LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling

Document Version
Accepted author manuscript

Link to publication record in Manchester Research Explorer

LLM-Generated Invariants for Bounded Model Checking Without Loop Unrolling

Muhammad A. A. Pirzada
muhammad.pirzada@manchester.ac.uk
The University of Manchester
Manchester, UK

Ahmed Bhayat
ahmed_bhayat@hotmail.com
The University of Manchester
Manchester, UK

Lucas C. Cordeiro
lucas.cordeiro@manchester.ac.uk
The University of Manchester
Manchester, UK

Giles Reger
giles.reger@manchester.ac.uk
The University of Manchester
Manchester, UK

ABSTRACT

We investigate a modification of the classical Bounded Model Checking (BMC) procedure that does not handle loops through unrolling but via modifications to the control flow graph (CFG). A portion of the CFG representing a loop is replaced by a node asserting invariants of the loop. We generate these invariants using Large Language Models (LLMs) and use a first-order theorem prover to ensure the correctness of the generated statements. We thus transform programs to loop-free variants in a sound manner. Our experimental results show that the resulting tool, ESBMC ibmc, is competitive with state-of-the-art formal verifiers for programs with unbounded loops, significantly improving the number of programs verified by the industrial-strength software verifier ESBMC and verifying programs that state-of-the-art software verifiers such as SeaHorn and VeriAbs could not.

KEYWORDS

Program Verification, Large Language Models, Bounded Model Checking, Invariant Generation

1 INTRODUCTION

Bounded Model Checking (BMC) is a well-established technique for hardware and software verification [7, 6]. Broadly, bounded model checkers work by exhaustively checking that no path through a program leads to an error state [18]. Loops are typically handled by unrolling. Thus, loops that cannot be statically bounded or have large static bounds pose problems for software model checkers [40,

23, 27]. In practice, for code involving loops with non-trivial bounds, BMC can find bugs, but cannot prove correctness. The main alternative to loop unrolling in software verification is to use a *loop invariant* to represent the program states that hold for all loop iterations. Loop invariants may be supplied manually [25] or discovered automatically [2]. Once these invariants are computed, they can be used for various applications, including proving correctness, code optimization and refactoring, debugging, and testing.

In the context of BMC, a loop can be replaced by a suitable invariant, resulting in a loop-free program, which classic BMC techniques can then handle using Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) solvers. The challenge here is to synthesize suitable invariants [51]. This has long been an active area of research, and various approaches exist to discover invariants. These include abstract interpretation [19], predicate abstraction [24], constraint solving [14, 3], and dynamic analysis [46]. However, none of these methods have become established, and most are limited to some particular class or domain of loops and fail to generalize to arbitrary programs.

With Large Language Models displaying emergent code generation and auto-formalization abilities [15, 57], there has been a surge in interest in their ability to generate invariants [55]. However, invariants generated by LLMs are unlike those generated by abstract interpretation, as there are no guarantees that the candidate formulas generated by an LLM are indeed invariants. Thus, relying on candidates returned by an LLM can easily lead to *unsoundness*. Hence, we combine an LLM with a first-order theorem prover. In particular, we use the superposition-based prover Vampire [38] to try and prove that candidate invariants are truly invariant. Candidates for which no proof is found are discarded.

To summarise, we attempt to complement BMC's existing strengths in bug finding by strengthening its verification abilities.. We achieve this as follows. We take a C program and use an LLM to annotate all loops with candidate loop invariants. The annotated program is then passed to the ESBMC verifier [42]. During symbolic execution, when ESBMC reaches a loop, instead of attempting to unroll it, ESBMC calls Vampire to prove the candidate invariants. If Vampire successfully proves one or more invariants, the loop is replaced by a formula asserting the proven invariants, and symbolic execution continues as normal. If the execution reaches an assertion, ESBMC attempts to prove it as normal i.e. by translation to SMT as described in Section 2.4.

Contributions. The main contributions of this paper are:

(1) A novel extension of BMC that incorporates automated loop invariant generation.
(2) The use of LLMs to produce invariant candidates for BMC of software without loop unrolling.
(3) An extension of our techniques to *quantified* invariants enabled by interfacing with the full first-order prover Vampire.

2 BACKGROUND AND RELATED WORK

2.1 Large Language Models

Large Language Models are a form of deep learning model based on the transformer architecture that has shown proficiency at text generation and language processing tasks [9]. They are commonly trained on large datasets and consist of billions of parameters. LLMs can be fine-tuned for performance on particular tasks, though as the model size grows, fine-tuning becomes less important [12]. Many LLMs have been developed, from the well-known ChatGPT [56] to Claude and Gemini.

It is known that the prompt provided to an LLM is crucial to the quality of the response received. It is also known that LLMs respond differently to prompts [10]. This has given rise to the field of *prompt engineering* [29]. The two best-known approaches are the chain of thought [53] and the tree of thought [58].

Chain of Thought. This approach guides the LLM in formulating an increasingly accurate answer with a more refined and detailed input prompt [59, 58]. White et al. [54] outline four main components that can be used to construct this type of prompt:

- *Input Semantics* – what the input will look like, e.g., a custom grammar and any associated rules.
- *Context Control* – to provide extra context to the LLM, such as a specific solved example of what you would like the LLM to do, thereby increasing the likelihood of the LLM generating more accurate responses to increase
- *Output Customisation* – how the output should be presented, i.e., is there any template that the LLM can populate with its response?
- *Error Identification* – for resolving the errors in the LLM generated responses, the improvement of the respective prompt – which is refining the question for a more appropriate answer (potentially with various approaches).

Tree of Thought. In contrast, in the Tree of Thought approach, queries are represented as a decision tree, and the responses are traversed in a breadth or depth-first manner [58]. The traversal undergoes a purification stage where only the relevant responses are taken to formulate an overall response. This is ideally used for more complicated problems that may require a larger, more complex, and, in some cases, verbose output [58, 41]. This is used for larger models like GPT-4 and GPT- Turbo - as they have a significantly higher context window of 8K to 128K tokens.

2.2 Program Verification

Broadly, the field of software verification can be split into two main categories. Techniques that require user input, such as those based on interactive theorem proving [49, 47] or those that require user

annotations of loop invariants [34, 1], and techniques that are fully automated such as Bounded Model Checking (BMC) [6], abstract interpretation [19] and symbolic execution [37]

BMC, in particular, has become an effective and popular verification technique. For loop-free programs, model checkers harness the surprising effectiveness of SAT and SMT solvers [28] to explore all program paths and either uncover bugs or verify safety and correctness. We discuss BMC in more depth in Section 2.4. However, as mentioned in Section 1, and in common with other techniques, BMC struggles with loops that cannot be statically bound or with large bounds [27]. Classic BMC works by unrolling loops, which does not scale well. A common method of handling loops is to use inductive loop invariants. An inductive loop invariant (or just *loop invariant* for short) is a logical assertion that holds at a loop head whenever program execution crosses this location. Synthesizing loop invariants is perhaps the main challenge in program verification [50] and has uses in other areas such as static analysis [43].

Due to the issue's centrality, various approaches have been explored. One approach is to solve a set of recurrence relations using a constraint solver. This is the approach taken by Ji et al. [36] and Amrollahi et al. [3], for example. The downside of the approach is that it is restricted to particular classes of loops. In the case of Ji et al., their approach is restricted to affine loops, while Amrollahi et al. target a generalization of the solvable loops.

Another popular approach is dynamic invariant generation [22, 45]. Dynamic generation proceeds by generating a large number of invariant templates. Actual program runs are then used to filter the templates. The remaining templates can then be passed to a verification engine to ensure they are true invariants. The downside of this approach is that, again, it is limited by the form of the pre-selected templates.

