LLM-based Generation of Weakest Preconditions and Precise Array Invariants

Daragh King
School of Computer Science and Statistics
Trinity College Dublin
Dublin, Ireland
kingd6@tcd.ie

Vasileios Koutavas
Lero, Research Ireland Centre for Software
School of Computer Science and Statistics
Trinity College Dublin
Dublin, Ireland
Vasileios.Koutavas@tcd.ie

Laura Kovács
Faculty of Informatics
TU Wien
Vienna, Austria
laura.kovacs@tuwien.ac.at

***Abstract*—The weakest precondition (WP) of a program describes the largest set of initial states from which all terminating executions of the program satisfy a given postcondition. The generation of WPs is an important task with practical applications in areas such as software verification and runtime error-checking. For programs containing loops, WP generation critically depends upon synthesizing loop invariants that are inductive in nature; these WPs then essentially embed the inductive properties required for program verification. This paper investigates the use of Large Language Models (LLMs) to generate WPs (and accompanying loop invariants) in order to prove the correctness of non-deterministic programs containing arrays, loops, and arithmetic. Specifically, we employ several models of ChatGPT to derive the WPs and invariants for the aforementioned programs. We then compare these LLM-derived WPs and invariants to their provable counterparts obtained by the MaxPrANQ tool. We find that the quality of the LLM-derived results can vary greatly and is highly dependent on the underlying model used by ChatGPT. This variance in performance propels us to outline directions for future work and discuss how LLMs and formal-tools can complement one another in generating valid WPs and strong invariants.**

***Index Terms*—LLMs, weakest, preconditions, invariants, precise, arrays, non-deterministic, correctness, verification**

I. Introduction

The weakest precondition (WP) of a program defines the input space from which all terminating program executions achieve the desired postcondition. In other words, when the initial state of a program adheres to the WP, it is guaranteed that the terminating program will behave correctly – at least according to its given postcondition. The utility of such a precondition is evident; WPs can find application in runtime error-checking [1], verification [2] and compositional symbolic execution [3].

The generation of WPs presents a significant technical challenge, especially when the programs being considered contain unbounded loops or arrays. These program features typically impose the additional baggage of reasoning about inductive properties like loop invariants [4], [5]. Though this task is difficult (and in general, undecidable), there still exists a rich variety of methods tackling this issue [6]–[9].

In this paper, we begin to explore a new frontier for the generation of WPs. Thus far, LLMs have displayed the ability to generate code [10], repair code [11] and infer program invariants [12], therefore, it stands to reason that they may be effective at generating WPs. In pursuit of this idea, we evaluate the effectiveness of several ChatGPT models (o1-preview [13], GPT-4o [14], and GPT-4 [15]) in generating the WPs and inductive invariants for a set of benchmark programs from [6]. These programs feature several aspects that make reasoning about their behaviour difficult – namely, non-determinism, loops, and arrays.

The MaxPrANQ tool [6] for WP inference features prominently in our analysis. The outputs of this provide the 'ground-truth' WPs and invariants for our benchmark programs. These are compared to their LLM-derived counterparts from our work, acting as a measure by which to judge the effectiveness of LLMs in generating WPs and accompanying invariants.

Following the above evaluation and a discussion of the results obtained, we then consider potential avenues for future work. In particular, we highlight how LLMs and the Vampire theorem-prover [4] could be integrated to effectually generate valid WPs and strong invariants. We also propose a high-level framework to guide future research in this setting.

Section II details how ChatGPT is used to generate WPs and invariants. Section III discusses our evaluation of ChatGPT in the aforementioned task. Section IV explores future research paths. Section V gives an overview of related works. Finally, Section VI provides concluding remarks.

II. Method

This section gives an overview of our experimental process.

A. Benchmark Programs

The benchmark programs within this study are borrowed from the work of [6]. These programs are challenging benchmarks for the software verification community as they each

This work was conducted with the financial support of the Research Ireland Centre for Research Training in Digitally-Enhanced Reality (d-real) under Grant No. 18/CRT/6224, Taighde Éireann – Research Ireland under Grant number 13/RC/2094_2, and the ERC Consolidator Grant ARTIST 101002685.

```c
void foo(int N, int a[N], int b[N], int c[N], int j)
{
  // Weakest Precondition
  int i = 0;
  if (N <= 0) goto end;
  // Loop Invariant:
  while (i < N) {
    int nd;
    if (nd) { b[i] = a[i] - j; }
    else { c[i] = a[i] - j; }
    i++;
  }
  i = 0;
  while (i < N) {
    if (b[i] < 0 || c[i] < 0) goto ERROR_1;
    i++;
  }
  goto end;
 ERROR_1:;
 end:;
   return;
}
```

