Clover: Closed-Loop Verifiable Code Generation

Chuyue Sun[1(✉)], Ying Sheng[1], Oded Padon[2], and Clark Barrett[1]

[1] Stanford University, Stanford, USA
{chuyues,ying1123}@stanford.edu, barrett@cs.stanford.edu
[2] VMware Research, Palo Alto, USA
oded.padon@gmail.com

Abstract. The use of large language models for code generation is a rapidly growing trend in software development. However, without effective methods for ensuring the correctness of generated code, this trend could lead to undesirable outcomes. In this paper, we introduce a new approach for addressing this challenge: the Clover paradigm, short for Closed-Loop Verifiable Code Generation, which uses consistency checking to provide a strong filter for incorrect code. Clover performs consistency checks among code, docstrings, and formal annotations. The checker is implemented using a novel integration of formal verification tools and large language models. We provide a theoretical analysis to support our thesis that Clover should be effective at consistency checking. We also empirically investigate its performance on a hand-designed dataset (CloverBench) featuring annotated Dafny programs at a textbook level of difficulty. Experimental results show that for this dataset: (i) LLMs are reasonably successful at automatically generating formal specifications; and (ii) our consistency checker achieves a promising acceptance rate (up to 87%) for correct instances while maintaining zero tolerance for adversarial incorrect ones (no false positives). Clover also discovered 6 incorrect programs in the existing human-written dataset MBPP-DFY-50.

1 Introduction

Large language models (LLMs) have recently demonstrated remarkable capabilities. They can engage in conversation, retrieve and summarize vast amounts of information, generate and explain text and code, and much more [7,17,48]. Among many possible applications, their ability to synthesize code based on natural language descriptions [14,16,38] is stunning and could potentially enhance the productivity of programmers significantly [62]. Indeed, futurists are already claiming that in the future, most code will be generated by LLMs (or their successors) and not by humans.

C. Sun and Y. Sheng—Equal Contribution.

G. Avni et al. (Eds.): SAIV 2024, LNCS 14846, pp. 134–155, 2024.
https://doi.org/10.1007/978-3-031-65112-0_7

However, there is a fundamental challenge that must be overcome before realizing this future. Currently, there is no trustworthy way to ensure the correctness of AI-generated code [40]. Without some quality control, the prospect of dramatically scaling up code generation is highly concerning and could lead to catastrophic outcomes resulting from faulty code [20,52,55]. For the most part, the current best practice for curating AI-generated artifacts is to have a human expert in the loop, e.g., [25]. While this is better than nothing, requiring human oversight of AI-generated code limits scalability. Furthermore, recent work [28,50,64,70] confirms the many risks and limitations of using AI even as a code assistant. Results suggest that developers with access to AI assistants write more insecure code, while at the same time having higher confidence in their code [52].

It is becoming clear that curating the quality of AI-generated content will be one of the most crucial research challenges in the coming years. However, in the specific case of generated code, *formal verification* can provide mathematically rigorous guarantees on the quality and correctness of code. What if there were a way to *automatically* apply formal verification to generated code? This would not only provide a scalable solution, but it could actually lead to a future in which generated code is *more reliable* than human-written code.

Currently, formal verification is only possible with the aid of time-consuming human expertise. The main hypothesis of this paper is that *LLMs are well-positioned to generate the collateral needed to help formal verification succeed*; furthermore, they can do this *without compromising the formal guarantees provided by formal methods*. To understand how, consider the following breakdown of formal verification into three parts: (i) construct a mathematical model of the system to be verified; (ii) provide a formal specification of what the system should do; and (iii) prove that the model satisfies the specification. For code, step (i) is simply a matter of converting the code into mathematical logic, which can be done automatically based on the semantics of the programming language. And step (iii) can often be done automatically thanks to powerful automated reasoning systems for Boolean satisfiability (SAT) and satisfiability modulo theories (SMT) [4]. In fact, a number of tools already exist that take a specification (the result of step (ii)) and some code as input and largely automate steps (i) and (iii) (e.g., [3,35,36]).[1] However, step (ii) appears to be a showstopper for automated formal verification of generated code, as traditionally, significant human expertise is required to create formal specifications and ensure that they are both internally consistent and accurately capture the intended functionality.

Two key insights suggest a way forward. The first insight is simply a shift in perspective: the result of any AI-based code generation technique should aim to include *not only code, but also formal specifications*. The second insight is that given these components (and a description in natural language), we can use formal tools coupled with generative AI techniques to *check their consistency*. We name our approach *Clover*, short for *Closed-loop Verifiable Code Generation*,

[1] Such tools have plenty of room for improvement and must be extended to more mainstream languages, but separate research efforts are addressing this.

and we predict that Clover, coupled with steadily improving generative AI and formal tools, will enable a future in which fully automatic, scalable generation of formally verified code is feasible. This paper charts the first steps toward realizing this vision.

The Clover paradigm consists of two phases: generation and verification. In this paper, we also assume that a precise natural language description of the desired functionality is available. In the first (generation) phase, some process is used to create code annotated with formal specifications. For simplicity, we refer to the formal specifications as "annotations" and the natural language descriptions as "docstrings" going forward. It is worth noting that, in other scenarios, including annotating an existing codebase or generating code given specifications, one or two of these components (code, annotations, docstrings) might already exist, in which case generative AI might be used to construct only the other(s). In fact, the second phase is completely agnostic to the process used in the first phase; we simply insist that the result of the first phase has all three components: code, annotations, and docstrings. In the second (verification) phase, a series of *consistency checks* are applied to the code, annotations, and docstrings (see Fig. 1). The Clover hypothesis is that if the consistency checks pass, then (i) the code is functionally correct with respect to its annotations; (ii) the annotations capture the full functionality of the code; and (iii) the code and its annotations also align with natural language descriptions of the functionality (docstrings).

The idea is that we can unleash increasingly powerful and creative generative AI techniques in the generation phase, and then use the verification phase as a strong filter that only approves of code that is formally verified, accurately documented, and internally consistent.

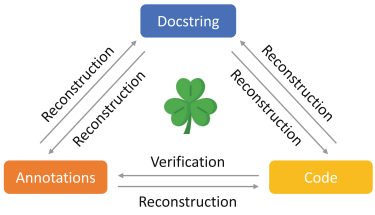


Fig. 1. The Clover paradigm

In this paper, we focus on the verification phase, though we also include some demonstrations of the generation phase in our evaluation. Our contributions include:[2]

- the Clover paradigm with a solution for the verification phase (Sect. 3.2);
- the CloverBench dataset, featuring manually annotated Dafny programs with docstrings, which contains both ground-truth examples and adversarial incorrect examples (Sect. 4.1);
- a demonstration of the feasibility of using GPT-4 to generate code, specifications, and both (Sect. 4.2);
- implementation and evaluation of the verification phase of the Clover paradigm using GPT-4 and the Dafny verification tool (Sect. 4.3, 4.4, and 4.5).

