

CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models

[ICSE' 23] Caroline Lemieux etc.

Lingyu Zhang

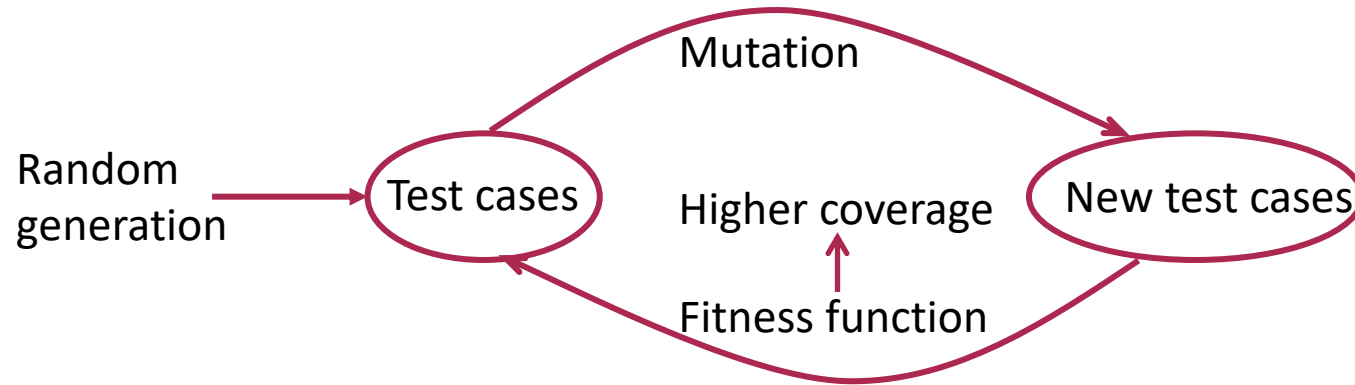
2023/05/30

Outline

- Search-Based Software Testing (SBST)
- SBST Suffers Coverage Plateaus
- Unleashing the Power of PLLM to Escape the Coverage Plateaus
 - Prompt Engineering
 - Post-processing
- Evaluation

Search-Based Software Testing (SBST)

- Automatically generate test cases with some form of evolutionary algorithm





SBST Suffers Coverage Plateaus

- Difficult to exercise PUT (Program Under Test) in an expected manner

```
<...omitted code...>
def _build_version_bump_position(pos: int) -> int:
    pos_min = -3
    pos_max = 2
    if (pos_min <= pos <= pos_max) is False:
        raise ValueError("Invalid position")
    # Turn position into a positive number
    if pos < 0:
        pos_max += 1
        return pos_max + pos
    return pos
<...omitted code...>
def bump_version(version: str, pos: int = 2,
                 pre_release: Optional[str]= None) -> str:
    ver_info = _build_version_info(version)
    pos = _build_version_bump_position(pos)
    bump_type = _build_version_bump_type(pos, pre_release)
    <...omitted code...>
    return out
```

Module under Test



Search stuck on generated test cases

Expected version string: " 1.2.2 "

Generated test case

```
def test_case_1():
    str_0 = 'a\t!sUo~AU'
    str_1 = bump_version(
        str_0)
```

