Formally Specifying the Intended Behavior of the Program: LLM-Driven Neuro-Symbolic Program Specification Synthesis

Cheng Wen[1], Junjie Hu[1], Yikun Hu[2], Jie Su[1], Bin Yu[2], Dugang Liu[3], Zhiwu Xu[3], Weidi Sun[4], Shengchao Qin[1,2]*, Cong Tian[2]

[1] Guangzhou Institute of Technology, Xidian University [2] School of Computer Science and Technology, Xidian University
[3] College of Computer Science and Software Engineering, Shenzhen University [4] School of Mathematical Sciences, Peking University

1. Introduction

```
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])%255;
5  assigns \nothing; */
6  unint08 CheckSumAdd08(const unint08 *buf, unint32 num) {
7      unint32 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 (unint32 i = 0; i < num; i++) {
15       sum = sum + buf[i];
16     }
17     return (unint08)(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   unint08 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: A simplified C program from an aerospace control system, annotated with ACSL specifications

Why generate and check?

An LLM drafts ACSL, but its output may contradict the code or fail a proof. Frama-C/WP checks each candidate.

AutoSpec+ repairs failed annotations with verifier feedback and reuses those that pass.

Code → requirement → proof

The frame-checking code needs function contracts, a loop invariant and a termination argument to prove its stated requirements.

01_Validate data length. If the length is not 19, increment both the continuous length error counter and the cumulative length error counter by 1, then return a validation failure.
02_Check if the frame count is updated. If the frame count does not update, increment both the continuous data update error counter and the cumulative data update error counter by 1, then return a validation failure.
03_Validate the frame header. If the frame header matches the expected value, reset the continuous frame header error counter.
04_Validate whether the checksum is correct. Calculate the checksum of the data. If the checksum is correct, reset the continuous checksum error counter and mark the communication as successful; otherwise, increment both the continuous and cumulative checksum error counters.
05_Update continuous length read counter. If the frame count matches the expected value, increment both the continuous and cumulative length read counters.

Figure 2: The Requirement Fg333saCheckFun

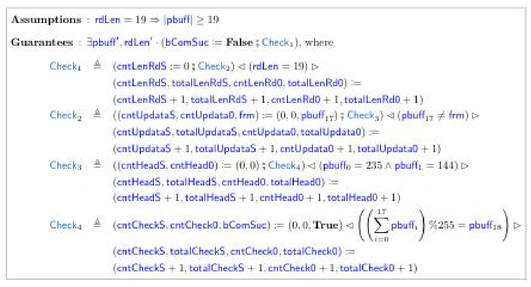


Figure 3: The Formal Model of Fg333saCheckFun

2. Existing Works

Earlier tools
Most tools only infer loop invariants.

AutoSpec+
Adds contracts, termination and cross-file proofs.

Table 1: C Program Specification Generation Tools

Method/Tool	Loop Invariant	Ranking Function	Function Contract	Complex Struct.	Inter-modular	Model Ensemble
Learning-based methods						
Code2Inv (Si et al., 2018)	✔	✘	✘	✘	✘	–
CLN2INV (Ryan et al., 2020)	✔	✘	✘	✘	✘	–
LIPuS (Yu et al., 2023)	✔	✘	✘	✘	✘	–
Clause2Inv (Cao et al., 2025)	✔	✘	✘	✘	✘	–
LLM agent-based methods						
AutoSpec (Wen et al., 2024)	✔	✘	✔	✔	✘	✘
ACInv (Liu et al., 2025)	✔	✘	✘	✔	✘	✘
SLD-Spec (Chen et al., 2025)	✔	✘	✔	✔	✘	✘
AutoSpec+	✔	✔	✔	✔	✔	✔

3. Our Neuro-Symbolic Workflow

Static analysis orders units; the LLM drafts ACSL; Frama-C/WP checks.

