Clover: Closed-Loop Verifiable Code Generation

Chuyue (Livia) Sun*, **Ying Sheng***,

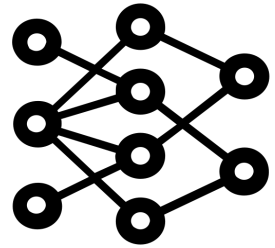
Oded Padon, Clark Barrett

Outline

- Motivation
- Background
- New Dataset (CloverBench)
- New Framework (Clover)
  - Reconstruction Test
  - Equivalence Checking
- Experiments
  - Clover Generation Phase
  - Clover Verification Phase
    - Ground Truth
    - Wrong Variants
- Limitations
- Conclusion

Motivation: Untrustworthy Code Generation

- Large Language Model (LLM) can generate code (Tabachnyk & Nikolov, 2022).



- How can we trust the generated code?

- Formal Verification!

Background: Formal Verification

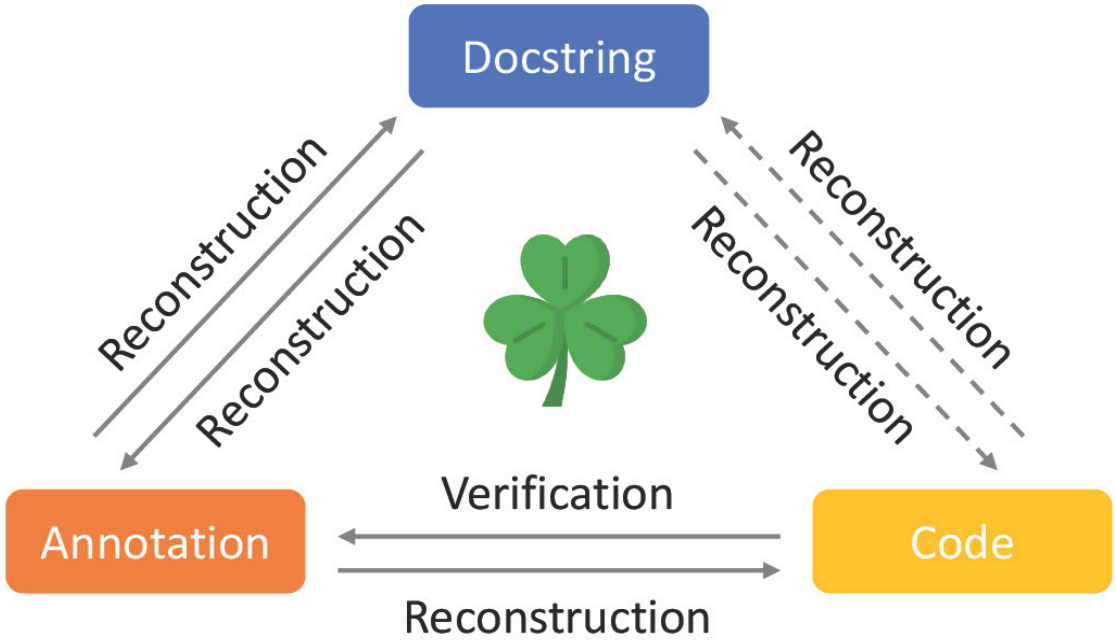
1. Construct a mathematical model of the system
2. Provide a formal specification of the system
3. Prove that the model satisfies the specification

Motivation: Key Insights

- Any AI-based code generation technique should aim to include not only code, but also **formal specifications** and **natural language docstrings**.
- We can use formal tools coupled with generative AI techniques to ensure that they are consistent.