Expected pos range: -3~2

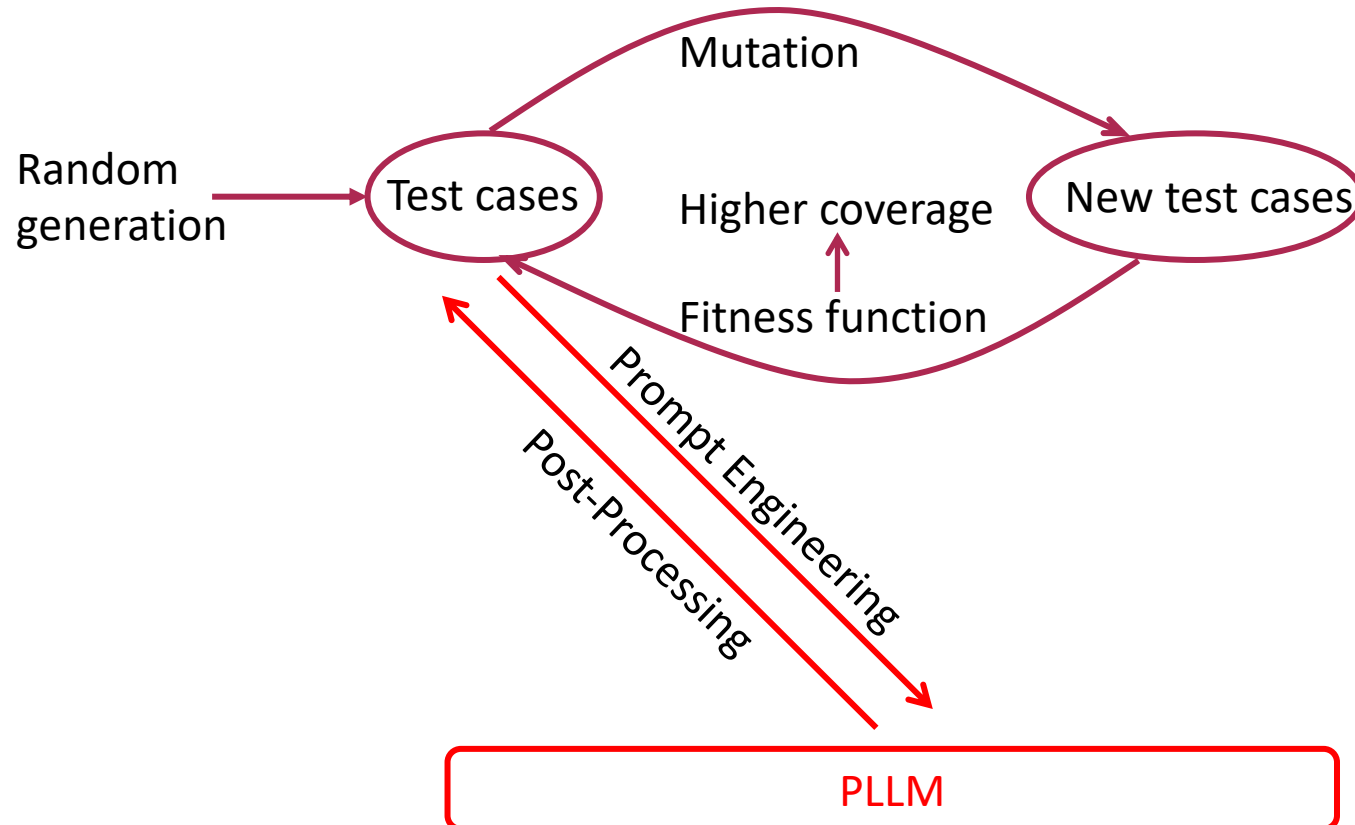
Generated test case

```
def test_case_2():
    str_0 = None
    int_0 = 1431
    str_1 = bump_version(
        str_0, int_0)
```

Hard to
mutate

Unleashing the Power of PLLM

- Query PLLM to generate tests for low coverage callables



Prompt Engineering

- Sampling low coverage callable
 - $cov(c^*)$: the coverage score of callable c^*

$$\text{Probability of sampling } c^* = \frac{1 - cov(c^*)}{\sum 1 - cov(c)}$$

- Zero-shot learning

Source Code including targeted callable X

```
# Unit test for function/method/constructor X
def test_X():
```

Incompatible Test Case Generated by PLLM

- Requiring test deserialization

<...omitted code...>

Module under Test

```
def _build_version_bump_position(pos: int) -> int:
    pos_min = -3
    pos_max = 2
    if (pos_min <= pos <= pos_max) is False:
        raise ValueError("Invalid position")
    # Turn position into a positive number
    if pos < 0:
        pos_max += 1
        return pos_max + pos
    return pos

<...omitted code...>

def bump_version(version: str, pos: int = 2,
                 pre_release: Optional[str]= None) -> str:
    ver_info = _build_version_info(version)
    pos = _build_version_bump_position(pos)
    bump_type = _build_version_bump_type(pos, pre_release)
    <...omitted code...>
    return out
```

```
def test_bump_version():
    assert bump_version('0.0.0') == '1.0.0'
    assert bump_version('0.0.0', 1) == '0.1.0'
```

deserialize to mutable test case

```
def test_case():
    str_0 = '0.0.0'
    str_1 = bump_version(str_0)
    int_0 = 1
    str_2 = bump_version(str_0, int_0)
```

Post-processing (Test Deserialization)

1. Rewrite PLLM Generations

Pynguin's Assumptions

Test cases are sequence of assignment statements

`v1 = 3` Constant
`v2 = bar(v1)` Function call
`v3 = [a, b]` List or dictionary

Single variable

- 1) Store value of standalone expression into a variable
- 2) Remove nested subexpressions

