Can ChatGPT support software verification?

Christian Janßen, Cedric Richter(✉), and Heike Wehrheim

Carl-von-Ossietzky Universität Oldenburg, Oldenburg, Germany
{christian.janssen1, cedric.richter, heike.wehrheim}@uol.de

Abstract. Large language models have become increasingly effective in software engineering tasks such as code generation, debugging and repair. Language models like ChatGPT can not only *generate* code, but also *explain* its inner workings and in particular its correctness. This raises the question whether we can utilize ChatGPT to support *formal software verification*.
In this paper, we take some first steps towards answering this question. More specifically, we investigate whether ChatGPT can generate *loop invariants*. Loop invariant generation is a core task in software verification, and the generation of *valid* and *useful* invariants would likely help formal verifiers. To provide some first evidence on this hypothesis, we ask ChatGPT to annotate 106 C programs with loop invariants. We check validity and usefulness of the generated invariants by passing them to two verifiers, Frama-C and CPAchecker. Our evaluation shows that ChatGPT is able to produce valid and useful invariants allowing Frama-C to verify tasks that it could not solve before. Based on our initial insights, we propose ways of combining ChatGPT (or large language models in general) and software verifiers, and discuss current limitations and open issues.

Keywords: Large language models · Invariant generation · Formal verification.

1 Introduction

Large language models (LLMs) [11,37,30] are increasingly employed to support software engineers in the generation, testing and repair of code [15,14,27]. Generative AI can, however, not only generate code, but also provide explanations of the inner workings of code and give arguments about its correctness. This raises the question whether LLMs can also support *formal software verification*.

In this paper, we provide a first step towards answering this question. In general, one can imagine various ways of supporting verifiers, depending on the verification approach they employ. Central to all verifiers are, however, techniques for dealing with loops. Specifically, for abstracting the behaviour of loops, verifiers aim at computing *loop invariants*. Our first step in evaluating ChatGPT's usefulness for software verification is thus the generation of loop invariants.

To this end, we ask ChatGPT to annotate C-programs with loop invariants. We have chosen 106 C-programs from the Loops category of the annual competition on software verification [7]. To enable the usage of these invariants by

D. Beyer and A. Cavalcanti (Eds.): FASE 2024, LNCS 14573, pp. 266–279, 2024.
https://doi.org/10.1007/978-3-031-57259-3_13

Prompt> Compute a loop invariant for the following program!

```
void func(unsigned int n)
{
  unsigned int x=n, y=0;
  //@ loop invariant [mask];
  while(x>0) {
    x--; y++;
  }
  assert(y==n);
}
```

Infilling provided by ChatGPT: `x+y==n`

Fig. 1. Example task: `loops/count_up_down-1`.

verifiers, we needed the invariants to be given in some formal language. For this, we have chosen ANSI/ISO C Specification Language (ACSL) [5], a design-by-contract like annotation language for C. Initial experiments confirmed that ChatGPT "knows" ACSL. The main part of our experiments then concerned the evaluation of the invariants with respect to (a) *validity* and (b) *usefulness* for verifiers. The first aspect required checking whether a proposed invariant is actually a proper invariant, i.e., whether the computed predicate holds at the beginning of the loop and after every loop iteration. We employ the state-of-the-art interactive verifier Frama-C [4] for this validity checking. For evaluating the usefulness of invariants, we provided two state-of-the-art verifiers (Frama-C SV [9] and CPAchecker [8]) with the code annotated by the proposed invariant, and evaluated whether the verifiers can then solve verification tasks which they could not solve without the invariant[1]. Our results confirm that ChatGPT can support software verifiers by providing valid and useful loop invariants, but also show that more work needs to be done – both conceptually and practically – to have LLMs provide a significant support for software verification.

2 Invariant Generation with ChatGPT

Our goal is to provide initial insights into the capabilities of large language models, specifically ChatGPT, to support formal software verification. For this, we propose the task of loop invariant generation.