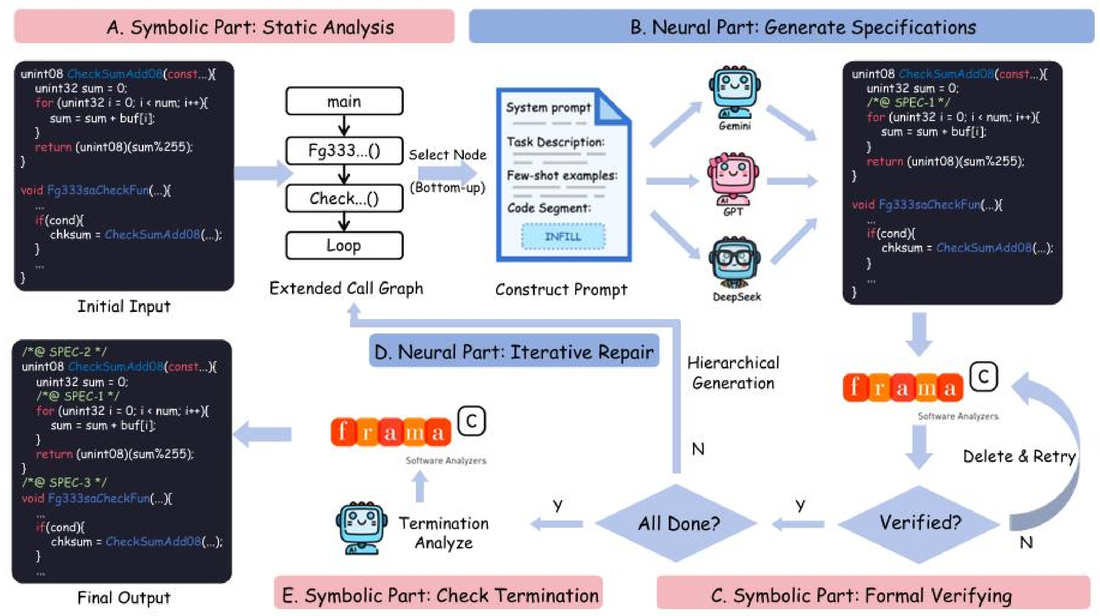


Figure 4: Overall Workflow of AutoSpec+'s Neuro-symbolic Loop

4. Cross-File Verification

A proved helper contract unlocks the caller proof

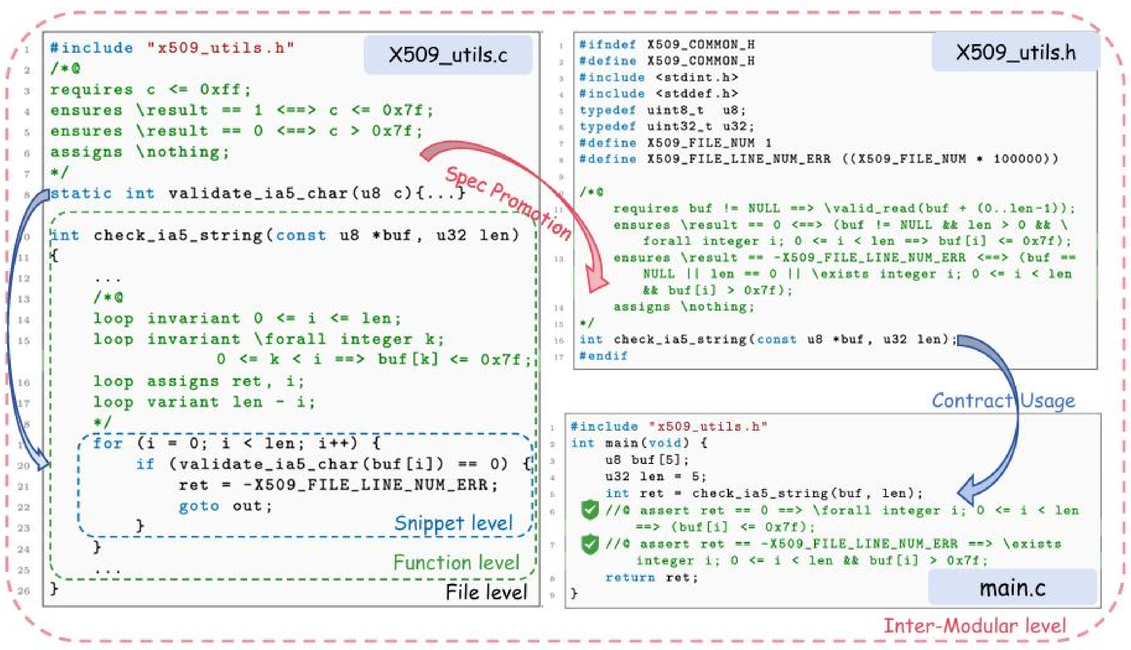


Figure 5: An X509 Parser Example with Multiple Files

AutoSpec+ proves two assertions in main.c by verifying check_ia5_string and exposing its contract in x509_utils.h.

