

PROOFROVER: Assured Automatic Programming via LLMs

Martin Mirchev



Andreea Costea



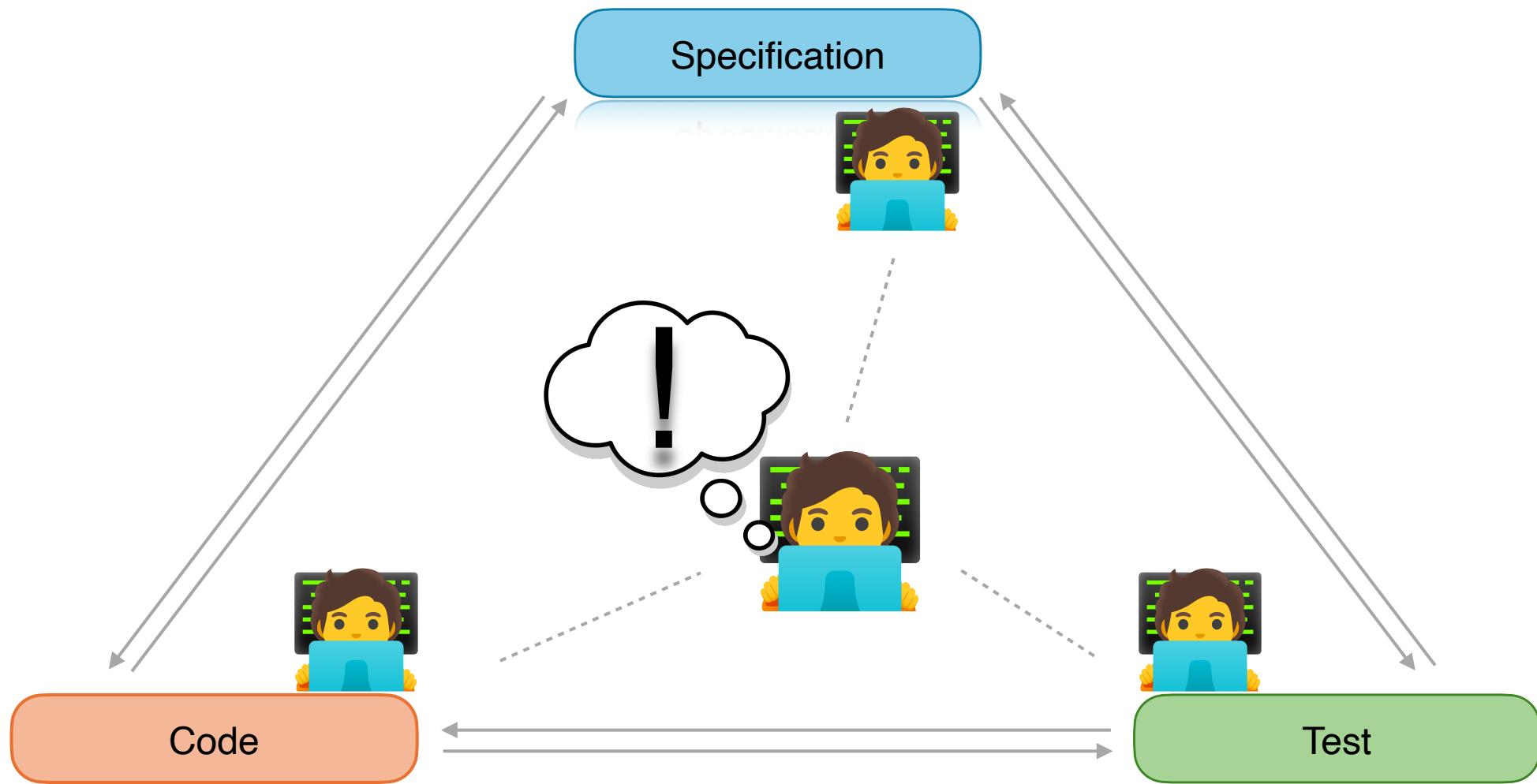
Abhishek Kr Singh

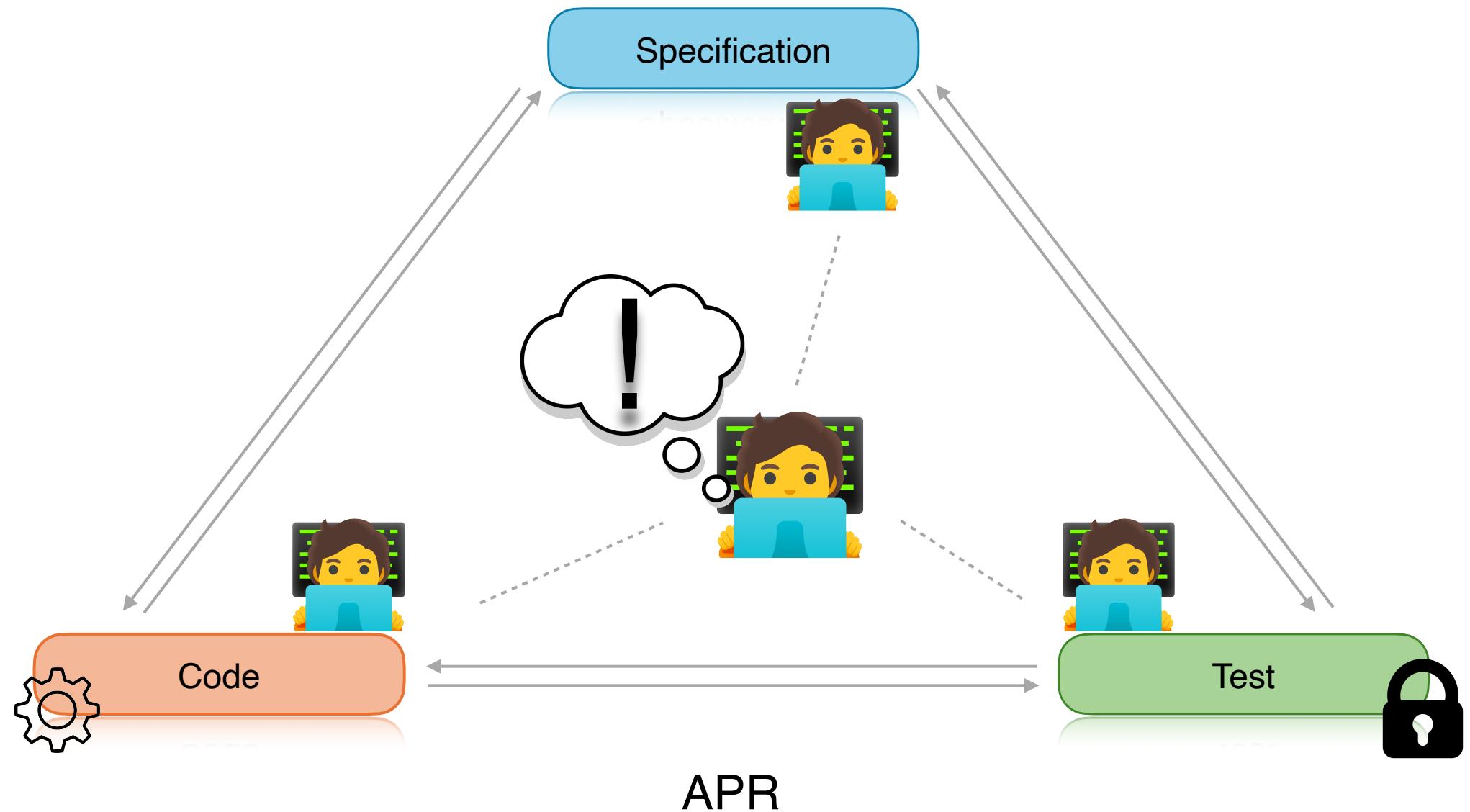


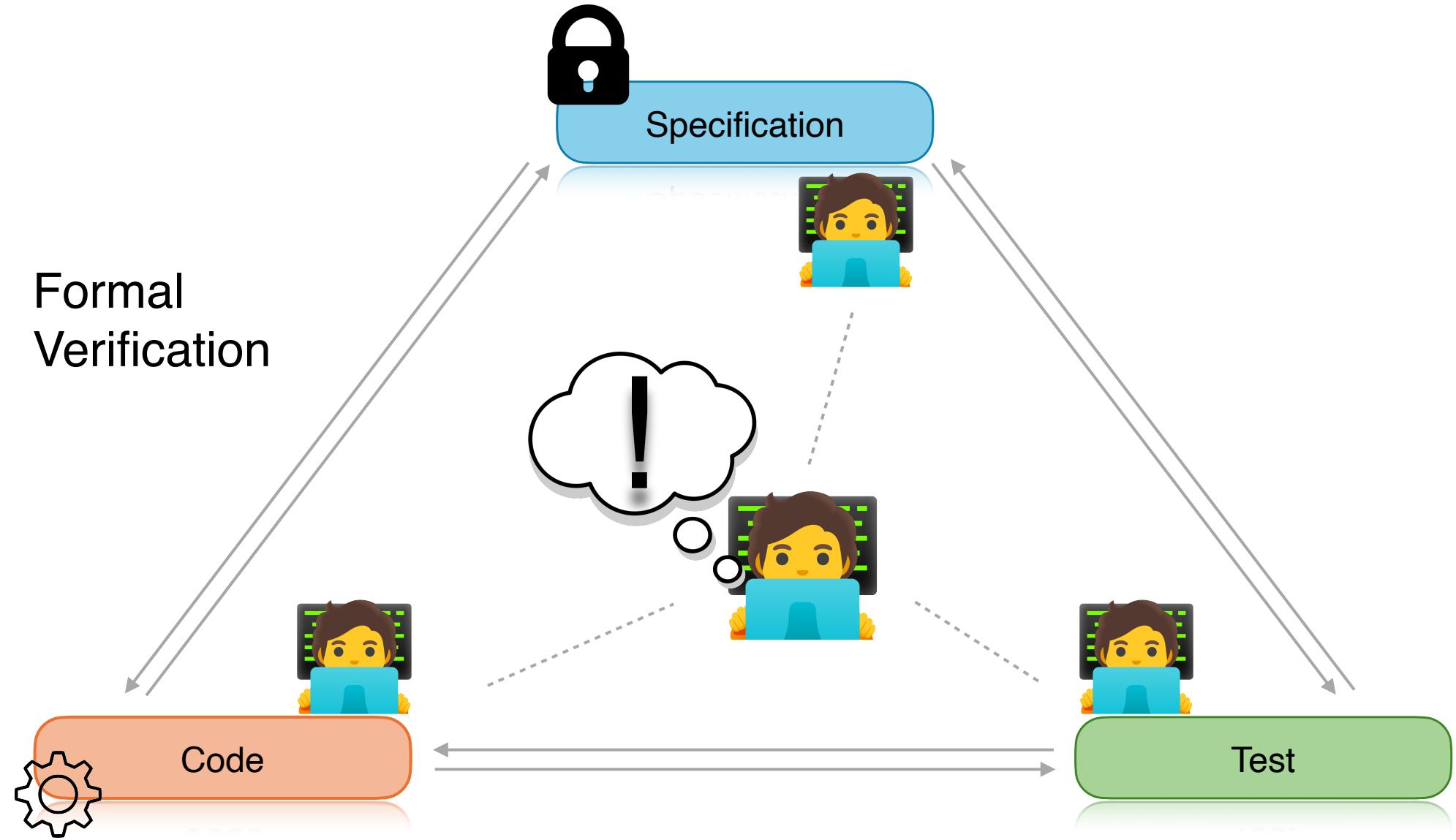
Abhik Roychoudhury

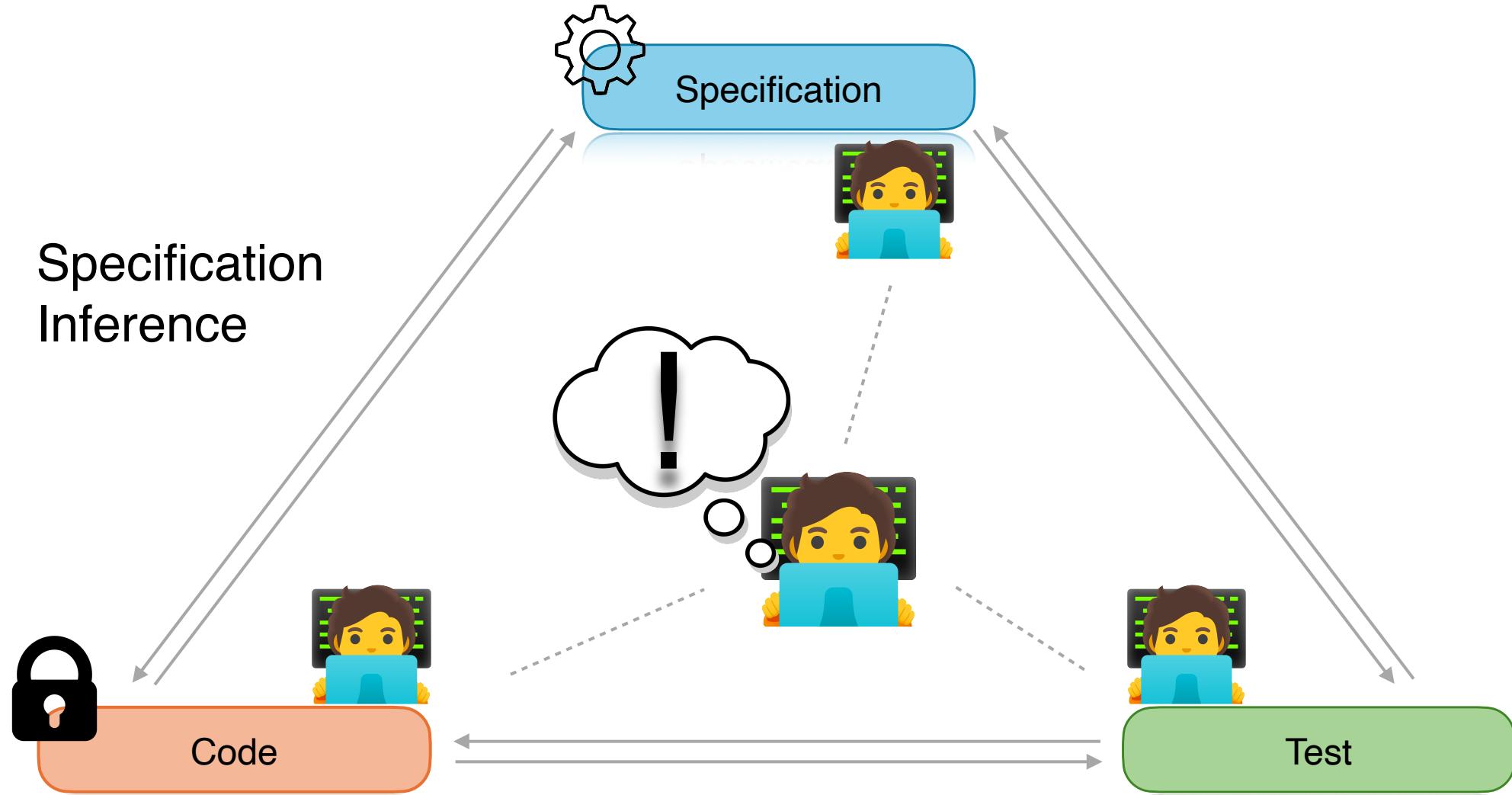


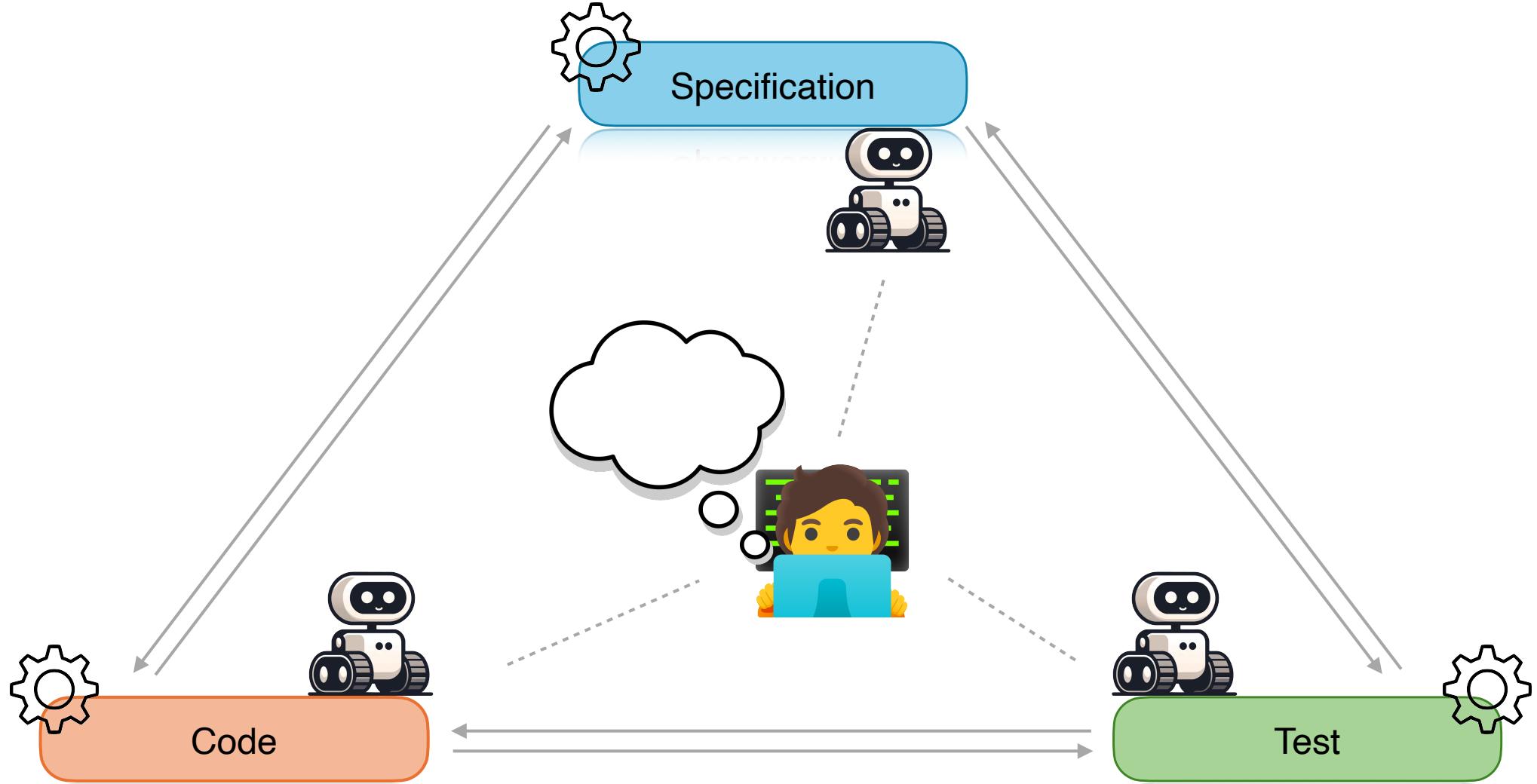


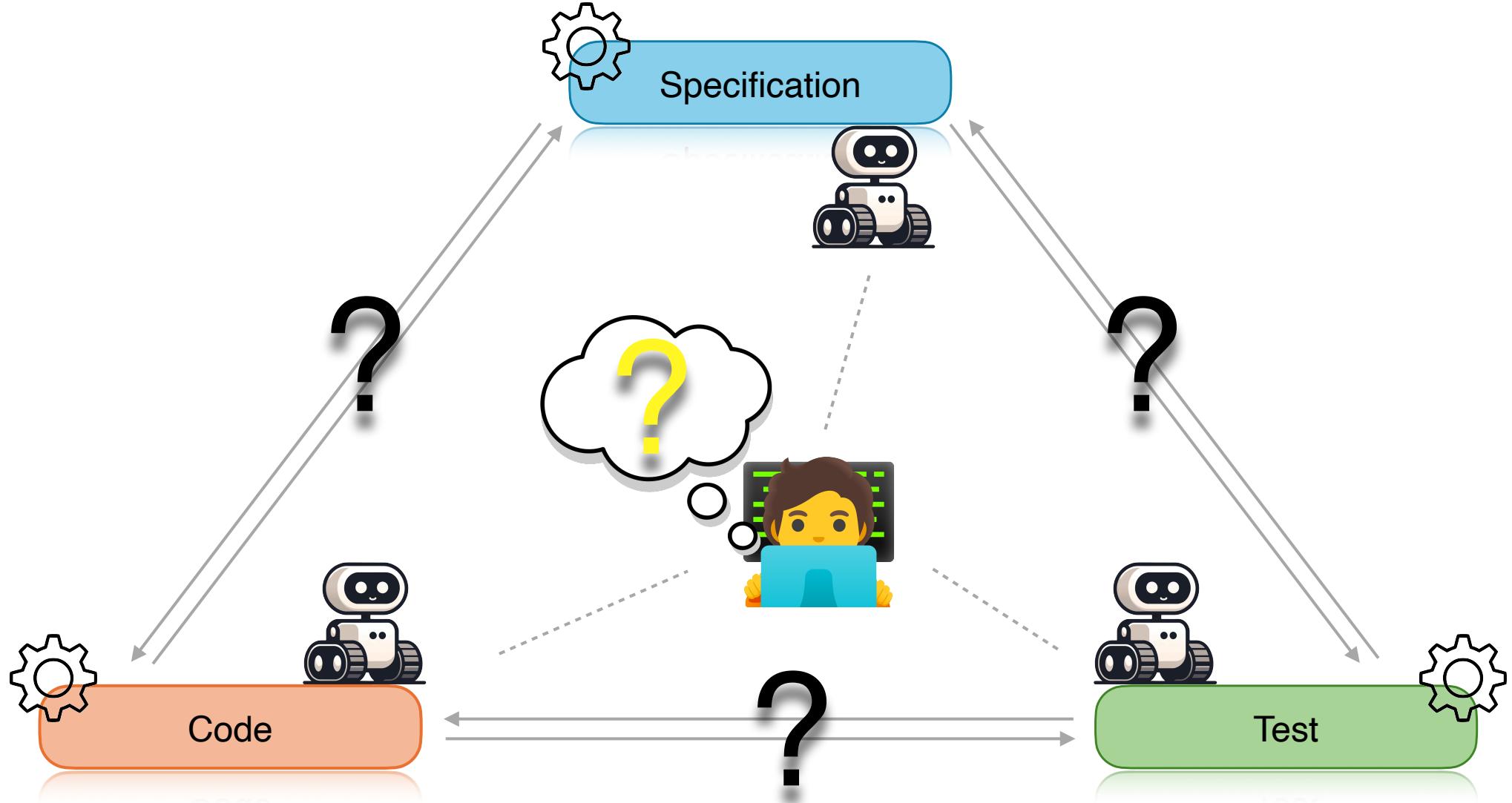












Align all the artefacts and in the process discover the user intent.

PROOFROVER: Assured Automatic Programming via LLMs

Scope: Dafny

```
1 method FindFirstOdd(arr: array<int>)
2     returns(odd: int)
3     requires arr != null
4     ensures arr[odd] % 2 != 0
5     ensures forall i :: 0 <= i < odd ==> arr[i] % 2 == 0
6 {
7     var found := false;
8     odd := -1;
9
10    for i := 0 to arr.Length
11        invariant 0 <= i <= arr.Length
12        invariant !found ==> odd == -1
13        invariant found ==>
14            0 <= odd < i && arr[odd] % 2 != 0
15        invariant forall j :: 0 <= j < i
