Towards Combining the Cognitive Abilities of Large Language Models with the Rigor of Deductive Progam Verification*

Bernhard Beckert[1][0000−0002−9672−3291], Jonas Klamroth[2][0000−0002−8013−9453], Wolfram Pfeifer[1][0000−0002−9478−9641], Patrick Röper[1], and Samuel Teuber[1][0000−0001−7945−9110]

[1] Karlsruhe Institute of Technology, Karlsruhe, Germany
{beckert, wolfram.pfeifer, teuber}@kit.edu
[2] FZI Research Center for Information Technology, Karlsruhe, Germany
klamroth@fzi.de

Abstract. Recent investigations hint at the ability of large language models (LLMs) to generate formal specifications for given program code. In this work, we systematically discuss and categorize different use cases and application scenarios that combine specification synthesis via LLMs with deductive program verification. We present preliminary quantitative experiments on the capabilities of LLMs to generate correct specifications. To this end, we use a prototypical integration of GPT (versions 3.5 and 4o) with the deductive program verifier KeY and the bounded model checker JJBMC. We evaluated our prototype on a set of Java programs that are partially annotated with specifications written in the Java Modeling Language (JML). We show that GPT 4o generates correct annotations in approximately half of all instances across the investigated scenarios. For the case of faulty specifications, we investigate how a feedback loop can help to improve the original answer. Finally, we present a vision of how Large Language Models may support rigorous formal verification of software systems and describe the necessary next steps in this direction.

Keywords: Deductive Program Verification, Large Language Models, Specification Generation, Design by Contract, Java Modeling Language

1 Introduction

To this day, the formal specification of software in a manner that admits full functional verification is a tedious task requiring many human work hours. The

* This version of the contribution has been accepted for publication, after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: https://doi.org/10.1007/978-3-031-75387-9_15. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms

program verification community has developed numerous symbolic AI tools that admit deductively proving functional properties and to a certain extent can derive specifications from code. But these tools lack a human developer´s *cognition* that admits an intelligent prediction of necessary annotations based on the surrounding code and specification. In the research line of subsymbolic AI, recent advances in large language models (LLMs) have shown how these models can bring a similar kind of cognition to code generation in scenarios where only a minimal description of the intended behavior is given [7]. While the cognitive ability of LLMs does not reach that of humans (yet), it is worthwhile to explore their use for specification generation (in addition to code generation). To this end, we see three main application scenarios for leveraging *Intersymbolic AI* [22] in program specification by pairing rigorous verification with a language model's cognition: (1) the generation of requirement specifications based on code, natural language, or based on other specification languages; (2) The generation of auxiliary specifications; (3) the co-development of specification and code. In use case (2), we can guarantee soundness, or *external consistency*, because flaws in *auxilliary* annotations can prevent a successful proof for the top-level specification, but can never lead to an unsound verification result. For use cases (1) and (3), however, we can only ensure *internal consistency*, i.e., we can check via verification whether the code satisfies the generated specification, but not whether the specification itself is adequate. To achieve the latter would require human intervention.

In preliminary experiments, we observed that OpenAI's ChatGPT has sufficient knowledge of Design by Contract to produce semantically valid contracts in the Java Modelling Language (JML) [14], a language for annotating Java code with contract-based specifications. Based on this observation, this work explores the potentials and limits of state-of-the-art Large Language Models, represented by OpenAI's GPT family, in JML annotation generation.

In Section 2, we outline potential use cases of LLM-based annotation generation in detail. Subsequently, we present a prototypical integration of JML verifiers (KeY [1] or JJBMC [4]) with an LLM (see Section 3) and report experimental results obtained with our prototype (see Section 4). Finally, we outline a vision for how LLM-based annotation generation can push software verification forward and present plans for our future research.

Related Work. Multiple prior works explored LLM capabilities for loop invariant or assertion generation for C [6, 13, 15, 26], Rust [27] or Java programs [20] although no verification w.r.t. a top-level-specification took place in the latter case. Contrary to the large majority of benchmarks evaluated in prior work and supported by JML's rich specification language, the benchmark set used for our evaluation contains many instances with a rich set of program primitives such as arrays (e.g. sorting or search) and object fields (e.g. array sets) requiring specifications with quantifiers and framing. In the same direction of Intersymbolic AI for program verification, Laurent and Platzer [17] proposed a technique for the combination of nondeterministic programming with reinforcement learning which is directly applicable to loop invariant synthesis. Lathouwers and Huisman

evaluated the use of ChatGPT for the generation of JML annotations for isolated methods [16]. Contrary to our work, the authors did not consider an automated feedback loop between a verifier and GPT and did not evaluate the generation of auxiliary annotations w.r.t. a top-level specification. Granberry *et al.* [11] provide a qualitative evaluation of the capabilities of GPT-4 for specification of C programs. In this way, the work is orthogonal to our quantitative analysis for the isolated methods use case (see Section 4). Additionally, we also evaluate the scenario of auxiliary annotation generation w.r.t. a given top-level specification. Sun *et al.* [24] propose an approach for consistency checking between generated (Dafny) code, specification, and comments to improve the quality of code/specification generations. This work can be seen as complementary to our work which explores the capabilities of a large language model for JML-based top-level and auxilliary specification generation.

