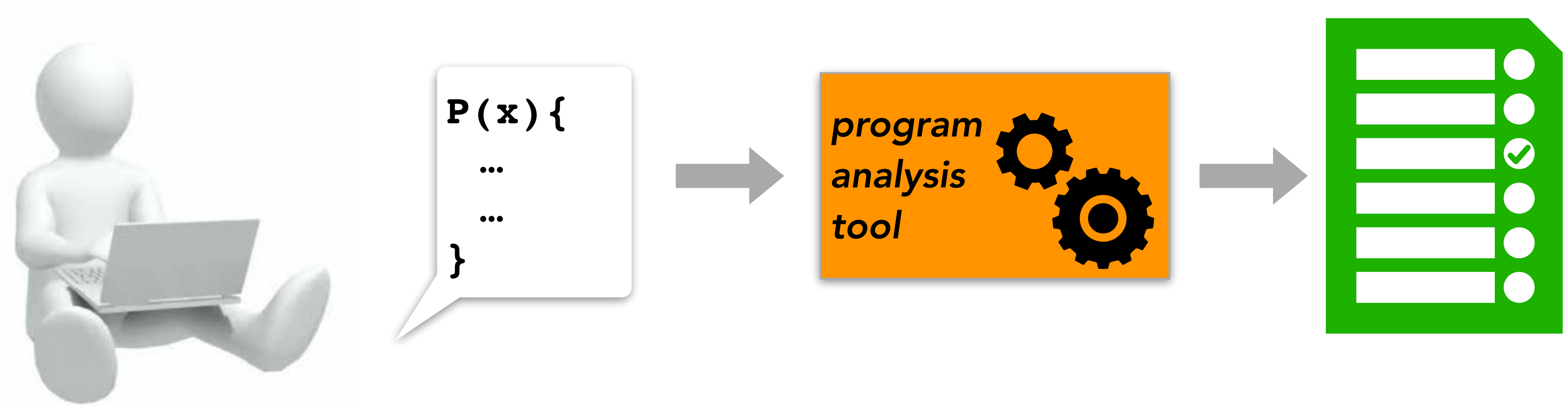
Training Binary Classifiers as Data Structure Invariants

Facundo Molina

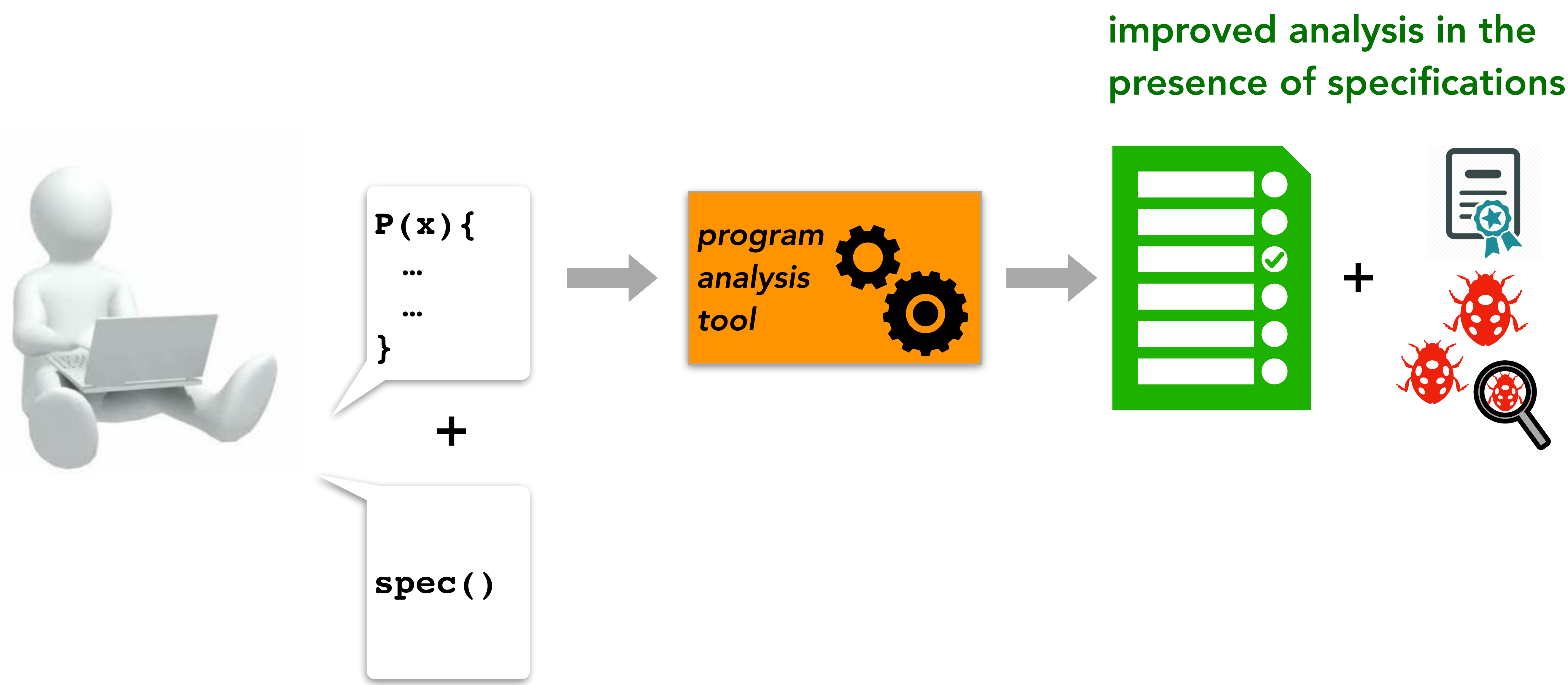
Department of Computer Science, University of Rio Cuarto, Argentina
CONICET, Argentina

in collaboration with Renzo Degiovanni, Pablo Ponzio, Germán Regis, Nazareno Aguirre and Marcelo Frias