Fig. 1. Original benchmark example: array_min_and_copy_shift.

contain features that make reasoning about their behaviour difficult. In concrete terms, each of these programs contains loops, arrays, arithmetic, and non-determinism. An example program can be seen in Fig. 1 (the code comments relating to the WPs and loop invariants were added for use when prompting ChatGPT – references to these comments can be seen in the prompt-text of Fig. 2). This example is representative of the benchmark programs overall, any differences generally arise in how the arrays are updated in the first *while* loop and how the postcondition is specified in the second *while* loop.

B. GPT Models

To increase the scope of our experiments in this setting, we chose to evaluate three GPT models in the generation of WPs and accompanying invariants. The models used are o1-preview [13], GPT-4o [14], and GPT-4 [15]. ChatGPT and the aforementioned models were chosen for their general ease-of-use and performance.

O1-preview is of particular interest here. The creators (OpenAI) claim: '[O1 models] can reason through complex tasks and solve harder problems than previous models in science, coding, and math.' [13]. Works such as [16] give some weight to this claim, particularly when the o1 models are compared to open-source LLMs.

C. Prompt

The prompt used to query ChatGPT for a program's WPs and accompanying invariants can be seen in Fig. 2, this prompt-text is then followed by the program being considered (an example program is omitted here for the sake of brevity). The prompt outlined was found to provide the most reliable results across the GPT models considered.

D. Method Outline

The high-level outline of our experiments is as follows:

```
Compute the weakest precondition
for the following C code. Return the
program with the weakest precondition
included immediately after line
'// Weakest Precondition', ensure
that valid C syntax and assertion
statements are used. Include the
inductive loop invariant on the line
'// Loop Invariant: ' only (do not
include it in the loop body).
```

Fig. 2. The ChatGPT prompt used for generating weakest preconditions and accompanying invariants.

1) Select a benchmark program B and append it to the prompt-text from Fig. 2 – this is the full prompt p.
2) Query ChatGPT using p, parsing the WP and accompanying invariants from the answer provided.
3) Compute B's true WP and accompanying invariants using MaxPrANQ.
4) Compare the ChatGPT-derived values (from Step 2) to their MaxPrANQ-derived counter-parts (from Step 3).

The above process is repeated for each benchmark program and GPT model.

III. Results

We evaluated three different GPT models in the generation of weakest preconditions (WPs) and accompanying invariants. This was done for a set of non-deterministic, array-manipulating programs containing loops and arithmetic. The following section describes our results (for additional details, see [17]).

A. Correctness Conditions

Before delving into the obtained results, we must first outline the conditions used to determine the correctness of an LLM-derived WP or invariant. In this setting, we deem an LLM-derived WP or invariant to be correct if:

1) It is semantically equivalent to the counterpart derived by the MaxPrANQ tool [6].
2) It is obtained through only one prompting of the relevant GPT model.

Regarding the first condition, this comparison was carried out 'by hand' for the current work, in future work this process could be automated using Vampire [4]. Additionally, the zero-shot prompting of the second condition was deemed appropriate for the goals of this current study, we believe it to be representative of how an LLM would be first queried for the tasks tackled here.

B. Weakest Precondition Generation

Table I summarises the results of using ChatGPT for generating WPs. The (✔)-mark in Table I indicates that the correctness conditions described in Section III-A have been achieved for a particular GPT model and benchmark program, whilst a (✘)-mark indicates the contrary.

Observing Table I, it is clear that o1-preview [13] achieved the best results; o1-preview generated the correct WPs for all

TABLE I
LLM-DERIVED WEAKEST PRECONDITION CORRECTNESS, WITH CORRECTNESS MEASURES AS DESCRIBED IN SECTION III-A.

Program Name	Correct Weakest Preconditions		
	O1-Preview	*GPT-4o*	*GPT-4*
array_init_addvar.c	✔	✘	✔
array_init_and_copy_const.c	✔	✔	✔
array_init_pair_symmetr.c	✔	✔	✘
nondet-array-bench8.c	✔	✘	✘
nondet-array-bench20.c	✔	✔	✔
array_init_increm_two_arrs_antisym.c	✔	✘	✘
array_init_increm_twice.c	✔	✔	✘
array_init_increm.c	✔	✘	✘
array_min_and_copy_shift.c	✔	✘	✘
nondet-array-bench15.c	✔	✘	✔
Total Correct	10/10	4/10	4/10

benchmark programs. GPT-4o [14] and GPT-4 [15] performed significantly worse; both only managed to generate the correct WPs for $4/10$ benchmark programs. This divergence in performance is to be expected; as alluded to in [13], o1-preview has been specifically tailored to perform well in code-reasoning settings such as this.

