ASE 2026 Mon 12 - Fri 16 October 2026 Munich, Germany

Large Language Model Enabled Symbolic Execution for Automated Functional Analysis

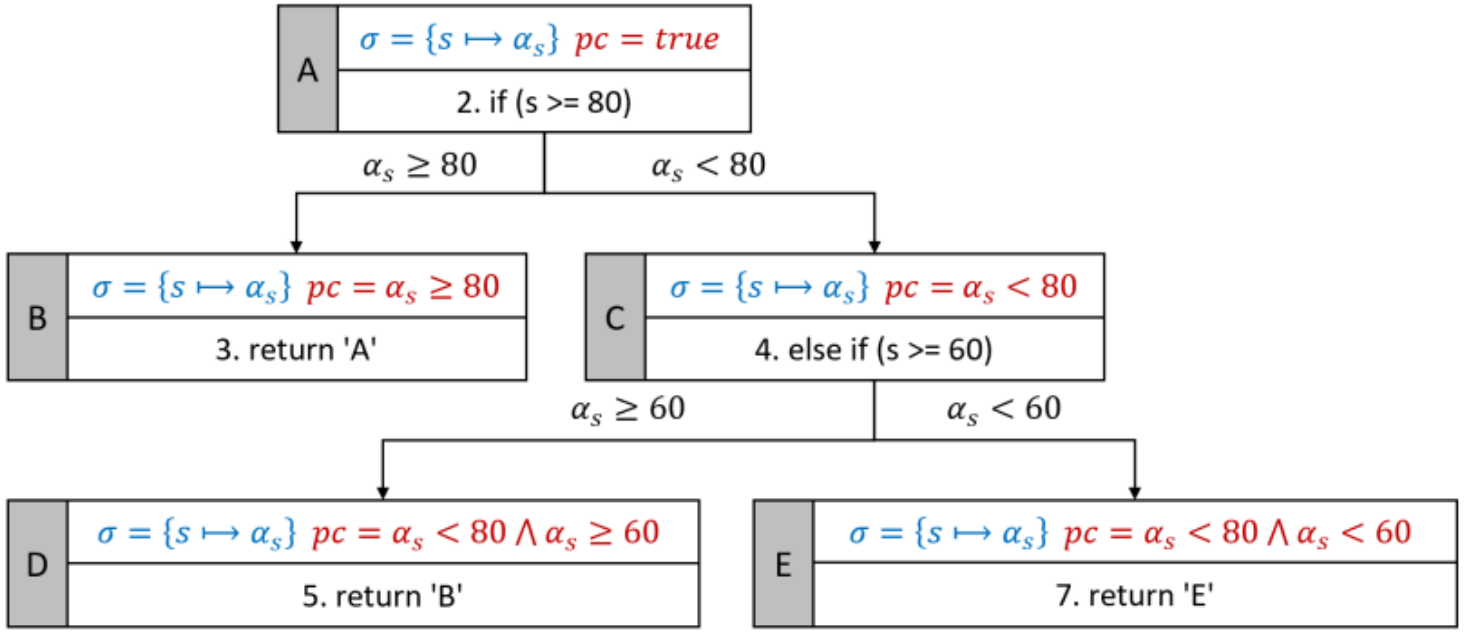
Meixi Liu, Zhenbang Chen, Xudong Wang, Ziran He,

Jiachen Gu, Minghui Chen, Lei Wang, Wei Dong



Symbolic Execution

```
1  public char s2g(int s) {
2     if (s >= 80)
3        return 'A';   // path 1
4     else if (s >= 60)
5        return 'B';   // path 2
6     else
7        return 'E';   // path 3
8  }
```



A precise program analysis technique that reasons about both data and control dependence

Execution Without Comprehension

- Symbolic execution is powerful at exploring feasible paths.
- But it lacks comprehension of the program's *functional semantics* -

  both at the high level and at the path level.

$\alpha_s \geq 80 \wedge \text{ret} = \text{'A'}$ // pc of path 1
$\alpha_s < 80 \wedge \alpha_s \geq 60 \wedge \text{ret} = \text{'B'}$ // pc of path 2
$\alpha_s < 80 \wedge \alpha_s < 60 \wedge \text{ret} = \text{'E'}$ // pc of path 3



valid score in [60, 80)
should return grade "B"

between path and functionality

Functional Reasoning is Needed!

What functionality does each feasible path implement?

Functional Analysis

Which functionalities are implemented and which are not?

Functional Coverage

Are there any inconsistencies between the path behavior and the specification?

Functional Defect Detection

Existing Approaches

Formal Specifications
Liu & Nakajima, TSE 2022



Labor-intensive to create manually

Differential Testing
McKeeman, Digital Technical Journal 1998



Demands significant expert effort

Metamorphic Testing
Barr et al., TSE 2015



Limited to specific scenarios

Specification Mining
Ammons et al., POPL 2002

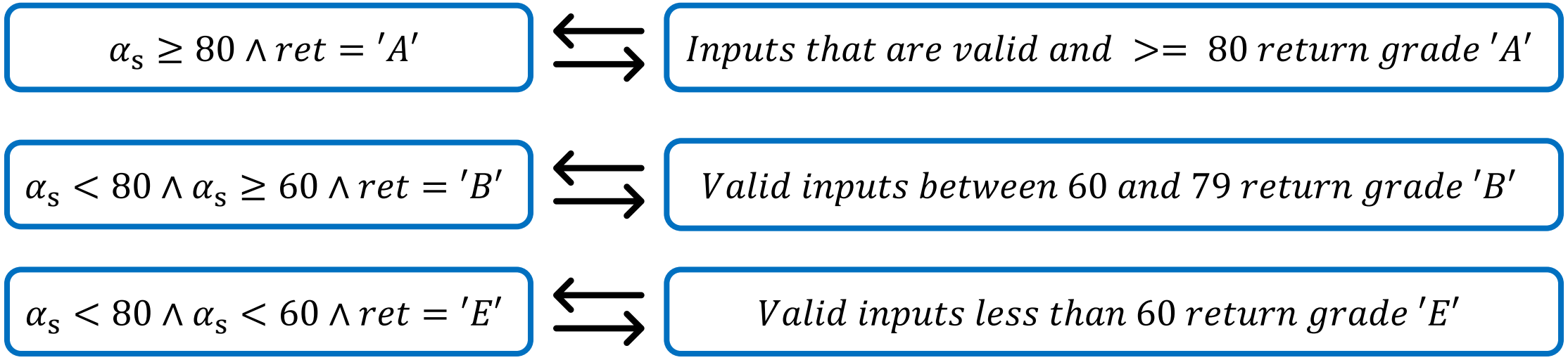


Suffer from imprecision and incompleteness

Automatically comprehending program functionality remains largely open!

