SpecGen: Automated Generation of Formal Program Specifications via Large Language Models

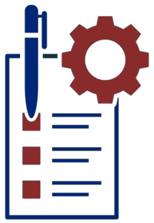
Lezhi Ma, Shangqing Liu, **Yi Li**, Xiaofei Xie, Lei Bu

Nanjing University (NJU)
Nanyang Technological University (NTU)
Singapore Management University (SMU)

April 30, 2025

Formal verification is promising, but …

- Formal verification is good for software quality assurance
  - Better than testing
    - New test cases must be created as code grows
    - Verification provides guarantees for all execution paths
- So, who formulates specifications?
  - Programmers? Probably not
  - Why they won't:
    - Too busy; yet another language to learn?
    - Specifications aren't cool
  - Software verification will be widely applied only if the cost of formulating specifications is reduced



Formal Specifications

Specifications written in JML

```java
public class LinearSearch {
    //@ invariant location >= -1
    private static /*@ spec_public*/ int location = -1;

     //@ requires array != null;
     //@ ensures \result == -1 <==>  (\forall int i; 0 <= i && i < array.length; array[i] != search);
     //@ ensures 0 <= \result && \result < array.length ==>  array[\result] == search;
     public static int linearSearch(int search, int array[]) {
        int c;
        //@ maintaining 0 <= c && c <= array.length;
        //@ maintaining (\forall int i; 0 <= i && i < c; array[i] != search);
        //@ decreases array.length - c;
        for (c = 0; c < array.length; c++) {
            if (array[c] == search) {
                location = c;
                break;
            }
        }
        if (c == array.length) {
            location = -1;
        }
        return location;
    }
}
```