Loop invariant generation. The goal of loop invariant generation is to generate *valid* and *useful* loop invariants for a given program. A valid loop invariant is an invariant that (1) holds true before the first loop execution and (2) after each loop iteration. A useful loop invariant is a valid loop invariant that is useful for proving the given program correct.

To understand this, let us consider the example task shown in Figure 1. Here, the large language model is tasked to analyze the given program and to propose a loop invariant. For the given program, the invariant `x + y == n` represents a *valid* loop invariant: as `x` is initialized to `n` and `y` to `0`, the invariant holds (1)

[1] In case of CPAchecker, we restrict CPAchecker's own invariant generation facilities as to be able to see the plain effect of the generated invariant.

before the first loop execution. The invariant furthermore holds (2) after each loop iteration as `y` is incremented each time `x` is decremented.

The provided loop invariant also is a *useful* loop invariant: As `x == 0` at the end of the loop execution and `x + y == n` holds after the loop execution, we can deduce that the assertion `y == n` is not violated after the loop execution. The invariants `x <= n` and `y >= 0` also represent valid loop invariants but they are not useful for proving the program correct.

The idea is now to let ChatGPT generate such loop invariants. To this end, we need to tell ChatGPT what its task is. As briefly mentioned in the introduction, we expect ChatGPT to give loop invariants in the form of ACSL (ANSI C Specification Language [5]) assertions. ACSL is a specification language for C and offers a number of keywords for specifications in a design-by-contract style. Among others, there is the keyword `loop invariant`. ACSL specifications are written inside comments of the form `//@`. Besides the plain code, Figure 1 also shows the *prompt* used to tell ChatGPT its task (first line), and the code location and form of the invariant we expect to be generated (`//@ loop invariant [mask]`)[2]. We thus phrase the task as an *infilling problem* [21], i.e., we require ChatGPT to fill in some meaningful contents for `[mask]`. In this example, ChatGPT returns the above discussed invariant. We arrived at this form of stating the task after several experiments with different prompts.

Feeding loop invariants into verifiers. For evaluation of the generated invariants, we need to determine their validity and usefulness. To this end, we first of all need to feed them into some verifier. Interactive verifiers natively provide ways of feeding in such inputs. In an interactive verification run, a software engineer provides program annotations (e.g., invariants) and the verifier tries to prove that some given specifications are never violated[3].

In this work, our goal is to evaluate the ability of large language models to support verifiers. Therefore, we replace the software engineer by ChatGPT and let it interact with the interactive verifier. Currently, the language model only interacts by exchanging loop invariants (which is inline with our evaluation goal). However, in future work it could be interesting to let the language model generate other types of annotations.

During our evaluation, we use the interactive verifier Frama-C [4] to evaluate the validity and usefulness of the provided invariants. For evaluating the usefulness, we furthermore employ an automatic verifier (CPAchecker [8]). To also allow for interaction in this case, we employ ACSL2Witness [10] to convert the ACSL annotated program to a correctness witness which CPAchecker is then able to use in its verification.

Related work. There are only a few works that address invariant generation via machine learning. The work in [32] uses large language models to predict invariants of Java programs. They specifically trained large language models to predict

[2] Prompt and answer from ChatGPT are abbreviated to fit the figure; the full prompt is given in the appendix.

[3] There exists a variety of properties that can be checked via verification; we focus here on checking for violations of assertions.

Daikon [20] generated invariants. Their evaluation does not consider validity or usefulness of the generated invariants but only concerns whether Daikon invariants can be recovered. In contrast, in this work, we rely on instruction-tuned large language models such as ChatGPT *without* any training and we use formal verification approaches to evaluate the validity and usefulness of loop invariants generated for C code.

Many approaches [36,31,22,35,12], which are related to or based on Syntax-Guided Synthesis, have addressed invariant generation via machine learning techniques. However, most of the existing techniques rely on traditional machine learning or graph neural network based techniques instead of large language models. We are interested in the capabilities of large language models in supporting C software verifiers.