2 Use Cases for Automated Specification Annotation

We envision three different use cases in which the generation of specifications by an LLM could be harnessed for verification purposes. Specifically, we consider (1) the generation of requirement specifications, (2) the generation of auxiliary specifications, and (3) the conversational co-development of specification and code. We will discuss each of the use cases in detail in the following sections. For illustration, we use the code snippets in Listings 1.1 to 1.3.

Listing 1.1 shows a single method, which can be given to the LLM with the task of generating a top level contract. This corresponds to use case (1).

Listing 1.2 contains a top-level method `f` and a method `g` called from within the body of `f`. The top-level specification is given as a clause in JML[3]. It states that after the termination of the method, the returned value (denoted by `\result`) has a specific value. This is an example of a very simple contract only containing a postcondition. Additionally, we could specify a method's precondition (via a `requires` clause), that the method terminates without exceptions (via the keyword `normal_behavior`), its write effects (via an `assignable` clause), or which exceptions are thrown (via `signals` clauses). In general, JML contains a lot of such features to specify program behavior in great detail. However, for the purpose of this paper, the basic method and loop specification elements are sufficient. For this listing, two of the use cases fit: the generation of auxiliary specifications (2) and the co-development of specification and code (3). The latter could for example occur when code from the method body of `f` is refactored into the new method `g`.

Listing 1.3 contains only a single method `f` with a contract, which states that after termination the variable `y` contains the sum of all values from 1 to `x`. Since the method body contains a loop, to be able to verify the contract, a loop invariant (more precisely, a loop *specification*, which also contains framing and

[3] In principle, the considerations in this section are valid for other contract-based specification languages (ACSL, Dafny, ...), but for the examples and experiments of this paper we focus on JML.

```
int g(int x) {
  return x+x;
}
```

Listing 1.1. Isolated method without annotation.

```
//@ ensures
//@   \result == -2*x;
int f(int x) {
  return g(-x);
}
int g(int x) {
  return x+x;
}
```

Listing 1.2. Called method without annotation

```
//@ ensures
//@   y == x*(x+1)/2;
void f(int x) {
  y = 0;
  while (x>0) {
    y+=x; x--;
  }
}
```

Listing 1.3. Specified method containing loop without annotation

termination information) is necessary. This listing demonstrates use case (2), where the LLM is tasked with the generation of this auxiliary specification.

2.1 Generating Requirement Specification

A major bottleneck of program verification is that the properties to be verified are only given informally as texts. Thus, any verification effort relies on the conversion of informal texts to formal specifications. This conversion is often tedious, error-prone, and time-consuming and thus presents a major hurdle for formal methods. To overcome some of the problems, formal requirement specifications for the verification target (code block, function, entire software) could be generated by an LLM from a human-provided natural language description. E.g. for Listing 1.1 an engineer would simply state the to-be-verified method `g` is "doubling the value of `x`" and the LLM would provide a suitable formal specification in the specification language of choice. While potentially providing huge benefits in terms of accessibility of formal specification, this approach has a major drawback: The verification of the generated specification does not give any guarantees with regard to the original, informal properties. Thus, verification here only guarantees *internal consistency* between code and generated specification (e.g. for Listing 1.1 we can only guarantee that a generated specification is indeed satisfied by the method `g`). A similar drawback is that the generated contract could be arbitrarily weak, that is, only state "ensures true" as a post-condition. However, the adequacy of the generated specification then needs to be checked by the human engineer and possibly improved (with or without help from the LLM). Thus, the complex task of guaranteeing that a specification actually reflects the intended behavior (which might be the real bottleneck) still falls into the hands of a human. To make this LLM application effective, useful summaries of a specification improving an engineer's understanding of the generated specification may prove to be useful.