The proof grows from loop snippets to functions, files and cross-file calls.

5. Evaluation

Verification results

Across the five suites shown, the ensemble proves 234 of 257 programs.

Table 2: Effectiveness of AutoSpec+ (including model ensemble and termination analysis)

Benchmark	Deepseek -v3.1	Qwen3 -235B	Kimi-K2	Gemini 2.5 Pro	GPT-4o	Model Ensemble	Success Rate	Termination Analysis	Verified Term. Rate
SyGuS (Alur et al., 2019)	93/133	101/133	104/133	131/133	105/133	132/133	99.30%	65/73/132	89.04%
OOPSLA-13 (Dillig et al., 2013)	27/46	22/46	26/46	41/46	23/46	41/46	89.13%	20/22/41	90.90%
Frama-C-Problems (Frama-C, 2026)	23/51	23/51	33/51	31/51	23/51	39/51	76.40%	37/37/39	100.00%
SV-COMP (Beyer, 2022)	6/21	2/21	4/21	18/21	4/21	18/21	85.71%	16/16/18	100.00%
X509 Parser (ANSSI-FR, 2026)	2/6	2/6	1/6	3/6	2/6	4/6	66.67%	4/4/4	100.00%

* Each cell reports the number of programs verified out of the total benchmark size. In Termination Analysis column, the format is Proven Termination / Intended Termination / Verified Partial Correctness. The difference between intended termination and verified partial correctness accounts for programs with inherent non-termination, as confirmed by manual inspection.

Pass@5: 91.05% versus 79.76% for the earlier AutoSpec.

Termination: 93.42% of verified, intended-terminating cases.

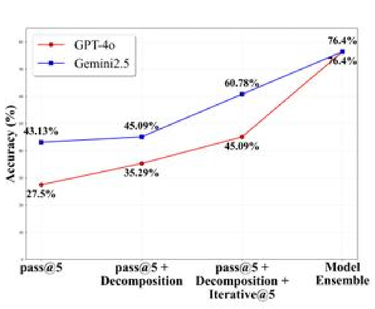


Figure 6: Ablation Study on *Frama-C-Problems* benchmark

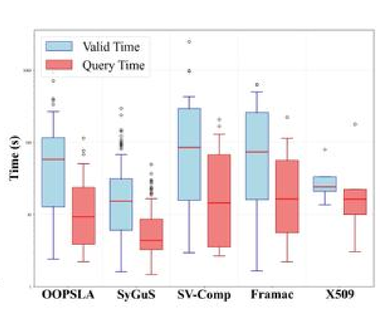


Figure 7: Time Cost Analysis (using *GPT-4o* as model backend)

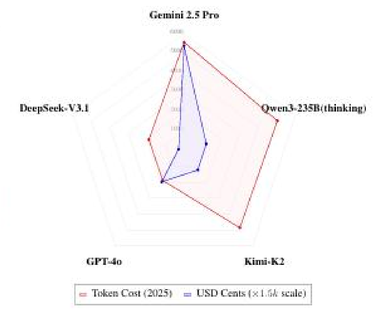


Figure 8: Token Cost and Expenditure Analysis

Takeaway: verifier-approved ACSL contracts enable compositional verification across functions and files.

Code: github.com/Xidian-ICTT-GZ/AutoSpec