Abstract interpretation [19, 20] is also widely applied. However, abstract interpretation methods are limited by the domain of abstraction. There are many other techniques, such as Craig interpolation [31] and the IC3 approach [11]. However, they all tend to be limited in the form of the loop they can handle or the form of the invariant they can produce.

This has led to interest in using neural techniques for invariant generation. Code2Inv [52] uses deep reinforcement learning to synthesize invariants as part of a counterexample-guided inductive synthesis loop. He et al. [32] use a neural net to remove redundant constraints introduced during abstract interpretation.

There has also been a surge of interest in using Large Language Models (LLMs) for verification. Janßen et al. [35] conduct exploratory work on getting LLMs to produce invariants. Pei et al. [48] use an LLM to predict whether a formula is likely to be a Daikon [22] generated invariant. Likewise, Chakraborty et al. [13] do not use an LLM to generate invariants directly but instead to rank candidate invariants. Finally, Wu et al. [55] is the work closest to ours. The authors use an LLM to generate candidate invariants, but unlike our approach, this is not integrated into an existing verification technique. We provide a more detailed comparison of our approach with the associated LEMUR tool in Section 6.1.

2.3 Representing Programs

In this work, we model the program to be analyzed as a state transition system corresponding to the program's control-flow graph (CFG). A node in the CFG represents either a (non-) deterministic assignment, a conditional statement, an assumption, or an assertion. An edge in the CFG represents a possible change in the program's control location. A CFG directly induces a transition system. We consider a labeled transition system (LSTS) $M = \langle S, I, T, L, F \rangle$ with states S, initial states $I \subset S$, transition relation $T \subset S \times S$, set of labels L, and a labeling function $F : S \to 2^L$. The CFG structure gives us S, I, and T. In addition, we consider programs with candidate loop invariants and form F such that it maps states that represent the head of a loop in the original program to the candidate invariants on that loop (and all other states to the empty set).

Example 2.1 (Running Example). Consider the C program in Listing 1. Figure 1 illustrates the LSTS for this C program where we omit the initial non-deterministic assignments. Only one state (i <= n) is labeled with a non-empty set (of candidate loop invariants).

Cycles in an LSTS can only be introduced by loops or recursive function calls in the original program. We assume programs are goto-free, so there is always a unique final state for each loop (directly reachable from the loop head).

2.4 Bounded Model Checking (BMC)

Given a transition system $M = \langle S, I, T, L, F \rangle$, a property ϕ, and a bound k, BMC unrolls the system k times and translates it into a verification condition ψ_k, such that ψ_k is satisfiable if and only if ϕ has a counterexample of length k or less [8]. The associated model checking problem is formulated by constructing ψ_k as follows:

$$\psi_k = \chi_k \land \bigvee_{i=0}^{k} \neg\phi(s_i), \qquad \chi_j = s_0 \in I \land \bigwedge_{i=0}^{j-1} T(s_i, s_{i+1})$$

As χ_j represents the executions of M of length j, the formula ψ_k can be satisfied if and only if, for some $j \leq k$, there exists a reachable state at step j in which ϕ is violated.

If ψ_k is satisfiable, then the satisfying assignment gives the values of the program variables to form a counterexample. If ψ_k is unsatisfiable, we can conclude that no error state is reachable in k steps or less. In this case, BMC techniques are incomplete because there might still be a counterexample longer than k. Completeness can only be ensured if we know an upper bound on the depth of the state space. This means that if we can ensure that we have already explored all the relevant states of the system, and searching any deeper only exhibits states that have already been verified [39]. This is trivial in the case of *loop-free* programs, as the upper bound is the maximum path through the program's transition system. In this work, we seek to transform a transition system into a loop-free one, moving any potential incompleteness into this transformation.

3 SIMPLIFYING LOOPS WITH INVARIANTS

This section introduces a method to transform an LSTS into a loop-free LSTS. This works by attempting to replace a loop with the set of loop invariants labeling the loop. The approach differs from other techniques that either verify each loop invariant or attempt to derive a set of valid invariants. Instead, we take a set of potential

Listing 1: Example C program (94.c from Code2Inv)

```c
int main() {
  int i = __VERIFIER_nondet_int();
  int j = __VERIFIER_nondet_int();
  int k = __VERIFIER_nondet_int();
  int n = __VERIFIER_nondet_int();
  __ESBMC_assume(k >= 0);
  __ESBMC_assume(n >= 0);
  i = 0;
  j = 0;
  __invariant(i >= 0);
  __invariant(j >= i);
  while (i <= n) {
      i = (i + 1);
      j = (j + i);
  }
  __VERIFIER_assert( ((i + (j + k)) > (2 * n)) );
}
```
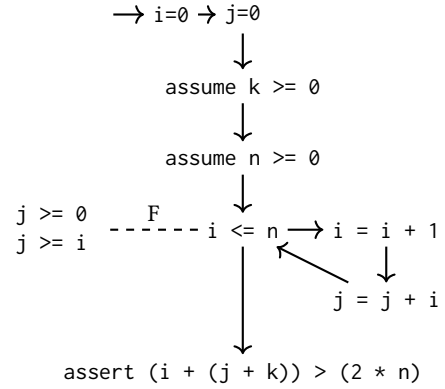

Figure 1: LSTS for program in Listing 1.

loop invariants and replace the loop with the subset that holds. This approach fits well with a setup that uses an LLM to generate likely invariants. While the transformed LSTS is not equivalent to the original LSTS, it soundly approximates it in the following sense. If for some property ϕ, there is no state reachable in the modified LSTS that satisfies ϕ, then there is no state reachable in the original LSTS that satisfies ϕ.

3.1 Transformation.

Let us consider a labeled state transition system $M[\pi]$ containing a sub-graph π that represents a loop, i.e., it has an initial state $h \in \pi$ such that $F(h) \neq \emptyset$ (h is a loop head) and an exit state $e \in \pi$ such that for any state $s \in \pi / \{h\}$ we have $F(s) = \emptyset$ (there are no nested loops as these are handled by transforming inner loops first) and both $T^*(h, s)$ and $T^*(s, e)$ where T^* is the transitive closure of T. Due to how M is constructed, the only cycles in π contain h. We rewrite $M[\pi]$ to $M[\{h, n, e\}]$, where n is some new state such that $T(h, n)$, $T(n, e)$ and n assumes a non-empty subset of invariants in $F(h)$ that hold (see below) as well as the negated loop condition. Repeating this process for all loop-representing sub-graphs yields

an acyclic transition system. In the remainder of this section, we write M' for the LSTS formed via the transformation just described from an LSTS M.

Spurious Counter-Examples. This transformation can trivially lead to spurious counter-examples. Even though tools such as ESMBC provide a verifiable counter-example when employing this transformation, the counter-example for an assertion after a loop is likely due to the over-approximation of the state space by the provided loop invariants rather than an actual bug. To understand why, consider the program P

```
int main() {
  int a,b = 0;
  __invariant(true);
  while (a < 10) { a++; b++ }
  __VERIFIER_assert(b != 0);
}
```

which will be (effectively) transformed into

```
int main() {
  int a,b = 0;
  __VERIFIER_assume(true);
  __VERIFIER_assume(a >= 10);
  __VERIFIER_assert(b != 0);
}
```

leading to a spurious counter-example e.g. `a = 10, b = 0`. Consider also our running example (Listing 1). If we only use the loop invariant `i >= 0`, then we can obtain the trivial counter-example `i = 2, n =1, j = k = 0` as we do not have the invariant `j >= i` ensuring `j` is large enough to make the assertion true.

This demonstrates that valid invariants may not fully capture the semantics of the loop. Hence, states not reachable in the original program M may be reachable in M'. For this reason, when considering soundness and completeness, we restrict our attention to proving program correctness.

