

## A. Tables for literature

Table 2 presents a classification of the surveyed literature based on their methodological approach. Most works fall under the “Prompt-only” category, where LLMs are used without further tuning. Some studies involve human-in-the-loop or iterative prompting, while others fine-tune LLMs for improved performance. A set of works integrates LLMs with verifiers, and a few adopt neuro-symbolic methods combining LLMs with SMT solvers or theorem provers. Other categories include baseline/manual approaches using controlled natural language, meta-analyses and tool support studies, and a single work proposing IDE integration. To provide readers with a more comprehensive reference, we have also included Table 3 in the Appendix. This table integrates the developed tools, their methodological classifications, and a brief summary of each work, offering a clear overview of the purpose, approach, and contributions of each paper at a glance.

**Table 2**  
Classification of Surveyed Literature by Methodology

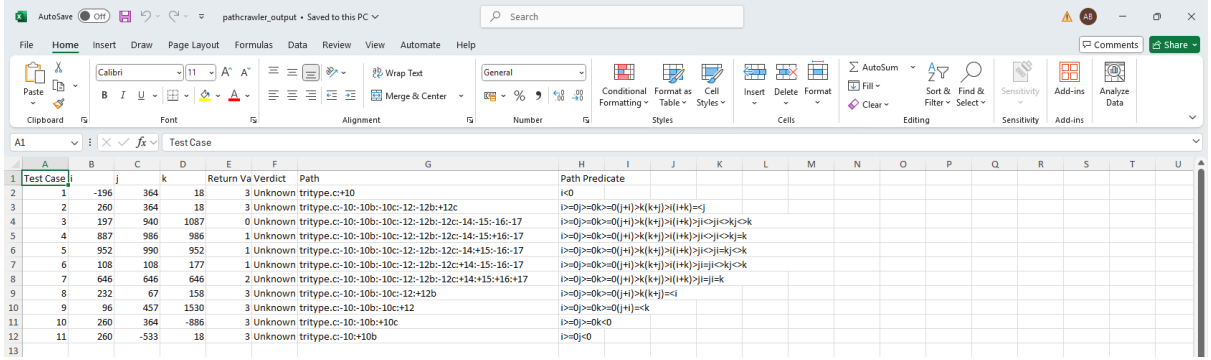
References	Classification
[42], [43], [5], [6], [44], [14], [16], [3], [27], [45], [46], [34], [33], [24], [30], [9], [5], [6], [41], [30], [27]	Prompt-only
[6], [5], [16], [33], [45]	Prompt + Iterative / Human-in-loop / CoT
[47], [18], [48], [17], [26], [16], [48]	Fine-tuned
[8], [9], [19], [49], [46], [30], [45]	Verifier-in-loop
[15], [10], [6]	Neuro-symbolic (LLM + SMT/Theorem Prover)
[13], [50], [18], [1], [13], [50]	Baseline / Manual / Controlled NL
[17], [51], [52], [11], [51]	Meta-analysis / Tool Support
[53]	IDE Integration Proposal