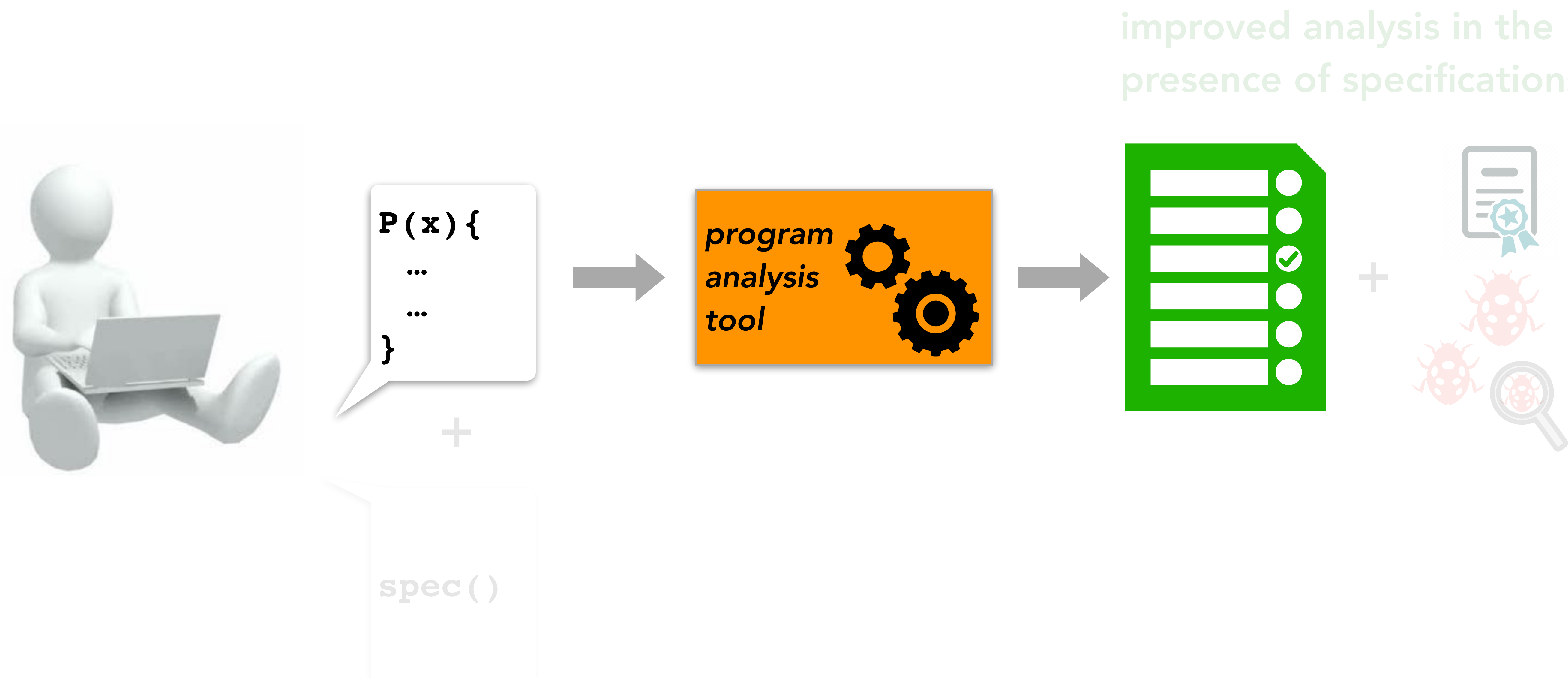
An Automated Analysis Scenario



An Automated Analysis Scenario

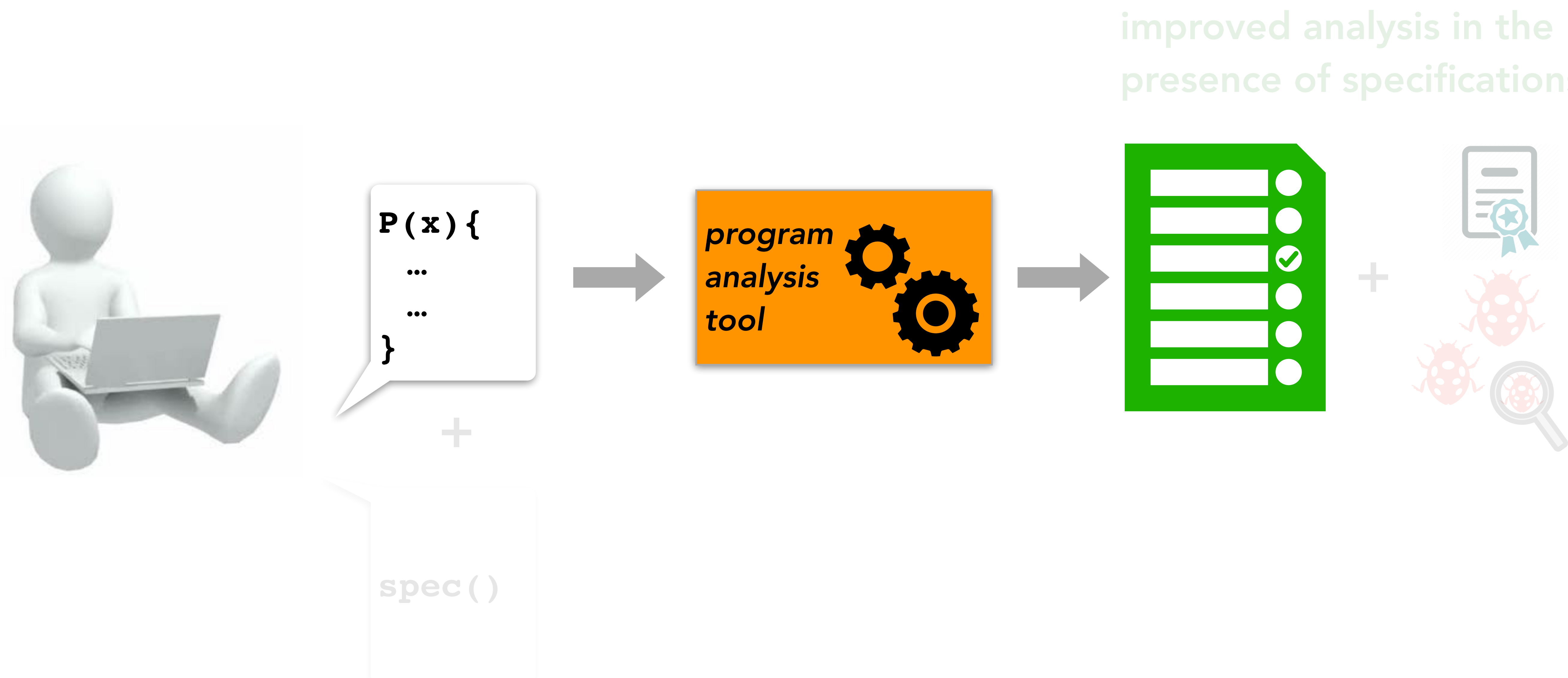


An Automated Analysis Scenario



unfortunately, specifications
are sometimes unavailable

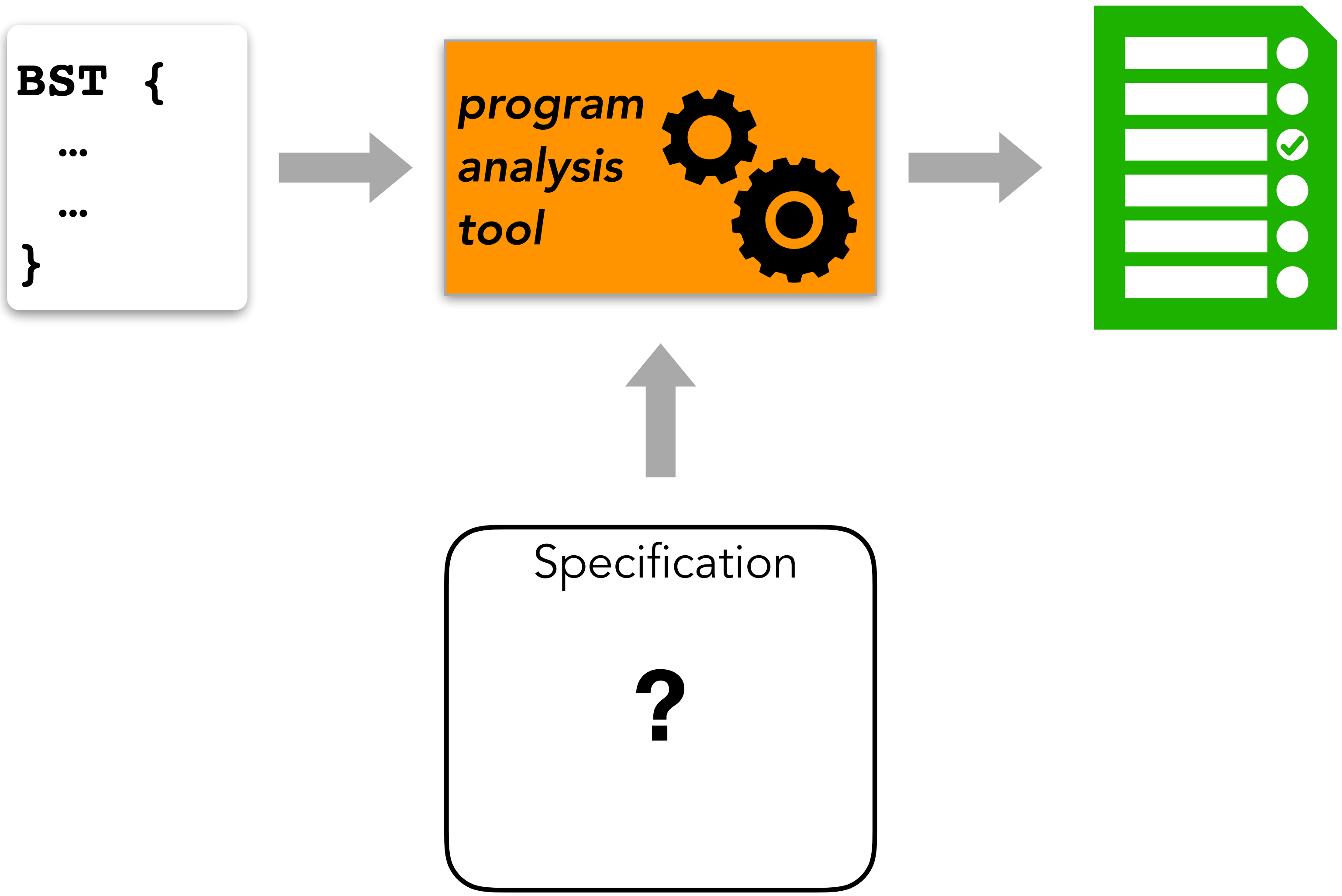
An Automated Analysis Scenario



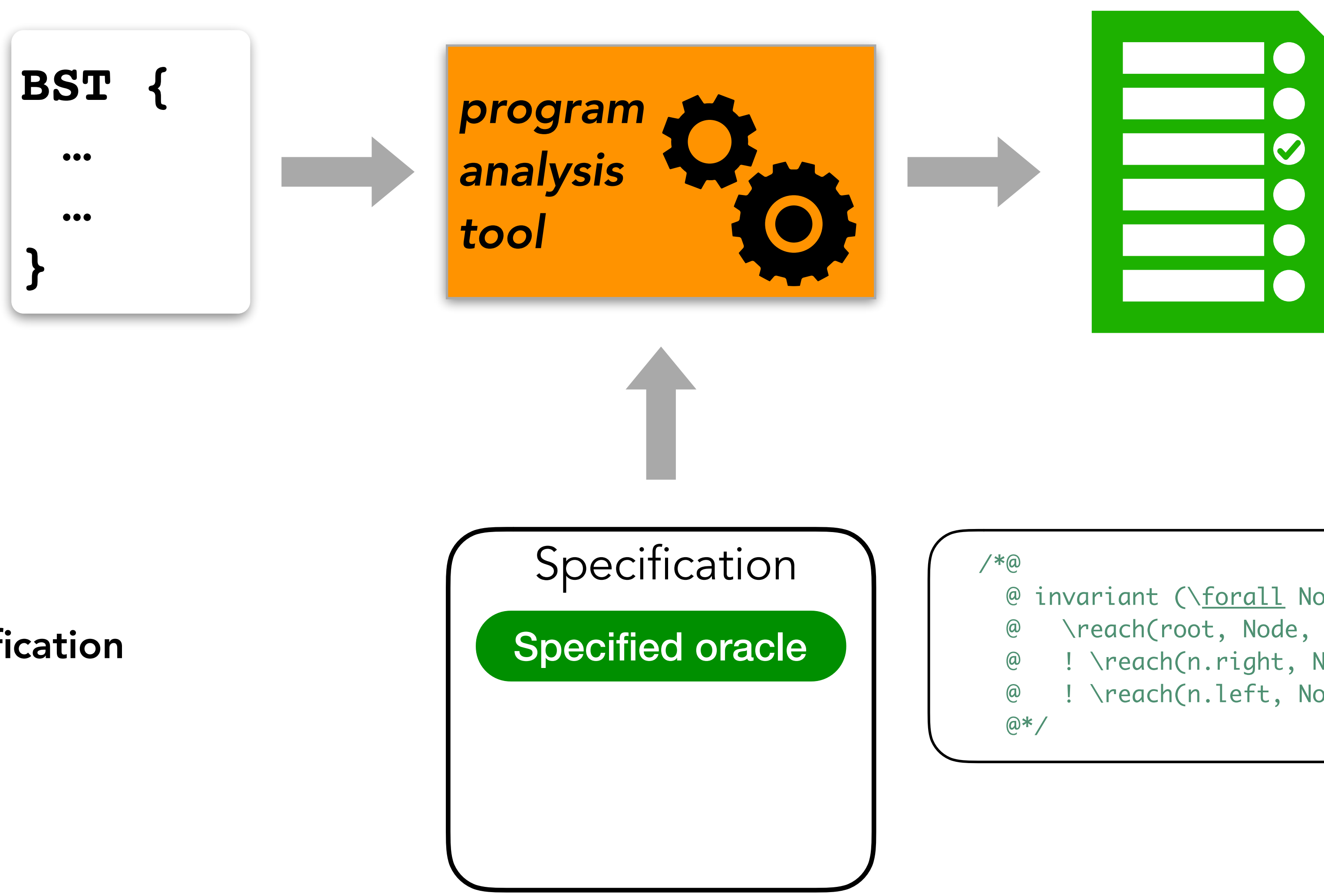
unfortunately, specifications are sometimes unavailable

This emphasizes the relevance of the ***oracle problem***

Approaches to the Oracle Problem



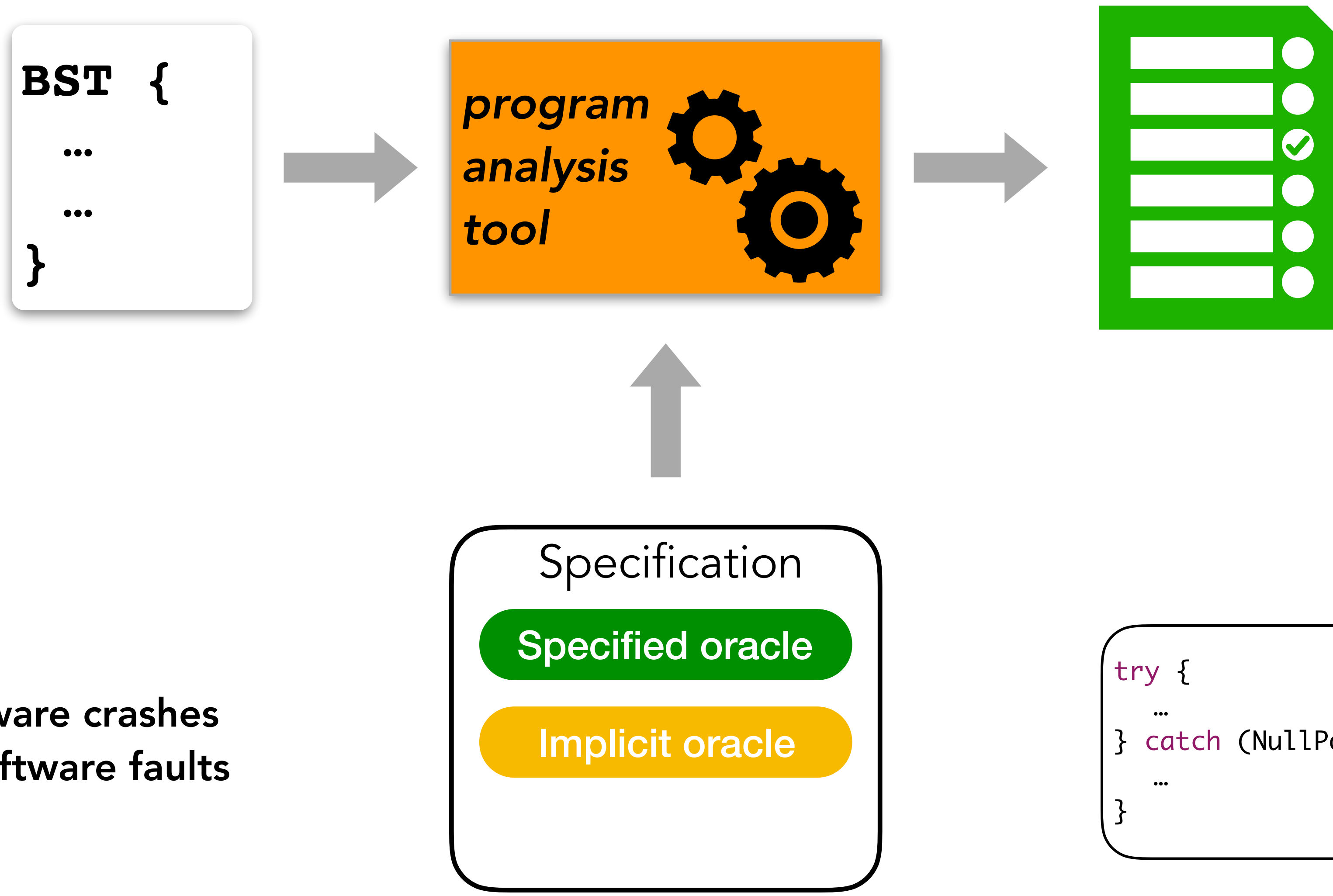
Approaches to the Oracle Problem



Manual specification

```
/*@
  @ invariant (\forall Node n;
  @   \reach(root, Node, left + right).has(n) implies
  @   ! \reach(n.right, Node, right + left).has(n) &&
  @   ! \reach(n.left, Node, left + right).has(n));
  @*/
```

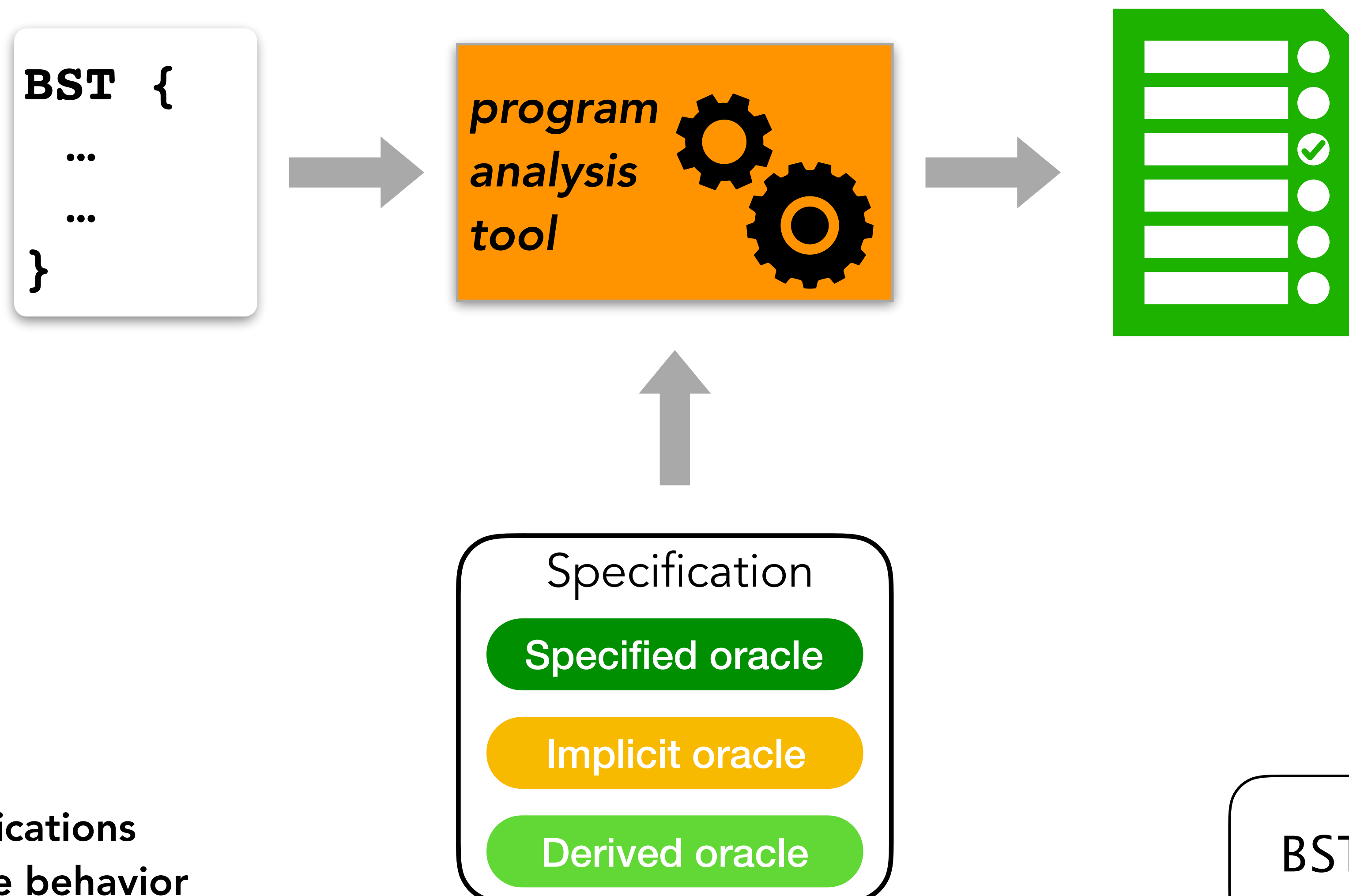
Approaches to the Oracle Problem



Analyze software crashes or general software faults

```
try {
   …
} catch (NullPointerException e) {
   …
}
```

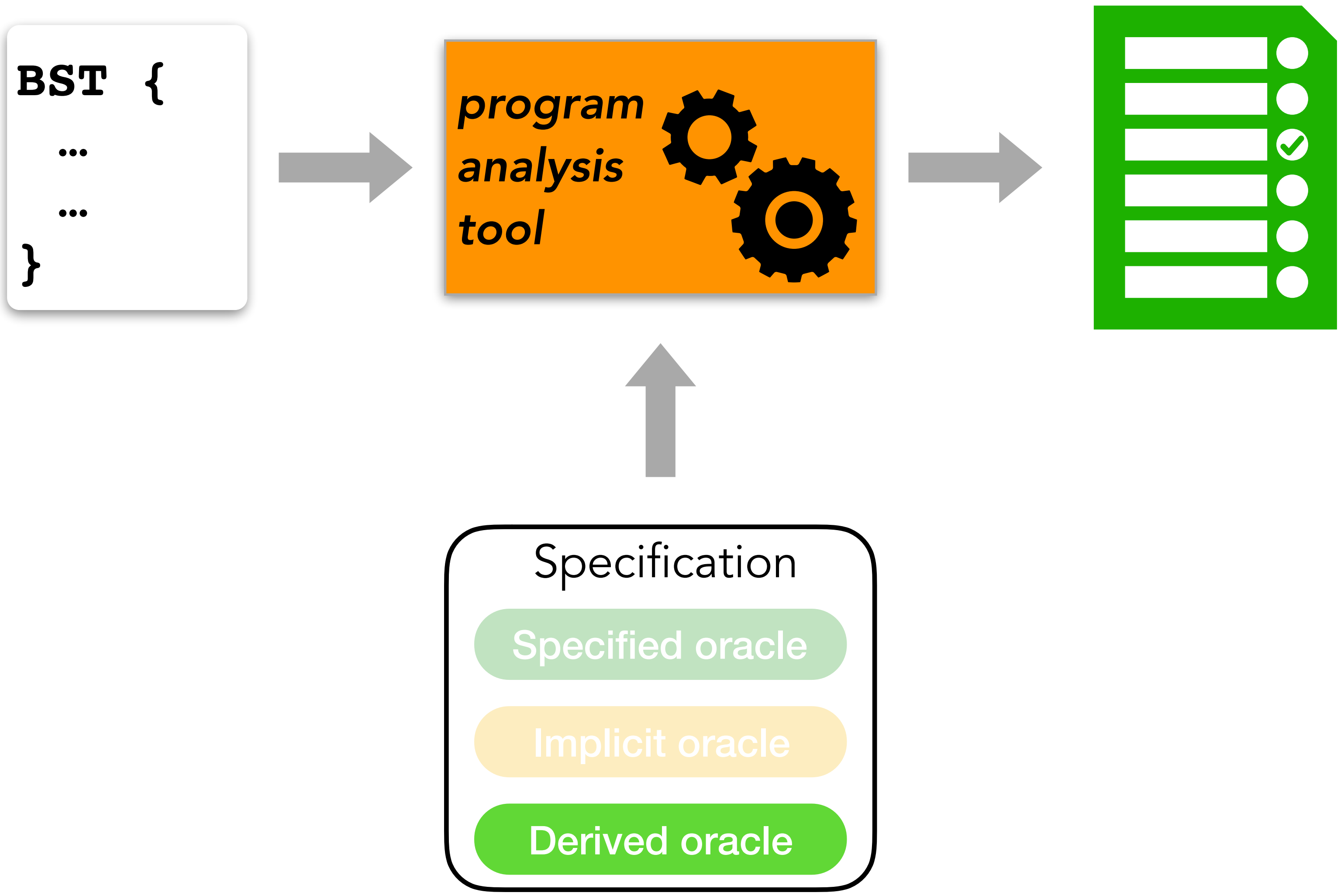
Approaches to the Oracle Problem



Derive specifications from software behavior

$\mathrm{BST} \equiv \mathrm{BST}'$

Approaches to the Oracle Problem



Invariants as Oracles

```
public class BST {

  public void BST() {

  }

  public void
  insert(…) {
    …
  }

  public void
  remove() {
    …
  }
  …
}
```