2.2 Generating Auxiliary Specifications

Another bottleneck for a lot of tools is the need for auxiliary specifications such as loop invariants, pre-/postconditions for helper functions, or other assertions. This is especially true for approaches like auto-active verification, where the interactions consist of textual annotations in the input artifact (a term coined by Leino and Moskal [18]), or for interactive proof assistants. LLMs can be used to generate such auxiliary specifications automatically. For example in Listing 1.2, an LLM could be used to annotate method `g` with a contract that allows the verification of `f` without reliance on unrolling. Similarly, in Listing 1.3, an LLM could be used to annotate the loop with a suitable loop specification that allows the verification of `f`. First and foremost, auxiliary specifications can be generated based on the method's code which describes its behavior precisely. Additionally, pre-existing top-level specifications can be used as a basis for the LLM's generations. This second input is of particular importance to later ensure that we can verify, both, the internal consistency between code and generated annotation (e.g. in Listing 1.2 we ensure that the LLM generated contract correctly reflects the behavior of `g`) as well as the external consistency between specification and top-level-specification of the calling method or surrounding code (for example in Listing 1.2 we also ensure that `f` is verifiable w.r.t. the generated contract for `g`). Moreover, providing pre-existing specifications to the LLM also helps as a prompting strategy as it can serve the LLM as a template for its generated annotations – in particular since invariants and submethod contracts often share similar clauses with the top-level annotation.

A major advantage of generating only auxiliary specifications in contrast to the previous scenario, is that the generated specification is not relevant to the soundness of the overall proof. In other words: If a verification attempt relying on generated auxiliary specification succeeds, the result is as good as any other verification (not relying on generated specification). This also means that no human has to check the generated specification for errors as those would necessarily show up during verification if present. A direct consequence of this fact is that generating arbitrarily many variations of such specifications and testing if any of those are suitable to conduct the proof is in principle a valid strategy.

2.3 Conversational Co-Development of Specification and Code

Recent studies demonstrated that the use of GitHub Copilot can significantly increase the speed of application development [21]. At the same time, it is common knowledge that LLMs may produce flawed code [7, 9] and that their use may thus introduce bugs into a codebase. By using LLMs for the co-development of specification annotations and code, we may be able to mitigate this drawback of today's LLM-based code generation. In this setting, a developer's change to the code base would be answered by appropriate changes to the corresponding specification. Similarly, given a change in the specification, the LLM would update the code accordingly. We can then use a verification tool to prove predefined top-level specifications correct after the changes are applied. For example, one

could imagine that in Listing 1.2 method `f` originally computed `-2*x` directly and that the computation was now extracted into a new method. It would then be the LLM's task to add a new annotation for `g` which reflects the code change and allows the verification of the top-level specification. Ideally, this approach would be conversational: By leveraging the support of LLMs for natural language understanding, the LLMs could pose questions to the developer to appropriately adapt the specification/code. Ahrendt *et al.* [2] discuss such a development model in which annotation generation methodologies like the one evaluated in Section 4 may be applicable.

3 A Prototype for Verified Annotation Generation

While the prior considerations about using LLMs for specification synthesis are tool agnostic, we implemented a prototype of the concept combining LLMs and the KeY verification tool, and we also performed experiments combining LLMs with the model checker JJBMC.

Verifiers. KeY [1] is a tool which allows verifying deductively that a Java program adheres to a specification written in JML. For verification, KeY uses a sequent calculus and symbolic execution. Rules can be applied automatically by a built-in proof search strategy or manually via a graphical interface by the user. Optionally, KeY can use SMT solvers as verification backend. One central concept of KeY is modular verification: By abstracting the behavior of called methods with contracts, during verification, only the implementation of a single method has to be considered at any given time. This leads to a clean separation of specification and implementation, where in case of changes in the program only a few contracts, often only a single one, have to be re-verified.

JJBMC [4] is a bounded model checker for Java based on JBMC [8]. However, in contrast to the original JBMC, which examines all methods provided at the same time via inlining, JJBMC works with contracts in JML and thus can be used for modular verification as well.

Prototype. We implemented a prototype integration for LLMs with the above-mentioned verifiers. Our integration prompts GPT to generate a contract for a given method or loop and then attempts to verify the returned annotation. In case the verification fails, we prompt GPT with primitive feedback on the failure reason (e.g. open proof branches, parsing errors, or traces) and request a new annotation. Our pipeline can be used to generate auxiliary annotations for submethods or loops (see also Section 2.2) in which case our pipeline also attempts to verify the calling/surrounding method w.r.t. the provided annotation. In this case, we consider an attempted annotation successful if the verifier can prove, both, internal consistency (the generated annotation describes the source code) and external consistency (the generated annotation allows the verification of a top-level specification). Moreover, our pipeline can be used for the JML annotation of isolated methods (see also Section 2.1) in which case we have no

guarantees beyond internal consistency (between code and generated annotation). Since our objective was to obtain a baseline for the performance of a naive GPT integration, we did not put significant effort into prompt engineering.

4 Experiments

We evaluated different aspects of our prototypical integration w.r.t. OpenAI's GPT 3.5 and the newly released GPT 4o. For the KeY integration prototype, we performed an extensive quantitative evaluation (see Section 4.1), for JJBMC we evaluated GPT's ability to interpret trace-based counterexamples (see Section 4.2). To this end, we also collected a set of benchmark instances with missing JML annotations which can be used to evaluate the performance of LLM-based annotation generation.

