Agentic Concolic Execution

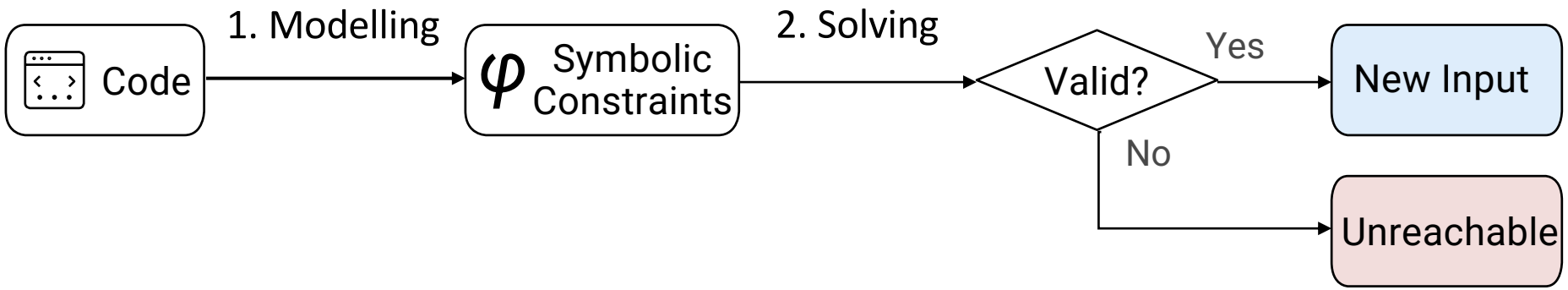
Zhengxiong Luo, **Huan Zhao**, Dylan Wolff, Cristian Cadar, Abhik Roychoudhury

*{luozx, abhik}@nus.edu.sg, {**zhaohuan**, wolffd}@comp.nus.edu.sg, c.cadar@imperial.ac.uk*

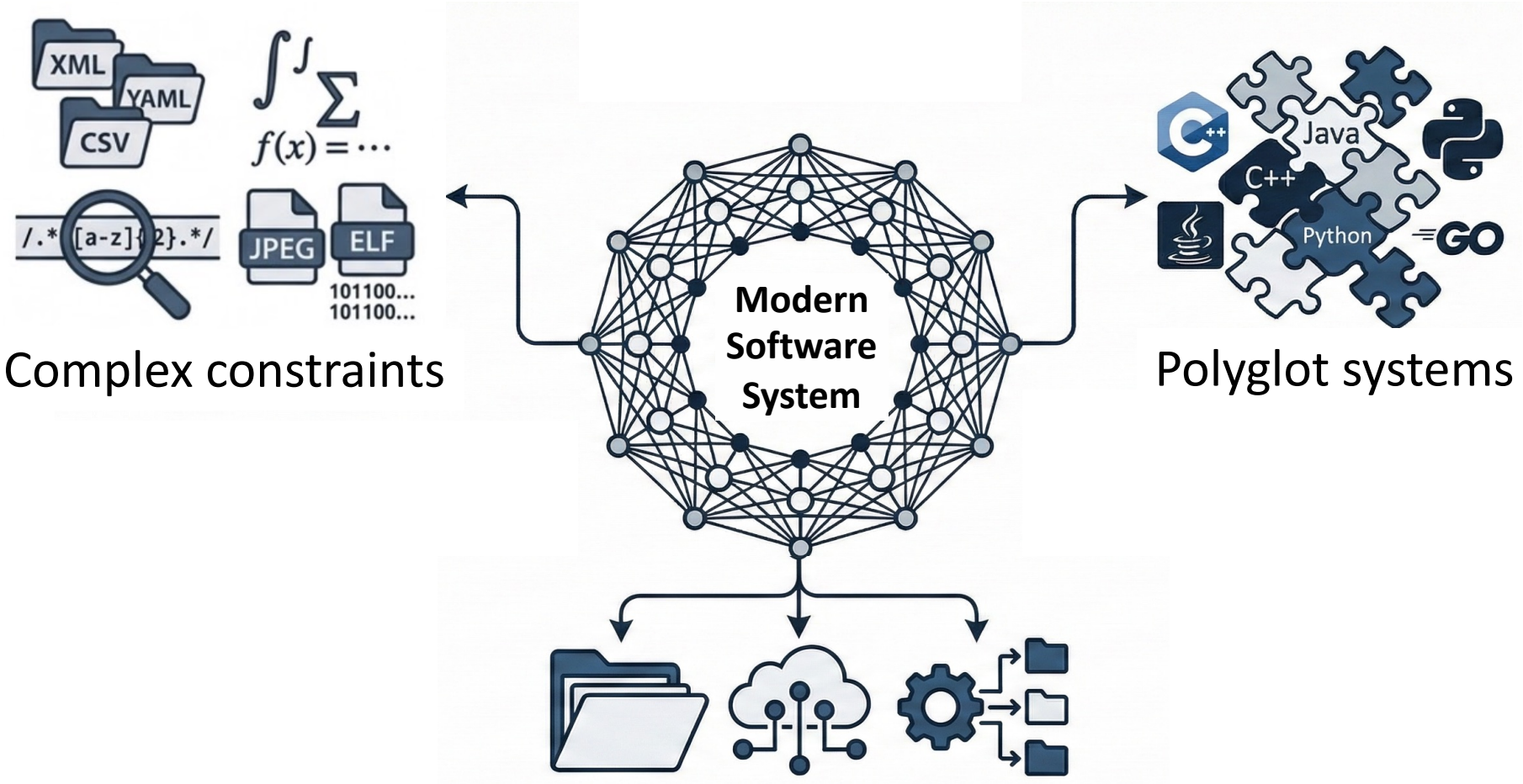
Imperial College
London

Dynamic Symbolic Execution (DSE)

- Widely-used for program analysis / vuln. detection
- Use ***symbolic values*** as program inputs to systematically explore possible program behaviors



