Clover: <u>Clo</u>sed-Loop <u>Ver</u>ifiable Code Generation

Chuyue Sun[1]*   Ying Sheng[1]*   Oded Padon[2]   Clark Barrett[1]

[1]Stanford University   [2]VMware Research

{chuyues, ying1123}@stanford.edu, oded.padon@gmail.com,
barrett@cs.stanford.edu

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 <u>Clo</u>sed-Loop <u>Ver</u>ifiable 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 [61]. Indeed, futurists are already claiming that in the future, most code will be generated by LLMs (or their successors) and not by humans.

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,

---

* Equal Contribution

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 [28,50,63,69] 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*, 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

---

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

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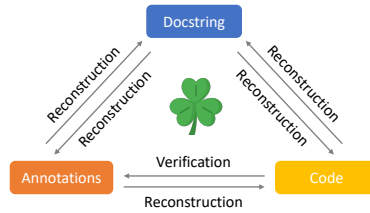
---

[2] In addition, a theoretical framework which argues for the trustworthiness of the Clover approach is available in Appendix A.1. All appendices will be included in an extended version of the paper on arxiv.

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.

```dafny
// 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] The CloverBench dataset and Clover consistency checking implementation will be made available after the anonymous review period.

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 Section 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][64].

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 [6,44,74]). 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 Section 4.

3.2 Clover Verification Phase

As mentioned in Section 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 Figure 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 Figure 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 Figure 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 Section 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 $\rightarrow$ 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 Section 2). (2. Annotations $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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 $\rightarrow$ 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.

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

---

[4] For more discussion about limitations and variants of, and future directions for Clover, see 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 $True$

Equivalence Checking for Code Standard equivalence checks for code include input-output comparisons, concolic testing ([8,9,33,62]), 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 Section 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 Section 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). 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 Section 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;
```

anno2doc To reconstruct a docstring from annotations, we ask GPT-4 to generate a new docstring three times independently in one session. If one of them

---

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

is equivalent to the original docstring, the check passes. We consider two doc-strings to be equivalent if they contain the same information about the functional behavior of the program, ignoring implementation details that do not affect functionality. To check this, we again use GPT-4 (see GPT-4 System Prompt in 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 (see Listing 1.10).