Our initial results on CloverBench are promising. Our implementation accepts 87% of the correct examples and rejects 100% of the adversarial incor-

[2] In addition, a theoretical framework which argues for the trustworthiness of the Clover approach is available in [61, Appendix A.1].

rect examples. We expect that the acceptance rate can be improved in a variety of ways while maintaining the strong ability to reject incorrect code. Beyond CloverBench, Clover also correctly detects 6 incorrect programs and accepts 89% of the correct programs in the external dataset MBPP-DFY-50 [44].[3]

2 Preliminaries: Deductive Program Verification

Deductive program verification provides a framework for mathematically proving that programs are correct [23,30]. A standard approach is to first *annotate* code with preconditions, postconditions, and loop invariants, and then check that the code satisfies the specification given by these annotations. That is, if the code is executed starting from a program state that satisfies the precondition, the resulting program state after executing the code will satisfy the postcondition. Checking whether a given piece of code meets the specification corresponding to some set of annotations can be done by checking the validity of logical formulas known as verification conditions, which is typically done automatically using satisfiability modulo theories (SMT) solvers. Dafny is a programming language used in our evaluation with state-of-the-art support for deductive verification [36]. Dafny's back-end includes both a compiler, capable of generating a runnable binary, and a verifier, which formally checks whether the code conforms to its specification.

In this paper, we assume annotations are given at the function level. For example, a function for finding the maximal element in an array of integers will have a precondition requiring that the input array is nonempty, and a postcondition ensuring that the return value is indeed the maximal element of the input array. Loops must be accompanied by loop invariants, which are used for a proof by induction on the number of loop iterations. For example, Listing 1.1 shows a Dafny function for finding the maximal element of an array, with a docstring, a precondition, two postconditions, and a loop invariant. Dafny is able to automatically verify this function with respect to these formal annotations.

Listing 1.1. Dafny function with consistent code, docstring, and annotations.

```
// Find the maxiaml element in an integer array
method maxArray(a: array<int>)returns (m: int)
 requires a.Length>=1
 ensures exists k :: 0<=k<a.Length && m==a[k]
 ensures forall k :: 0<=k<a.Length ==> m>=a[k]
{
 m := a[0];
 var i := 1;
 while (i < a.Length)
 invariant 0<=i<=a.Length &&
           (forall k :: 0<=k<i ==> m>=a[k]) &&
           (exists k::0<=k<i && m==a[k])
 {
  m := if m>a[i] then  m else a[i];
  i := i + 1;
 }
}
```

[3] https://github.com/ChuyueSun/Clover.

Listing 1.2. Example of generated docstring.

```
"This method returns the maximum value, m, in the integer array a, ensuring
    that m is greater than or equal to all elements in a and that m is indeed
     an element of a"
```

Listing 1.3. Example of generated annotations.

```
requires a.Length > 0;
ensures forall k::0<=k<a.Length ==> a[k]<=m
ensures exists k::0<=k<a.Length && a[k]==m
```

Listing 1.4. Example of generated code (loop invariant omitted).

```
var i := 0;
m := a[0];
while i<a.Length {
 if (a[i] > m) { m := a[i]; }
 i := i+1;
}
```

3 Clover

3.1 Clover Generation Phase

As mentioned in Sect. 1, Clover expects the output of the generation phase to consist of code, annotations, and docstrings. These could be generated in a variety of ways. In this paper, we include a feasibility study for three possible instances of the generation phase.

First, we consider the case where the annotations (i.e. the formal specifications) are provided, and an LLM is asked to generate the code. This is analogous to the standard synthesis problem that is well-studied in PL research [26,42,43,65].

Second, we explore the opposite: generating annotations given the code. This use case could be relevant for someone trying to verify legacy code.

Finally, we explore the possibility of generating both code and annotations from a precise natural language description. This use case aligns with our proposed vision that LLMs should include specifications when generating code from natural language.

Our goal with these evaluations is not necessarily to chart new research directions, as all of these directions are worthy of a much more targeted research effort (and indeed, there are many such efforts underway [6,44,75]). Rather, our goal here is simply to demonstrate the feasibility of different instances of the generation phase in order to lend credibility to the overall Clover vision. We report on an evaluation of each of these use cases in Sect. 4.

3.2 Clover Verification Phase

As mentioned in Sect. 1, Clover expects the input of the verification phase to contain three components: code, annotations, and docstrings. *Additionally, we*

expect that each of the three components provides sufficient detail to unambiguously determine a unique result of running the code on any given input. The verification phase checks the consistency of every pair of components, as shown in Fig. 1, and succeeds only if all checks pass. Docstrings and annotations are consistent if they contain the same information, i.e., they imply each other semantically. The notion of consistency between a docstring and code is similar. On the other hand, to assess the consistency between code and annotations, we can leverage deductive verification tools.

One key idea used to check consistency between components in Fig. 1 is *reconstruction testing*. Given the three components (code, docstring, annotations) as input, we try to reconstruct a single component from a single other component, and then we check if the reconstructed result is equivalent to the original component. We do this for five out of the six (directed) edges of Fig. 1. A special case is checking that the code conforms to the annotations, where we use formal verification based on deductive verification tools instead of a reconstruction test. For an input instance to pass the verification phase, it must pass all six tests. For the reconstruction itself, we use an LLM (our evaluation uses GPT-4), and for equivalence testing, we use LLMs to compare text, formal tools to compare annotations, and pointwise sampling to compare code. A running example is provided in Sect. 3.3. Listings 1.2, 1.3, and 1.4 are examples of generated artifacts. We explain how these checks are done in detail next. Pseudocode is shown in Algorithm 1.[4]

Code-Annotations Consistency. (1. Code → Annotations: Soundness) A deductive verification tool (our evaluation uses Dafny) checks that the code satisfies the annotations. This is a standard formal verification check (see Sect. 2). (2. Annotations → Code: Completeness) To prevent annotations that are too trivial from being accepted, we test whether the annotations are strong enough by testing if they contain enough information to reconstruct functionally equivalent code. Given the annotations, we use an LLM to generate new code. Then, we check the equivalence between the generated and the original code. If the equivalence check passes, the annotations are considered complete.

Annotation-Docstring Consistency. (1. Annotations → Docstring) An LLM is asked to generate a new docstring from the annotations. Then, the new and the original docstrings are checked for semantic equivalence. (2. Docstring → Annotations) An LLM is asked to generate new annotations from the docstring. Then, the new and the original annotations are checked for logical equivalence.

Code-Docstring Consistency. (1. Docstring → Code) An LLM is asked to generate code from the docstring. Then, the new and the original code are checked for functional equivalence. (2. Code → Docstring) An LLM is asked to generate a new docstring from the code. Then, the new and the original docstrings are checked for semantic equivalence.

[4] For more discussion about limitations and variants of, and future directions for Clover, see [61, Appendix A.4].