Clover: Closed-Loop Verifiable Code Generation

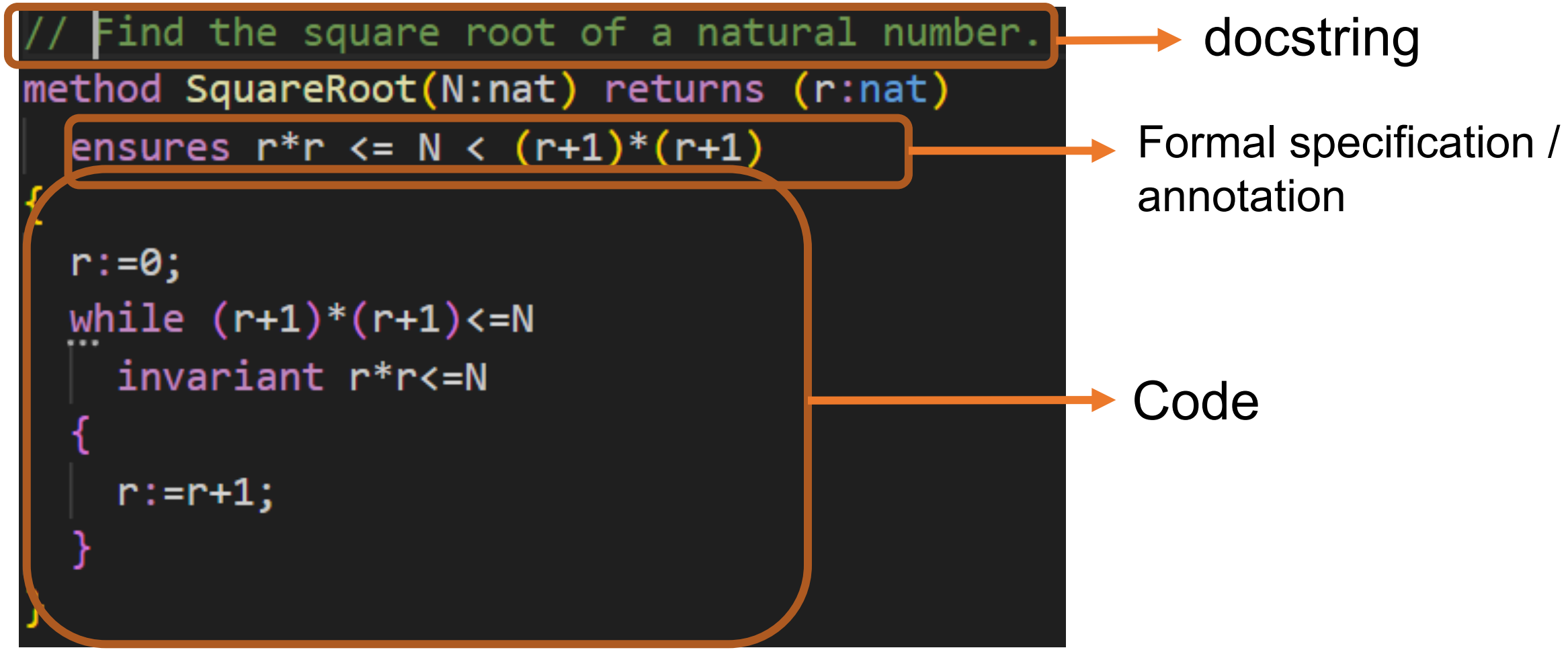
- Phase One: Generation
- Phase Two: Verification



Our Contribution

- Clover proposes a solution for trustworthy code generation
  Checking correctness → Checking consistency
- **CloverBench** dataset features 60 manually annotated Danfy programs with docstrings
- Feasibility demonstration of using GPT-4 to generate annotations
- Evaluation of the Clover framework using GPT-4 and D4fny