Listing 1.10: 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.'''
```

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 (Section 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 [2,14,29,70,34], 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 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 Section 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 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 Section 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 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

---

[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 Appendix A.7.

(See 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 Table 8 in 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.

(a) Code generation.

(b) Annotation generation

(c) Code and Annotation Generation
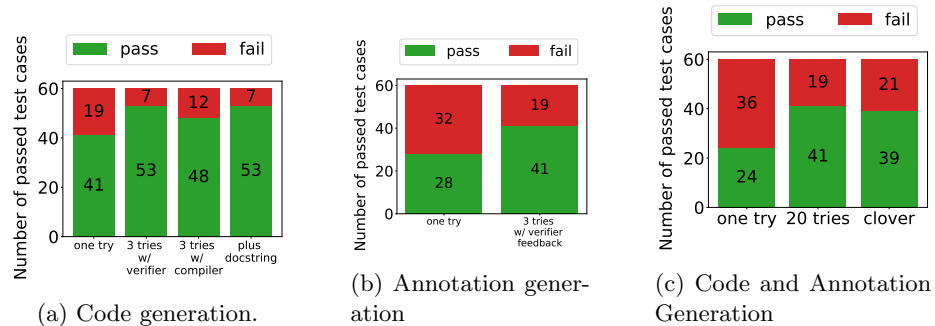
Fig. 2: Generation phase feasibility study

4.3   Verification Phase: Results on Ground-Truth Examples

Our main experiment evaluates the capabilities of the Clover consistency check-
ing algorithm.[9] For each example in CloverBench, we run all 6 checks described
in Section 3.2. For checks that use Dafny, we use three tries and provide feed-
back 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 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 Tables 11 and 12 in the Appendix.

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

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%)

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

---

[9] 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.

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 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 $k = 1$.

	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

Table 3: Categories of adversarial incorrect examples.

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.[10] 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.[11]

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.

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: Rejection rates for adversarial incorrect examples.

---

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

[11] Detailed results of the Clover checks for the 27 correct benchmarks are in the Appendix in Table 23).

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.[12]

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, becuase 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.

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%)

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

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

---

[12] For complete results, see Tables 15 and 16 for C1, Tables 17 and 18 for C2, Tables 19 and 20 for C3 and Tables 21 and 22 for C6 in Appendix A.10.

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 [72,22,5] and to use execution traces [21,58]. While the aforementioned works 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 [39,51,68].

Various approaches have been explored for self-correction in code generation, as surveyed in [49], including self-consistency [65], 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 [13,14,54,12], 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 [19,31,37,73] on a learning-based approach for verifying correctness. [31,38,57,66] study various approaches for reranking a model's output, and [10] propose a self-repair method combining LLMs and formal verification strategies.

Finally, there has recently been a marked and rapid surge of interest in using LLMs to generate formal annotations for verification purposes. [67] 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 Clover-Bench 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.

References

1. Alur, R., Bodík, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: 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., Odena, A., Nye, M.I., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C.J., Terry, M., Le, Q.V., Sutton, C.: Program synthesis with large language models. CoRR **abs/2108.07732** (2021), `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., Olausson, T.X., Wong, L., Grand, G., Tenenbaum, J.B., Ellis, K., Solar-Lezama, A.: Top-down synthesis for library learning. Proc. ACM Program. Lang. **7**(POPL), 1182–1213 (2023). https://doi.org/10.1145/3571234, `https://doi.org/10.1145/3571234`

6. Brandfonbrener, D., Raja, S., Prasad, T., Loughridge, C., Yang, J., Henniger, S., Byrd, W.E., Zinkov, R., Amin, N.: Verified multi-step synthesis using large language models and monte carlo tree search. arXiv preprint arXiv:2402.08147 (2024)

7. Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y.T., Li, Y., Lundberg, 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_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, `https://doi.org/10.1145/2408776.2408795`

10. Charalambous, Y., Tihanyi, N., Jain, R., Sun, Y., Ferrag, M.A., Cordeiro, L.C.: 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, `https://doi.org/10.48550/arXiv.2305.14752`

11. Chaudhuri, S., Ellis, K., Polozov, O., Singh, R., Solar-Lezama, A., Yue, Y.: Neurosymbolic programming. Found. Trends Program. Lang. **7**(3), 158–243 (2021). https://doi.org/10.1561/2500000049, `https://doi.org/10.1561/2500000049`

12. Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J.G., Chen, W.: Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397 (2022)

13. Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J., Chen, W.: Codet: Code generation with generated tests. In: The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023. OpenReview.net (2023), `https://openreview.net/pdf?id=ktrw68Cmu9c`

14. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger,

G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F.P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W.H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A.N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., Zaremba, W.: Evaluating large language models trained on code. CoRR **abs/2107.03374** (2021), `https://arxiv.org/abs/2107.03374`

15. Chen, X., Lin, M., Schärli, N., Zhou, D.: Teaching large language models to self-debug. arXiv preprint arXiv:2304.05128 (2023)

16. Cheng, Z., Xie, T., Shi, P., Li, C., Nadkarni, R., Hu, Y., Xiong, C., Radev, D., Ostendorf, M., Zettlemoyer, L., Smith, N.A., Yu, T.: 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., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., Schuh, P., Shi, K., Tsvyashchenko, S., Maynez, J., Rao, A., Barnes, P., Tay, Y., Shazeer, N., Prabhakaran, V., Reif, E., Du, N., Hutchinson, B., Pope, R., Bradbury, J., Austin, J., Isard, M., Gur-Ari, G., Yin, P., Duke, T., Levskaya, A., Ghemawat, S., Dev, S., Michalewski, H., Garcia, X., Misra, V., Robinson, K., Fedus, L., Zhou, D., Ippolito, D., Luan, D., Lim, H., Zoph, B., Spiridonov, A., Sepassi, R., Dohan, D., Agrawal, S., Omernick, M., Dai, A.M., Pillai, T.S., Pellat, M., Lewkowycz, A., Moreira, E., Child, R., Polozov, O., Lee, K., Zhou, Z., Wang, X., Saeta, B., Diaz, M., Firat, O., Catasta, M., Wei, J., Meier-Hellstern, K., Eck, D., Dean, J., Petrov, S., Fiedel, N.: Palm: Scaling language modeling with pathways. CoRR **abs/2204.02311** (2022). https://doi.org/10.48550/arXiv.2204.02311, `https://doi.org/10.48550/arXiv.2204.02311`

18. Churchill, B.R., Padon, O., Sharma, R., Aiken, A.: 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, `https://doi.org/10.1145/3314221.3314596`

19. Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., Schulman, J.: Training verifiers to solve math word problems. CoRR **abs/2110.14168** (2021), `https://arxiv.org/abs/2110.14168`

20. Cotroneo, D., Improta, C., Liguori, P., Natella, R.: Vulnerabilities in AI code generators: Exploring targeted data poisoning attacks. CoRR **abs/2308.04451** (2023). https://doi.org/10.48550/arXiv.2308.04451, `https://doi.org/10.48550/arXiv.2308.04451`

21. Ding, Y., Steenhoek, B., Pei, K., Kaiser, G., Le, W., Ray, B.: Traced: Execution-aware pre-training for source code. arXiv preprint arXiv:2306.07487 (2023)

22. Ellis, K., Wong, C., Nye, M.I., Sablé-Meyer, M., Morales, L., Hewitt, L.B., Cary, L., Solar-Lezama, A., Tenenbaum, J.B.: 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, `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., Singh, A., Merenstein, A., Su, W., Smolka, S.A., Zadok, E., Balasubramanian, N.: Specnfs: A challenge dataset towards extracting formal models from natural language specifications. In: Calzolari, N., Béchet, F., Blache, P., Choukri, K., Cieri, C., Declerck, T., Goggi, S., Isahara, H., Maegaard, B., Mariani, J., Mazo, H., Odijk, J., Piperidis, S. (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, `https://doi.org/10.1561/2500000010`

27. Hahn, C., Schmitt, F., Tillman, J.J., Metzger, N., Siber, J., Finkbeiner, B.: Formal specifications from natural language. CoRR **abs/2206.01962** (2022). https://doi.org/10.48550/arXiv.2206.01962, `https://doi.org/10.48550/arXiv.2206.01962`

28. Hendler, J.: Understanding the limits of ai coding. Science **379**(6632), 548–548 (2023)

29. Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., Steinhardt, J.: Measuring coding challenge competence with APPS. In: Vanschoren, J., Yeung, S. (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-Abstract-round2.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., Wang, C., Yang, M., Codas, A., Encarnación, M., Lahiri, S.K., Musuvathi, M., Gao, J.: Fault-aware neural code rankers. In: NeurIPS (2022), `http://papers.nips.cc/paper_files/paper/2022/hash/5762c579d09811b7639be2389b3d07be-Abstract-Conference.html`

32. Key, D., Li, W., Ellis, K.: I speak, you verify: Toward trustworthy neural program synthesis. CoRR **abs/2210.00848** (2022). https://doi.org/10.48550/arXiv.2210.00848, `https://doi.org/10.48550/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, `https://doi.org/10.1145/360248.360252`

34. Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., Yih, W., Fried, D., Wang, S.I., Yu, T.: DS-1000: A natural and reliable benchmark for data science code generation. In: Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J. (eds.) International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA. Proceedings of

Machine Learning Research, vol. 202, pp. 18319–18345. PMLR (2023), `https://proceedings.mlr.press/v202/lai23b.html`

35. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.: Verus: Verifying rust programs using linear ghost types (extended version) (2023)

36. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International conference on logic for programming artificial intelligence and reasoning. pp. 348–370. Springer (2010)

37. Li, Y., Lin, Z., Zhang, S., Fu, Q., Chen, B., Lou, J., Chen, W.: On the advance of making language models better reasoners. CoRR **abs/2206.02336** (2022). https://doi.org/10.48550/arXiv.2206.02336, `https://doi.org/10.48550/arXiv.2206.02336`

38. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al.: Competition-level code generation with alphacode. Science **378**(6624), 1092–1097 (2022)

39. Liu, C., Wu, X., Feng, Y., Cao, Q., Yan, J.: Towards general loop invariant generation via coordinating symbolic execution and large language models. arXiv preprint arXiv:2311.10483 (2023)

40. Liu, J., Xia, C.S., Wang, Y., Zhang, L.: 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, `https://doi.org/10.48550/arXiv.2305.01210`

41. Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al.: Self-refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems **36** (2024)

42. Manna, Z., Waldinger, R.: Knowledge and reasoning in program synthesis. Artificial intelligence **6**(2), 175–208 (1975)

43. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Communications of the ACM **14**(3), 151–165 (1971)

44. Misu, M.R.H., Lopes, C.V., Ma, I., Noble, J.: Towards ai-assisted synthesis of verified dafny methods. arXiv preprint arXiv:2402.00247 (2024)

45. Mohammed, N., Lal, A., Rastogi, A., Roy, S., Sharma, R.: Enabling memory safety of c programs using llms. arXiv preprint arXiv:2404.01096 (2024)

46. Mondal, R., Tang, A., Beckett, R., Millstein, T., Varghese, G.: 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., Inala, J.P., Wang, C., Gao, J., Solar-Lezama, A.: 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, `https://doi.org/10.48550/arXiv.2303.08774`

49. Pan, L., Saxon, M., Xu, W., Nathani, D., Wang, X., Wang, W.Y.: Automatically correcting large language models: Surveying the landscape of diverse self-correction strategies. arXiv preprint arXiv:2308.03188 (2023)

50. Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., Karri, R.: 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, `https://doi.org/10.1109/SP46214.2022.9833571`

51. Pei, K., Bieber, D., Shi, K., Sutton, C., Yin, P.: Can large language models reason about program invariants? In: Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J. (eds.) International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA. Proceedings of Machine Learning Research, vol. 202, pp. 27496–27520. PMLR (2023), `https://proceedings.mlr.press/v202/pei23a.html`

52. Perry, N., Srivastava, M., Kumar, D., Boneh, D.: Do users write more insecure code with AI assistants? CoRR **abs/2211.03622** (2022). https://doi.org/10.48550/arXiv.2211.03622, `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, `https://doi.org/10.1145/2908080.2908093`

54. Ryan, G., Jain, S., Shang, M., Wang, S., Ma, X., Ramanathan, M.K., Ray, B.: Code-aware prompting: A study of coverage guided test generation in regression setting using llm. arXiv preprint arXiv:2402.00097 (2024)

55. Sandoval, G., Pearce, H., Nys, T., Karri, R., Garg, S., Dolan-Gavitt, B.: Lost at C: A user study on the security implications of large language model code assistants. In: Calandrino, J.A., Troncoso, C. (eds.) 32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023. USENIX Association (2023), `https://www.usenix.org/conference/usenixsecurity23/presentation/sandoval`

56. Saunders, W., Yeh, C., Wu, J., Bills, S., Ouyang, L., Ward, J., Leike, J.: Self-critiquing models for assisting human evaluators. arXiv preprint arXiv:2206.05802 (2022)

57. Shi, F., Fried, D., Ghazvininejad, M., Zettlemoyer, L., Wang, S.I.: 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, `https://doi.org/10.18653/v1/2022.emnlp-main.231`

58. Shi, K., Dai, H., Ellis, K., Sutton, C.: 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. 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: 2022-07-26

62. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M.K., Alur, R.: TRANSIT: specifying protocols with concolic snippets. In: Boehm, H., 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, `https://doi.org/10.1145/2491956.2462174`

63. Vaithilingam, P., Zhang, T., Glassman, E.L.: Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In: Barbosa, S.D.J., Lampe, C., Appert, C., Shamma, D.A. (eds.) CHI '22: CHI Conference on Human Factors in Computing Systems, New Orleans, LA, USA, 29 April 2022 - 5 May 2022, Extended Abstracts. pp. 332:1–332:7. ACM (2022). https://doi.org/10.1145/3491101.3519665, `https://doi.org/10.1145/3491101.3519665`

64. Waldinger, R.J., Lee, R.C.: Prow: A step toward automatic program writing. In: Proceedings of the 1st international joint conference on Artificial intelligence. pp. 241–252 (1969)

65. Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., Zhou, D.: Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171 (2022)

66. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E.H., Le, Q.V., Zhou, D.: Chain-of-thought prompting elicits reasoning in large language models. In: NeurIPS (2022), `http://papers.nips.cc/paper_files/paper/2022/hash/9d5609613524ecf4f15af0f7b31abca4-Abstract-Conference.html`

67. Wen, C., Cao, J., Su, J., Xu, Z., Qin, S., He, M., Li, H., Cheung, S.C., Tian, C.: Enchanting program specification synthesis by large language models using static analysis and program verification. arXiv preprint arXiv:2404.00762 (2024)

68. Wu, H., Barrett, C., Narodytska, N.: Lemur: Integrating large language models in automated program verification (2023)

69. 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, `https://doi.org/10.1145/3487569`

70. Yin, P., Li, W., Xiao, K., Rao, A., Wen, Y., Shi, K., Howland, J., Bailey, P., Catasta, M., Michalewski, H., Polozov, O., Sutton, C.: 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, `https://doi.org/10.18653/v1/2023.acl-long.9`

71. Yin, P., Neubig, G.: A syntactic neural model for general-purpose code generation. In: Barzilay, R., Kan, M. (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, `https://doi.org/10.18653/v1/P17-1041`

72. Zelikman, E., Huang, Q., Poesia, G., Goodman, N.D., Haber, N.: Parsel: A (de-) compositional framework for algorithmic reasoning with language models. arXiv preprint arXiv:2212.10561 (2023)

73. Zhang, T., Yu, T., Hashimoto, T., Lewis, M., Yih, W., Fried, D., Wang, S.: Coder reviewer reranking for code generation. In: Krause, A., Brunskill, E., Cho, K., Engelhardt, B., Sabato, S., Scarlett, J. (eds.) International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA. Proceedings of Machine Learning Research, vol. 202, pp. 41832–41846. PMLR (2023), `https://proceedings.mlr.press/v202/zhang23av.html`

74. 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, p. 129414G. International Society for Optics and Photonics, SPIE (2023). https://doi.org/10.1117/12.3011627, https://doi.org/10.1117/12.3011627

A   Appendix

A.1   An Analytical Model for Clover Reconstruction Tests

As described in Section 3.2, all but one of the six Clover consistency checks relies on reconstructing one of the components (see Figure 1). These reconstructions rely on assumptions about the LLM model used for reconstruction that have, until now, been implicit. In this section, we make these assumptions explicit and provide a theoretical model and analysis for those assumptions. For the purpose of the analysis we focus on a single directed edge from domain A to domain B (e.g., code to docstring).