**Table 3**  
Classification and Description of Surveyed Literature

Ref.	Tool / Work	Classification	Description
[44]	Domain Model Extractor	Fine-tuned	Generates domain models from NL requirements; evaluated in an industrial case study for performance and accuracy.
[51]	Symbolic NLP vs. ChatGPT	Prompt-only	Compares symbolic NLP and ChatGPT in generating correct JML from NL preconditions.
[53]	BPM-to-NL Translation	Prompt-only	Translates business process models to NL to support better stakeholder validation.
[46]	Lemur	Verifier-in-loop	Integrates LLMs with automated reasoners and defines sound transition rules for verification.
[7]	GPT-4o for VeriFast Verification	Verifier-in-loop	Assesses GPT-4o's performance in generating C specs for VeriFast; captures issues in functional correctness.
[19]	ARSENAL	Fine-tuned	Extracts requirements from NL and performs automatic verification.
[25]	Laurel for Dafny	Prompt-only + Verifier-in-loop	Automates generation of Dafny assertions to support SMT-based verification.
[3]	PathCrawler + EVA	Verifier-in-loop + Prompt	PathCrawler generates context-aware ACSL annotations; EVA reduces runtime errors in Frama-C.
[54]	NL to Temporal Logic Translation	Prompt-only	Automatically translates NL into temporal logic for formal verification.
[10]	Req2Spec	Prompt-only	Converts NL requirements into formal specs (e.g., HANFOR); 71% accuracy on BOSCH data.
[12]	AssertLLM	Multi-LLMs / Prompt-only	Uses 3 customized LLMs to generate assertions from hardware design specs; 89% correctness achieved.
[9]	SpecSyn	Fine-tuned	Synthesizes software specifications from NL, improving over prior tools by 21%.
[8]	nl2spec	Prompt-only + Iterative Refinement	Iteratively generates formal specs from NL requirements, reducing ambiguity.
[55]	LLM-based Requirement Coverage	Prompt-only	Maps low-level requirements to high-level ones with 99.7% recall in coverage detection.
[13]	SpecLLM	Prompt-only	Uses LLMs to create and review VLSI design specs, enhancing chip documentation.
[16]	NL-to-LTL via LLMs	Prompt-only	Converts unstructured NL to NL-LTL pairs with 94.4% accuracy.
[56]	ANTONIO Toolkit	Verifier-in-loop	Introduces an NLP Verification Pipeline with metrics, gaps, and semantic subspace verification proposals.
[5]	GPT-3.5 for Code Verification	Prompt-only	Uses GPT-3.5 to verify code against requirements, providing feedback on requirement satisfaction.
[14]	GPT-4o + Claude for Smart Grid	Verifier-in-loop	Applies GPT-4o and Claude 3.5 for smart grid requirement verification, reaching 79–94% F1-scores.
[57]	Systematic Review	Meta-analysis	Surveys NL-to-specification literature across domains and academic sources.
[21]	SAT-LLM	Neuro-symbolic (LLM + SMT)	Combines LLMs with SMT to detect complex conflicts in requirements.
[23]	Thor	Neuro-symbolic	Integrates LLMs with theorem provers using class methods like Hammers for proof completion.
[40]	Dafny Task Gen w/ CoT	Prompt + Retrieval + CoT	GPT-4 and PaLM-2 generate verified Dafny tasks via retrieval-augmented CoT prompting.
[22]	LeanDojo + ReProver	Retrieval-augmented	Retrieval-augmented LLM-based prover improves Lean theorem proving on 98K+ samples.
[58]	SynVer for C	Verifier-in-loop	Synthesizes and verifies C programs using VST with improved automation.
[24]	Copilot + Formal Methods	IDE Integration Proposal	Suggests integrating formal tools (e.g., Dafny, Coq, KeY) into IDEs like Copilot.
[59]	Robotic Systems Review	Meta-analysis	Reviews 10 years of literature on formal verification in autonomous robotic systems.
[52]	NLP for Software Dev	Survey / Meta-analysis	Evaluates NLP techniques in software development life cycle.
[50]	RML (1986)	Manual / Controlled NL	Introduces controlled NL framework for precise and consistent requirements writing.
[49]	ESBMC-AI	Neuro-symbolic	Uses LLMs + formal verification to detect vulnerabilities in software.
[48]	Specification Consistency Framework	Manual / Baseline	Aligns oral and formal specifications using semantic reasoning and input-output analysis.
[60]	LLM Safety Req. Pipeline	Prompt-only	Uses LLMs to decompose and refine autonomous vehicle safety requirements.
[61]	NLP Tools for TFM	Prompt-only	Compares NLP pipelines for topological modeling; CoreNLP and FreeLing perform best.

## B. Work in progress and initial experiment setup

As initial experiment setup, we re-simulated the methodology presented in [3] by applying it to Tri-Type.c program, which was selected from the online interface of the PathCrawler tool available in the Frama-C ecosystem. The program was analysed using the original workflow – combining Large Language Models (LLMs) with symbolic outputs from PathCrawler and EVA – to generate ACSL specifications. The use of PathCrawler provided concrete path-based input/output examples that guided the LLM toward generating more context-relevant and semantically aligned annotations. Figure 1 shows the output of the Pathcrawler tool. Our plan is to conduct experiments with simple and complex programs, representing a diverse set of procedural constructs and control flows, allowing us to evaluate the methodology across a realistic and varied set of inputs. We have created a public repository on GitHub [https://github.com/arshadbeg/OVERLAY2025\\_SupportingDocs.git](https://github.com/arshadbeg/OVERLAY2025_SupportingDocs.git) to add resources.



Test Case	i	j	k	Return Va	Verdict	Path	Path Predicate
1	-196	364	18	3	Unknown	tritype.c:10	i < 0
2	260	364	18	3	Unknown	tritype.c:10-10b:10c:12-12b:12c	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i)
3	197	940	1087	0	Unknown	tritype.c:10-10b:10c:12-12b:12c:14-15:16:17	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)
4	887	986	986	1	Unknown	tritype.c:10-10b:10c:12-12b:12c:14-15:16:17	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)
5	952	990	952	1	Unknown	tritype.c:10-10b:10c:12-12b:12c:14-15:16:17	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)
6	108	108	177	1	Unknown	tritype.c:10-10b:10c:12-12b:12c:14-15:16:17	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)
7	646	646	646	2	Unknown	tritype.c:10-10b:10c:12-12b:12c:14-15:16:17	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)
8	232	67	158	3	Unknown	tritype.c:10-10b:10c:12-12b	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)
9	96	457	1530	3	Unknown	tritype.c:10-10b:10c:12	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)
10	260	364	-886	3	Unknown	tritype.c:10-10b:10c	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)
11	260	-533	18	3	Unknown	tritype.c:10-10b	(i > 0) & (k > 0) & (j > 0) & (i + j <= k) & (i + k <= j) & (j + k <= i) & (i < 0) & (j < 0) & (k < 0)