Background: Deductive Program Verification

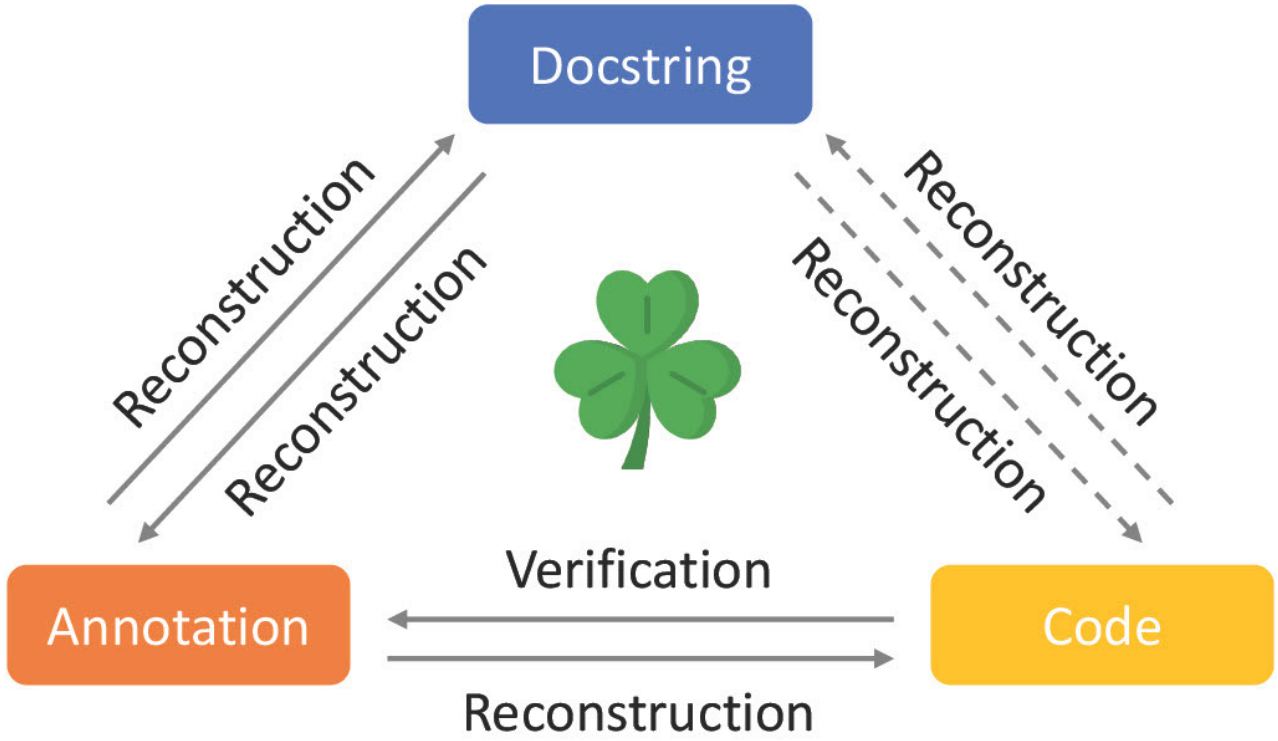


Dataset: CloverBench

- Ground Truth
  - Unit Tests
  - Annotation Template
- Wrong variants
  - C1
  - C2
  - C3

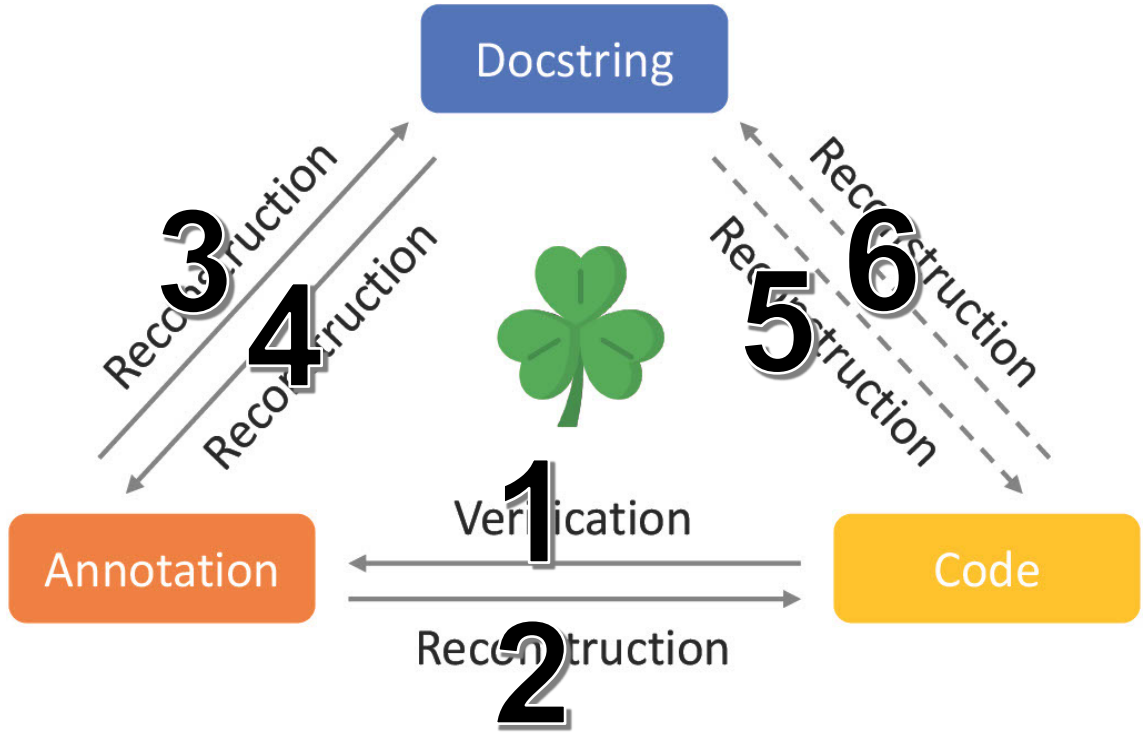
```
// Find the square root of a natural number.
method SquareRoot(N:nat) returns (r:nat)
  ensures r*r <= N < (r+1)*(r+1)
{
  r:=0;
  while (r+1)*(r+1)<=N
    invariant r*r<=N
  {
    r:=r+1;
  }
}
```