Applying DSE on Modern Software



Complex constraints

Polyglot systems

Intricate environmental interactions

A Motivating Example (1/2)

```
1  // assume start < end
2  int count(float start, float end) {
3     int iter_count = 0;    // final answer
4     for (float cur = start; cur != end; iter_count++) {
5         unsigned temp;
6         memcpy(&temp, &cur, sizeof(float));
7         temp++;
8         memcpy(&cur, &temp, sizeof(float));
9     }
10
11    if (iter_count <= 20)
12        printf("BUG triggered!");
13 }
```

A Motivating Example (2/2)

```
1  // assume start < end
2  int count(float start, float end) {
3     int iter_count = 0;    // final answer
4     for (float cur = start; cur != end; iter_count++) {
5        unsigned temp;
6        memcpy(&temp, &cur, sizeof(float));   temp <- cur
7        temp++;                               temp <- temp + 1
8        memcpy(&cur, &temp, sizeof(float));   cur  <- temp
9     }
10
11    if (iter_count <= 20)
12        printf("BUG triggered!");
13 }
```

Count # of **representable** floating point values in [start, end)

Challenge-1: Symbolic Modeling

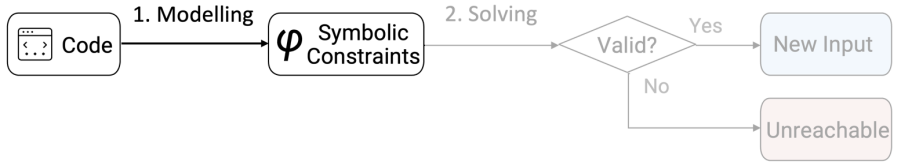
```
1  // assume start < end
2  int count(float start, float end) {
3     int iter_count = 0;    // final answer
4     for (float cur = start; cur != end; iter_count++) {
5         unsigned temp;
6         memcpy(&temp, &cur, sizeof(float));
7         temp++;
8         memcpy(&cur, &temp, sizeof(float));
9     }
10
11    if (iter_count <= 20)
12        printf("BUG triggered!");
13 }
```

floating-point

type cast

pointer

external function



Complex and incomplete symbolic modelling

Huge manual effort to support each *language & code construct & env*

Challenge-2: Constraint Solving

```
1  // assume start < end
2  int count(float start, float end) {
3     int iter_count = 0;    // final answer
4     for (float cur = start; cur != end; iter_count++) {
5         unsigned temp;
6         memcpy(&temp, &cur, sizeof(float));
7         temp++;
8         memcpy(&cur, &temp, sizeof(float));
9     }
10
11    if (iter_count <= 20)
12        printf("BUG triggered!");
13 }
```

f

Symbolic Store σ

$\texttt{start} \mapsto \alpha$

$\texttt{end} \mapsto \beta$

$\texttt{iter\_count} \mapsto i$

Initial Input

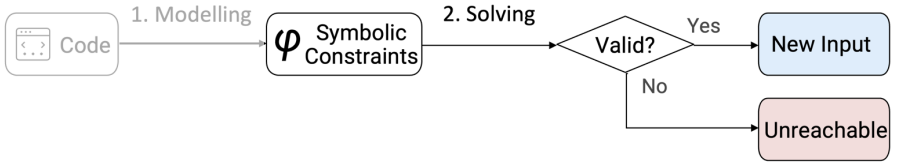
`start = 1.0`

`end = 1.00001`

(84 for-loop iterations)

Path Constraints > 1000 *clauses*

$$\bigwedge_{i=0}^{n-1} \left(f^i(\alpha) \neq \beta\right) \wedge \left(f^n(\alpha) = \beta\right) \wedge (n > 20)$$



Verbose, mechanical constraint representation

Solvers struggle with large constraints or hard theories (e.g. string, floating-point, …)

LLM is Here to Rescue…

Symbolic Modeling

Possess vast knowledge *across*


Programming Languages


Code Constructs

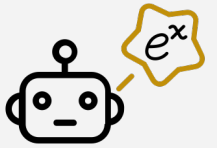

Program Environment

Constraint Solving

$$\bigwedge_{i=0}^{n-1} \left(f^i(\alpha) \neq \beta\right) \wedge \left(f^n(\alpha) = \beta\right) \wedge (n > 20)$$