Beyond invariants, there also exist other ways to support software verifiers. For example, the work in [3,23] supports verifiers with neural-network based termination analyses. However, these approaches are often deeply integrated. We chose loop invariant generation as many software verifiers already support the exchange of invariants.

3 Evaluation

We evaluate ChatGPT on the task of loop invariant generation in C code. For the evaluation, we use a benchmark of 106 verification tasks taken from the SV-COMP Loops category [7]. We have chosen all tasks which (a) have ACSL annotations (to be able to compare the generated with manually constructed invariants), (b) have one loop only and (c) are correct, i.e., the assertions in the code are valid. During our evaluation, we remove all ACSL invariant annotations and let ChatGPT regenerate them. Now, based on our evaluation setup we aim to answer the following research question:

Can ChatGPT support software verifiers with valid and useful loop invariants?

Experimental setup. For generating loop invariants, we employ the ChatGPT (GPT-3.5) snapshot from June 2023. The model is queried via the OpenAI API[4]. During our evaluation, we set the sampling temperature[5] of ChatGPT to 0.2 and sample up to k ($k = 5$) completions per task. We collect all invariants by parsing the generated completions with the infillings.

For checking the validity of the generated invariants, we use the interactive verifier Frama-C [4]. We annotate each task with one of the n generated invariants. In total, we thus generate up to n annotated versions of each task which we use for validation. We count loop invariants as validated only if Frama-C WP can validate them within 10s[6].

[4] https://platform.openai.com/, accessed in Sept. 2023

[5] The temperature controls the randomness of ChatGPT's outputs; a lower temperature leads to more deterministic outputs. We have chosen a low temperature to obtain invariants in a processable format.

[6] Note that a negative answer of Frama-C does not necessarily mean that the candidate invariant is invalid.

Table 1. Results for 106 verification tasks, divided by subcategory of the Loops category (giving total number of tasks, number of successfully validated invariants, number of verified tasks per verifier using either the generated or the human provided invariant of the benchmark, and in gray the number of useful invariants)

Subcategory	Tasks		Frama-C		*k*-induction	
	total	val-invs.	GPT invs.	Human invs.	GPT invs.	Human invs.
loop-accelaration	15	8	1 (1)	2 (2)	6 (3)	6 (3)
loop-crafted	2	2	0 (0)	0 (0)	2 (0)	2 (0)
loop-industry-pat.	1	1	0 (0)	0 (0)	1 (0)	1 (0)
loop-invariants	8	4	3 (3)	3 (3)	0 (0)	1 (1)
loop-invgen	3	3	0 (0)	0 (0)	0 (0)	0 (0)
loop-lit	13	4	1 (1)	4 (4)	3 (2)	4 (3)
loop-new	7	4	1 (1)	1 (1)	0 (0)	0 (0)
loop-simple	1	1	1 (1)	1 (1)	1 (0)	1 (0)
loop-zilu	22	18	10 (10)	11 (11)	11 (6)	10 (5)
loops	13	13	5 (5)	6 (6)	8 (1)	8 (1)
loops-crafted-1	21	17	0 (0)	0 (0)	4 (3)	7 (6)
total	106	75	22 (22)	28 (28)	36 (15)	40 (19)

For evaluating the usefulness of the generated invariants, we now annotate the task with the validated invariants from the previous step. If multiple invariants are validated per task, we conjunct them to a single invariant and annotate the task with the conjuncted invariant[7]. As verifiers, we consider the interactive verifier Frama-C SV [9][8] and the automatic verifier CPAchecker [8]. We configure CPAchecker to run *k*-induction without loop unrolling (similar to [10] to be able to see the effect of the generated invariant). Note that this restricts CPAcheckers facilities for verification. Finally, all verifier and validation runs are executed via BenchExec [6] on a 24-core machine with 128GB RAM running Ubuntu 22.04 with a maximum timelimit of 900s.

