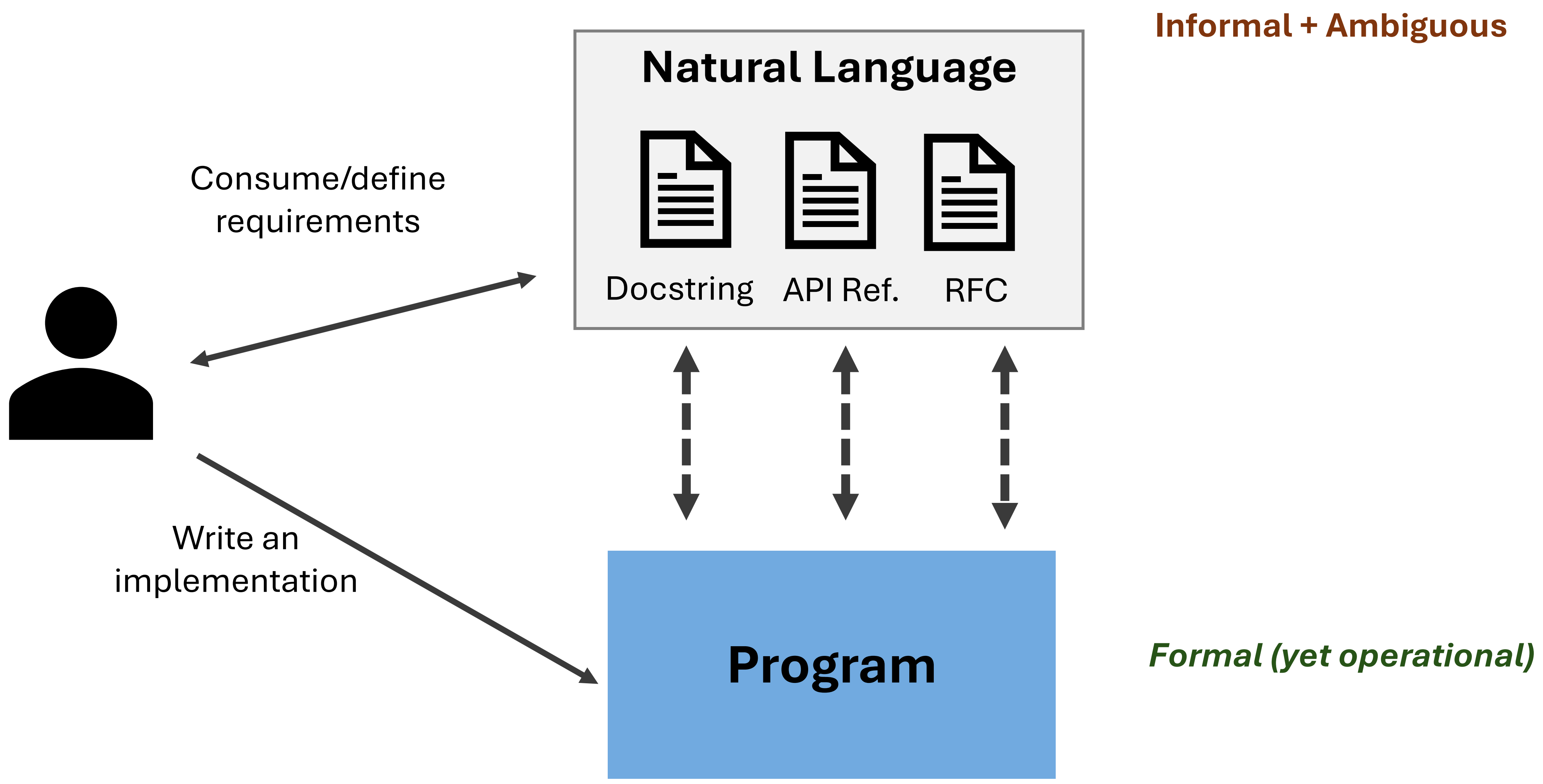
Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions?