Soundness. Informally, by *sound* we mean that no invalid assertions can be proven. That is, there are no false positives. The above transformation is sound. Given a desired property $\neg\phi$, if no state s satisfying ϕ is reachable in M' then no such state is reachable in M, i.e., we cannot say that M' has the property $\neg\phi$ if M does not. The transformation cannot remove reachable states as any invariant used to replace the loop must be true at the final state of the loop.

Completeness. By completeness, we mean that every valid assertion can be proven. The above transformation is incomplete. As explained previously, a set of invariants may fail to capture the complete semantics of a loop. In such cases, the transformation *over-approximates* the set of reachable states. Given a desired property $\neg\phi$, it is possible that no state satisfying ϕ is reachable in M while a satisfying state *is* reachable in M' leaving the property $\neg\phi$ false in M'. The transformation of P above is an example of this.

3.2 Implementation in ESBMC

We implemented[1] the loop replacement technique described above in the ESBMC verifier [42] using Vampire [38] to reason about loop

---

[1]Our code can be found at https://github.com/esbmc/esbmc/tree/ahmed-vampire-for-loops

invariants due to its ability to reason efficiently with quantifiers and theories. We chose to extend ESBMC - instead of other tools - as model checkers are very strong at finding bugs (showing unsafe behavior), and we wanted to improve their performance at proving safety, making them better general-purpose tools. Our approach is applicable to other tools and techniques since invariants are crucial to almost all imperative software verification methods. ESBMC uses an LLVM frontend to parse a C program and store it in an internal representation, which can be modeled using a CFG. It then uses symbolic execution to convert this CFG into a Single Static Assignment (SSA) form in which every variable is assigned at most once, and there are no loops. As mentioned, ESBMC handles loops during symbolic execution by unrolling them to some depth k. The SSA form induces a transition system that can be converted into a decidable fragment of first-order logical formulas, as Section 2.4 shows.

We modify ESBMC's symbolic execution procedure to handle loops differently. We use our running example to expound the changes. Let t represent the program before the loop. In SSA form (and ignoring variables that are not modified in the loop), t is:

```
i_0 = 0;
j_0 = 0;
```

When symbolic execution of the program reaches the loop head s, instead of unrolling, we replace the loop with the invariants $i \in F(s)$ that hold at the start of the loop (the base case) and on every step of the loop (the step case). For the base case, we form $|t|$, the logical representation of t, and attempt to prove that the program state at the loop head implies the invariant i i.e. we give the first-order problem $|t| \rightarrow i$ to Vampire.

For the step case, we must capture the fact that, when starting execution from some arbitrary iteration, we cannot assume anything about the value of the variables modified within the loop (the *loop variables*). To do this, we insert into the CFG nodes that havoc (set to non-deterministic values) the loops variables. In our running example, we insert the following assignments into the CFG before the loop head:

```
i_1 = __VERIFIER_nondet_int();
j_1 = __VERIFIER_nondet_int();
```

Let s be the loop head `i_1 <= n`. We then symbolically execute the loop body a single time to obtain b the loop body `i_2 = i_1 + 1; j_2 = j_1 + i_2;` from which we form $|b|$ its logical representation. We use Vampire to attempt to prove that $i \wedge |s| \wedge |b| \rightarrow i'$ where i' is i expressed in terms of program variables at the loop end. In our running example i could be `i_1 >= 0` in which case i' would be `i_2 >= 0`.

We apply the transformation to all invariants $i \in F(s)$ such that the base case and step case hold. As we do this iteratively, we can strengthen the step case by adding all invariants already proven to the antecedent. Since they have already been proven invariant, assuming they hold at the start of an arbitrary loop iteration is safe. In our running example, if we prove that $i \geq 0$ is invariant, we can assume this when attempting to show that $j \geq i$ is invariant holds.

This raises the interesting question of which order to attempt step case proofs, as this has a material impact on what can be proven. We iterate to avoid making an arbitrary decision until no
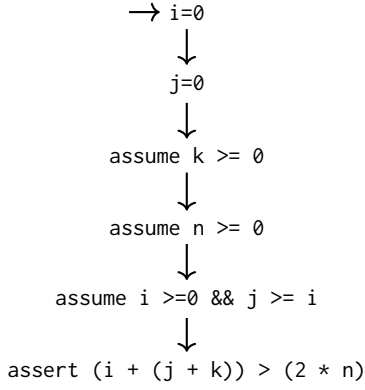
```
              → i=0
                ↓
               j=0
                ↓
          assume k >= 0
                ↓
          assume n >= 0
                ↓
      assume i >=0 && j >= i
                ↓
    assert (i + (j + k)) > (2 * n)
```

Figure 2: Simplified transition system from LSTS in Figure 1.

new invariants are proven. So, for example, we may attempt to prove $j \geq i$ first, fail, prove $i \geq 0$, and then go back to proving $j \geq i$, this time armed with the invariant $i \geq 0$. Checking through all permutations of invariant candidates can, in theory, lead to exponential proof attempts. In practice, invariants can often be proven independently of each other, reducing the search space, and a small number of passes often suffices.

Once we have a set of invariants, we unwind symbolic execution before the nodes used to havoc loop variables, insert the invariants as assumptions, and continue with symbolic execution. If no loop invariants are found, we abort rather than unrolling the loop, as unrolling would be unsound in this context.

3.3 Illustrative Examples

Firstly, let us consider our running example in Figure 1. Vampire can prove both invariants to hold (base and step case). The simplified loop-free transition system is given in Figure 2. ESBMC verifies this program as it can quickly conclude that no bad states are reachable in the loop-free transition system.

Secondly, consider the C program given in Figure 3 along with its LSTS and the simplified transition system given by the above process. As outlined by Sharma et al. [50], this program is difficult to generate invariants due to the split case. Once we have invariants, we can use Vampire to prove the base and step cases and replace the loop, leading to a loop-free state transition system, which is quickly verified.

The work was relatively straightforward in both cases once the appropriate invariant had been identified. The question remains: *can we generate such invariants automatically using LLMs?* We can ask an LLM for a certain number of invariants that can be used to prove an assertion following a loop. Figure 4 provides some of the conversations (we omit some explanation) we had with an LLM (ChatGPT 3.5) to identify the loop invariants we have used in this second example (the invariants for the running example were also obtained via LLM). This validated that the LLM can generate exactly the loop invariants we need, so how do we mechanize this process to apply automatically at scale?

```c
int main()
{
  int x = 0;
  int y = 50;
  __invariant(0 <= x && x <= 100);
  __invariant(x <= 50 && y == 50
          || x > 50 && y == x);
  while (x < 100) {
    x = x + 1;
    if(x > 50){
      y = y + 1;
    }
  }
  __VERIFIER_assert(y == 100 );
}
```
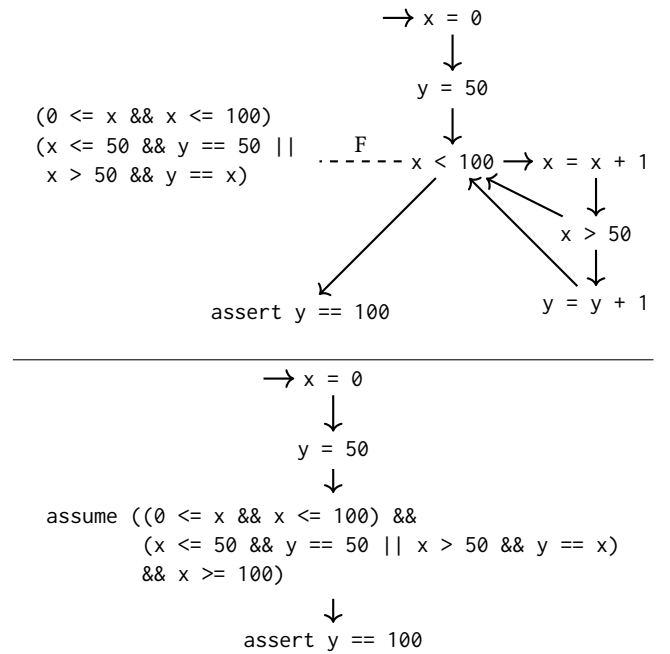
```
                        → x = 0
                           ↓
                         y = 50
(0 <= x && x <= 100)        ↓
(x <= 50 && y == 50 ||   F
 x > 50 && y == x)    - - - - x < 100 → x = x + 1
                          ↙        ↑↖        ↓
                                          x > 50
                                             ↓
        assert y == 100          y = y + 1
```

```
              → x = 0
                 ↓
              y = 50
                 ↓
  assume ((0 <= x && x <= 100) &&
      (x <= 50 && y == 50 || x > 50 && y == x)
      && x >= 100)
                 ↓
          assert y == 100
```

Figure 3: C program (top), its LSTS (middle), and the transformed LSTS (bottom).