Class Member Invariant

Precondition

Postcondition

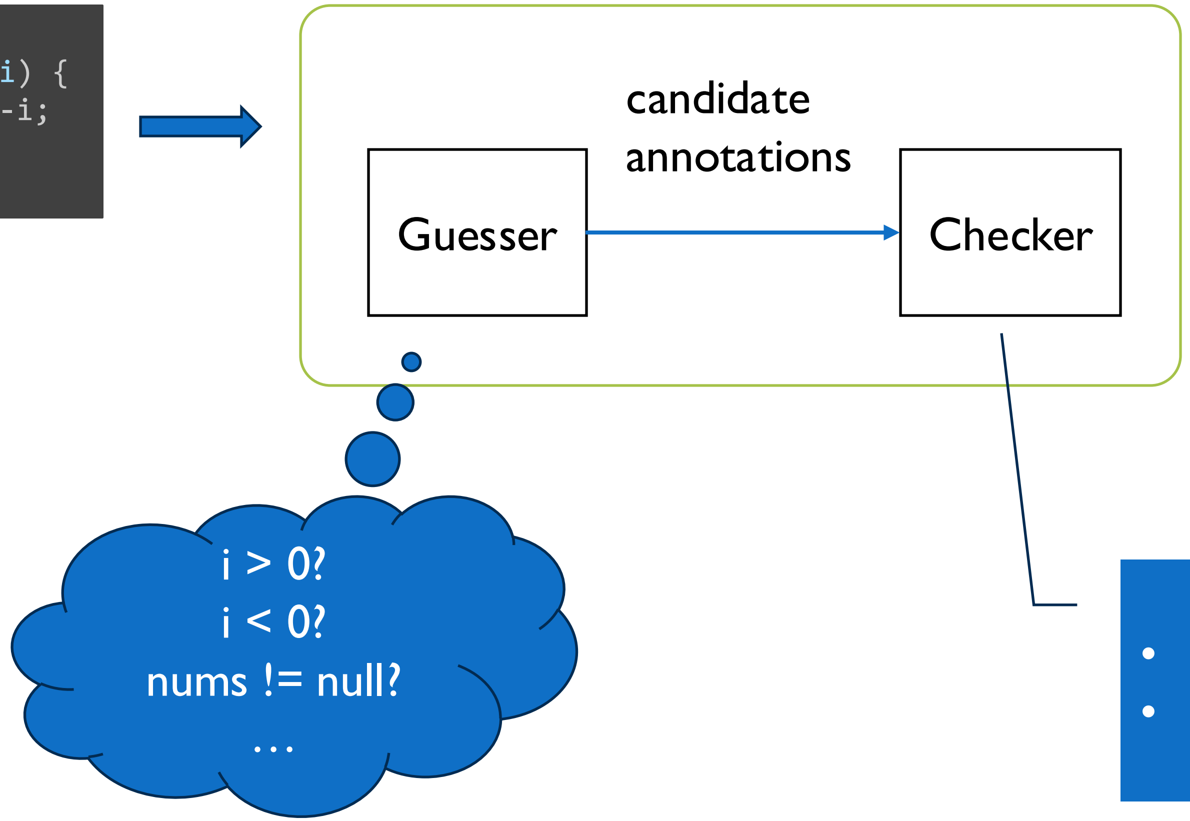
Loop Invariant

Related works: Houdini / Daikon

Invariant synthesis circa 2000 …

```
class C {
    int f (int i) {
        return -i;
    }
}
```

Program




```
class C {
    //@ requires i > 0
    //@ ensures i < 0
    int f (int i) {
        return -i;
    }
}
```

Annotated Program

- Houdini: verification
- Daikon: testing

Limitations of template-based methods

- Limited candidate invariant templates (within given budget)
  - Low success rate: most candidates are ruled out by verifier
  - Less semantic info: those survived are usually too trivial
    - e.g., "nums != null", "\result[i] >= 0"

```java
class TwoSum {
    public int[] twoSum(int[] nums, int target) {
        int n = nums.length;
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                if (nums[i] + nums[j] == target) {
                    return new int[]{i, j};
                }
            }
        }
        return new int[0];
    }
}
```

Ground Truth Specifications (Strong)

```java
//@ ensures \result.length == 2 ==> nums[\result[0]] + nums[\result[1]] == target;
//@ ensures \result.length == 0 ==> (\forall int i; 0 <= i && i < nums.length; (\forall int j; i + 1 <= j && j <
↪   nums.length; nums[i] + nums[j] != target));
```

```java
//@ requires nums != null;
//@ ensures \result != null;
```

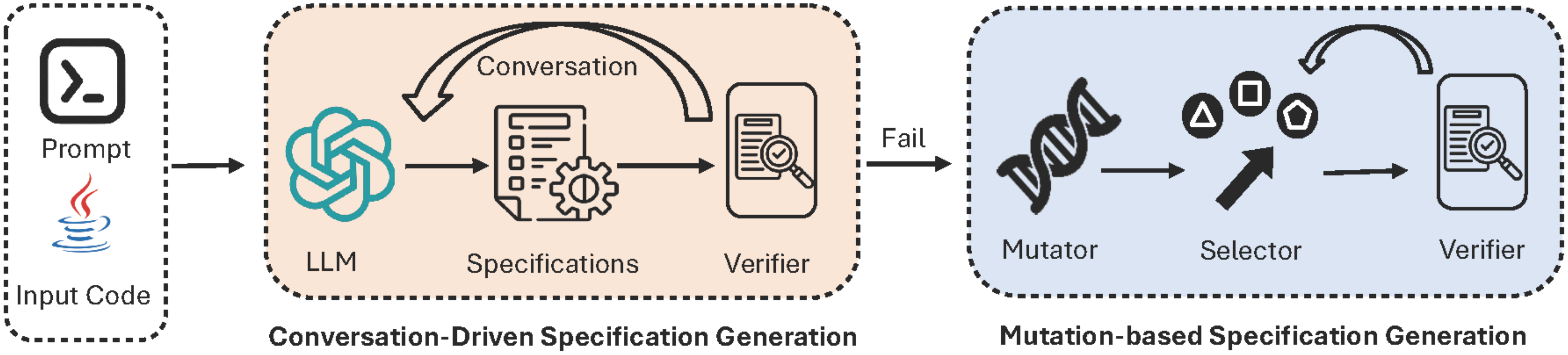
Specifications by Daikon

Not strong enough!