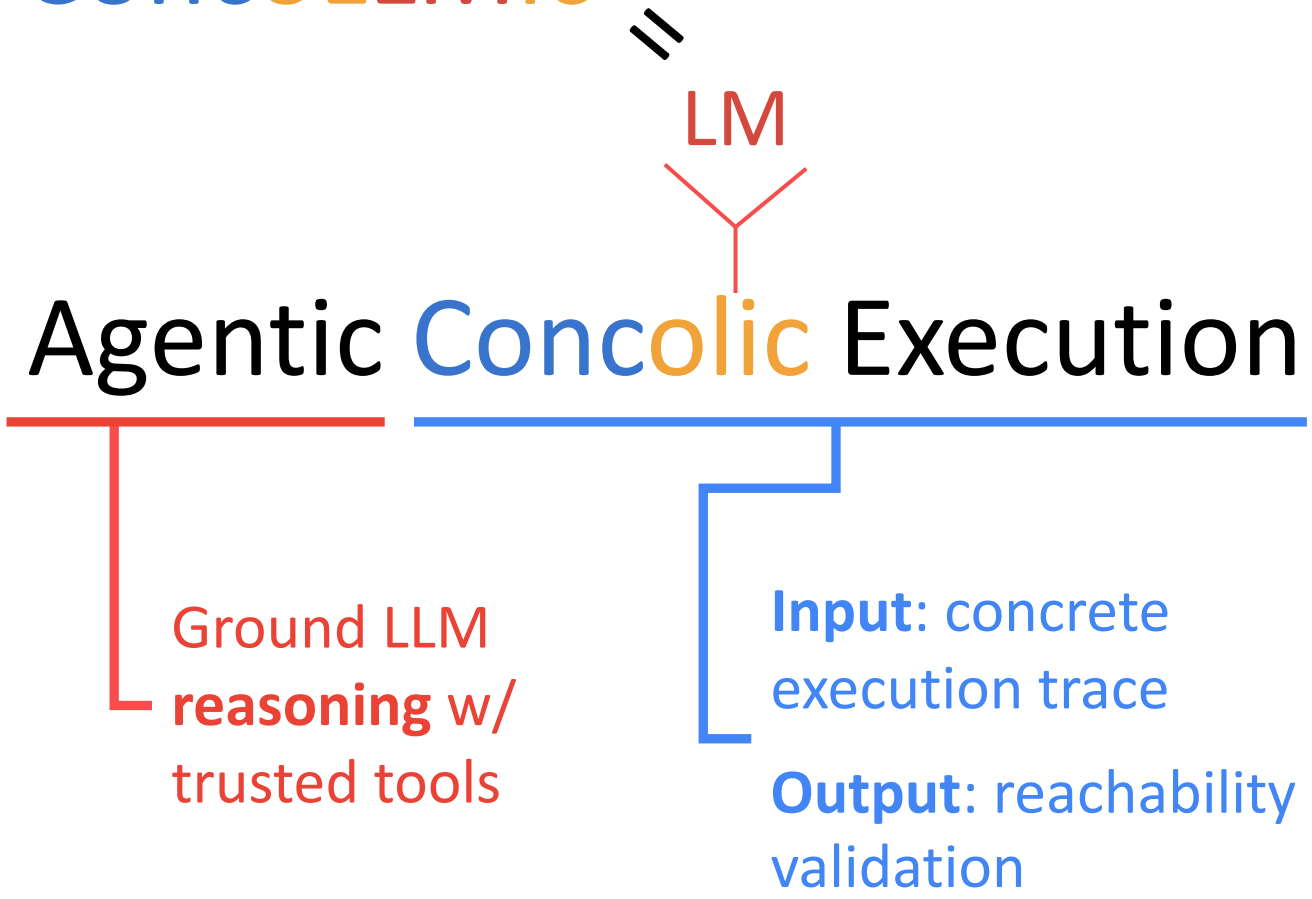
find 2 FP numbers w/ ≤ 20 *representable numbers in between*


Solve

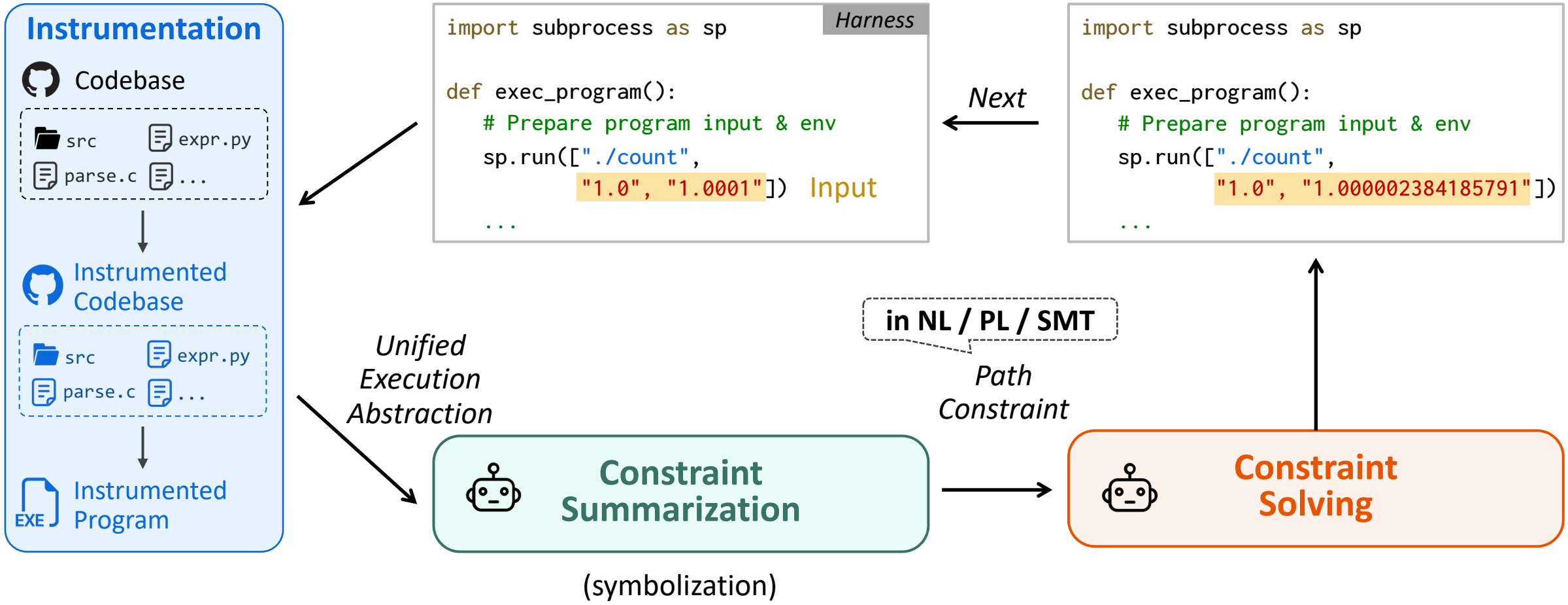


Risk: LLMs are slow, untrustworthy, and can hallucinate

Our work: ConcoLLMic



ConcoLLMic: Overview



ConcoLLMic: Execution Tracking

- *Source-code* level instrumentation – LLMs' multilingual capability

enter [func_name] [BB_ID] ⇨
Original Code Block
exit [func_name] [BB_ID] ⇨

```
int main(int argc , char **argv) {
    fprintf(stderr, "enter main 1");
    float start = atof(argv [1]), end = atof(argv [2]);
    count(start , end);
    fprintf(stderr, "exit main 1");
}
```