Algorithm 1. Clover Consistency Check ($k = 1$)

Input: Docstring d, annotations a, code c.
Output: $True/False$

Set number of tries $m = 3$
if Dafny fails to verify a, c **then** ▷ annotation soundness
  Return $False$
for $i = 1$ to m **do** ▷ annotation completeness
  Call LLM to generate code c' from a.
  if c' successfully compiles **then**
    break
  else
    Provide feedback from failed compilation to LLM
if c' is not equivalent to c **then**
  Return $False$
for $i = 1$ to m **do** ▷ doc2code
  Call GPT-4 to generate code c' from d.
  if c' successfully compiles **then**
    break
  else
    Provide feedback from failed compilation to LLM
if c' is not equivalent to c **then**
  Return $False$
for $i = 1$ to m **do** ▷ code2doc
  Call GPT-4 to generate docstring d'_i from c.
if all d'_i are not equivalent to d **then**
  Return $False$
for $i = 1$ to m **do** ▷ doc2anno
  Call GPT-4 to generate annotations a' from d.
  if a' successfully compiles **then**
    break
  else
    Provide feedback from failed compilation to LLM
if a' is not equivalent to a **then**
  Return $False$
for $i = 1$ to m **do** ▷ anno2doc
  Call GPT-4 to generate docstring d' from a.
if all d' are not equivalent to d **then**
  Return $False$
Return $True$

We consider the methods used for equivalence checking to be parameters to Clover. We discuss some possibilities (including those used in our evaluation) below.

Equivalence Checking for Code. Standard equivalence checks for code include input-output comparisons, concolic testing ([8,9,33,63]), and even full formal equivalence checking (e.g., [18]). Our evaluation checks that the outputs agree on a set of inputs included as part of the CloverBench dataset. This test is, of course, imprecise, but our evaluation suggests that it suffices for the level of complexity in CloverBench. For example, the generated code of Listing 1.4 is equivalent to the original code in Listing 1.1, and indeed our equivalence check succeeds for this example. More advanced equivalence checking techniques might be required for more complex examples.

Equivalence Check for Docstrings. Checking equivalence between docstrings is challenging, as natural language is not mathematically precise. In our evaluation, we ask an LLM (GPT-4) to check whether two docstrings are semantically equivalent. For example, it accepts Listing 1.2 as equivalent to the docstring in Listing 1.1. Other NLP-based semantic comparisons may also be worth exploring.

Equivalence Check for Annotations. To check the equivalence of two sets of annotations, we write the equivalence as a formal lemma and ask a formal tool (in our evaluation, we again use Dafny) to prove the lemma. This method is sound in the sense that it succeeds only if the two sets of annotations are indeed equivalent. For example, we are able to automatically prove that the annotations in Listing 1.3 are equivalent to those in Listing 1.1. Note that this process may fail, even on equivalent annotations, due to the limitations of the verification tool being used. The specific equivalence checking template we use is described in Sect. 4.1 and is included as part of our CloverBench dataset.

Although there are many approximate approaches, the two parts that leverage formal tools, the soundness check and the equivalence check for annotations, are exact. The equivalence check used for code is also strong, though not perfect. *These checks strongly contribute to the lack of false positives in our evaluation.*[5]

3.3 Consistency Checking Example

For illustration purposes, before the evaluation section, we describe how each step described above is carried out for the `maxArray` example (Listing 1.1).

Listing 1.5. Annotation Input

```
method foo(a: array<int>) returns (m: int)
 requires a.Length >= 1
 ensures (forall k :: 0<=k<a.Length ==> m>=a[k]) && (exists k :: 0<=k<a.
     Length && m==a[k])
{
//TOFILL
}
```

[5] An analytical model of reconstruction tests is provided in [61, A.1].

Listing 1.6. Code Input

```
method foo(a: array<int>) returns (m:
     int)
//TOFILL
{
 m := a[0];
 var i := 1;
 while (i < a.Length)
 invariant 0<=i<=a.Length &&
           (forall k :: 0<=k<i ==> m
    >=a[k]) &&
           (exists k :: 0<=k<i && m==
    a[k])
 {
  m := if m>a[i] then  m else a[i];
  i := i + 1;
 }
}
```

Listing 1.7. Generated code

```
method foo(a: array<int>) returns (m
    : int)
{
  var i := 0;
  m := a[0];
  while i<a.Length
  {
    if(a[i] > m) {
      m := a[i];
    }
    i := i+1;
  }
}
```

Listing 1.8. Docstring Input

```
// specification: Returns the maximum value m present in the array a.
method maxArray(a: array<int>) returns (m: int)
//TOFILL
```

anno-sound. Soundness is checked by simply running the Dafny verifier on the annotated code shown in Listing 1.1.

anno-complete. For the annotations to be complete with respect to the code, we must be able to reconstruct the code from the annotations alone. Therefore, we ask GPT-4 to generate code from the anonymized function signature and the annotations (Listing 1.5). In the prompt, we instruct the LLM to generate code based on the Dafny specification in natural language, without providing any few-shot examples. We run and provide feedback from the Dafny compiler up to three times to help GPT-4 fix its code generation. For this example, GPT-4 generates the correct code on the first try, shown in Listing 1.7. Then, we check if the generated code is equivalent to the original ground-truth code by comparing their outputs.[6]

doc2anno. We try to reconstruct equivalent ground-truth annotations from the docstring alone. First, we call GPT-4 with the docstring and the function signature (Listing 1.8) asking for the annotations. To eliminate simple syntax errors, we try to compile the generated annotations with an empty code body and use error messages generated by the Dafny compiler as feedback (up to 3 times). Results presented in Sect. 4.2, above, suggest that the feedback mechanism is quite important. For our example, GPT-4 generates correct annotations on the first try, shown in Listing 1.9.

Listing 1.9. Generated annotations

```
  requires a.Length > 0;
  ensures forall k :: 0 <= k < a.Length ==> a[k] <= m;
  ensures exists k :: 0 <= k < a.Length && a[k] == m;
```

[6] Example code for this check is shown in [61, Appendix A.8].

anno2doc. To reconstruct a docstring from annotations, we ask GPT-4 to generate a new docstring three times independently in one session in plain natural language. If one of them is equivalent to the original docstring, the check passes. We consider two docstrings to be equivalent if they contain the same information about the functional behavior of the program, ignoring implementation details that do not affect functionality. In the prompt, we ask, "Do these two docstrings describe the exact same functional behavior of a Dafny program? Return 'Yes' or 'No'." followed by the two docstrings in question (see GPT-4 System Prompt in [61, Appendix A.7]). Note that the two calls to GPT-4 are independent to ensure that the second call contains no memory of the first call. That is, the answer to the question of whether the original and the generated docstrings are semantically equivalent is unaffected (other than by bias inherent in the model) by the first call to generate an equivalent docstring from the original. For our example, GPT-4 generates a correct docstring on the first try, shown below:

> This method returns the maximum value, m, in the integer array a, ensuring that m is greater than or equal to all elements in a and that m is indeed an element of a.

code2doc. The process is almost identical to `anno2doc`. The only difference is that in order to ensure the code provides all the information needed for the docstring generation, we embed the preconditions into the code in the form of `assert` statements.

doc2code. This process leverages one of the most common use cases of GPT-4: generating code from a natural language description. The concrete steps are similar to that described in `anno-complete`. The only difference is that instead of using verifier-generated error messages, we use compiler-generated error messages since we want to ensure that the code generation relies only on the docstring.