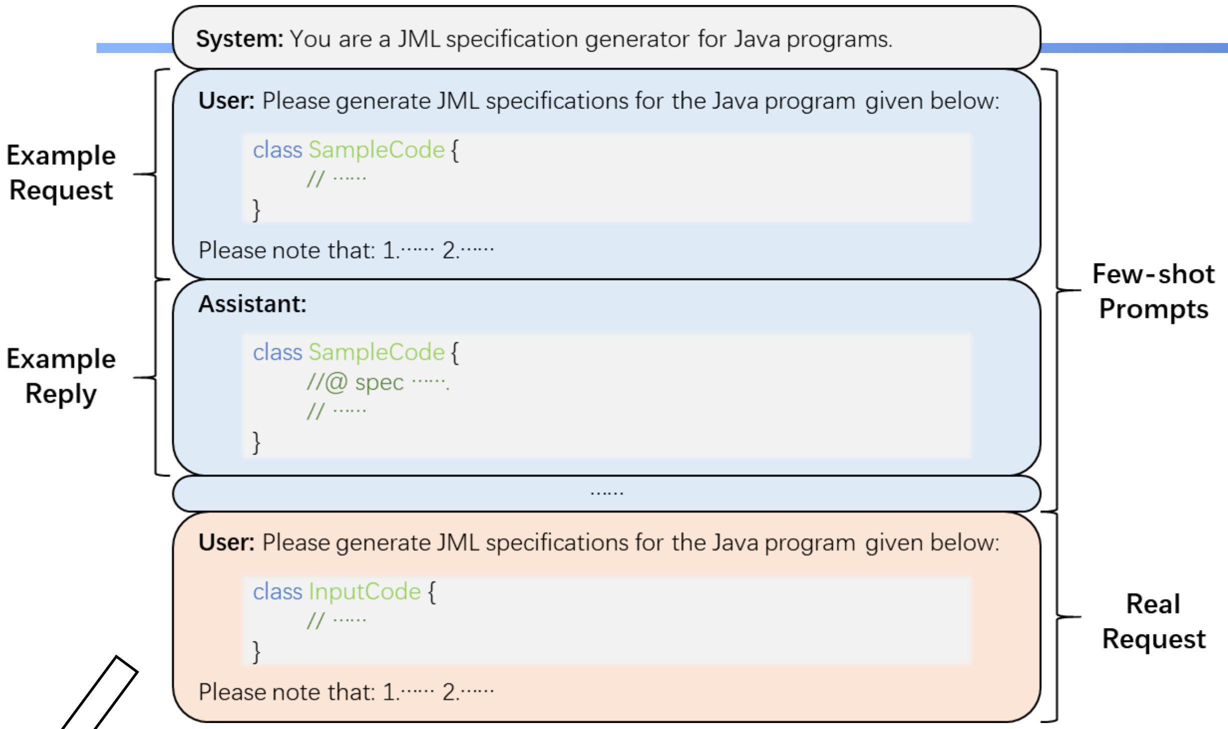
Generated by SpecGen

Specification generation powered by LLMs

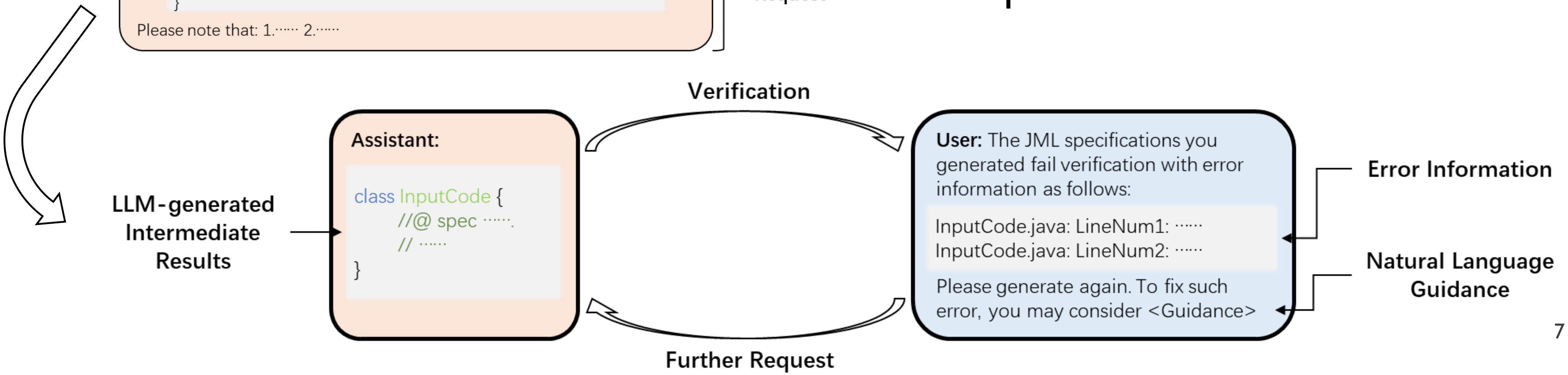
- SpecGen Overview
  - Phase 1: Conversation-driven Specification Generation
  - Phase 2: Mutation-based Specification Generation