Figure 1: Output of the Pathcrawler for TriType Example

### Analysis of Tritype Example

**Note:** We used OpenAI’s ChatGPT-4o to assist in generating descriptive analysis text based on the output of the PathCrawler tool. All generated content was reviewed and verified by the authors. The function Tritype is:

```
/* Should return the type of the triangle
   which has sides of these lengths.
   3 = not a triangle
   2 = equilateral triangle
   1 = isosceles triangle
   0 = other triangle
*/
int Tritype(double i, double j, double k){
    int trityp = 0;
    if (i < 0.0 || j < 0.0 || k < 0.0) // line 10
        return 3;
    if (i + j <= k || j + k <= i || k + i <= j) // line 12
        return 3;
    if (i == j) trityp = trityp + 1; // line 14
    if (i == k) trityp = trityp + 1; // line 15
    if (j == k) trityp = trityp + 1; // line 16
    if (trityp >= 2) // line 17
        trityp = 2;
    return trityp;
}
```

PathCrawler successfully instrumented and analysed the Tritype function using the all-path coverage criterion. While 100% branch coverage was achieved, none of the 11 generated test cases were categorised

as successful or failed, leaving all as "unknown." This implies that the tool executed all branches in the code, yet could not conclusively determine the verdict for any specific path, possibly due to incomplete post-condition specifications or ambiguity in return value interpretation. The test harness was properly compiled, and the environment was initialised and cleaned up, suggesting that the tool functioned without interruptions. However, the presence of infeasible paths (8 of 19) emphasises the complexity of the control flow. The inability to evaluate paths conclusively indicates a need to refine input constraints or further investigate symbolic execution limitations. Ultimately, this dataset offers deep insights into path feasibility without final verdict certainty.

The test generation summary reveals a comprehensive attempt by PathCrawler to traverse 19 total paths within the `Tritype` function. Of these, 11 were covered and 8 marked as infeasible. The success of achieving 100% branch coverage suggests an effective instrumentation and exploration of all conditional branches in the function. However, none of the generated test cases resulted in a definitive "success" or "failure" outcome, indicating that path postconditions or functional assertions may not have been defined or interpreted robustly enough for classification. The discrepancy between path coverage and test verdicts signals potential limitations in either the underlying test oracle or symbolic constraint solving. Nevertheless, all test cases completed without runtime errors or crashes, which strengthens the confidence in static path feasibility. This situation is common in symbolic execution where certain paths are executable in theory but do not yield assertable functional correctness outcomes.

Several test cases (such as 3, 4, and 7) represent crucial boundary classifications in triangle identification logic. Test Case 3, returning 0, indicates that the input values do not form a valid triangle—corroborated by the path predicate involving non-equal sides and valid sum conditions. Test Cases 4 to 7 exhibit values returning 1 or 2, which correspond to isosceles and equilateral triangle scenarios. These variations affirm the tool's capability to explore distinct triangle classifications based on the input side lengths. However, all cases have an "unknown" verdict. This suggests that while execution paths were valid, postconditions or specification assertions did not yield sufficient evidence for a conclusive outcome. It is also possible that comparison thresholds or floating-point equality checks introduced uncertainty. The tool's symbolic reasoning did cover a wide input range, showcasing strength in data diversity.

This selection emphasises the diversity in input ranges—both positive and negative, and also includes floating-point values. Negative and invalid inputs, such as in Case 1, correctly trace to a return of 3, denoting an error or invalid triangle. Equilateral triangles like in Case 7 return 2, consistent with the expected logic. The presence of precise floating values (e.g., 886.99) highlights the need for robust type handling in symbolic execution and comparison logic. The return values indicate that, functionally, the outputs align with expectations, even if PathCrawler's analysis labels the verdict as unknown.

Examining the path predicates reveals detailed symbolic constraints applied during test generation. For example, Case 3 checks all triangle inequality constraints and non-equality of sides. Case 7's predicate confirms all sides are equal, ideal for verifying the equilateral branch of logic. These conditions provide assurance that symbolic execution adheres to logical expectations derived from triangle properties. However, predicates like those in Cases 1, 10, and 11 reflect invalid or boundary-breaking inputs—some involving negative sides. While such inputs yield predictable output values, their classification remains "unknown" due to a lack of explicit functional assertions. The predicates affirm that PathCrawler applies comprehensive constraint exploration but might benefit from clearer postcondition verification to move cases from "unknown" to "success" or "failure."