Benchmark Selection. For our evaluation, we collected a set of JML annotated Java files as a benchmark set. First, we collected JML annotated files from the repositories of our tool KeY. Additionally, we repurposed JML exercises from the last 10 iterations of our formal methods lecture as benchmarks. We categorize the benchmarks into three classes: isolated methods, contracts for submethods (called by a method annotated with a top-level specification), and loop specifications (within a specified method). The programs mainly contain simple algorithms, which often use arrays, and the specifications involve a lot of quantifiers and basic features of the Java Modeling Language (JML). In particular, we made sure not to use some of the more "exotic" and rarely used features of JML, such as history constraints or ownership modifiers. Examples of benchmarks are a method to calculate the greatest common divisor, an insertion sort implementation, or a method to detect whether an array contains a palindrome. An overview of important features of Java and JML that were used in the benchmark examples can be found in Table 1. We summarize all features that showed up in the original file from which we constructed the benchmark instance. This is an interesting statistic because during specification generation, these primitives either need to be reconstructed (for the missing specification) or parsed (for the surrounding code) by the LLM. Note that most of the benchmarks use arrays and quantifiers, about two third of them require non-empty assignable clauses (that is, not just `\nothing`), and in about 40 the `\old` operator is used to refer to the pre-state. Also, a particularly interesting feature is the use of method calls in the specification, which is needed in 24 cases. Moreover, we have 22 benchmark instances that access an object's field. The following steps were conducted in the preparation of the benchmarks: (1) Each file was checked for provability via KeY with default settings and a step limit of 8,000 rule applications; (2) In each benchmark file one annotation was removed (i.e., GPT had to fill in one method contract or loop annotation). Using this approach, we were able to generate multiple benchmarks from a single JML annotated Java file by masking different parts of the annotation. In total, we constructed a benchmark set with 37 examples for the "isolated method", 27 examples for the "loop invariant", and 14 for the "submethods" category.

Table 1. Overview of a selection of Java/JML features used in the benchmarks.

	Isolated Methods	Submethods	Invariants	Total
# Benchmarks	36	27	14	77
Quantifiers	32	24	14	70
Arrays	30	23	13	66
Non-empty assignable clause	24	17	11	52
Referring to the pre-state (`\old`)	18	11	8	37
(Pure) Method calls in specification	10	7	7	24
Field access	11	7	3	21

Table 2. Overview on experimental results: For each category, we report the mean (μ) and standard deviation (σ) of our success rate when verifying GPT-3.5 or GPT-4o generated JML contracts in KeY.

Category	# Benchmarks	$\mu \pm \sigma$ of success rate (%)	
		GPT 3.5	GPT 4o
Isolated Method	36	52.2 ± 4.3	$\mathbf{62.0 \pm 1.6}$
Submethods	14	19.3 ± 12.1	$\mathbf{40.5 \pm 4.1}$
Invariants	27	37.0 ± 7.4	$\mathbf{67.9 \pm 5.7}$

4.1 Experimental Results

We evaluated our prototype, both, w.r.t. GPT 3.5 and w.r.t. the newly released GPT 4o. An overview on the success of the approach across all three problem categories is provided in Table 2. For our experiments, we used KeY with its default configuration and a limit of 10,000 steps per invocation. Additionally, we set a timeout of 100 seconds for any invocation of KeY. If GPT did not provide a correct annotation within 8 answers, we aborted the pipeline. For each task category, we measure how many of the benchmarks (total: # Benchmarks) were solved and report means and standard deviations resp. across 10 and 3 repetitions for GPT 3.5 and GPT 4o[4]. Due to excessive timeouts, we had to exclude one benchmark from the Isolated Method category. While GPT was able to provide numerous verifiable annotations for isolated methods, the results are weaker for submethod and invariant generation with the exception of invariant generation using GPT 4o. This is expected, as the generation of an annotation that is sufficient for proving top-level specifications is a harder task than providing *any* internally consistent specification. Notably, we observe that GPT 4o outperforms GPT 3.5's capabilities across all three benchmark categories. In the following, we will discuss the results for the three categories in more detail.

Annotation Generation for Isolated Methods The observation that this benchmark category is very successful may be of little surprise: Given, that we *only* check whether the annotation is verifiable for the given method, the

[4] As GPT 4o is significantly more expensive than GPT 3.5 and we observed low variance after 3 experiment runs, we decided to omit further repetition for GPT 4o.

provided contracts may be arbitrarily weak. For example, a contract `requires true; ensures true;` would count as success raising the questions about the nature of contracts generated by GPT in this scenario. A manual inspection of successfully verified contracts found no cases in which the generated contract was of this undesirable form. GPT 4o neither produces the kind of trivial contract mentioned above. The generated contracts contained specific, meaningful (though sometimes incomplete) descriptions of the implementation's behavior. While GPT 3.5 solves 52.2% of benchmarks on average, across all 10 iterations we were able to find solutions for 27 of the 36 considered benchmarks (i.e. for 75% of benchmarks). We observe that the LLMs usually do not generate `normal_behavior` without explicit prompting which is of particular importance for submethod contracts (see below). In some cases, we observed that running the procedure multiple times produced different, complementary contracts that could be combined.