- *Unified* abstraction of execution traces

Global context information ⇨
Executed code
Untaken branch condition with coverage info ⇨

```
// Omitted Imports, Definitions & Variable Declarations
// assume start < end
int count(float start, float end) {
   int iteration_count = 0;    // final answer
   for (float cur = start; cur != end; iteration_count++) {
       unsigned temp;
       memcpy(&temp, &cur, sizeof(float));
       temp++;
       memcpy(&cur, &temp, sizeof(float));
   }

   if (iteration_count <= 20)
       // Unexecuted line 12 removed. Line cov: 0/1
}
```

ConcoLLMic: Summarization Agent (1/4)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Execution Abstraction

① Func Call Chain

```
[src/main.c] main => [src/count.c] count
```

② Executed Lines

count.c (13 lines total)

```
// Omitted Imports, Definitions & Variable Declarations
1  // assume start < end
2  int count(float start, float end) {
3     int iteration_count = 0;    // final answer
4     for (float cur = start; cur != end; iteration_count++) {
5         unsigned temp;
6         memcpy(&temp, &cur, sizeof(float));
7         temp++;
8         memcpy(&cur, &temp, sizeof(float));
9     }
10
11    if (iteration_count <= 20)
12        // Unexecuted line 12 removed. Line cov: 0/1
13 }
```

Given : ① *harness* ② *execution abstraction*...
Task : select target branch & generate path constraints
Tools : <CODEREQUEST, THINK, CHOOSEBRANCH, SUMMARIZE>

ConcoLLMic: Summarization Agent (2/4)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Execution Abstraction

① Func Call Chain

```
[src/main.c] main => [src/count.c] count
```

② Executed Lines

count.c (13 lines total)

```
// Omitted Imports, Definitions & Variable Declarations
1  // assume start < end
2  int count(float start, float end) {
3     int iteration_count = 0;    // final answer
4     for (float cur = start; cur != end; iteration_count++) {
5         unsigned temp;
6         memcpy(&temp, &cur, sizeof(float));
7         temp++;
8         memcpy(&cur, &temp, sizeof(float));
9     }
10
11    if (iteration_count <= 20)
12        // Unexecuted line 12 removed. Line cov: 0/1
13 }
```

Given : ① *harness* ② *execution abstraction*...
Task : select target branch & generate path constraints
Tools : <CODEREQUEST, THINK, CHOOSEBRANCH, SUMMARIZE>

{CODEREQUEST} count.c, line 12

```
|12|Uncovered|:   printf("BUG triggered!");
```

ConcoLLMic: Summarization Agent (3/4)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Execution Abstraction

① Func Call Chain

```
[src/main.c] main => [src/count.c] count
```

② Executed Lines

count.c (13 lines total)

```
// Omitted Imports, Definitions & Variable Declarations
1  // assume start < end
2  int count(float start, float end) {
3      int iteration_count = 0;    // final answer
4      for (float cur = start; cur != end; iteration_count++) {
5          unsigned temp;
6          memcpy(&temp, &cur, sizeof(float));
7          temp++;
8          memcpy(&cur, &temp, sizeof(float));
9      }
10
11     if (iteration_count <= 20)
12         // Unexecuted line 12 removed. Line cov: 0/1
13 }
```

Given : ① *harness* ② *execution abstraction*...
Task : select target branch & generate path constraints
Tools : <CODEREQUEST, THINK, CHOOSEBRANCH, SUMMARIZE>

{CODEREQUEST} count.c, line 12

```
|12|Uncovered|:   printf("BUG triggered!");
```

{CHOOSEBRANCH}
[*target_branch*] `if (iteration_count <= 20) => True`
[*rationale*] It represents a key behavior & has 0% cov.
[*lines_to_cover*] count.c:12-12

Your selection is recorded!

ConcoLLMic: Summarization Agent (4/4)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Execution Abstraction

① Func Call Chain

```
[src/main.c] main => [src/count.c] count
```

② Executed Lines

count.c (13 lines total)

```
// Omitted Imports, Definitions & Variable Declarations
1  // assume start < end
2  int count(float start, float end) {
3      int iteration_count = 0;    // final answer
4      for (float cur = start; cur != end; iteration_count++) {
5          unsigned temp;
6          memcpy(&temp, &cur, sizeof(float));
7          temp++;
8          memcpy(&cur, &temp, sizeof(float));
9      }
10
11     if (iteration_count <= 20)
12         // Unexecuted line 12 removed. Line cov: 0/1
13 }
```

Given : ① *harness* ② *execution abstraction*...
Task : select target branch & generate path constraints
Tools : <CODEREQUEST, THINK, CHOOSEBRANCH, SUMMARIZE>

{CODEREQUEST} count.c, line 12

```
|12|Uncovered|:    printf("BUG triggered!");
```

{CHOOSEBRANCH}
[*target_branch*] `if (iteration_count <= 20) => True`
[*rationale*] It represents a key behavior & has 0% cov.
[*lines_to_cover*] count.c:12-12

Your selection is recorded!

{SUMMARIZE} [*path_constraint*]
a) 2 CLI arguments: `start < end`
b) `start`, `end` are valid floating-points
c) # IEEE-754 floating-point $\in$ `[start, end)` ≤ 20

ConcoLLMic: Solving Agent (1/5)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Path Constraint (for target branch)

a) 2 CLI arguments: `start < end`

b) `start`, `end` are valid floating-points

c) # IEEE-754 floating-point $\in$ `[start, end)` ≤ 20

Given : ① *harness* ② *new path constraints* ...
Task : solve path constraint for a new input…
Tools : <ExecuteCode, QuerySMT, Think, GenerateTest>

ConcoLLMic: Solving Agent (2/5)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Path Constraint (for target branch)

a) 2 CLI arguments: `start < end`

b) `start`, `end` are valid floating-points

c) `#` IEEE-754 floating-point $\in$ `[start, end)` ≤ 20

Given : ① *harness* ② *new path constraints* ...
Task : solve path constraint for a new input…
Tools : <EXECUTECODE, QUERYSMT, THINK, GENERATETEST>

{THINK} Original range [`1.0`,`1.00001`) contains > 20 representable floats.
Let's fix `start=1.0` and calculate `end`

{THINK} For float around `1.0`, the smallest increment is $\sim 2^{-23} \approx 1.19e^{-7}$
So the range is $\sim 20 \times 1.19e^{-7} \approx 2.38e^{-6}$ ... Let me try to verify ...