`z = foo(bar(2))`



`int_0 = 2`
`var_0 = bar(int_0)`
`z = foo(var_0)`

Post-processing (Test Deserialization)

2. Partial Parsing

```
x = 3
y = UNKNOWN_FUNCTION(x)
z = foo(y)
w = bar(x)
```

Pyguin → Drop All

```
x = 3
y = UNKNOWN_FUNCTION(x)
z = foo(y)
w = bar(x)
```

CODAMOSA →
x = 3
w = bar(x)

3. Callables Expansion

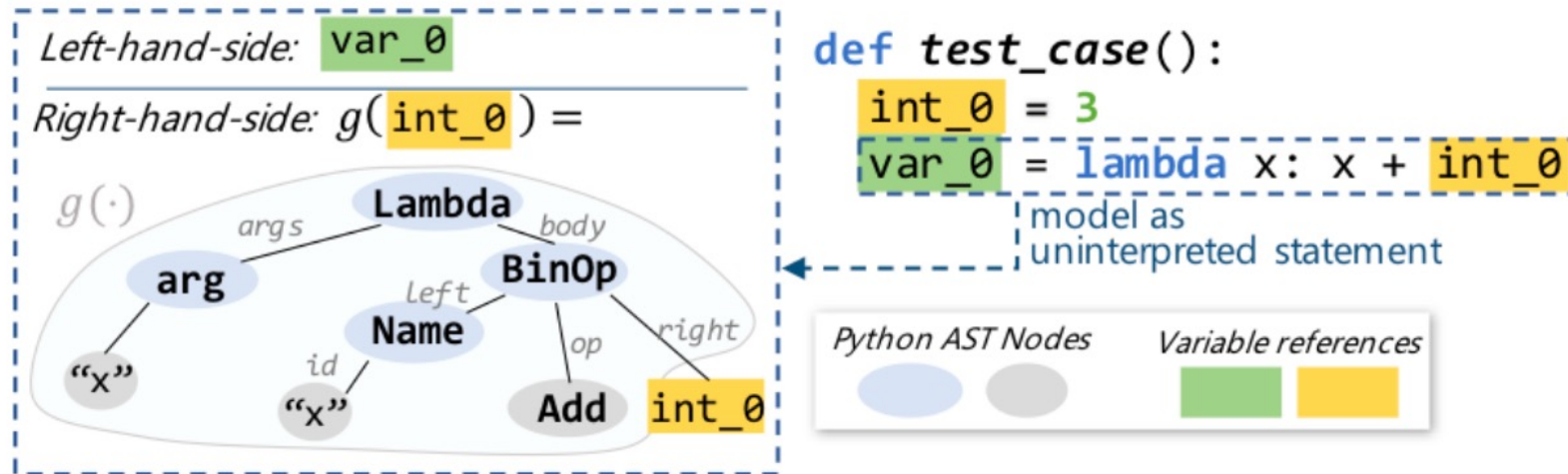
Pyguin: only parse callables in PUT

CODAMOSA: track callables via import statements in PUT

Post-processing (Test Deserialization)

4. Uninterpreted Statements

- Observation: different syntactical constructs of rhs were crucial to increasing coverage
- Add a new type statement in Pyguin: $lhs = g(vr_0, \dots, vr_n)$, where g is an operator over vr_0, \dots, vr_n