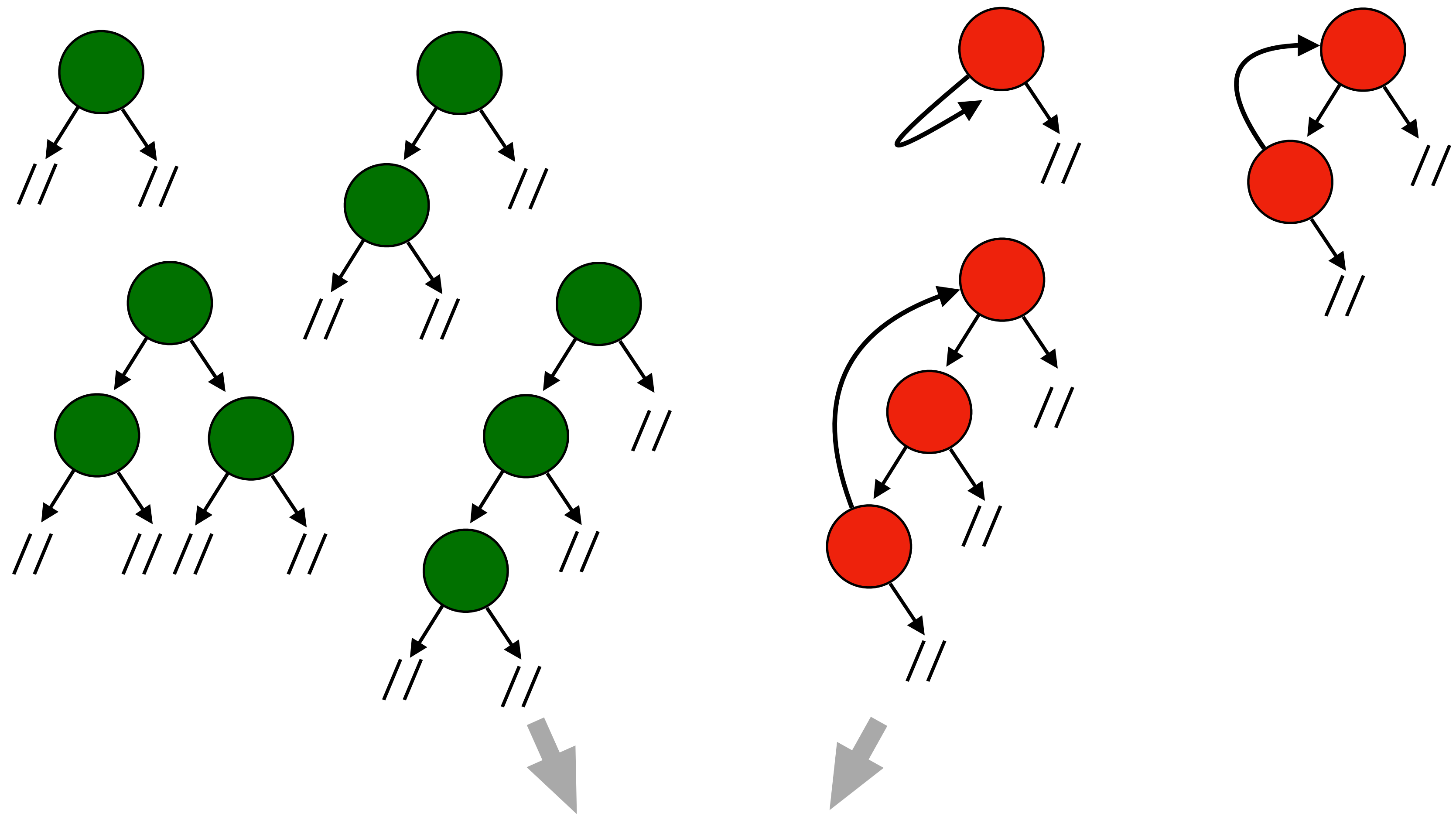
```
/*@
  @ invariant (\forall Node n;
  @   \reach(root, Node, left + right).has(n) implies
  @   ! \reach(n.right, Node, right + left).has(n) &&
  @   ! \reach(n.left, Node, left + right).has(n));
  @*/
```

Invariants as Oracles

```
public class BST {

  public void BST() {

  }

  public void
  insert(…) {
    …
  }

  public void
  remove() {
    …
  }
  …
}
```

correct program states

incorrect program states



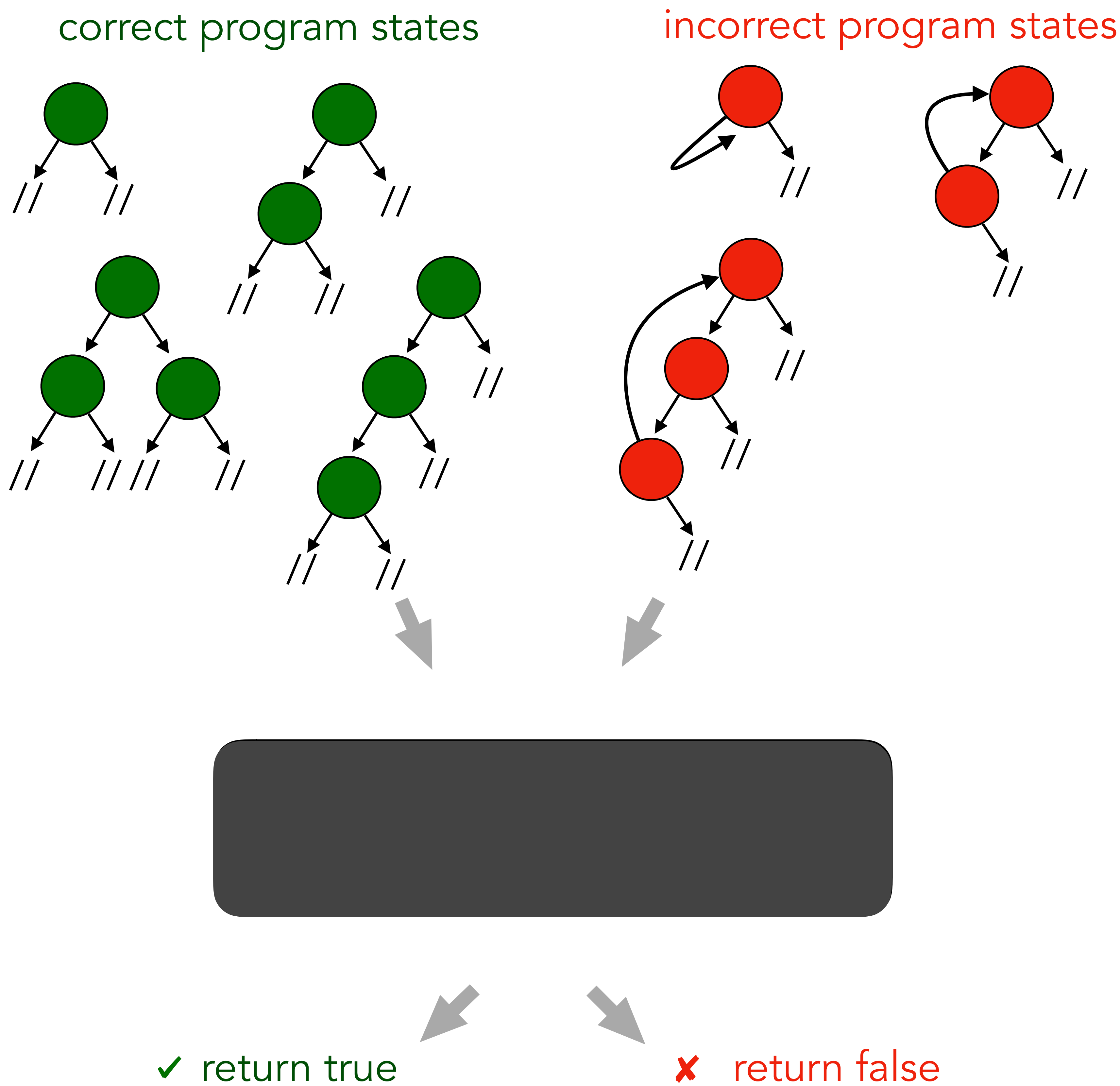

```
/*@
  @ invariant (\forall Node n;
  @   \reach(root, Node, left + right).has(n) implies
  @   ! \reach(n.right, Node, right + left).has(n) &&
  @   ! \reach(n.left, Node, left + right).has(n));
  @*/
```

✓ return true

✘ return false

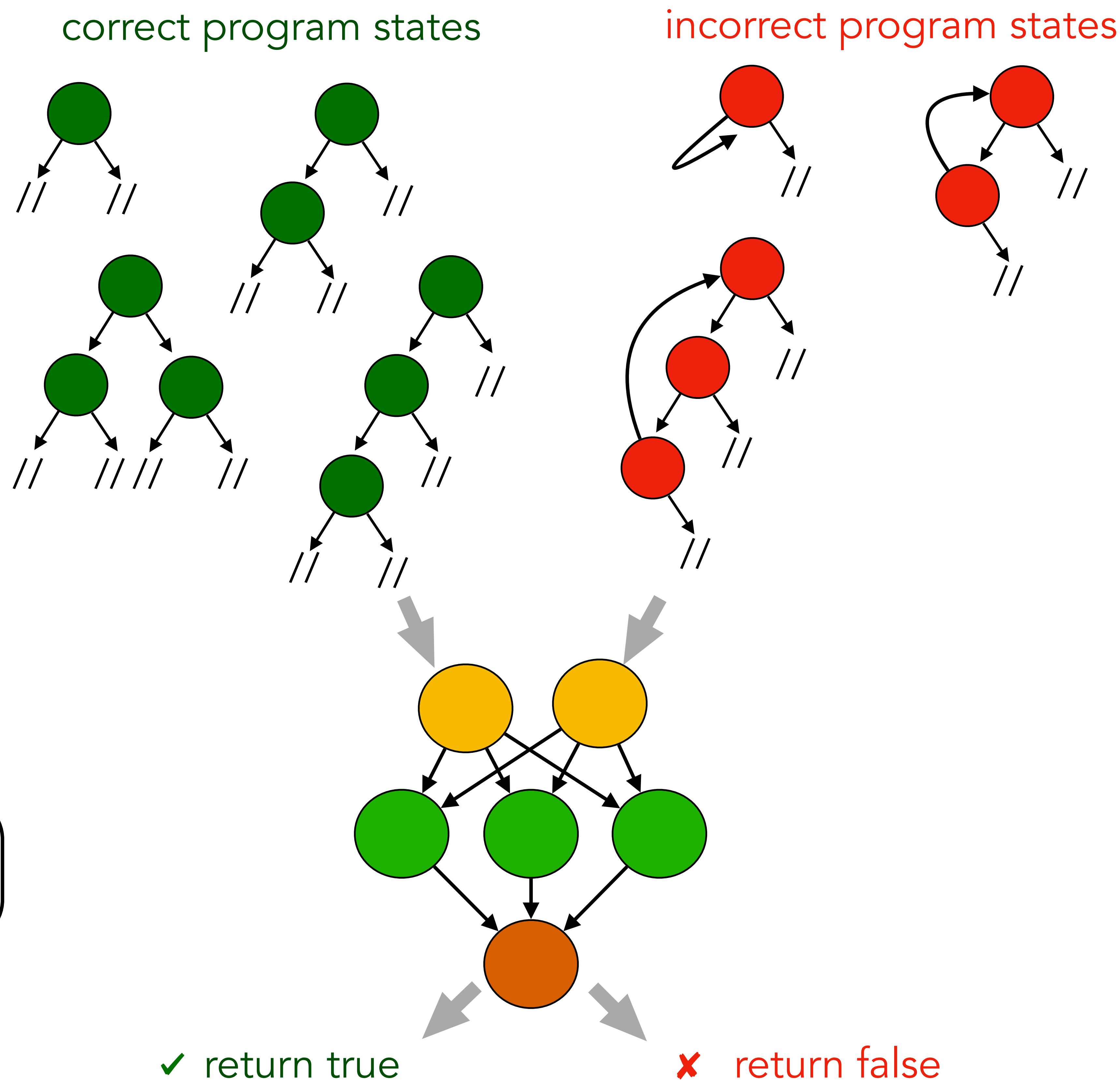
Invariants as Oracles

```
public class BST {

  public void BST() {

  }

  public void
  insert(…) {
    …
  }

  public void
  remove() {
    …
  }
  …
}
```



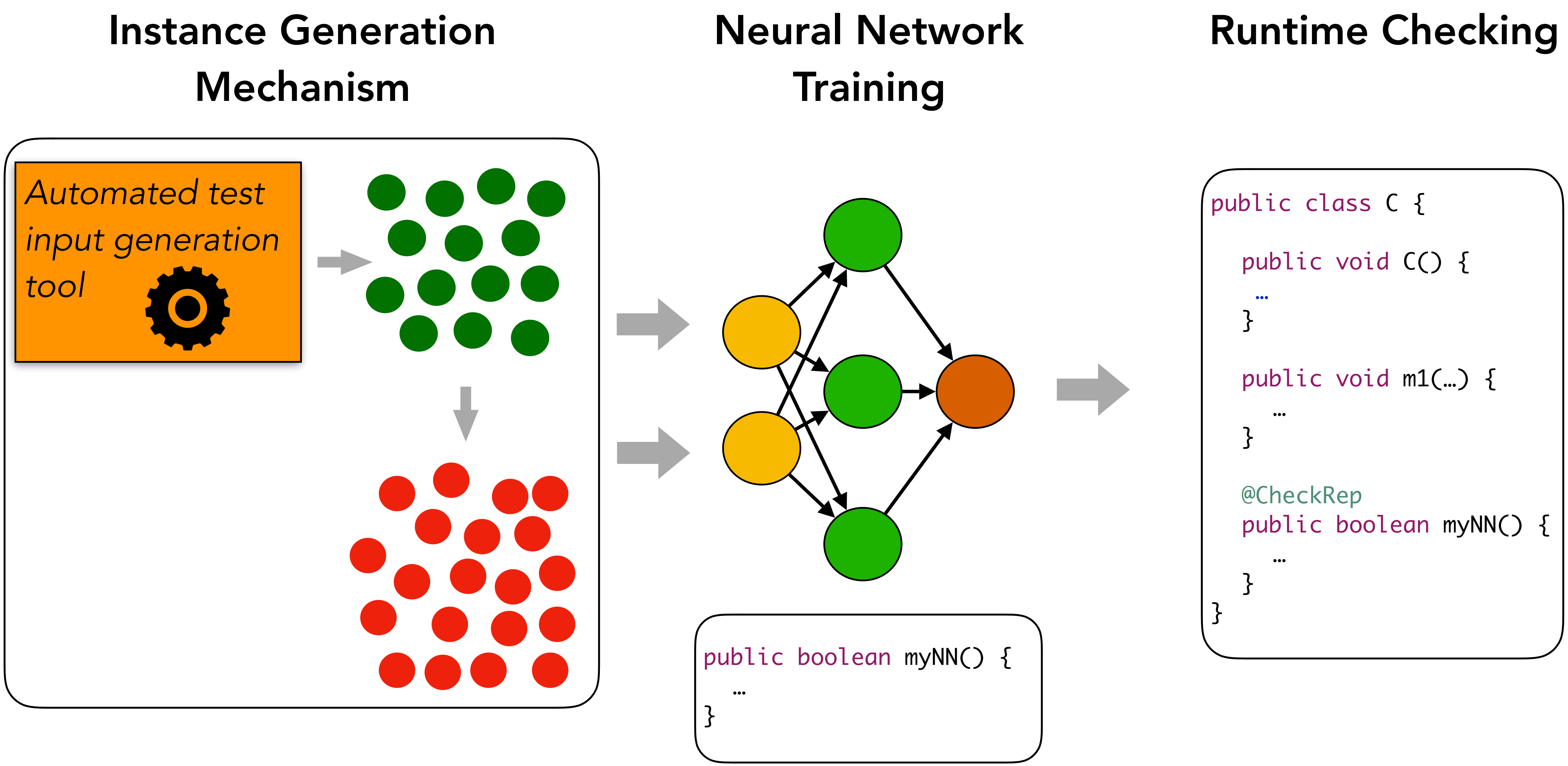
Invariants as Oracles

```java
public class BST {

  public void BST() {

  }

  public void
  insert(…) {
    …
  }

  public void
  remove() {
    …
  }
  …
}
```



Train a machine learning classifier in order to be used as a test oracle

An Overview of the Approach



Instances Generation Mechanism

Positive instances: generated using assumed-correct building routines

```
public class C {

  public void C() {
     …
  }

  public void m1() {
     …
  }

  public void m2() {
     …
  }
  …
}
```