Your reasoning is recorded!

ConcoLLMic: Solving Agent (3/5)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Path Constraint (for target branch)

a) 2 CLI arguments: `start < end`
b) `start`, `end` are valid floating-points
c) # IEEE-754 floating-point ∈ `[start, end)` ≤ 20

Given : ① *harness* ② *new path constraints* ...
Task : solve path constraint for a new input…
Tools : <EXECUTECODE, QUERYSMT, THINK, GENERATETEST>

{THINK} Original range [1.0,1.00001) contains > 20 representable floats.
Let's fix `start=1.0` and calculate `end`

{THINK} For float around 1.0, the smallest increment is ~ $2^{-23} \approx 1.19e^{-7}$
So the range is ~ $20 \times 1.19e^{-7} \approx 2.38e^{-6}$ ... Let me try to verify ...

Your reasoning is recorded!

{EXECUTECODE}

```
def next_float(f): ...  # calculate next float

current = 1.0
for i in range(19):
    current = next_float(current)
print(next_float(current)); # the 20th value
```

Exec. Result: `stdout = 1.000002384185791`

ConcoLLMic: Solving Agent (4/5)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Path Constraint (for target branch)

a) 2 CLI arguments: `start < end`
b) `start`, `end` are valid floating-points
c) # IEEE-754 floating-point ∈ `[start, end)` ≤ 20

Given : ① *harness* ② *new path constraints* ...
Task : solve path constraint for a new input…
Tools : <EXECUTECODE, QUERYSMT, THINK, GENERATETEST>

{THINK} Original range [1.0,1.00001) contains > 20 representable floats.
Let's fix `start=1.0` and calculate `end`

{THINK} For float around 1.0, the smallest increment is ~ $2^{-23} \approx 1.19e^{-7}$
So the range is ~ $20 \times 1.19e^{-7} \approx 2.38e^{-6}$ ... Let me try to verify ...

Your reasoning is recorded!

{EXECUTECODE}

```
def next_float(f): ...  # calculate next float

current = 1.0
for i in range(19):
    current = next_float(current)
print(next_float(current)); # the 20th value
```

Exec. Result: `stdout = 1.000002384185791`

{GENERATETEST} [*is_satisfiable*] ✓

```
subprocess.run(["./count", "1.0", "1.000002384185791"])
```

ConcoLLMic: Solving Agent (5/5)

Harness

```
subprocess.run(["./count", "1.0", "1.0001"])
```

Path Constraint (for target branch)

a) 2 CLI arguments: `start < end`
b) `start`, `end` are valid floating-points
c) # IEEE-754 floating-point ∈ `[start, end)` ≤ 20

from summarization agent

{CHOOSEBRANCH}
[*target_branch*] `if (iteration_count <= 20) => True`
[*rationale*] ...
[*lines_to_cover*] **count.c:12-12**

Execution + Reachability Validation

Given : ① *harness* ② *new path constraints* ...
Task : solve path constraint for a new input…
Tools : <EXECUTECODE, QUERYSMT, THINK, GENERATETEST>

{THINK} Original range [1.0,1.00001) contains > 20 representable floats.
Let's fix `start=1.0` and calculate `end`

{THINK} For float around 1.0, the smallest increment is ~ $2^{-23} \approx 1.19e^{-7}$
So the range is ~ $20 \times 1.19e^{-7} \approx 2.38e^{-6}$ ... Let me try to verify ...

Your reasoning is recorded!

{EXECUTECODE}

```
def next_float(f): ...  # calculate next float

current = 1.0
for i in range(19):
    current = next_float(current)
print(next_float(current)); # the 20th value
```

Exec. Result: `stdout = 1.000002384185791`

{GENERATETEST} [*is_satisfiable*] ✓

```
subprocess.run(["./count", "1.0", "1.000002384185791"])
```

Evaluation: Setup

LLM: *claude-sonnet-3.7* (200K context window)

Benchmark: **8 × Monolingual** **4 × Polyglot**

* Max: 83 KLoC

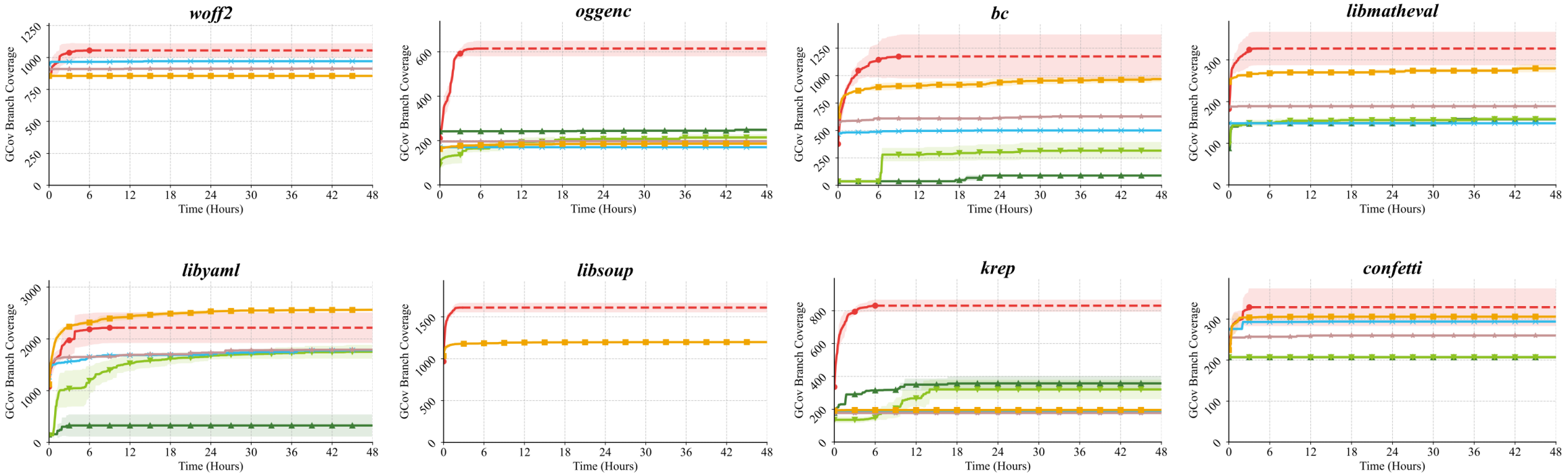
Baselines: State-of-the-Art DSEs & Fuzzer