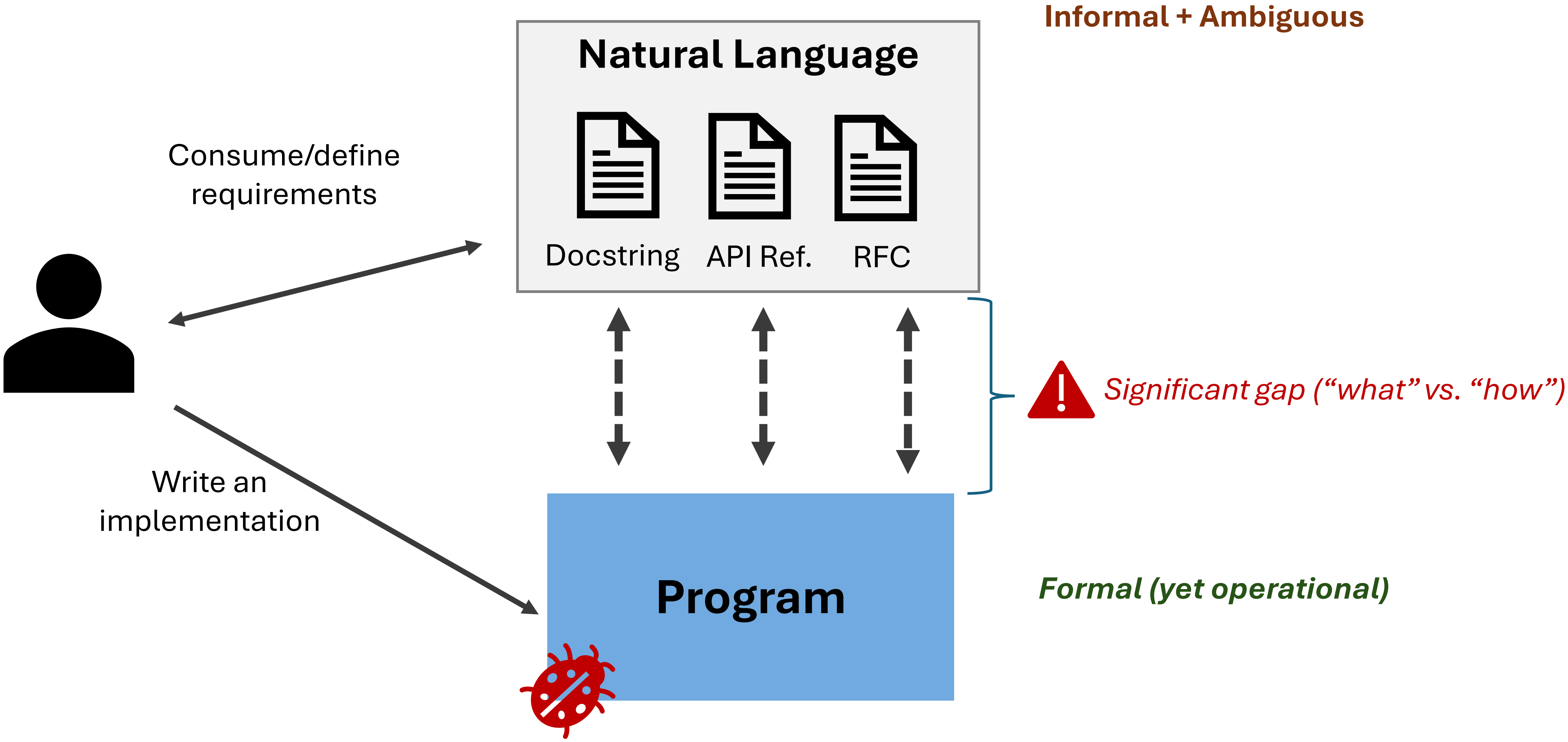
Madeline Endres (University of Michigan)

Sarah Fakhoury, Saikat Chakraborty, **Shuvendu Lahiri** (Microsoft Research)

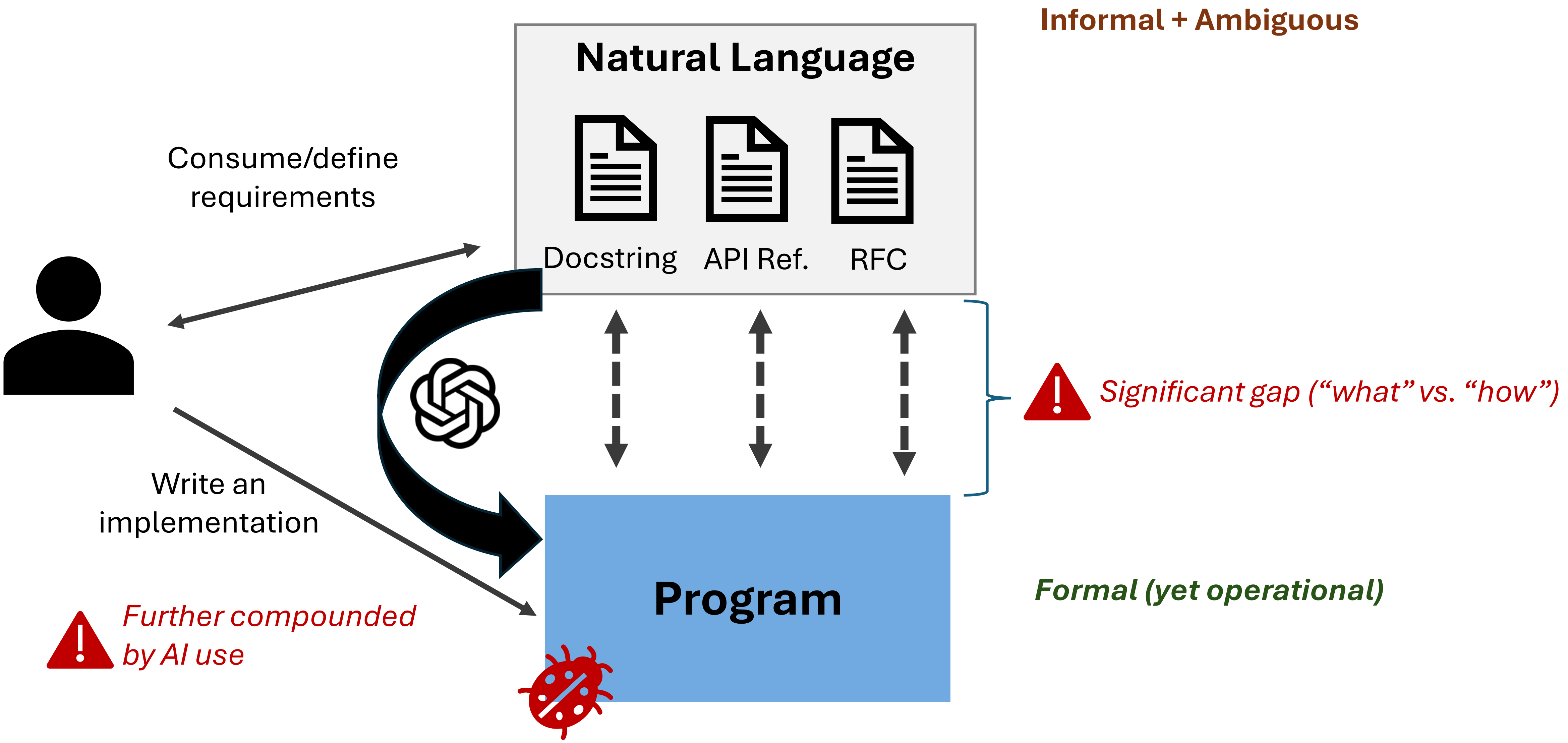
Software requirements are often specified informally