Results. Our main results are shown in Table 1. On the left side of the table, we show the total number of tasks per subcategory (total) and the number of tasks where at least one of the generated invariants can be validated (val-invs.). On the right side of the table, we report on the verification results obtained from executing Frama-C and CPAchecker (using *k*-induction without loop unfolding) on the verification tasks with at least one validated invariant. We report the total number of tasks that can be verified with a ChatGPT provided invariant (GPT invs.) and a human provided invariant (Human invs.), i.e., the ACSL invariant given in the benchmark. In addition, we also report the number of useful invariants in gray brackets. Useful here means that the verifier cannot complete the verification task without the invariant.

[7] The logical conjunction of two valid invariants is again a valid invariant.

[8] Frama-C SV is a version of Frama-C specifically configured to work well on SV-COMP task.

```
1 void func() {
2   unsigned int x = 0, y = 1;
3   //@ loop invariant [mask];
4   while (x < 6) { x++; y *= 2; }
5   assert(y % 3 != 0);
6 }
```

Infilling provided by ChatGPT: `x <= 6 && y == pow(2, x)`
Human: `(x==0 && y==1) || (x==1 && y==2) || (x==2 && y==4) || ...`

Fig. 2. Example task: `loop-accelaration/underapprox_1-2`

ChatGPT can generate valid loop invariants. We find that ChatGPT can generate valid loop invariants for 75 out of 106 tasks (as validated by Frama-C). Note that ChatGPT proposes loop invariant candidates for all 106 tasks and by manual inspection we found that some of the generated loop invariant candidates are still *meaningful*, even though they are not validated by Frama-C. An example is shown in Figure 2. ChatGPT produces a meaningful loop invariant candidate, but Frama-C rejects the candidate due to technical reasons[9]. The human-annotated invariant avoids this problem by enumerating all variable assignments. In total, we found by manual inspection that 10 out of 31 invariant candidates not validated by Frama-C are meaningful.

Interestingly, we found during our manual inspection that ChatGPT in many cases seems to apply a set of useful *heuristics* to determine loop invariant candidates. One of the most successful heuristic applied by ChatGPT on our benchmark is the *copy assertion* heuristic. Here, ChatGPT proposes an invariant that is equivalent to a condition found in a nearby assertion. The heuristic is applied in 30 out of 106 tasks and 23 of the resulting invariants are validated.

ChatGPT can support verifiers with useful loop invariants. We find that ChatGPT can produce useful invariants that can support software verifiers in their verification tasks. In comparison to the human-provided invariants, ChatGPT produced useful invariants for 22 out of 28 tasks in the case of Frama-C and for 15 out of 19 tasks in the case of CPAchecker's k-induction. Interestingly, we find one example in the loop-zilu subcategory where the invariant proposed by ChatGPT is more useful for CPAchecker than the human annotated invariant. The example is shown in Figure 3. Here, ChatGPT proposes the invariants `j >= 0` and `k >= 0` conjuncted with the human-provided invariant which is obviously useful to prove that `k >= 0` holds true at the end of the loop. Note that, while this seems to be a case where the copy assertion heuristics is effective, Frama-C does not validate the invariant candidate `k >= 0` alone. The conjunction with `j<=n && k>=n-j` is important to validate the invariant. Still, by manual inspection we find that the copy assertion heuristic of ChatGPT is effective for providing useful invariants in 11 out of 22 cases for Frama-C and in 5 out of 15 cases for k-induction.

[9] Frama-C reports an invalid conversion from integer type to a floating point type due to the `pow` operator and thereby fails.

```
1 void func(int k, int j, int n) {
2   if (!(n>=1 && k>=n && j==0)) return;
3   //@ loop invariant [mask];
4   while (j<=n-1) { j++; k--; }
5   assert(k>=0);
6 }
```

Infilling provided by ChatGPT: `j >= 0 && k >= 0 && j <= n && k >= n - j`
Human: `j <= n <= k + j`

Fig. 3. Example task: `loop-zilu/benchmark04_conjunctive`.

4 Limitations and Open Issues

We discuss limitations and open issues in using large language models for supporting software verifiers.