SymCC SymSan

& Two variants
- Pending
- Float

Evaluation: Coverage – Monolingual (C/C++)

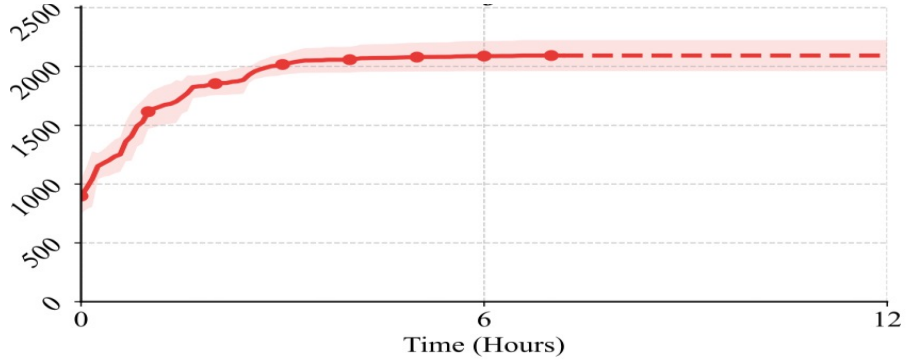
KLEE	KLEE-Pending	SymCC	SymSan	AFL++
+233%	+135%	+130%	+115%	+81%



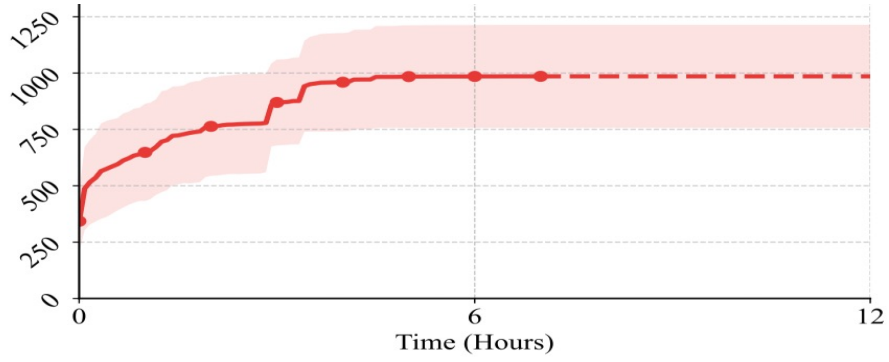
Evaluation: Coverage – Polyglot

Tested with *zero* additional setup effort. Consistent coverage growth.