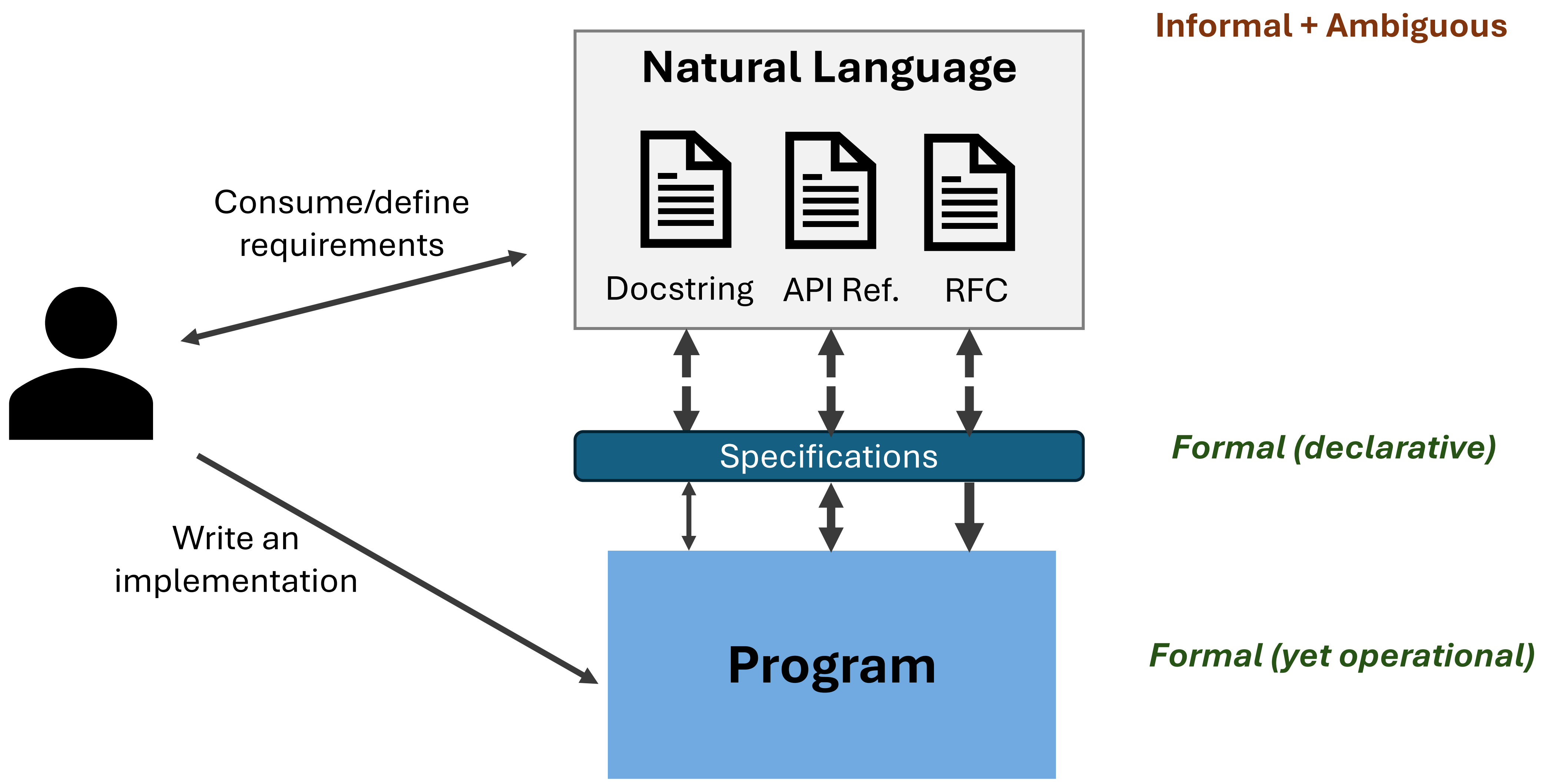
Software requirements are often specified informally