1. Conversation-driven specification generation

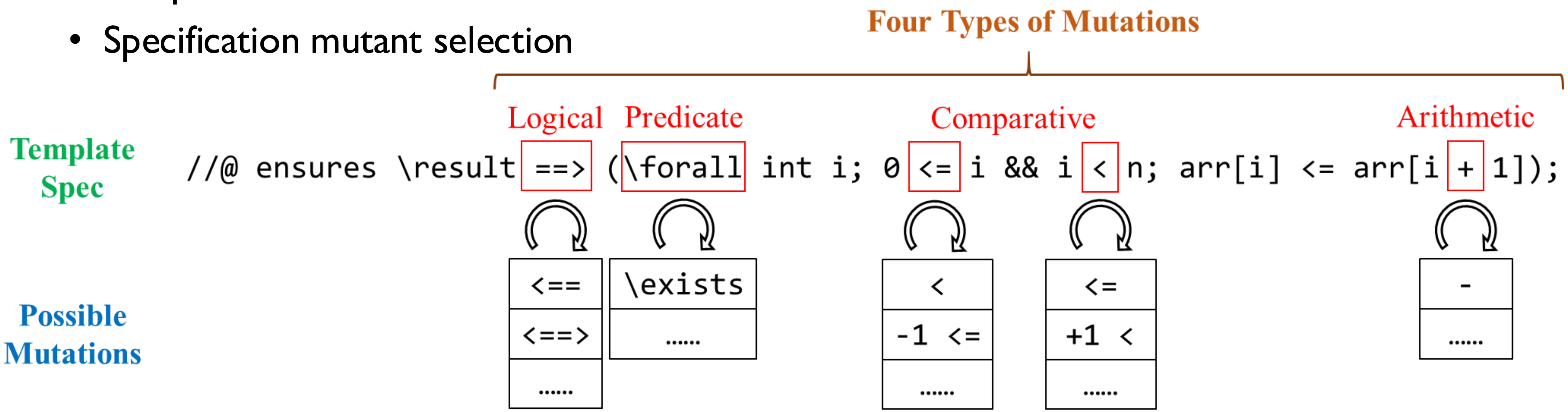


- Conversational generation
  - Few-shot Learning: proposed in pairs of pseudo requests and replies
  - Use verification error messages to guide LLM in generating correct specs



2. Mutation-based specification generation

- Conversation alone is not enough
  - Verification failure information is often too generic
  - LLMs may fail to grab the root cause when complex behaviors are involved
  - But they can often get close enough
- Increase the chance of success by introducing more diversity
  - Template mutation
  - Specification mutant selection