Our Insight

- Symbolic paths serve as structured units for functional analysis
- Automatically linking path constraints to their functional semantics

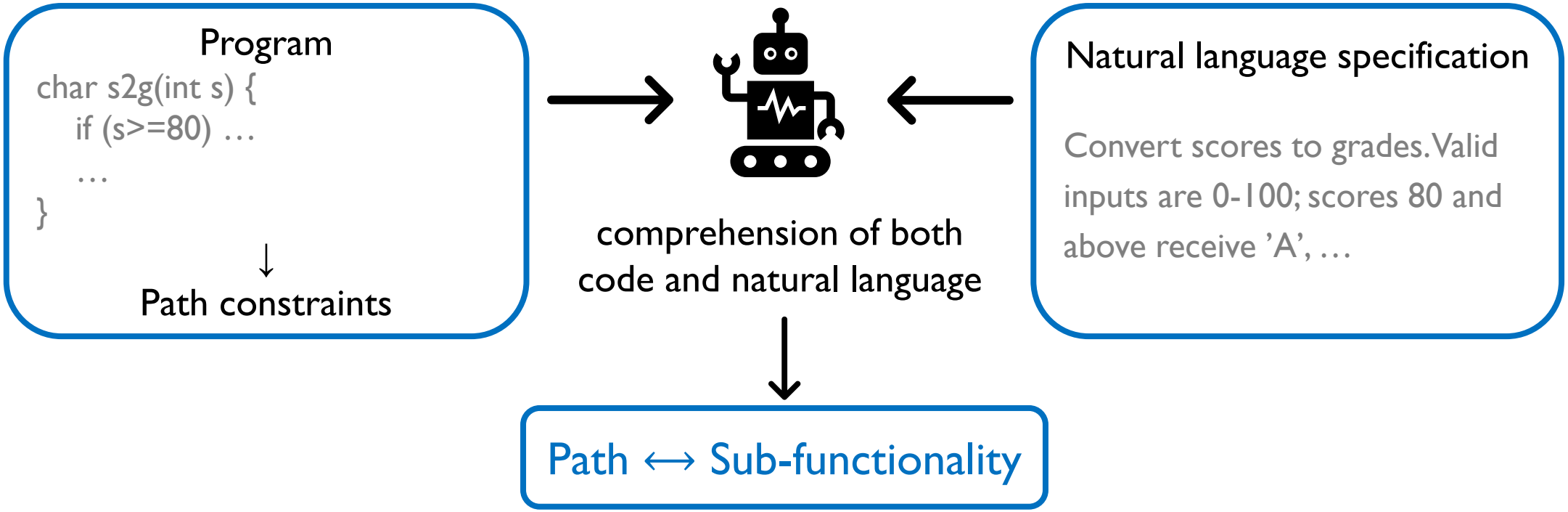


comprehension of both
code and natural language

Key Idea

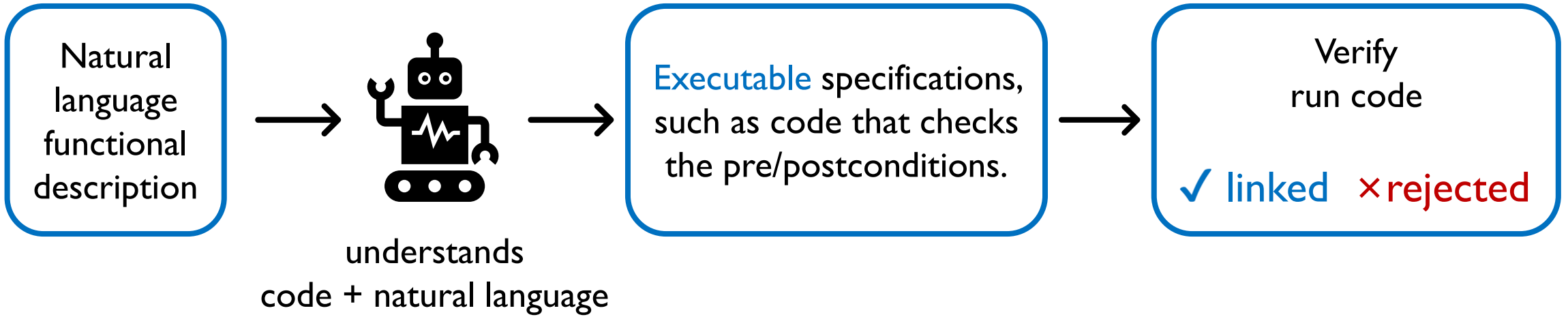
Leveraging Large Language Models (LLMs) to connect

symbolic execution paths with requirement-derived functional specifications



Technical Challenges (1/2)