Recovery from invalid annotations. In case GPT did not provide a verifiable annotation for the first attempt, we performed up to 8 iterations in which GPT was fed with primitive information on the kind of failure encountered and asked for an updated annotation. Thus, we analyzed after how many iterations GPT provided the correct annotation. Overall, we received 188 correct answers by GPT 3.5. As can be seen in Figure 1, GPT 3.5 often provides the correct answer on the first try and the likelihood of obtaining a correct answer decreases with conversation length. Similar behavior is observable for GPT 4o. We also analyzed the reasons why verification failed. We categorize the errors into syntactic errors (JML was not parseable) and semantic errors (verification failed) and summarize the results in Table 3. The most commonly encountered failure case is an incomplete proof, i.e. KeY cannot verify that the method satisfies the provided specification or times out during the attempt. For isolated methods, this means the generated specification does not match the code's behavior, i.e. there is an internal inconsistency. The most commonly identifiable syntactic error concerns the incorrect usage of variables. We also observe significantly fewer syntactic errors with GPT 4o vs. GPT 3.5. Given the high share of timeouts (likely a result of excessive rule application for proving a wrong annotation), quickly finding bugs in wrong annotations may be a worthwhile extension of the approach. While the feedback provided to GPT sometimes helps to repair annotations, the question of how to prompt GPT to obtain improved contracts and how to represent feedback on errors w.r.t. the current annotation remains a major avenue for future research.

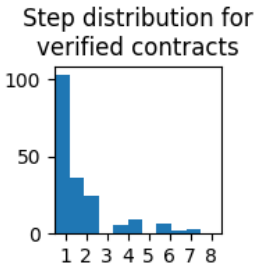

Fig. 1. Number of observed executions where successful contract generation happened in i-th iteration (out of 8) for GPT 3.5.

Table 3. Reasons for failed verification during annotation generation for isolated methods (percentages do not sum to 100 due to rounding errors).

Error Category	Share of error (%)	
	GPT 3.5	GPT 4o
Syntactic Errors		
Loop Invariant Generation	4.3	1.3
Unknown variable names	11.8	2.2
Incorrect usage of `\result`	1.1	0.3
Other parsing errors	17.1	10.7
Semantic Errors		
Incomplete proof	49.5	70.3
Timeout of Verifier	16.0	15.1

Data Contamination. The observed results raise the question of whether parts of our benchmark set were part of GPT's training dataset. While the solutions to the JML exercises from our lecture have not been publicized online, the benchmark instances stemming from the KeY repository are available on GitHub. Unfortunately, it is impossible to check for data contamination due to the taciturnity of OpenAI about used training data. However, one (weak) indicator of memorization (i.e. the language model *remembering* the correct solution instead of predicting it) might be the repetition of the exact same solution, i.e. the language model repeating the same solution multiple times. For GPT 3.5, omitting benchmarks where we observe the bit-precise repetition of contracts, we find 14 of the 27 instances that were solved in at least one of the 10 runs remain. For GPT 4o, we cannot provide a similar number as the experiments were only run for 3 repetitions. Another indicator for memorization may be a language model's token log probabilities: Text that was used for training might have a higher log probability. While log-probabilities are neither available for GPT 3.5 nor GPT 4o, evaluating log-probabilities for the predecessor model GPT 3 (model codename davinci-002) showed no significant correlation between success and log-probability ($r = 0.11$ with $p = 0.49$ for GPT 3.5, $r = 0.18$ with $p = 0.32$ for GPT 4o). Thus, while we cannot rule out data contamination for our benchmark data set, we also were not able to find strong evidence in favor of this hypothesis.

Auxilliary Annotation Generation For auxiliary annotation generation, we evaluated, both, the LLM's capabilities for generating loop invariants and its capabilities for generating contracts for submethods. In both cases, we also verified a surrounding or calling method annotated with a specification (the top-level specification was also an input for the LLM). This allows us to evaluate whether the LLM is capable of generating annotations that help with the completion of a given specification task. For annotation of submethods, we observe that GPT 4o significantly outperforms GPT 3.5: While GPT 3.5 solves 19.2% of the benchmark instances on average, GPT 4o solves 40.5% of the instances on average. An example of the kind of contracts generated by GPT can be found in