Clover Verification Phase

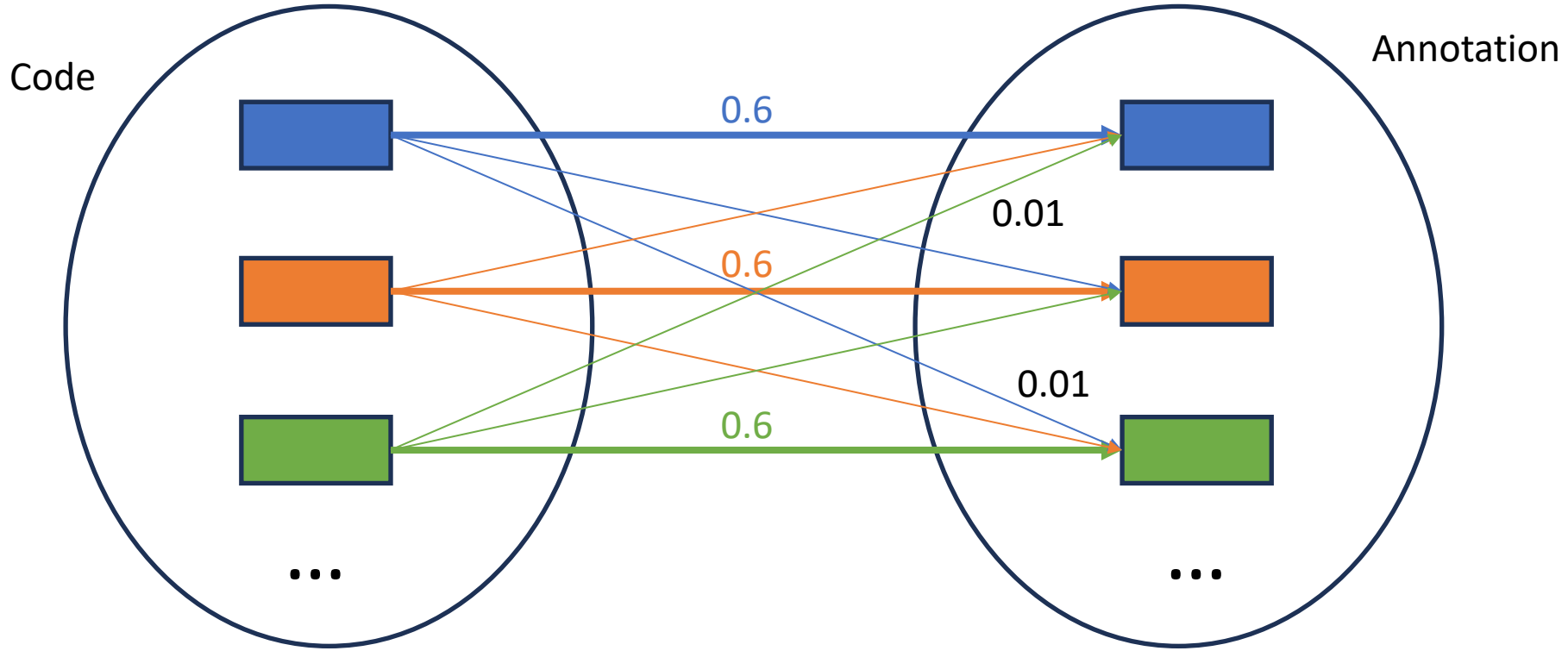


Six Consistency Checks

1. **anno-sound**: Verifier
2. **anno-complete**: LLM + Compiler
3. **anno2doc**: LLM
4. **doc2anno**: LLM + Verifier
5. **doc2code**: LLM + Compiler
6. **code2doc**: LLM



Reconstruction Testing



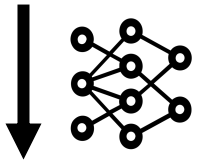
- Correct mapping has significant probability to be reconstructed.
- Incorrect mapping has negligible probability to be reconstructed.

Equivalence checker

- Docstring equivalence: semantic checker
- Code equivalence: unit tests
- Annotation equivalence: formal tools

Docstring equivalence

```
1. Find the square root of a natural number.

2. The dafny annotation defines a method called 'foo' which takes a natural
number 'N' as an argument and returns another natural number 'r'.
The 'ensures' clause specifies the postcondition that the square of the returned
number 'r' should be less than or equal to 'N', but 'N' should be less than the
square of the number 'r' incremented by 1. This essentially defines 'r' as the