Although expected, the above contrast in performance is still somewhat startling – begging the question as to how exactly these 'reasoning' improvements are built into OpenAI's new o1 models. Unfortunately, due to OpenAI's closed nature and the recent advent of these models, this answer may not be forthcoming. This disparity in performance (when considered with the restricted prompt-limits of the o1 models [13]) helps motivate our future research directions in Section IV.

C. Invariant Generation

For the benchmark programs studied, each GPT model also attempted to generate valid inductive invariants. Although each GPT model produced *some* invariant, we observed a divergence in whether these generated invariants were *sufficient* for program correctness (recall that 'true' is a trivially valid invariant for all loops). In fact, in every case where a GPT model failed to produce the correct WP, it also failed to produce a loop invariant strong enough for program correctness.

These results complement the works of [18]–[21] by further lending support to the notion that LLMs can generate program invariants. Furthermore, we have observed that LLMs can generate strong loop invariants over arrays – which, to our knowledge, has not been demonstrated before comprehensively.

D. Correcting GPT Outputs

Considering the WP results in Table I, we sought to perform some initial investigations to observe whether the failing GPT scenarios could be corrected using additional prompting. We utilised singular additional prompts that followed the general form of: 'This weakest precondition is incorrect. Consider the case when ...'. The case(s) used within the aforementioned correction-prompt were derived manually for this work.

Using correction-prompts in this manner, GPT-4o was able to attain the correct WP for $1/6$ of its failing benchmarks;

```
void foo(int N, int a[N], int b[N], int c[N], int j)
{
  // Weakest Precondition
  assert(N > 0);
  for (int k = 0; k < N; k++) {
    assert(a[k] - j >= 0);
    assert(b[k] >= 0);
    assert(c[k] >= 0);
  }
  int i = 0;
  if (N <= 0) goto end;
  // Loop Invariant: For all indices k from 0 to
  ↪  i-1, b[k] >= 0 and c[k] >= 0
  while (i < N) {
    int nd;
    if (nd) { b[i] = a[i] - j; }
    else { c[i] = a[i] - j; }
    i++;
  }
  i = 0;
  while (i < N) {
    if (b[i] < 0 || c[i] < 0) goto ERROR_1;
    i++;
  }
  goto end;
ERROR_1:;
end:;
  return;
}
```

Fig. 3. Altered benchmark example: array_min_and_copy_shift. This includes the weakest precondition and loop invariant generated by o1-preview.

GPT-4 managed to perform slightly better, managing to derive the correct WPs for $4/6$ of its failing benchmarks. It is worth noting that this manner of correction-prompting is a delicate matter in its own right; further investigation into how this is best performed and standardised is a worthy line of future work.

E. Can O1-Preview Fail?

The results of Table I might seem to naively suggest that o1-preview is an infallible generator of WPs. In this experimental setup, and for these particular benchmarks, this certainly seems to be the case. In order to confirm (or dispel) this notion, we sought to examine a number of programs beyond the benchmark set described in Section II-A. This led us to cases where o1-preview failed to generate correct WPs (for details, see [17]) – the preconditions generated were found to be valid but not weakest. It should be noted there is no MaxPrANQ-derived ground-truth for these cases (as they lie outside the benchmark set of Section II-A), therefore the true WPs were derived manually for comparison.

IV. FUTURE RESEARCH DIRECTIONS

As evidenced by our results in the previous section, using LLMs to generate correct weakest preconditions (WPs) is certainly not a panacea. We ultimately found that GPT models not specialised for reasoning tasks (i.e. not o1-preview [13]) could perform quite poorly in generating WPs. Although o1-preview performed well for the benchmark programs, we would argue that its closed nature, restrictive prompt-limits [13], and cost make it unsuitable as a general method for generating

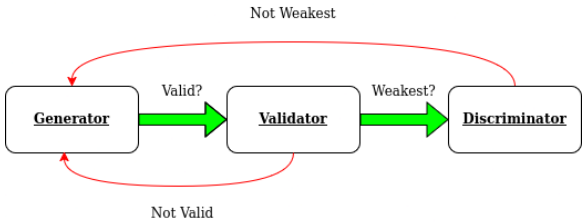

Fig. 4. *GVD* framework for WP generation.