Assume each domain is equipped with a semantic equivalence relation, denoted by $\equiv$. Each domain can therefore be partitioned into equivalence classes. For $X \in \{A, B\}$, we use $e(X)$ to denote the set of equivalence classes of X, and for $x \in X$ we use $[x]$ to denote the equivalence class x belongs to. For docstrings, the equivalence relation represents semantic equivalence as understood by a human expert; for code, the equivalence relation is functional equivalence; and for annotations, it is logical equivalence.

We further assume a *ground truth consistency relation* between A and B, denoted by $G \subseteq A \times B$. The ground truth consistency represents the consistency we assume to exist between docstrings, annotations, and code, as described in Section 3.2. We assume the consistency relation satisfies the following properties that link it to the equivalence relation: For any $x, x' \in A$ and $y, y' \in B$, $(x \equiv x' \wedge y \equiv y') \to ((x, y) \in G \leftrightarrow (x', y') \in G)$ and $((x, y) \in G \wedge (x', y') \in G) \to (x \equiv x' \leftrightarrow y \equiv y')$. That is, consistency is preserved when substituting equivalent objects, and any object may be consistent with at most one equivalence class from the other domain.

We now formally define and analyze the *single-edge Clover consistency test*, which aims to be an approximate test for G. For the analysis, we assume a probability distribution $\mathcal{D}$ on $A \times B$. The test relies on a *transfer model* and the analysis assumes it is *transfer-rational*, as defined below.

Definition 1 (Transfer Model). *Given domains A and B, a transfer model for A and B is a function $M : A \times B \to \mathbb{R}$ such that for each $x \in A$, $M(x, \cdot)$ is a probability distribution over B. Here $M(x, y)$ denotes the probability of transferring $x \in A$ to $y \in B$.*

Definition 2 (Transfer-Rational Model). *Let M be a transfer model for A and B. We say M is transfer-rational if for each $x \in A$ there is a unique $[y] \in e(B)$ that maximizes $\sum_{y' \in [y]} M(x, y')$. In this case, we define the transfer function of M, $f^M : A \to e(B) = \lambda x. \arg\max_{[y] \in e(B)} \sum_{y' \in [y]} M(x, y')$.*

Intuitively, the transfer model is meant to approximate a mapping based on the ground truth consistency (G). In the context of Clover, the domains are among docstring, annotation, and code, and the transfer model is given by an LLM (GPT-4). For example, when A is docstrings and B is annotations, the

distribution $M(x,\cdot)$ represents the output distribution of GPT-4 on an input docstring x with a suitable prompt for generating an annotation corresponding to the docstring x. In our evaluation, we use 3 tries with Dafny feedback to run the reconstruction test. In this case, the transfer model is given by this combined use of GPT-4 and Dafny.

We now fix a transfer-rational model M, and define the single-edge Clover consistency check.

Definition 3 (Single-Edge Clover Consistency Check). *For input $x \in A, y \in B$, the single-edge Clover consistency check (for the edge from A to B) is a procedure that draws y' from the distribution $M(x,\cdot)$, and then accepts if $y' \equiv y$ and otherwise rejects.*

Note that the check relies on being able to check equivalence in domain B.[13]

We now analyze the probability that the single-edge Clover consistency check is correct. Our analysis relies on two assumptions: one relating the transfer model M with the ground truth consistency G, and another ensuring that M's distributions are concentrated.

Assumption 1 (Consistency Alignment) *Let c_1 be the probability that $y \in f^M(x)$ when x, y are sampled from $A \times B$ according to $\mathcal{D}$ conditioned on $(x, y) \in G$. Similarly, let c_0 be the probability that $y \in f^M(x)$ when x, y are sampled from $A \times B$ according to $\mathcal{D}$ conditioned on $(x, y) \notin G$. We assume that c_1 is close to 1, and c_0 is close to 0.*

Assumption 2 (Concentration) *Consider x, y sampled from $A \times B$ according to $\mathcal{D}$ conditioned on $(x, y) \in G$ and $y \in f^M(x)$. We assume that for some significant $0 < l \leq 1$ (e.g., 30%), the following holds with probability $\geq p_c$ (p_c close to 1): $\sum_{y' \in f^M(x)} M(x, y') \geq l$. Similarly, consider x, y sampled from $A \times B$ according to $\mathcal{D}$ conditioned on $(x, y) \notin G$ and $y \notin f^M(x)$. We assume that for some negligible u, the following holds with probability $\geq p_c$: $\max_{[y_1] \in e(Y), [y_1] \neq f^M(x)} \sum_{y_2 \in [y_1]} M(x, y_2) \leq u$.*

Intuitively, the concentration assumption means that with high probability ($\geq p_c$), sampling from M is the same as applying f^M, and specifically that the second most likely equivalence class is much less likely than the maximal one (i.e., the one given by f^M).

Theorem 3. *Under Assumptions 1 (Consistency) and 2 (Concentration), consider (x, y) sampled from $A \times B$ according to $\mathcal{D}$ conditioned on $(x, y) \in G$; the single-edge Clover consistency check will accept (x, y) with probability $A \geq$*

---

[13] We assume a perfect equivalence check to keep the analysis simple and illustrative. In practice, the equivalence tests do incur some imprecision. But accounting for this imprecision using a probabilistic model is cumbersome because the distribution on the equivalence checks Clover performs depends on both the input distribution and on the transfer model.

$l \cdot p_c \cdot c_1$. *Similarly, consider (x, y) sampled from $A \times B$ according to $\mathcal{D}$ conditioned on $(x, y) \notin G$; the single-edge Clover consistency check will will accept with probability $R \leq u \cdot p_c \cdot (1 - c_0) + (1 - p_c)(1 - c_0) + c_0$.*

The proof of Theorem 3 is in Appendix A.3.

Theorem 3 ensures that under our assumptions, the probability of accepting a consistent input is significant, and the probability of accepting an inconsistent input is negligible. We can increase the gap by repeating the reconstruction test several times and accepting if any of them accept. As discussed in Section 4, our evaluation shows the results for both 1 and 10 reconstruction attempts.

From single-edge to full Clover consistency checking. Our analysis focused on a single, directed reconstruction edge from Figure 1, while full Clover consistency checking uses five reconstruction edges and a single verification edge, and accepts only if all six checks accept. We do not attempt to theoretically analyze the full check, because we do not assume the edges to be independent (so multiplying acceptance probabilities is not necessarily meaningful). In our experiments, we empirically measure the acceptance rate of each edge, and also observe that the edges are not independent (see Section A.2). In real experiments, the combined use of GPT-4 and Dafny may not satisfy our assumptions because of the tools' limitations (Dafny may time out or return unknown, and GPT-4 may make mistakes or hallucinate). Especially the u in Assumption 2 could be non-negligible. However, the end-to-end evaluation shows that the six checks together do give promising true positive and false positive rates. The analytical model can be treated as one guide to understanding what properties of the reconstruction model are helpful for ensuring accurate reconstruction results.

A.2   Explaining the Evaluation

edge	Accept Correct A	Accept Incorrect R^{C1}	Accept Incorrect R^{C2}	Accept Incorrect R^{C3}	Accept Incorrect R^{C6}
anno-sound	1	−	−	−	−
anno-complete	0.88	−	−	−	−
doc2anno	0.85	0.05	0	−	0
anno2doc	1	0.30	0.4	−	0.30
code2doc	0.97	0.05	−	0.15	0.28
doc2code	0.82	−	−	0.48	0.03

Table 6: Empirically measured values for A and R when $k = 1$. Entries shown as "−" are omitted because for that category and check, the assumptions of the analytical model are violated.

Here, we empirically estimate the values of A and R from Theorem 3 based on our experiments. That is, we estimate the acceptance rate for correct and

incorrect inputs for each directed edge. Each cell in Table 6 represents the percentage of reconstructed components that successfully pass the equivalence check in the five categories: ground-truth , C1, C2, C3 and C6 (Table 3).[14]

As mentioned above, in the first column, the discrepancy between the measured acceptance rate and the ideal perfect acceptance rate comes partly from reconstruction failures and partly from equivalence checker failures. For example, the doc2anno acceptance rate is 0.85, not 1. Apart from the failure to generate the correct annotation, there are also cases where the generated annotation is correct but unable to be verified by our annotation equivalence checking template in Section 4.1 (See Appendix A.7 for an example).

Overall, the measured aggregated acceptance rate for the first column is 0.75. This is higher than would be expected if each check were independent (the product of the entire column is 0.59). This is because, in practice, they are not independent: easier examples that pass the tests on one edge tend to also pass the tests on other edges. In C2 and C6, doc2anno has a zero acceptance rate, and the overall acceptance is zero. In C1 and C3, none of the edges are zero, but the overall acceptance is still zero. Note that the anno2doc and code2doc acceptance rate is high for C1, C2, and C6. This is because our current docstring equivalence checker is good at detecting contradictory information but not the addition or omission of information due to a slightly strengthened or weakened annotation.

A.3   Proof of Theorem 3

1. Let (x, y) be sampled from $\mathcal{D}$ with the condition $(x, y) \in G$. From Assumption 1, with probability $\geq c_1$, we have $y \in f^M(x)$. Then, according to Assumption 2 and the perfect equivalence oracle, with probability p_c, the reconstruction from x to y will succeed with probability $\geq l$. Therefore, the accept probability is $\geq l \cdot p_c \cdot c_1$, denoted as A.

2. Let (x, y) be sampled from $\mathcal{D}$ with the condition $(x, y) \notin G$. There are 3 cases:
   - From Assumption 1, with probability c_0, there is $y \in f^M(x)$, and it is trivial that the accept probability ≤ 1.
   - With probability $1 - c_0$, there is $y \notin f^M(x)$. Then from Assumption 2, with probability p_c, the reconstruction from x to y will succeed with probability $\leq u$.
   - Finally, the last case is that the bounds in Assumption 2 do not hold, which will happen with probability $(1 - c_0)(1 - p_c)$. Clearly, in this case, the accept probability ≤ 1.

   By aggregating all the cases, the accept probability is $\leq c_0 + (1 - c_0) \cdot p_c \cdot u + (1 - c_0)(1 - p_c)$.

---

[14] Note that our incorrect examples are constructed with the aim of making them hard to reject, i.e., by considering only the cases that can pass Dafny verification. The measured values for R are thus likely to be higher than the value for a more natural distribution.

A.4   Discussion

A.4.1   Limitations There are many limitations in the proposed paradigm. For one, the capabilities of LLMs (GPT-4 in particular) are limited. The generation of docstrings, annotations, and code also has inherent limitations. For example, our use of annotations is only for specifying functionality, not implementation details, e.g., an annotation can force an array to be sorted but cannot easily restrict the algorithm used for sorting. In this paper, as a first step, we only aim to check functional consistency (correctness), not the performance of the implementation.

As mentioned in Section 3, if the oracle used for consistency checking is misaligned with human understanding (ground-truth), e.g., if it interprets a sorting algorithm as getting the maximum value, there is no way to correct it without human intervention. But in practice, we think this will rarely happen (see Assumption 1). As another example, if the docstring, annotations, and code all miss the same edge case, the error cannot be detected. While such an example is internally consistent, it may not be consistent with human understanding of good coding practice. So far, we haven't detected this in our experiments, but these and other problems may appear in more complicated examples. To acheive our eventual vision for Clover, we expect that additional breakthroughs, or additional human-in-the-loop steps, or both, may be needed.

A.4.2   Clover Variants Clover checks three components for consistency. However, other variants are possible. Currently, most attempts at code generation produce only the code and docstring. We expect that a Clover-like approach with only code and docstrings would help detect some inconsistencies, but would not ensure implementation correctness, as docstrings are not sufficiently precise. Incorporating unit tests into Clover is a potential improvement we've earmarked for future endeavors. We recognize the potential advantages of unit tests; however, they come with their own set of limitations. Admittedly, in certain scenarios, unit tests can provide a quick and effective sanity check on system functionalities. However, generating unit tests can sometimes prove more complex than creating annotations. Unit tests, if not transparent, can be difficult or even impossible to explain, eroding user confidence due to their opacity. If an LLM is adept at producing effective unit tests, it suggests an ability to anticipate execution outcomes to a certain degree. However, full-fledged execution with numerous computational steps remains an unsolved challenge for LLMs. Additionally, compared to annotations, unit tests offer a less robust assurance of system correctness.

A.4.3   Future Research A successful Clover paradigm relies on many components. To maximize the capabilities of Clover, there are several foundational topics that should be explored. Each of these areas can be advanced individually, and notably, they possess wider applicability beyond just the scope of Clover.

One foundational element is the ability to generate high-quality code, annotations, and docstrings. Clearly, the verification phase cannot compensate for

poor generation, it can only detect and flag such examples. Better equivalence checking would also improve Clover's abilities. Currently, it is most challenging to perform equivalence checks on docstrings. Equivalence of annotations relies on the logical power of solvers in the back end of Dafny, whose performance and capabilities can be improved. Equivalence checking for code is also challenging; techniques like fuzzing and concolic testing (and even full formal equivalence checking) could be leveraged to improve this step.

A.4.4 Data Contamination We want to point out that the current version of CloverBench has some limitations. We hand-crafted it starting with simple textbook-level examples so as to have a baseline for more advanced work. But we must acknowledge the possibility of indirect data contamination. While we expect most of our examples are not explicitly present in the training data (Dafny is not a widely-used language, and we wrote the examples ourselves), there's a considerable chance that GPT-4 has encountered analogous data in the past. Even if only code with a similar functionality in another language has been seen in the training data, our hand-crafted examples can be affected. Some soft evidence for this is the observation that GPT-4 can sometimes generate the correct code even with incomplete docstrings or annotations. We noticed that often, a descriptive function signature alone can be quite revealing. To mitigate this potential bias in our experiments, we opted to replace the function names with generic, non-descriptive identifiers. In future work, we plan to update Clover-Bench with more sophisticated examples, which we hope will help mitigate the risk of inaccurate conclusions due to data contamination in future experiments.

A.4.5 Reasons for Reconstruction Failure using GPT-4 We have observed that GPT-4 is not very capable at producing correct syntax for the latest version (4.0.0) of Dafny, likely due to limited training data. One can imagine that an LLM trained or fine-tuned on Dafny 4.0.0 would easily acheive a higher acceptance rate. Some evidence that GPT-4 is not familiar with the current Dafny syntax is as follows. Annotations must include a `modifies clause` or `reads clause` for methods that access memory. In particular, `reads array` is required when the method reads from `array`, and GPT-4 misses it almost 100% of the time on its first try at generation. Luckily, with Dafny's compiler-generated error messages, GPT-4 is often able to add the needed `modifies clause` or `reads clause`. Another example is that Dafny used to require annotations to be separated by semicolons, or assert explicitly that an array is not null `requires array!=null` in the pre-conditions. These are not required any more, but GPT-4 still largely adheres to those deprecated rules.

A.5 Open Model Results

We present the results of CodeLlama-34b on Clover tests in Table 7. The results indicate that CodeLlama-34b is incapable of dealing with Dafny code, neither verification logic. In the early stages of this project, we briefly experimented with

several other open models and discovered that most are significantly lacking in their ability to handle tasks related to program verification. While many open models have shown impressive results in other popular datasets, our dataset offers a perspective on the extent of the gap in knowledge coverage between open models and GPT-4. Specifically, open models perform notably poorly with low-resource languages. But GPT-4 highlights the potential of AI in verification tasks. Therefore, one of our goals is to showcase its promise and to garner more attention towards integrating verification into the workflow from both AI and verification specialists.

Metric	anno_sound	anno_complete	code2doc	Clover 6 edges
ground truth	60/60	6/60	8/60	2/60
Metric	**doc2code**	**anno2doc**	**doc2anno**	
ground truth	2/60	48/60	2/60	

Table 7: Ground Truth Acceptance by CodeLlama-34b

A.6   Dafny Feedback Example

In this example, the first try at generating annotations (Listing 1.12) fails given the input in Listing 1.11. But after receiving the error message in Listing 1.13, GPT-4 is able to add the necessary annotations `modifies a`.

Listing 1.12: Generated Annotations for UpdateElements

Listing 1.11: UpdateElements Example Input