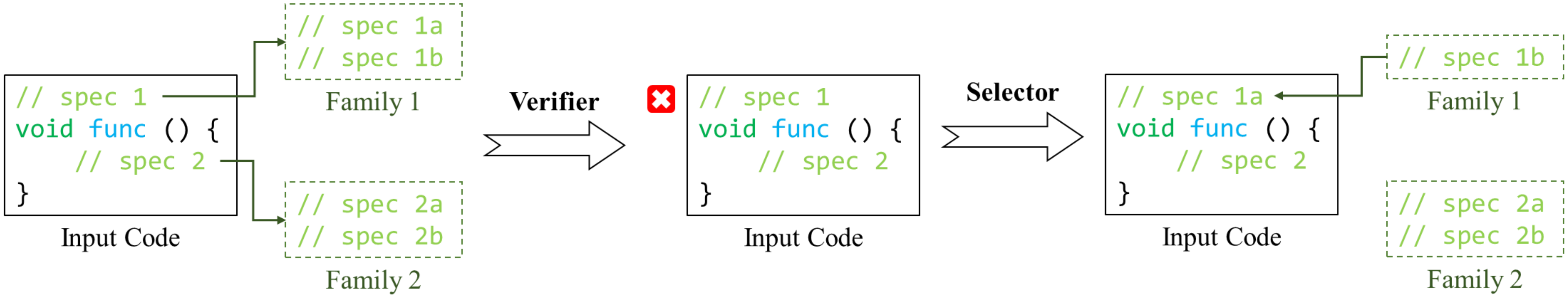
Mutant selection

- Heuristic-based selection
  - Arrange the operators into different priorities
  - Rate the candidates according to mutation operation types
  - Choose candidate according to the rating

$$\hat{e} = \arg\max_{e \in E_f} \sum_{m \in M} (times(m, e, e_t) \cdot weight(m))$$

E_f: a family of mutated specifications that come from the same template specification e_t

M : the set of all mutations



Evaluation

- Datasets
  - 265 SV-COMP Java classes (LOC 22.51)
  - 20 from Nilizadeh et al. (LOC 20.77)
  - 100 supplemental samples prepared by us
  - 50 Defect4J programs
- Implementation
  - LLM: GPT 3.5
  - Verifier: OpenJML
- Metrics
  - Success rate
  - Success probability
  - Number of verifier calls
  - User quality rating

- Research Questions
  - Can our method outperform prior works (i.e., Daikon, Houdini, vanilla LLM)?
  - How do different mutation operators contribute to the results?
  - How do candidate selection strategies effect the efficiency of the method?
  - To what extend can the invariants capture the semantics of the input program?

Evaluation: RQ1

Does SpecGen outperform prior works?

- SpecGen successfully generates specs for most (279/385) programs
- SpecGen successfully generates specs for 14 programs that no other baseline does

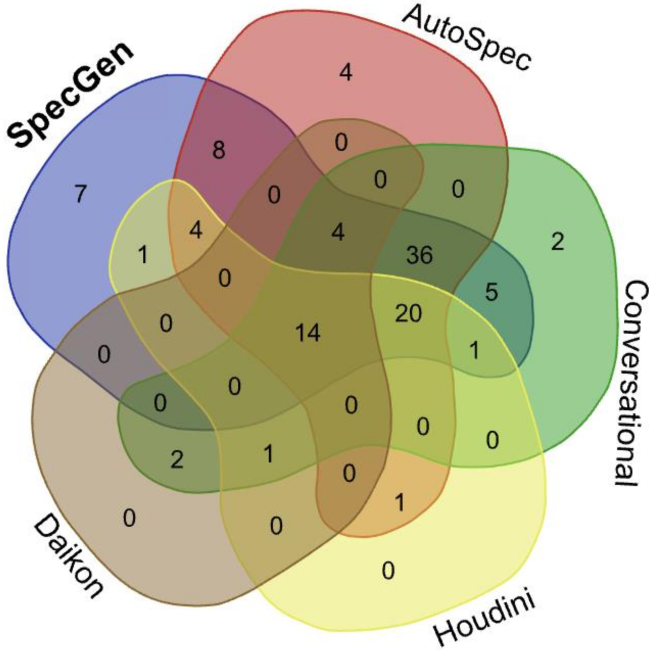


TABLE II: Number of programs that successfully pass the verifier and average success probability.

