diagnostics, up to 5 iterations

Verifier feedback doubles as a repair signal: five repair iterations grow the corpus from 373,236 to 504,587 verified samples (pass rate 52.9% → 71.5%).

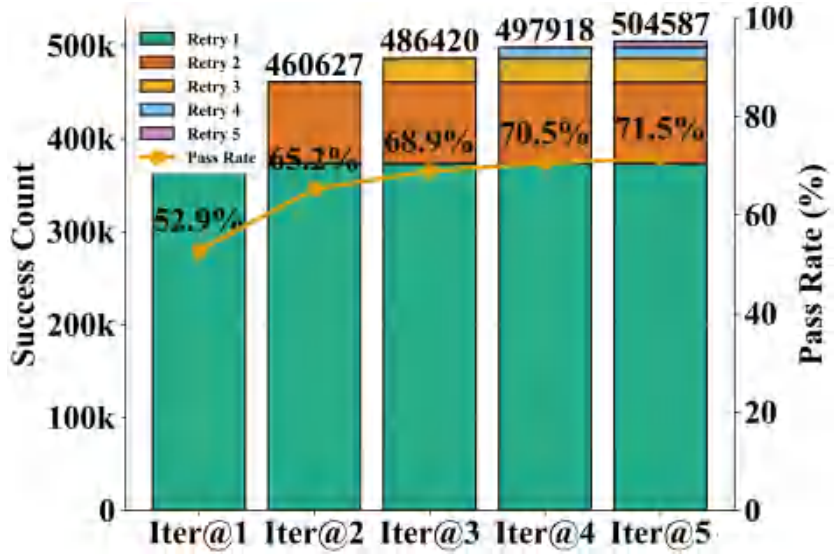


Figure 3: verified count and pass rate across iterations

Corpus Quality: What 504,587 Verified Programs Look Like

Category	Metric	Avg.	Max.
Code	Lines of Code	14.4	409
	Loops (num, nested)	(0.3, 0.2)	(22, 18)
	Call Depth / Pointer	1.0 / 0.50	2 / 1
Spec	Lines of Spec.	16.58	208
	Pred. / Logic / Lemma	1.5 / 0.6 / 1.2	76 / 31 / 161
	Quantifiers / VCs	0.72 / 11.8	1 / 1607

Table 2: Statistics of the verified C/ACSL corpus

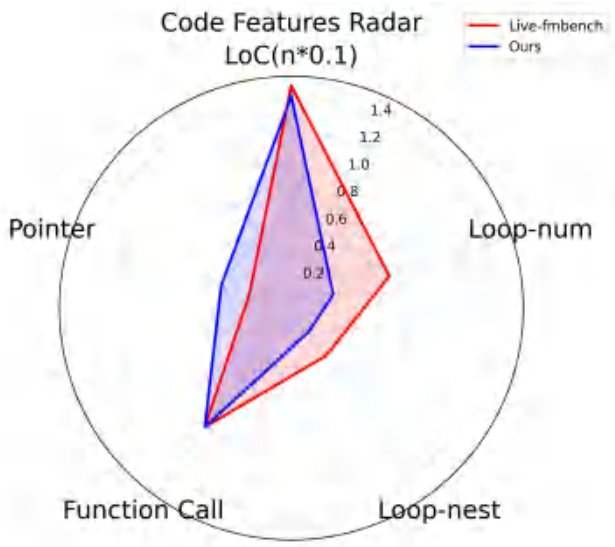


Fig. 6: Code features

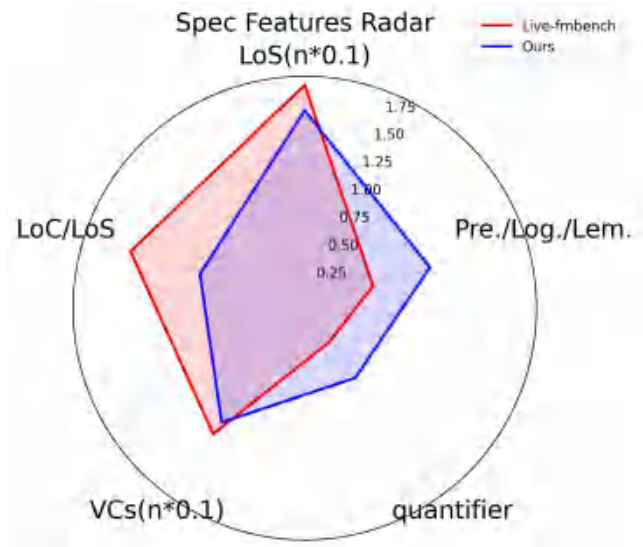


Fig. 7: Spec. features

Deliberately broad coverage

Small samples teach basic ACSL syntax and frame conditions;

Large ones bring predicates, lemmas, and quantifiers — up to 208 spec lines, 1,607 VCs.

Every sample is a **reusable** component for harder tasks.

Compared with Existing Benchmark —— Live-FMBench

Close on most dimensions; complementary in pointer and logical-specification features (Figs. 6–7).

Beyond Pretraining: Building a Harder Benchmark

The verified corpus is not only training data — the same components can be composed into new, harder evaluation tasks.