Cooperation between Language Model and Software Verifier. Our evaluation has shown that large language models such as ChatGPT are already capable of producing valid and useful loop invariants for our benchmark tasks. However, to be useful in practice, there are several challenges we have to master. A key challenge is the communication and cooperation between large language model and software verifier. Currently, we have implemented a top-down approach for invariant generation, i.e., we start by querying the language model for invariant candidates, validate them and then provide them to a verifier. The LLM has no knowledge about the specifics of the underlying validator or the verifier used in the process. This can ultimately hinder the large language model from generating valid (as validated by the validator) or useful (as determined by the verifier) loop invariants. During our evaluation, we already have encountered an example where this knowledge gap leads to meaningful but not validated invariant candidates (see Figure 2). Here, the language model has no knowledge about the specifics of the validator used (Frama-C) or at least is not informed that the proposed expression leads to a parsing error. Communicating this information allows the large language model to self-debug [17] its invariant proposals and thereby propose invariant candidates that are validated by the validator and that are useful for the verifier. For example, if we report the implicit conversion error back to ChatGPT, it generates a new invariant candidate (`y == 1 « x`) for our example in Figure 2 that is validated by our validator. Overall, we envision a cooperative approach between large language model, invariant validator and software verifier as shown in Figure 4. In an inner loop, the large language model cooperates with the validator to identify valid loop invariants. Here, the language model proposes invariant candidates, obtains feedback from the validator and refines its invariant suggestion. In the outer loop, the language model cooperates in the same way with the

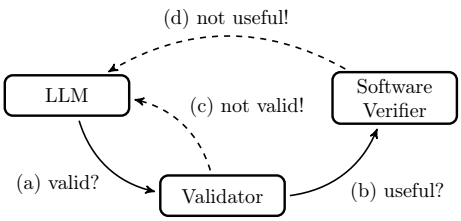

Fig. 4. Conceptual overview.

software verifier to find useful loop invariants. This work already implements (a) the validation of invariant candidates and (b) the verification with useful invariants. The key challenge is now to determine which feedback is needed from (c) the validator or (d) the software verifier to effectively guide the language model to valid and useful invariants.

A subsequent study [28] provides first insights in the feasibility of our approach. By providing feedback to the language model (in form of error messages produced by Frama-C), the authors showed that language models can effectively repair its invariant proposals. We believe that providing more detailed feedback (e.g. by providing a more detailed reasoning why the validation process fails) can further boost the performance of language model based invariant generation.

Finally, we can envision that our approach to language model and verifier cooperation may be useful beyond invariant generation. For example, TriCo [2] proposes to check the conformity between implementation and code specification with a verifier. A large language model could react to conformity violations and repair either the implementation or the specification.

Unified assertion language. Our approach for invariant generation requires that large language models, validators and software verifiers communicate invariants with a common specification language (e.g., ACSL in our case). However, in practice, there exists a zoo of interactive verifiers such as Dafny [29], Frama-C [4], KeY [1], KIV [19], and VeriFast [25] and automated software verifiers such as CBMC [18], CPAchecker [8], Symbiotic [13], and Ultimate Automizer [24]. All of them implement their own custom way to communicate invariants. Therefore, we either have to find a way to unify the communication of invariants between systems or we have to define *transformations* that convert between communication formats. In this work, we have already employed the transformation ACSL2Witness [10] to convert ACSL to a format understandable by automated software verifiers. In the future, we plan to explore alternative transformations to support a wider range of validators and verifiers.

Known limitations of LLMs. Large language models have many known limitations such as hallucinations [26], input length limitations [30], and limited reasoning capabilities [34]. All of this can significantly limit the ability of large language models to produce valid and useful loop invariants or to support software verifiers in general. However, active research is underway to overcome these limitations, and a number of proposals have already been made to reduce hallucinations [33], increase input length [16], or improve the reasoning performance [38] of large language models. It would be interesting for future work to evaluate how these solutions impact the loop invariant generation abilities of large language models.