16            ==> ((found ==> arr[j] % 2 != 0 || j == odd)
17                  && (!found ==> arr[j] % 2 == 0))
18    {
19        if arr[i] % 2 != 0 {
20            odd := i;
21            found := true;
22            break;
23        }
24    }
25 }
```

Verification-aware language
Syntactically close to Java, C#
Transpiles to many languages

LLM can generate specs [FSE 2024]

- 0% to 64% verified code on 178 problems.

[FSE 2024] Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble.
Towards AI-Assisted Synthesis of Verified Dafny Methods

Scope: Dafny

```
1 method FindFirstOdd(arr: array<int>)
2     returns(odd: int)
3     requires arr != null
4     ensures arr[odd] % 2 != 0
5     ensures forall i :: 0 <= i < odd ==> arr[i] % 2 == 0
6 {
7     var found := false;
8     odd := -1;
9
10    for i := 0 to arr.Length
11        invariant 0 <= i <= arr.Length
12        invariant !found ==> odd == -1
13        invariant found ==>
14            0 <= odd < i && arr[odd] % 2 != 0
15        invariant forall j :: 0 <= j < i
16            ==>((found ==> arr[j] % 2 != 0 || j == odd)
17                && (!found ==> arr[j] % 2 == 0))
18    {
19        if arr[i] % 2 != 0 {
20            odd := i;
21            found := true;
22            break;
23        }
24    }
25 }
```

Prompt

Generate a method that finds the first odd value in an array.

Verification Result

line 4: Error 1: index out of range.
line 5: Error 2: index out of range.
line 6: Error 3: A postcondition might not hold on this path.
line 4: This is the postcondition that might not hold.
No quick fixes available.

method AllEven()

```
{
    var x := new int[]{2, 2, 4};
    var s := FindFirstOdd(x);
    assert s == -2;
}
```

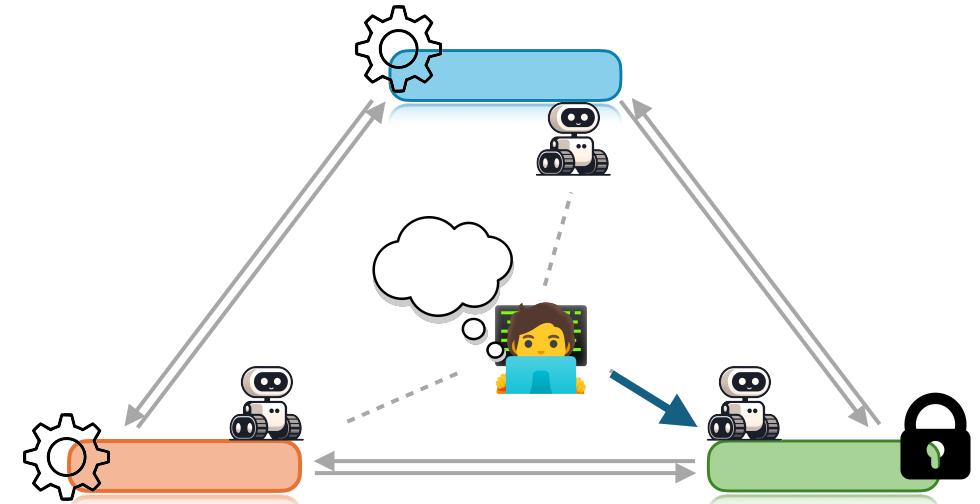
Multiple sources of intent:

- User Intent: Natural Language
- Code, Test, Spec

Ambiguous
Precise

Observation 1: Artifacts may contradict each other.

Observation 2: Artifacts almost align.



How do we reach conformance among these artifacts?