integer square root of 'N'. This function will incrementally increase 'r' until
'r*r' is greater than 'N', at which point it will return the 'r' that makes 'r*r'
less than 'N' but '(r+1)*(r+1)' greater than 'N'.
```



```
Yes, both docstrings describe the same functional behavior.
They both describe a function that finds the square root of a natural number.
In the second docstring, this is described by the conditions on 'r' in the postcondition.
However, the second docstring provides more specific detail about the program's functionality,
such as the method's name and input argument.
```

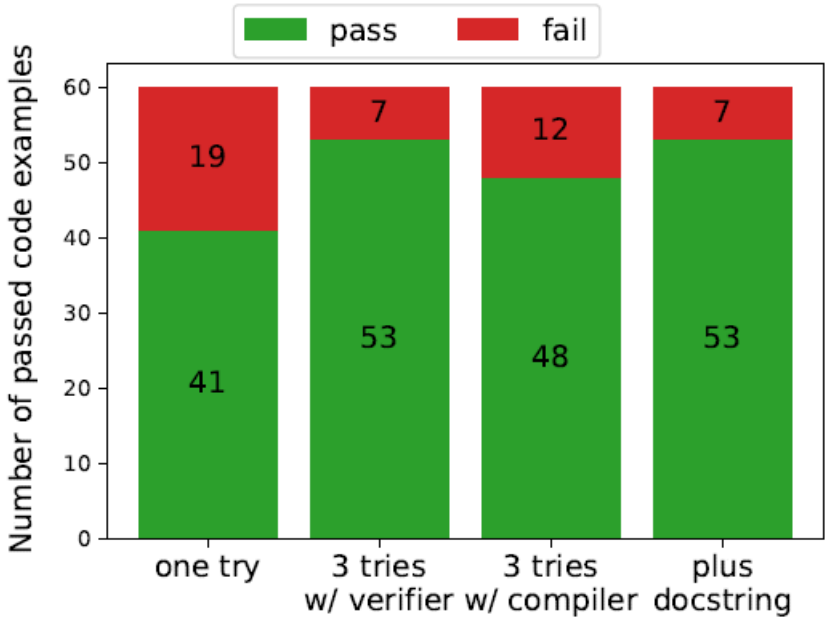
Code Equivalence

```
method TestSquareRoot(){
  var test1 := SquareRoot(4);
  print("Test 1: SquareRoot(4) = ", test1, "\n");

  var test2 := SquareRoot(10);
  print("Test 2: SquareRoot(10) = ", test2, "\n");

  var test3 := SquareRoot(0);
  print("Test 3: SquareRoot(0) = ", test3, "\n");

  var test4 := SquareRoot(15);
  print("Test 4: SquareRoot(15) = ", test4, "\n");

  var test5 := SquareRoot(25);
  print("Test 5: SquareRoot(25) = ", test5, "\n");
}
```