4 GENERATING USEFUL INVARIANTS WITH LLMS

The previous section described a method to remove invariant-annotated loops but relied heavily on the quality of loop invariants. Here, we propose a method for using LLMs to suggest useful loop invariants.

4.1 Approach

The strongest invariants we could generate would lead to the same set of k-reachable states. This is too strong. Given the task of proving that the program satisfies a given property $\neg\phi$ it would be sufficient if a state satisfying ϕ were reachable in the original program if and only if it were reachable in the transformed program

```
Provide two loop invariants for the loop in the
program that are strong enough to prove the
assertion at the end of the program.

Answer:
Invariant 1: 0 <= x && x <= 100

This ensures that x is always within the expected
range throughout the loop's execution.

Invariant 2: x <= 50 && y == 50 || x > 50 && y == x

This captures the relationship between x and y:
If x is less than or equal to 50, y remains 50.
If x is greater than 50, y increments in tandem with
x, effectively making y equal to x.
```

Figure 4: ChatGPT 3.5 conversation to generate lopo invariants in Figure 3.

(the if direction giving completeness and the only-if direction giving soundness). Identifying such a set of invariants is, in general, undecidable.

However, we do know that loop invariants used in program verification tend to follow common patterns, as discussed by Furia et al. [26] in their systematic study. They classify loop invariants as either being a bounding invariant, placing some bound on key terms, or an essential invariant, as some mutation of the postcondition. Both observations suggest that it is likely that LLMs may be able to infer likely invariants from other examples (as bounding invariants follow common patterns) or in a predictable way from the given assertion. It also suggests that we may need to infer multiple invariants for a single loop.

4.2 Initial Invariant Generation Pipeline

We constructed an invariant generation pipeline to generate and use invariants to verify C programs. This is illustrated in Figure 5 (the filtering box in dashed lines is introduced later).

Labelling Loops. The first step in the pipeline identifies all loops within the program and inserts a comment, e.g., \\Line A, before each loop. This allows the prompt to use this comment to refer to particular loops as previous work [55] noted that LLMs are not good at reasoning about line numbers.

Prompt. Following the chain-of-thought approach outlined in Section 2.1, the LLM prompt consists of three parts. Firstly, some worked examples are given where a C program is provided with (a) correct invariant(s) and an explanation for why these invariants hold. In this case, we use the LLM itself to generate the explanation. Secondly, the C program to be verified is provided (with loops labeled as above). Finally, the query asks for an invariant in a particular format. The template for this prompt is given in Figure 6

It is important to note that there is a certain degree of non-determinism in LLM-generated responses - regardless of how meticulously crafted the prompt is - the LLM is more than likely to generate some random responses. There is, therefore, a trade-off. If the prompt is too weak, then the LLM may not have enough information. However, if the prompt is too long, that might lead the LLM to some level of confusion on easier tasks; finding the balance in the prompt is the goal of ensuring the best responses are obtained.

Verification. Once the LLM returns the invariant(s), the C program is modified to replace the label(s) with the associated invariants. The resultant C file is passed to the modified ESBMC described in Section 3 for verification via invariant-based loop simplification.

Iteration. The LLM may not produce usable (syntactically correct) or useful (resulting in successful verification) invariants from a single prompt. We can increase the likelihood through careful crafting of the prompt and choice of LLM. Still, ultimately, we must ensure that the LLM has some latitude to generate incorrect responses - while ensuring that time and resources do not get wasted. Additionally, some programs may require multiple invariants, and it is unclear how many to ask for. We address both issues by iteratively retrying the prompt, increasing the number of invariants to ask for after every n attempts.

We always run with a maximum iteration limit. If an open-source LLM is used, then the number of iterations is not a bounded resource - as they are free to use. However, if closed-source LLMs such as GPT are used, then every LLM-based prompt and the respective generated response has some financial cost depending on the token usage and limit per request, so assigning a limit is necessary.

4.3 Pipeline Modifications

After establishing the initial pipeline, experimentation led to further modifications.

Filtering Suggested Invariants. We observed that generated invariants often led to parsing errors during verification. There were two main classes of parsing errors:

(1) The LLM generated incorrect C code, and
(2) Hallucinating variables, i.e., generating an invariant over a program variable that did not exist in the program.

Class 2, hallucination, occurred rarely during our experiments and did not appear to impact the result significantly. We inspected the invariants generated on all failed runs, and no failures were due to hallucination issues. Most issues fell into the first class, and we focused on this. Instead of providing further constraints in the prompt (i.e. attempting to teach the LLM what a correctly formed invariant should look like), we performed a lightweight syntactic check on the generated invariants. This utilized a relatively constrained regular expression

`__invariant\((\w|\s|\|\||&&|!|=|<|>|\?|:|\(|\))*\);`

to limit the invariants used for verification. Each invariant is checked against the pattern and discarded if it does not match. This allows an iteration to proceed even if some generated invariants are syntactically invalid.

Reducing Distractions in the Prompt. A separate hypothesis to explain the number of parsing errors was that the explanations
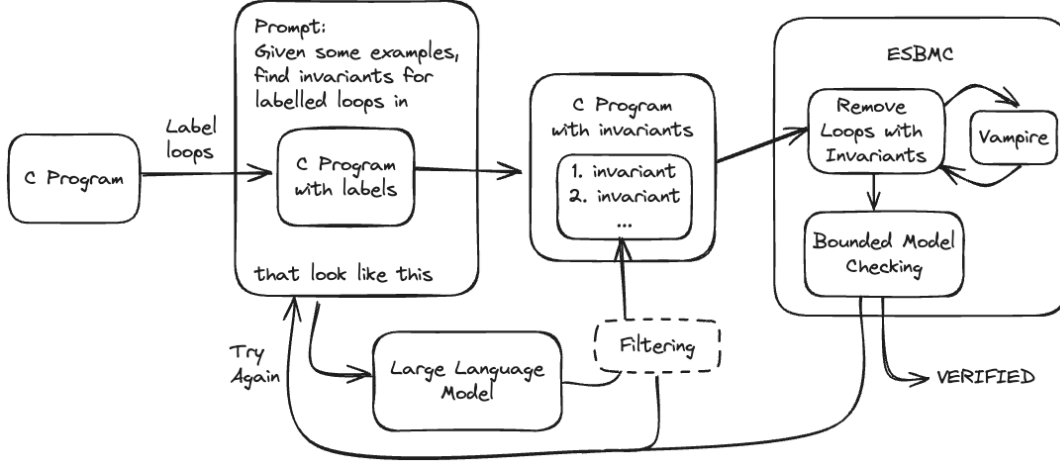
Figure 5: Illustration of how ESBMC ibmc works.

```
<Examples with explanation>


Based on these examples provided above can you
generate  a C invariant for the following code,


<Labelled Input C Program>


Print an invariant for this loop that holds in the
form '__invariant(...);'. They should help prove the
assertions. You can utilise '&&' or '||' if required.
No explanation.  Your answer should be in the form
'__invariant(...);'
```

Figure 6: Prompt template for ESBMC ibmc.

provided in the prompts could cause potential interference with the performance of the invariant generation. It provides inputs that may look like invariants but are formatted as explanations. We created an alternative prompt to eliminate this noise source that removed the explanations from examples. This led to a simplified chain of thought prompt engineering approach.