The analysis indicates high internal code coverage but limited external verdict classification due to insufficient assertions or postcondition specifications. While symbolic execution succeeded in evaluating feasible and infeasible paths and fully traversing conditional logic, the lack of verdicts suggests an

improvement opportunity in integrating output expectations with test oracles. PathCrawler’s utility in this context lies more in code path analysis and constraint checking than in final correctness validation. Future work may involve explicitly encoding expected behavior (e.g., via assertion annotations or return value comparisons) to convert "unknown" results into definitive outcomes. Despite these limitations, the tool has proven its capability in surfacing hidden paths and exercising diverse execution flows. Particularly in legacy or critical systems like `Tritype`, such analysis can prevent unanticipated bugs or logic flaws through exhaustive test path enumeration.

## Analysis of Verification on `baseline_Example1-Tritype.c`

The baseline prompt is available on page 20-21, [3]. The verification of `baseline_Example1-Tritype.c` using Frama-C and four SMT solvers (Alt-Ergo, Z3, CVC4, and CVC5) revealed a consistent pattern. With minimal ACSL specification, only two implicit verification goals—termination and unreachability—were generated and successfully verified by all four provers. This indicates that the function is syntactically well-formed and does not exhibit trivial issues like infinite loops or unreachable code paths. However, due to the absence of user-defined postconditions or preconditions, the analysis provides limited insight into functional correctness. The warning about the missing `assigns` clause suggests that memory side-effects are not specified, potentially causing inaccurate assumptions for callers. Similarly, the absence of RTE (Run-Time Error) guards indicates that common runtime errors like overflows or division by zero are not being verified. While the results demonstrate soundness at a structural level, the lack of deep specification significantly limits the utility of the verification. All provers perform equally under these trivial conditions.

## Comparison of Prover Results on `baseline_Example1-Tritype.c`

**Table 4**  
Prover Results for `baseline_Example1-Tritype.c`

Prover	Total Goals	Proved	Notes
Alt-Ergo	2	2	All default goals proved
Z3	2	2	All default goals proved
CVC4	2	2	All default goals proved
CVC5	2	2	All default goals proved

## Analysis of Verification on `pathcrawler_augmented_Example1-Tritype.c`

The pathcrawler augmented prompt is available on page 21-22, [3] while EVA augmented prompt is available on page 24, [3]. The `pathcrawler_augmented_Example1-Tritype.c` file, enriched with detailed ACSL annotations, exhibits significantly different verification behavior. A total of 20 goals were generated, including 18 based on user-specified preconditions and postconditions, plus the standard termination and unreachability checks. The analysis reveals a distinct divide in prover

effectiveness. While all provers verified termination and basic logic, their ability to handle complex ensures clauses varied. Z3 and CVC5 each failed to prove seven goals—Z3 due to timeouts and CVC4 due to unknown statuses—highlighting their struggles with intricate logical paths and case distinctions. Alt-Ergo and CVC5 fared slightly better, with only five unverified goals each. Notably, the most complex properties, such as correct classification of triangle types (Scalene, Isosceles, Equilateral) and handling of inequality rules, were consistently problematic across all provers. This reflects the challenges solvers face when dealing with disjunction-heavy logic or subtle arithmetic constraints embedded in functional specifications.

## Comparison of Prover Results on `pathcrawler_augmented_Example1-Tritype.c`

**Table 5**

Prover Results for `pathcrawler_augmented_Example1-Tritype.c`

Prover	Total Goals	Proved	Failed Type	Failed Count
Z3	20	13	Timeout	7
Alt-Ergo	20	15	Timeout	5
CVC4	20	13	Unknown	7
CVC5	20	15	Timeout	5