1. Discover the common intent.
2. Repair the inconsistencies.
3. Align specification with user intent.

A. Co-evolution

B. Intent discovery

A. Program-Proof Co-evolution

```
1  method FindFirstOdd(arr: array<int>)
2      returns(odd: int)
3      requires arr != null
4      ensures arr[odd] % 2 != 0
5      ensures forall i :: 0 <= i < odd ==> arr[i] % 2 == 0
6  {
7      var found := false;
8      odd := -1;
9
10     for i := 0 to arr.Length
11         invariant 0 <= i <= arr.Length
12         invariant !found ==> odd == -1
13         invariant found ==>
14             0 <= odd < i && arr[odd] % 2 != 0
15         invariant forall j :: 0 <= j < i
16             ==> ((found ==> arr[j] % 2 != 0 || j == odd)
17                 && (!found ==> arr[j] % 2 == 0))
18     {
19         if arr[i] % 2 != 0 {
20             odd := i;
21             found := true;
22             break;
23         }
24     }
25 }
```

A. Program-Proof Co-evolution

```
1 method FindFirstOdd(arr: array<int>)
2     returns(odd: int)
3     requires arr != null
4     ensures arr[odd] % 2 != 0
5     ensures forall i :: 0 <= i < odd ==> arr[i] % 2 == 0
6 {
7     var found := false;
8     odd := -1;
9
10    for i := 0 to arr.Length
11        invariant 0 <= i <= arr.Length
12        invariant !found ==> odd == -1
13        invariant found ==>
14            0 <= odd < i && arr[odd] % 2 != 0
15        invariant forall j :: 0 <= j < i
16            ==> ((found ==> arr[j] % 2 != 0 || j == odd)
17                && (!found ==> arr[j] % 2 == 0))
18    {
19        if arr[i] % 2 != 0 {
20            odd := i;
21            found := true;
22            break;
23        }
24    }
25 }
```

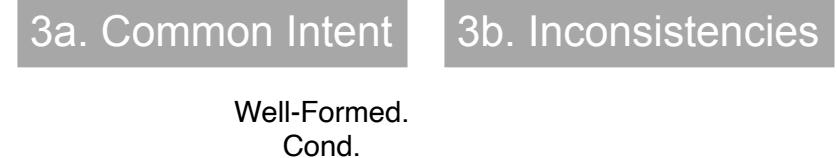
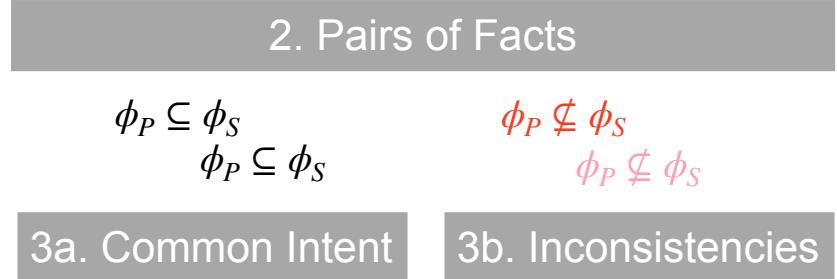
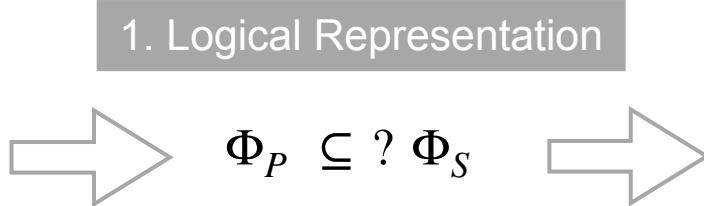
1. Logical Representation

$$\Phi_P \subseteq ?\Phi_S$$

Logical representation of the intent captured in the program and specification.

A. Program-Proof Co-evolution

```
1 method FindFirstOdd(arr: array<int>)
2     returns(odd: int)
3     requires arr != null
4     ensures arr[odd] % 2 != 0
5     ensures forall i :: 0 <= i < odd ==> arr[i] % 2 == 0
6 {
7     var found := false;
8     odd := -1;
9
10    for i := 0 to arr.Length
11        invariant 0 <= i <= arr.Length
12        invariant !found ==> odd == -1
13        invariant found ==>
14            0 <= odd < i && arr[odd] % 2 != 0
15        invariant forall j :: 0 <= j < i
16            ==> ((found ==> arr[j] % 2 != 0 || j == odd)
17                && (!found ==> arr[j] % 2 == 0))
18    {
19        if arr[i] % 2 != 0 {
20            odd := i;
21            found := true;
22            break;
23        }
24    }
25 }
```



Decompose the two sources of intents into granular facts.

A. Program-Proof Co-evolution

```
1 method FindFirstOdd(arr: array<int>)
2     returns(odd: int)
3     requires arr != null
4     ensures arr[odd] % 2 != 0
5     ensures forall i :: 0 <= i < odd ==> arr[i] % 2 == 0
6 {
7     var found := false;
8     odd := -1;
9
10    for i := 0 to arr.Length
11        invariant 0 <= i <= arr.Length
12        invariant !found ==> odd == -1
13        invariant found ==>
14            0 <= odd < i && arr[odd] % 2 != 0
15        invariant forall j :: 0 <= j < i
16            ==> ((found ==> arr[j] % 2 != 0 || j == odd)
17                && (!found ==> arr[j] % 2 == 0))
18    {
19        if arr[i] % 2 != 0 {
20            odd := i;
21            found := true;
22            break;
23        }
24    }
25 }
```

Distinguish between hard intent and soft intent.

Soft intent might still contain useful information. Don't discard it completely

1. Logical Representation

$$\Phi_P \subseteq ?\Phi_S$$

2. Pairs of Facts

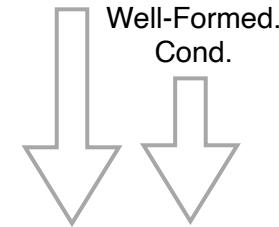
$$\phi_P \subseteq \phi_S$$

$$\phi_P \subseteq \phi_S$$

$$\phi_P \not\subseteq \phi_S$$

$$\phi_P \not\subseteq \phi_S$$

3a. Common Intent



3b. Inconsistencies



4a. Hard Intent



4b. Soft Intent



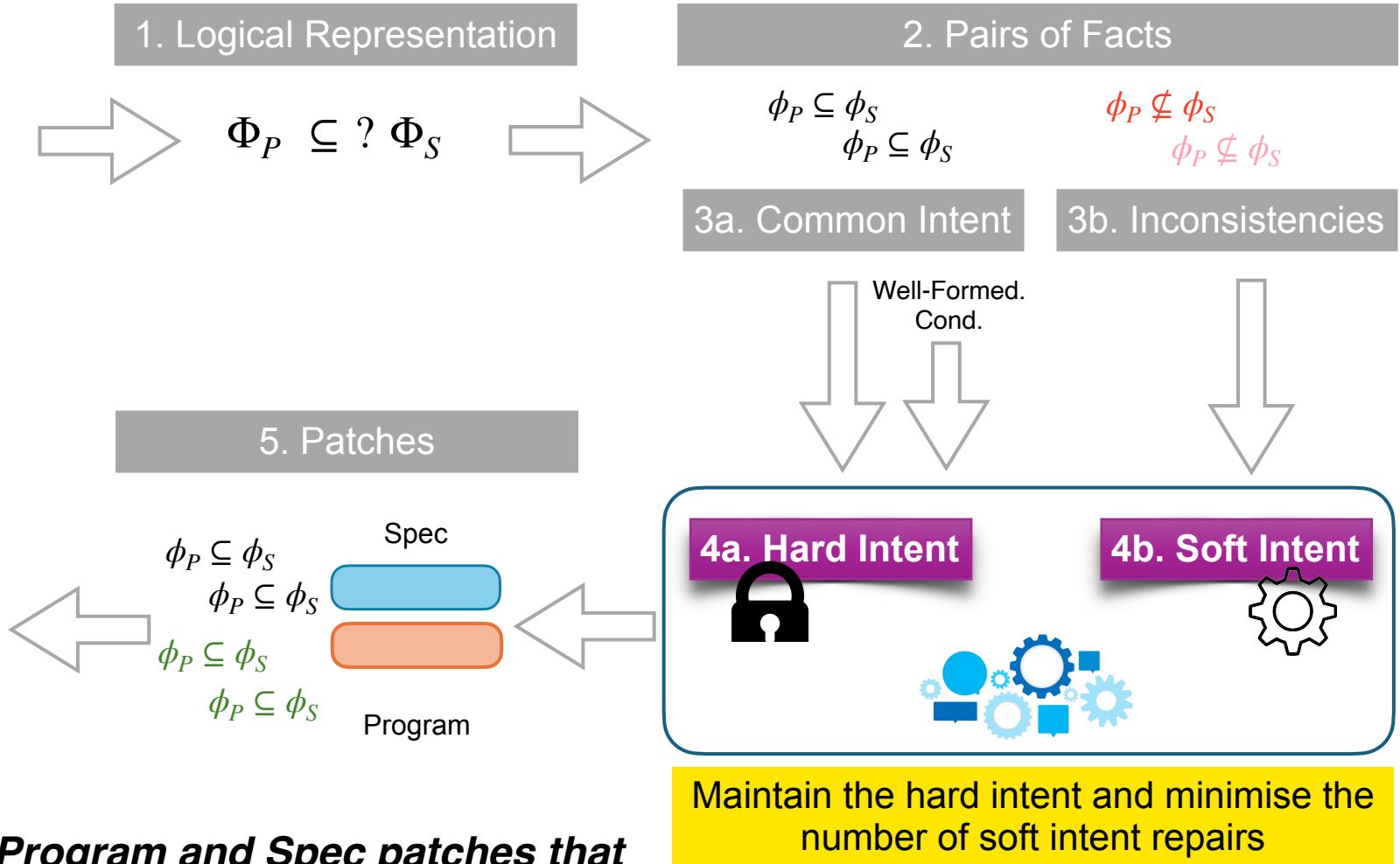
Maintain the hard intent and minimise the number of soft intent repairs