Reliably linking path constraints to LLM-generated specifications

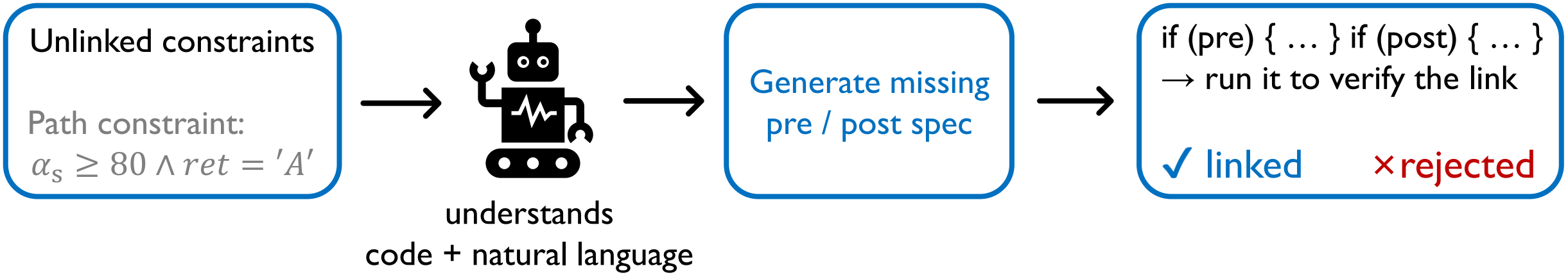


No intermediate representation → No semantic gap

Easily adaptable to different programming languages

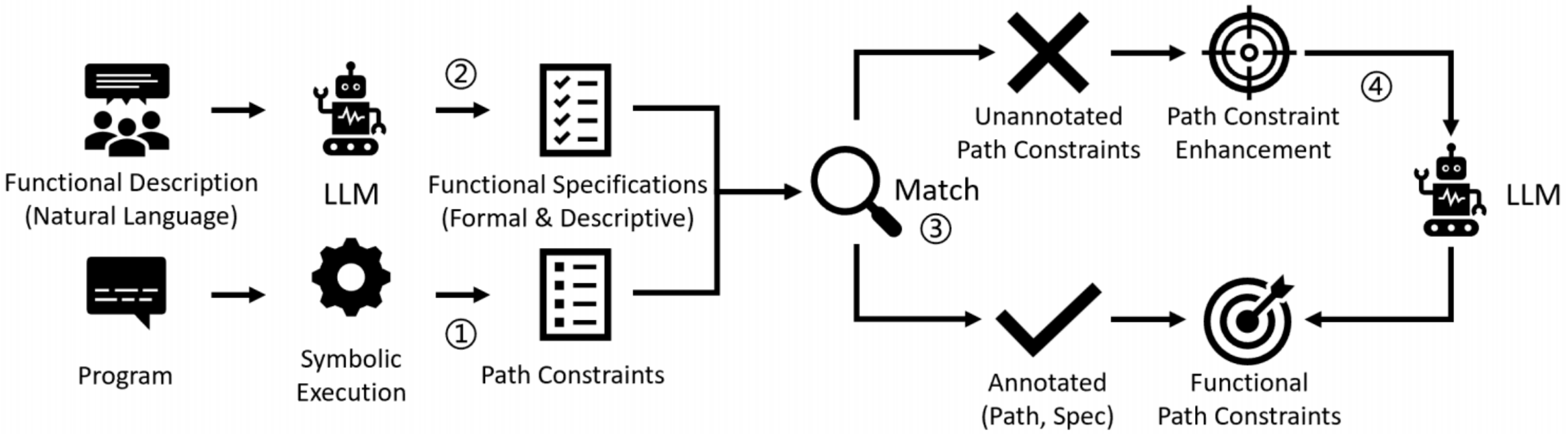
Technical Challenges (2/2)

Handling path constraints that cannot be linked to any specification

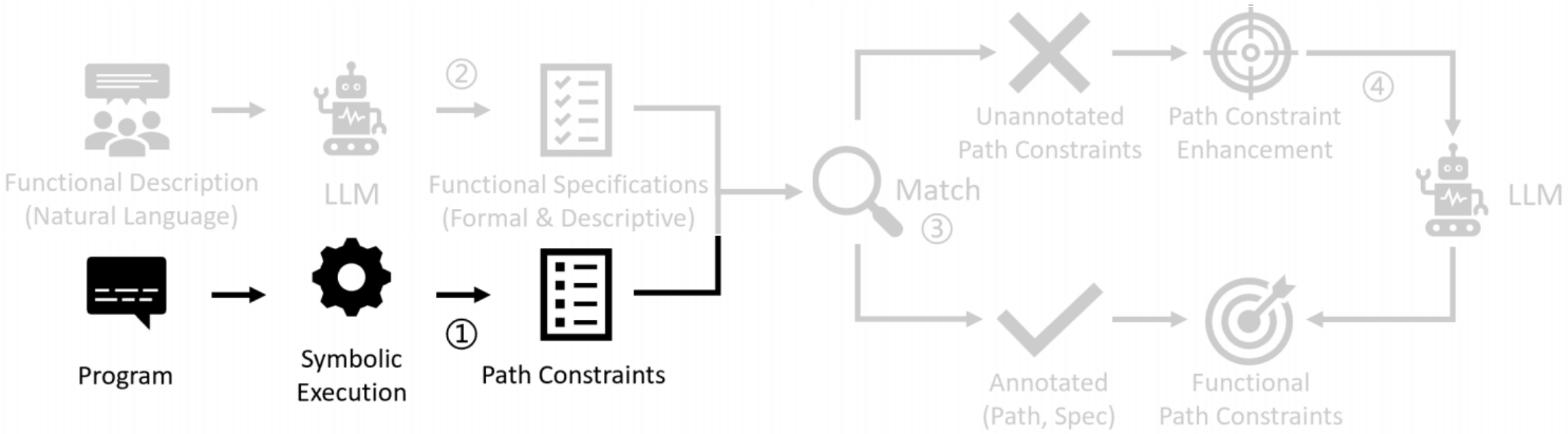


Generate what is missing, then validate it by execution.