Approach		SV-COMP (265)		*SpecGenBench* Sequential (26)		Branched (23)		Single-path Loop (24)		Multi-path Loop (26)		Nested Loop (21)		Overall (385)	
		Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.
Daikon		51	-	10	-	10	-	0	-	1	-	0	-	72	-
Houdini		56	-	14	-	11	-	10	-	4	-	3	-	98	-
Few-shot LLM	0-shot	81	18.28%	23	74.19%	17	58.55%	5	7.08%	7	18.13%	2	3.33%	135	22.93%
	2-shot	83	18.79%	20	61.06%	17	53.91%	8	19.29%	13	26.58%	4	3.81%	145	23.48%
	4-shot	94	19.40%	23	73.85%	20	57.33%	10	23.40%	12	24.95%	5	6.24%	164	25.25%
Conversational		146	30.95%	23	82.49%	20	75.43%	12	27.02%	13	35.38%	4	9.20%	218	35.95%
AutoSpec		156	42.26%	24	85.38%	21	85.20%	22	57.00%	16	35.38%	8	12.38%	247	46.13%
SpecGen		**179**	**40.41%**	**24**	**92.31%**	20	**79.57%**	**23**	**73.75%**	**20**	**60.38%**	**13**	**36.55%**	**279**	**59.97%**

Evaluation: RQ1

Performance comparison on more practical Java programs, i.e., Defect4J

- Average LOC ~374.78 and cyclomatic complexity ~18.29
- Daikon is doing better, mainly for simple getter/setter methods
- SpecGen still performs reasonably well (38/50) and outperforms the baselines

TABLE VI: Performance on programs collected from 9 repositories in Defects4J.

Approaches	chart (7)		cli (5)		codec (4)		compress (6)		jackson (7)		jxpath (6)		lang (7)		math (4)		time (4)		Total (50)	
	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.	Num.	Prob.
Daikon	3	-	3	-	0	-	1	-	3	-	2	-	2	-	1	-	0	-	15	-
4-shot LLM	1	7.14%	2	7.33%	2	9.71%	3	11.67%	3	9.52%	1	2.78%	4	11.67%	3	17.50%	1	5.00%	20	8.97%
Conversational	4	47.62%	4	60.00%	3	41.67%	1	11.11%	2	19.05%	5	55.56%	3	28.57%	3	66.67%	3	41.67%	28	39.33%
SpecGen	**6**	**68.57%**	**4**	**68.00%**	**4**	**65.00%**	**4**	**36.67%**	**4**	**42.86%**	**5**	**63.33%**	**5**	**65.71%**	**3**	**45.00%**	**3**	**35.00%**	**38**	**55.20%**

Evaluation: RQ2

TABLE III: Effectiveness of different types of mutations.

Approach	*SpecGenBench* Sequential (26)	*SpecGenBench* Branched (23)	*SpecGenBench* Single-path (24)	*SpecGenBench* Multi-path (26)	*SpecGenBench* Nested (21)	SV-COMP (265)	Total (385)
w/o Predicative	24	20	20	19	9	167	259
w/o Logical	24	18	14	18	10	151	235
w/o Comparative	24	19	13	12	7	148	223
w/o Arithmetic	23	19	18	**21**	11	170	262
SpecGen	**24**	**20**	**23**	20	**13**	**179**	**279**

- The comparative mutation contributes the most to the performance
- The predicative and arithmetic mutations are less important
- When combining them together, SpecGen achieves the best performance

Evaluation: RQ3

How do candidate selection strategies affect the efficiency of the method?

- The heuristic selection strategy effectively improves the efficiency of SpecGen
- Especially useful to for programs that have more complex structures, e.g., loops

Strategy	SV-COMP	SpecGenBench					Total
		Sequential	Branched	Single-path	Multi-path	Nested	
Random	9.41	2.70	2.32	32.62	55.16	41.97	18.44
Heuristic	**8.91**	**2.59**	**2.16**	**24.21**	**43.58**	**34.99**	**15.51**

Average numbers of verifier calls in a single run under different specification selection strategies