4 Evaluation

We have implemented a first prototype of our Clover consistency checking algorithm using GPT-4 [48] as the LLM and using the Dafny programming language and verification tool [36]. We selected Dafny because it provides a full-featured and automatic deductive verification toolkit including support for a rich language of formal specifications and a backend compiler linking to a verifier. But Clover can be instantiated using any language and tool supporting deductive program verification. Note that it is also crucial that the selected LLM has a good understanding of the programming language. In our case, we were pleasantly surprised to discover that GPT-4 understands Dafny programs well enough to perform the translations between code, docstrings, and annotations that Clover relies on (Sect. 4.2), despite the fact that Dafny is not a mainstream programming language. In our evaluation, we use Dafny version 4.0.0.50303 with Z3 version 4.8.12. The evaluation also uses a concrete set of Dafny examples which we describe next.

4.1 Dataset: CloverBench

4.1.1 Dafny

There have been several popular datasets for code generation in different domains [2,14,29,34,72], but none of them contain annotations or use the Dafny language. Furthermore, we wanted to carefully curate the programs used to test our first Clover prototype. In particular, as mentioned above, we require the docstring and annotations to precisely specify a unique output for every input. For these reasons, we introduce a new hand-crafted dataset we call CloverBench. We expect to add and improve it over time, but at the time of writing, it is based on 60 small hand-written example programs as might be found in standard CS textbooks.[7] For each program, there are five variants: a "ground-truth" variant whose code, annotations, and docstring are correct and consistent (verified by hand); and 4 adversarial incorrect variants. Associated with each example, there is also one set of inputs and one Dafny code template for annotation equivalence checking. We discuss possible data contamination issues in [61, Appendix A.4].

It is worth noting that recently, independent and concurrent work [6,44] on Dafny annotation generation has produced some Dafny examples with annotations that are similar to CloverBench. However, there are only a limited number of these benchmarks, and they do not always meet the strict criteria we have imposed in this paper (single-method code with precise specifications), and thus our carefully curated CloverBench is still needed. In MCTS [6], only 5 examples are provided. In dafny-synthesis [44], the authors translate some programs from MBPP [2], a data set of Python programs, into Dafny. We do evaluate Clover on a subset of these benchmarks in Sect. 4.3, below.

Set of Inputs. Each program in CloverBench contains five individual tests designed to run that program on a specific input value. We use these tests as a rough check for whether a piece of generated code is equivalent to the original code. If the generated code has the same output as the original code for all five tests, then the code is considered to be equivalent (See [61, Appendix A.8]).

Annotation Equivalence Checking Template. Each template can be used to formally verify the consistency of two sets of annotations with Dafny. For two sets of annotations a and b to be equivalent, the preconditions and postconditions of a and b must be verified to be equivalent separately. We use a script to automatically create annotation templates.[8]

4.2 Generation Phase

As mentioned in Sect. 3.1, we explored three use cases for the generation phase. In all cases, we use GPT-4 as the generating LLM.

First, we ask GPT-4 to generate the code from specifications for each of the 60 examples in CloverBench under various conditions. We manually checked

[7] Since we wanted to concentrate on the most basic scenario initially, our initial dataset only features examples containing exactly one method and no helper functions.

[8] Details and an example are shown in [61, Appendix A.7].

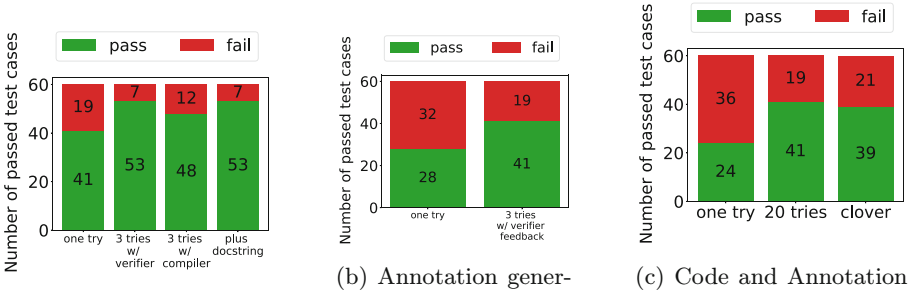
the generated code for correctness. Figure 2a shows the results. The first bar ("one try") shows the result when asking GPT-4 to produce the code, given the annotations, in a single try. The next bar allows GPT-4 to try three times, each time providing the output of the Dafny compiler and verifier as feedback (See [61, Appendix A.6] for an example of using Dafny feedback). The next is similar but uses the output of only the Dafny compiler. In the last bar, we allow three tries, with feedback from the Dafny compiler and verifier, and we also provide the docstring. We see that, at its best, GPT-4 can correctly provide the code for 53 out of 60 examples, and it does best when it gets the most feedback from Dafny. This suggests that GPT-4 is already performing reasonably well as a code synthesis tool for Dafny programs.

Second, we asked GPT-4 to generate full annotations (pre-conditions, post-conditions, and loop invariants) from the code alone. Figure 2b shows the results. In one try, GPT-4 succeeds on 28 of 60 programs. Given three tries and maximal feedback from Dafny, this improves to 41 out of 60. Though not perfect, out of the box, GPT-4 can produce correct annotations for the majority of programs in our simple set of benchmarks. This suggests that using LLMs for generating annotations is feasible, and we expect that further efforts in this direction (including fine-tuning models for the task) will likely lead to even stronger capabilities.

Finally, for the last experiment, we ask GPT-4 to generate both the code and the annotations from the docstring alone. Figure 2c shows the results. On the first try, GPT-4 succeeds on 24 of 60 programs. However, if we simply do 20 independent tries and test whether GPT-4 succeeds on any of these tries, the number improves to 41. This naturally raises the question: how can we leverage multiple LLM tries without having to check each one by hand? This is exactly what the verification phase is for! The last column in the figure shows that if we run the Clover verification phase, it accepts at least one correct answer for 39 of 41 examples for which GPT-4 generates a correct answer. Further more the Clover verification check never accepts an incorrect answer. Full results are reported in [61, Appendix A.10]. Thus, we can fully automatically generate 39 of 60 programs from natural language alone, with the guarantee that the generated programs pass all Clover consistency checks. While these numbers must be improved and more complicated examples must be tried, these early results are promising and suggest that these ideas should be explored further.