Listing 1.4: In this case, GPT 4o had to create a contract for a submethod that performs a cyclic rotation of array elements by an offset `len`. The method contains three loops and operates over two arrays. GPT 4o generated this contract after two failed attempts due to timeouts. In this instance, we observe that GPT 4o demonstrates the correct usage of quantifiers, the assignable clause, and the `\old` annotation. In case we cannot verify a benchmark, there are two semantic failure cases: Either we cannot verify the top-level specification using the contract generated for the submethod or we can verify the top-level specification w.r.t. the contract, but the contract cannot be verified w.r.t. the submethod. For loop invariants, we find that GPT 4o also outperforms GPT 3.5 reaching a success rate of 67.9%. It is worth to note that across all three benchmark runs, GPT 4o generated correct invariants for 22 out of 27 benchmarks (i.e. for 81.4% of benchmarks). This indicates, that repeated sampling generates more correct annotations.

```
/*@ normal_behavior
  @ requires a != null && 0 <= len && len <= a.length;
  @ assignable a[*];
  @ ensures (\forall int i; 0 <= i && i < len; a[i] == \old(a[a.length - len + i]));
  @ ensures (\forall int i; len <= i && i < a.length; a[i] == \old(a[i - len]));
  @*/
```

Listing 1.4. Example of an annotation generated by GPT 4o for a method performing a cyclic rotation of an array: The contract is sufficient to prove a top-level-specification of a calling method.

4.2 Correcting Generated Specifications with Traces

To explore the possibility of recovering from faulty generated specifications, we provided counterexamples to the LLM in the form of traces of the proposed solutions. For this purpose, we utilized a bounded model checker capable of verifying JML contracts: JJBMC [4]. This tool is deemed suitable due to its comparative speed and its ability to generate traces pinpointing the specific verification conditions and inputs that lead to the failed proof attempt. Since we observed in the previous section that about 50% of the generated specifications are still flawed fast and accurate feedback is essential which is underscored by the large share of timeouts. Our setup for JJBMC closely mirrored that of the previous experiments: First, we asked the LLM to generate an (initial) specification. Subsequently, this specification was verified with JJBMC. In cases where JJBMC found a counterexample, we fed this counterexample back to the LLM and requested a corrected version of the specification. We aborted either if the verification succeeded or if a predetermined number of attempts was reached.

We have not yet conducted experiments as thorough as those in previous sections, because JJBMC is not readily applicable to the considered benchmark instances (e.g., due to special KeY-specific JML constructs or unsupported JML features in the benchmark). However, we were able to conduct some preliminary manual tests. The results of these tests were mixed, leading to three major takeaways which we discuss below.

The provided invariant does not hold for the following instantiations of variables:
k = 0
a = [0, 0, 0]
i = 1

You're right; we still need to refine the loop invariant to handle the specific case where the array length is odd. In such cases, the middle element should remain unchanged.
Let's correct the loop invariant to ensure it properly handles the cases when the length of the array is odd. We need to ensure that the middle element remains unchanged throughout the loop execution.
Here's the corrected code:
...

Fig. 2. Example of a conversation with ChatGPT confronting it with a counterexample to its generated invariant

Correcting Simple Errors is Possible. The initial attempts by the LLM sometimes contain easy-to-avoid mistakes. A common mistake is not to provide an assignable clause when generating subcontracts or invariants. Assignable clauses in JML are used to specify the locations of the heap that are allowed to be modified by the specified part of code. Leaving out such a clause defaults to the most general assumption that the entire heap could be modified. This omission is problematic as it will not allow for the verification of the top-level method if it allows for anything less than the modification of the entire heap. Providing the LLM with the corresponding JJBMC error typically results in it correctly adding the missing clause, thereby avoiding the mistake. We observed that similar simple errors, where the specification was not provable due to a wrong assignable clause, were also correctly identified and fixed by the LLM when given appropriate feedback from the verification tool.

LLM Correctly Interprets Most Traces. Another observation is that the LLM often correctly summarizes the meaning of counterexamples to its specifications. For instance, consider the example illustrated in Fig. 2. A concrete counterexample is provided that shows that a loop-invariant does not hold for a certain set of values. The LLM correctly explains this scenario, however, the conclusions it draws regarding the newly generated JML are often incorrect. This suggests that further experiments are needed. It is possible that different or more refined prompts could lead to better results, or it may indicate that achieving correct specifications may require significant adjustments to our approach.

One-Shot Strength. Our manual experiments reinforce the earlier observation that the system mainly gets it right on the first try or not at all. Ignoring simple mistakes as mentioned above, we found no instance where an initially incorrect specification was fully corrected by providing counterexamples alone. While we

found several promising examples where the LLM initially corrected its specification in a seemingly positive direction, continued exposure to new counterexamples eventually led to the recycling of old versions or even complete deterioration, sometimes even resulting in alterations to the original code. Consequently, we never successfully corrected an initially faulty specification. Although intermediate improvements appeared promising, the inability to find complex specifications using our proposed method must be considered a negative outcome. Future experiments, incorporating various adaptations and enhancements to our experimental setup, are necessary to reach a conclusive verdict on this matter.