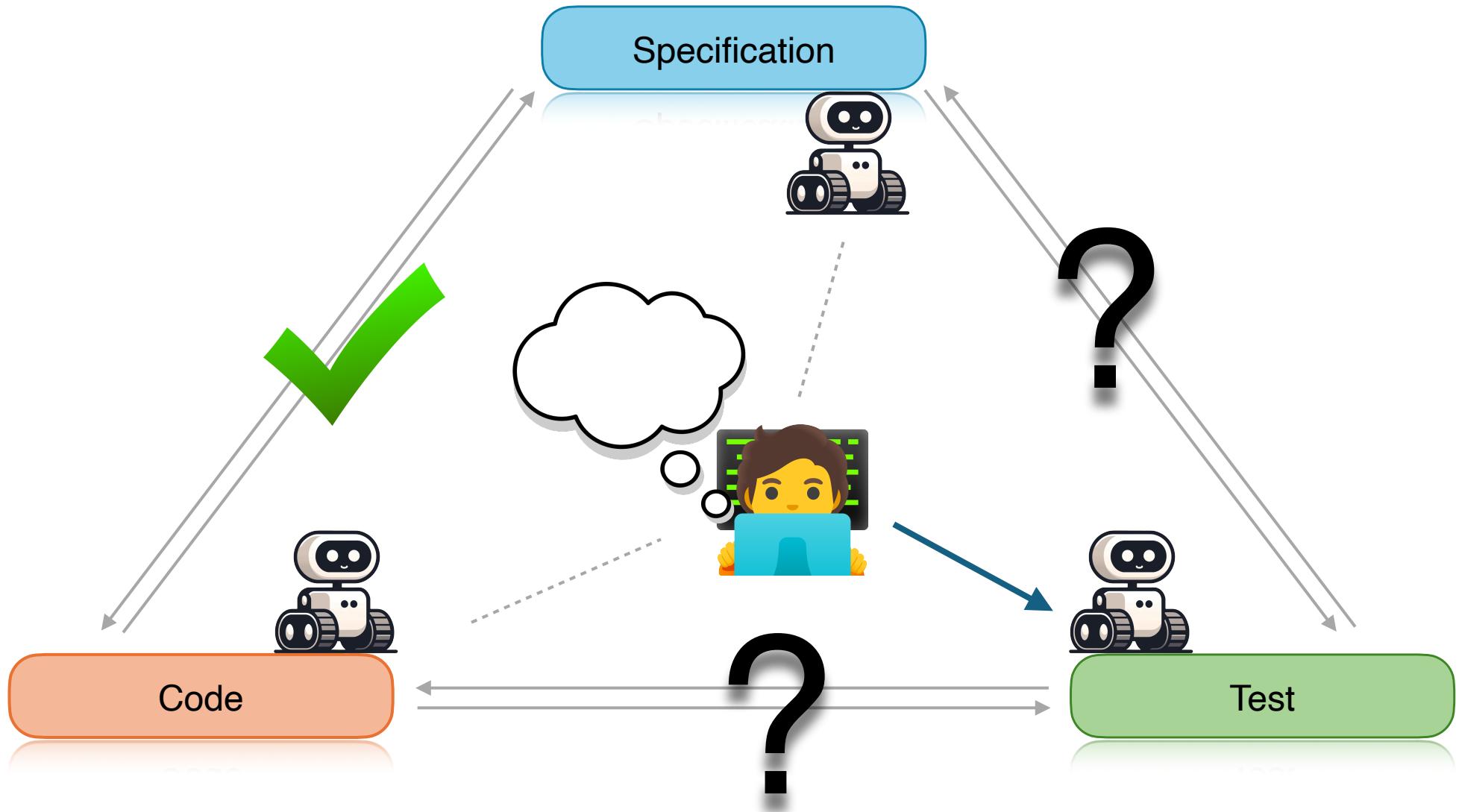
A. Program-Proof Co-evolution

```

1  method FindFirstOdd(arr: array<int>)
2      returns(odd: int)
3      requires arr != null
4      ensures arr[odd] % 2 != 0
5      ensures forall i :: 0 <= i < odd ==> arr[i] % 2 == 0
6  {
7      var found := false;
8      odd := -1;
9
10     for i := 0 to arr.Length
11         invariant 0 <= i <= arr.Length
12         invariant !found ==> odd == -1
13         invariant found ==>
14             0 <= odd < i && arr[odd] % 2 != 0
15         invariant forall j :: 0 <= j < i
16             ==> ((found ==> arr[j] % 2 != 0 || j == odd)
17                 && (!found ==> arr[j] % 2 == 0))
18
19     if arr[i] % 2 != 0 {
20         odd := i;
21         found := true;
22         break;
23     }
24 }

```





We reached Program-Spec conformance. What about User Intent?

B. Align Spec with User Intent

```
method OddInArray()  
{  
    var x := new int[]{2,3,4};  
    var s := FindFirstOdd(x);  
    assert s >= 0;  
}
```



```
method OddInArray(int[] x) returns (s:int)  
    requires x == new int[]{2,3,4}  
    ensures s >= 0  
{  
    s := FindFirstOdd(x);  
}
```

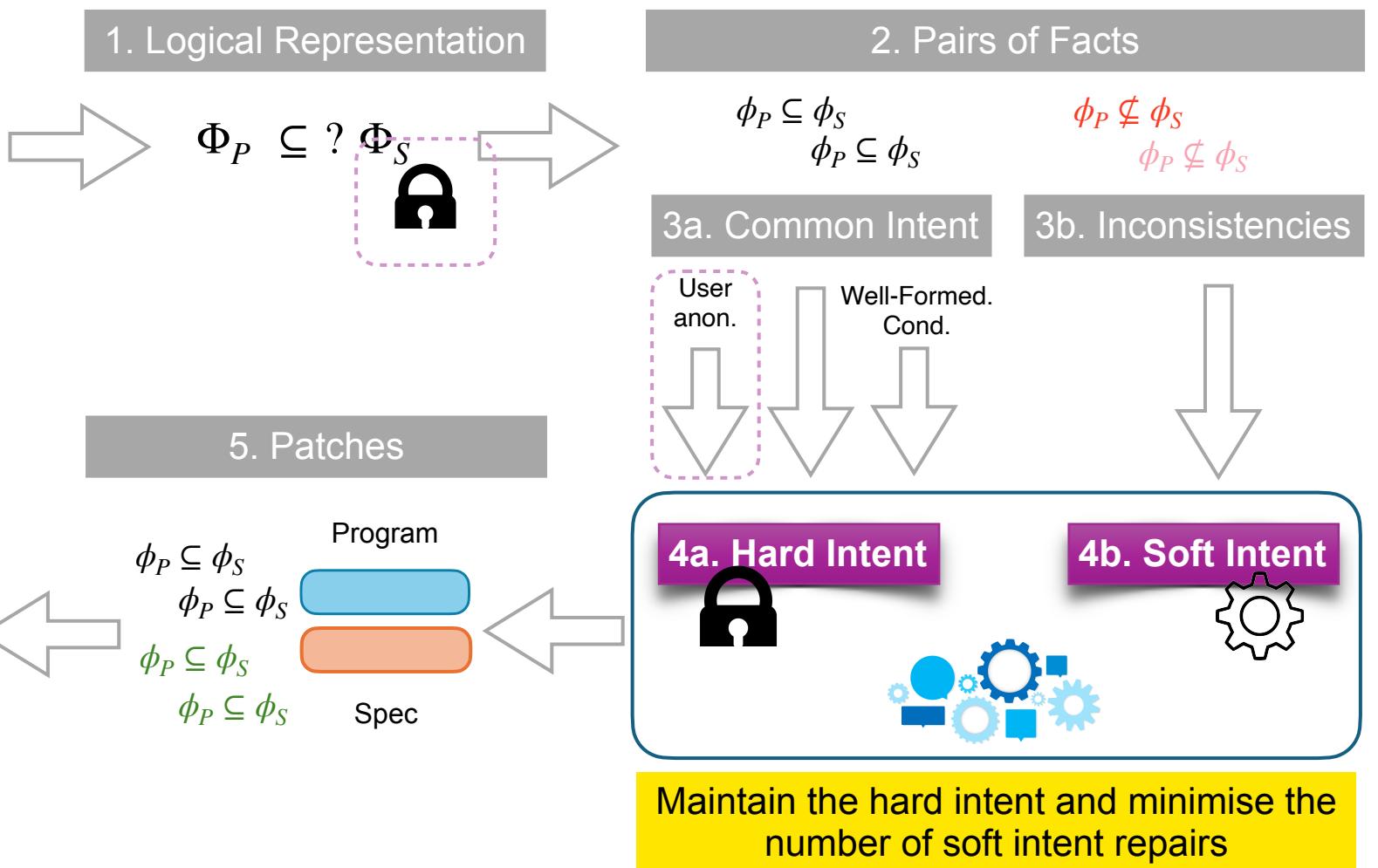


*Interact with user via tests.
Insight: treat testing as a verification problem ...*