4.3 Verification Phase: Results on Ground-Truth Examples

Our main experiment evaluates the capabilities of the Clover consistency checking algorithm. During consistency checking, we consider everything that appears in the body of a method, including assertions and invariants, to be part of the code, and consider the annotations to consist only of pre- and post-conditions. The reason for this is for modularity. We need to be able to separate out the annotation and have it generate the code. The assertions and invariants in the code have no context without the code, and are thus meaningless without it; moreover, the pre- and post-conditiosn contain all the information necessary to reconstruct the code. Thus, this division makes the most sense for Clover.



(a) Code generation.

(b) Annotation generation

(c) Code and Annotation Generation

Fig. 2. Generation phase feasibility study

For each example in CloverBench, we run all 6 checks described in Sect. 3.2. For checks that use Dafny, we use three tries and provide feedback from Dafny's compiler after each try. We also evaluate the effect of multiple independent runs, meaning that we repeat each of the 6 checks k times. If any one of the k attempts succeeds, then the check is considered to have passed. The results are summarized in Table 1. When $k = 1$, we see that our Clover implementation accepts 45 of 60 correct ("ground truth") examples and rejects all incorrect examples. When $k = 10$, Clover accepts 52 of 60 correct examples and rejects all incorrect ones. Details on each of the 6 checks for the ground truth examples are shown in Table 2. All acceptance rates are above 80%. Failures are mostly due to incorrect or imprecise reconstruction. More details can be found in [61, Appendix A.4.5]. We expect that using better LLMs (either better general-purpose LLMs or LLMs fine-tuned for program verification or a specific language or both) will improve the acceptance rate. For the complete experimental results, see [61, Appendix A.10]. Since our ground-truth examples are hand-written and hand-checked for correctness, it is not surprising that all pass the Dafny verifier (i.e., all annotations are sound). Annotation completeness requires successful synthesis of code from annotations, and here, we get an 88% acceptance rate when $k = 1$, which goes up to 95% with $k = 10$. The main reason for failure is incorrect generation of Dafny syntax by GPT-4. In doc2anno generation, we generate annotations from docstrings. The main failure comes again from GPT-4 generating incorrect

Table 1. Summary of the experimental results for the verification phase.

	Accept (k = 1)	Accept (k = 10)
Ground-Truth	45/60 (75%)	52/60 (87%)
Adversarial-C1	0/60 (0%)	0/60 (0%)
Adversarial-C2	0/60 (0%)	0/60 (0%)
Adversarial-C3	0/60 (0%)	0/60 (0%)
Adversarial-C6	0/60 (0%)	0/60 (0%)

Table 2. Ground-truth acceptance for each of the 6 Clover checks.

Ground-Truth	Accept (k = 1)	Accept (k = 10)
anno-sound	60/60 (100%)	60/60 (100%)
anno-complete	53/60 (88%)	57/60 (95%)
doc2anno	51/60 (85%)	53/60 (88%)
anno2doc	60/60 (100%)	60/60 (100%)
code2doc	58/60 (97%)	60/60 (100%)
doc2code	49/60 (82%)	56/60 (93%)

Dafny syntax. anno2doc and code2doc have perfect acceptance rates. On the one hand, this is because GPT-4 is very good at synthesizing natural language. On the other hand, our docstring equivalence checker is not very strong and skews towards acceptance. As long as they do not directly contradict each other, information omissions or additions in docstrings frequently go unnoticed by GPT-4. Improving this equivalence checker is one important direction for future work. doc2code generation shares the same issues as anno-complete and doc2anno: failure because of invalid Dafny syntax generation. It also improves significantly (93% vs 82%) using $k = 10$ instead of $k = 1$.

Table 3. Categories of adversarial incorrect examples.

	Code	Annotations	Docstring	Note
C0	–	–	–	omitted: ground-truth
C1	–	–	mutated	strengthen/weaken docstring
C2	–	mutated	–	weaken annotation
C3	-	mutated	mutated	weaken annotations and docstring simultaneously
C4	mutated	–	–	omitted: cannot pass soundness check
C5	mutated	–	mutated	omitted: cannot pass soundness check
C6	mutated	mutated	-	code still satisfies annotations
C7	mutated	mutated	mutated	omitted: non-sense or is a variant of another ground-truth

4.4 Verification Phase: Results on MBPP-DFY-50

To explore the effectiveness of Clover on external datasets, we ran Clover on the MBPP-DFY-50 dataset [44], which consists of 50 Dafny programs translated by hand from Python, with docstrings and annotations. Our run revealed a number of interesting things about this dataset. First, 17 of the 50 samples are out of scope for Clover. Two are out of scope because the docstrings are not precise

enough to specify a unique output for each input. The other 15 require auxiliary functions or predicates. Extending Clover to such benchmarks is on our roadmap for future work. Of the remaining 33 programs, 24 are accepted by Clover, and 9 are rejected.

Looking closely at the 9 rejected samples, we determined that 6 are, in fact, incorrect: 5 have factual contradictions in their docstrings and pre-conditions; and 1 has trivial (too weak) post-conditions that do not reflect the requirements in the docstrings.[9] The remaining 3 are false negatives: correct programs that do not pass all of the Clover checks. We determined that the 24 accepted benchmarks are all correct (0 false positives), once again demonstrating that Clover provides a strong filter against incorrect code. Overall, after correctly categorizing the 33 benchmarks, Clover achieves an 89% acceptance rate ($k = 10$) while maintaining a 100% rejection rate for incorrect benchmarks.[10]

4.5 Verification Phase: Results on Adversarial Examples

As mentioned, for each program in our dataset, we created 4 adversarial incorrect versions. Here we describe them in more detail. Table 3 lists all possible ways we can mutate the ground-truth example, while still ensuring that it passes the Dafny verification check (anno-sound). Thus, for these examples, a naive approach using only Dafny (as in [44]) would result in a 100% false positive rate. However, Clover with its 6 consistency checks is able to reject all of them (0% false positive rate). Category C0 is the ground-truth where we mutate nothing. Categories 1 to 7 cover all the possible ways we can mutate C0. Category C1 contains programs in which the docstring is incorrect and the other two are the same as the ground-truth. Category C2 contains programs in which the annotations are incorrect and the other two are the same as the ground-truth. To ensure these examples are not trivially rejected by the Dafny soundness check, we only weaken the annotations to ensure that the code still satisfies the mutated annotations. Category C3 contains programs in which both the annotations and the docstring are mutated. The mutated annotations and docstring are simultaneously weakened, but the two are consistent. Category C6 contains programs in which the annotations and code are consistent but inconsistent with the docstring and thus not detectable by Dafny. Categories C4 and C5 are omitted because they are trivially rejected by the Dafny verifier (i.e., they always fail the soundness check). C7 is also omitted because it's not clear how meaningful it is to change all three, and, excluding the corner case when all three are changed to be mutually consistent, benchmarks in this category should be strictly easier to detect than those in the other categories.