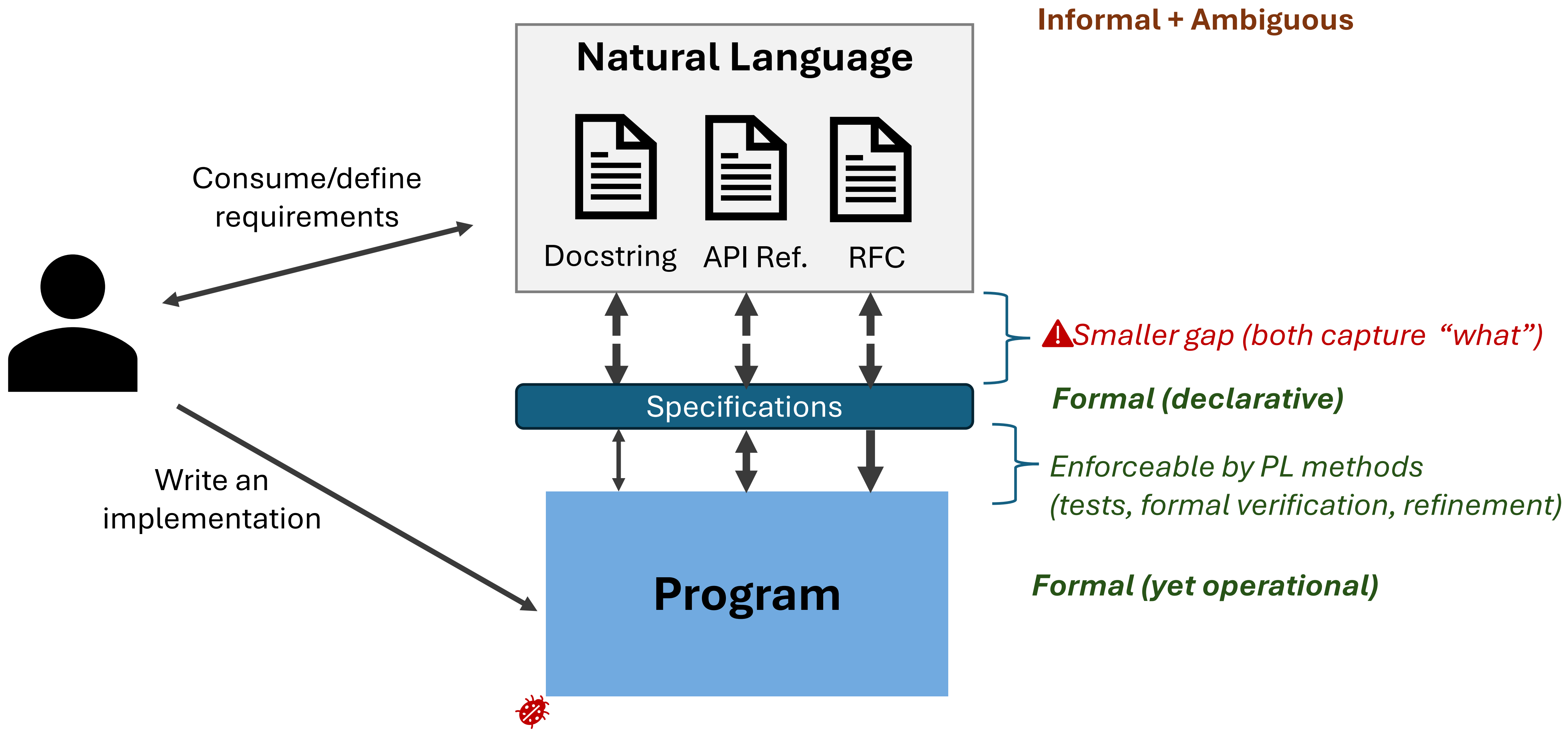
Software requirements are often specified informally



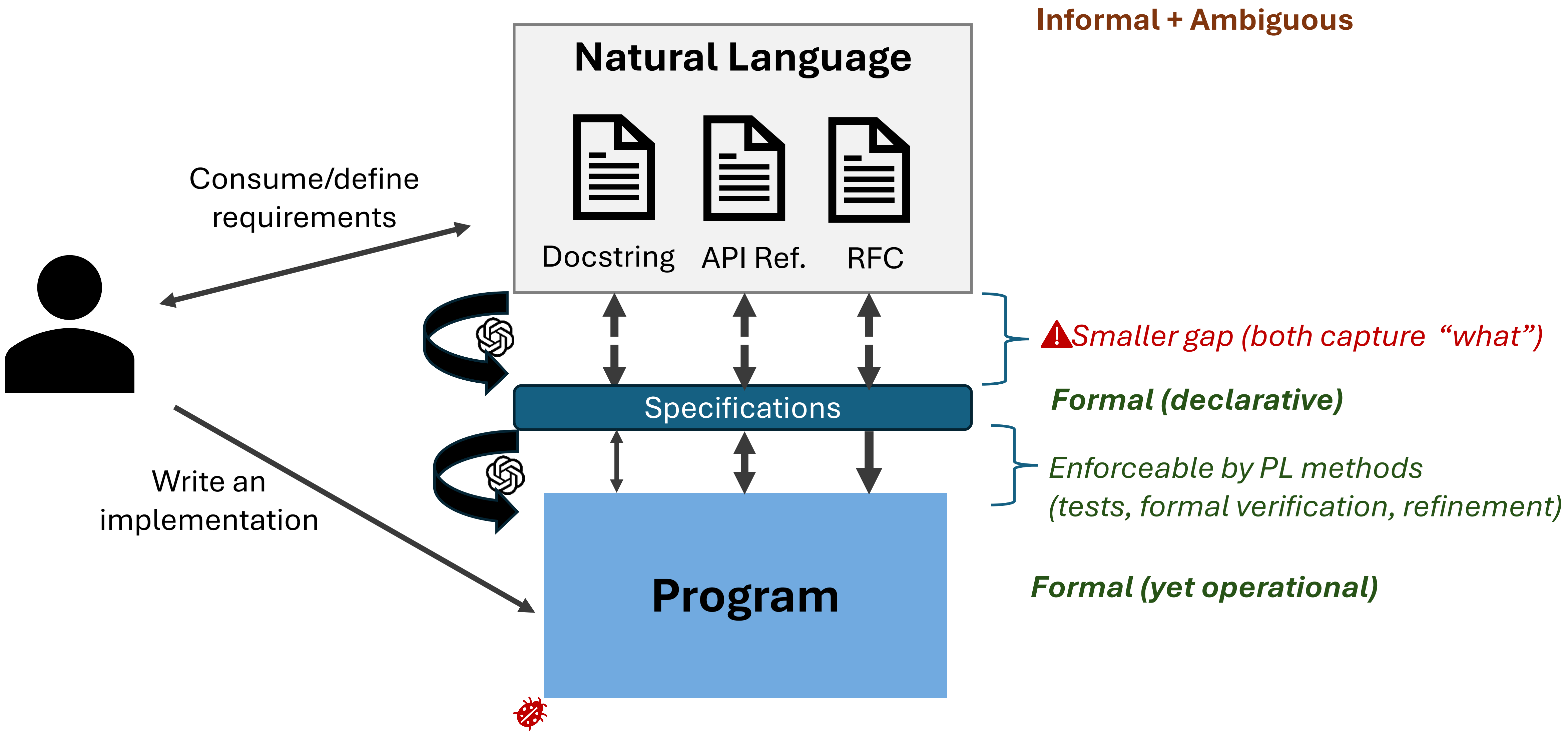
Proposal: Formal specifications can reduce the gap