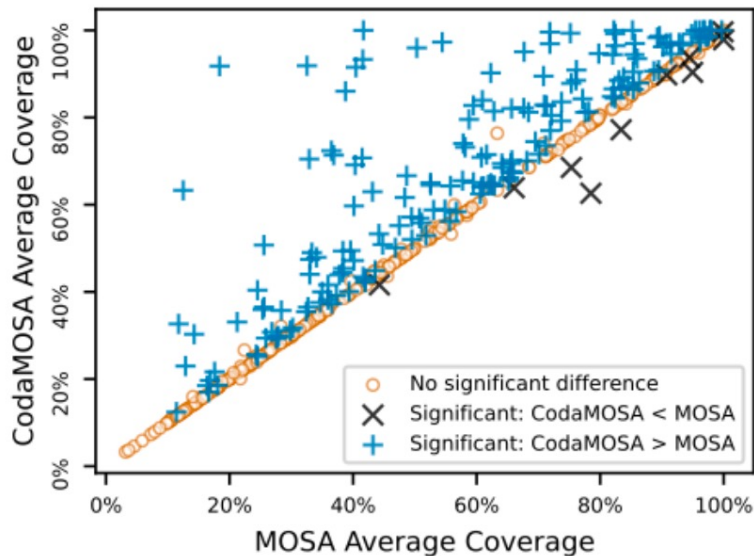


- Implement a mutation operator

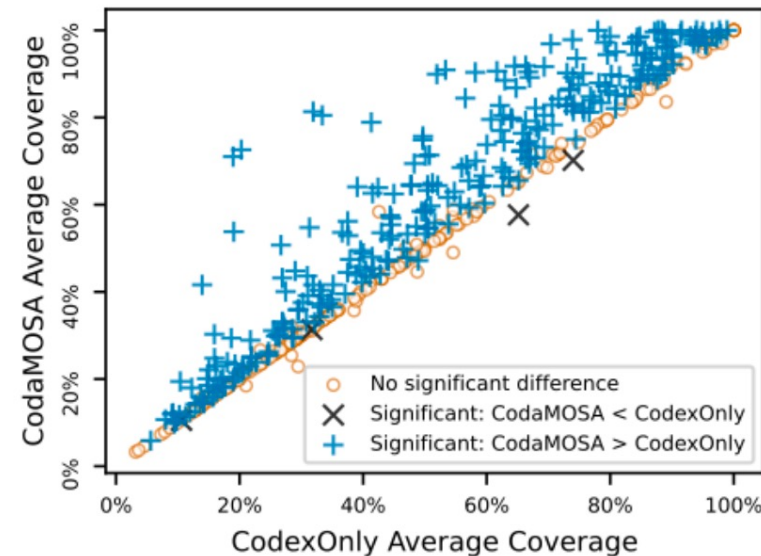
`var_0 = lambda x: x + int_0` $\xrightarrow{\text{mutation}}$ `var_0 = lambda x: x + int_1`

Evaluation

- How does CODAMOSA compare to our baselines on our benchmark set?
 - BaseLines: MOSA and CodexOnly



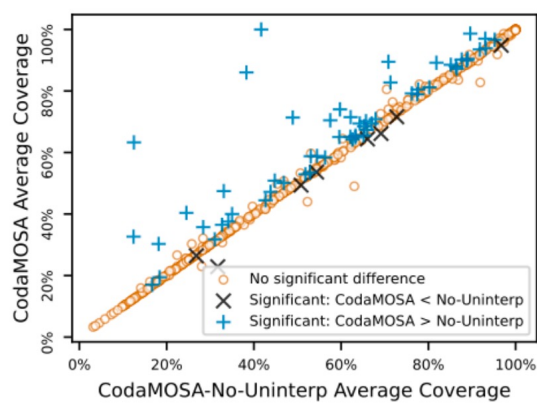
(a) Comparison to MOSA baseline. CODAMOSA has significantly higher coverage on 173 benchmarks; lower on 10.



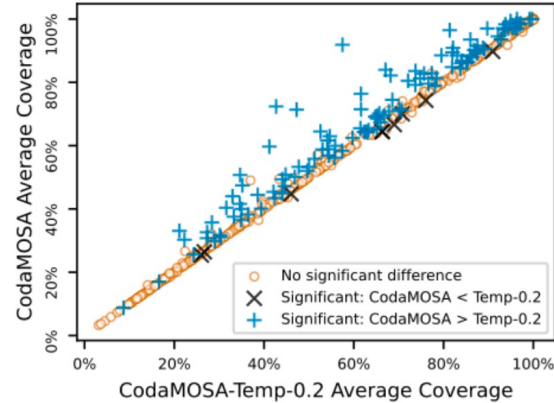
(b) Comparison to CODEXONLY baseline. CODAMOSA has significantly higher coverage on 279 benchmarks; lower on 4.

Evaluation

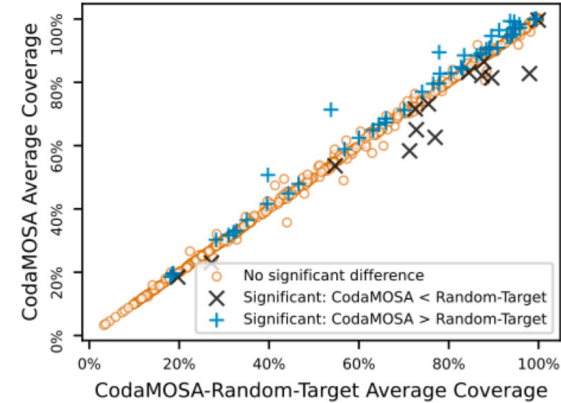
- How do our design decisions (uninterpreted statements, Codex hyper-parameters, low-coverage targeting, prompting) affect test effectiveness?