Instances Generation Mechanism

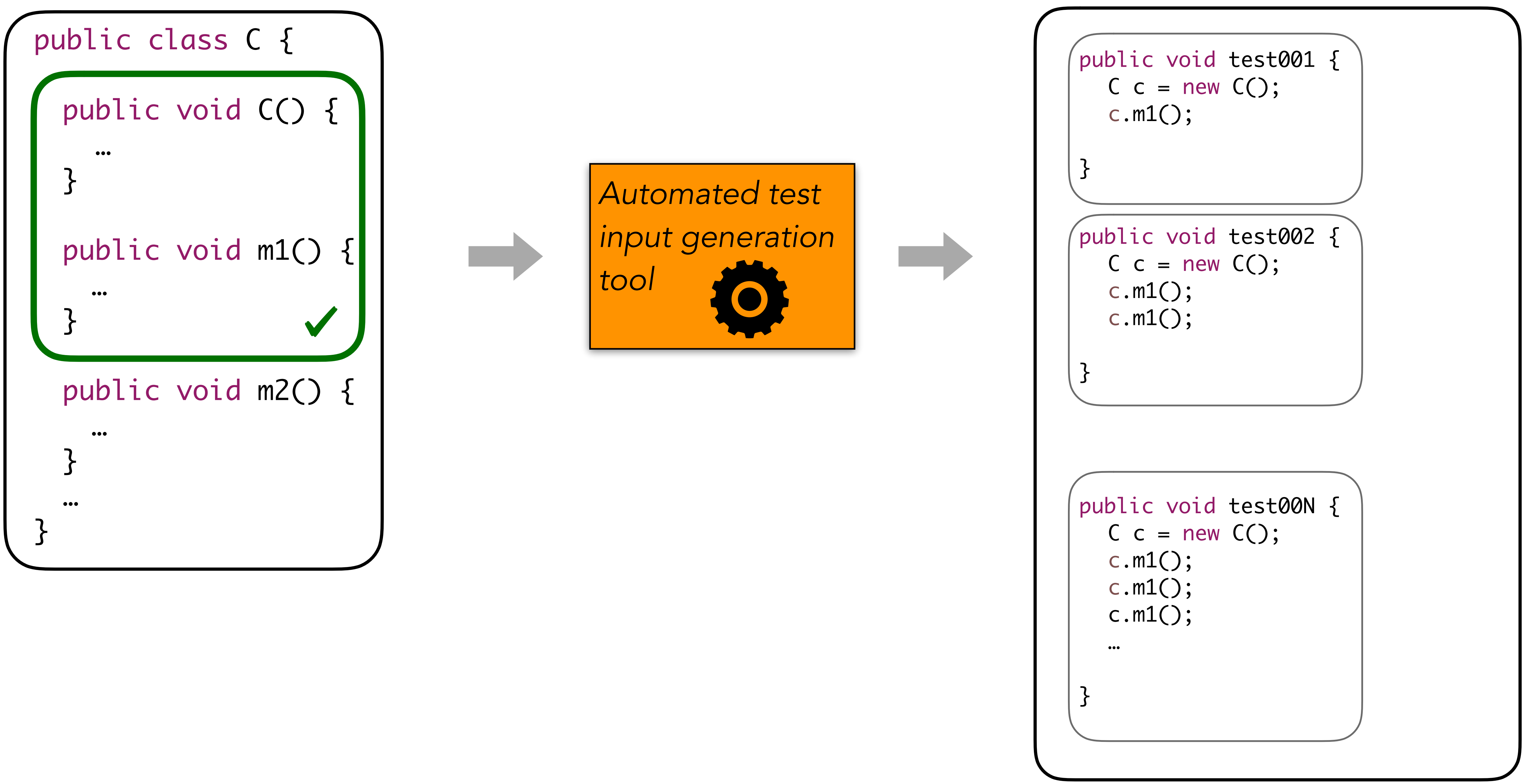
Positive instances: generated using assumed-correct building routines

Identification of assumed-correct builders

```
public class C {
  public void C() {
    …
  }

  public void m1() {
    …
  }
  public void m2() {
    …
  }
  …
}
```

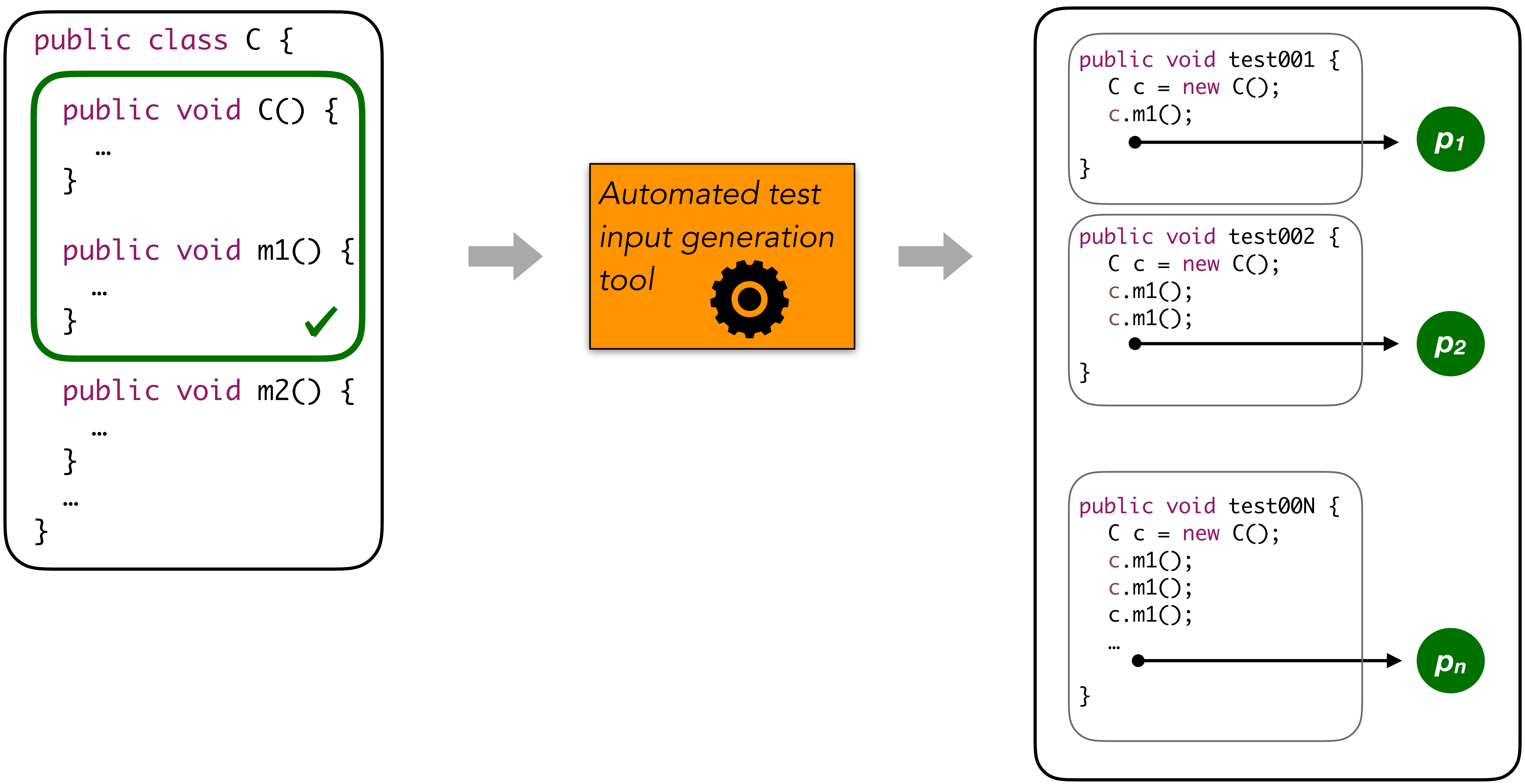
Instances Generation Mechanism

Positive instances: generated using assumed-correct building routines



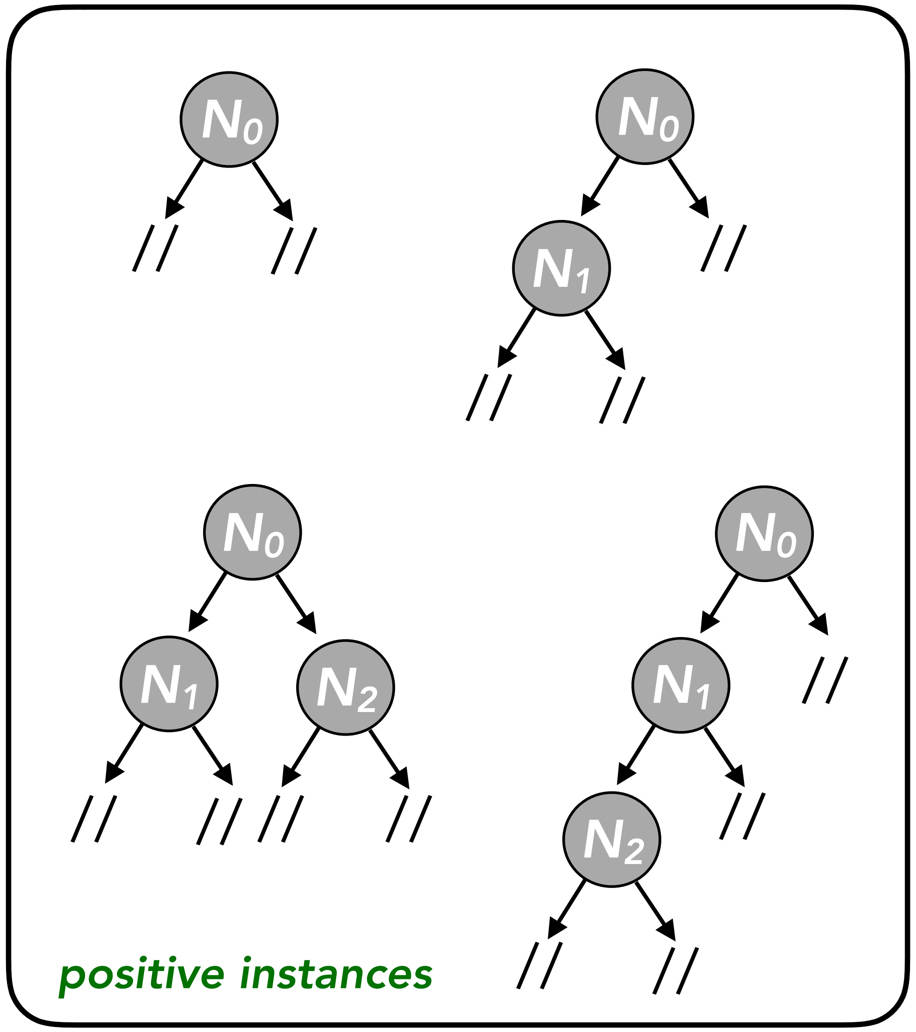
Instances Generation Mechanism

Positive instances: generated using assumed-correct building routines

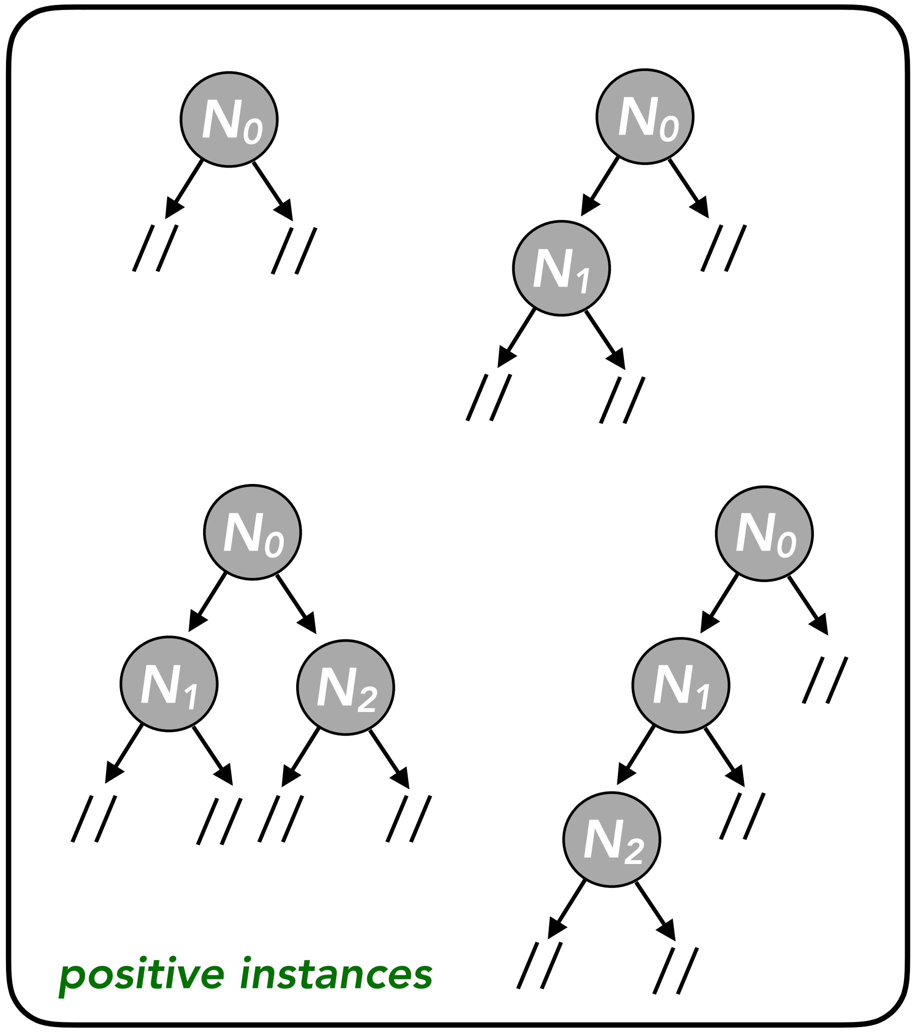


Negative instances: build a ***relational summary*** from valid instances, and mutate valid objects off the summary

Negative instances: build a ***relational summary*** from valid instances, and mutate valid objects off the summary



Negative instances: build a ***relational summary*** from valid instances, and mutate valid objects off the summary

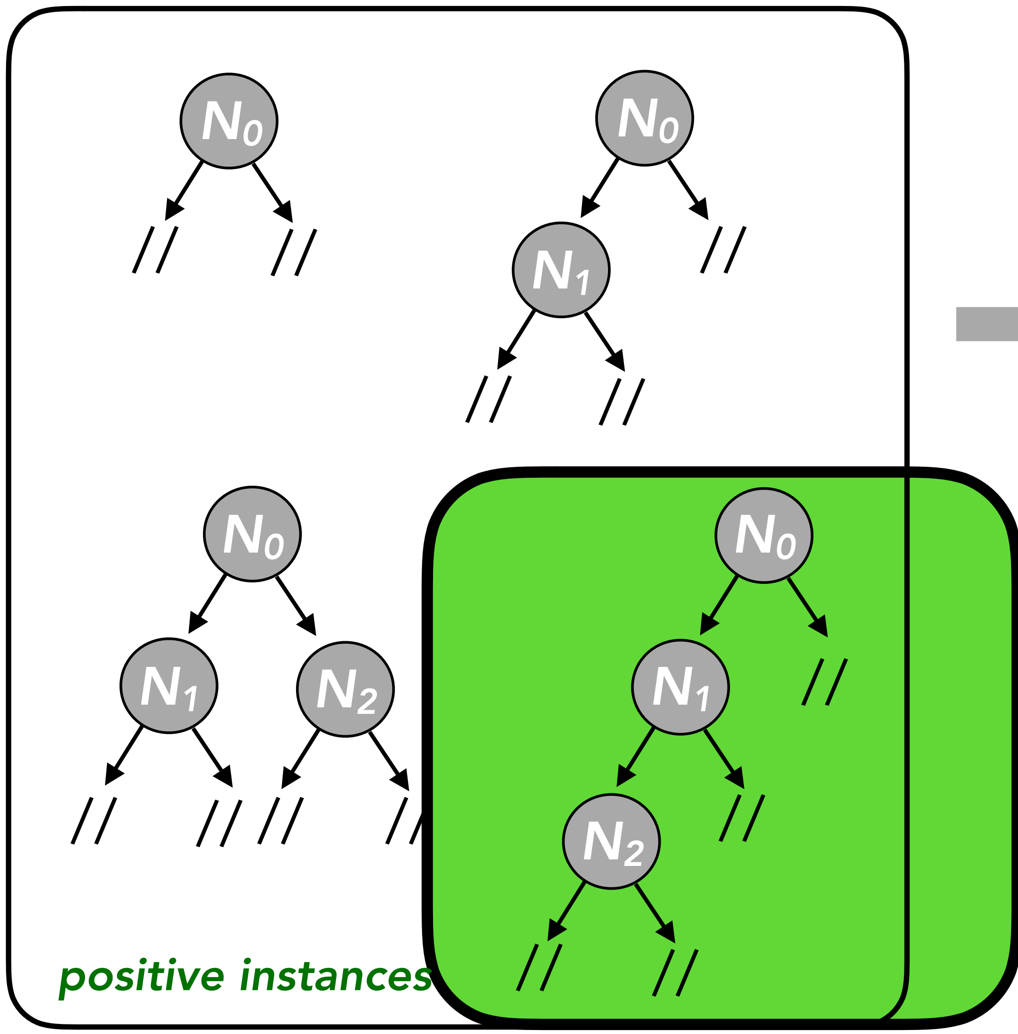


left	*null*	N_0	N_1	N_2
N_0				
N_1				
N_2				

right	*null*	N_0	N_1	N_2
N_0				
N_1				
N_2				

relational summary

Negative instances: build a ***relational summary*** from valid instances, and mutate valid objects off the summary