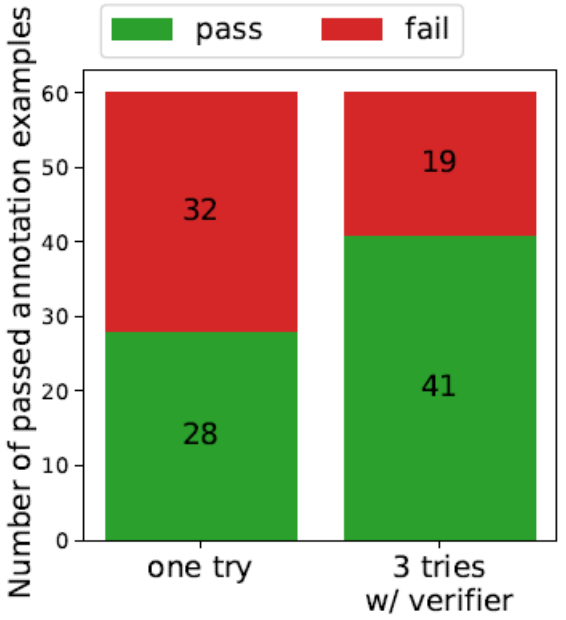
Annotation Equivalence

```
predicate post_original(N:nat,r:nat)
  requires pre_original(N,r){
  ( r*r <= N < (r+1)*(r+1))
}

predicate post_gen(N:nat,r:nat)
  requires pre_original(N,r){
  true // (#POST) && ... (#POST)
}

lemma post_eq(N:nat,r:nat)
  requires pre_original(N,r )
  requires pre_gen(N,r )
  ensures post_original(N,r ) <==> post_gen(N,r )
{
}
```

Experiments: Clover Generation Phase



(a) Code generation.



(b) Annotation generation

Experiments: Clover Verification Phase

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

Table 1: Summary of the experiments results on the verification phase.

- 87% acceptance for ground-truth
- 100% rejection for wrong

Experiments: Incorrect Variants

- C1: Annotation wrong, docstring and code same as ground-truth.
- C2: Docstring wrong, annotation and code same as ground-truth.
- C3: Docstring same as ground-truth, annotation and code mutated in such a way that the Dafny verification check succeeds.

Experiments: Incorrect Variants

Category	C1 Reject		C2 Reject		C3 Reject	
	k=1	k=10	k=1	k=10	k=1	k=10
anno-sound	32/60 (53%)	32/60 (53%)	0/60 (0%)	0/60 (0%)	0/60 (0%)	0/60 (0%)
anno-complete	25/60 (42%)	16/60 (27%)	8/60 (13%)	8/60 (13%)	47/60 (78%)	20/60 (33%)
doc2anno	60/60 (100%)	60/60 (100%)	60/60 (100%)	60/60 (100%)	60/60 (100%)	60/60 (100%)
anno2doc	31/60 (52%)	18/60 (30%)	44/60 (73%)	44/60 (73%)	42/60 (70%)	42/60 (70%)
code2doc	2/60 (3%)	0/60 (0%)	49/60 (82%)	47/60 (78%)	39/60 (65%)	34/60 (57%)
doc2code	11/60 (18%)	7/60 (12%)	32/60 (53%)	32/60 (53%)	52/60 (87%)	52/60 (87%)

Table 5: Rejection rates for incorrect examples.

Limitations

- Capabilities of LLM (GPT-4)
- Annotations only specify functionalities
- Internally consistent but misalign with human intention

Conclusion

- Clover Framework: Closed-Loop Verifiable Code Generation
  - Reduces the problem of checking correctness to consistency
- CloverBench: 60 textbook exaples
  - 87% acceptance for ground truth
  - 100% rejection for incorrect examples
- Future work
  - Better verification tools
  - Improve LLM generation
  - Better equivalence checking