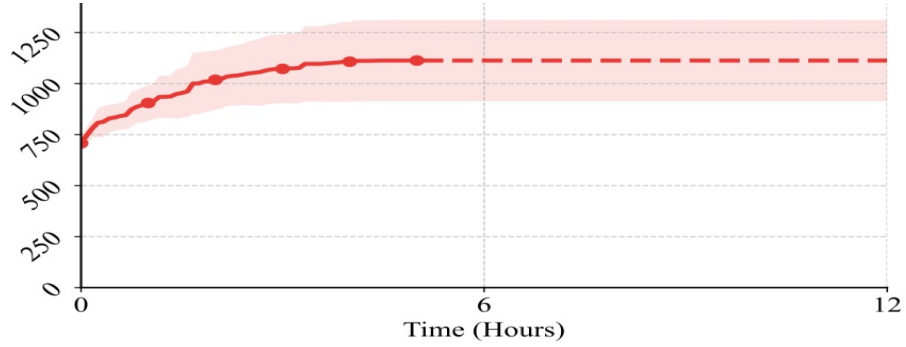
ultrajson



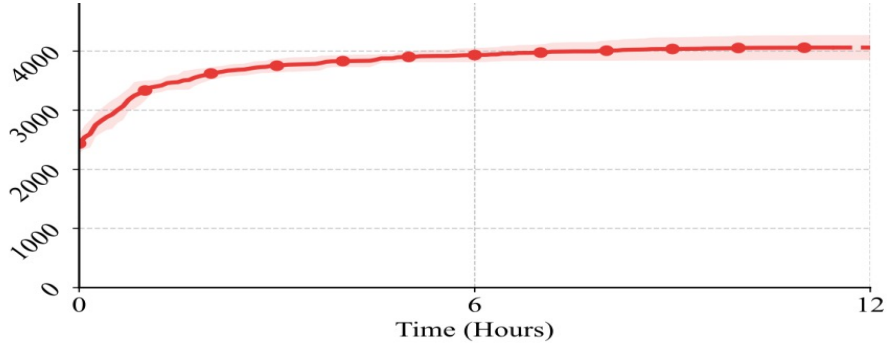
jansi



py4j



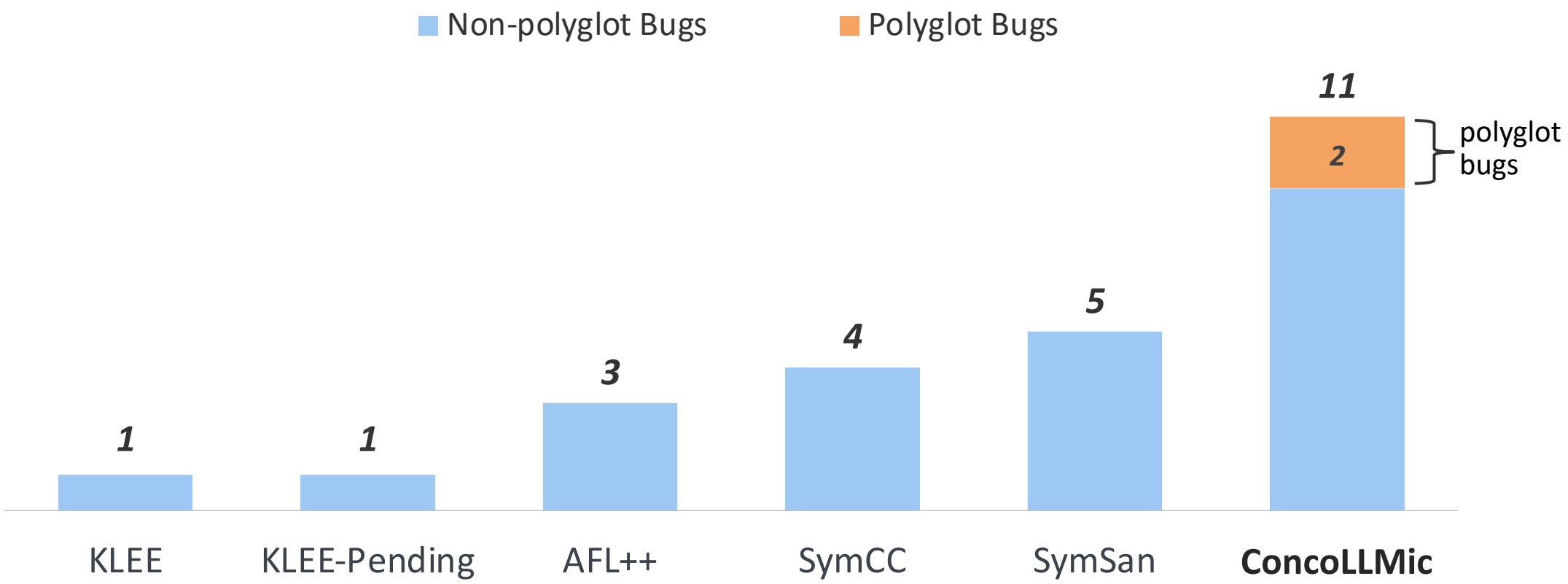
protobuf



Languages

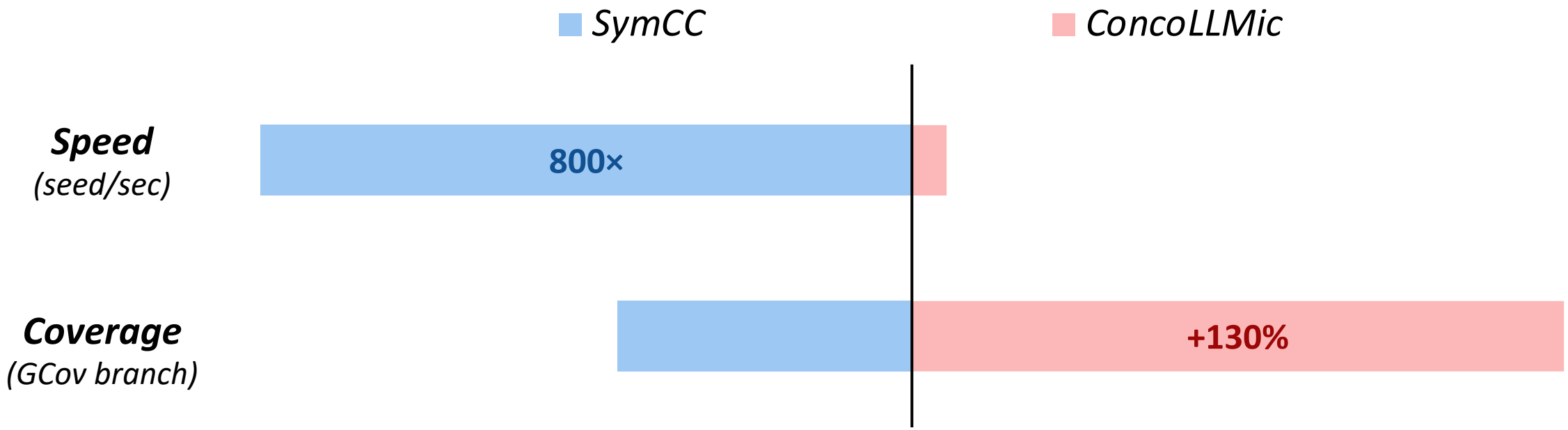
- Python
- C
- Java
- C++
- Go

Evaluation: Bug Detection



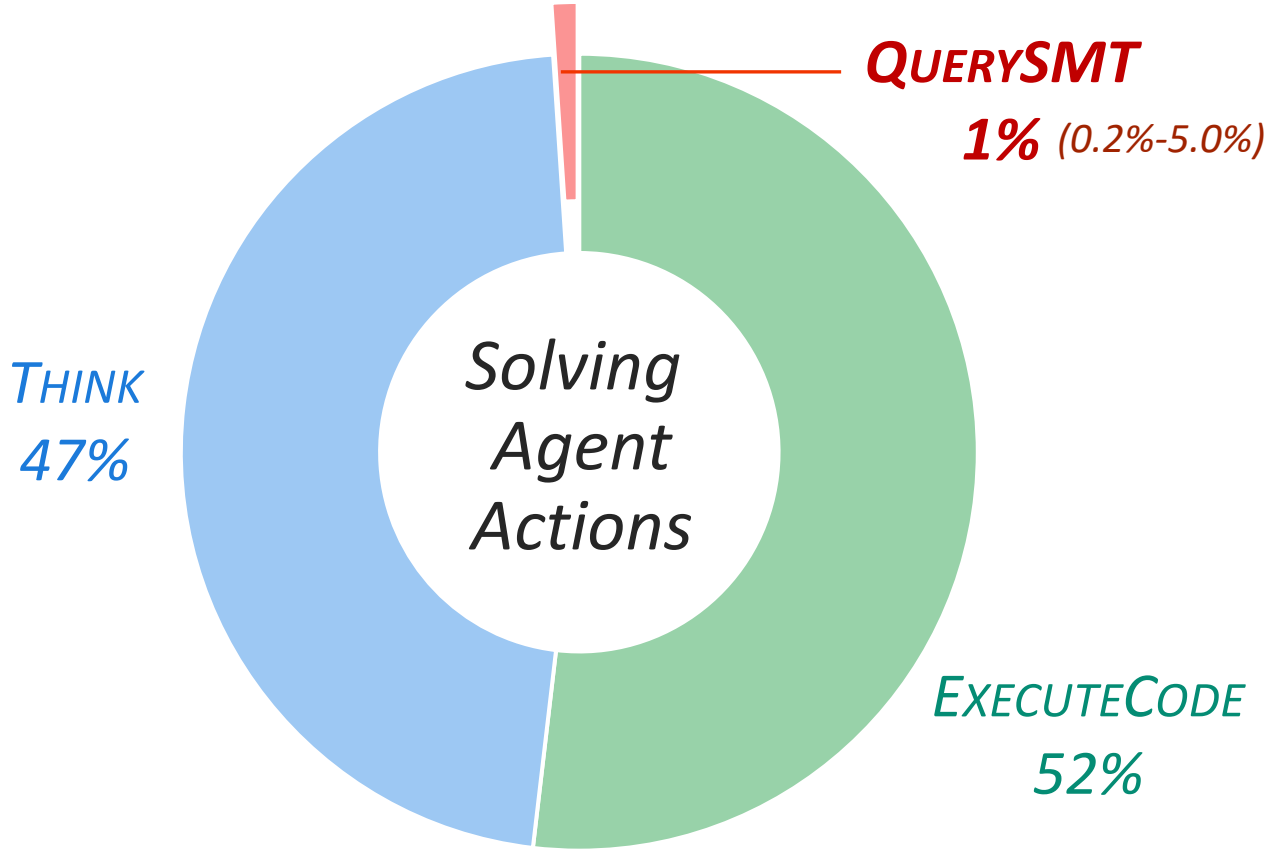
Findings 1: Slower Pace, Superior Results

✓ ConcoLLMic prioritizes input quality over quantity

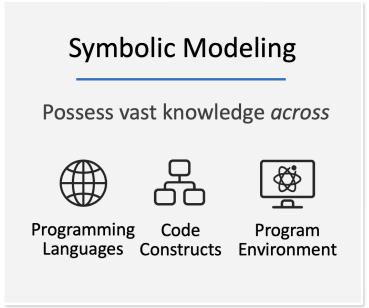


