User-intent formalization problem for programs

Shuvendu Lahiri
Research in Software Engineering (RiSE),
Microsoft Research

Contributors: Sarah Fakhoury, Madeline Endres (intern), Saikat Chakraborty, Nikhil Swamy, Tahina Ramananandro, Markus Kuppe, Jubi Taneja, Madan Musuvathi, ....
Software requirements are often specified informally

Further compounded by AI use

Significant gap ("what" vs. "how")
Vision: Formal specifications can reduce the gap

Consumption/definition of requirements

Write an implementation

Will create more trust in AI generated code

User-intent formalization (UIF) for programs

• Problem
  • Evaluate the quality of (LLM-generated) formal specification given informal artefacts
    • Natural language + code (interpreted neurally)
    • Like “autoformalization” problem for math theorems, but crucial differences
      • [Autoformalization with Large Language Models, Wu et al. NeurIPS’22]
    • Input can optionally contain formal artefacts such as tests/specs etc.

• Challenge: Not a pure Programming Languages (PL) problem

• Solution:
  • Adopt the approach of machine learning (ML) folks of establishing benchmarks
    (examples and {automated, objective} metrics)
  • Make the metrics PL based (like nl2code generation)
User-intent formalism (UIF) for different programming languages

• UIF for **mainstream languages** (Python, Java), and use case
  • Endres, Fakhoury, Chakraborty, Lahiri *FSE’24*

• UIF for **verification-aware languages** (Dafny, F*, Verus, ...)
  • Lahiri (in preparation)

• UIF for **effectively analyzable symbolic languages** (EASL)
  • 3DGen: Fakhoury, Kuppe, Lahiri, Ramananandro, Swamy
  • vectorizeGPT: Taneja, Yan, Lahiri (in preparation)
UIF for **mainstream languages** (Python, Java)

Can Large Language Models Transform Natural Language Intent into Formal Method Postconditions? Endres, Fakhoury, Chakraborty, Lahiri *FSE’24*
UIF for **mainstream languages** (Python, Java)

```
[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.""
    return_list = [i for i in numbers if numbers.count(i) == 1]
    assert all(i in return_list for i in numbers if numbers.count(i) == 1)
```

**Formal Specifications in Python**

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

**Wrong**

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

**Right**

```
assert all(numbers.count(i) == 1 for i in return_list)
```
Problem formulation and evaluation metrics

• Given
  - NL description \( nl \) for a method \( m \)
  - Hidden tests \( T \) and hidden reference implementation \( I \)

• Generate a postcondition \( S \) of \( m \) from \( nl \)

• Evaluation metrics
  - **Test-set Soundness**: \( S \) passes on \( I \) for all the tests in \( T \)
  - **Bug Completeness**: \( S \) discriminates against buggy implementations \( \{I'\} \)
    - Inspired by mutation-testing
    - **Insight**: Generate list of buggy \( I' \) by sampling LLM responses given \( nl \) and evaluating using \( T \)
RQ1: Evaluation on basic Python programs

• **Dataset:** EvalPlus (HumanEval + extensive test suite)

• **Models:** GPT-3.5, GPT-4 and open source StarCoder

• **Takeaways:**
  - GPT-4 significantly better at producing test-set sound (~96% in 10 tries) and complete (~62% in 10 tries) specifications
  - Much more pronounced for completeness since `assert True` and `assert isinstance(return_list, list)` are sound but not discriminatory
  - The metrics correlate strongly with the result of manual labeling (by authors) of the generated specifications in most cases
  - Challenge: need to rank specifications in increasing order of completeness for practical usage
RQ2: Can GPT-4 generated specifications find real-world bugs?

- Experimental setup with Defects4J [Just, Jalali, Ernst. ISSTA 2014]
  - Given two versions of a code: B (buggy), F (fixed)
  - Prompted LLMs to generate a postcondition S given B
  - Bug-discriminating postcondition
    - Check if S fails B and succeeds F for some test t in provided test suite T

- GPT-4 found 47 bug-discriminating postcondition of the 525 bugs analyzed
  - 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
UIF for verification-aware languages
(Dafny, F*, Verus, ...)

11
Challenge

• Earlier approached do not readily apply
  • Specifications contain ghost variables and complex quantifiers (cannot be evaluated using dynamic methods)
  • Trying to verify specification against reference implementation would likely not be automated (intermediate lemmas and invariants)

• Our approach: symbolically test specifications (given tests as input/output examples)
  • Given
    • A method signature
      ```
      method Foo(x): (returns y) requires P(x) ensures Q(x, y)
      ```
    • A set of input/output tests \( T \)

  • **Specification Soundness** (for a test \((i, o)\)) //Boolean metric
    • \{P\} x := i; y := o; \{Q\} is valid

  • **Specification Completeness** (for a test \((i, o)\)) //Quantitative metric
    • Fraction of mutants \( o' \) of \( o \), s.t. \{P\} x := i; y := o'; \{Q\} is not valid
Evaluation (preliminary)

• Dataset: ~200 Dafny specifications for MBPP-Dafny dataset
  • [Towards AI-Assisted Synthesis of Verified Dafny Methods. Misu, Lopes, Ma, Noble. FSE’24]
  • 50 hand-written, 153 GPT-4 generated and manually labeled \{incorrect, weak, strong\}

• Problem: Evaluate the soundness/completeness metrics

• Result
  • Translated the metrics into Dafny verification problems
    • Requires auxiliary assertions for arrays/sequences to help quantifier instantiation
  • Automated metrics gets parity with the human-labeling
  • Finds instances where a “strong” specification is not complete
"Write a function to find the shared elements from the given two lists."

**Predicate**

```plaintext
predicate InArray(a: array<int>, x: int)
  reads a
  {exists i :: 0 <= i < a.Length && a[i] == x}
```

**Method**

```plaintext
method SharedElements(a: array<int>, b: array<int>) returns (result: seq<int>)
  ensures forall x :: x in result ==> (InArray(a, x) && InArray(b, x))
  ensures forall i, j :: 0 <= i < j < |result| ==> result[i] != result[j]
```

```plaintext
method SharedElementsTest(){
  var a1:= new int[] [3, 4, 5, 6];
  var a2:= new int[] [5, 7, 4, 10];
  var res1:=SharedElements(a1,a2);
    //expected[4, 5];
}
```

Our metric marks this as a weak specification (wrt the test)

Changing `==>` to `<==>` makes it a strong invariant by our metric.
Vision: Formal specifications can reduce the gap

Problem formulation and evaluation metrics

- Given
  - NL description $nl$ for a method $m$
  - Hidden tests $T$ and hidden reference implementation $l$
- Generate a postcondition $S$ of $m$ from $nl$
- Evaluation metrics
  - Test-set Soundness: $S$ passes on $l$ for all the tests in $T$
  - Bug Completeness: $S$ discriminates against buggy implementations ($l'$)
    - Inspired by mutation-testing
    - Insight: Generate list of buggy $l'$ by sampling LLM responses given $nl$ and evaluating using $T$

User-intent formalism (UIF) for different programming languages

- UIF for mainstream languages (Python, Java), and use case
  - Endres, Fakhoury, Chakraborty, Lahiri FSE'24
- UIF for verification-aware languages (Dafny, F*, Verus, ...)
  - Lahiri (in preparation)
- UIF for effectively analyzable symbolic languages (EASL)
  - 3DGen: Fakhoury, Kuppe, Lahiri, Ramananandro, Swamy
  - vectorizeGPT: Taneja, Yan, Lahiri (in preparation)