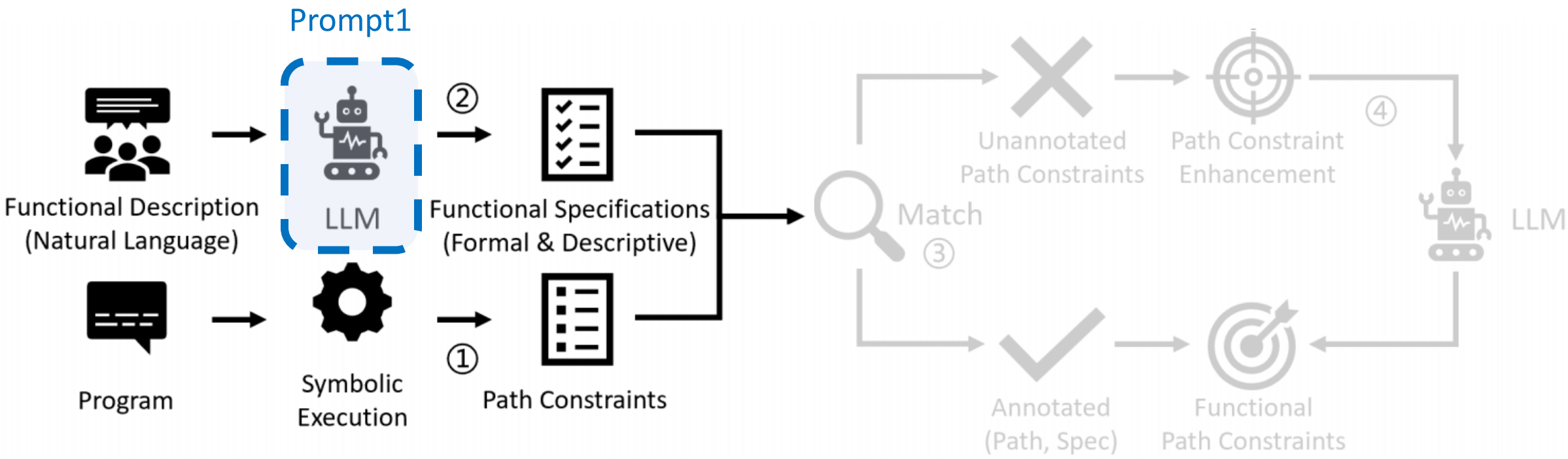
Workflow of FuSE



Workflow of FuSE



Workflow of FuSE



Prompt 1: Requirement-derived Specifications

Prompt input

Natural-language requirements

Function signature and target language

The implementation is excluded

Requested output

For each sub-functionality:

Description

Executable Precondition

Executable Postcondition

Natural Language to Functional Specification

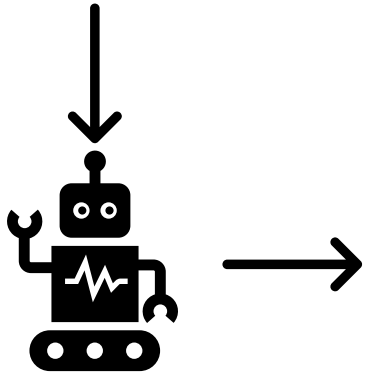
Convert scores to grades.

Valid inputs are 0-100;

scores 80 and above receive 'A',

scores 60 and above receive 'B',

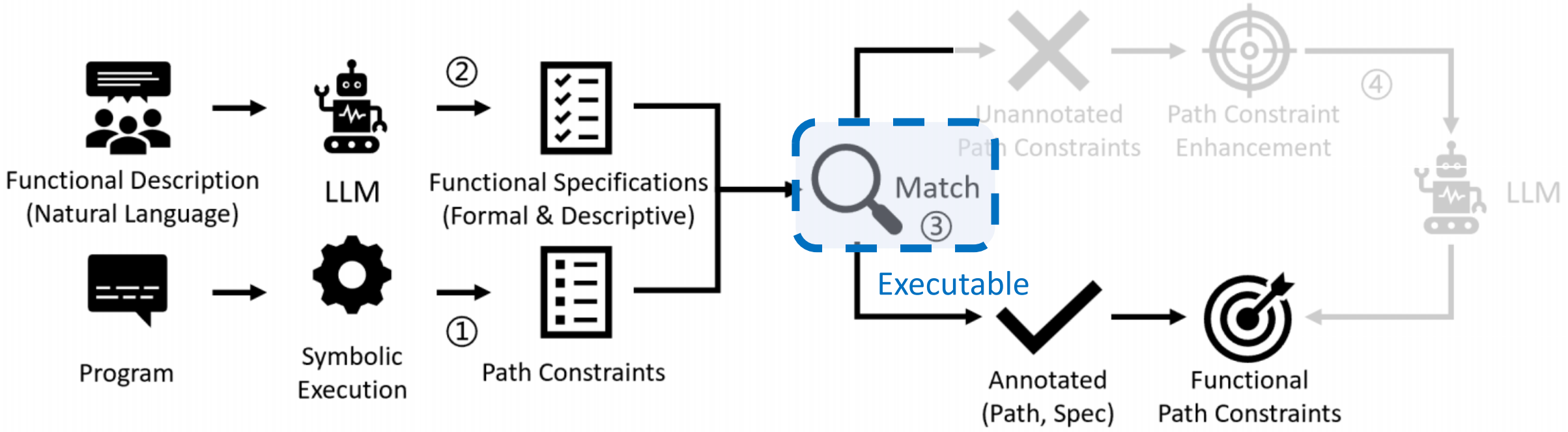
and scores below 60 receive 'E'.



```
# Partition of sub-functionality 1
Description: Inputs that are valid and ≥ 80 should return grade 'A'.
Precondition: s >= 80 && s <= 100
Postcondition: return == 'A'
# Partition of sub-functionality 2
Description: Valid inputs between 60 and 79 should return grade 'B'.
Precondition: s >= 60 && s < 80
Postcondition: result == 'B'
# Partition of sub-functionality 3
Description: Inputs that are valid and less than 60 should return grade 'E'.
Precondition: s >= 0 && s < 60
Postcondition: return == 'E'
# Partition for sub-functionality 4
Description: Any input outside [0, 100] should be treated as invalid.
Precondition: s < 0 || s > 100
Postcondition: Throws an IllegalArgumentException.
```

Each partition includes a description and a pre/postcondition pair

Workflow of FuSE



Path - Specification Matching

Pre passes, Post passes ⇆ Annotate the path and mark the specification covered.

Pre passes, Post fails ⇆ Annotate the intended functionality and report an inconsistency candidate.

No Pre passes ⇆ Send the unannotated path to Prompt 2.