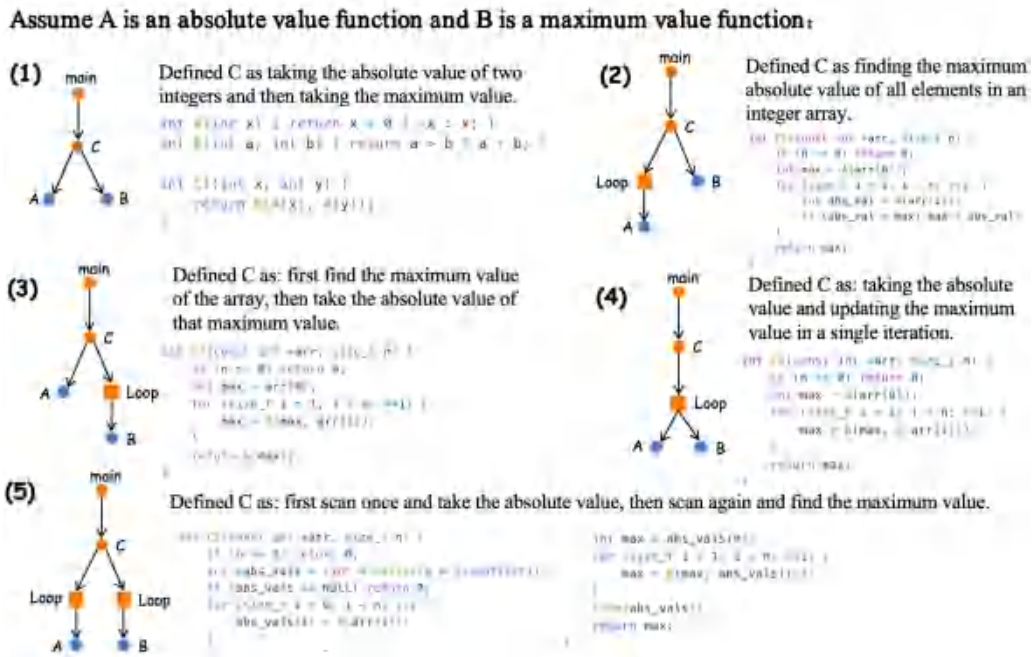


Figure 5: Five composition strategies for 2 verified components

Sample 2–4 components → LLM synthesizes caller C → Frama-C verification → LLM-as-a-judge → manual review.

Callees untouched; callers must reason across their contracts.

Level	Tasks	Calls	Loops	LoC
Simple (2 calls)	185	3.16	2.00	60.3
Medium (3 calls)	133	4.18	3.00	81.7
Complex (≥4)	177	5.48	4.05	108.7
Overall	**495**	**4.26**	**3.01**	**83.4**

Table 3: Benchmark statistics by call-count level

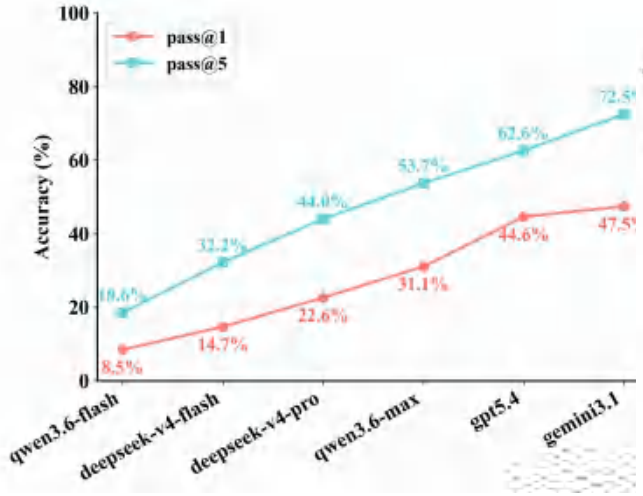


Fig. 8: Overall pass@1/pass@5

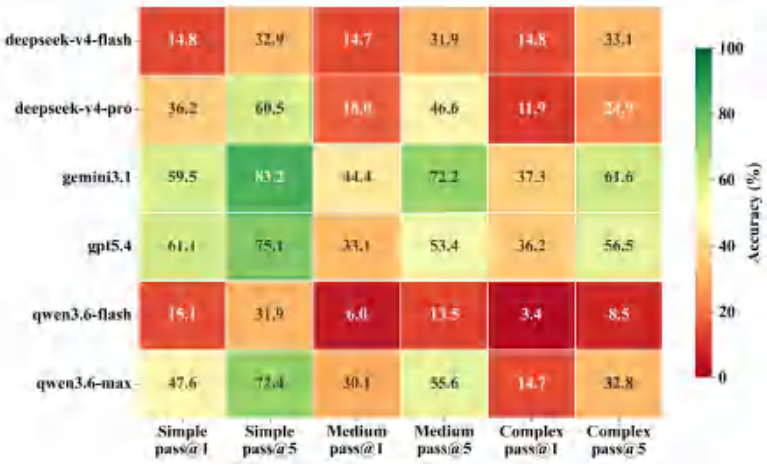


Fig. 9: By call-count level

Conclusion and Future Work

Summary

1 Verified Corpus

504,587 Frama-C/WP-verified C/ACSL samples — first corpus at training scale

1 Compositional Benchmark

495 curated tasks, 2–10 functions; difficulty stratified by call count

1 Synthesis Method

verifier-in-the-loop composition — reusable for new tasks and benchmarks

Limitations

- Data distribution: translated from Rust/Verus seeds — may inherit seed-bias rather than native C code patterns.
- Grouping measures call-count occurrences, not deeper call-chain depth.

Future work

- Multi-language dataset: the same task in multiple specification languages.
- Fine-tune ACSL generation models and agentic synthesis with verifier feedback.

Dataset: huggingface.co/datasets/noBuggie/AcslBench · Video: youtu.be/kgvvTajX2Mk · Thank you — questions welcome