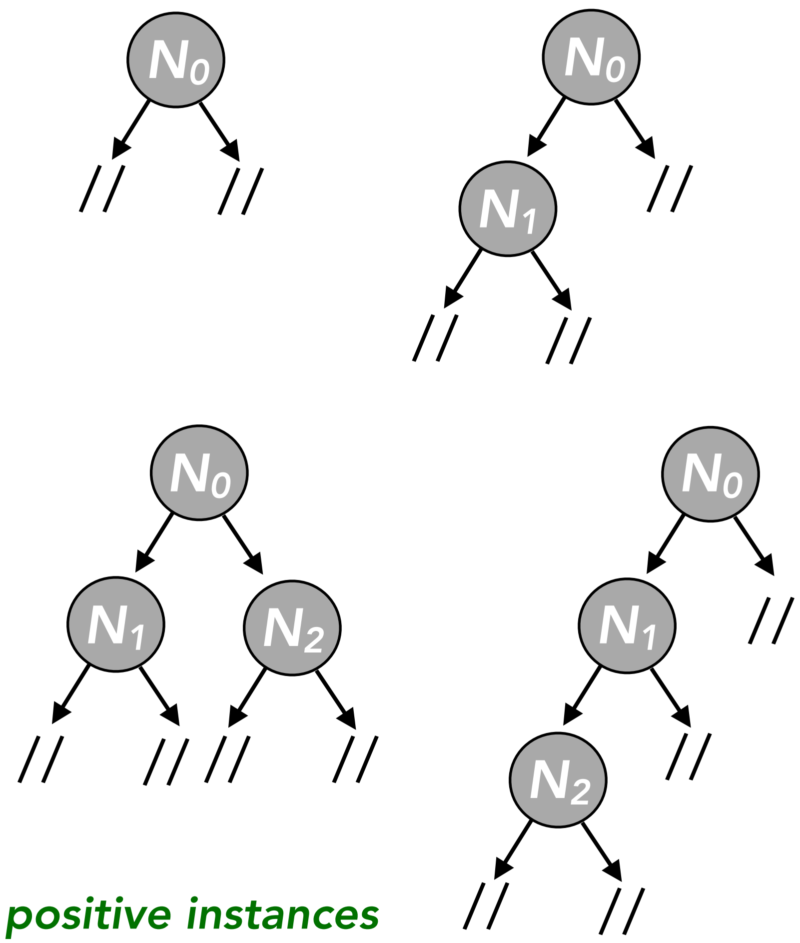


left	*null*	N_0	N_1	N_2
N_0	✓		✓	
N_1	✓			✓
N_2	✓			

right	*null*	N_0	N_1	N_2
N_0	✓			✓
N_1	✓			
N_2	✓			

relational summary

Negative instances: build a ***relational summary*** from valid instances, and mutate valid objects off the summary



positive instances

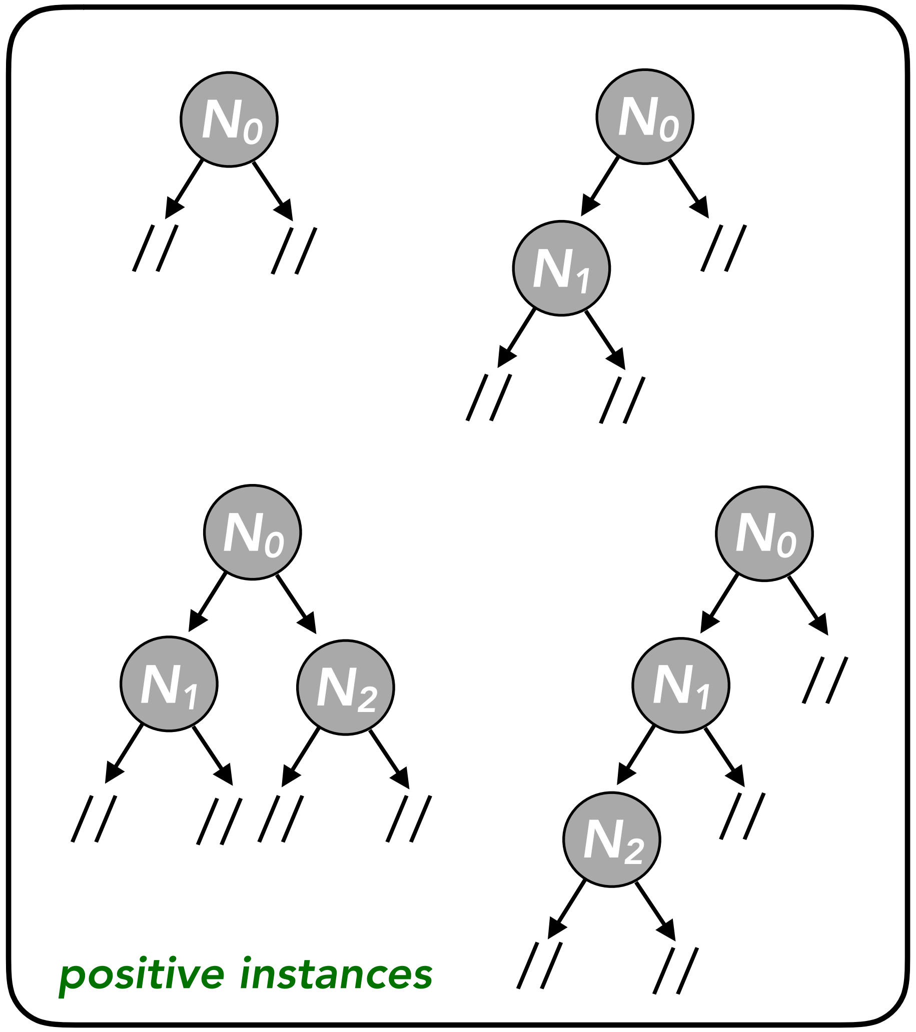
left	*null*	N_0	N_1	N_2
N_0	✓		✓	
N_1	✓			✓
N_2	✓			

right	*null*	N_0	N_1	N_2
N_0	✓			✓
N_1	✓			
N_2	✓			

relational summary

negative instances

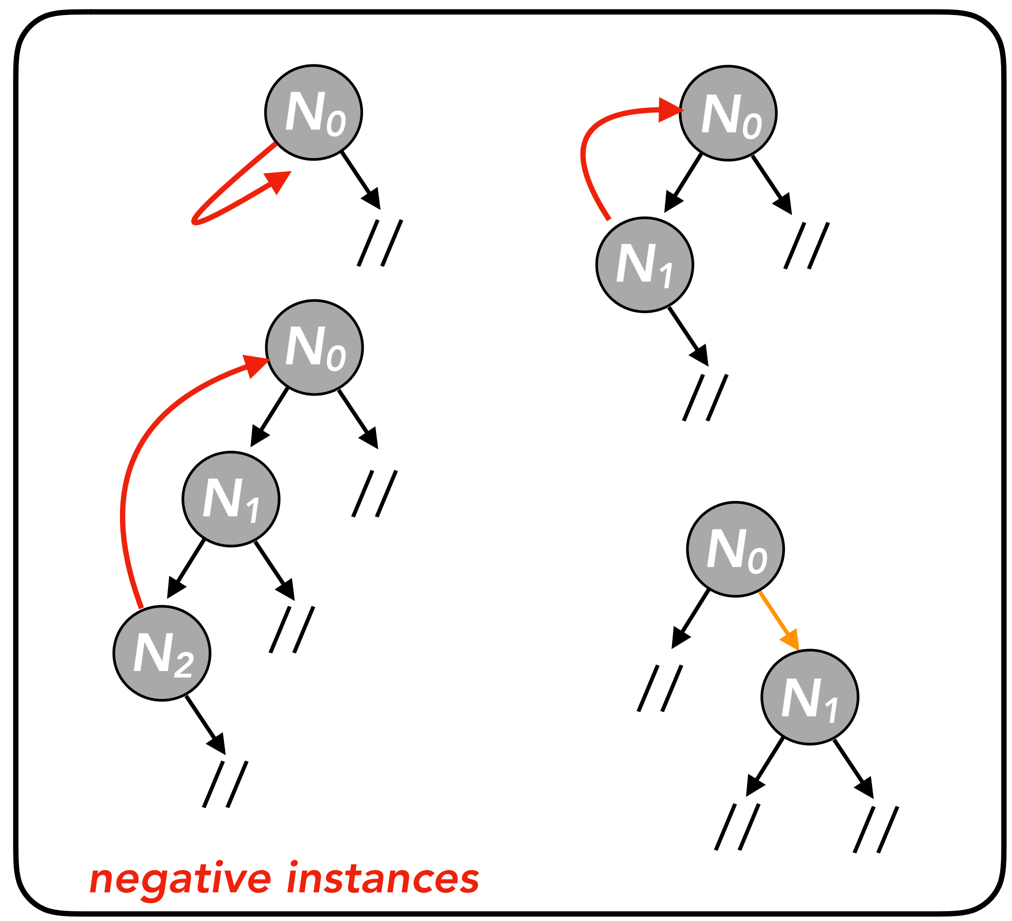
Negative instances: build a ***relational summary*** from valid instances, and mutate valid objects off the summary



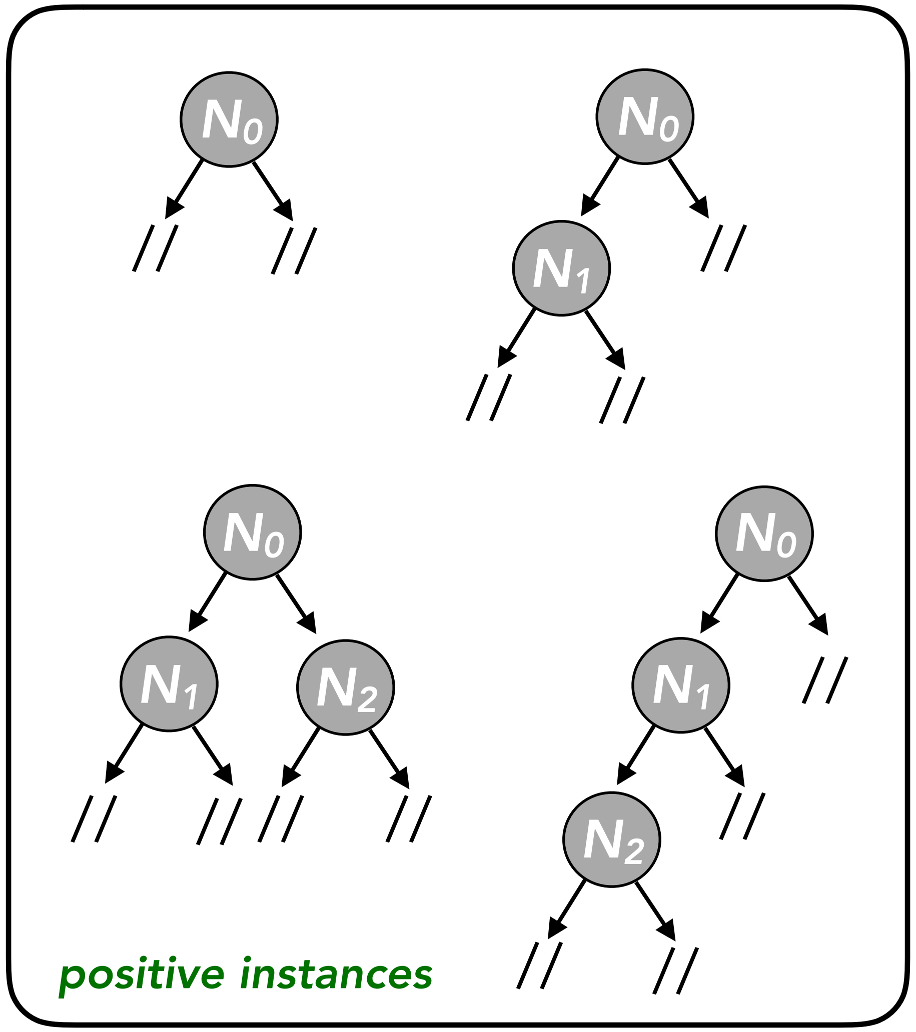
left	null	N_0	N_1	N_2
N_0	✓	✗	✓	
N_1	✓	✗		✓
N_2	✓	✗		

right	null	N_0	N_1	N_2
N_0	✓		✗	✓
N_1	✓			
N_2	✓			

relational summary



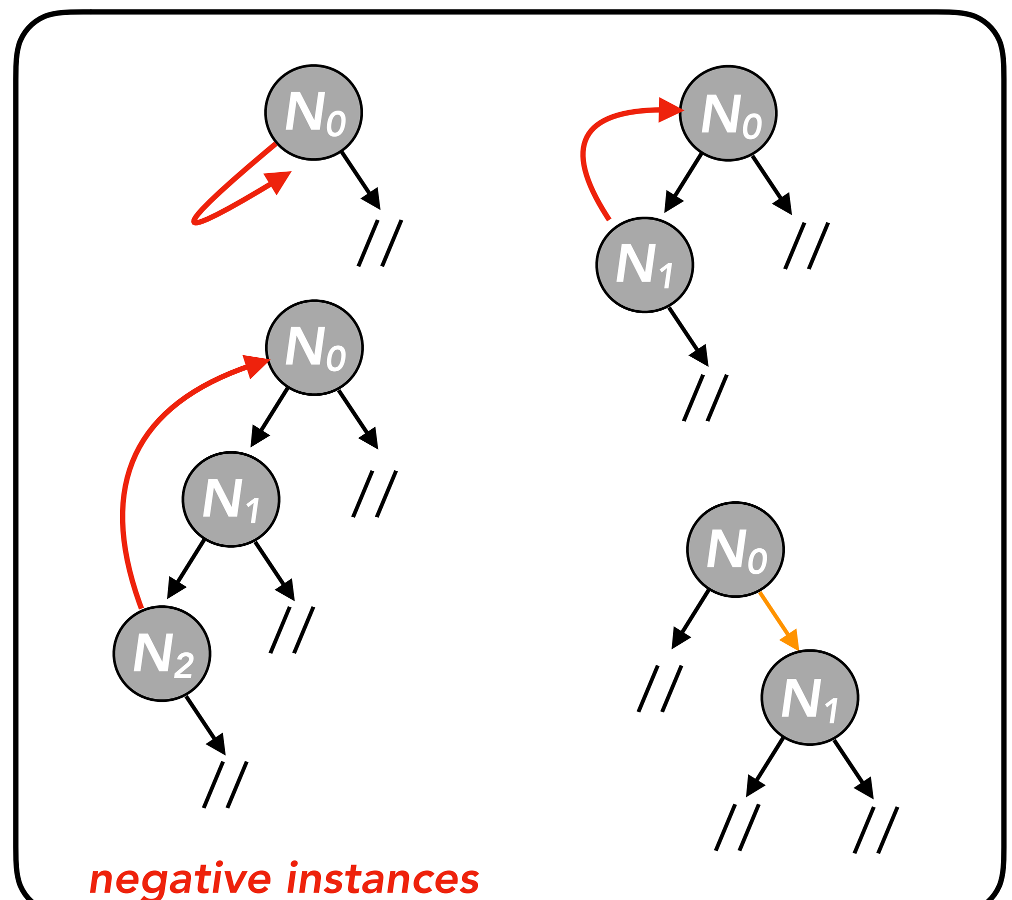
Negative instances: build a ***relational summary*** from valid instances, and mutate valid objects off the summary