Combined Prompt. Early experimentation demonstrated that, in some cases, the full prompt with all examples was necessary to produce the correct invariant. Still, in other cases, its distraction prevented any useful invariants from being produced. To get both benefits, we included a pipeline that attempted both prompts one after the other (in terms of iteration limits, both calls count as one iteration).

4.4 Implementation

We have implemented the above pipeline and its extensions as a wrapper around the extended ESBMC described in Section 3 and call the result *ESBMC ibmc.* The above options give us six variants of the tool where we (i) either choose to filter the generated invariants or not and (ii) choose a full, constrained, or combined prompt. We

limit the number of generated invariants to 3 and increase this number every 5 iterations (returning to 1 every 15 iterations).

For the LLM, we select OpenAI's ChatGPT 3.5 Turbo-Instruct. Significant research has been conducted comparing various LLMs for code generation tasks [21, 44, 60]. Consistently, OpenAI's GPT and Llama top the performance charts, hence our decision to go with ChatGPT. Zheng et al. [60] show that GPT-3.5 is good at performing certain code generation tasks in Java and is also very capable at doing so in Python. Alongside this, the series of tasks conducted by Nejjar et al.[44] show that GPT-3.5 and GPT-4 achieve 4 stars and, in some cases, 5 stars for certain code generation tasks. Additionally, Du et al.[21] show that GPT-3.5 and GPT-4 significantly outperform all the other models in their study for "class-level coding tasks with both greedy sampling and nucleus sampling". All of this lead us to select GPT as the LLM for integrating this tool.

However, the reason for choosing GPT-3.5-Turbo-Instruct was not just a model choice. It was a cost-effective choice, and the model is fast. Specifically, the instruct model can generate concise answers based on instructions and detailed context.[2] [3]

5 EXPERIMENTAL ANALYSIS

Our experimental analysis of ESBMC ibmc aims to address the following research questions.

RQ1. Can our chosen LLM (GPT-3.5-Turbo-Instruct) generate invariants useful for program verification?

RQ2. Is it better to use more or less explanation in the chain-of-thought prompt?

RQ3. Should we always syntactically filter invariants?

RQ4. Should we ask the LLM for fewer or more invariants?

RQ5. Does ESBMC ibmc verify programs that cannot be verified by the state-of-the-art?

5.1 Experimental Setup

Our experiments use the following setup.

Table 1: Results from option comparison. Unique solved benchmarks concerning companion experiments in parenthesis. Time (in seconds) and Iterations are for successful runs. err gives mean iterations with parsing errors.

	Solved	Time		Iterations		
		mean	max	mean	max	err
			Full Prompt			
no-filter	74 (18)	59.7	316.0	10.8	31	7.8
filter	62 (9)	71.0	273.1	11.8	30	0.2
			Constrained Prompt			
no-filter	64 (8)	59.7	231.2	9.3	30	6.3
filter	65 (11)	49.1	352.4	7.8	29	0.1
			Combined Prompt			
no-filter	79 (15)	109.1	643.5	17.6	59	13.2
filter	77 (13)	98.0	567.2	14.8	58	0.2

Benchmarks. The benchmarks are from the code2inv benchmark set[4] developed for learning-based approaches to invariant generation [51]. We added macros for assertions and assumptions specific to each verification tool to prepare the programs for our experiments.The benchmarks contain one loop and the biggest program contains 53 lines of code. The updated files are available at our repository.[5]

Resources. The machine used to run all experiments is a Macbook Pro with a 2.8 GHz Quad-Core Intel Core i7 Processor and 16 GB memory. Experiments were run with a time limit of 600 seconds and ESBMC ibmc was run with an iteration limit (for invariant generation) of 30 iterations per benchmark. However, we used 60 iterations/20 minutes for combined prompts primarily to see how many benchmarks exceeded 600s and do any hit the timeout of 1200s or not. However, this didn't help much; Table 1 shows the maximum solution time was 645s. [6]

5.2 Comparing ESBMC ibmc Options

To address RQ1-3, we compared the six variations of ESBMC ibmc. The results are given in Table 1. We discuss each Research Question.

RQ1: Can our chosen LLM (GPT-3.5-Turbo-Instruct) generate invariants useful for program verification? For this work, which focuses on a single loop and a goal, invariant usefulness is directly related to whether it can be used to verify the given program, i.e., whether it's related to the goal. Overall, ESBMC ibmc verifies 101 out of 133 programs (across all options), and the initial pipeline (Full Prompt/no-filter in Table 1) verifies 74 programs. This confirms that the LLM can generate useful invariants for program verification. As a baseline, only 10 of these programs could be verified by ESBMC's standard bounded-model checking algorithm.
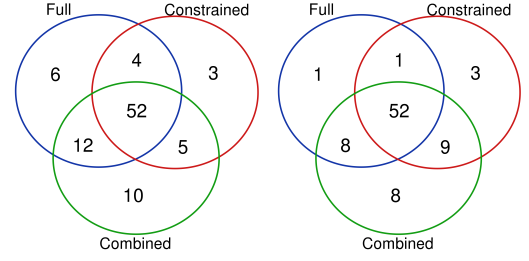

Figure 7: Comparing programs verified for the three prompt styles for no filtering (left) and filtering (right).
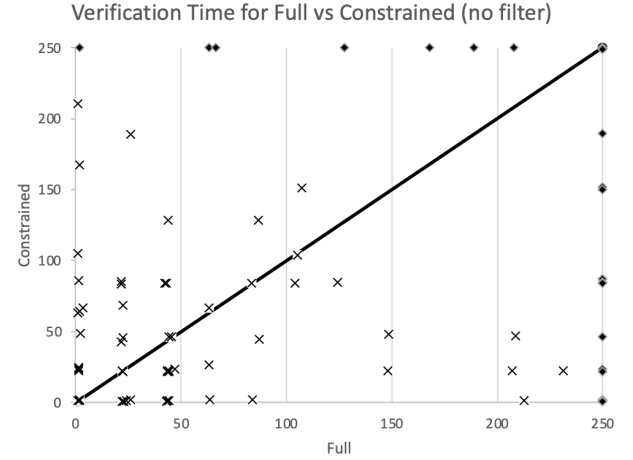

Figure 8: Comparing verification time (seconds) for Full vs Constrained prompts (no filter).

Performance would likely improve with a more recent model. Recall that one reason for the current choice was cost and validating the technique with a cost-effective LLM is useful as cost is likely to be important to verification efforts. Future work will extend experiments to recent LLMs (including open source).

> The answer to RQ1 is that our LLM generated many invariants that were used to verify programs that could not be verified (by ESBMC) without them.

RQ2: Is it better to use more or less explanation in the chain-of-thought prompt? We answer this question by comparing the Full vs Constrained vs Combined approaches in Table 1. Figure 7 (left) shows the distribution of problems solved by each approach without filtering.

The first observation is that there is no clear-cut advantage to more explanation (Full) or less explanation (Constrained) in the chain-of-thought prompt. Although without filtering, more explanation is beneficial in general. Figure 8 plots verification time in the Full and Constrained prompts. It shows that many programs not verified using one prompt were hard to use the other.

We might expect the Combined prompt to solve the union of the Full and Constrained prompts. However, from Figure 7 (left), we can see this is not the case. The simple explanation here is that running both prompts takes longer, and programs verified towards the end of the time limit for a single prompt may not be verified when using the Combined prompt.