5 Vision and Research Plan

Deductive verification is important for guaranteeing dependability of critical software. It can be used in a co-development manner or even to find bugs in deployed software [5, 10, 12]. However, while worthwhile, verifying a larger software library using tools such as KeY is a work-intensive endeavor that requires large amounts of manual annotation work. We believe that integrating Large Language Models into the annotation generation process bears the potential to significantly reduce this workload: If we can reduce the human annotation work-load to writing (or, in the next step, maybe even only checking) *top-level* specifications, this would enable us to exploit the pre-existing machinery for Java verification via KeY to verify much larger parts of the Java Standard Library or to verify other large software projects without the manual effort incurred today. We envision as a long term goal that an integration of Large Language Models with autoactive verifiers will eventually be able to verify algorithms from, say, the Java Standard Library solely based on a human-provided top-level specification. While the results presented in Section 4 represent a promising first step in this direction, putting this approach into practice requires significant further research which we outline throughout this section.

Recovery from invalid annotations. While GPT's performance in a "one-shot" setting is surprisingly strong, we believe that providing well-presented, fine-grained feedback to a proposed annotation can push the capabilities of LLMs for annotation generation even further. The great advantage of using LLMs in (auxiliary) annotation generation is the observation that we can check whether a provided annotation is helpful. The KeY community has put great effort into the development of tools that make it easier for humans to understand the reasons why a proof fails. Some examples of this are counterexample generation, trace generation using JJBMC, KeY's source code view which maps constraints in the sequent back to JML expressions, and the proof branch structure which tells us in what way an annotation is insufficient. It is now the time to repurpose all these utilities and to make this information processable by LLMs. To this end, LLMs capable of performing program trace reasoning may be a viable option [19].

Large-scale annotation. In the experiments outlined in Section 4 we assumed that all but one annotation had already been provided. In future research, we

plan to derive strategies allowing for the annotation of entire classes w.r.t. a single given top-level specification.

A standard library of JML predicates. Even when a large language model correctly classifies the behavior of a method, it may still provide the wrong annotation due to hard-to-spot errors in the generated predicates. For example, it might correctly classify a method as sorting, but introduce a subtle indexing bug into the predicate ensuring that the result is a permutation of the original method. This raises the question of whether we need a standard library of common JML predicates that are less error-prone than native encodings by the LLM.

Influence of model architecture. In future research, we plan to evaluate how much our baseline results are improved through the use of other models (e.g. the Llama model family [23, 25] or other closed-source models). Additionally, it is possible to create custom LLMs that are specifically tailored to a given task via fine-tuning. This could be used to improve performance by explicitly training the model on correct examples of contracts. We also plan to evaluate this approach in future research.

Dataset curation. To develop and test prompt engineering approaches as well as to fine-tune models for annotation generation, we require a larger set of benchmark instances. To this end, we hope to collaborate with other JML-based verification tools to construct a larger code corpus, potentially using translation tools in between different JML-dialects [3]. Additionally, the generation of synthetic data via machine learning methods (e.g. using similar techniques to Laurent and Platzer [17]) may be a viable option to generate data for fine-tuning and/or prompt engineering.

Generation from informal description. In this work, we considered only the generation of specifications from existing code or other generations without additional input on what this specification is supposed to contain. In the future, we plan to investigate how additional sources of information can be leveraged to improve the quality of the generated specifications. There is a wide range of possible sources for that including but not limited to an informal description of the behavior of the code, test cases, examples of correct behavior, and documentation.

Acknowledgements. This work was supported by funding from the pilot program Core-Informatics of the Helmholtz Association (HGF) as well as the DFG projects BE 2334/9-1 and UL 433/3-1.

References

1. Ahrendt, W., Grebing, S.: Using the KeY Prover. In: Deductive Software Verification - The KeY Book - From Theory to Practice. Ed. by W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P.H. Schmitt, and M. Ulbrich, pp. 495–539. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6_15

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. LNCS, vol. 13701, pp. 174–187. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-19849-6_11
3. Armborst, L., Lathouwers, S., Huisman, M.: Joining Forces! Reusing Contracts for Deductive Verifiers Through Automatic Translation. In: Herber, P., Wijs, A. (eds.) iFM 2023 - 18th International Conference, iFM 2023, Leiden, The Netherlands, November 13-15, 2023, Proceedings. LNCS, vol. 14300, pp. 153–171. Springer, Heidelberg (2023). https://doi.org/10.1007/978-3-031-47705-8_9
4. Beckert, B., Kirsten, M., Klamroth, J., Ulbrich, M.: Modular Verification of JML Contracts Using Bounded Model Checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I. LNCS, vol. 12476, pp. 60–80. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-61362-4_4
5. Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.: Formally Verifying an Efficient Sorter. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I. LNCS, vol. 14570, pp. 268–287. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-57246-3_15
6. 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. (2023). https://doi.org/10.18653/V1/2023.FINDINGS-EMNLP.614
7. 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 (2021). arXiv: 2107.03374
8. Cordeiro, L.C., Kesseli, P., Kroening, D., Schrammel, P., Trtik, M.: JBMC: A Bounded Model Checking Tool for Verifying Java Bytecode. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I. LNCS, vol. 10981, pp. 183–190. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-319-96145-3_10
9. Dakhel, A.M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M.C., Jiang, Z.M.(: GitHub Copilot AI pair programmer: Asset or Liability? J. Syst. Softw. **203**, 111734 (2023). https://doi.org/10.1016/J.JSS.2023.111734