5 Conclusion

In this work, we provided a first step towards answering the question whether large language models can support formal software verification. For this, we

have evaluated ChatGPT on the task of loop invariant generation. Our evaluation shows that ChatGPT can support software verifiers by providing valid and useful loop invariants. We plan to further improve the support for software verification in future work by a cooperative approach that enables exchange of information between large language models, invariant validators and software verifiers. In particular, we intend to develop methods for providing feedback to LLMs whenever candidate invariants are found to not be valid.

A Prompting ChatGPT

The full prompt and the answer of ChatGPT for the example task `loops/count_up_down-1` is shown in Figure 5. We use the same prompt for all tasks. The answer of ChatGPT can slightly vary between executions. Therefore, we generate up to k answers and collect invariants from all answers via a regular expression.

`Prompt>` Compute a loop invariant for the annotated loop including [invariant] in the following C code. Please use the format of ACSL annotations and always end your response with //@ loop invariant X ; where X is the computed invariant.

```
[...]
unsigned int __VERIFIER_nondet_uint();

int main()
{
  unsigned int n = __VERIFIER_nondet_uint();
  unsigned int x=n, y=0;
  //@ loop invariant [invariant];
  while(x>0)
  {
    x--;
    y++;
  }
  __VERIFIER_assert(y==n);
}
```

`ChatGPT>`

The loop invariant for the given code can be stated as follows:

```
//@ loop invariant y + x == n ;
```

[...]

Fig. 5. Full prompt and answer for example task: `loops/count_up_down-1`.

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H.: The KeY tool. Softw. Syst. Model. **4**(1), 32–54 (2005). https://doi.org/10.1007/s10270-004-0058-x, https://doi.org/10.1007/s10270-004-0058-x
2. Ahrendt, W., Gurov, D., Johansson, M., Rümmer, P.: Trico - triple co-piloting of implementation, specification and tests. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles - 11th International Symposium, ISoLA 2022, Rhodes, Greece, October 22-30, 2022, Proceedings, Part I. Lecture Notes in Computer Science, vol. 13701, pp. 174–187. Springer (2022). https://doi.org/10.1007/978-3-031-19849-6_11, https://doi.org/10.1007/978-3-031-19849-6_11
3. Alon, Y., David, C.: Using graph neural networks for program termination. In: Roychoudhury, A., Cadar, C., Kim, M. (eds.) Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022. pp. 910–921. ACM (2022). https://doi.org/10.1145/3540250.3549095, https://doi.org/10.1145/3540250.3549095
4. Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021). https://doi.org/10.1145/3470569, https://doi.org/10.1145/3470569
5. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, http://frama-c.com/download/acsl.pdf
6. Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on SV-COMP 2016). In: Chechik, M., Raskin, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016. Lecture Notes in Computer Science, vol. 9636, pp. 887–904. Springer (2016). https://doi.org/10.1007/978-3-662-49674-9_55, https://doi.org/10.1007/978-3-662-49674-9_55
7. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS. Lecture Notes in Computer Science, vol. 13994, pp. 495–522. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_29, https://doi.org/10.1007/978-3-031-30820-8_29
8. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16, https://doi.org/10.1007/978-3-642-22110-1_16
9. Beyer, D., Spiessl, M.: The static analyzer Frama-C in SV-COMP (competition contribution). In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022. Lecture Notes in Computer Science, vol. 13244, pp. 429–434. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_26, https://doi.org/10.1007/978-3-030-99527-0_26
10. Beyer, D., Spiessl, M., Umbricht, S.: Cooperation between automatic and interactive software verifiers. In: Schlingloff, B., Chai, M. (eds.) Software Engineering and Formal Methods - 20th International Conference, SEFM 2022. Lecture Notes in