left	null	N_0	N_1	N_2
N_0	✓	✗	✓	
N_1	✓	✗		✓
N_2	✓	✗		

right	null	N_0	N_1	N_2
N_0	✓		✗	✓
N_1	✓			
N_2	✓			

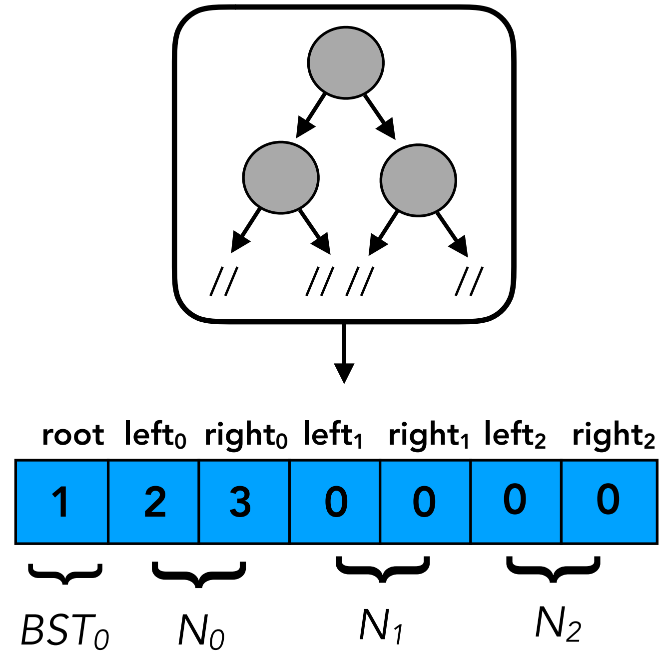
relational summary



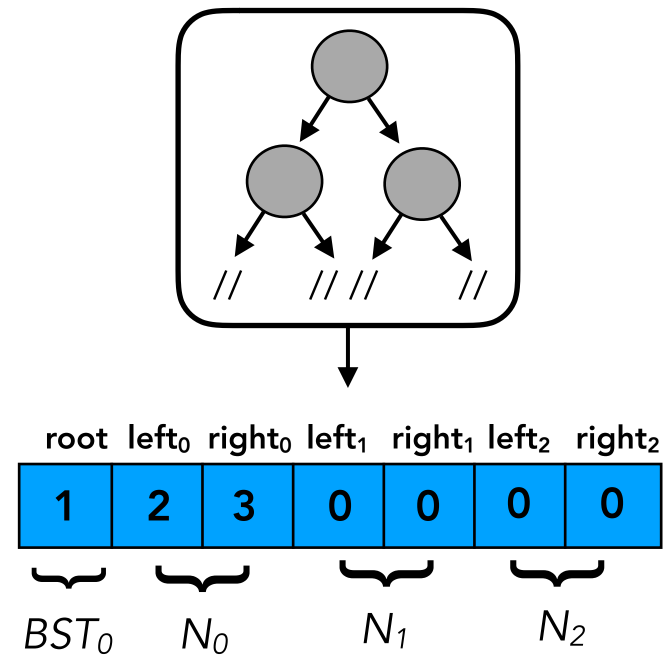
False negative instances may be generated

Vector Representation and Network Architecture

Vector representation using the Korat approach



Vector Representation and Network Architecture



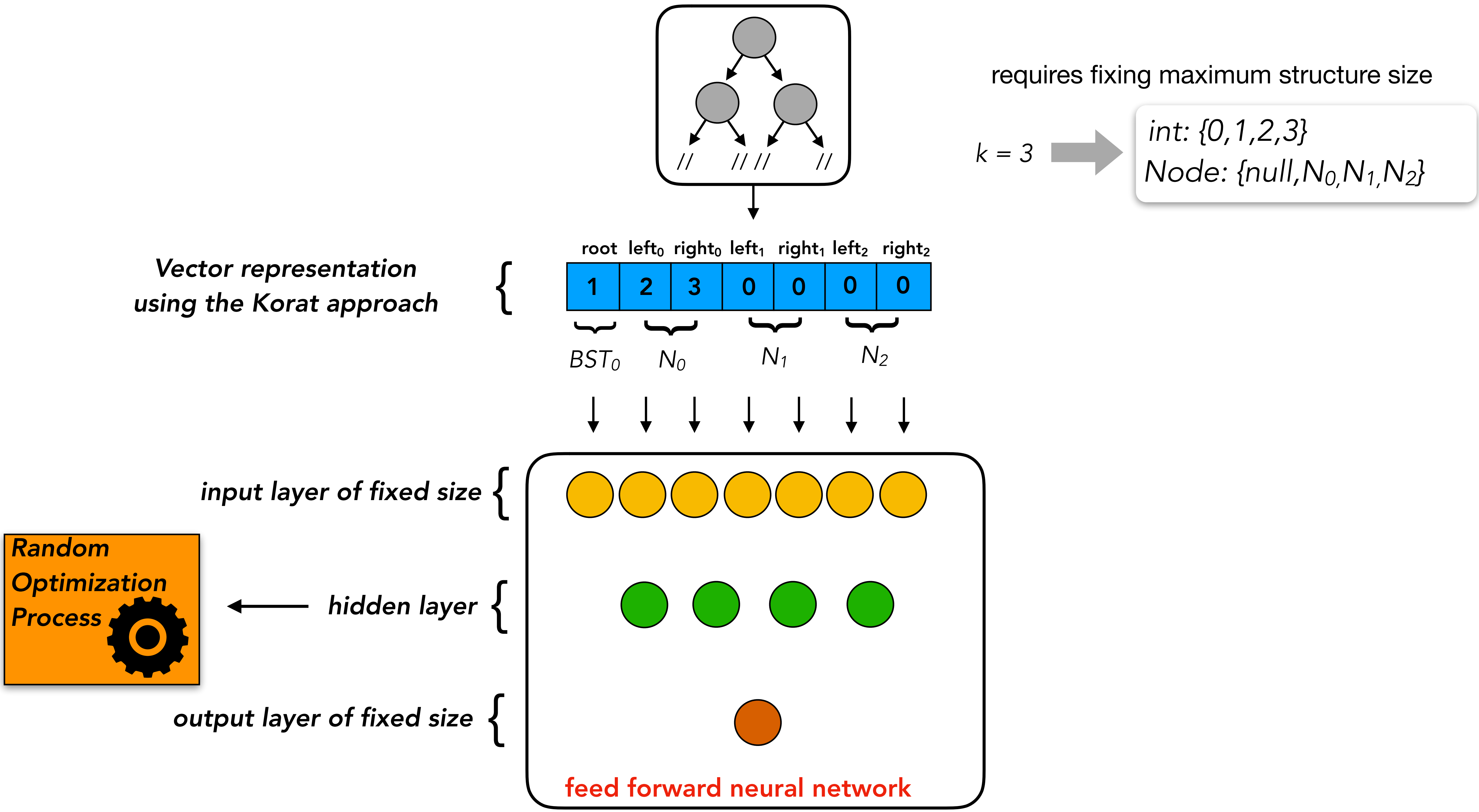
requires fixing maximum structure size

k = 3

int: {0,1,2,3}
Node: {null,N_0,N_1,N_2}

Vector representation using the Korat approach

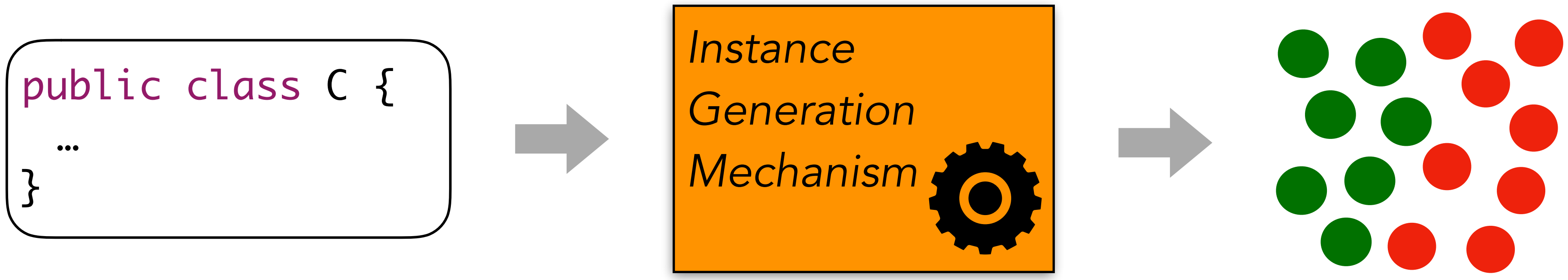
Vector Representation and Network Architecture



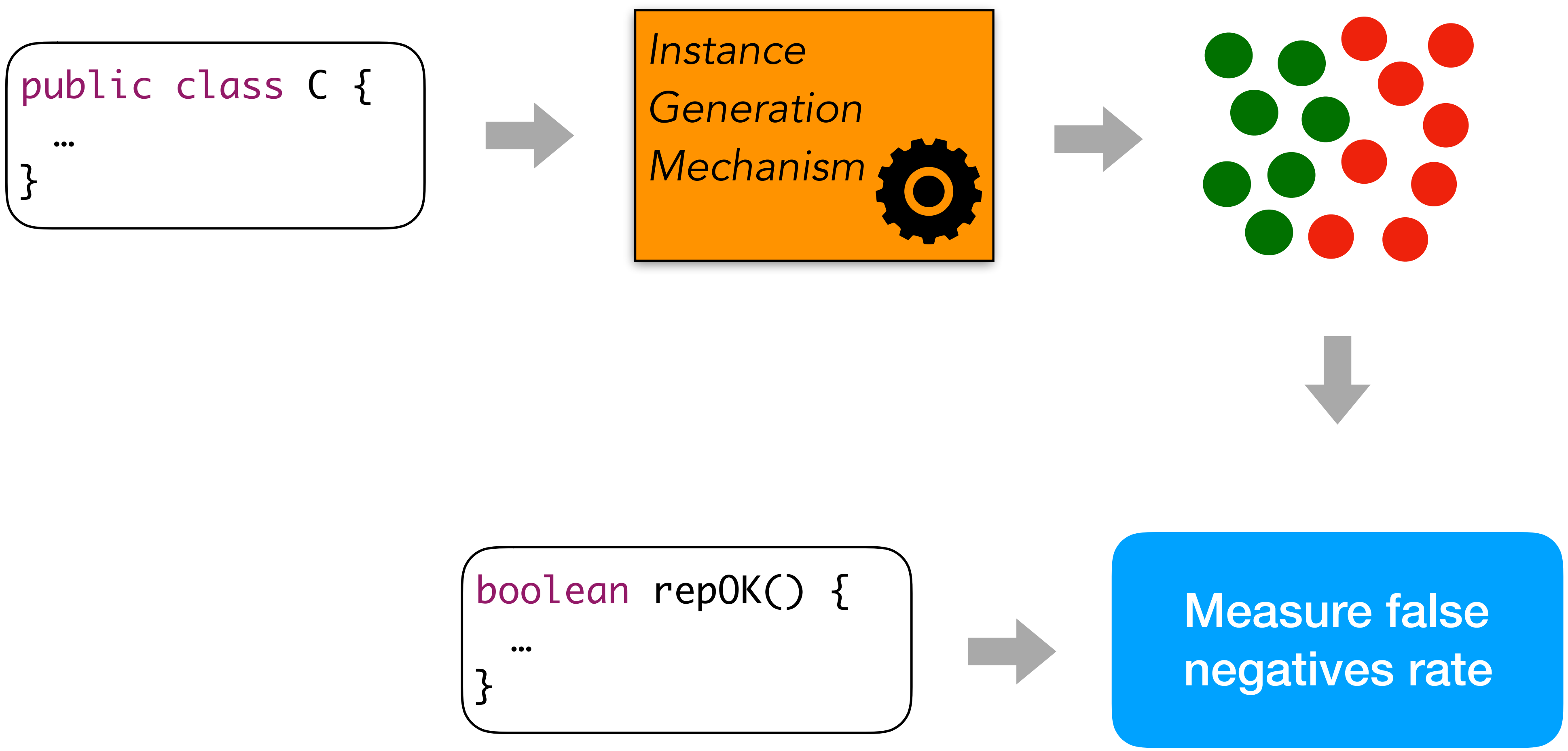
Evaluation

- ***RQ1: is the mechanism for negative instance generation effective?***
- ***RQ2: are neural networks precise in classifying valid/invalid data structure objects?***
- ***RQ3: can automated analysis be improved by the use of NN invariants?***

RQ1: Measure of False Negatives Rate



RQ1: Measure of False Negatives Rate



RQ1: Measure of False Negatives Rate

Singly Sorted List

bound	negative instances	false negatives rate
3	75	6,67 %
4	455	4,18 %
5	2418	2,81 %
6	10306	2,32 %
7	33949	2,24 %
8	83131	2,36 %

Binary Search Tree

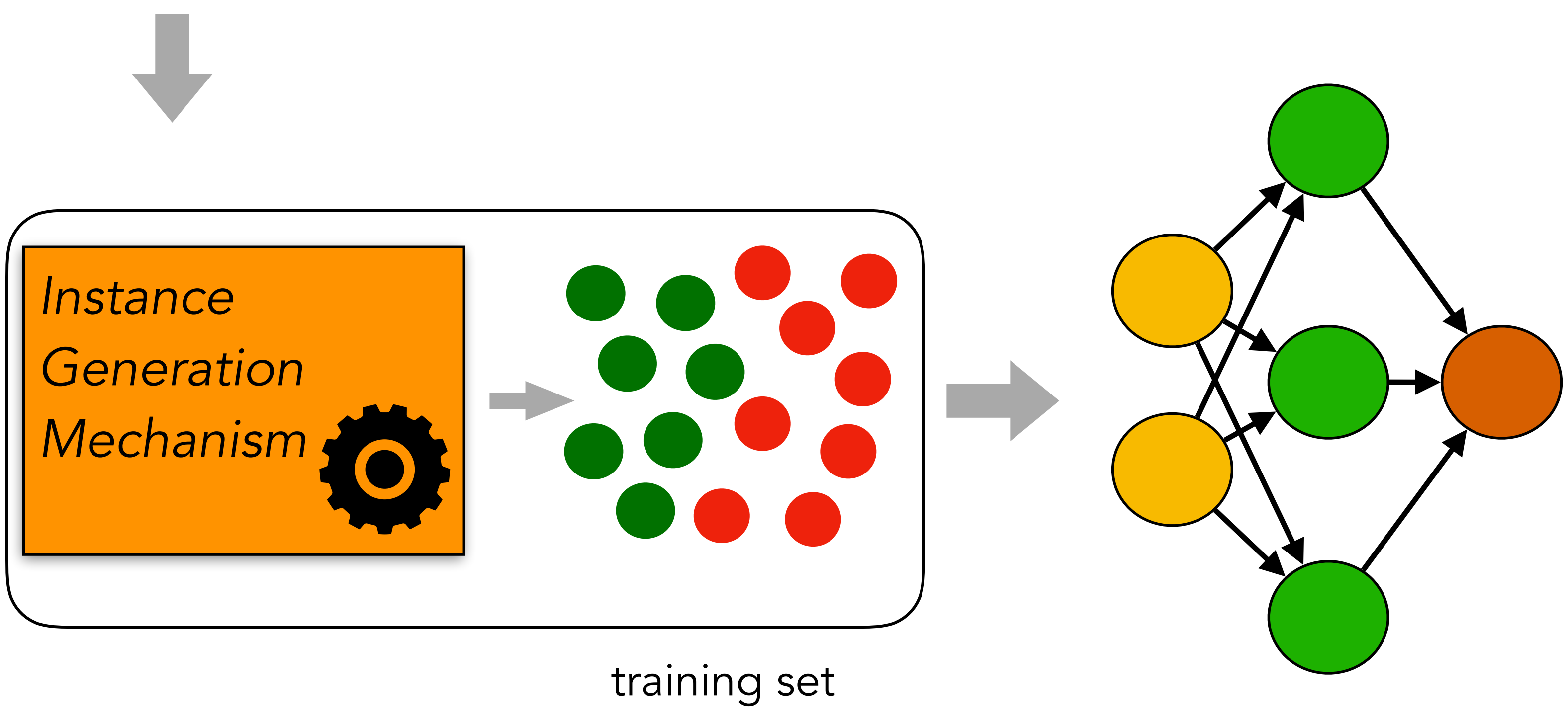
bound	negative instances	false negatives rate
3	200	4,50 %
4	1098	3,19 %
5	5089	0,26 %
6	18445	1,98 %
7	50808	1,65 %
8	106663	1,59 %