10. de Gouw, S., de Boer, F.S., Bubel, R., Hähnle, R., Rot, J., Steinhöfel, D.: Verifying OpenJDK's Sort Method for Generic Collections. J. Autom. Reason. **62**(1), 93–126 (2019). https://doi.org/10.1007/S10817-017-9426-4
11. Granberry, G., Ahrendt, W., Johansson, M.: Specify What? A Case-Study using GPT-4 and Formal Methods For Specification Synthesis. In: AI for Math Workshop @ ICML 2024 (2024). https://openreview.net/forum?id=ZRTcPkNl7v
12. Hiep, H.A., Maathuis, O., Bian, J., Boer, F.S.D., van Eekelen, M.C.J.D., Gouw, S.D.: Verifying OpenJDK's LinkedList using KeY. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, 26th Intl. Conf. TACAS, Dublin, Ireland, Part II. LNCS,vol. 12079, pp. 217–234. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-45237-7_13
13. Janßen, C., Richter, C., Wehrheim, H.: Can ChatGPT support software verification? In: Beyer, D., Cavalcanti, A. (eds.) Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings. LNCS, vol. 14573, pp. 266–279. Springer, Heidelberg (2024). https://doi.org/10.1007/978-3-031-57259-3_13
14. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.: *JML Reference Manual*. Draft revision 2344. May 2013. http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf.
15. 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). arXiv: 2311.07948
16. Lathouwers, S., Huisman, M.: Survey of annotation generators for deductive verifiers. Journal of Systems and Software **211**, 111972 (2024). https://doi.org/10.1016/j.jss.2024.111972
17. Laurent, J., Platzer, A.: Learning to Find Proofs and Theorems by Learning to Refine Search Strategies: The Case of Loop Invariant Synthesis. In: Koyejo, S., Mohamed, S., Agarwal, A., Belgrave, D., Cho, K., Oh, A. (eds.) Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022 (2022)
18. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Ball, T., Zuck, L., Shankar, N. (eds.) Usable Verification Workshop (2010). https://fm.csl.sri.com/UV10
19. Ni, A., Allamanis, M., Cohan, A., Deng, Y., Shi, K., Sutton, C., Yin, P.: NExT: Teaching Large Language Models to Reason about Code Execution. CoRR **abs/2404.14662** (2024). arXiv: 2404.14662
20. 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.) Proceedings of the 40th International Conference on Machine Learning. Proceedings of Machine Learning Research, pp. 27496–27520. PMLR (2023). https://proceedings.mlr.press/v202/pei23a.html
21. Peng, S., Kalliamvakou, E., Cihon, P., Demirer, M.: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. CoRR **abs/2302.06590** (2023). arXiv: 2302.06590
22. Platzer, A.: Intersymbolic AI: Interlinking Symbolic AI and Subsymbolic AI. In: Margaria, T., Steffen, B. (eds.) ISoLA 2024. LNCS, Springer, Heidelberg (2024)

23. Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Canton-Ferrer, C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., Synnaeve, G.: Code Llama: Open Foundation Models for Code. CoRR **abs/2308.12950** (2023). arXiv: 2308.12950
24. Sun, C., Sheng, Y., Padon, O., Barrett, C.: Clover: Closed-Loop Verifiable Code Generation. In: Avni, G., Giacobbe, M., Johnson, T.T., Katz, G., Lukina, A., Narodytska, N., Schilling, C. (eds.) AI Verification, pp. 134–155. Springer Nature Switzerland, Cham (2024)
25. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., Lample, G.: LLaMA: Open and Efficient Foundation Language Models. CoRR **abs/2302.13971** (2023). arXiv: 2302.13971
26. Wu, H., Barrett, C., Narodytska, N.: Lemur: Integrating Large Language Models in Automated Program Verification. In: The Twelfth International Conference on Learning Representations (2024). https://openreview.net/forum?id=Q3YaCghZNt
27. Yao, J., Zhou, Z., Chen, W., Cui, W.: Leveraging Large Language Models for Automated Proof Synthesis in Rust. CoRR **abs/2311.03739** (2023). arXiv: 2311.03739