An annotation and a fully validated match are different outcomes

Path - Specification Matching

$$pc \Longrightarrow Pre \ \wedge pc \Rightarrow Post$$

$\alpha_s \geq 80 \wedge \text{ret} = \text{'A'}$ // pc of path 1
$\alpha_s < 80 \wedge \alpha_s \geq 60 \wedge \text{ret} = \text{'B'}$ // pc of path 2
$\alpha_s < 80 \wedge \alpha_s < 60 \wedge \text{ret} = \text{'E'}$ // pc of path 3

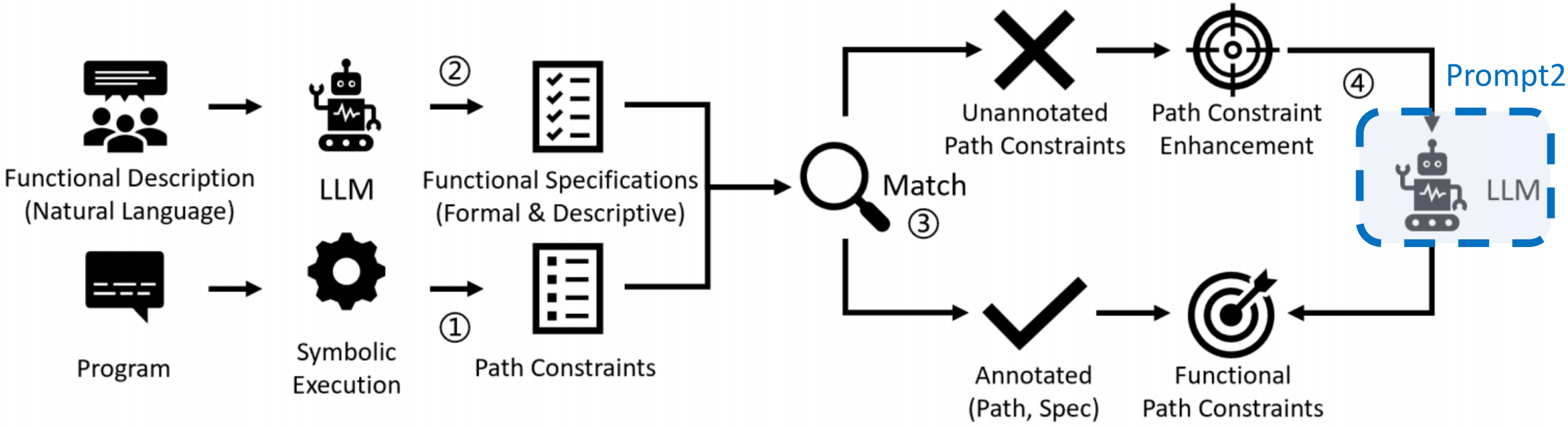
```
// pc of path 2
# Description: Valid inputs between 60 and 79 should return grade 'B'.
# Precondition: s >= 60 && s < 80
# Postcondition: result == 'B'
```
$\alpha_s < 80 \wedge \alpha_s \geq 60 \wedge \text{ret} = \text{'B'}$

$$pc \Longrightarrow Pre \ \wedge \neg(pc \Rightarrow Post)$$

$\longrightarrow$

potential implementation-specification inconsistencies
and reported as defect candidates for user confirmation

Workflow of FuSE



Prompt 2: Specifications for Unannotated Paths

Prompt input

A path has no matching precondition.

Provide the requirements, signature,

and path constraint.

Requested output

Generate one description with executable

Precondition and Postcondition.

Check both against path constraint.

Retry failures up to five attempts.

If validation keeps failing, the path remains unannotated.

Path Constraint Enhancement

$\alpha_s \geq 80 \wedge \text{ret} = 'A'$ // pc of path 1
$\alpha_s < 80 \wedge \alpha_s \geq 60 \wedge \text{ret} = 'B'$ // pc of path 2
$\alpha_s < 80 \wedge \alpha_s < 60 \wedge \text{ret} = 'E'$ // pc of path 3



// pc of path 1
Description: Inputs that are valid and ≥ 80 should return grade 'A'.
Precondition: s >= 80
Postcondition: result == 'A'
$\alpha_s \geq 80 \wedge \text{ret} = 'A'$
// pc of path 3
Description: Inputs that are valid and less than 60 should return grade 'E'.
Precondition: s < 60
Postcondition: result == 'E'
$\alpha_s < 80 \wedge \alpha_s < 60 \wedge \text{ret} = 'E'$

Prompt 3: Tests for Uncovered Specifications

Prompt input	Requested output
Original requirements	A concrete input I
Function signature	An expected output O
One uncovered specification:	Validate Pre(I) and Post(I, O).
Description, Pre, and Post	Accept the pair only if both pass.

Retry syntax or consistency failures up to five attempts.

Functional Specification Enhancement

```
# Partition of sub-functionality 1
Description: Inputs that are valid and ≥ 80 should return grade 'A'.
Precondition: s >= 80 && s <= 100
Postcondition: return == 'A'
# Partition of sub-functionality 2
Description: Valid inputs between 60 and 79 should return grade 'B'.
Precondition: s >= 60 && s < 80
Postcondition: result == 'B'
# Partition of sub-functionality 3
Description: Inputs that are valid and less than 60 should return grade 'E'.
Precondition: s >= 0 && s < 60
Postcondition: return == 'E'
# Partition for sub-functionality 4
Description: Any input outside [0, 100] should be treated as invalid.
Precondition: s < 0 || s > 100
Postcondition: Throws an IllegalArgumentException.
```

```
# Partition of sub-functionality 1
Description: Inputs that are valid and ≥ 80 should return grade 'A'.
Precondition: s >= 80 && s <= 100
Postcondition: return == 'A'
Input: s = 100, Output: return = 'A'
# Partition of sub-functionality 3
Description: Inputs that are valid and less than 60 should return grade 'E'.
Precondition: s >= 0 && s < 60
Postcondition: return == 'E'
Input: s = 0, Output: return = 'E'
# Partition for sub-functionality 4
Description: Any input outside [0, 100] should be treated as invalid.
Precondition: s < 0 || s > 100
Postcondition: Throws an IllegalArgumentException.
Input: s = -5, Output: throw new IllegalArgumentException
```

```
1  public char s2g(int s) {
2     if (s >= 80)
3        return 'A';   // path 1
4     else if (s >= 60)
5        return 'B';   // path 2
6     else
7        return 'E';   // path 3
8  }
```

Unmatched specs (Partition 4) reveal a requirement-implementation inconsistency.
FuSE's test ($s = -5 \Rightarrow$ IllegalArgumentException) exposes it as a defect.