[9] The 6 incorrect samples are shown in [61, Appendix A.9].

[10] Detailed results of the Clover checks for the 27 correct benchmarks are in [61, Appendix A.10]).

Table 4. Rejection rates for adversarial incorrect examples.

Category	C1 Reject		C2 Reject		C3 Reject		C6 Reject	
	k = 1	k = 10	k = 1	k = 10	k = 1	k = 10	k = 1	k = 10
anno-sound	0/60 (0%)	0/60 (0%)	0/60 (0%)	0/60 (0%)	0/60 (0%)	0/60 (0%)	0/60 (0%)	0/60 (0%)
anno-complete	7/60 (12%)	3/60 (5%)	26/60 (43%)	16/60 (27%)	26/60 (43%)	21/60 (35%)	33/60 (55%)	27/60 (45%)
doc2anno	57/60 (95%)	54/60 (90%)	60/60 (100%)	60/60 (100%)	44/60 (73%)	30/60 (50%)	60/60 (100%)	60/60 (100%)
anno2doc	42/60 (70%)	34/60 (57%)	24/60 (60%)	13/60 (22%)	24/60 (60%)	4/60 (7%)	42/60 (70%)	27/60 (45%)
code2doc	57/60 (95%)	54/60 (90%)	0/60 (0%)	0/60 (0%)	51/60 (85%)	43/60 (72%)	43/60 (72%)	40/60 (67%)
doc2code	39/60 (65%)	37/60 (62%)	11/60 (18%)	4/60 (7%)	31/60 (52%)	18/60 (30%)	58/60 (97%)	55/60 (92%)

Table 4 shows the results of the 6 checks for each category. We observe that doc2anno has the highest rejection rate. This is because we use Dafny to do a formal equivalence check, which guarantees that only logically equivalent annotations are accepted. Overall, there are no false positives (no incorrect example passes all 6 checks), as summarized in Table 1.[11]

4.6 A Preliminary Study with Verus

As mentioned, we chose Dafny for our primary study because of its maturity as a deductive verification tool. A natural question is how Clover performs with other systems and languages. To gain some understanding of this, we did a preliminary study using Verus [35], a deductive verification tool for a subset of the Rust programming language. Verus and Dafny share the common goal of integrating verification into the development process, but they differ in several ways. For instance, Verus is designed to be more performant but less automatic than Dafny. This means that it often requires more proof effort than Dafny to verify the same program. Verus is also less mature than Dafny, having been developed only recently.

We implemented 41 ground-truth examples in Verus [35] and used the same approach used with Dafny to perform the Clover consistency checks (except that formal checks were done with the Verus tool instead of Dafny). Also, because the Verus specification format is very new, we started each LLM prompt with a few simple examples of Verus specification syntax. For our 41 examples, Clover accepts 32 of 41 when $k = 1$ and 36 out of 41 when $k = 10$. Full results are shown in Table 5. These early results suggest that Clover can be used with other languages and deductive verification tools.

5 Related Work

Code Generation. Besides well-known work [14,16,38] on code generation using LLMs, [26] is a survey on program synthesis before the era of LLMs. Other works using neural approaches for program synthesis include [2,5,71]. To scale up code generation, researchers have tried to decompose the whole task into smaller steps [5,22,73] and to use execution traces [21,58]. While the aforementioned works

[11] For complete results, see Tables in [61, Appendix A.10].

Table 5. Verus Ground-truth acceptance for each of the 6 Clover checks.

Ground-Truth	Accept (k = 1)	Accept (k = 10)
anno-sound	41/41 (100%)	41/41 (100%)
anno-complete	39/41 (95%)	40/41 (98%)
doc2anno	33/41 (80%)	36/41 (88%)
anno2doc	41/41 (100%)	41/41 (100%)
code2doc	41/41 (100%)	41/41 (100%)
doc2code	41/41 (100%)	41/41 (100%)

synthesize code from natural language, another common theme is to synthesize programs from specifications [1,11,53,59]. Translation between natural and formal language has also been studied in [24,27,60], and LLMs have been used to predict program invariants [39,51,69].

Various approaches have been explored for self-correction in code generation, as surveyed in [49], including self-consistency [66], self-debugging [15,56], and self-improvement [41]. In [47], self-debugging has shown to be limited compared to human-level debugging.

Verified Generation. Prior works acknowledge that verifying whether a generated program is correct is challenging. In [40], a test-case-based approach is demonstrated to be insufficient. Other previous attempts include [32], which asks the model to generate assertions along with the code, and [12–14,54], which study the generation of unit tests and how to use the generated tests to increase the generation accuracy. There is also a line of work [19,31,37,74] on a learning-based approach for verifying correctness. [31,38,57,67] study various approaches for reranking a model's output, and [10] propose a self-repair method combining LLMs and bounded model checking to locate software vulnerabilities and derive counterexamples.

Finally, there has recently been a marked and rapid surge of interest in using LLMs to generate formal annotations for verification purposes. [68] generates specifications by leveraging LLMs and techniques from static analysis and program verification. Research in specific domains includes examples like [45], which proposes a framework for porting C to Checked-C to enable memory safety for C programs, and [46], which uses LLMs to synthesize verified router configurations in networking. Most closely related to our work is [6], which uses Monte Carlo Tree Search to help with the multi-step synthesis of annotated Dafny programs, and [44], which explores prompting techniques for generating Dafny programs. In contrast to our work, both of these focus on generation rather than verification. Furthermore, they use only the soundness check, whereas Clover requires a stronger set of six consistency checks.

6 Conclusion

We have introduced Clover, a paradigm for closed-loop verifiable code generation, together with a new dataset CloverBench featuring 60 hand-crafted Dafny examples. We reduce the problem of checking correctness to the more accessible problem of checking consistency. Initial experiments using GPT-4 on CloverBench are promising. We show an 87% acceptance rate for ground-truth examples in CloverBench and a 100% rejection rate for incorrect examples. Clover also accurately detects 6 incorrect samples and accepts 89% correct ones in the existing human-written dataset MBPP-DFY-50 [44]. There are many avenues for future work, including: better verification tools, improving LLM capabilities for generating code, annotations, and docstrings, improving LLM capabilities for understanding Dafny and Verus syntax, and scaling up to more challenging examples.

Acknowledgements. This work was supported in part by an Amazon Research Award and the Stanford Center for Automated Reasoning (Centaur).