Proposal: Formal specifications can reduce the gap

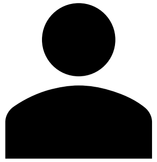


Proposal: Formal specifications can reduce the gap



Motivating example

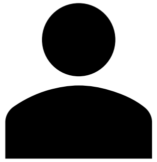
[1,2,3,2,4] -> [1,3,4]



```python
def remove_duplicates(numbers: List[int]):
    """ From a list of integers, remove all elements that occur more than once,
        Keep order of elements left the same as in the input."""
```

Motivating example

[1,2,3,2,4] -> [1,3,4]



```python
def remove_duplicates(numbers: List[int]):
    """ From a list of integers, remove all elements that occur more than once,
        Keep order of elements left the same as in the input."""
```

Formal Specifications in Python

```python
assert len(set(numbers)) == len(set(return_list))
```

Motivating example

[1,2,3,2,4] -> [1,3,4]

```python
def remove_duplicates(numbers: List[int]):
    """ From a list of integers, remove all elements that occur more than once,
        Keep order of elements left the same as in the input."""
```

Formal Specifications in Python

```python
assert len(set(numbers)) == len(set(return_list))
```

Motivating example

[1,2,3,2,4] -> [1,3,4]

```python
def remove_duplicates(numbers: List[int]):
    """ From a list of integers, remove all elements that occur more than once,
        Keep order of elements left the same as in the input."""
```

Formal Specifications in Python

✗
```python
assert len(set(numbers)) == len(set(return_list))
```

✓
```python
assert all(numbers.count(i) == 1 for i in return_list)
```

Motivating example

[1,2,3,2,4] -> [1,3,4]

```python
def remove_duplicates(numbers: List[int]):
    """ From a list of integers, remove all elements that occur more than once,
        Keep order of elements left the same as in the input."""
```

Formal Specifications in Python

```python
assert len(set(numbers)) == len(set(return_list))
```

```python
assert all(numbers.count(i) == 1 for i in return_list)
```

```python
assert all(i in return_list for i in numbers if numbers.count(i) == 1)
```

Problem formulation

- Given
  - NL description nl for a method m
- Generate a postcondition S of m from nl

Research Questions:

1. Benchmark and metrics
   1. How do we characterize if a specification S captures the intent in nl?
   2. How good are LLMs at user-intent-formalization?
2. What are good real-world application of user-intent-formalization?

Contributions

1. **Semantics-based metrics** for evaluating user-intent-formalization (similar to code generation)

2. **Empirical evaluation of LLMs** for the task of user-intent-formalization

3. Application: **Finding historical real-world bugs**

Problem formulation (ideal)

- Given
  - NL description nl for a method m
  - (hidden) reference implementation I
- Generate a postcondition S of m from nl
- Evaluation metrics (intuition)
  - **Soundness**: I satisfies S
  - **Completeness**: S discriminates I from any buggy implementations

Problem formulation (based on tests)

- Given
  - NL description nl for a method m
  - (hidden) reference implementation I + a set of input/output tests T
- Generate a postcondition S of m from nl
- Evaluation metrics (intuition)
  - **Test-set Soundness**: S is consistent with I for each test t in T
  - **Test-set Completeness**: S discriminates I from any buggy implementations on some test t in T
  - Score = $\begin{cases} 0 & \text{if unsound} \\ \text{|buggy mutants discriminated|/|mutants|} & \end{cases}$

Buggy mutant generation

Leverage LLMs!

1. Prompt GPT-3.5 to enumerate 200 solutions to nl prompt
2. Group mutants by the subset of tests in T they pass [**natural bugs**]
3. If too few distinct mutants,
    1. Prompt GPT-3.5 to enumerate 200 “buggy” solutions to nl prompt
    2. Group mutants by the subset of tests in T they pass [**artificial bugs**]

Buggy mutant generation

Leverage LLMs!

1. Prompt GPT-3.5 to enumerate 200 solutions to nl prompt
2. Group mutants by the subset of tests in T they pass [**natural bugs**]
3. If too few distinct mutants,
    1. Prompt GPT-3.5 to enumerate 200 “buggy” solutions to nl prompt
    2. Group mutants by the subset of tests in T they pass [**artificial bugs**]

Hypothesis

- More space of mutations (compared to traditional mutant generation through mutating program elements)
- More natural and subtly incorrect mutants?