## C. Performance of provers on 36 Examples of Frama-C Tutorial

### RTE Tool Output Summary

C File	Goals	Proved	Qed	Timeout	Term.	Unreach.	Alt-Ergo	Assigns Missing
01-abs-0.c	1	2 / 3	0	1	1	1	0	Yes
01-abs-1.c	2	3 / 4	1	1	1	1	0	Yes
01-abs-2.c	4	6 / 6	4	0	1	1	0	No
01-abs-3.c	5	7 / 7	5	0	1	1	0	No
02-max-0.c	2	4 / 4	2	0	1	1	0	Yes
02-max-1.c	5	5 / 7	3	2	1	1	0	Yes
02-max-2.c	1	3 / 3	0	0	1	1	1	Yes
02-max-3.c	6	7 / 8	4	1	1	1	1	Yes
02-max-4.c	8	10 / 10	7	0	1	1	1	Yes
03-max_ptr-0.c	4	4 / 6	2	2	1	1	0	Yes
03-max_ptr-1.c	6	6 / 8	4	2	1	1	0	Yes
03-max_ptr-2.c	6	8 / 8	4	0	1	1	2	Yes
03-max_ptr-3.c	4	6 / 6	3	0	1	1	1	Yes
03-max_ptr-4.c	8	10 / 10	6	0	1	1	2	No
04-incr_a_by_b-0.c	5	3 / 7	0	4	1	1	1	Yes
04-incr_a_by_b-1.c	7	9 / 9	4	0	1	1	3	No
04-incr_a_by_b-fail.c	7	8 / 9	4	1	1	1	2	No
04-swap-0.c	6	4 / 8	2	4	1	1	0	Yes
04-swap-1.c	12	14 / 14	9	0	1	1	3	Yes
05-abs-0.c	1	2 / 3	0	1	1	1	0	Yes
05-abs-1.c	7	9 / 9	7	0	1	1	0	No
05-abs-2.c	6	8 / 8	6	0	1	1	0	No
06-max_abs-0.c	2	2 / 2	2	0	—	—	0	Yes
06-max_abs-1.c	13	11 / 13	9	2	—	—	2	Yes

C File	Goals	Proved	Qed	Timeout	Term.	Unreach.	Alt-Ergo	Assigns Missing
06-max_abs-2.c	13	12 / 13	11	1	–	–	1	Yes
06-max_abs-3.c	13	13 / 13	11	0	–	–	2	Yes
07-reset_array-0.c	3	2 / 4	1	2	–	1	0	Yes
07-reset_array-1.c	13	15 / 15	9	0	1	1	4	No
08-binary_search-1.c	27	29 / 29	13	0	1	1	14	No
09-sqrt-0.c	6	4 / 7	2	3	–	1	1	Yes

## Z3 Prover Output Summary

C File	Goals	Proved	Qed	Timeout	Term.	Unreach.	Z3	Assigns Missing
01-abs-0.c	0	2 / 2	1	1	0	0	0	Yes
01-abs-1.c	1	3 / 3	1	1	1	0	0	Yes
01-abs-2.c	3	5 / 5	1	1	3	0	0	No
01-abs-3.c	4	6 / 6	1	1	4	0	0	No
02-max-0.c	2	4 / 4	1	1	2	0	0	Yes
02-max-1.c	5	5 / 7	1	1	3	0	2	Yes
02-max-2.c	1	3 / 3	1	1	0	1	0	Yes
02-max-3.c	6	7 / 8	1	1	4	1	1	Yes
02-max-4.c	8	10 / 10	1	1	7	1	0	Yes
03-max_ptr-0.c	0	2 / 2	1	1	0	0	0	Yes
03-max_ptr-1.c	2	4 / 4	1	1	2	0	0	Yes
03-max_ptr-2.c	2	4 / 4	1	1	2	0	0	Yes
03-max_ptr-3.c	2	4 / 4	1	1	1	1	0	Yes
03-max_ptr-4.c	4	6 / 6	1	1	4	0	0	No
04-incr_a_by_b-0.c	0	2 / 2	1	1	0	0	0	Yes
04-incr_a_by_b-1.c	2	4 / 4	1	1	1	1	0	No
04-incr_a_by_b-fail.c	2	3 / 4	1	1	1	0	1	No
04-swap-0.c	2	4 / 4	1	1	2	0	0	Yes
04-swap-1.c	8	10 / 10	1	1	7	1	0	Yes
05-abs-0.c	0	2 / 2	1	1	0	0	0	Yes
05-abs-1.c	6	8 / 8	1	1	6	0	0	No
05-abs-2.c	5	7 / 7	1	1	5	0	0	No
06-max_abs-0.c	2	2 / 2	0	0	2	0	0	Yes
06-max_abs-1.c	13	11 / 13	0	0	9	2	2	No
06-max_abs-2.c	13	12 / 13	0	0	11	1	1	No
06-max_abs-3.c	13	13 / 13	0	0	11	2	0	No
07-reset_array-0.c	1	1 / 2	0	1	0	0	1	Yes
07-reset_array-1.c	11	13 / 13	1	1	8	3	0	No
08-binary_search-1.c	18	18 / 20	1	1	11	5	2	No

## CVC4 Prover Output Summary

C File	Goals	Proved	Qed	Timeout	Term.	Unreach.	CVC4	Assigns Missing
01-abs-0.c	0	2 / 2	1	1	0	0	0	Yes
01-abs-1.c	1	3 / 3	1	1	1	0	0	Yes
01-abs-2.c	3	5 / 5	1	1	3	0	0	No
01-abs-3.c	4	6 / 6	1	1	4	0	0	No
02-max-0.c	2	4 / 4	1	1	2	0	0	Yes
02-max-1.c	5	5 / 7	1	1	3	0	2	Yes
02-max-2.c	1	3 / 3	1	1	0	1	0	Yes
02-max-3.c	6	7 / 8	1	1	4	1	1	Yes
02-max-4.c	8	10 / 10	1	1	7	1	0	Yes
03-max_ptr-0.c	0	2 / 2	1	1	0	0	0	Yes