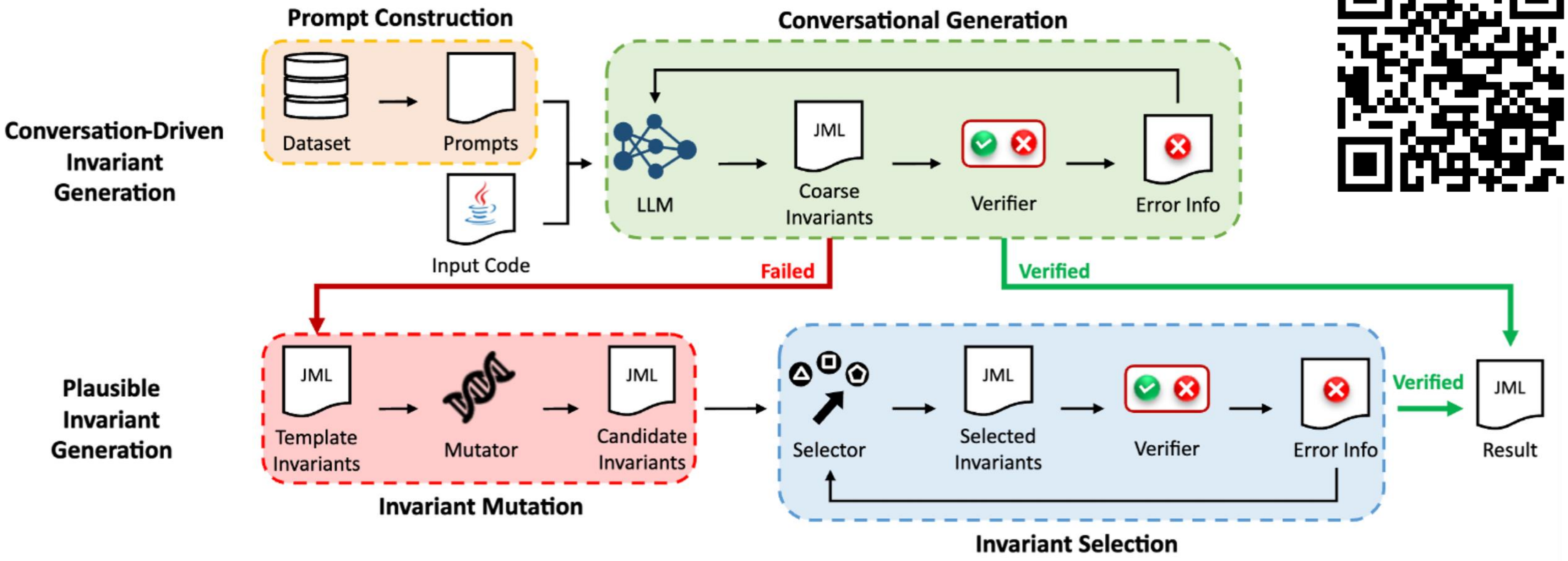
Evaluation: RQ4

Test case	Houdini	Daikon	SpecGen	Oracle
Absolute	3.50	3.36	**4.85**	5.00
AddLoop	2.40	1.33	**4.57**	5.00
Conjunction	4.50	3.50	**5.00**	5.00
ConvertTemperature	2.33	2.50	**5.00**	5.00
Disjunction	2.50	3.50	**5.00**	5.00
FizzBuzz	2.63	2.86	**5.00**	5.00
IsCommonFactor	2.00	4.13	**4.14**	4.71
IsPalindrome	1.83	1.17	**4.75**	5.00
IsSubsequence	2.43	1.13	**4.14**	4.00
IsSuffix	2.20	1.50	**4.33**	4.63
MulLoop	1.88	1.25	**3.33**	5.00
MySqrt	2.00	2.80	**3.75**	4.25
Perimeter	1.00	2.80	**4.78**	5.00
SmallestEvenMul	2.57	1.00	**4.50**	5.00
Swap	1.00	2.00	**5.00**	4.88
Average	2.32	2.32	**4.54**	4.83

- Quality of the generated specs
  - Evaluate through user study
  - 15 experienced Java developers
  - Rating scale: 1~5
  - Daikon and Houdini tend to generate trivial specs

Summary

yi_li@ntu.edu.sg
@liyistc



https://sites.google.com/view/specgen/home