References

1. Alur, R., et al.: Syntax-guided synthesis. In: Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20–23, 2013, pp. 1–8. IEEE (2013). https://ieeexplore.ieee.org/document/6679385/
2. Austin, J., et al.: Program synthesis with large language models. CoRR abs/2108.07732 (2021). arXiv: 2108.07732. https://arxiv.org/abs/2108.07732
3. Barnes, J.: SPARK: The Proven Approach to High Integrity Software. Altran Praxis (2012)
4. Biere, A., Heule, M., van Maaren, H.: Handbook of Satisfiability, vol. 185. IOS Press (2009)
5. Bowers, M., et al.: Top-down synthesis for library learning. In: Proceedings of the ACM Program. Lang. **7**(POPL), 1182–1213 (2023). https://doi.org/10.1145/3571234
6. Brandfonbrener, D., et al.: Verified multi-step synthesis using large language models and monte Carlo tree search. arXiv preprint arXiv:2402.08147 (2024)
7. Bubeck, S., et al.: Sparks of artificial general intelligence: early experiments with GPT-4. arXiv preprint arXiv:2303.12712 (2023)
8. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Draves, R., van Renesse, R. (eds.) 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8–10, 2008, San Diego, California, USA, Proceedings, pp. 209–224. USENIX Association (2008). http://www.usenix.org/events/osdi08/tech/full%5C_papers/cadar/cadar.pdf
9. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013). https://doi.org/10.1145/2408776.2408795
10. Charalambous, Y., et al.: A new era in software security: towards self-healing software via large language models and formal verification. CoRR abs/2305.14752 (2023). https://doi.org/10.48550/arXiv.2305.14752. arXiv: 2305.14752

11. Chaudhuri, S., et al.: Neurosymbolic programming. Found. Trends Program. Lang. **7**(3), 158–243 (2021). https://doi.org/10.1561/2500000049
12. Chen, B., et al.: CodeT: code generation with generated tests. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1–5, 2023. OpenReview.net (2023). https://openreview.net/pdf?id=ktrw68Cmu9c
13. Chen, B., et al.: Codet: code generation with generated tests. arXiv preprint arXiv:2207.10397 (2022)
14. Chen, M., et al.: Evaluating large language models trained on code. CoRR abs/2107.03374 (2021). arXiv: 2107.03374. https://arxiv.org/abs/2107.03374
15. Chen, X., et al.: Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128 (2023)
16. Cheng, Z., et al.: Binding language models in symbolic languages. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1–5, 2023. OpenReview.net (2023). https://openreview.net/pdf?id=lH1PV42cbF
17. Chowdhery, A., et al.: PaLM: scaling language modeling with pathways. CoRR abs/2204.02311 (2022). https://doi.org/10.48550/arXiv.2204.02311. arXiv: 2204.02311. https://doi.org/10.48550/arXiv.2204.02311
18. Churchill, B.R., et al.: Semantic program alignment for equivalence checking. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22–26, 2019, pp. 1027–1040. ACM (2019). https://doi.org/10.1145/3314221.3314596
19. Cobbe, K., et al.: Training verifiers to solve math word problems. CoRR abs/2110.14168 (2021). arXiv: 2110.14168. https://arxiv.org/abs/2110.14168
20. Cotroneo, D., et al.: Vulnerabilities in AI code generators: exploring targeted data poisoning attacks. CoRR abs/2308.04451 (2023). https://doi.org/10.48550/arXiv.2308.04451. arXiv: 2308.04451
21. Ding, Y., et al.: TRACED: execution-aware pre-training for source code. arXiv preprint arXiv:2306.07487 (2023)
22. Ellis, K., et al.: DreamCoder: bootstrapping inductive program synthesis with wake-sleep library learning. In: Freund, S.N., Yahav, E. (eds.) PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20–25, 2021, pp. 835–850. ACM (2021). https://doi.org/10.1145/3453483.3454080
23. Floyd, R.W.: Assigning meanings to programs. In: Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics, pp. 19–32. American Mathematical Society (1967)
24. Ghosh, S., et al.: SpecNFS: a challenge dataset towards extracting formal models from natural language specifications. In: Calzolari, N., et al. (eds.) Proceedings of the Thirteenth Language Resources and Evaluation Conference, LREC 2022, Marseille, France, 20–25 June 2022, pp. 2166–2176. European Language Resources Association (2022). https://aclanthology.org/2022.lrec-1.233
25. Github Copilot. Github Copilot: Your AI Pair Programmer. https://github.com/features/copilot
26. Gulwani, S., Polozov, O., Singh, R.: Program synthesis. Found. Trends Program. Lang. **4**(1–2), 1–119 (2017). https://doi.org/10.1561/2500000010
27. Hahn, C., et al.: Formal specifications from natural language. CoRR abs/2206.01962 (2022). https://doi.org/10.48550/arXiv.2206.01962. arXiv: 2206.01962

28. Hendler, J.: Understanding the limits of AI coding. Science **379**(6632), 548–548 (2023)
29. Hendrycks, D., et al.: Measuring coding challenge competence with APPS. In: Vanschoren, J., Yeung, S.-K. (eds.) Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual (2021). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstractround2.html
30. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259. http://doi.acm.org/10.1145/363235.363259
31. Inala, J.P., et al.: Fault-aware neural code rankers. In: NeurIPS. 2022 (2022). http://papers.nips.cc/paper%5C_files/paper/2022/hash/5762c579d09811b7639be2389b3d07be-Abstract-Conference.html
32. Key, D., Li, W.-D., Ellis, K.: I speak, you verify: toward trustworthy neural program synthesis. CoRR abs/2210.00848 (2022). https://doi.org/10.48550/arXiv.2210.00848. arXiv: 2210.00848
33. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
34. Lai, Y., et al.: DS-1000: a natural and reliable benchmark for data science code generation. In: Krause, A., et al. (eds.) International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA, vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 18319–18345 (2023). https://proceedings.mlr.press/v202/lai23b.html
35. Lattuada, A., et al.: Verus: verifying rust programs using linear ghost types (extended version) (2023). arXiv: 2303.05491 [cs.LO]
36. Leino, K.R.M.: Dafny: an automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR 2010. LNCS (LNAI), vol. 6355, pp. 348–370. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17511-4_20
37. Li, Y., et al.: On the advance of making language models better reasoners. CoRR abs/2206.02336 (2022). https://doi.org/10.48550/arXiv.2206.02336. arXiv: 2206.02336
38. Li, Y., et al.: Competition-level code generation with Alphacode. Science **378**(6624), 1092–1097 (2022)
39. Liu, C., et al.: Towards general loop invariant generation via coordinating symbolic execution and large language models. arXiv preprint arXiv:2311.10483 (2023)
40. Liu, J., et al.: Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. CoRR abs/2305.01210 (2023). https://doi.org/10.48550/arXiv.2305.01210. arXiv: 2305.01210
41. Madaan, A., et al.: Self-refine: iterative refinement with self-feedback. In: Advances in Neural Information Processing Systems, vol. 36 (2024)
42. Manna, Z., Waldinger, R.: Knowledge and reasoning in program synthesis. Artif. Intell. **6**(2), 175–208 (1975)
43. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Commun. ACM **14**(3), 151–165 (1971)
44. Misu, Md.R.H., et al.: Towards AI-Assisted synthesis of verified Dafny methods. In: arXiv preprint arXiv:2402.00247 (2024)
45. Mohammed, N., et al.: Enabling memory safety of C programs using LLMs. arXiv preprint arXiv:2404.01096 (2024)