```
method foo(a: array<int>)
  //TOFILL
{
  a[4], a[8] := a[4] + 3, a[8] + 1;
  a[7], a[8] := 516, a[8] - 1;
}
```

```
requires a != null && a.Length > 8
ensures a[4] == old(a[4]) + 3
ensures a[7] == 516
ensures a[8] == old(a[8])
```

Listing 1.13: Dafny Feedback

```
Error: assignment might
    update an array element
    not in the enclosing
    context's modifies clause
```

A.7   Supplimentary Template and Examples

In Listing 1.15, we give a template for verifying annotation equivalence for the ground-truth example `max_array` (Listing 1.14). Annotation equivalence checking is done by verifying the template with Dafny's verifier. If the lemmas pre_eq

and `post_eq` are both verified, then it means that Dafny has successfully verified the equivalence of pre- and postconditions respectively.

In more detail, `predicate pre_original` states the full preconditions of the ground-truth example, and `predicate post_original` states the full postconditions. `predicate pre_gen`'s body will be replaced by the generated preconditions and `predicate post_gen`'s body will be replaced by the generated postconditions. The lemma `pre_eq` states that the generated preconditions are true if and only if the original preconditions are true. The lemma `post_eq` states that the generated postconditions are true if and only if the original postconditions are true. The above example is simple enough to be proven by Dafny's verifier.

Note that the template is sound but not complete, that is, there could be cases when two predicates are indeed equivalent but Dafny cannot prove it. An example is shown in Listing 1.16.

Listing 1.14: maxArray

```
method maxArray(a: array<int>) returns (m: int)
  requires a.Length >= 1
  ensures forall k :: 0 <= k < a.Length ==> m >= a[k]
  ensures exists k :: 0 <= k < a.Length && m == a[k]
{
  m := a[0];
  var index := 1;
  while (index < a.Length)
    invariant 0 <= index <= a.Length
    invariant forall k :: 0 <= k < index ==> m >= a[k];
    invariant exists k :: 0 <= k < index && m == a[k];
    decreases a.Length - index
  {
    m := if m>a[index] then  m else a[index];
    index := index + 1;
  }
}
```

Listing 1.15: Annotation Equivalence Checking Template for maxArray

```
predicate pre_original(a: array<int>,m: int)
  reads a
{
  ( a.Length >= 1)
}

predicate pre_gen(a: array<int>,m: int)
  reads a
{
  true // (#PRE) && ... (#PRE)
}

lemma pre_eq(a: array<int>,m: int)
```

```
  ensures pre_original(a,m ) <==> pre_gen(a,m )
{
}

predicate post_original(a: array<int>,m: int)
  requires pre_original(a,m)
  reads a
{
  ( forall k :: 0 <= k < a.Length ==> m >= a[k]) &&
  ( exists k :: 0 <= k < a.Length && m == a[k])
}

predicate post_gen(a: array<int>,m: int)
  requires pre_original(a,m)
  reads a
{
  true // (#POST) && ... (#POST)
}

lemma post_eq(a: array<int>,m: int)
  requires pre_original(a,m )
  requires pre_gen(a,m )
  ensures post_original(a,m ) <==> post_gen(a,m )
{
}
```

Listing 1.16: Instantiated Annotation Equivalence Checking Template for only_once. The original and generated postconditions describe the same property: element key only appears once in the array a. But they cannot be verified as equivalent by the annotation template. Lemma post_eq will fail with an empty body.

```
predicate pre_original<T(==)>(a: array<T>,key: T,b:bool)
  reads a
{
  true
}

predicate pre_gen<T(==)>(a: array<T>,key: T,b:bool)
  reads a
{
  true
}

lemma pre_eq<T(==)>(a: array<T>,key: T,b:bool)
  ensures pre_original(a,key,b ) <==> pre_gen(a,key,b )
{
}

predicate post_original<T(==)>(a: array<T>,key: T,b:bool)
```

```
  requires pre_original (a,key ,b)
  reads a
{
  ( (multiset(a[..])[key] ==1 ) <==> b)
}

predicate post_gen <T(==) >(a: array <T>,key: T,b:bool)
  requires pre_original (a,key ,b)
  reads a
{
  (b <==> ((exists i :: 0 <= i < a.Length && a[i] == key) &&
    (forall i, j :: 0 <= i < j < a.Length && a[i] == key ==>
    a[j] != key)))
}

lemma post_eq <T(==) >(a: array <T>,key: T,b:bool)
  requires pre_original (a,key ,b )
  requires pre_gen (a,key ,b )
  ensures post_original (a,key ,b ) <==> post_gen (a,key ,b )
{
}
```

A.8 Input/Output Tests Template in CloverBench

Here is an example of the input/output test code we use in CloverBench for code equivalence check. We compare the output from Listing 1.17 with the output when the `method maxArray` implementation is replaced by the generated code. If the outputs are equal, we consider the two codes to be equivalent.

Listing 1.17: Ground truth unit tests for max_array

```
method maxArray (a: array <int >) returns (m: int)
  requires a.Length >= 1
  ensures forall k :: 0 <= k < a.Length ==> m >= a[k]
  ensures exists k :: 0 <= k < a.Length && m == a[k]
{
  m := a [0];
  var index := 1;
  while (index < a.Length)
    invariant 0 <= index <= a.Length
    invariant forall k :: 0 <= k < index ==> m >= a[k]
    invariant exists k :: 0 <= k < index && m == a[k]
    decreases a.Length - index
  {
    m := if m>a[index] then  m else a[index];
    index := index + 1;
  }
}
```

```dafny
method TestMethod(){
  var a1 := new int[5];
  a1[0] := 1; a1[1] := 2; a1[2] := 3; a1[3] := 4; a1[4] := 5;
  var test1 := maxArray(a1);
  print("Test 1: maxArray([1,2,3,4,5]) = ", test1, "\n");

  var a2 := new int[5];
  a2[0] := -1; a2[1] := -2; a2[2] := -3; a2[3] := -4; a2[4]
    := -5;
  var test2 := maxArray(a2);
  print("Test 2: maxArray([-1,-2,-3,-4,-5]) = ", test2, "\n")
    ;

  var a3 := new int[3];
  a3[0] := 0; a3[1] := 0; a3[2] := 0;
  var test3 := maxArray(a3);
  print("Test 3: maxArray([0,0,0]) = ", test3, "\n");

  var a4 := new int[2];
  a4[0] := 5; a4[1] := 10;
  var test4 := maxArray(a4);
  print("Test 4: maxArray([5,10]) = ", test4, "\n");

  var a5 := new int[1];
  a5[0] := 99;
  var test5 := maxArray(a5);
  print("Test 5: maxArray([99]) = ", test5, "\n");
}

method Main(){
  TestMethod();
}
```

GPT-4 Prompt

code2anno:
You are an expert in Dafny. Fill in the weakest precondition and
strongest postconditions for the dafny programs so that the dafny
programs can be verified. Do not change provided code. Exclude
"requires true", "requires array!=null", "requires natural number
>=0". Do not assume input array or seq is non-empty. Do not
assume input integers are non-negative unless necessary. Replace
the //TOFILL string with the actual pre- and postconditions.
Return the whole verifiable program.

anno2code:
You are an expert in dafny. You are given a dafny program with
annotations. Replace //TOFILL with the actual dafny code so that
it can be verified. Return the whole program. If loop is needed,
use while instead of for. Do not use helper functions. DO NOT
modify the function signature and annotations.

doc2anno:
You are an expert in Dafny. Generate the weakest preconditions
and strongest postconditions for the dafny programs based on the
docstring. Do not change the provided code. Exclude "requires
true", "requires array!=null", "requires natural number >=0".
Do not assume the input array or seq is non-empty. Do not use
self-defined functions. Do not use int.MaxValue or int.MinValue.
Do not assume input integers are non-negative unless necessary.
Return only the annotations in code format starting with ''' and
end with '''. Do not return method implementation.

anno2doc:
You are an expert in dafny. Give one docstring of the given dafny
annotation. Make sure to capture all details described in the
annotation.

code2doc:
You are an expert in dafny. Give one docstring of the given dafny
code's functional behavior. Do not mention implementation details.
Assume 'assert' as preconditions.

doc2code:
You are an expert in dafny. You are given a dafny program
docstring. Replace //TOFILL with the actual dafny code without
annotation. Return the whole program.

docstring equivalence checker:
Determine if two docstrings describe the exact same functional
behavior of a dafny program. Make sure all details are exactly
the same.

A.9   Wrong Ground-Truth programs in MBPP-DFY-50 [44]

Task 472 docstring states arrays can be empty or non-empty but the annotations require the array to be non-empty.

Listing 1.18: mbpp-50 task 472

```
/* task_description: Write a method in Dafny to check whether
    the given integer array contains consecutive numbers or
    not.
preconditions:
- There are no preconditions, the method will always work.
    Arrays can be empty or non-empty.
postconditions:
- If the array contains consecutive numbers, the result is
    true
- If the array does not contain consecutive numbers, the
    result is false.
*/
method ContainsConsecutiveNumbers(a: array<int>) returns (
    result: bool)
    requires a.Length>0
    ensures result <==> (exists i :: 0 <= i < a.Length - 1 &&
     a[i] + 1 == a[i + 1])
{
    result := false;
    for i := 0 to a.Length - 1
        invariant 0 <= i <= a.Length - 1
        invariant result <==> (exists k :: 0 <= k < i && a[k]
     + 1 == a[k + 1])
    {
        if a[i] + 1 == a[i + 1] {
            result := true;
            break;
        }
    }
}
```

Task 567 docstring states arrays can be empty or have any length but the annotations require the array to be non-empty.

Listing 1.19: mbpp-50 task 567

```
/* task_description: Write a method in Dafny to check whether
    a specified array is sorted.
preconditions:
- There are no preconditions, the method will always work.
    Arrays can be empty or have any length.
postconditions:
- If the method returns true, the array is sorted in non-
    decreasing order.
```

```
- If the method returns false, the array is not sorted in non
    -decreasing order.
*/

method IsSorted(a: array<int>) returns (sorted: bool)
    requires a.Length > 0
    ensures sorted <== forall i, j :: 0 <= i < j < a.Length
    ==> a[i] <= a[j]
    ensures !sorted ==> exists i, j :: 0 <= i < j < a.Length
    && a[i] > a[j]
{
    sorted := true;
    for i := 0 to a.Length - 1
        invariant 0 <= i < a.Length
        invariant sorted <== forall k, l :: 0 <= k < l < i
    ==> a[k] <= a[l]
        invariant !sorted ==> exists k :: 0 <= k < i && a[k]
    > a[k+1]
    {
        if a[i] > a[i + 1]
        {
            sorted := false;
            break;
        }
    }
    sorted := sorted;
}
```

Task 576 annotations do not state the condition when the return is false.

Listing 1.20: mbpp-50 task 576

```
/* task_description: Write a method in Dafny to check whether
    a list is sublist of another or not.
preconditions:
- There are no preconditions, the method will always work.
    Sequences are always not null.
postconditions:
- If the result is true, then the subsequence exists in the
    main sequence.
- If the result is false, then the subsequence does not exist
    in the main sequence.
*/
method IsSublist(sub: seq<int>, main: seq<int>) returns (
    result: bool)
    ensures true <== (exists i :: 0 <= i <= |main| - |sub| &&
    sub == main[i..i + |sub|])
{
    if |sub| > |main| {
        return false;
    }
```

```
    for i := 0 to |main| - |sub| + 1
        invariant 0 <= i <= |main| - |sub| + 1
        invariant true <== (exists j :: 0 <= j < i && sub ==
    main[j..j + |sub|])
    {
        if sub == main[i..i + |sub|] {
            result := true;
        }
    }
    result := false;
}
```

Task 632 docstring says there are no preconditions, but there is a Dafny annotation requiring the array to have length at least 2. (Note that we rewrote the MoveZeroesToEnd to get rid of the swap helper method to run Clover consistency check, as Clover does not yet support helper methods.)

Listing 1.21: mbpp-50 task 632

```
/* task_description: Write a method in Dafny to move all
   zeroes to the end of the given array.
preconditions:
- There are no preconditions, the method will always work.
postconditions:
- The length of the output array must be the same as the
   length of the input array.
- All zeroes in the input array are at the end of the output
   array.
- The relative order of the non-zero elements should be the
   same as in the input array.
- The number of zeroes in the input and output arrays should
   be the same.
*/

method MoveZeroesToEnd(arr: array<int>)
    requires arr.Length >= 2
    modifies arr
    // Same size
    ensures arr.Length == old(arr.Length)
    // Zeros to the right of the first zero
    ensures forall i, j :: 0 <= i < j < arr.Length && arr[i]
    == 0 ==> arr[j] == 0
    // The final array is a permutation of the original one
    ensures multiset(arr[..]) == multiset(old(arr[..]))
    // Relative order of non-zero elements is preserved
    ensures forall n, m /* on old array */:: 0 <= n < m < arr
    .Length && old(arr[n]) != 0 && old(arr[m]) != 0 ==>
            exists k, l /* on new array */:: 0 <= k < l < arr
    .Length && arr[k] == old(arr[n]) && arr[l] == old(arr[m])
    //ensures IsOrderPreserved(arr[..], old(arr[..]))
```