Research Questions

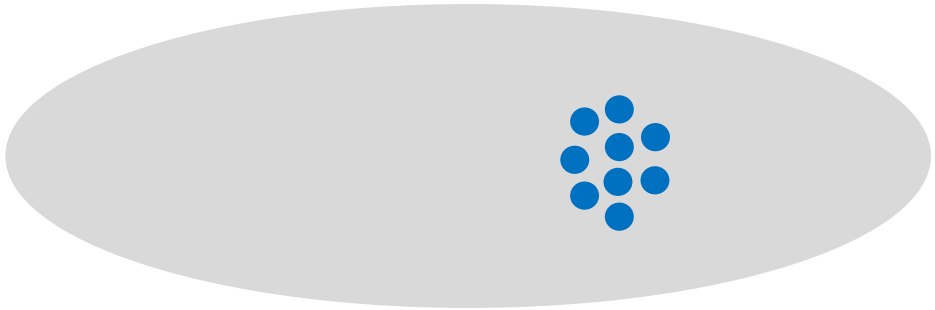
- RQ1: Effectiveness in Automated Generation.
- Can FuSE annotate program paths automatically?

- RQ2: Practical Utility.
- Does FuSE improve automated functional testing outcomes (functional coverage and confirmed functional defects)?

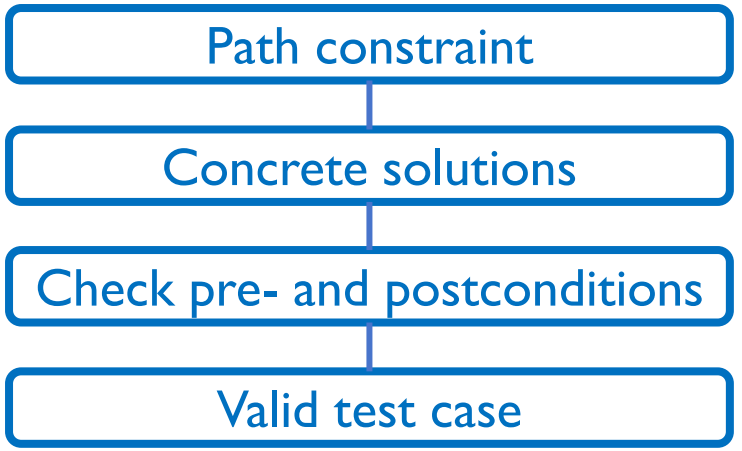
- RQ3: Quality of Automated Annotations.
- What is the quality of its functional annotations?

Under-approximation Validation

Executable validation instead of logical implication checking



Under-approximation: verified subset of the solution set



Satisfies both constraint and spec, follows the intended path

Not Adopted: logical implication check

- Requires a unifies formal representation
- Constraint is logic, spec may use library calls and helper functions
- Hard to encode uniformly across tools and languages

Accepted trade-off

- Only concrete solutions are checked
- Implication not guaranteed, may miss some issues
- Sufficient: goal is correct test cases, not exhaustive verification