WPs. Additionally, LLM-derived WPs or invariants cannot be soundly relied upon without the backing of a formal-tool's analysis – it should be noted that the outputs of a formal-tool (i.e. MaxPrANQ [6]) were central to this study.

With these considerations in mind, we want to advance the following hypothesis for guiding future research: *LLMs and formal software analysis tools can be integrated in a mutually beneficial fashion to generate valid WPs and strong invariants*.

This paper acts as the first piece of evidence for the proposed hypothesis. An actionable step beyond this would be to employ the Vampire theorem-prover [4] as a means to validate LLM-derived WPs. In addition, and as mentioned in Section III-A, another line of work would be to standardise and automate how correction-prompts are created for guiding LLMs towards generating valid WPs and accompanying invariants.

These piece-meal research directions are all well and good, but a prudent future-facing approach would be to specify a potential framework that these work-pieces could fall into. The following *GVD* framework is our attempt at this.

The *GVD* (Generate-Validate-Discriminate) framework is a high-level description of co-operating components for the generation of valid WPs, it is motivated by a similar work-flow (for invariant generation) described in [19]. The *GVD* consists of three parts:

- *Generator*: proposes candidate WPs.
- *Validator*: ensures that the candidate WP does not include initial states that would violate the programs given postcondition.
- *Discriminator*: attempts to provide evidence that the candidate WP is (or is not) truly weakest.

In pursuit of our hypothesis we envision that the *Validator* and *Discriminator* would be formal-tools like Vampire [4] or MaxPrANQ, whilst the *Generator* would consist of an LLM.

Although we harbour biases towards what tools might fit certain components, the description of the *GVD* parts above is purposefully generic. This generic-ness ensures that *GVD* is not tied to any particular tools or software, thus encouraging experimentation. For example, if a likelihood of weakness suffices for a particular candidate precondition, then some form of fuzz-testing [22] could be used as a *Discriminator*.

V. Related Work

Recent works employ LLMs in conjunction with formal-tools to generate invariants and reason about program behaviour. The authors of [12] use LLMs to generate loop invariants, these are then validated by the Vampire theorem-prover [4] and used to replace loops within bounded model checking [18]. The work of [21] introduces the *Lemur* framework. Here, LLMs are used to generate and repair candidate invariants, these are then used with ESBMC [23] and UAtomizer [24] in an attempt to complete program verification. In addition, the authors of [19] perform an exploratory analysis into using LLMs to generate loop invariants, Frama-C [25] acts as a means of validation in this instance.

Other works use LLMs and neighbouring machine-learning methods to generate additional program properties. The authors of [20] fine-tune LLMs in an attempt to infer invariants for Java programs (such that these invariants are likely to be Daikon [8] generated). In a task adjacent to invariant generation, the authors of [26] use LLMs to generate postconditions from natural language documentation. Additionally, [27] does not use LLMs, but instead opts to use deep-reinforcement-learning to synthesize program invariants.

The task of generating preconditions has been approached in both static and dynamic manners. Recently on the static end of the spectrum, the authors of [6] and [7] utilise methods of logical abductive inference to generate the provably weakest preconditions for a class of deterministic and non-deterministic, array-manipulating programs. In the dynamic realm, works such as [9] and [28] employ data-driven program execution loops in order to learn likely preconditions.

VI. Conclusions

In this paper we have outlined our initial investigations into the efficacy of using LLMs for generating weakest preconditions (WPs) and accompanying invariants for non-deterministic, array-manipulating programs featuring loops.

We found that more advanced versions of ChatGPT fared well in this setting. In particular, o1-preview [13] generated the correct WPs and inductive loop invariants for the benchmark programs considered – these results were verified using the MaxPrANQ WP inference tool [6]. Although o1-preview performed impressively here, we highlighted instances where o1-preview failed in generating correct WPs.

Other GPT models, GPT-4o [14] and GPT-4 [15], did not perform as well as o1-preview. In a majority of studied cases, both models failed to generate the correct WPs and sufficient invariants for program correctness. We also found evidence that these GPT-versions appeared to benefit from additional guidance in the form of correction-prompts.