```
    // Number of zeros is preserved
{
    var i := 0;
    var j := 0;

    assert 0 <= i  <= arr.Length;
    assert forall k :: 0 <= k < arr.Length ==> arr[k] == old(
    arr[k]);
    //assert(forall n, m :: 0 <= n < m < arr.Length  ==> arr[
    n] == old(arr[n]) && arr[m] == old(arr[m]));
    while j < arr.Length
        invariant 0 <= i <= j <= arr.Length
        // Elements to the right of j are unchanged
        invariant forall k :: j <= k < arr.Length ==> old(arr
    [k]) == arr[k]
        // Everything to the left of i is non-zero
        invariant forall k :: 0 <= k < i ==> arr[k] != 0
        // Everything between i and j, but excluding j, is
    zero
        invariant forall k :: i <= k < j ==> arr[k] == 0
        // If there there are zeros, they are to the right of
     i
        invariant forall k :: 0 <= k < j && arr[k] == 0 ==> k
     >= i
        // No new numbers are added, up to j
        invariant forall k :: 0 <= k < j && arr[k] != old(arr
    [k]) ==> exists l :: 0 <= l < j && arr[k] == old(arr[l])
        // The new array up to j is always a permutation of
    the original one
        invariant multiset(arr[..]) == multiset(old(arr[..]))
        // Relative order of non-zero elements is always
    preserved
        //invariant IsOrderPreserved(arr[..], old(arr[..]))
        invariant forall n, m /* on old */:: 0 <= n < m < j
    && old(arr[n]) != 0 && old(arr[m]) != 0 ==>
            exists k, l /* on new */:: 0 <= k < l < i && arr[
    k] == old(arr[n]) && arr[l] == old(arr[m])
    {

        if arr[j] != 0
        {
            if i != j
            {
                assert(arr[j] != 0);
                swap(arr, i, j);
                assert(forall k :: 0 <= k <= j ==> exists l
    :: 0 <= l <= j && arr[k] == old(arr[l]));
            }
            i := i + 1;
        }
```

```
        j := j + 1;
    }
    assert j == arr.Length;
}

method swap(arr: array<int>, i: int, j: int)
    requires arr.Length > 0
    requires 0 <= i < arr.Length && 0 <= j < arr.Length
    modifies arr
    ensures arr[i] == old(arr[j]) && arr[j] == old(arr[i])
    ensures forall k :: 0 <= k < arr.Length && k != i && k !=
     j ==> arr[k] == old(arr[k])
    ensures multiset(arr[..]) == multiset(old(arr[..]))
{
        var tmp := arr[i];
        arr[i] := arr[j];
        arr[j] := tmp;
}
```

Task 644 docstring states input k should be between 0 and the length of the array but the annotations state k is greater than or equal to 2.

Listing 1.22: mbpp-50 task 644

```
/* task_description: Write a method in Dafny to reverse an
    array up to a given k.
preconditions:
- k should be between 0 and the length of the array.
postconditions:
- The input array is modified.
- The values of the array up to k are reversed.
- The values of the array after k remain unchanged.
*/

method ReverseUptoK(s: array<int>, k: int)
    modifies s
    requires 2 <= k <= s.Length
    ensures forall i :: 0 <= i < k ==> s[i] == old(s[k - 1 -
    i])
    ensures forall i :: k <= i < s.Length ==> s[i] == old(s[i
    ])
{
  var l := k - 1;
  var i := 0;
  while (i < l-i)
    invariant 0 <= i <= (l+1)/2;
    invariant forall p :: 0 <= p < i || l-i < p <= l ==> s[p]
     == old(s[l-p]);
    invariant forall p :: i <= p <= l-i ==> s[p] == old(s[p])
    ;
```

```
        invariant forall p :: k <= p < s.Length ==> s[p] ==
    old(s[p])
  {
    s[i], s[l-i] := s[l-i], s[i];
    i := i + 1;
  }

}
```

Task 803 docstring states that if the result is false, there is no integer i such that i * i == n, but the corresponding annotation adds unnecessary bounds making the postcondition a tautology.

Listing 1.23: mbpp-50 task 803

```
/* task_description: Write a method in Dafny to check whether
    the given number is a perfect square or not.
preconditions:
-  n should be non-negative.
postconditions:
- If the result is true, there exists an integer i such that
    i * i == n.
- If the result is false, there is no integer i such that i *
    i == n.
*/