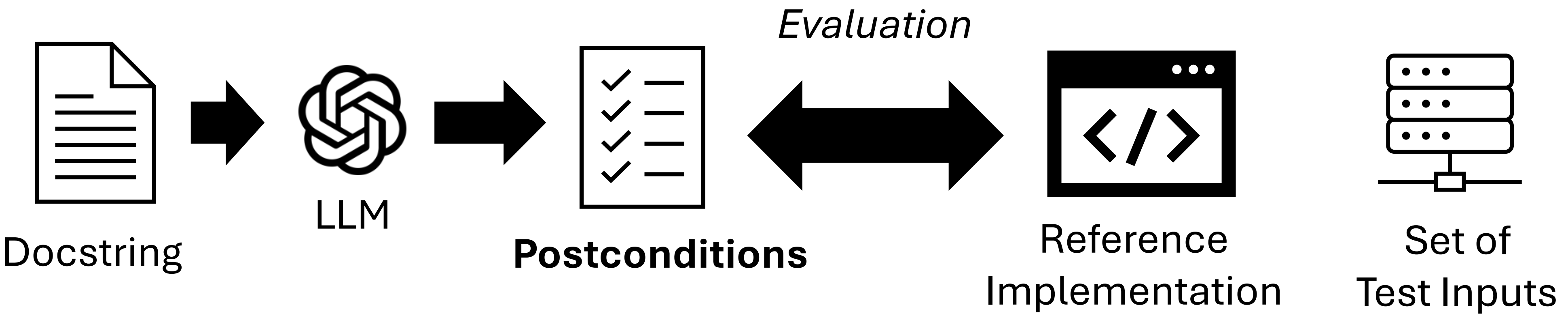
RQ1: How good are LLMs at generating specs from Natural Language?

Evaluation Methodology: **EvalPlus**
[Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. Liu et al. NeurIPS'23]

For each problem in HumanEvall, we used LLMs to generate a set of postconditions. We consider the following ablations[1]:

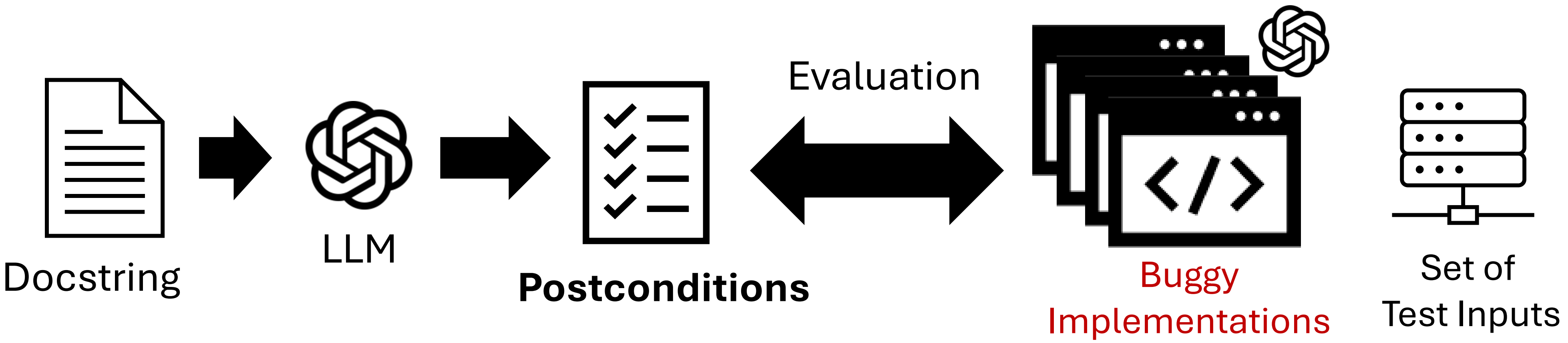
1. Model (GPT 3.5 and GPT 4 and StarCoder)
2. Prompting with NL only vs. NL + reference solution

RQ1: Postcondition *Soundness*



Model	Prompt	Prompt has: NL Only=✗ ref code=✓	Accept @ 1	Accept @ 5	Accept @ 10	x/164 correct
GPT-3.5	base	✗	0.46	0.80	0.87	143
GPT-3.5	base	✓	0.49	0.81	**0.88**	**145**
GPT-3.5	simple	✗	0.55	**0.82**	0.87	143
GPT-3.5	simple	✓	**0.56**	**0.82**	**0.88**	144
GPT-4	base	✗	0.63	0.83	0.88	144
GPT-4	base	✓	0.71	0.89	0.91	150
GPT-4	simple	✗	**0.77**	**0.94**	**0.96**	**158**
GPT-4	simple	✓	0.76	0.92	**0.96**	157
StarChat	base	✗	0.21	0.61	0.82	134
StarChat	base	✓	0.20	0.59	0.77	126
StarChat	simple	✗	**0.25**	**0.69**	0.85	139
StarChat	simple	✓	0.23	0.67	**0.86**	**141**