> The answer to RQ2 is that more explanation is better overall but can also be distracting, so a combination is best. This also suggests that further experimentation with the amount of explanation could yield improved results.

RQ3: Should we always syntactically filter invariants? Table 1 records the mean number of iterations skipped due to parsing errors (err column). For non-filtered experiments, this ranges between 67% - 75% of iterations wasted due to the LLM producing syntactically incorrect invariants. This should suggest that removing these would improve performance, but while it mostly eliminates parsing errors, the performance improvements are less clear. For the Full prompt, we see a drop in verified programs and, more interestingly, a large gap in the symmetric difference (of 27 programs). This suggests that (i) the syntactic filtering may be over-constrained (i.e. it rejects useful invariants) and that (ii) allowing many more iterations to proceed increases the chances of finding a useful invariant (from within the constraints). We see a similar symmetric difference for the Constrained prompt, although, in this case, filtering leads to more programs being verified. This suggests that the more complex invariants that the filtering discarded for the Full prompt came from the additional explanation in that prompt.

> The answer to RQ3 is that filtering significantly reduces parsing errors. Still, the current implementation does not cover all cases, and the addition of filtering restricts the freedom of LLM-generated responses. This suggests that exploring techniques that relied more on guiding the LLM to generate syntactically correct invariants rather than post-filtering would be better.

Summary. We have demonstrated that ESBMC ibmc can consistently solve around 58% of the programs in our benchmark set (using the combined approach). However, there is still work to be done to understand how best to combine different techniques – only 39 programs (29%) are verified by all 6 option combinations, but 101 programs (76%) are verified by at least one.

While we didn't report results from multiple runs, we observed that the LLM was stable in the invariants it returned for a program across 30 iterations (e.g., the order might change, but the actual invariants less so), leading to stable results.

5.3 How many Invariants to Ask For

To address RQ4, we first examine how many invariants were generated on successful runs in our previous experiments. This information is given in Table 2. This shows that many programs were verified when 3 invariants were generated, although ESBMC may not have needed all 3 to verify the program. Non-determinism of

Table 2: Generated invariants used for successful verification.

	no-filter			filter		
	Full	Constr.	Comb.	Full	Constr.	Comb.
1	37	35	47	25	36	56
2	23	19	17	23	18	14
3	14	10	15	15	12	8

Table 3: Comparing ESBMC ibmc with other state-of-the-art verification tools.

Tool	Solved	Unique
ESBMC bmc	16	0
ESBMC k-induction	46	0
SeaHorn	115	10
VeriAbs	73	2
ESBMC ibmc	101	2

the LLM means that a single useful invariant could be generated at any point. Approaches that remove noise tend to use (slightly) fewer invariants, suggesting that noise reduction is in tension with generating more complex invariants.

Note that it is impossible to judge how close we are to the 'minimum' number of invariants required since we have no ground truths for the benchmarks. We found that at least half of the solved benchmarks needed more than 1, but only 1 required more than 3.

Next, we run an additional experiment asking the LLM to provide "up to 20" invariants on every iteration using the constrained prompt and syntactic filtering. The first observation is that the LLM was not very good at generating many different invariants. The most unique invariants returned was 10, but the mean was 2.6. This approach verified 61 programs, including 1 not verified by previous approaches. The time and iterations required (105.2 s and 9.8 s on average) are significantly worse than when asking for fewer invariants.

> The answer to RQ4 is that the approach taken in ESBMC ibmc (asking for a few invariants and increasing this limit if no useful invariants are produced) seems preferable to asking for a large set every time.

5.4 Comparing ESBMC ibmc with the State-of-the-Art

To address RQ5, we compare ESBMC ibmc to three state-of-the-art program verification tools. The results are given in Table 3 and Figure 9. As before, all experiments here ran for 600 s. We compare other tools against the hypothetical "best" version of ESBMC ibmc that uses the right option for each benchmark.

Comparing with ESBMC modes. We compare ESBMC ibmc with running ESBMC (v7.1) in the standard bounded model checking (bmc) mode, which simply unrolls loops (as described above), and the *k*-induction mode (kind), which implements three different cases called *base case*, *forward condition*, and *inductive step* [27]. For *k*-induction, we set options for interval analysis and the addition of
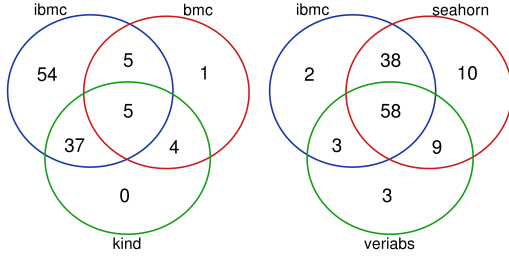
Figure 9: Comparing programs verified by ESBMC ibmc compared to other ESBMC modes (left) and other tools (right).
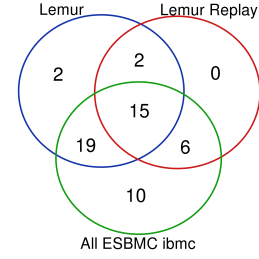

Figure 10: Comparing programs verified by Lemur, our replay of Lemur using Lemur's invariants, and ESBMC ibmc on programs that Lemur generated invariants for.

symbolic execution value sets, running with options −k-induction −interval-analysis −add-symex-value-sets. Together, these modes verify 5 programs not handled by ESBMC ibmc with bmc verifying one program (*2.c*) not handled by *k*-induction by unwinding the loop 1000 times. This suggests that ESBMC ibmc could benefit from running these modes before invoking the LLM.

Comparing with Other Tools. We compare ESBMC ibmc against VeriAbs (v1.5.1) [16] and SeaHorn (v10.0.0) [30]. VeriAbs is a portfolio of 12 verification techniques that uses slicing and strategy selection to identify which techniques to apply to (sub)-programs. SeaHorn uses abstract interpretation to compute inductive invariants and performs verification via a Horn-clause encoding and subsequent constraint solving. For SeaHorn, we ran using options pdf −m64, and for VeriAbs, we ran using options −sv22 −property-file unreach-call.prp −32. SeaHorn verifies most programs with VeriAbs verifying fewer than many of the individual runs of ESBMC ibmc. Each tool verified some programs uniquely whilst only 58 programs were solved by all tools.

> The answer to RQ5 is that ESBMC ibmc significantly improves the number of programs verified by ESBMC and verifies programs that state-of-the-art software verifiers such as SeaHorn and VeriAbs could not.

6 DISCUSSION

6.1 Comparison with LEMUR Framework

The LEMUR Framework by Haoze et al. [55] is similar to our work in that they use an LLM to generate program invariants and use ESBMC [42] and UAutomizer [33] to verify programs. We compare our two approaches and compare results.

Comparing Approaches. LEMUR introduces a framework for automated program verification which decomposes the verification task into sub-goals and utilizes an LLM to complete some creative sub-goals (generating and repairing loop invariants) and verification tools ESBMC [42] and UAutomizer [33] to complete verification sub-goals. This approach differs from ours as it is reactive. If a particular invariant is not useful, the framework can attempt to repair it. Another difference is that as it does not need to reason about invariants directly, it can easily handle complex programs. However, as the framework treats verification tools such as ESBMC as black-boxes and communicates via assertions and assumptions

(instead of rewriting the program as we do), it is limited in how much advantage it can take of the generated invariants.

The work uses ChatGPT-4, in contrast to our choice of ChatGPT 3.5 Turbo-Instruct, which (most likely) impacts the framework's performance (although at a greater cost). They did not have the same issues with generating syntactically incorrect invariants, possibly due to the LLM choice. They noted that the LLM generated invariants that used operators not present in the program being verified, demonstrating its ability to generalize well. We had a similar experience with ChatGPT 3.5 Turbo-Instruct.