C File	Goals	Proved	Qed	Timeout	Term.	Unreach.	CVC4	Assigns Missing
03-max_ptr-1.c	2	4 / 4	1	1	2	0	0	Yes
03-max_ptr-2.c	2	4 / 4	1	1	2	0	0	Yes
03-max_ptr-3.c	2	4 / 4	1	1	1	1	0	Yes
03-max_ptr-4.c	4	6 / 6	1	1	4	0	0	No
04-incr_a_by_b-0.c	0	2 / 2	1	1	0	0	0	Yes
04-incr_a_by_b-1.c	2	4 / 4	1	1	1	1	0	No
04-incr_a_by_b-fail.c	2	3 / 4	1	1	1	0	1	No
04-swap-0.c	2	4 / 4	1	1	2	0	0	Yes
04-swap-1.c	8	10 / 10	1	1	7	1	0	Yes
05-abs-0.c	0	2 / 2	1	1	0	0	0	Yes
05-abs-1.c	6	8 / 8	1	1	6	0	0	No
05-abs-2.c	5	7 / 7	1	1	5	0	0	No
06-max_abs-0.c	2	2 / 2	0	0	2	0	0	Yes
06-max_abs-1.c	13	11 / 13	0	0	9	2	2	No
06-max_abs-2.c	13	12 / 13	0	0	11	1	1	No
06-max_abs-3.c	13	13 / 13	0	0	11	2	0	No
07-reset_array-0.c	1	1 / 2	0	1	0	0	1	Yes
07-reset_array-1.c	11	13 / 13	1	1	8	3	0	No
08-binary_search-1.c	18	16 / 20	1	1	11	3	4	No

## CVC5 Prover Output Summary

C File	Goals	Proved	Qed	Timeout	Term.	Unreach.	CVC5	Assigns Missing
01-abs-0.c	0	2 / 2	1	1	0	0	0	Yes
01-abs-1.c	1	3 / 3	1	1	1	0	0	Yes
01-abs-2.c	3	5 / 5	1	1	3	0	0	No
01-abs-3.c	4	6 / 6	1	1	4	0	0	No
02-max-0.c	2	4 / 4	1	1	2	0	0	Yes
02-max-1.c	5	5 / 7	1	1	3	0	2	Yes
02-max-2.c	1	3 / 3	1	1	0	1	0	Yes
02-max-3.c	6	7 / 8	1	1	4	1	1	Yes
02-max-4.c	8	10 / 10	1	1	7	1	0	Yes
03-max_ptr-0.c	0	2 / 2	1	1	0	0	0	Yes
03-max_ptr-1.c	2	4 / 4	1	1	2	0	0	Yes
03-max_ptr-2.c	2	4 / 4	1	1	2	0	0	Yes
03-max_ptr-3.c	2	4 / 4	1	1	1	1	0	Yes
03-max_ptr-4.c	4	6 / 6	1	1	4	0	0	No
04-incr_a_by_b-0.c	0	2 / 2	1	1	0	0	0	Yes
04-incr_a_by_b-1.c	2	4 / 4	1	1	1	1	0	No
04-incr_a_by_b-fail.c	2	3 / 4	1	1	1	0	1	No
04-swap-0.c	2	4 / 4	1	1	2	0	0	Yes
04-swap-1.c	8	10 / 10	1	1	7	1	0	Yes
05-abs-0.c	0	2 / 2	1	1	0	0	0	Yes
05-abs-1.c	6	8 / 8	1	1	6	0	0	No
05-abs-2.c	5	7 / 7	1	1	5	0	0	No
06-max_abs-0.c	2	2 / 2	0	0	2	0	0	Yes
06-max_abs-1.c	13	11 / 13	0	0	9	2	2	No
06-max_abs-2.c	13	12 / 13	0	0	11	1	1	No
06-max_abs-3.c	13	13 / 13	0	0	11	2	0	No
07-reset_array-0.c	1	1 / 2	0	1	0	0	1	Yes
07-reset_array-1.c	11	13 / 13	1	1	8	3	0	No
08-binary_search-1.c	18	16 / 20	1	1	11	3	4	No