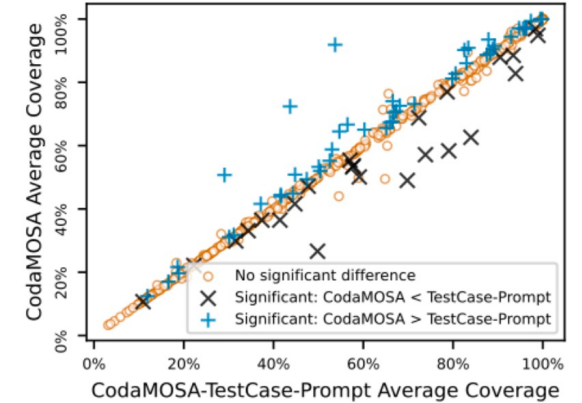
(c) Effect of uninterpreted statements. Using them, CODAMOSA has significantly higher coverage on 57 benchmarks; lower on 8.



(d) Effect of temperature. Default temp. 0.8 achieves significantly higher coverage than temp. 0.2 on 113 benchmarks; lower on 9.



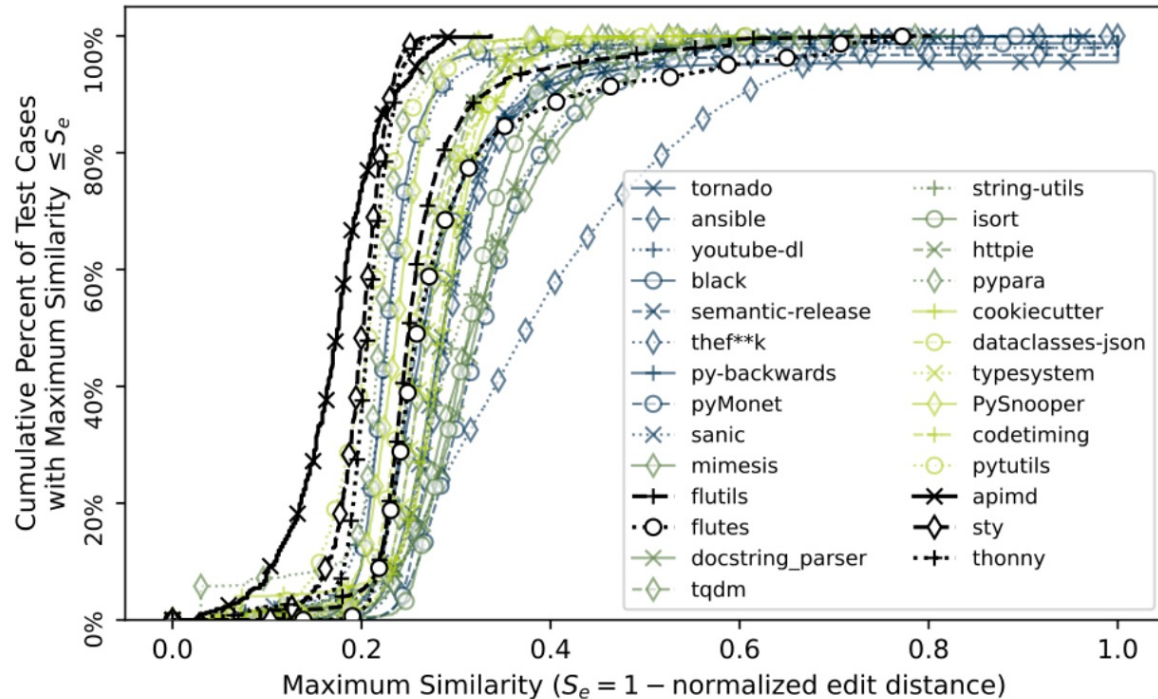
(e) Targeting low-coverage functions has significantly higher coverage than targeting random ones on 50 benchmarks; lower on 14.



(f) Default CODAMOSA has significantly higher coverage than test-case-prompted version on 49 benchmarks; lower on 24.

Evaluation

- Are Codex generations copied from out-of-prompt files in the module under test's codebase?



Takeaway. Most Codex-generated tests are not very similar to out-of-prompt test cases. On benchmarks likely outside of Codex's training set, CODAMOSA performed well.

Fig. 4: Cumulative percent of Codex-generated test cases with maximum similarity less than what is designated on x -axis. Max is over all out-of-prompt test cases in the repository.

Thanks

Comments are welcome!