method IsPerfectSquare(n: int) returns (result: bool)
    requires n >= 0
    ensures result == true ==> (exists i: int :: 0 <= i <= n
   && i * i == n)
    ensures result == false ==> (forall a: int :: 0 < a*a < n
    ==> a*a != n)
{
    var i := 0;
    while (i * i < n)
        invariant 0 <= i <= n
        invariant forall k :: 0 <= k < i ==> k * k < n
    {
        i := i + 1;
    }
    return i * i == n;
}
```

A.10 More Detailed Experiment Results

Row No.	test	correct	correct&accept	incorrect	incorrect&accept
1	abs	20	20	0	0
2	all_digits	1	1	19	0
3	array_append	11	11	9	0
4	array_concat	1	1	19	0
5	array_copy	0	0	20	0
6	array_product	12	8	8	0
7	array_sum	2	1	18	0
8	avg	15	15	5	0
9	below_zero	1	0	19	0
10	binary_search	11	5	9	0
11	bubble_sort	0	0	20	0
12	cal_ans	11	11	9	0
13	cal_sum	14	14	6	0
14	canyon_search	0	0	20	0
15	compare	20	20	0	0
16	convert_map_key	0	0	20	0
17	copy_part	0	0	20	0
18	count_lessthan	0	0	20	0
19	double_quadruple	20	18	0	0
20	even_list	0	0	20	0
21	find	13	9	7	0
22	has_close_elements	0	0	20	0
23	insert	0	0	20	0
24	integer_square_root	10	6	10	0
25	is_even	20	20	0	0
26	is_palindrome	4	3	16	0
27	linear_search1	12	9	8	0
28	linear_search2	11	10	9	0
29	linear_search3	17	15	3	0
30	longest_prefix	4	2	16	0
31	max_array	15	13	5	0
32	min_array	16	13	4	0
33	min_of_two	20	20	0	0
34	modify_2d_array	4	2	16	0
35	multi_return	20	20	0	0
36	online_max	0	0	20	0
37	only_once	0	0	20	0
38	quotient	18	14	2	0
39	remove_front	10	5	10	0
40	replace	0	0	20	0
41	return_seven	20	20	0	0
42	reverse	0	0	20	0
43	rotate	0	0	20	0
44	selectionsort	0	0	20	0
45	seq_to_array	0	0	20	0
46	set_to_seq	0	0	20	0
47	slope_search	1	0	19	0
48	swap	19	19	1	0
49	swap_arith	6	6	14	0
50	swap_bitvector	20	20	0	0
51	swap_in_array	18	15	2	0
52	swap_sim	18	18	2	0
53	test_array	16	16	4	0
54	triple	20	20	0	0
55	triple2	20	20	0	0
56	triple3	20	20	0	0
57	triple4	20	20	0	0
58	two_sum	0	0	20	0
59	update_array	17	10	3	0
60	update_map	0	0	20	0

Table 8: End2End generation

Row No.	test	anno-complete	doc2anno	doc2code	anno2doc	code2doc	3-edges
1	abs	A	A	A	A	A	A
2	array_append	A	R	A	A	A	R
3	array_concat	A	R	A	A	A	R
4	array_copy	A	A	A	A	A	A
5	array_product	A	R	A	A	A	R
6	array_sum	A	A	A	A	A	A
7	avg	A	A	A	A	A	A
8	binary_search	A	R	A	A	A	R
9	cal_ans	A	A	A	A	A	A
10	cal_sum	A	A	A	A	A	A
11	compare	A	A	A	A	A	A
12	double_quadruple	A	A	A	A	A	A
13	find	A	A	A	A	A	A
14	is_prime	R	R	A	A	A	R
15	linear_search1	A	A	A	A	A	A
16	linear_search2	A	A	A	A	A	A
17	max_array	A	A	A	A	A	A
18	max_of_two	A	A	A	A	A	A
19	min3	A	A	A	A	A	A
20	min_array	A	A	A	A	A	A
21	min_of_two	A	A	A	A	A	A
22	multi_return	A	A	A	A	A	A
23	pop	A	A	A	A	A	A
24	push	A	A	A	A	A	A
25	quotient	A	A	A	A	A	A
26	remove_front	A	A	A	A	A	A
27	replace	A	R	A	A	A	R
28	return_seven	A	A	A	A	A	A
29	reverse	A	A	A	A	A	A
30	swap	A	A	A	A	A	A
31	swap_arith	A	A	A	A	A	A
32	swap_bitvector	A	A	A	A	A	A
33	swap_in_array	A	A	A	A	A	A
34	swap_sim	A	A	A	A	A	A
35	test_array	A	R	A	A	A	R
36	triple	A	A	A	A	A	A
37	triple2	A	A	A	A	A	A
38	triple3	A	A	A	A	A	A
39	triple4	A	A	A	A	A	A
40	two_sum	A	R	A	A	A	R
41	update_array	R	A	A	A	A	R

Table 9: Verus CLOVER ground truth experiments with k=1

Row No.	test	anno-complete	doc2anno	doc2code	anno2doc	code2doc	3-edges
1	abs	A	A	A	A	A	A
2	array_append	A	A	A	A	A	A
3	array_concat	A	A	A	A	A	A
4	array_copy	A	A	A	A	A	A
5	array_product	A	A	A	A	A	A
6	array_sum	A	A	A	A	A	A
7	avg	A	A	A	A	A	A
8	binary_search	A	R	A	A	A	R
9	cal_ans	A	A	A	A	A	A
10	cal_sum	A	A	A	A	A	A
11	compare	A	A	A	A	A	A
12	double_quadruple	A	A	A	A	A	A
13	find	A	A	A	A	A	A
14	is_prime	R	R	A	A	A	R
15	linear_search1	A	A	A	A	A	A
16	linear_search2	A	A	A	A	A	A
17	max_array	A	A	A	A	A	A
18	max_of_two	A	A	A	A	A	A
19	min3	A	A	A	A	A	A
20	min_array	A	A	A	A	A	A
21	min_of_two	A	A	A	A	A	A
22	multi_return	A	A	A	A	A	A
23	pop	A	A	A	A	A	A
24	push	A	A	A	A	A	A
25	quotient	A	A	A	A	A	A
26	remove_front	A	A	A	A	A	A
27	replace	A	R	A	A	A	R
28	return_seven	A	A	A	A	A	A
29	reverse	A	A	A	A	A	A
30	swap	A	A	A	A	A	A
31	swap_arith	A	A	A	A	A	A
32	swap_bitvector	A	A	A	A	A	A
33	swap_in_array	A	A	A	A	A	A
34	swap_sim	A	A	A	A	A	A
35	test_array	A	R	A	A	A	R
36	triple	A	A	A	A	A	A
37	triple2	A	A	A	A	A	A
38	triple3	A	A	A	A	A	A
39	triple4	A	A	A	A	A	A
40	two_sum	A	R	A	A	A	R
41	update_array	A	A	A	A	A	A

Table 10: Verus CLOVER ground truth experiments with k=10

Row No.	test	anno-complete	doc2anno	doc2code	anno2doc	code2doc	3-edges
1	abs	A	A	A	A	A	A
2	all_digits	A	A	A	A	A	A
3	array_append	A	A	A	A	A	A
4	array_concat	A	A	A	A	A	A
5	array_copy	A	A	A	A	A	A
6	array_product	A	A	A	A	A	A
7	array_sum	A	A	A	A	A	A
8	avg	A	A	A	A	A	A
9	below_zero	A	A	A	A	A	A
10	binary_search	A	A	A	A	A	A
11	bubble_sort	A	A	A	A	A	A
12	cal_ans	A	A	A	A	A	A
13	cal_sum	A	A	A	A	A	A
14	canyon_search	A	A	A	A	A	A
15	compare	A	A	A	A	A	A
16	convert_map_key	R	R	R	A	A	R
17	copy_part	A	A	A	A	A	A
18	count_lessthan	R	A	R	A	A	R
19	double_quadruple	A	A	A	A	A	A
20	even_list	A	R	A	A	A	R
21	find	A	A	A	A	A	A
22	has_close_elements	A	A	A	A	A	A
23	insert	A	A	A	A	A	A
24	integer_square_root	A	A	A	A	A	A
25	is_even	A	A	A	A	A	A
26	is_palindrome	A	A	A	A	A	A
27	linear_search1	A	A	A	A	A	A
28	linear_search2	A	A	A	A	A	A
29	linear_search3	A	A	A	A	A	A
30	longest_prefix	A	A	A	A	A	A
31	max_array	A	A	A	A	A	A
32	min_array	A	A	A	A	A	A
33	min_of_two	A	A	A	A	A	A
34	modify_2d_array	A	R	A	A	A	R
35	multi_return	A	A	A	A	A	A
36	online_max	R	R	R	A	A	R
37	only_once	A	R	A	A	A	R
38	quotient	A	A	A	A	A	A
39	remove_front	A	A	A	A	A	A
40	replace	A	A	A	A	A	A
41	return_seven	A	A	A	A	A	A
42	reverse	A	A	A	A	A	A
43	rotate	A	A	A	A	A	A
44	selectionsort	A	A	A	A	A	A
45	seq_to_array	A	A	A	A	A	A
46	set_to_seq	A	A	A	A	A	A
47	slope_search	A	R	A	A	A	R
48	swap	A	A	A	A	A	A
49	swap_arith	A	A	A	A	A	A
50	swap_bitvector	A	A	A	A	A	A
51	swap_in_array	A	A	A	A	A	A
52	swap_sim	A	A	A	A	A	A
53	test_array	A	A	A	A	A	A
54	triple	A	A	A	A	A	A
55	triple2	A	A	A	A	A	A
56	triple3	A	A	A	A	A	A
57	triple4	A	A	A	A	A	A
58	two_sum	A	A	A	A	A	A
59	update_array	A	A	A	A	A	A
60	update_map	A	R	R	A	A	R

Table 11: CLOVER ground truth experiments with k=10: Each row represents one example. Each column represents one directed edge. In each cell, there is either **A** or **R**. **A** means that the directed edge is accepted by our checker and **R** means that the cell is rejected by our checker. In each row, if all edges are accepted, then the example is accepted.

Row No.	test	anno-complete	doc2anno	doc2code	anno2doc	code2doc	3-edges
1	abs	A	A	A	A	A	A
2	all_digits	A	A	A	A	A	A
3	array_append	A	A	A	A	A	A
4	array_concat	A	A	A	A	A	A
5	array_copy	A	A	A	A	A	A
6	array_product	A	A	A	A	A	A
7	array_sum	A	A	A	A	A	A
8	avg	A	A	A	A	A	A
9	below_zero	A	A	A	A	A	A
10	binary_search	A	A	A	A	A	A
11	bubble_sort	A	A	A	A	A	A
12	cal_ans	A	A	A	A	A	A
13	cal_sum	A	A	A	A	A	A
14	canyon_search	A	A	R	A	A	R
15	compare	A	A	A	A	A	A
16	convert_map_key	R	R	R	A	A	R
17	copy_part	A	A	A	A	A	A
18	count_lessthan	R	R	R	A	A	R
19	double_quadruple	A	A	R	A	A	R
20	even_list	R	R	A	A	A	R
21	find	A	A	A	A	A	A
22	has_close_elements	A	A	A	A	A	A
23	insert	A	A	A	A	A	A
24	integer_square_root	A	A	A	A	A	A
25	is_even	A	A	A	A	A	A
26	is_palindrome	A	A	A	A	A	A
27	linear_search1	A	A	A	A	A	A
28	linear_search2	A	A	A	A	A	A
29	linear_search3	A	A	A	A	A	A
30	longest_prefix	A	A	A	A	A	A
31	max_array	A	A	A	A	A	A
32	min_array	A	A	A	A	A	A
33	min_of_two	A	A	A	A	A	A
34	modify_2d_array	A	R	A	A	R	R
35	multi_return	A	A	R	A	A	R
36	online_max	R	R	R	A	A	R
37	only_once	A	R	A	A	A	R
38	quotient	A	A	A	A	A	A
39	remove_front	A	A	A	A	A	A
40	replace	A	R	A	A	A	R
41	return_seven	A	A	A	A	A	A
42	reverse	A	A	A	A	A	A
43	rotate	A	A	A	A	A	A
44	selectionsort	A	A	A	A	A	A
45	seq_to_array	R	A	R	A	A	R
46	set_to_seq	R	A	R	A	A	R
47	slope_search	A	R	R	A	R	R
48	swap	A	A	R	A	A	R
49	swap_arith	A	A	A	A	A	A
50	swap_bitvector	A	A	A	A	A	A
51	swap_in_array	A	A	A	A	A	A
52	swap_sim	A	A	A	A	A	A
53	test_array	A	A	A	A	A	A
54	triple	A	A	A	A	A	A
55	triple2	A	A	A	A	A	A
56	triple3	A	A	A	A	A	A
57	triple4	A	A	A	A	A	A
58	two_sum	A	A	A	A	A	A
59	update_array	A	A	A	A	A	A
60	update_map	R	R	R	A	A	R

Table 12: CLOVER ground truth experiments when k=1

Row No.	test	max3tries	oneTry	noVerify_max3tries	withDoc_max3tries	max3tries_Claude
1	abs	A	A	A	A	A
2	all_digits	A	A	A	A	R
3	array_append	A	A	A	A	A
4	array_concat	A	A	A	A	A
5	array_copy	A	A	A	A	A
6	array_product	A	R	A	A	A
7	array_sum	A	R	A	A	A
8	avg	A	A	R	A	A
9	below_zero	A	R	R	A	A
10	binary_search	A	A	A	A	A
11	bubble_sort	A	A	A	A	A
12	cal_ans	A	R	A	A	A
13	cal_sum	A	A	A	A	A
14	canyon_search	A	R	R	A	R
15	compare	A	A	A	A	A
16	convert_map_key	R	R	R	R	R
17	copy_part	A	A	A	A	A
18	count_lessthan	R	R	R	R	R
19	double_quadruple	A	A	A	A	A
20	even_list	R	R	R	R	R
21	find	A	A	A	A	A
22	has_close_elements	A	R	R	A	A
23	insert	A	R	A	A	R