## Execution Times



File Name	Alt-ergo Time(s)	Z3 Time(s)	CVC4 Time(s)	CVC5 Time(s)
01-abs-0.c	0.01	0.02	0.02	0.01
01-abs-1.c	0.02	0.04	0.02	0.001
01-abs-2.c	0.05	0.06	0.05	0.003
01-abs-3.c	0.04	0.03	0.03	0.004
02-max-0.c	0.02	0.02	0.01	0.002
02-max-1.c	0.07	0.06	0.05	0.004
02-max-2.c	0.08	0.07	0.05	0.002
02-max-3.c	0.12	0.14	0.13	0.004
02-max-4.c	0.05	0.06	0.05	0.007
03-max_ptr-0.c	0.02	0.03	0.02	0.001
03-max_ptr-1.c	0.03	0.04	0.03	0.002
03-max_ptr-2.c	0.06	0.06	0.06	0.002
03-max_ptr-3.c	0.09	0.09	0.07	0.003
03-max_ptr-4.c	0.07	0.07	0.08	0.004
04-incr_a_by_b-0.c	0.03	0.04	0.04	0.002
04-incr_a_by_b-1.c	0.02	0.03	0.01	0.003
04-incr_a_by_b-fail.c	0.03	0.04	0.04	0.002
04-swap-0.c	0.01	0.02	0.02	0.002
04-swap-1.c	0.04	0.05	0.03	0.004
05-abs-0.c	0.05	0.07	0.06	0.004
05-abs-1.c	0.07	0.08	0.07	0.006
05-abs-2.c	0.09	0.10	0.09	0.005
06-max_abs-0.c	0.03	0.04	0.03	0.002
06-max_abs-1.c	0.11	0.12	0.11	0.009
06-max_abs-2.c	0.14	0.15	0.14	0.011
06-max_abs-3.c	0.13	0.14	0.13	0.011
07-reset_array-0.c	0.02	0.03	0.02	0.002
07-reset_array-1.c	0.08	0.10	0.09	0.008
08-binary_search-1.c	0.25	0.28	0.30	0.064

Table 10: Execution times of Alt-Ergo, Z3, CVC4 and CVC5 on C files

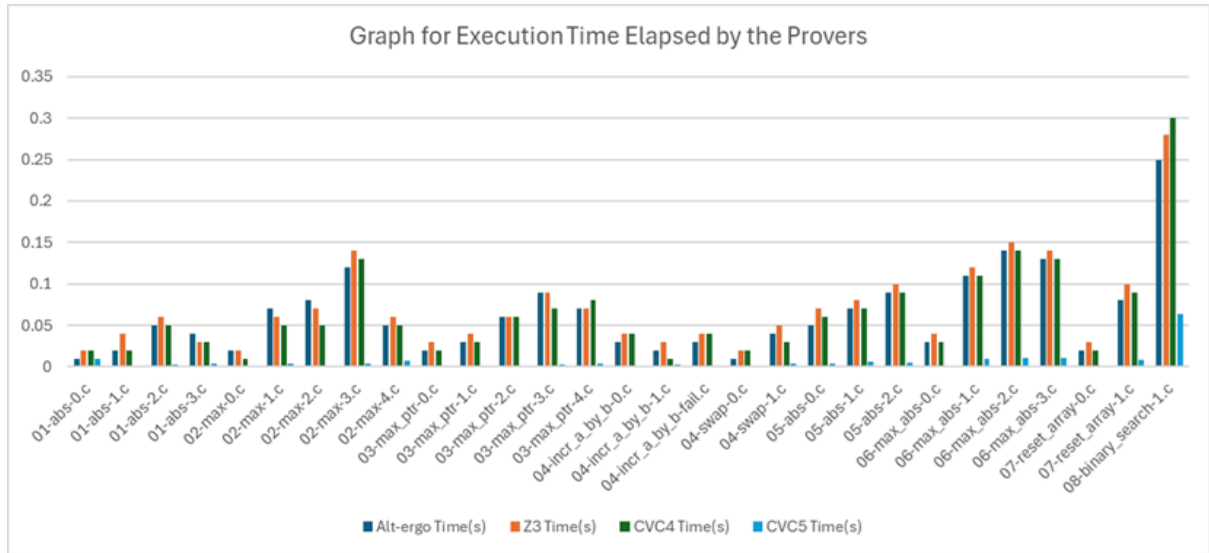


Figure 2: Graph of Execution Times of Alt-Ergo, Z3, CVC4 and CVC5

## D. Description of Challenges and Future Directions

**C1: Semantic Ambiguity in Natural Language** - The ambiguity introduced in software requirements through the use of natural language is evident. The context-dependent terms, implicit references, and domain-specific jargon complicate accurate interpretation of requirements for automatic specification generation. The use of LLMs helps address this challenge but mis-interpretation of user intent still leads to incorrect outputs. Removing ambiguity requires structured domain knowledge aligned with external ontologies. As the literature synthesis suggests, improved models and effective human-in-the-loop strategies increase the accuracy in generating correct and verifiable specifications.