RQ1: Postcondition *Completeness*



Model	Prompt	Prompt has: NL Only=✗ ref code=✓	% bug-complete	% problems with bug-complete	% problems union bug-complete	Avg bug-completeness-score for correct postconditions *Natural bugs*	*All bugs*
GPT-3.5	base	✗	15.4	42.1	48.2	0.62	0.72
GPT-3.5	base	✓	**18.5**	**47.0**	**49.4**	**0.70**	**0.76**
GPT-3.5	simple	✗	8.1	29.3	33.5	0.44	0.55
GPT-3.5	simple	✓	14.0	37.2	41.5	0.58	0.62
GPT-4	base	✗	**35.1**	**61.6**	**62.2**	**0.81**	**0.85**
GPT-4	base	✓	34.9	58.0	61.6	0.78	0.82
GPT-4	simple	✗	9.2	26.2	29.3	0.40	0.52
GPT-4	simple	✓	8.9	29.3	36.0	0.47	0.56
StarChat	base	✗	0.8	7.3	8.5	0.13	0.24
StarChat	base	✓	1.4	9.1	11.0	**0.23**	0.30
StarChat	simple	✗	1.5	6.7	7.3	0.16	0.24
StarChat	simple	✓	**3.0**	**17.1**	**17.7**	0.23	**0.36**

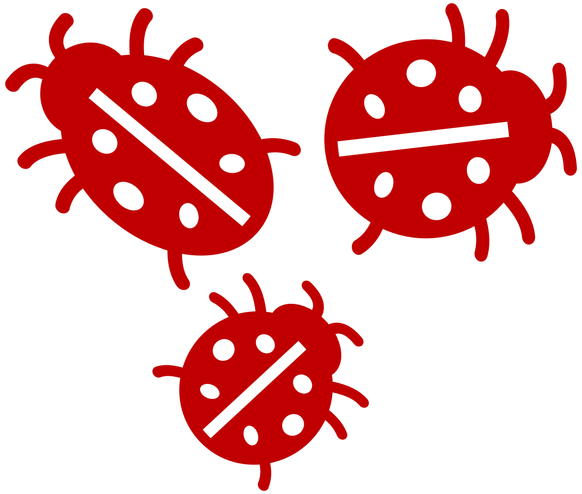
GPT-4 subsbtantially better at complete specs

Common postcondition categories on HumanEval

Category	Example Postconditon	% Prevalent	Avg. Bug-complete-score (*Natural*/*All*)
Type Check	`isinstance(return_val, int)`	47.4	*0.14 / 0.27*
Format Check	`return_val.startswith("ab")`	11.2	0.43 / 0.57
Arithmetic Bounds	`return_val >= 0`	30.8	0.23 / 0.34
Arithmetic Equality	`return_val[0] == 2 * input_val`	17.5	**0.82 / 0.89**
Container Property	`len(return_val) > len(input_val)`	27.0	0.45 / 0.57
Element Property	`return_val[0] % 2 == 0`	12.6	0.39 / 0.53
Forall-Element Property	`all(ch.isalpha() for ch in return_val)`	8.3	0.23 / 0.44
Implication	`(return_val==False) if 'A'not in string`	12.7	0.58 / 0.64
Null Check	`return_val is not None`	4.4	0.40 / 0.50
Average			0.32 / 0.46

RQ2: Can GPT-4 generated specifications find real-world bugs?

Evaluate on **Defects4J** dataset of real-world bugs and fixes in mature Java projects



Our postconditions leverage functional Java syntax introduced in Java 8. Not all bugs in Defects4J are Java 8 syntax compatible.

Our NL2Spec Defects4J subset contains 525 bugs from 11 projects. These bugs implicate 840 buggy Java methods.

[Defects4J: a database of existing faults to enable controlled testing studies for Java programs. 2014. Rene Just, Darioush Jalali, Michael Ernst]

RQ2: *Bug Finding: Experiments*

We use GPT-4 to generate 10 postconditions and 10 preconditions for each **buggy** function.

We consider two ablations (33,600 total GPT-4 calls)

- NL + Buggy Method Code + Relevant File Context
- NL + Relevant File Context

For each, we measure:

Correctness	Bug-discriminating
Does the **spec pass** the tests on correct code?	If it is correct, does the **spec fail** any of the tests on buggy code?

Defects4J results

Model	Prompt has: NL Only = ✗ buggy code = ✓	Compiles @1	Compiles @5	Compiles @10	Test-set correct @1	Test-set correct @5	Test-set correct @10	# distinguishable bugs
GPT-4	✗	0.65	0.86	0.89	0.32	0.57	0.66	35
GPT-4	✓	0.73	0.90	**0.93**	0.39	0.66	**0.75**	**47**
StarChat	✗	0.25	0.68	0.83	0.11	0.38	0.55	19
StarChat	✓	0.29	0.72	**0.84**	0.12	0.39	**0.56**	**24**

Across ablations, **65 bugs (12.5% of all bugs) are plausibly caught** by generated specifications

- We manually verify a subset of bug catching conditions

Defects4J results

Model	Prompt has: NL Only = ✗ buggy code = ✓	Compiles @1	Compiles @5	Compiles @10	Test-set correct @1	Test-set correct @5	Test-set correct @10	# distinguishable bugs
GPT-4	✗	0.65	0.86	0.89	0.32	0.57	0.66	35
GPT-4	✓	0.73	0.90	**0.93**	0.39	0.66	**0.75**	**47**
StarChat	✗	0.25	0.68	0.83	0.11	0.38	0.55	19
StarChat	✓	0.29	0.72	**0.84**	0.12	0.39	**0.56**	**24**

Across ablations, **65 bugs (12.5% of all bugs) are plausibly caught** by generated specifications

- We manually verify a subset of bug catching conditions

Complementary to prior assertion generation approaches TOGA [Dinella, Ryan, Mytkowicz, Lahiri, ICSE'22] and Daikon [Ernst et al. ICSE'99]

- TOGA mostly finds expected exceptional bugs. TOGA can only tolerate bugs during testing, and cannot prevent bugs in production.
- Daikon specs overfit the regression tests and bug-discriminating specs are unsound