24	integer_square_root	A	R	A	A	A
25	is_even	A	A	A	A	A
26	is_palindrome	A	A	A	A	A
27	linear_search1	A	A	A	A	A
28	linear_search2	A	A	A	A	A
29	linear_search3	A	A	A	A	A
30	longest_prefix	A	A	A	A	A
31	max_array	A	A	A	A	A
32	min_array	A	A	A	A	A
33	min_of_two	A	A	A	A	A
34	modify_2d_array	A	A	A	A	A
35	multi_return	A	A	A	A	A
36	online_max	R	R	R	R	R
37	only_once	A	A	A	A	A
38	quotient	A	A	A	A	R
39	remove_front	A	A	A	A	A
40	replace	A	A	A	A	A
41	return_seven	A	A	A	A	A
42	reverse	A	R	A	A	A
43	rotate	A	A	A	A	R
44	selectionsort	A	A	A	A	A
45	seq_to_array	R	R	R	R	R
46	set_to_seq	R	R	R	R	R
47	slope_search	A	A	A	A	A
48	swap	A	A	A	A	A
49	swap_arith	A	R	A	A	A
50	swap_bitvector	A	R	A	A	A
51	swap_in_array	A	A	A	A	A
52	swap_sim	A	A	A	A	A
53	test_array	A	A	A	A	A
54	triple	A	A	A	A	A
55	triple2	A	A	A	A	A
56	triple3	A	A	A	A	A
57	triple4	A	A	A	A	A
58	two_sum	A	R	R	A	A
59	update_array	A	A	A	A	A
60	update_map	R	R	R	R	R

Table 13: Ablation studies that compare code generation under different configurations. Each column represents one configuration. We have: max3tries (a maximum of 3 tries with verifier feedback), oneTry (the first try), noVerify_max3tries (a maximum of 3 tries with only compiler and no verifier feedback), withDoc_max3tries (a maximum of 3 tries with verifier feedback plus docstrings), and max3tries_Claude (same as max3tries using Claude API).

Row No.	test	max3tries	oneTry
1	abs	A	A
2	all_digits	A	A
3	array_append	A	A
4	array_concat	A	A
5	array_copy	A	A
6	array_product	A	A
7	array_sum	A	R
8	avg	A	A
9	below_zero	R	R
10	binary_search	A	R
11	bubble_sort	R	R
12	cal_ans	A	A
13	cal_sum	A	R
14	canyon_search	A	R
15	compare	A	R
16	convert_map_key	R	R
17	copy_part	R	R
18	count_lessthan	R	R
19	double_quadruple	A	R
20	even_list	R	R
21	find	A	A
22	has_close_elements	A	A
23	insert	R	R
24	integer_square_root	A	R
25	is_even	A	A
26	is_palindrome	R	R
27	linear_search1	A	A
28	linear_search2	A	R
29	linear_search3	R	R
30	longest_prefix	R	R
31	max_array	A	R
32	min_array	A	A
33	min_of_two	A	A
34	modify_2d_array	R	R
35	multi_return	A	A
36	online_max	R	R
37	only_once	R	R
38	quotient	A	A
39	remove_front	A	R
40	replace	R	R
41	return_seven	A	A
42	reverse	R	R
43	rotate	A	R
44	selectionsort	R	R
45	seq_to_array	A	A
46	set_to_seq	R	R
47	slope_search	R	R
48	swap	A	A
49	swap_arith	A	R
50	swap_bitvector	A	A
51	swap_in_array	A	A
52	swap_sim	A	A
53	test_array	A	A
54	triple	A	A
55	triple2	A	A
56	triple3	A	A
57	triple4	A	A
58	two_sum	A	R
59	update_array	A	A
60	update_map	R	R

Table 14: Ablation studies for annotation generation from pure code: note that this is the only place where we count loop invariants in annotations. **A** means (1) generated annotations are equivalent to the original and (2) generated loop invariants are enough to have the code verified by Dafny.

Row No.	test	anno-sound	anno-complete	doc2anno	doc2code	anno2doc	code2doc
1	abs	A	A	R	R	R	R
2	all_digits	A	A	R	R	R	R
3	array_append	A	A	R	A	R	R
4	array_concat	A	A	R	A	A	R
5	array_copy	A	A	R	A	R	R
6	array_product	A	A	R	A	R	R
7	array_sum	A	A	A	A	A	R
8	avg	A	A	A	R	R	R
9	below_zero	A	A	R	R	R	R
10	binary_search	A	A	R	A	A	R
11	bubble_sort	A	A	R	A	R	R
12	cal_ans	A	A	R	R	R	R
13	cal_sum	A	A	R	R	R	R
14	canyon_search	A	A	R	R	R	R
15	compare	A	A	A	A	R	R
16	convert_map_key	A	R	R	R	A	R
17	copy_part	A	A	R	A	A	A
18	count_lessthan	A	R	R	R	R	R
19	double_quadruple	A	A	R	A	A	R
20	even_list	A	R	R	R	A	A
21	find	A	A	R	A	R	R
22	has_close_elements	A	A	R	R	R	R
23	insert	A	A	R	R	R	R
24	integer_square_root	A	A	R	A	R	R
25	is_even	A	A	R	R	R	R
26	is_palindrome	A	A	R	R	R	R
27	linear_search1	A	A	R	R	A	R
28	linear_search2	A	A	R	R	R	R
29	linear_search3	A	A	R	A	A	R
30	longest_prefix	A	A	R	A	R	R
31	max_array	A	A	R	R	R	R
32	min_array	A	A	R	R	R	R
33	min_of_two	A	A	R	R	R	R
34	modify_2d_array	A	A	R	R	R	R
35	multi_return	A	A	R	R	R	R
36	online_max	A	R	R	R	R	R
37	only_once	A	A	R	R	R	R
38	quotient	A	A	R	A	A	R
39	remove_front	A	A	R	R	R	R
40	replace	A	A	R	R	R	R
41	return_seven	A	A	R	R	R	R
42	reverse	A	A	R	R	R	R
43	rotate	A	A	R	A	A	R
44	selectionsort	A	A	R	A	A	R
45	seq_to_array	A	R	R	R	R	R
46	set_to_seq	A	R	R	R	R	R
47	slope_search	A	A	R	A	A	R
48	swap	A	A	R	R	A	R
49	swap_arith	A	A	R	R	R	R
50	swap_bitvector	A	A	R	A	A	R
51	swap_in_array	A	A	R	R	R	R
52	swap_sim	A	A	R	R	R	R
53	test_array	A	A	R	R	R	R
54	triple	A	A	R	R	R	R
55	triple2	A	A	R	A	A	R
56	triple3	A	A	R	R	R	R
57	triple4	A	A	R	R	R	R
58	two_sum	A	A	R	A	A	A
59	update_array	A	A	R	R	R	R
60	update_map	A	R	R	R	A	R

Table 15: C1 when k=1: all examples are rejected

Row No.	test	anno-sound	anno-complete	doc2anno	doc2code	anno2doc	code2doc
1	abs	A	A	R	R	R	R
2	all_digits	A	A	R	R	R	R
3	array_append	A	A	R	A	R	R
4	array_concat	A	A	R	A	A	R
5	array_copy	A	A	R	A	R	R
6	array_product	A	A	R	A	A	R
7	array_sum	A	A	A	A	A	R
8	avg	A	A	A	R	R	R
9	below_zero	A	A	R	R	R	R
10	binary_search	A	A	R	A	A	R
11	bubble_sort	A	A	R	A	A	R
12	cal_ans	A	A	R	R	R	R
13	cal_sum	A	A	R	R	R	R
14	canyon_search	A	A	R	R	R	R
15	compare	A	A	A	A	R	R
16	convert_map_key	A	R	R	R	A	A
17	copy_part	A	A	R	A	A	A
18	count_lessthan	A	R	R	R	R	R
19	double_quadruple	A	A	R	A	A	R
20	even_list	A	A	R	R	A	A
21	find	A	A	R	A	R	R
22	has_close_elements	A	A	R	R	A	R
23	insert	A	A	R	A	A	R
24	integer_square_root	A	A	R	A	A	R
25	is_even	A	A	A	A	R	R
26	is_palindrome	A	A	R	R	R	R
27	linear_search1	A	A	R	R	A	R
28	linear_search2	A	A	R	R	R	R
29	linear_search3	A	A	A	A	A	R
30	longest_prefix	A	A	R	A	A	R
31	max_array	A	A	R	R	R	R
32	min_array	A	A	R	R	A	R
33	min_of_two	A	A	R	R	R	R
34	modify_2d_array	A	A	R	R	R	R
35	multi_return	A	A	R	R	R	R
36	online_max	A	R	R	R	R	R
37	only_once	A	A	R	R	R	R
38	quotient	A	A	R	A	A	R
39	remove_front	A	A	R	R	R	R
40	replace	A	A	R	R	R	R
41	return_seven	A	A	R	R	R	R
42	reverse	A	A	R	R	R	R
43	rotate	A	A	R	A	A	A
44	selectionsort	A	A	R	A	A	A
45	seq_to_array	A	A	R	R	R	R
46	set_to_seq	A	A	R	R	R	R
47	slope_search	A	A	R	A	A	R
48	swap	A	A	R	R	A	R
49	swap_arith	A	A	R	R	R	R
50	swap_bitvector	A	A	A	A	A	R
51	swap_in_array	A	A	R	R	R	R
52	swap_sim	A	A	R	R	A	R
53	test_array	A	A	R	R	R	R
54	triple	A	A	R	R	R	R
55	triple2	A	A	R	A	A	R
56	triple3	A	A	R	R	R	R
57	triple4	A	A	R	R	R	R
58	two_sum	A	A	R	A	A	A
59	update_array	A	A	R	R	R	R
60	update_map	A	A	R	R	A	R

Table 16: C1 when k=10: all rejected

Row No.	test	anno-sound	anno-complete	doc2anno	doc2code	anno2doc	code2doc
1	abs	A	R	R	A	R	A
2	all_digits	A	R	R	A	A	A
3	array_append	A	A	R	A	R	A
4	array_concat	A	A	R	A	A	A
5	array_copy	A	A	R	A	R	A
6	array_product	A	A	R	A	A	A
7	array_sum	A	A	R	A	R	A
8	avg	A	A	R	A	R	A
9	below_zero	A	R	R	A	R	A
10	binary_search	A	A	R	A	A	A
11	bubble_sort	A	A	R	A	A	A
12	cal_ans	A	R	R	A	R	A
13	cal_sum	A	R	R	A	A	A
14	canyon_search	A	R	R	R	A	A
15	compare	A	A	R	A	A	A
16	convert_map_key	A	R	R	R	A	A
17	copy_part	A	R	R	A	A	A
18	count_lessthan	A	R	R	R	R	A
19	double_quadruple	A	A	R	R	R	A
20	even_list	A	R	R	A	A	A
21	find	A	A	R	A	R	A
22	has_close_elements	A	A	R	A	A	A
23	insert	A	A	R	A	R	A
24	integer_square_root	A	R	R	A	R	A
25	is_even	A	A	R	A	A	A
26	is_palindrome	A	A	R	A	A	A
27	linear_search1	A	A	R	A	A	A
28	linear_search2	A	R	R	A	A	A
29	linear_search3	A	A	R	A	A	A
30	longest_prefix	A	A	R	A	R	A
31	max_array	A	R	R	A	A	A
32	min_array	A	A	R	A	A	A
33	min_of_two	A	A	R	A	A	A
34	modify_2d_array	A	R	R	A	A	A
35	multi_return	A	A	R	R	A	A
36	online_max	A	R	R	R	R	A
37	only_once	A	A	R	A	A	A
38	quotient	A	A	R	A	A	A
39	remove_front	A	R	R	A	A	A
40	replace	A	A	R	A	A	A
41	return_seven	A	R	R	A	A	A
42	reverse	A	R	R	A	R	A
43	rotate	A	A	R	A	A	A
44	selectionsort	A	A	R	A	A	A
45	seq_to_array	A	R	R	R	A	A
46	set_to_seq	A	R	R	R	A	A
47	slope_search	A	R	R	R	A	A
48	swap	A	A	R	R	R	A
49	swap_arith	A	R	R	A	R	A
50	swap_bitvector	A	A	R	A	R	A
51	swap_in_array	A	A	R	A	A	A
52	swap_sim	A	A	R	A	R	A
53	test_array	A	R	R	A	R	A
54	triple	A	A	R	A	R	A
55	triple2	A	A	R	A	R	A
56	triple3	A	A	R	A	R	A
57	triple4	A	A	R	A	A	A
58	two_sum	A	R	R	A	A	A
59	update_array	A	R	R	A	R	A
60	update_map	A	R	R	R	A	A

Table 17: C2 when k=1: all examples are rejected

Row No.	test	anno-sound	anno-complete	doc2anno	doc2code	anno2doc	code2doc
1	abs	A	A	R	A	R	A
2	all_digits	A	R	R	A	A	A
3	array_append	A	A	R	A	R	A
4	array_concat	A	A	R	A	A	A
5	array_copy	A	A	R	A	R	A
6	array_product	A	A	R	A	A	A
7	array_sum	A	A	R	A	A	A
8	avg	A	A	R	A	R	A
9	below_zero	A	R	R	A	A	A
10	binary_search	A	A	R	A	A	A
11	bubble_sort	A	A	R	A	A	A
12	cal_ans	A	R	R	A	R	A
13	cal_sum	A	R	R	A	A	A
14	canyon_search	A	A	R	A	A	A
15	compare	A	A	R	A	A	A
16	convert_map_key	A	R	R	R	A	A
17	copy_part	A	A	R	A	A	A
18	count_lessthan	A	R	R	R	A	A
19	double_quadruple	A	A	R	A	R	A
20	even_list	A	A	R	A	A	A
21	find	A	A	R	A	A	A
22	has_close_elements	A	A	R	A	A	A
23	insert	A	A	R	A	A	A
24	integer_square_root	A	A	R	A	A	A
25	is_even	A	A	R	A	A	A