**C2: Lack of Ground Truth Datasets** - public datasets involving industry based software is difficult to locate, arguably due to the sensitivity of software pieces and underlying IP rights. This scarcity of standard datasets comprising of natural language requirements together with formal logic poses an obstacle in the much required steps of model training, evaluation, and reproducibility (core components of achieving better accuracy in mainstream NLP tasks). It also involves a scalability challenge. Advancing the field requires high-quality, and annotated datasets that span varied industries and requirement types. The synthesised datasets partially solve the problem. The availability of high-quality, curated and annotated standardised formal specification datasets involving varied industries and requirement types would push the boundary in this domain. These datasets should capture real-world software complexity and domain diversity.

**C3: Tool Interoperability** - There is a significant interest in developing tool-chains to integrate formal verification tools for verifying critical software from the last two decades. Formal verification process rely on diverse tools with incompatible formats and limited integration capabilities, which makes end-to-end automation difficult. The process of integrating formal verification tools involves manual or semi-automatic intervention. The process becomes difficult in the absence of shared standards or modular pipelines. Achieving seamless and low-effort tool interoperability requires standardised interfaces and ontologies.

**C4: Traceability Across Artefact Lifecycles** - A typical software development life-cycle involves multiple phases. Maintaining traceability and alignment throughout the development cycle is the core requirement. Achieving traceability manually can compromise system integrity. Effective traceability links rationale, changes and dependencies across software artefacts i.e. text, models, code, and tests. LLMs can assist in impact analysis to maintain traceability, but imposes a challenge of collaborative, explainable, and context-aware evolution process.

**C5: Explainability and User Trust** - For LLM-generated formal specifications to be trusted, users must understand how outputs were derived and whether they reflect intended meaning. Current models offer limited transparency, often lacking rationale or input attribution. Addressing this involves adding annotations in the implementation and can potentially be resolved through use of LLMs. But LLM generated annotations can be under-developed or involve quality check using a human-in-the-loop, especially ensuring trust in safety-critical systems and impose the challenge of deploying inspection, refinement, and human oversight.

**F1: Human-in-the-loop Formalisation** - In this approach, the formalisation is semi-automated and guided by a domain expert. LLMs can assist in proposing logic, trace links, or suggest refinements which can be approved or revised by the users. The process ensures less ambiguity, improved accuracy and increased user trust. It enhances learning as well because improvement in model outputs is ensured through interaction. This process can be facilitated by better visualisation, iteration, and feedback support. This approach facilitates adopting new methodologies as human judgment remains central to critical

decisions.

**F2: Multi-modal Artefact Alignment** - Software requirements can be documented through a combination of raised through text, diagrams, tables, and spreadsheets, depending on the methodology adopted by the developers. Alignment between these can reduce ambiguity and lead to better formal specifications. Multi-modal support in the tools should be available and can be achieved through representation learning and semantic matching. LLMs having structural and visual modalities can improve context interpretation, resulting in more comprehensive models. Formalisation pipelines and redundant artefact types can model real-world complexity, impacting the correctness and reliability of generated specifications.

**F3: Standardised Benchmarks** - The future direction of developing standardised benchmark has one-to-one mapping to identified challenge C2. Having standardised benchmarks to simulate and compare performance is a well-established research area [refereces]. Advancing the field requires high-quality, and annotated datasets that span varied industries and requirement types. Though the synthesised datasets have partially solved the problem, the availability of high-quality and annotated standardised formal specifications datasets remains limited. However, such datasets involving varied industries and requirement types can push the boundary in this domain. These datasets should capture real-world software complexity and domain diversity. As the field matures, such resources will become an essential component for progress.

**F4: Neuro-symbolic Reasoning** - Neuro-symbolic systems combine the adaptability of LLMs with the precision of symbolic reasoning. In these setups, neural networks suggest possible specifications While logic-based tools check or refine them. This helps improve consistency, enforce constraints, and verify correctness. Symbolic features like type rules or domain-specific logic can also guide the learning process. While building such systems is complex, the hybrid method offers the potential of lowering hallucinations and boosting clarity. Ongoing research could lead to reliable, interpretable models that merge statistical learning with formal methods.

**F5: Interactive Traceability Tools** - Improving traceability of software requirements through interactive user-friendly tools is a well-established area of requirements engineering. Traceability needs to be built into the formalisation process from the start and interactive tools can link requirements to models, tests, and verification results. This supports debugging, system updates, and audits. Features like visual navigation, filtering, and tagging help teams track changes and spot dependencies. LLMs can help by proposing trace links, flagging inconsistencies, or explaining revisions but users must stay in control of key decisions. These tools should work within development environments and support team collaboration. When traceability is clear and usable, it boosts system transparency and helps ensure compliance—especially in regulated fields like aerospace and healthcare. Traceability tools like EARS, FRET and Doors are industry-proven. Industry-based case studies are a valuable avenue for future work in this field.