Red Black Tree

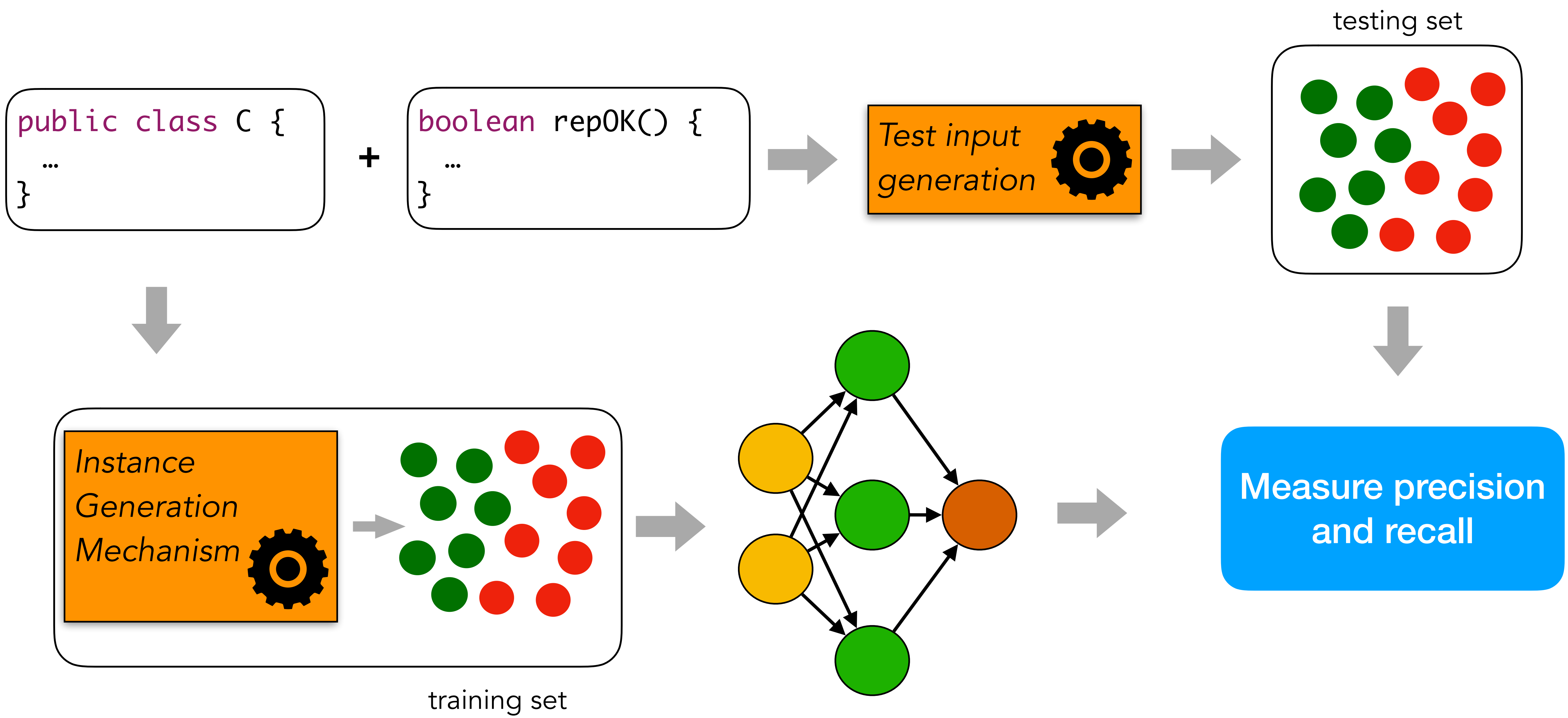
bound	negative instances	false negatives rate
3	144	5,56 %
4	604	3,31 %
5	2377	2,23 %
6	8535	1,53 %
7	26894	1,16 %
8	72099	0,96 %

RQ2: Neural Networks Precision and Recall
Classifying Data Structure Objects

```
public class C {
  …
}
```



RQ2: Neural Networks Precision and Recall Classifying Data Structure Objects



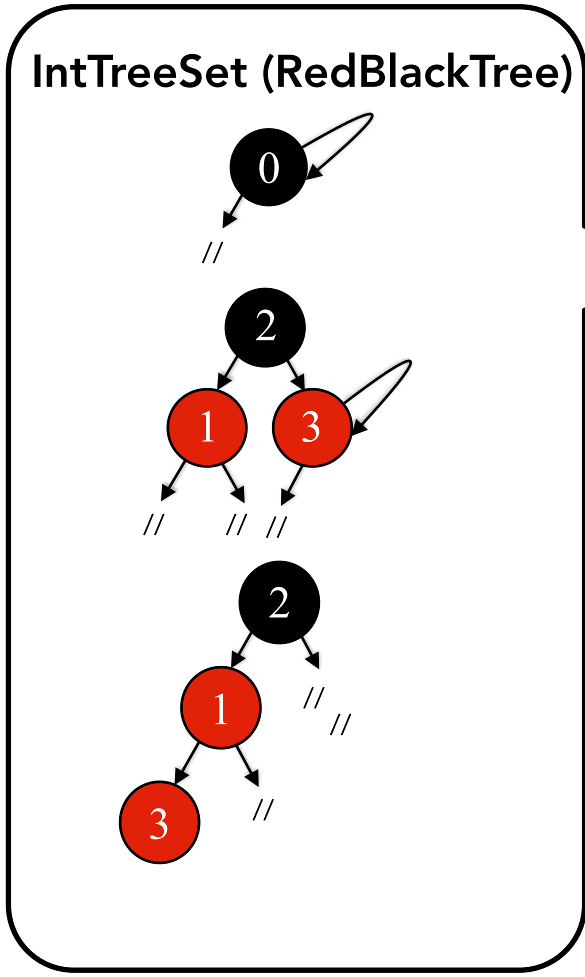
RQ2: Neural Networks Precision and Recall Classifying Data Structure Objects

	Training Set		Validation Set ✗	Validation Set ✓	Precision	Recall
SinglyLinkedList - 7	33456	✗	725966	7	99%	99%
	7867	✓	0	55897	99%	99%
SinglySortedList - 7	33949	✗	1617109	2	99%	99%
	830	✓	51	873	99%	94%
DoublyLinkedList - 7	76574	✗	8914742	8	99%	99%
	9381	✓	1	960800	99%	99%
BinaryTree - 7	21080	✗	17845	3	99%	99%
	567	✓	102	524	99%	83%
RedBlackTree - 7	26894	✗	110834	267	99%	99%
	464	✓	232	679	71%	74%

RQ3: Improved Bug Finding?

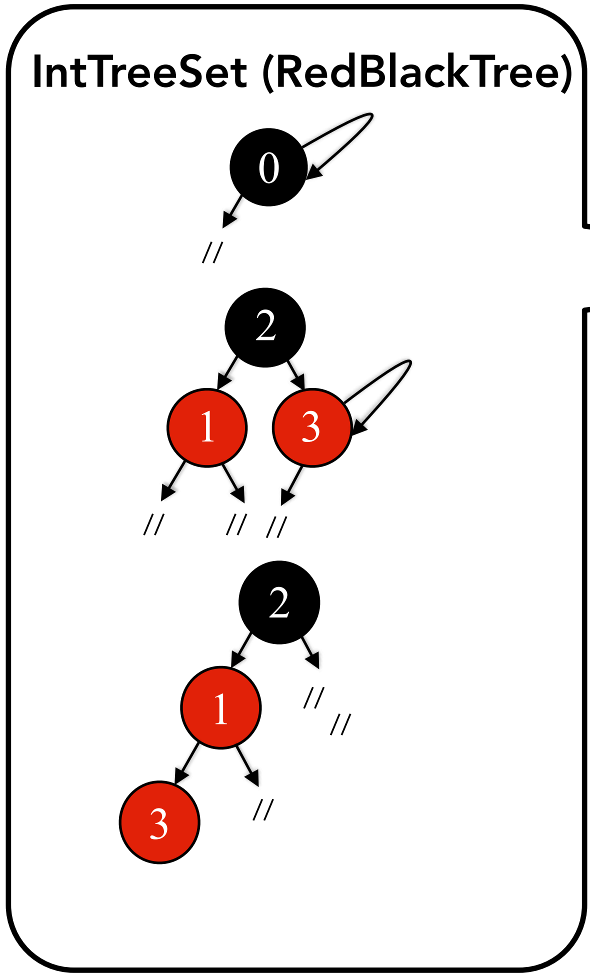
Case study	#bugs	#found without oracle	#found using NN
IntTreeSet	5	0	5
Antlr	1	0	1
FibHeap	1	0	1
BinTree	1	0	1
BinHeap	1	0	1
Schedule	8	3	4

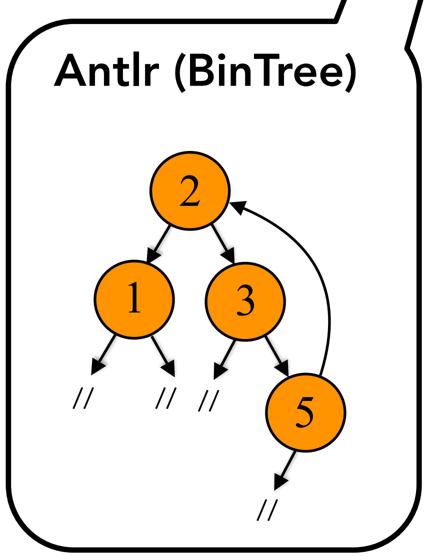
RQ3: Improved Bug Finding?



Case study	#bugs	#found without oracle	#found using NN
IntTreeSet	5	0	5
Antlr	1	0	1
FibHeap	1	0	1
BinTree	1	0	1
BinHeap	1	0	1
Schedule	8	3	4

RQ3: Improved Bug Finding?

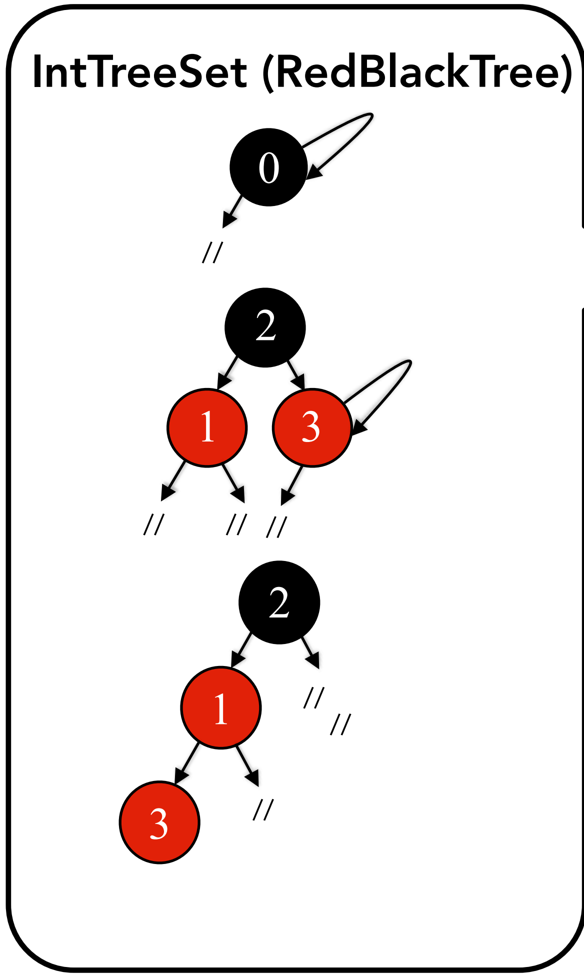


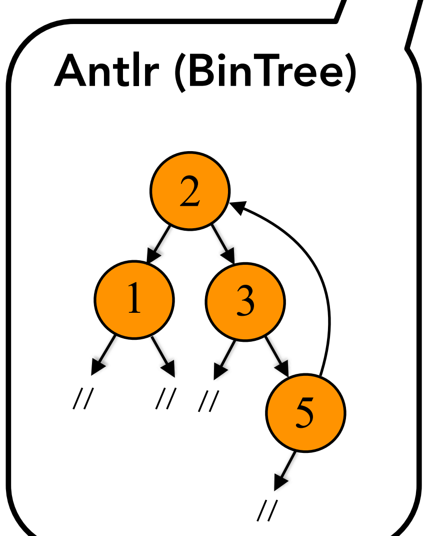


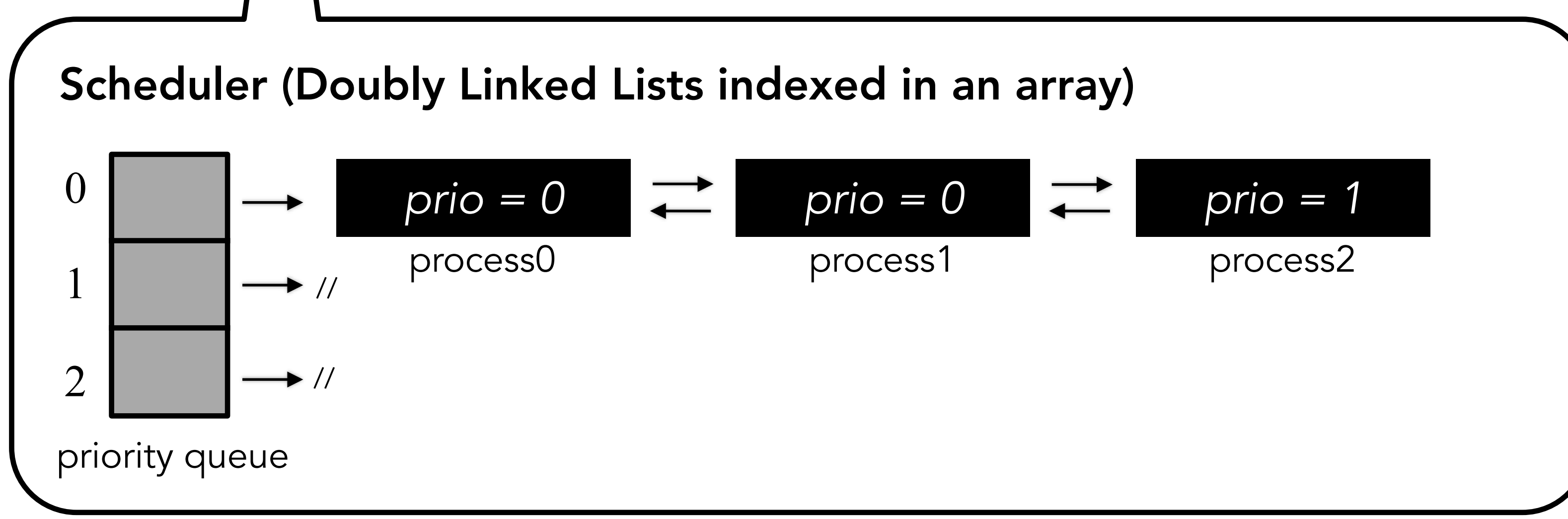
Case study	#bugs	#found without oracle	#found using NN
IntTreeSet	5	0	5
Antlr	1	0	1
FibHeap	1	0	1
BinTree	1	0	1
BinHeap	1	0	1
Schedule	8	3	4

RQ3: Improved Bug Finding?