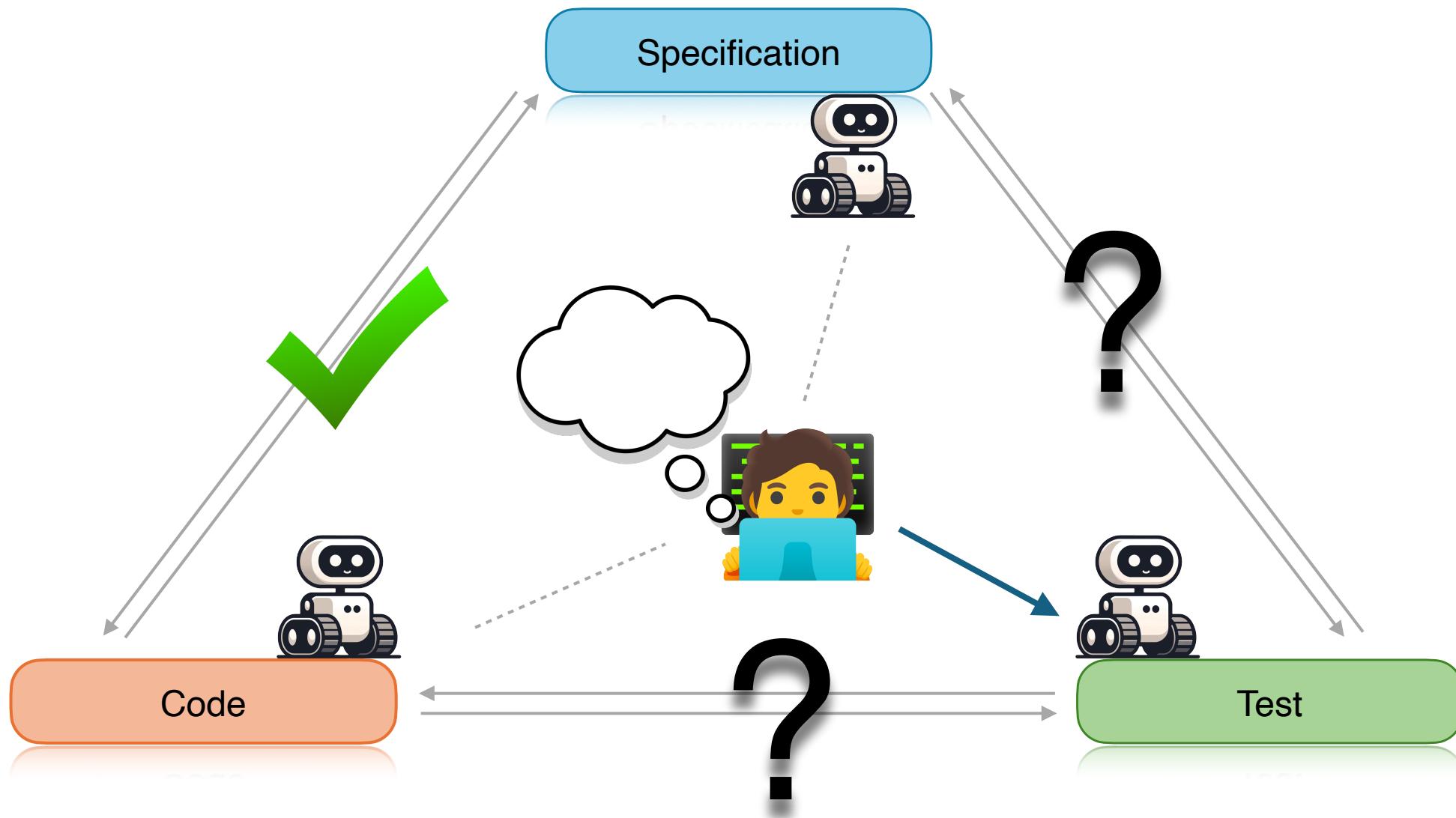
```

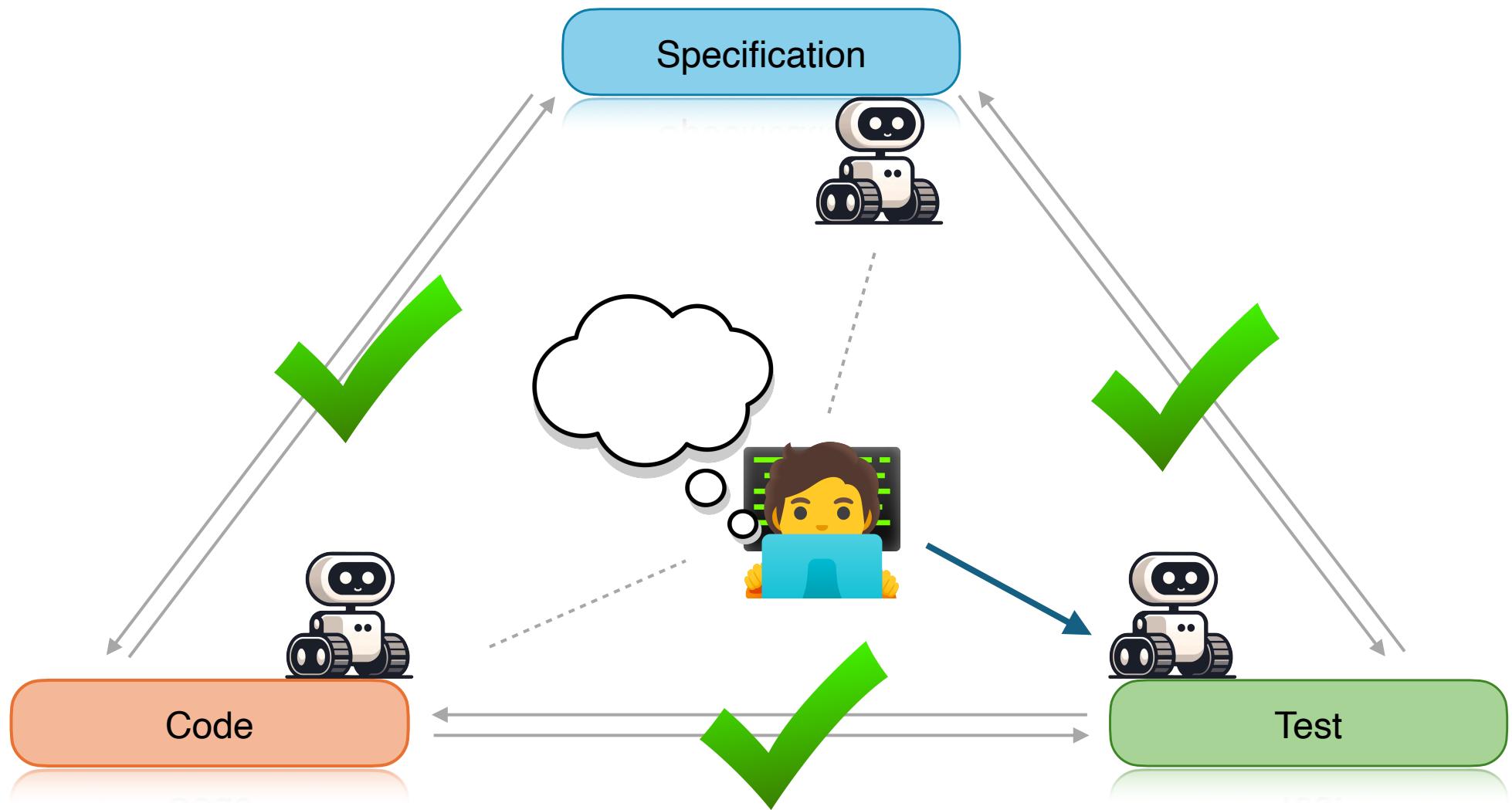
method OddInArray(int[] x) returns (s:int)
  requires x == new int[]{2,3,4}
  ensures s >= 0
{
  s := FindFirstOdd(x);
}

```

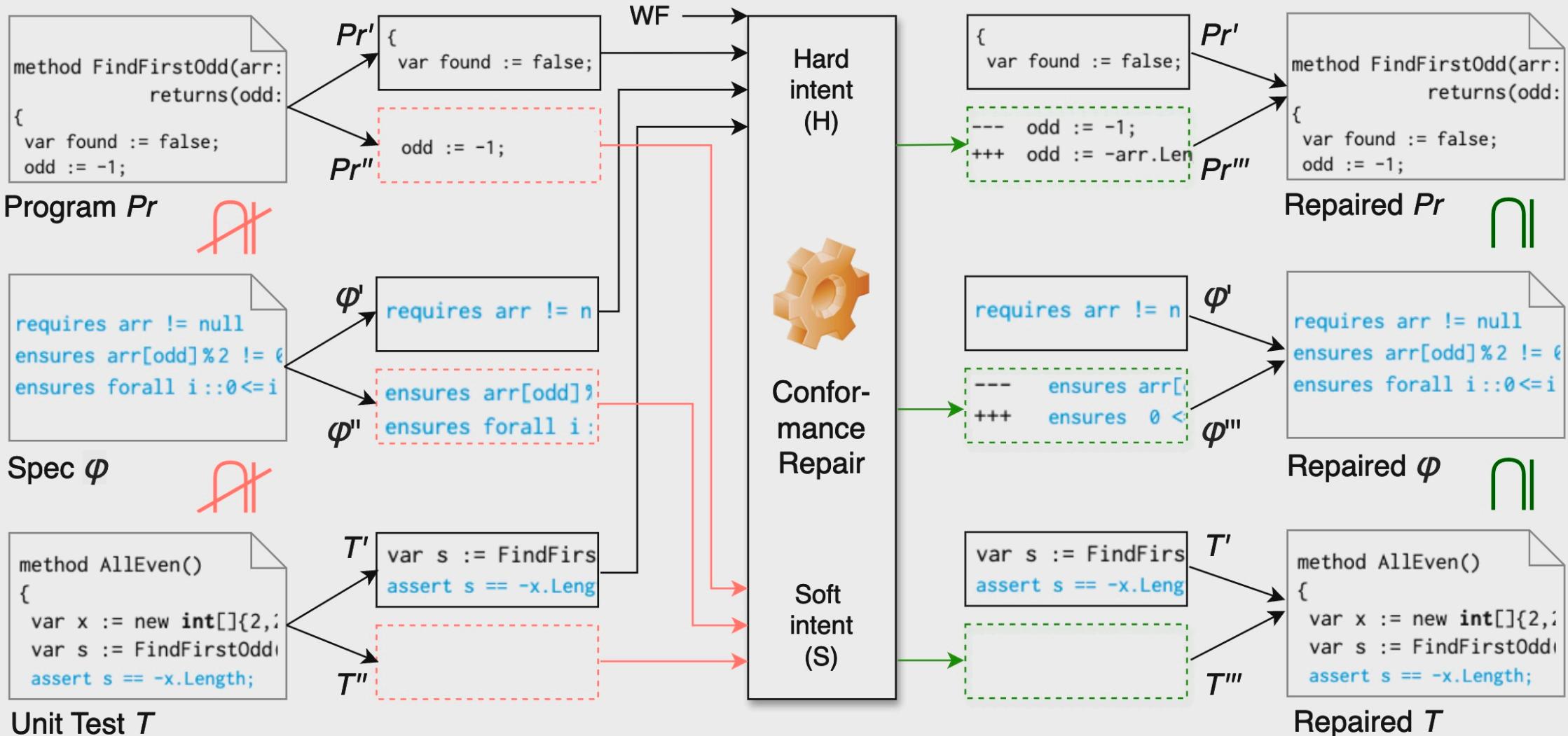



... and launch another program-proof co-evolution campaign.