RQ2: Example triggered bug from *Defects4J*

```java
/**
 * <p>Render the specified text and return the rendered Options
 * in a StringBuffer.</p>
 *
 * @param sb The StringBuffer to place the rendered text into.
 * @param width The number of characters to display per line
 * @param nextLineTabStop The position on the next line for the first tab.
 * @param text The text to be rendered.
 * @return the StringBuffer with the rendered Options contents.
 */

protected StringBuffer renderWrappedText(StringBuffer sb, int width,int nextLineTabStop, String text)
{
    int pos = findWrapPos(text, width, 0);
    if (pos == -1)
    {
        sb.append(rtrim(text));
        return sb;
    }

    sb.append(rtrim(text.substring(0, pos))).append(defaultNewLine);

    final String padding = createPadding(nextLineTabStop);
    final String padding = createPadding(0);

    while (true)
    {
        text = padding + text.substring(pos).trim();
```

```
Width = 17

1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-16-17-18-19-20-21

Formatted Text Example
    This text is
    formatted correctly


Formatted Text Example
    This text is
        formatted incorrectly
```

```java
// All lines must be less than or equal to the specified width
assert returnValue.toString().lines().allMatch(line -> line.length() <= width);
```

https://issues.apache.org/jira/browse/CLI-151

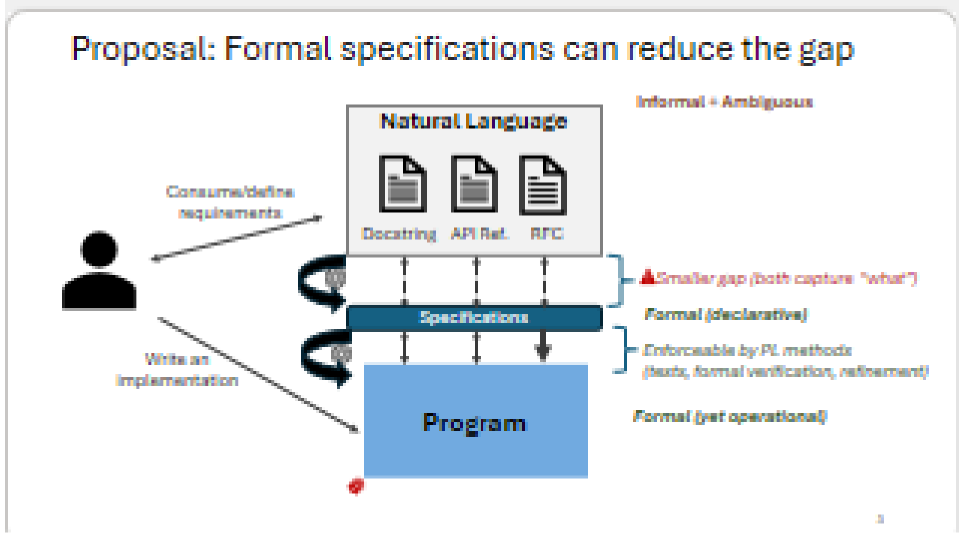
Ongoing works around user-intent-formalization

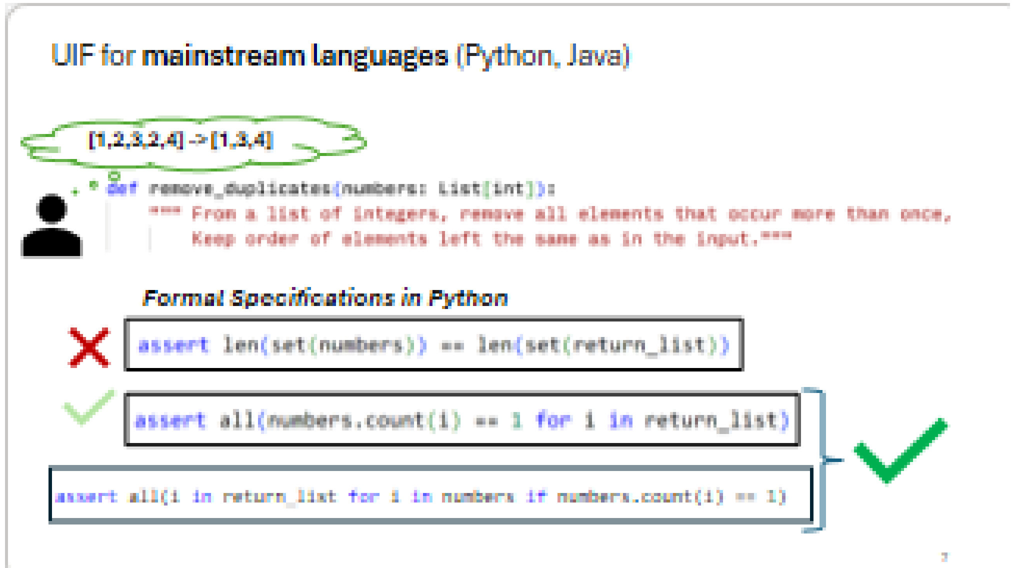
Evaluating user-intent-formalization for **verification-aware languages (**Verus, Dafny, F*) [Lahiri FMCAD'24]

TiCoder: Improving code-generation via user-intent-formalization with tests [LLM-based Test-driven Interactive Code Generation: User Study and Empirical Evaluation, Fakhoury, Naik, Sakkas, Chakraborty, Lahiri, TSE'24]

Real-world application on generating verified parsers through user-intent-formalization of RFC documents

Questions





Paper page

Trusted AI-assisted Programming project