Case study	#bugs	#found without oracle	#found using NN
IntTreeSet	5	0	5
Antlr	1	0	1
FibHeap	1	0	1
BinTree	1	0	1
BinHeap	1	0	1
Schedule	8	3	4







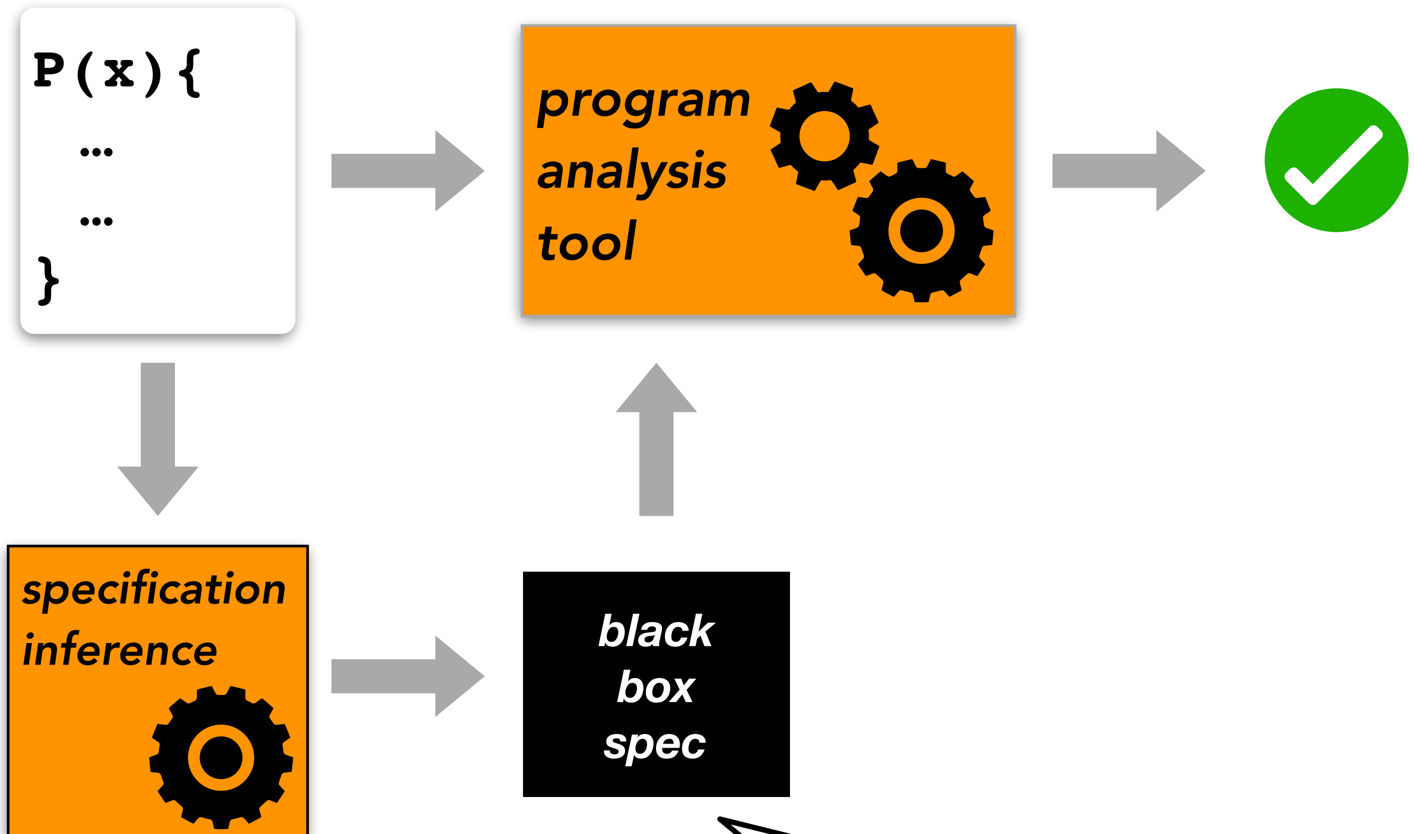
Remarks

- We defined a technique to generate valid and invalid objects from a set of assumed-correct object builders
- We proposed to use the generated objects to train a neural network to distinguish valid from invalid
- We used the trained neural network in place of an invariant for bug finding

Thank you :)

Questions?

Providing Specifications for Program Analysis

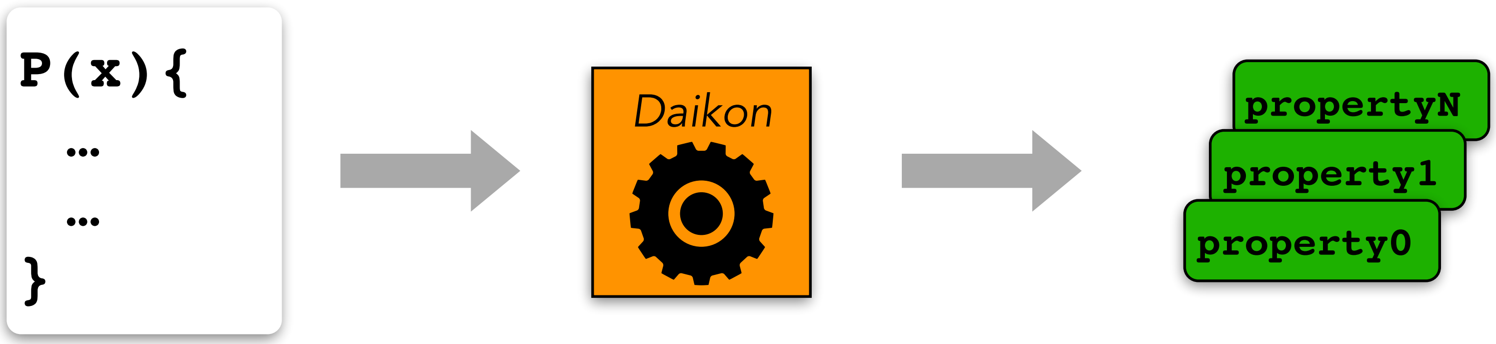


Daikon: derivation of ***likely*** invariants based on different executions of ***P***

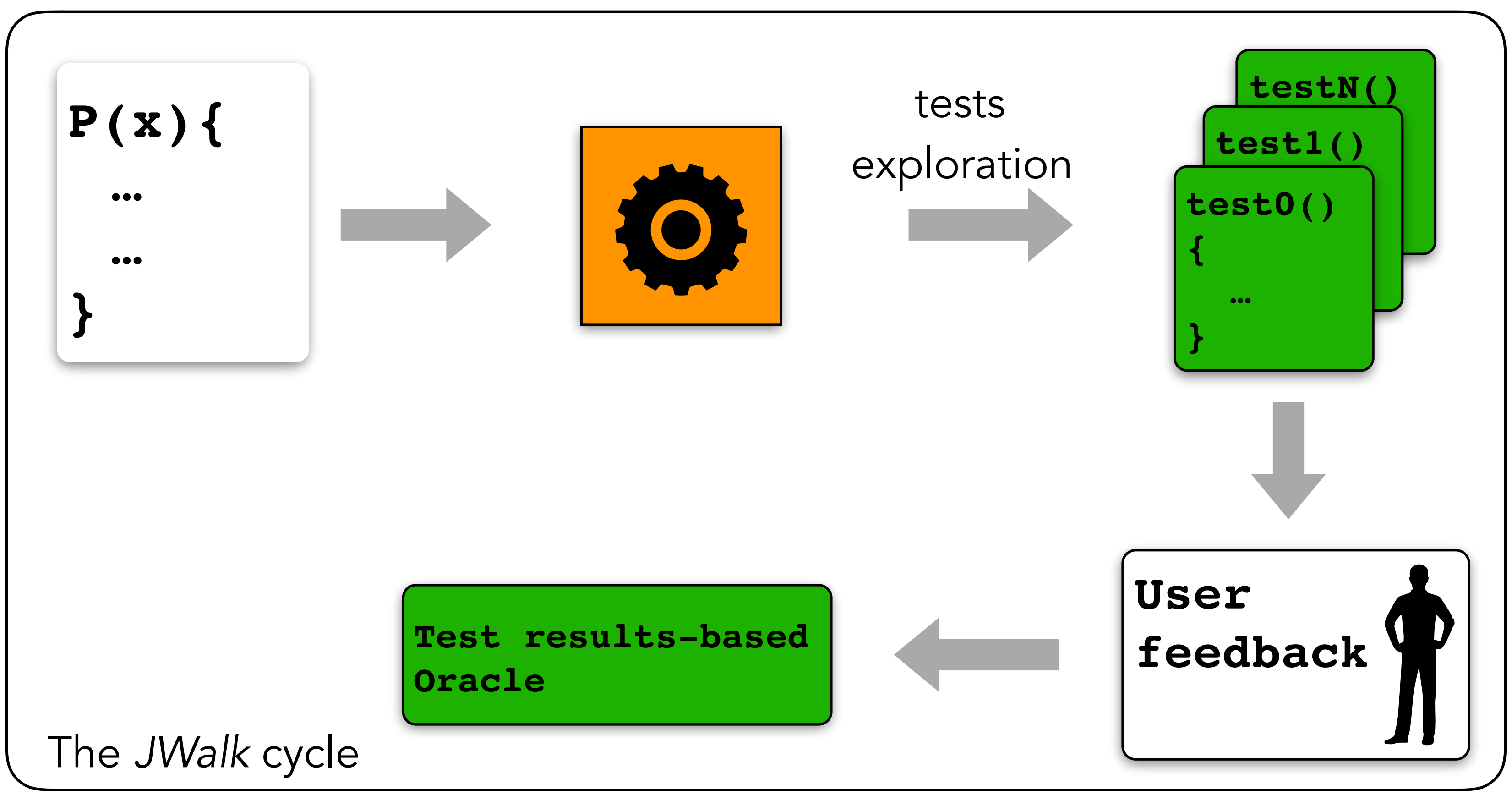
Can be thought as a classifier of correct and incorrect software behaviors

Current approaches to the Lack of Specifications

- Derivation of ***likely*** invariants based on different ***P*** executions



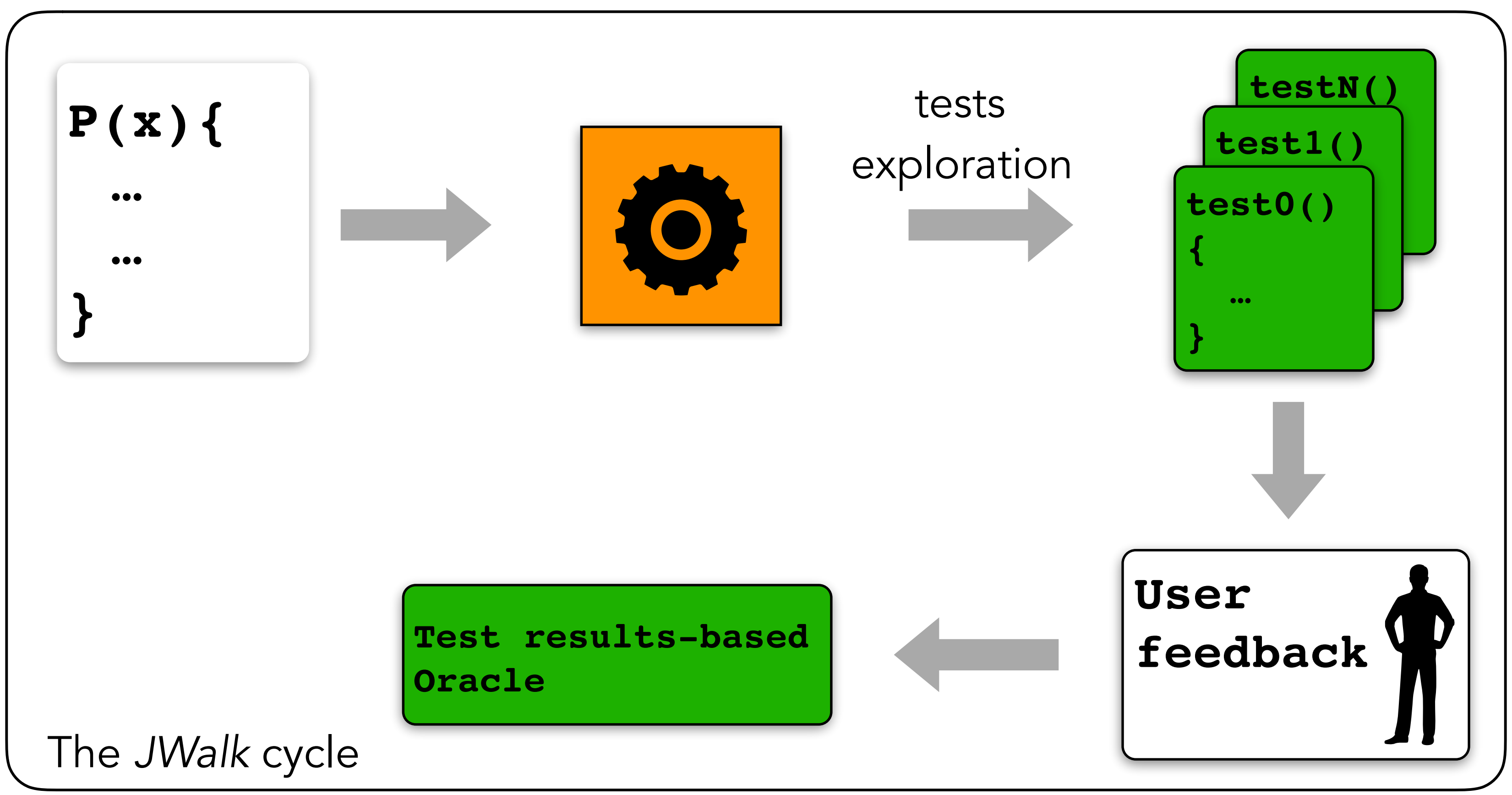
- ***JWalk***: allows dynamic inference of specifications with a feedback-based approach



Current approaches to the Lack of Specifications

- Derivation of ***likely*** invariants based on different ***P*** executions

- ***JWalk***: allows dynamic inference of specifications with a feedback-based approach



✓ human readable properties

✗ complex structural constrains are missed

✗ scenario specific oracles

Daikon in the SinglyLinkedList case

```java
public class SinglyLinkedList {
  private Node header;
  private int size;

  public void SinglyLinkedList() {
    header = new Node(0);
    size = 0;
  }

  public void add(int n) {
    …
  }

}

public class Node {
  private int value;
  private Node next;
  …
}
```

```java
public void test00100() {
public void test002() {
public void test001() {
  SinglyLinkedList l = new SinglyLinkedList();
  l.add(0);
}
```

Daikon

```java
this.header.value == 0;
this.size >= 0;
```

✗ the acyclicity property is missed

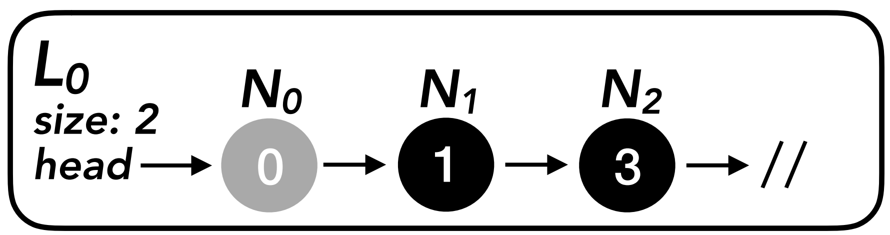
✗ the size property is missed

Precision and Recall vs the True Invariant for scope 7

	Validation Set ✗	Validation Set ✓	Precision	Recall
✗	9	725964	100%	1%
✓	0	55987	7%	100%

Instances as Vectors

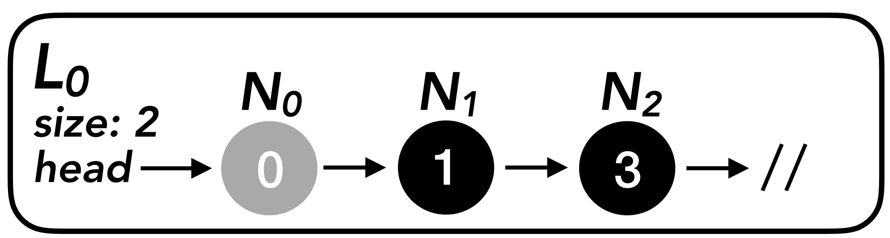
- Given an bound ***k***: use it to define the maximum number of different values for each field
- The vector representation will have as many positions as necessary to represent ***k*** objects/values of each type



Instances as Vectors

- Given an bound ***k***: use it to define the maximum number of different values for each field
- The vector representation will have as many positions as necessary to represent ***k*** objects/values of each type

Example:



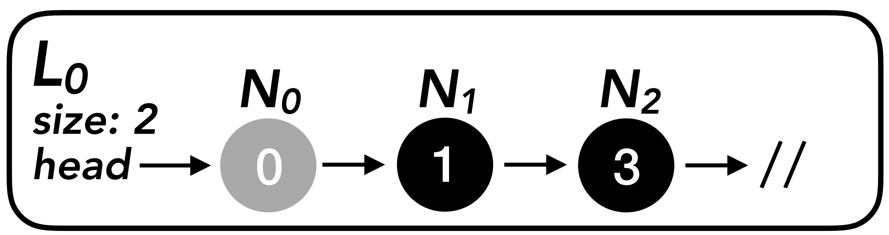
k = *3*

int: 0..3
Node: 0..3

Instances as Vectors

- Given an bound ***k***: use it to define the maximum number of different values for each field
- The vector representation will have as many positions as necessary to represent ***k*** objects/values of each type

Example:



k = *3*

int: 0..3
Node: 0..3

size	head	val_0	$next_0$	val_1	$next_1$	val_2	$next_2$
2	1	0	2	1	3	3	0
L_0		N_0		N_1		N_2	

RQ1: Measure of False Negatives Rate