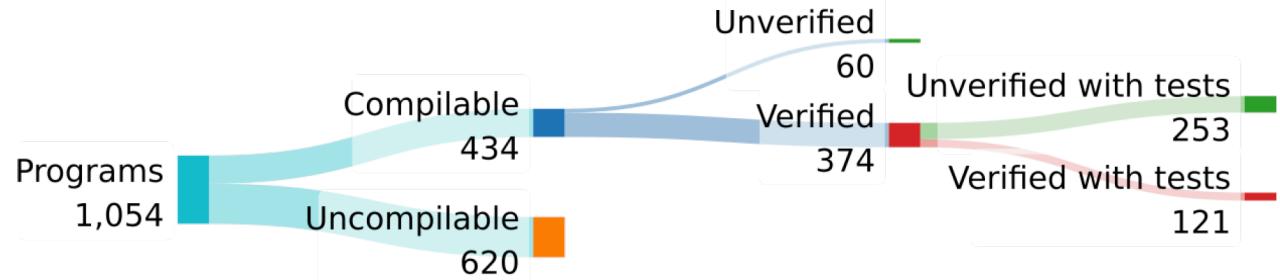


To sum up:



Evaluation and Results

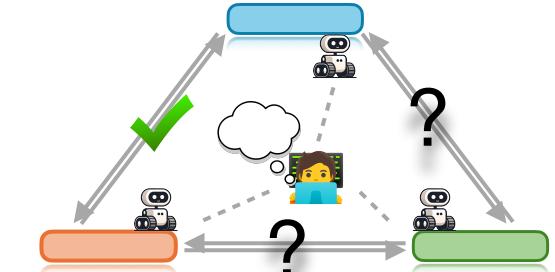
MBPP-DFY [FSE 2024]:



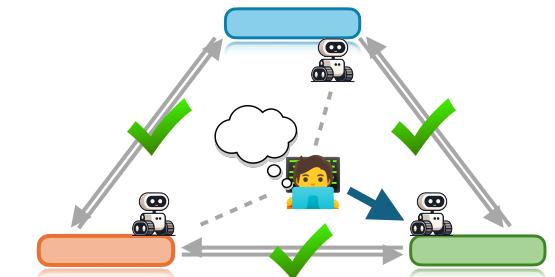
Setup:

- Fully static approach
- GPT-4o, Sonnet-3.5
 - Temperature: 0.3, 0.7, 1.0
- Baselines:
 - Naive Repair - with verification failures info
 - Chain-of-Thought - same prompt as ProofRover, bar Hard/Soft intent
- Metrics: conformance, quality of specs

For non-conforming code, ProofRover **aligns about 73% and 28%** more programs with specifications than the Naive Repair baseline and the Chain-of-Thought repair, resp.



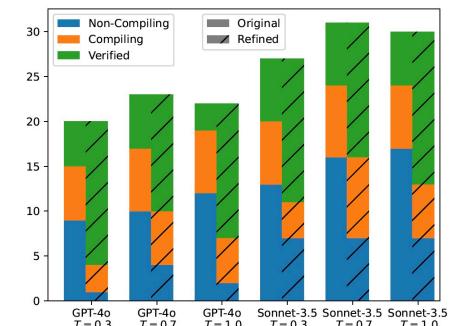
For verified code, ProofRover performs similar to the Naive Repair and the Chain-of-Thought repair.

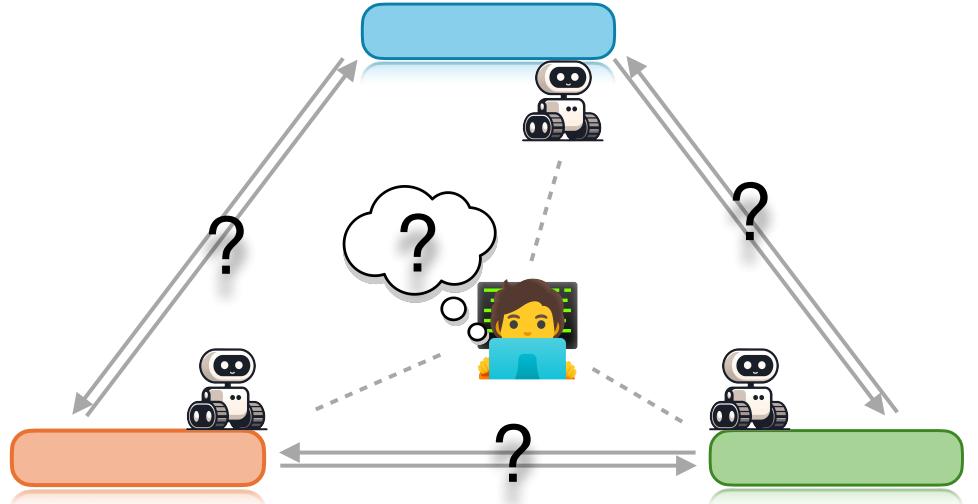


Without tests, ProofRover increases **specification completeness** (behaviour coverage) by up to 10%.

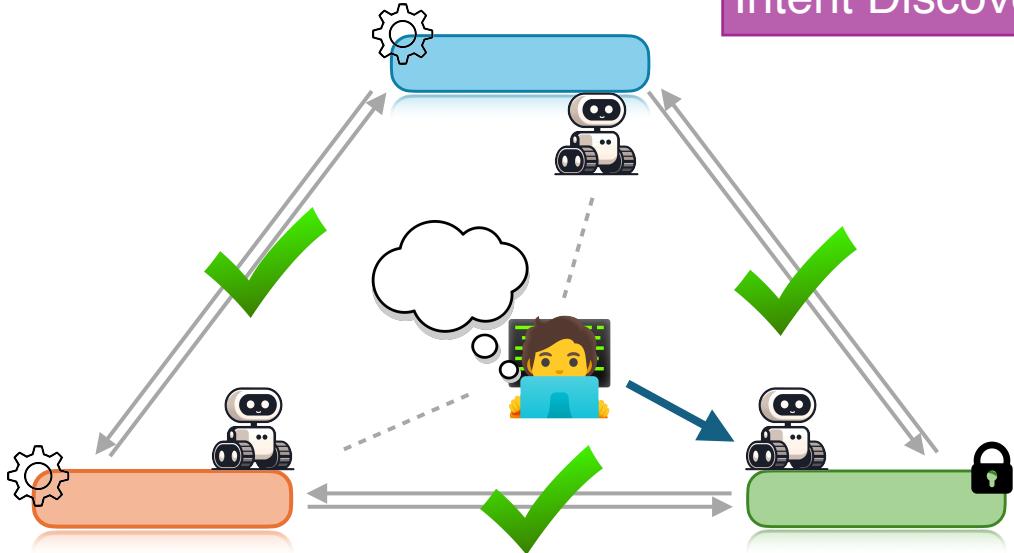
Adding tests, ProofRover increases **specification completeness** on average by 60%.

The discovered, unambiguous user intent produces 3x more verified code than the original intent.

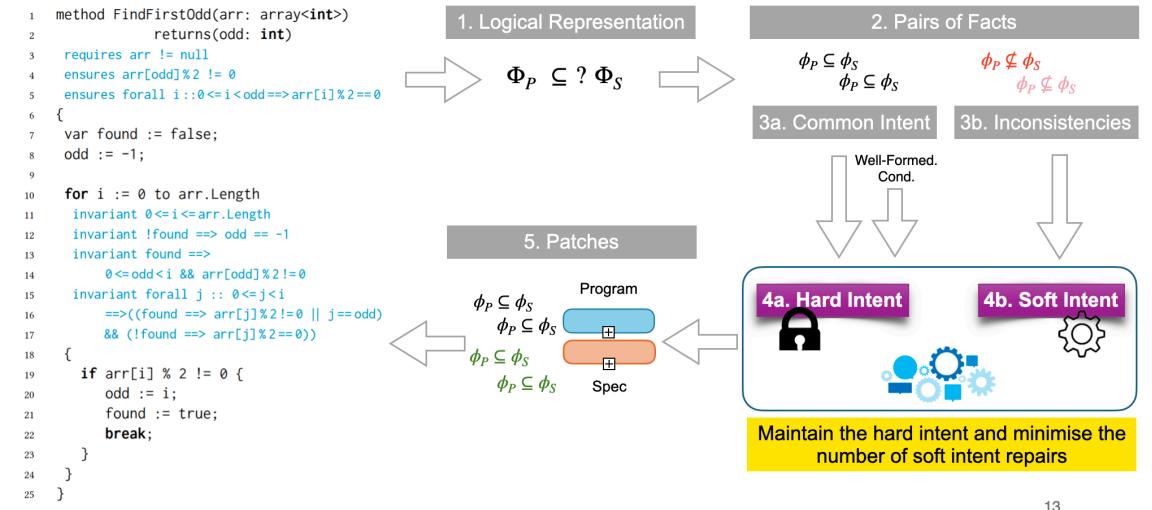




Artifacts Alignment



Intent Discovery



Program-Proof Co-evolution

Automated Assurance

1. Can verification-aware languages scale up?
2. Patch Validation: Can we use hard/soft intent as a means to measure validation?
3. Maintainability: for whom?