Evaluation: manual inspection of a random sample of programs

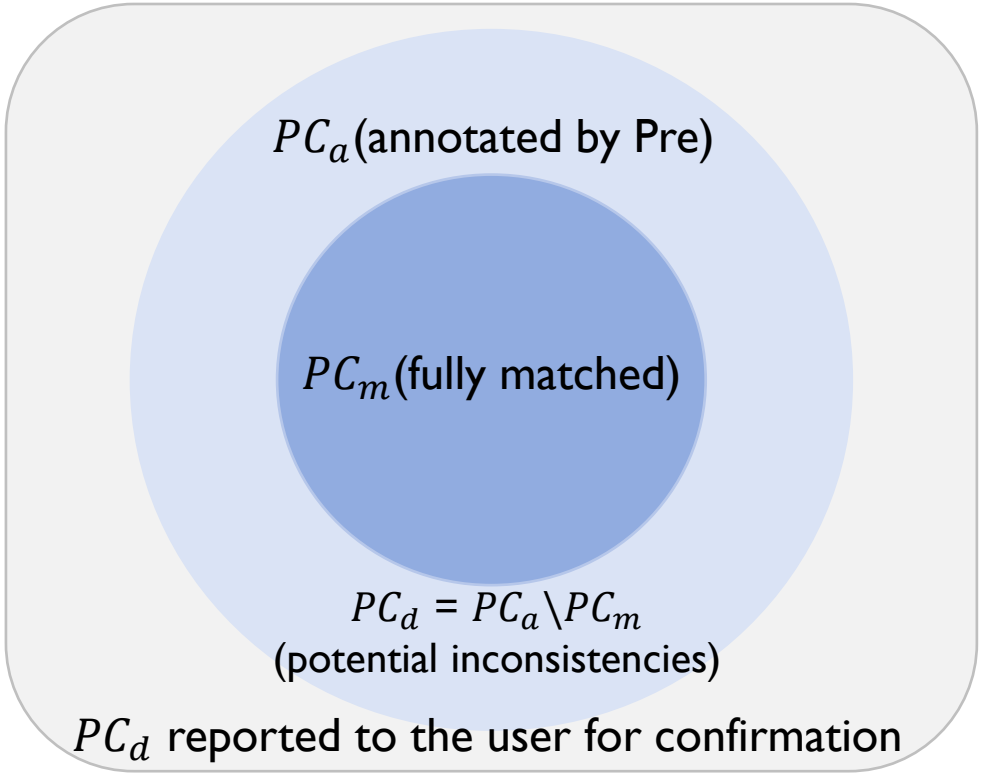
Evaluation Design

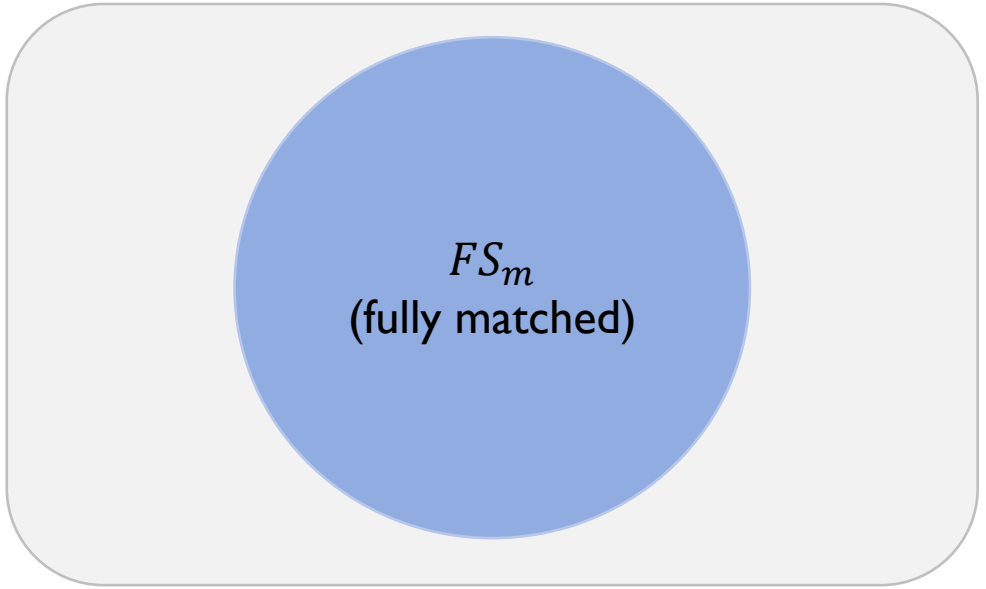
4	2	40	5×
languages C/C++/Java/Python	LLMs GPT-4o + Qwen3-Max	participants controlled user study	independent runs mean results reported

Baseline: UnitTestBot.

Subjects include HumanEval-X, logic_bombs, and real-world open-source programs.

Metrics





Initial: matching phase only
Final: matching + enhancement

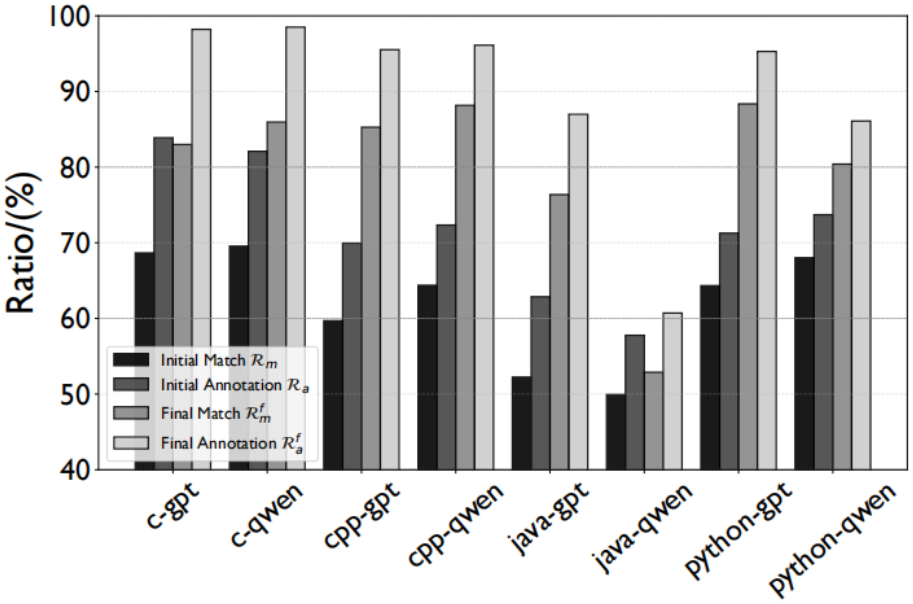
$$R_m = \frac{|PC_m|}{|PC|}$$
Path Match Rate(RQ1)
Initial R_m | Final R_{m}^f

$$R_a = \frac{|PC_a|}{|PC|}$$
Path Annotation Rate(RQ1)
Initial R_a | Final R_a^f

$$R_c = \frac{|FS_m|}{|FS|}$$
Functional Coverage(RQ2)
Initial R_c | Final R_c^f

RQ1: Path Annotation

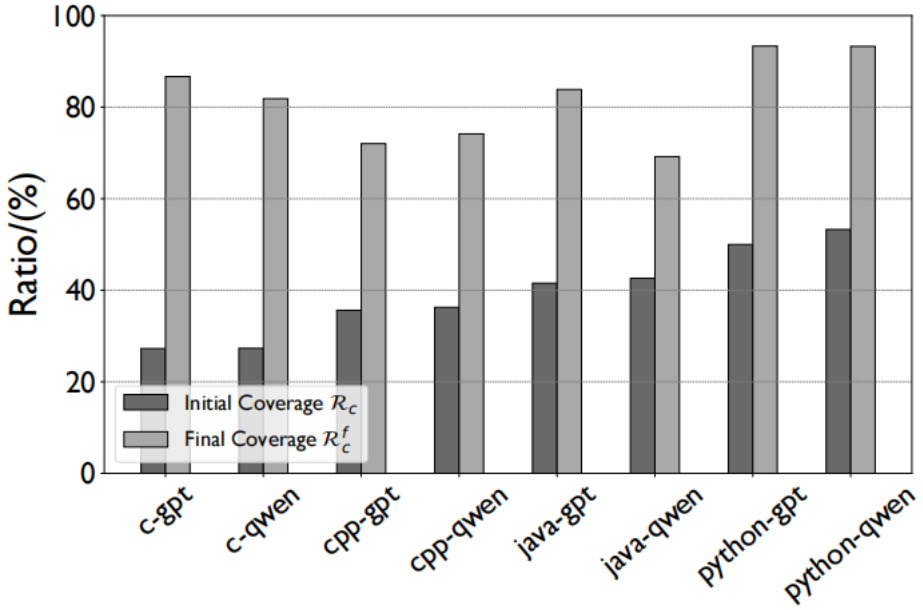
Lang. (No.)	$\sharp PC$	LLM	$\mathcal{R}_m$	$\mathcal{R}_a$	$\sharp \mathcal{PC}_d$	$\mathcal{R}_m^e$	$\mathcal{R}_m^f$	$\mathcal{R}_a^f$
C(32)	67	gpt	0.69	0.84	10	+0.14	0.83	0.98
		qwen	0.70	0.82	8	+0.16	0.86	0.99
C++(65)	201	gpt	0.60	0.70	21	+0.26	0.85	0.96
		qwen	0.64	0.72	16	+0.24	0.88	0.96
Java(164)	843	gpt	0.52	0.63	90	+0.24	0.76	0.87
		qwen	0.50	0.58	66	+0.03	0.53	0.61
Python(164)	932	gpt	0.64	0.71	65	+0.24	0.88	0.95
		qwen	0.68	0.74	53	+0.12	0.80	0.86