The LEMUR framework took a different approach to prompt engineering. Instead of following a Chain of Thought approach (recall that this involves providing examples and explanation of the task to solve), they followed a more chat-like interaction approach while also using in-context learning, i.e., encouraging the LLM to format the output in a specific way.

Comparing Results. The LEMUR framework was evaluated on the Code2inv benchmarks used in this work. As their pipeline handled multiple loops (a non-fundamental engineering-related limitation of how we currently prompt the LLM), they could run SV-COMP [4] benchmarks with short reachability. We were unable to directly compare LEMUR with ESBMC ibmc on identical hardware, since we failed in multiple attempt to set up LEMUR on different platforms. Additionally, LEMUR requires costly LLM queries to run. In place of a direct comparison, we extracted all invariants generated by LEMUR in its initial call to its LLM and replayed these using our extended version of ESBMC, positively answering the question *Would LEMUR benefit from this extension?*

Lemur generated invariants for 65 out of 133 programs (after verifying 42 benchmarks by direct call to ESBMC *k*-induction or UAutomizer) and verified 38 of these, leaving 27 unverified. All 65 programs were annotated with invariants and passed to ESBMC ibmc. Figure 10 illustrates this result and includes the results of using ESBMC ibmc to generate the invariants. Using our extended ESBMC, we re-verified all but 2 of the programs LEMUR verified and verified an additional 6 using LEMUR's invariants. Further, there were 10 programs that ESBMC ibmc verified using only the invariants it produced. These results suggest that (i) ESBMC ibmc could benefit from some of the invariant generation used in LEMUR and (ii) LEMUR could benefit from our ESBMC extension.

6.2 Extending to Quantified Invariants

While proving difficult for existing verification techniques, our running example can, in principle, be handled by current approaches. Current techniques are good at synthesizing quantifier-free, conjunctive invariants [50]. Where a quantified invariant is required, they are liable to struggle. An LLM is not limited in this way and can easily generate quantified invariants. By pairing an LLM with a full first-order prover that can reason about quantifiers, we ought to be able to verify more complex benchmarks that require invariants not fitting into the quantifier-free, conjunctive fragment. While this is ongoing work, the early results are promising. Consider the program in Listing 2. [7] While traditional approaches struggle with it (VeriAbs can verify the benchmark, but SeaHorn, ESBMC, and UAutomize are all unable to do so), ChatGPT returns the required invariant $\forall x. \, 0 \leq x < i \rightarrow res[x] = a[x] + b[x]$ at first asking. Vampire can easily verify the invariant's correctness and hence the assertion's correctness.

We are not limited to universally quantified assertions. Consider, for example, the program in Listing 3. Once again, the leading verifiers (including VeriAbs in this case) cannot prove the existence of an index i such that $a[i] == 500$. On the other hand, ChatGPT immediately provides the necessary invariant $\forall x. \, 0 \leq x < i \rightarrow a[x] = x$. Vampire then easily proves the inductiveness of the invariant and then the assertion. There is nothing to prevent the usage of even more complex invariants, such as those containing quantifier alternations, since Vampire can handle full first-order logic. We can even extend to verifying functional programs since Vampire supports reasoning about higher-order functions [5]. That said, significant work remains to strengthen the combination of an LLM and a first-order prover. Vampire does not support reasoning about bit-vectors which restricts our ability to verify safety properties involving machine integers. Likewise, reasoning about pointer-based data structures is not currently supported since ESBMC has no support for expressing shape predicates [17], and it is unclear whether Vampire can reason about them effectively.

7 CONCLUSION

In this paper, we have explored using LLM-generated invariants to replace loops in the context of BMC with promising results. On programs that contain unbounded loops, our tool ESBMC ibmc outperforms the current best ESBMC strategy for such benchmarks, k-induction. Furthermore, our approach handled certain benchmarks that none of the current state-of-the-art verifiers can. Moreover, we have demonstrated that by pairing an LLM with a full first-order prover, we can go beyond quantifier-free invariants and handle invariants and assertions involving arbitrary quantification. That being said, our work also raises many interesting questions that we hope to address eventually.

Does the use of LLMs scale to larger, industrial-sized benchmarks? With LLMs featuring ever-increasing context window sizes, we

Listing 2: Example C program requiring universally quantified invariant

```c
int main() {
  int a[100000];
  int b[100000];
  int res[100000];
  int i = 0;
  int x;
  while (i < 100000)
  {
    res[i] = a[i] + b[i];
    i++;
  }
  __VERIFIER_assert(__forall(x, (!((x < i)
&& (x>= 0)) || (res[x] == a[x] + b[x])) ));
}
```

Listing 3: Example C program requiring eistentially quantified invariant

```c
int main() {
  int n =  1000000;
  int a[n];
  int i = 0;
  while(i < n){
    a[i] = i;
    i++;
  }
  __VERIFIER_assert(__exists(x, (a[x] == 500)));
}
```

hope this is feasible. However, we think that LLM performance on invariant generation can be improved in the absence of conjecture. We do not currently use the (symbolic) trace formed by unrolling a loop body. In theory, this data could be very helpful to an LLM or other learner to generate interesting invariants.

Does the use of LLMs remove the need for other invariant generation techniques? We think not, or at least not currently. An interesting line of work to explore is combining invariants generated by LLMs with invariants generated by other whitebox and blackbox techniques.

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. 2016. Deductive software verification-the key book. *Lecture notes in computer science*, 10001.

[2] Omar M. Alhawi, Herbert Rocha, Mikhail R. Gadelha, Lucas C. Cordeiro, and Eddie Batista de Lima Filho. 2021. Verification and refutation of C programs based on k-induction and invariant inference. *Int. J. Softw. Tools Technol. Transf.*, 23, 2, 115–135.

---

[7] Since ESBMC does not currently support an assertion language that contains quantifiers, we internally add special semantics to the standard C function `__forall` as a temporary solution. The programs in Listings 2 and 3 are slightly simplified versions of the actual programs verified, which can be found in our repository (see Footnote 5) Since the other verification engines we tested do not support quantified invariants, we had to simulate these using assertions within a loop. The version of the benchmarks without quantifiers can also be found in the repository.

[3] Daneshvar Amrollahi, Ezio Bartocci, George Kenison, Laura Kovács, Marcel Moosbrugger, and Miroslav Stankovič. 2022. Solving invariant generation for unsolvable loops. In *International Static Analysis Symposium*. Springer, 19–43.

[4] Dirk Beyer. 2024. State of the art in software verification and witness validation: SV-COMP 2024. In *30th International Conference Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Lecture Notes in Computer Science). Vol. 14572. Springer, 299–329.

[5] Ahmed Bhayat and Giles Reger. 2020. A combinator-based superposition calculus for higher-order logic. In *International Joint Conference on Automated Reasoning*. Springer, 278–296.

[6] Armin Biere. 2021. Bounded model checking. In *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications. Vol. 336. Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, (Eds.) IOS Press, 739–764.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems TACAS* (Lecture Notes in Computer Science). Vol. 1579. Springer, 193–207.

[8] Armin Biere, Marijn Heule, and Hans van Maaren. 2009. *Handbook of satisfiability*. Vol. 185. IOS press.

[9] Idan A Blank. 2023. What are large language models supposed to model? *Trends in Cognitive Sciences*.

[10] Víctor A. Braberman, Flavia Bonomo-Braberman, Yiannis Charalambous, Juan Gabriel Colonna, Lucas C. Cordeiro, and Rosiane de Freitas. 2024. Tasks people prompt: A taxonomy of LLM downstream tasks in software verification and falsification approaches. *CoRR*, abs/2404.09384.

[11] Aaron R Bradley. 2011. SAT-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 70–87.

[12] Tom Brown et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33, 1877–1901.