46. Mondal, R., et al.: What do LLMs need to synthesize correct router configurations? In: Proceedings of the 22nd ACM Workshop on Hot Topics in Networks, pp. 189–195 (2023)
47. Olausson, T.X., et al.: Is self-repair a silver bullet for code generation? In: The Twelfth International Conference on Learning Representations (2023)
48. OpenAI. GPT-4 Technical Report. CoRR abs/2303.08774 (2023). https://doi.org/10.48550/arXiv.2303.08774. arXiv: 2303.08774
49. Pan, L., et al.: Automatically correcting large language models: surveying the landscape of diverse self-correction strategies. arXiv preprint arXiv:2308.03188 (2023)
50. Pearce, H., et al.: Asleep at the Keyboard? Assessing the security of github copilot's code contributions. In: 43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22–26, 2022, pp. 754–768. IEEE (2022). https://doi.org/10.1109/SP46214.2022.9833571
51. Pei, K., et al.: Can large language models reason about program invariants? In: Krause, A., et al. (eds.) International Conference on Machine Learning, ICML 2023, 23–29 July 2023, Honolulu, Hawaii, USA, vol. 202. Proceedings of Machine Learning Research. PMLR, 2023, pp. 27496–27520 (2023). https://proceedings.mlr.press/v202/pei23a.html
52. Perry, N., et al.: Do UsersWrite more insecure code with AI assistants?" CoRR abs/2211.03622 (2022). https://doi.org/10.48550/arXiv.2211.03622
53. Polikarpova, N., Kuraj, I., Solar-Lezama, A.: Program synthesis from polymorphic refinement types. In: Krintz, C., Berger, E.D. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13–17, 2016, pp. 522–538. ACM (2016). https://doi.org/10.1145/2908080.2908093
54. Ryan, G., et al.: Code-aware prompting: a study of coverage guided test generation in regression setting using LLM. arXiv preprint arXiv:2402.00097 (2024)
55. Sandoval, G., et al.: Lost at C: a user study on the security implications of large language model code assistants. In: Joseph, A. (eds.) 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9–11, 2023. Calandrino and Carmela Troncoso. USENIX Association (2023). https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval
56. Saunders. W., et al.: Self-critiquing models for assisting human evaluators. arXiv preprint arXiv:2206.05802 (2022)
57. Shi, F., et al.: Natural language to code translation with execution. In: Goldberg, Y., Kozareva, Z., Zhang, Y. (eds.) Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, EMNLP 2022, Abu Dhabi, United Arab Emirates, December 7–11, 2022, pp. 3533–3546. Association for Computational Linguistics (2022). https://doi.org/10.18653/v1/2022.emnlp-main.231. url: https://doi.org/10.18653/v1/2022.emnlp-main.231
58. Shi, K., et al.: CrossBeam: learning to search in bottom-up program synthesis. In: The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25–29, 2022. OpenReview.net (2022). https://openreview.net/forum?id=qhC8mr2LEKq
59. Solar-Lezama, A.: Program Synthesis by Sketching. University of California, Berkeley (2008)
60. Sun, C., Hahn, C., Trippel, C.: Towards improving verification productivity with circuit-aware translation of natural language to SystemVerilog assertions. In: First International Workshop on Deep Learning-aided Verification (2023)
61. Sun, C., et al.: Clover: closed-loop verifiable code generation (2024). arXiv: 2310.17807 [cs.AI]

62. Tabachnyk, M., Nikolov, S.: ML-enhanced code completion improves developer productivity. Blog (2022). https://blog.research.google/2022/07/ml-enhanced-code-completion-improves.html. Accessed 26 July 2022
63. Udupa, A., et al.: TRANSIT: specifying protocols with concolic snippets. In: Boehm, H.-J., Flanagan, C. (eds.) ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16–19, 2013, pp. 287–296. ACM (2013). https://doi.org/10.1145/2491956.2462174
64. Vaithilingam, P., Zhang, T., Glassman, E.L.: Expectation vs. experience: evaluating the usability of code generation tools powered by large language models. In: Barbosa, D.J.S., et al. (eds.) CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022–5 May 2022, pp. 332:1–332:7, Extended Abstracts. ACM (2022). https://doi.org/10.1145/3491101.3519665
65. Waldinger, R.J., Lee, R.C.T.: PROW: a step toward automatic program writing. In: Proceedings of the 1st International Joint Conference on Artificial Intelligence, pp. 241–252 (1969)
66. Wang, X., et al.: Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171 (2022)
67. Wei, J., et al.: Chain-of-thought prompting elicits reasoning in large language models. In: NeurIPS (2022). http://papers.nips.cc/paper%5C_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html
68. Wen, C., et al.: Enchanting program specification synthesis by large language models using static analysis and program verification. arXiv preprint arXiv:2404.00762 (2024)
69. Wu, H., Barrett, C., Narodytska, N.: Lemur: integrating large language models in automated program verification (2023). arXiv: 2310.04870 [cs.FL]
70. Xu, F.F., Vasilescu, B., Neubig, G.: In-IDE code generation from natural language: promise and challenges. ACM Trans. Softw. Eng. Methodol. **31**(2), 29:1-29:47 (2022). https://doi.org/10.1145/3487569
71. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. In: Barzilay, R., Kan, M.-Y. (eds.) Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers, pp. 440–450. Association for Computational Linguistics (2017). https://doi.org/10.18653/v1/P17-1041
72. Yin, P., et al.: Natural language to code generation in interactive data science notebooks. In: Rogers, A., Boyd-Graber, J.L., Okazaki, N. (eds.) Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023, pp. 126–173. Association for Computational Linguistics (2023). https://doi.org/10.18653/v1/2023.acl-long.9
73. Zelikman, E., et al.: Parsel: A (de-) compositional framework for algorithmic reasoning with language models. arXiv preprint arXiv:2212.10561 (2023)
74. Zhang, T., et al.: Coder reviewer reranking for code generation. In: Krause, A., et al. (eds.) International Conference on Machine Learning, ICML 2023, 23–29 July 2023, Honolulu, Hawaii, USA, vol. 202. Proceedings of Machine Learning Research, pp. 41832–41846. PMLR (2023). https://proceedings.mlr.press/v202/zhang23av.html
75. Zhou, B., Ding, G.: Survey of intelligent program synthesis techniques. In: Saxena, S., Zhao, C. (eds.) International Conference on Algorithms, High Performance Computing, and Artificial Intelligence (AHPCAI 2023), vol. 12941. International Society for Optics and Photonics. SPIE, 2023, 129414G (2023). https://doi.org/10.1117/12.3011627