The aforementioned correction-guidance phenomenon, in conjunction with the practical limits of o1-preview [13], led us to discuss how LLMs could stand to complement formal program-reasoning tools (and vice versa). We then considered a number of future research directions and introduced the *GVD* (Generate-Validate-Discriminate) framework to guide this work. Additionally, we highlighted how the Vampire theorem-prover [4] could fit into this framework, and how it could be be used alongside LLMs to generate valid WPs and strong invariants.

References

[1] W. Weimer and G. C. Necula, "Finding and preventing run-time error handling mistakes," in *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, 2004, pp. 419–431.

[2] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The journal of logic and algebraic programming*, vol. 78, no. 5, pp. 293–303, 2009.

[3] J. Fragoso Santos, P. Maksimović, G. Sampaio, and P. Gardner, "Javert 2.0: Compositional symbolic execution for javascript," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–31, 2019.

[4] L. Kovács and A. Voronkov, "First-order theorem proving and vampire," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 1–35.

[5] C. A. Furia, B. Meyer, and S. Velder, "Loop invariants: Analysis, classification, and examples," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, pp. 1–51, 2014.

[6] S. Sumanth Prabhu, D. D'Souza, S. Chakraborty, R. Venkatesh, and G. Fedyukovich, "Weakest precondition inference for non-deterministic linear array programs," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2024, pp. 175–195.

[7] S. Sumanth Prabhu, G. Fedyukovich, and D. D'Souza, "Maximal quantified precondition synthesis for linear array loops," in *European Symposium on Programming*. Springer, 2024, pp. 245–274.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of computer programming*, vol. 69, no. 1-3, pp. 35–45, 2007.

[9] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 42–56, 2016.

[10] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests," *arXiv preprint arXiv:2207.10397*, 2022.

[11] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of code language models on automated program repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1430–1442.

[12] M. A. Pirzada, G. Reger, A. Bhayat, and L. C. Cordeiro, "Llm-generated invariants for bounded model checking without loop unrolling," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1395–1407.

[13] OpenAI, "Introducing OpenAI o1-preview," 2024. [Online]. Available: https://openai.com/index/introducing-openai-o1-preview/

[14] ——, "Hello GPT-4o," 2024. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[15] ——, "GPT-4," 2024. [Online]. Available: https://openai.com/index/gpt-4-research/

[16] I. Mirzadeh, K. Alizadeh, H. Shahrokhi, O. Tuzel, S. Bengio, and M. Farajtabar, "Gsm-symbolic: Understanding the limitations of mathematical reasoning in large language models," *arXiv preprint arXiv:2410.05229*, 2024.

[17] D. King, "Llm-wp-generation: Formalise-camera-ready (experimental results)," Jan. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.14761572

[18] A. Biere, "Bounded model checking," in *Handbook of satisfiability*. IOS press, 2021, pp. 739–764.

[19] C. Janßen, C. Richter, and H. Wehrheim, "Can chatgpt support software verification?" in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2024, pp. 266–279.

[20] K. Pei, D. Bieber, K. Shi, C. Sutton, and P. Yin, "Can large language models reason about program invariants?" in *International Conference on Machine Learning*. PMLR, 2023, pp. 27 496–27 520.

[21] H. Wu, C. Barrett, and N. Narodytska, "Lemur: Integrating large language models in automated program verification," *arXiv preprint arXiv:2310.04870*, 2023.

[22] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[23] R. S. Menezes, M. Aldughaim, B. Farias, X. Li, E. Manino, F. Shmarov, K. Song, F. Brauße, M. R. Gadelha, N. Tihanyi *et al.*, "Esbmc v7. 4: Harnessing the power of intervals: (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2024, pp. 376–380.

[24] M. Heizmann, D. Dietsch, M. Greitschus, J. Leike, B. Musa, C. Schätzle, and A. Podelski, "Ultimate automizer with two-track proofs: (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2016, pp. 950–953.

[25] D. Beyer and M. Spiessl, "The static analyzer frama-c in sv-comp (competition contribution)," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2022, pp. 429–434.

[26] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri, "Can large language models transform natural language intent into formal method postconditions?" *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1889–1912, 2024.

[27] X. Si, A. Naik, H. Dai, M. Naik, and L. Song, "Code2inv: A deep learning framework for program verification," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II 32*. Springer, 2020, pp. 151–164.

[28] S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta, "Dynamic inference of likely data preconditions over predicates by tree learning," in *Proceedings of the 2008 international symposium on Software testing and analysis*, 2008, pp. 295–306.