Computer Science, vol. 13550, pp. 111–128. Springer (2022). https://doi.org/10.1007/978-3-031-17108-6_7, https://doi.org/10.1007/978-3-031-17108-6_7
11. Brown, T.B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D.M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., Amodei, D.: Language models are few-shot learners. In: Larochelle, H., Ranzato, M., Hadsell, R., Balcan, M., Lin, H. (eds.) Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual (2020), https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html
12. Chakraborty, S., Lahiri, S.K., Fakhoury, S., Lal, A., Musuvathi, M., Rastogi, A., Senthilnathan, A., Sharma, R., Swamy, N.: Ranking llm-generated loop invariants for program verification. In: Bouamor, H., Pino, J., Bali, K. (eds.) Findings of the Association for Computational Linguistics: EMNLP 2023, Singapore, December 6-10, 2023. pp. 9164–9175. Association for Computational Linguistics (2023), https://aclanthology.org/2023.findings-emnlp.614
13. Chalupa, M., Strejcek, J., Vitovská, M.: Joint forces for memory safety checking revisited. Int. J. Softw. Tools Technol. Transf. **22**(2), 115–133 (2020). https://doi.org/10.1007/s10009-019-00526-2, https://doi.org/10.1007/s10009-019-00526-2
14. Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J., Chen, W.: Codet: Code generation with generated tests. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net (2023), https://openreview.net/pdf?id=ktrw68Cmu9c
15. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code. CoRR **abs/2107.03374** (2021), https://arxiv.org/abs/2107.03374
16. Chen, S., Wong, S., Chen, L., Tian, Y.: Extending context window of large language models via positional interpolation. CoRR **abs/2306.15595** (2023). https://doi.org/10.48550/arXiv.2306.15595, https://doi.org/10.48550/arXiv.2306.15595
17. Chen, X., Lin, M., Schärli, N., Zhou, D.: Teaching large language models to self-debug. CoRR **abs/2304.05128** (2023). https://doi.org/10.48550/arXiv.2304.05128, https://doi.org/10.48550/arXiv.2304.05128
18. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15, https://doi.org/10.1007/978-3-540-24730-2_15

19. Ernst, G., Pfähler, J., Schellhorn, G., Haneberg, D., Reif, W.: KIV: overview and verifythis competition. Int. J. Softw. Tools Technol. Transf. **17**(6), 677–694 (2015). https://doi.org/10.1007/s10009-014-0308-3, https://doi.org/10.1007/s10009-014-0308-3
20. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C.: The daikon system for dynamic detection of likely invariants. Sci. Comput. Program. **69**(1-3), 35–45 (2007). https://doi.org/10.1016/j.scico.2007.01.015, https://doi.org/10.1016/j.scico.2007.01.015
21. Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, S., Zettlemoyer, L., Lewis, M.: Incoder: A generative model for code infilling and synthesis. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net (2023), https://openreview.net/pdf?id=hQwb-lbM6EL
22. Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 499–512. ACM (2016). https://doi.org/10.1145/2837614.2837664, https://doi.org/10.1145/2837614.2837664
23. Giacobbe, M., Kroening, D., Parsert, J.: Neural termination analysis. In: Roychoudhury, A., Cadar, C., Kim, M. (eds.) Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022. pp. 633–645. ACM (2022). https://doi.org/10.1145/3540250.3549120, https://doi.org/10.1145/3540250.3549120
24. Heizmann, M., Hoenicke, J., Podelski, A.: Software model checking for people who love automata. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 36–52. Springer (2013). https://doi.org/10.1007/978-3-642-39799-8_2, https://doi.org/10.1007/978-3-642-39799-8_2
25. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: Verifast: A powerful, sound, predictable, fast verifier for C and java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011). https://doi.org/10.1007/978-3-642-20398-5_4, https://doi.org/10.1007/978-3-642-20398-5_4
26. Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y., Madotto, A., Fung, P.: Survey of hallucination in natural language generation. ACM Comput. Surv. **55**(12), 248:1–248:38 (2023). https://doi.org/10.1145/3571730, https://doi.org/10.1145/3571730
27. Jiang, N., Liu, K., Lutellier, T., Tan, L.: Impact of code language models on automated program repair. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023. pp. 1430–1442. IEEE (2023). https://doi.org/10.1109/ICSE48619.2023.00125, https://doi.org/10.1109/ICSE48619.2023.00125
28. Kamath, A., Senthilnathan, A., Chakraborty, S., Deligiannis, P., Lahiri, S.K., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Finding inductive loop invariants using large