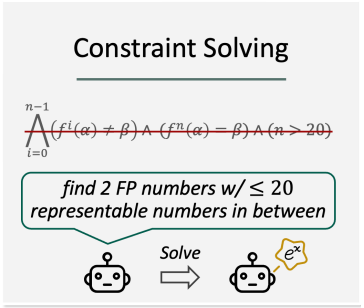
Findings 2: Reasoning Beyond Formal Solvers

✓ LLM agents reason about code path *without* relying on SMT



Summary





Risk: LLMs are slow, untrustworthy, and can hallucinate

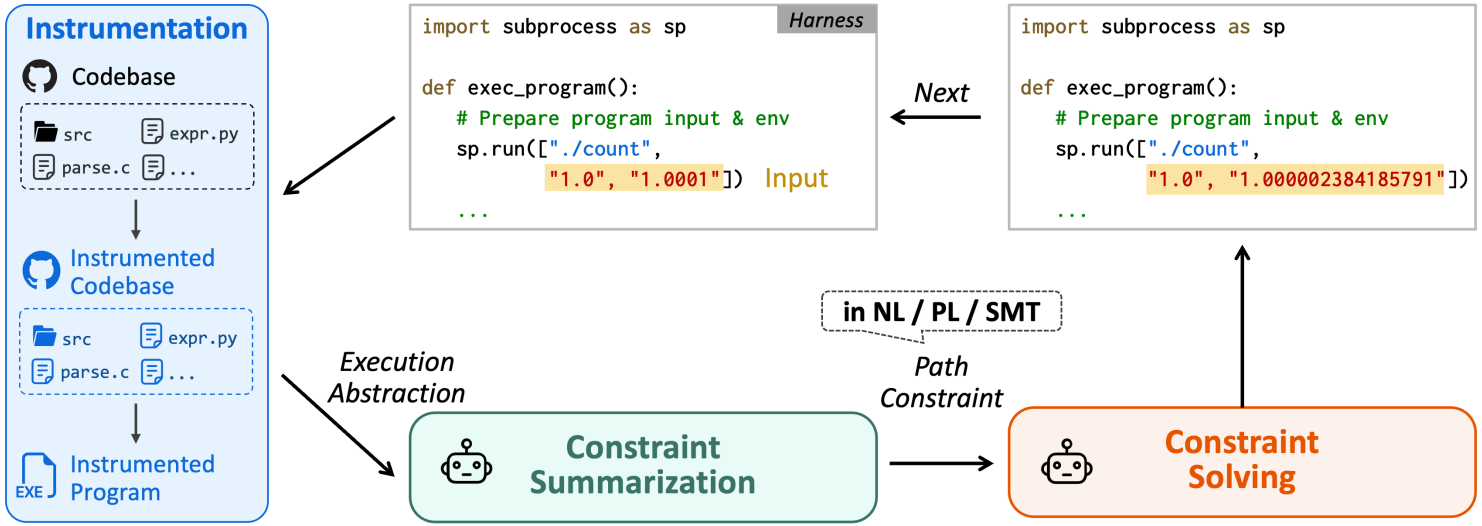
LM = ConcoLLMic

Agentic Concolic Execution

Ground LLM **reasoning** w/ trusted tools

Input: concrete execution trace

Output: reachability validation



Paper

Code