With GPT-4o, final path annotation rates reach 87-98% across four languages

RQ2: Functional Coverage

Lang. (No.)	♯PC	LLM	♯FS	$\mathcal{R}_p$	$\mathcal{R}_c$	$\mathcal{R}_c^e$	$\mathcal{R}_c^f$
C(32)	67	gpt	136	0.89	0.27	+0.60	0.87
		qwen	144	0.88	0.27	+0.55	0.82
C++(65)	201	gpt	344	0.85	0.36	+0.36	0.72
		qwen	369	0.89	0.36	+0.38	0.74
Java(164)	843	gpt	1028	0.93	0.42	+0.42	0.84
		qwen	850	0.85	0.43	+0.26	0.69
Python(164)	932	gpt	953	0.99	0.50	+0.43	0.93
		qwen	929	0.99	0.53	+0.40	0.93



FuSE significantly enhances functional testing, reaching 69%-93% coverage on benchmarks

RQ2: Defect Candidates

Root cause	C++	Java	Python
Functional Defect	10 (47.62%)	61 (67.03%)	31 (44.93%)
LLM Hallucination	13 (61.90%)	21 (23.08%)	36 (52.17%)
Ambiguous Requirement	2 (9.52%)	9 (9.89%)	11 (15.94%)
Unique Candidates	21	91	69

A candidate can involve more than one root cause

Manual inspection confirms 45% - 67% of reported candidates PC_d

RQ2: User Study

- 40 participants with development and testing experience
- 18 programs across C/C++, Java, and Python
- One hour per session, six programs per participant
- Compare *FuSE* annotations with *UnitTestBot* tests

50.83% ⟶ **66.67%**

+15.83%

Higher Functional coverage

22.50% ⟶ **44.17%**

+21.67%

More Confirmed functional defects

RQ3: Manual Inspection

Metric	C++	Java	Python
Consistency	92%	85%	95%
Correctness	84%	90%	90%
Completeness	63%	75%	75%

Manual inspection shows high consistency (up to 0.95) and correctness (up to 0.90).

RQ3: User Ratings

Attribute	Formal specification	Functional description
Correctness	4.28 / 5	4.58 / 5
Accuracy	4.22 / 5	4.38 / 5
Completeness	4.03 / 5	4.08 / 5

Usefulness: 3.95 / 5 Necessity: 3.95 / 5

Participants generally rate annotations favorably

Conclusion

Execution Without Comprehension

- Symbolic execution is powerful at exploring feasible paths.
- But it lacks comprehension of the program's *functional semantics* - both at the high level and at the path level.

$\alpha_s \geq 80 \wedge ret = 'A'$ // pc of path 1
$\alpha_s < 80 \wedge \alpha_s \geq 60 \wedge ret = 'B'$ // pc of path 2
$\alpha_s < 80 \wedge \alpha_s < 60 \wedge ret = 'E'$ // pc of path 3

? Gap

valid score in [60, 80) should return grade "B"

between path and functionality

3

Our Insight

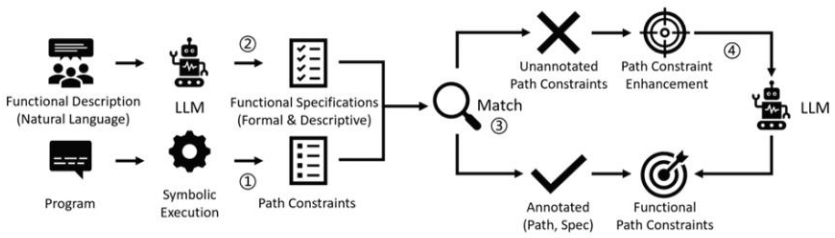
- Symbolic paths serve as structured units for functional analysis
- Automatically linking path constraints to their functional semantics

$\alpha_s \geq 80 \wedge ret = 'A'$	*Inputs that are valid and >= 80 return grade 'A'*
$\alpha_s < 80 \wedge \alpha_s \geq 60 \wedge ret = 'B'$	*Valid inputs between 60 and 79 return grade 'B'*
$\alpha_s < 80 \wedge \alpha_s < 60 \wedge ret = 'E'$	*Valid inputs less than 60 return grade 'E'*

comprehension of both code and natural language

6

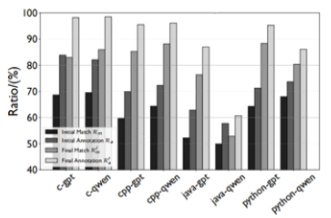
Workflow of FuSE



10

RQ1: Path Annotation

Lang. (No.)	$\sharp PC$	LLM	$\mathcal{R}_m$	$\mathcal{R}_a$	$\sharp PC_d$	$\mathcal{R}_m^e$	$\mathcal{R}_m^f$	$\mathcal{R}_a^f$
C(32)	67	gpt	0.69	0.84	10	+0.14	0.83	0.98
		qwen	0.70	0.82	8	+0.16	0.86	0.99
C++(65)	201	gpt	0.60	0.70	21	+0.26	0.85	0.96
		qwen	0.64	0.72	16	+0.24	0.88	0.96
Java(164)	843	gpt	0.52	0.63	90	+0.24	0.76	0.87
		qwen	0.50	0.58	66	+0.03	0.53	0.61
Python(164)	932	gpt	0.64	0.71	65	+0.24	0.88	0.95
		qwen	0.68	0.74	53	+0.12	0.80	0.86



With GPT-4o, final path annotation rates reach 87-98% across four languages

26

ASE 2026 Mon 12 - Fri 16 October 2026 Munich, Germany

Thank you!
Q&A

Artifact: https://doi.org/10.5281/zenodo.22053988