Thank you!

PS Conformance

Model	Temp. T	Naive Repair		Chain-of-Thought		PROOFROVER	
		Aligned	Avg. Time	Aligned	Avg. Time	Aligned	Avg. Time
GPT-4O	0.3	11	43s	15	55s	23	111s
	0.7	13	46s	15	55s	23	81s
	1.0	13	39s	13	58s	22	160s
SONNET-3.5	0.3	12	45s	18	74s	27	249s
	0.7	12	46s	21	73s	31	236s
	1.0	14	49s	24	87s	30	252s
Average		12.8 (21.3%)	45s	17.3 (28.9%)	67s	22.2 (37%)	182s

Table 1. Reaching conformance between program and specification on **datasetNV** (60 subjects)

PST Conformance

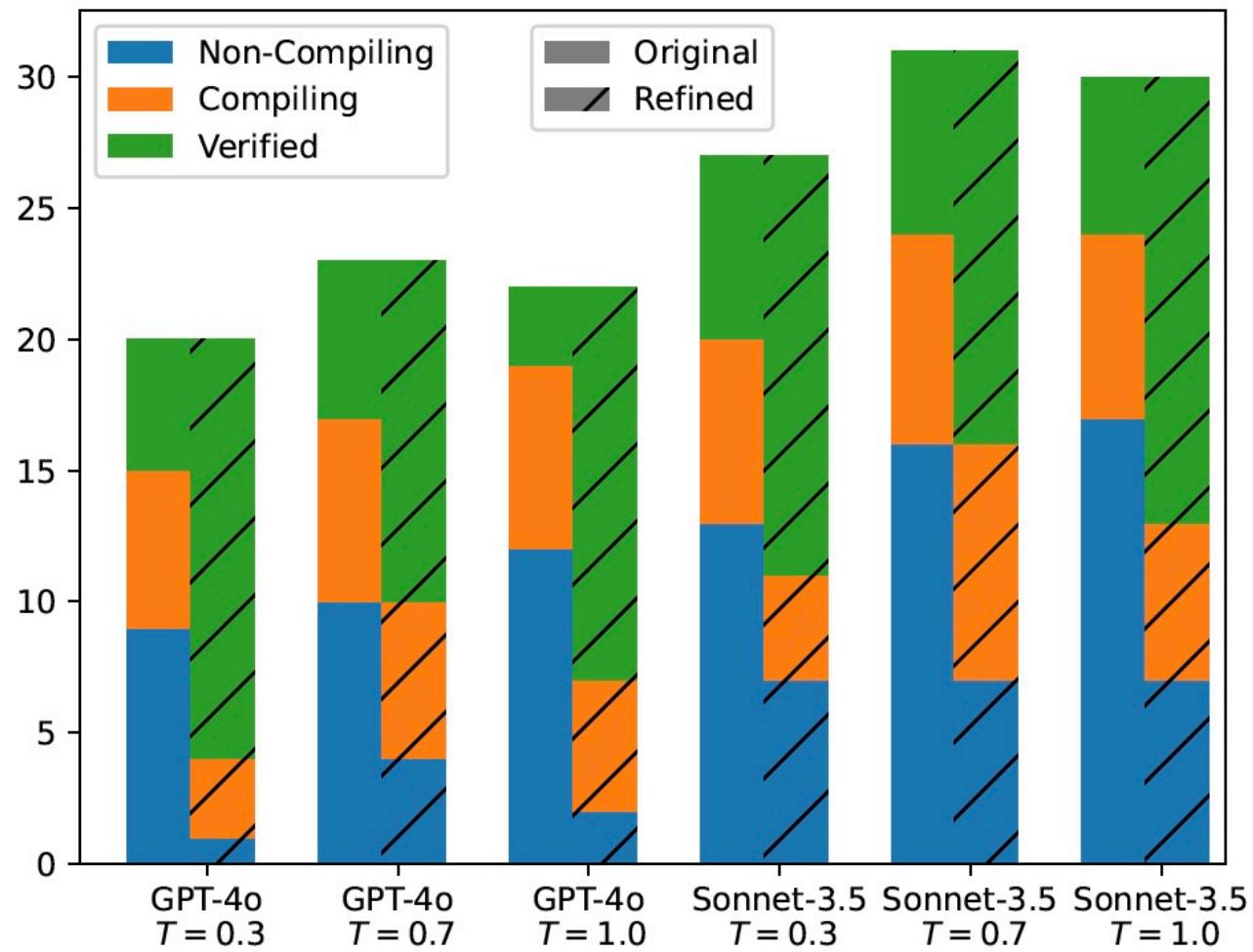
Model	Temp. T	Naive Repair		Chain-of-Thought		PROOFROVER	
		Aligned	Avg. Time	Aligned	Avg. Time	Aligned	Avg. Time
GPT-4O	0.3	66	45s	66	58s	69	94s
	0.7	69	48s	66	64s	71	70s
	1.0	77	40s	77	58s	75	67s
SONNET-3.5	0.3	67	48s	69	60s	73	143s
	0.7	68	49s	68	62s	79	140s
	1.0	69	46s	78	63s	88	195s
Average		69.3 (27.3%)	46s	70.6 (27.9%)	61s	75.83 (30%)	118s

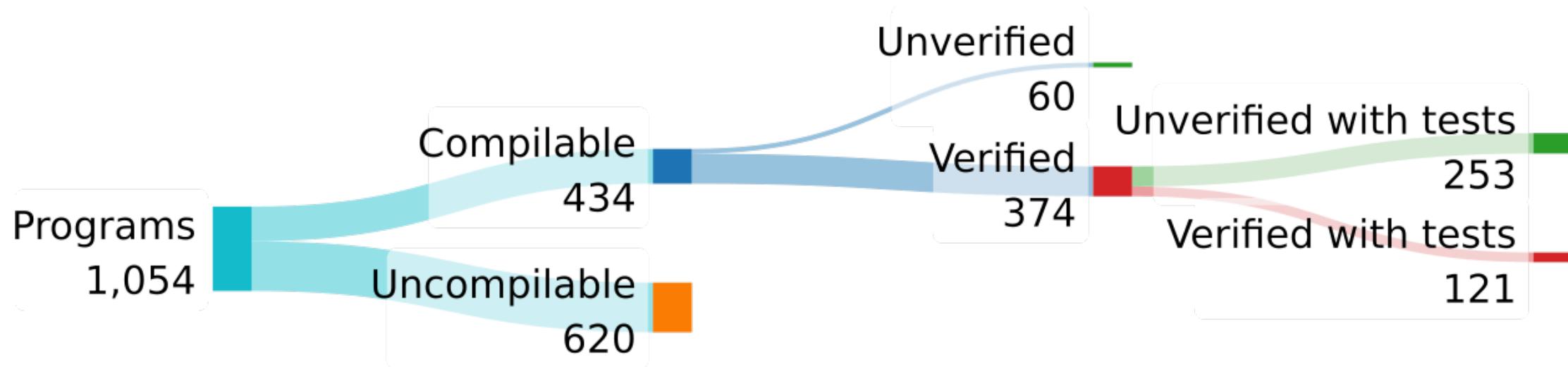
Table 2. Reaching conformance between program, specification and tests on **datasetV** (253 subjects)

Completeness

Model	Temperature T	datasetNV		datasetV	
		Initial (None)	PS	Initial(PS)	PST
GPT-4o	0.3	0.38	0.49	0.28	0.89
	0.7	0.42	0.41	0.31	0.88
	1.0	0.51	0.56	0.29	0.89
SONNET-3.5	0.3	0.53	0.45	0.23	0.84
	0.7	0.50	0.61	0.22	0.86
	1.0	0.53	0.54	0.30	0.89
Average		0.48	0.51	0.27	0.88

Assessing the ambiguity of the intent





Patch samples

```
4 --- ensures arr[odd] %2 != 0;
4 +++ ensures 0 <= odd < arr.Length ==> arr[odd] %2 != 0;

5 --- ensures forall i::0 <= i < odd ==> arr[i] % 2 == 0;
5 +++ ensures 0 <= odd < arr.Length ==> (forall i::0 <= i < odd ==> arr[i] % 2 == 0);

6 +++ ensures (forall i :: 0 <= i< arr.Length ==> arr[i] % 2 == 0) ==> odd == -1

6 +++ ensures (forall i :: 0 <= i< arr.Length ==> arr[i] % 2 == 0) ==> odd == -arr.Length
8 --- odd := -1;
8 +++ odd := -arr.Length;
```

Heuristic for selecting soft intent facts as repair candidates

Generally, we choose the fact that is inconsistent with the most hard intent facts.

If two facts break even, we next choose the one which is inconsistent with the soft intent facts.

If they further break even, we then choose the strongest among the two if they are in an implication relation, i.e. $A \Rightarrow B$, or pick randomly otherwise.

Passification example

```
var x:=1; var y:=2;           (assume true;, assert x + y >= 2)
```

$$\text{true} \Rightarrow ((x = 1) \Rightarrow ((y = 2) \Rightarrow (x + y \geq 2))).$$