26	is_palindrome	A	A	R	A	A	A
27	linear_search1	A	A	R	A	A	A
28	linear_search2	A	A	R	A	A	A
29	linear_search3	A	A	R	A	A	A
30	longest_prefix	A	A	R	A	A	A
31	max_array	A	R	R	A	A	A
32	min_array	A	A	R	A	A	A
33	min_of_two	A	A	R	A	A	A
34	modify_2d_array	A	A	R	A	A	A
35	multi_return	A	A	R	A	A	A
36	online_max	A	R	R	R	R	A
37	only_once	A	A	R	A	A	A
38	quotient	A	A	R	A	A	A
39	remove_front	A	R	R	A	A	A
40	replace	A	A	R	A	A	A
41	return_seven	A	R	R	A	A	A
42	reverse	A	R	R	A	R	A
43	rotate	A	A	R	A	A	A
44	selectionsort	A	A	R	A	A	A
45	seq_to_array	A	R	R	A	A	A
46	set_to_seq	A	A	R	A	A	A
47	slope_search	A	A	R	A	A	A
48	swap	A	A	R	A	A	A
49	swap_arith	A	R	R	A	A	A
50	swap_bitvector	A	A	R	A	R	A
51	swap_in_array	A	A	R	A	A	A
52	swap_sim	A	A	R	A	R	A
53	test_array	A	R	R	A	R	A
54	triple	A	A	R	A	A	A
55	triple2	A	A	R	A	R	A
56	triple3	A	A	R	A	R	A
57	triple4	A	A	R	A	A	A
58	two_sum	A	A	R	A	A	A
59	update_array	A	R	R	A	A	A
60	update_map	A	R	R	R	A	A

Table 18: C2 when k=10: all examples are rejected

Row No.	test	anno-sound	anno-complete	doc2anno	doc2code	anno2doc	code2doc
1	abs	A	R	A	R	R	R
2	all_digits	A	R	R	R	A	A
3	array_append	A	A	R	A	R	R
4	array_concat	A	A	R	A	R	R
5	array_copy	A	A	A	A	R	R
6	array_product	A	A	R	R	A	R
7	array_sum	A	A	R	A	R	R
8	avg	A	R	R	A	R	R
9	below_zero	A	R	R	A	A	A
10	binary_search	A	A	R	A	A	R
11	bubble_sort	A	A	R	R	A	R
12	cal_ans	A	R	A	R	R	R
13	cal_sum	A	R	R	R	R	R
14	canyon_search	A	R	R	R	R	R
15	compare	A	A	R	A	A	R
16	convert_map_key	A	R	R	R	A	R
17	copy_part	A	A	R	A	A	R
18	count_lessthan	A	R	R	R	R	R
19	double_quadruple	A	R	A	R	R	R
20	even_list	A	A	R	A	A	A
21	find	A	A	R	A	A	R
22	has_close_elements	A	A	R	R	A	A
23	insert	A	R	R	R	A	R
24	integer_square_root	A	A	R	A	A	R
25	is_even	A	A	R	A	R	A
26	is_palindrome	A	A	R	A	A	A
27	linear_search1	A	A	R	A	A	R
28	linear_search2	A	A	R	A	A	R
29	linear_search3	A	A	R	A	A	R
30	longest_prefix	A	A	A	R	A	R
31	max_array	A	R	R	R	A	R
32	min_array	A	A	R	A	A	R
33	min_of_two	A	A	A	R	A	R
34	modify_2d_array	A	A	R	A	A	R
35	multi_return	A	A	A	R	R	R
36	online_max	A	R	R	R	R	R
37	only_once	A	A	R	A	R	A
38	quotient	A	A	R	A	A	R
39	remove_front	A	R	R	R	A	R
40	replace	A	A	R	A	A	A
41	return_seven	A	R	A	A	A	R
42	reverse	A	R	A	R	R	R
43	rotate	A	A	R	A	A	R
44	selectionsort	A	A	R	R	A	R
45	seq_to_array	A	R	R	R	A	R
46	set_to_seq	A	R	R	R	A	R
47	slope_search	A	A	R	A	A	R
48	swap	A	R	R	R	R	R
49	swap_arith	A	R	A	R	R	R
50	swap_bitvector	A	A	A	R	R	R
51	swap_in_array	A	A	R	A	A	R
52	swap_sim	A	A	A	A	R	R
53	test_array	A	R	R	R	R	R
54	triple	A	A	A	R	A	R
55	triple2	A	R	A	A	A	R
56	triple3	A	R	A	R	R	R
57	triple4	A	R	R	R	R	R
58	two_sum	A	A	R	A	A	R
59	update_array	A	R	A	R	R	R
60	update_map	A	R	R	R	A	A

Table 19: C3 when k=1: all examples are rejected

Row No.	test	anno-sound	anno-complete	doc2anno	doc2code	anno2doc	code2doc
1	abs	A	A	A	R	A	R
2	all_digits	A	R	R	R	A	A
3	array_append	A	A	R	A	A	R
4	array_concat	A	A	R	A	R	A
5	array_copy	A	A	A	A	A	R
6	array_product	A	A	R	A	A	R
7	array_sum	A	A	A	A	A	R
8	avg	A	A	A	A	A	R
9	below_zero	A	R	A	A	A	A
10	binary_search	A	A	R	A	A	R
11	bubble_sort	A	A	R	A	A	A
12	cal_ans	A	R	A	A	A	A
13	cal_sum	A	R	A	R	A	R
14	canyon_search	A	A	R	A	A	A
15	compare	A	A	R	A	A	R
16	convert_map_key	A	R	R	R	A	R
17	copy_part	A	A	R	A	A	R
18	count_lessthan	A	R	A	R	A	R
19	double_quadruple	A	R	A	R	A	R
20	even_list	A	A	R	A	A	A
21	find	A	A	R	A	A	R
22	has_close_elements	A	A	R	A	A	A
23	insert	A	R	A	R	A	R
24	integer_square_root	A	A	R	A	A	A
25	is_even	A	A	R	A	R	A
26	is_palindrome	A	A	R	A	A	A
27	linear_search1	A	A	A	A	A	R
28	linear_search2	A	A	A	A	A	R
29	linear_search3	A	A	R	A	A	R
30	longest_prefix	A	A	A	A	A	R
31	max_array	A	R	A	R	A	R
32	min_array	A	A	A	A	A	R
33	min_of_two	A	A	A	A	A	R
34	modify_2d_array	A	A	R	A	A	A
35	multi_return	A	A	A	A	A	R
36	online_max	A	R	R	A	A	R
37	only_once	A	A	R	A	A	A
38	quotient	A	A	R	A	A	A
39	remove_front	A	R	R	R	A	R
40	replace	A	A	R	A	A	A
41	return_seven	A	R	A	A	A	R
42	reverse	A	R	A	R	A	R
43	rotate	A	A	A	A	A	R
44	selectionsort	A	A	A	A	A	R
45	seq_to_array	A	R	R	R	A	R
46	set_to_seq	A	A	A	A	A	R
47	slope_search	A	A	R	A	A	R
48	swap	A	R	R	R	A	R
49	swap_arith	A	R	A	R	A	R
50	swap_bitvector	A	A	A	A	A	R
51	swap_in_array	A	A	R	A	A	R
52	swap_sim	A	A	A	A	A	R
53	test_array	A	R	R	R	R	R
54	triple	A	A	A	A	A	R
55	triple2	A	A	A	A	A	R
56	triple3	A	R	A	R	A	R
57	triple4	A	R	R	R	R	R
58	two_sum	A	A	R	A	A	R
59	update_array	A	R	A	R	A	R
60	update_map	A	R	A	R	A	A

Table 20: C3 when k=10: all examples are rejected

Row No.	test	anno-sound	anno-complete	doc2anno	doc2code	anno2doc	code2doc
1	abs	A	A	R	R	R	R
2	all_digits	A	R	R	R	A	A
3	array_append	A	R	R	R	A	A
4	array_concat	A	A	R	R	R	R
5	array_copy	A	R	R	R	R	R
6	array_product	A	A	R	R	R	R
7	array_sum	A	R	R	R	R	R
8	avg	A	A	R	R	R	R
9	below_zero	A	R	R	R	A	R
10	binary_search	A	R	R	R	A	R
11	bubble_sort	A	A	R	R	R	R
12	cal_ans	A	R	R	R	R	A
13	cal_sum	A	A	R	R	R	R
14	canyon_search	A	A	R	R	R	A
15	compare	A	A	R	R	R	R
16	convert_map_key	A	R	R	R	R	R
17	copy_part	A	R	R	R	A	R
18	count_lessthan	A	R	R	R	R	A
19	double_quadruple	A	R	R	R	R	R
20	even_list	A	R	R	R	A	R
21	find	A	A	R	R	A	R
22	has_close_elements	A	A	R	R	A	A
23	insert	A	A	R	R	R	R
24	integer_square_root	A	A	R	R	A	R
25	is_even	A	R	R	A	A	A
26	is_palindrome	A	R	R	R	A	A
27	linear_search1	A	A	R	R	R	R
28	linear_search2	A	R	R	R	A	R
29	linear_search3	A	A	R	R	R	R
30	longest_prefix	A	R	R	R	R	R
31	max_array	A	R	R	R	R	R
32	min_array	A	A	R	A	A	A
33	min_of_two	A	A	R	R	R	A
34	modify_2d_array	A	R	R	R	R	R
35	multi_return	A	A	R	R	R	R
36	online_max	A	R	R	R	R	R
37	only_once	A	A	R	R	R	A
38	quotient	A	R	R	R	R	R
39	remove_front	A	R	R	R	R	A
40	replace	A	R	R	R	A	R
41	return_seven	A	A	R	R	R	R
42	reverse	A	A	R	R	A	A
43	rotate	A	R	R	R	R	R
44	selectionsort	A	A	R	R	A	R
45	seq_to_array	A	R	R	R	R	A
46	set_to_seq	A	R	R	R	R	A
47	slope_search	A	A	R	R	R	R
48	swap	A	A	R	R	R	A
49	swap_arith	A	R	R	R	R	R
50	swap_bitvector	A	A	R	R	R	R
51	swap_in_array	A	R	R	R	A	R
52	swap_sim	A	A	R	R	R	R
53	test_array	A	R	R	R	R	R
54	triple	A	A	R	R	R	R
55	triple2	A	R	R	R	R	R
56	triple3	A	A	R	R	R	R
57	triple4	A	R	R	R	R	R
58	two_sum	A	R	R	R	A	A
59	update_array	A	R	R	R	R	R
60	update_map	A	R	R	R	R	R

Table 21: C6 when k=1: all examples are rejected

Row No.	test	anno-sound	anno-complete	doc2anno	doc2code	anno2doc	code2doc
1	abs	A	A	R	R	R	R
2	all_digits	A	R	R	R	A	A
3	array_append	A	R	R	R	A	A
4	array_concat	A	A	R	R	R	R
5	array_copy	A	R	R	R	R	R
6	array_product	A	A	R	R	R	R
7	array_sum	A	R	R	R	A	R
8	avg	A	A	R	R	R	R
9	below_zero	A	R	R	R	A	R
10	binary_search	A	R	R	R	A	R
11	bubble_sort	A	A	R	R	R	R
12	cal_ans	A	R	R	R	R	A
13	cal_sum	A	A	R	R	R	R
14	canyon_search	A	A	R	A	A	A
15	compare	A	A	R	R	R	R
16	convert_map_key	A	R	R	R	A	R
17	copy_part	A	R	R	R	A	R
18	count_lessthan	A	R	R	R	R	A
19	double_quadruple	A	A	R	R	R	R
20	even_list	A	R	R	R	A	A
21	find	A	A	R	A	A	R
22	has_close_elements	A	A	R	R	A	A
23	insert	A	A	R	R	A	R
24	integer_square_root	A	A	R	R	A	R
25	is_even	A	R	R	A	A	A
26	is_palindrome	A	A	R	R	A	A
27	linear_search1	A	A	R	R	R	R
28	linear_search2	A	R	R	R	A	R
29	linear_search3	A	A	R	R	A	R
30	longest_prefix	A	R	R	R	A	R
31	max_array	A	R	R	R	R	R
32	min_array	A	A	R	A	A	A
33	min_of_two	A	A	R	R	R	A
34	modify_2d_array	A	R	R	R	R	R
35	multi_return	A	A	R	R	R	R
36	online_max	A	A	R	R	A	R
37	only_once	A	A	R	R	R	A
38	quotient	A	R	R	R	A	R
39	remove_front	A	R	R	R	A	A
40	replace	A	R	R	R	A	R
41	return_seven	A	A	R	R	R	R
42	reverse	A	A	R	R	A	A
43	rotate	A	R	R	R	A	R
44	selectionsort	A	A	R	R	A	A
45	seq_to_array	A	R	R	R	A	A
46	set_to_seq	A	R	R	R	A	A
47	slope_search	A	A	R	R	A	R
48	swap	A	A	R	R	R	A
49	swap_arith	A	A	R	A	R	R
50	swap_bitvector	A	A	R	R	R	R
51	swap_in_array	A	R	R	R	A	A
52	swap_sim	A	A	R	R	R	R
53	test_array	A	A	R	R	R	R
54	triple	A	A	R	R	R	R
55	triple2	A	R	R	R	R	R
56	triple3	A	A	R	R	R	R
57	triple4	A	R	R	R	R	R
58	two_sum	A	R	R	R	A	A
59	update_array	A	A	R	R	A	R
60	update_map	A	R	R	R	A	R

Table 22: C6 when k=10: all examples are rejected

Row No.	test	doc2code	code2doc	doc2anno	anno2doc	anno-sound	anno-complete
1	task_id_101	A	A	A	A	A	A
2	task_id_267	A	A	A	A	A	A
3	task_id_404	A	A	A	A	A	A
4	task_id_770	A	A	R	A	A	A
5	task_id_599	A	A	A	A	A	A
6	task_id_610	A	A	A	A	A	A
7	task_id_605	A	A	A	A	A	A
8	task_id_600	A	A	A	A	A	A
9	task_id_760	A	A	A	A	A	A
10	task_id_616	A	A	A	A	A	A
11	task_id_62	A	A	A	A	A	A
12	task_id_77	A	A	A	A	A	A
13	task_id_431	A	A	A	A	A	A
14	task_id_433	A	A	A	A	A	A
15	task_id_625	A	A	A	A	A	A
16	task_id_741	A	A	A	A	A	A
17	task_id_227	A	A	A	A	A	A
18	task_id_435	A	A	A	A	A	A
19	task_id_447	A	A	A	A	A	A
20	task_id_441	A	A	A	A	A	A
21	task_id_454	A	A	A	A	A	A
22	task_id_127	A	A	A	A	A	A
23	task_id_474	A	A	A	A	A	A
24	task_id_627	A	A	R	A	A	A
25	task_id_58	A	A	A	A	A	A
26	task_id_733	A	A	R	A	A	A
27	task_id_743	A	A	A	A	A	A

Table 23: MBPP-DFY-50 ground truth results when k=10