language models. CoRR **abs/2311.07948** (2023). https://doi.org/10.48550/ARXIV.2311.07948, https://doi.org/10.48550/arXiv.2311.07948
29. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20, https://doi.org/10.1007/978-3-642-17511-4_20
30. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P.F., Leike, J., Lowe, R.: Training language models to follow instructions with human feedback. In: NeurIPS (2022), http://papers.nips.cc/paper_files/paper/2022/hash/b1efde53be364a73914f58805a001731-Abstract-Conference.html
31. Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 42–56. ACM (2016). https://doi.org/10.1145/2908080.2908099, https://doi.org/10.1145/2908080.2908099
32. Pei, K., Bieber, D., Shi, K., Sutton, C., Yin, P.: Can large language models reason about program invariants? In: Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J. (eds.) ICML. Proceedings of Machine Learning Research, vol. 202, pp. 27496–27520. PMLR (2023), https://proceedings.mlr.press/v202/pei23a.html
33. Peng, B., Galley, M., He, P., Cheng, H., Xie, Y., Hu, Y., Huang, Q., Liden, L., Yu, Z., Chen, W., Gao, J.: Check your facts and try again: Improving large language models with external knowledge and automated feedback. CoRR **abs/2302.12813** (2023). https://doi.org/10.48550/arXiv.2302.12813, https://doi.org/10.48550/arXiv.2302.12813
34. Rae, J.W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, H.F., Aslanides, J., Henderson, S., Ring, R., Young, S., Rutherford, E., Hennigan, T., Menick, J., Cassirer, A., Powell, R., van den Driessche, G., Hendricks, L.A., Rauh, M., Huang, P., Glaese, A., Welbl, J., Dathathri, S., Huang, S., Uesato, J., Mellor, J., Higgins, I., Creswell, A., McAleese, N., Wu, A., Elsen, E., Jayakumar, S.M., Buchatskaya, E., Budden, D., Sutherland, E., Simonyan, K., Paganini, M., Sifre, L., Martens, L., Li, X.L., Kuncoro, A., Nematzadeh, A., Gribovskaya, E., Donato, D., Lazaridou, A., Mensch, A., Lespiau, J., Tsimpoukelli, M., Grigorev, N., Fritz, D., Sottiaux, T., Pajarskas, M., Pohlen, T., Gong, Z., Toyama, D., de Masson d'Autume, C., Li, Y., Terzi, T., Mikulik, V., Babuschkin, I., Clark, A., de Las Casas, D., Guy, A., Jones, C., Bradbury, J., Johnson, M.J., Hechtman, B.A., Weidinger, L., Gabriel, I., Isaac, W., Lockhart, E., Osindero, S., Rimell, L., Dyer, C., Vinyals, O., Ayoub, K., Stanway, J., Bennett, L., Hassabis, D., Kavukcuoglu, K., Irving, G.: Scaling language models: Methods, analysis & insights from training gopher. CoRR **abs/2112.11446** (2021), https://arxiv.org/abs/2112.11446
35. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Bengio, S., Wallach, H.M., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal,

Canada. pp. 7762–7773 (2018), https://proceedings.neurips.cc/paper/2018/hash/65b1e92c585fd4c2159d5f33b5030ff2-Abstract.html
36. Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2inv: A deep learning framework for program verification. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12225, pp. 151–164. Springer (2020). https://doi.org/10.1007/978-3-030-53291-8_9, https://doi.org/10.1007/978-3-030-53291-8_9
37. Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., Chi, E.H., Hashimoto, T., Vinyals, O., Liang, P., Dean, J., Fedus, W.: Emergent abilities of large language models. Trans. Mach. Learn. Res. **2022** (2022), https://openreview.net/forum?id=yzkSU5zdwD
38. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: NeurIPS (2022), http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html