[13] Saikat Chakraborty, Shuvendu K Lahiri, Sarah Fakhoury, Madanlal Musuvathi, Akash Lal, Aseem Rastogi, Aditya Senthilnathan, Rahul Sharma, and Nikhil Swamy. 2023. Ranking LLM-generated loop invariants for program verification. *arXiv preprint arXiv:2310.09342*.

[14] Krishnendu Chatterjee, Hongfei Fu, Amir Kafshdar Goharshady, and Ehsan Kafshdar Goharshady. 2020. Polynomial invariant generation for non-deterministic recursive programs. In *Conference on Programming Language Design and Implementation (PLDI)*, 672–687.

[15] Mark Chen et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

[16] Bharti Chimdyalwar, Priyanka Darke, Avriti Chauhan, Punit Shah, Shrawan Kumar, and R. Venkatesh. 2017. VeriAbs: verification by abstraction (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems TACAS* (Lecture Notes in Computer Science). Vol. 10206. Springer, 404–408.

[17] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. 2007. Automated verification of shape, size and bag properties. In *Conference on Engineering Complex Computer Systems (ICECCS 2007)*, 307–320.

[18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, (Eds.) 2018. *Handbook of Model Checking*. Springer.

[19] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*. ACM, 238–252.

[20] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of programming languages (POPL)*. ACM, 84–96.

[21] Xueying Du et al. 2023. Classeval: a manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*.

[22] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69, 1-3, 35–45.

[23] Stephan Falke, Florian Merz, and Carsten Sinz. 2013. The bounded model checker LLBMC. In *Automated Software Engineering (ASE)*. IEEE, 706–709.

[24] Cormac Flanagan and Shaz Qadeer. 2002. Predicate abstraction for software verification. In *Principles of Programming Languages (POPL)*. ACM, 191–202.

[25] Robert W. Floyd. 1967. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19, 19–32.

[26] Carlo A Furia, Bertrand Meyer, and Sergey Velder. 2014. Loop invariants: analysis, classification, and examples. *ACM Computing Surveys (CSUR)*, 46, 3, 1–51.

[27] Mikhail Y. R. Gadelha, Hussama Ibrahim Ismail, and Lucas C. Cordeiro. 2017. Handling loops in bounded model checking of C programs via k-induction. *Int. J. Softw. Tools Technol. Transf.*, 19, 1, 97–114.

[28] Vijay Ganesh and Moshe Y Vardi. 2020. On the unreasonable effectiveness of SAT solvers. (2020). https://www.cs.rice.edu/~vardi/papers/SATSolvers21.pdf.

[29] Louie Giray. 2023. Prompt engineering with chatgpt: a guide for academic writers. *Annals of biomedical engineering*, 51, 12, 2629–2633.

[30] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification CAV* (Lecture Notes in Computer Science). Vol. 9206. Springer, 343–361.

[31] Arie Gurfinkel, Simone Fulvio Rollini, and Natasha Sharygina. 2013. Interpolation properties and SAT-based model checking. In *Automated Technology for Verification and Analysis*. Springer, 255–271.

[32] Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2020. Learning fast and precise numerical analysis. In *Conference on Programming Language Design and Implementation (PLDI)*, 1112–1127.

[33] Matthias Heizmann, Daniel Dietsch, Marius Greitschus, Jan Leike, Betim Musa, Claus Schätzle, and Andreas Podelski. 2016. Ultimate Automizer with two-track proofs - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Lecture Notes in Computer Science). Vol. 9636. Springer, 950–953.

[34] Bart Jacobs, Jan Smans, and Frank Piessens. 2010. A quick tour of the VeriFast program verifier. In *Programming Languages and Systems*. Kazunori Ueda, (Ed.) Springer Berlin Heidelberg, Berlin, Heidelberg, 304–311.

[35] Christian Janßen, Cedric Richter, and Heike Wehrheim. 2024. Can ChatGPT support software verification? In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 266–279.

[36] Yucheng Ji, Hongfei Fu, Bin Fang, and Haibo Chen. 2022. Affine loop invariant generation via matrix algebra. In *International Conference on Computer Aided Verification (CAV)*. Springer, 257–281.

[37] James C. King. 1976. Symbolic execution and program testing. In *Communications of the ACM* number 7. Vol. 19, 385–394. DOI: 10.1145/360248.360252.

[38] Laura Kovács and Andrei Voronkov. 2013. First-order theorem proving and vampire. In *International Conference on Computer Aided Verification (CAV)*. Springer, 1–35.

[39] Daniel Kroening, Joël Ouaknine, Ofer Strichman, Thomas Wahl, and James Worrell. 2011. Linear completeness thresholds for bounded model checking. In *International Conference on Computer Aided Verification (CAV)*. Springer, 557–572.

[40] Daniel Kroening and Michael Tautschnig. 2014. CBMC - C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (LNCS). Vol. 8413. Springer, 389–391.

[41] Jieyi Long. 2023. Large language model guided tree-of-thought. *arXiv preprint arXiv:2305.08291*.

[42] Rafael Sá Menezes et al. 2024. ESBMC v7.4: harnessing the power of intervals - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (Lecture Notes in Computer Science). Vol. 14572. Springer, 376–380.

[43] Markus Müller-Olm and Helmut Seidl. 2004. Computing polynomial program invariants. *Information Processing Letters*, 91, 5, 233–244.

[44] Mohamed Nejjar, Luca Zacharias, Fabian Stiehle, and Ingo Weber. 2023. LLMs for science: usage for code generation and data analysis. *arXiv:2311.16733*. arXiv preprint.

[45] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. Dig: a dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23, 4, 1–30.

[46] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2012. Using dynamic analysis to discover polynomial and array invariants. In *International Conference on Software Engineering (ICSE)*. IEEE, 683–693.

[47] 2012. Introduction to the Coq proof-assistant for practical software verification. *Tools for Practical Software Verification: LASER*. Springer, 45–95.

[48] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants? In *International Conference on Machine Learning*. PMLR, 27496–27520.

[49] Viorel Preoteasa, Ralph-Johan Back, and Johannes Eriksson. 2015. Verification and code generation for invariant diagrams in Isabelle. *Journal of Logical and Algebraic Methods in Programming*, 84, 1, 19–36.

[50] Rahul Sharma, Isil Dillig, Thomas Dillig, and Alex Aiken. 2011. Simplifying loop invariant generation using splitter predicates. In *Computer Aided Verification (CAV)*. Springer, 703–719.

[51] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems*, 31.

[52] Xujie Si, Aaditya Naik, Hanjun Dai, Mayur Naik, and Le Song. 2020. Code2inv: a deep learning framework for program verification. In *International Conference on Computer Aided Verification (CAV)*. Springer, 151–164.

[53] Jason Wei and Denny Zhou. 2022. Language models perform reasoning via chain of thought. (2022). https://research.google/blog/language-models-perform-reasoning-via-chain-of-thought/.

[54] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A

prompt pattern catalog to enhance prompt engineering with ChatGPT. *arXiv preprint arXiv:2302.11382.*

[55] Haoze Wu, Clark Barrett, and Nina Narodytska. 2024. Lemur: integrating large language models in automated program verification. (2024). arXiv: 2310.04870 [cs.FL].

[56] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. 2023. A brief overview of chatgpt: the history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica*, 10, 5, 1122–1136.

[57] Yuhuai Wu, Albert Qiaochu Jiang, Wenda Li, Markus Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. 2022. Autoformalization with large language models. *Advances in Neural Information Processing Systems*, 35, 32353–32368.

[58] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

[59] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. *arXiv preprint arXiv:2210.03493.*

[60] Zibin Zheng, Kaiwen Ning, Yanlin Wang, Jingwen Zhang, Dewu Zheng, Mingxi Ye, and Jiachi Chen. 2023. A survey of large language models for code: evolution, benchmarking, and future trends. *arXiv preprint arXiv:2311.10372.*