Efficient Synthesis of Method Call Sequences for Test Generation and Bounded Verification

Yunfan Zhang Ruidong Zhu Yingfei Xiong Tao Xie

PEKING UNIVERSITY

ASE 2022

Outline

1. Background and motivation
2. Algorithm with a running example
3. Evaluation

Background

- Test Generation

```
? ?
void fun1(int x, int y) {
    int u = f(x);
    int v = g(y);
    ...
    if (u < v) {
        // ERROR!
    }
}
```

- Bounded Verification

A bounded input space

```
void fun2(int x, int y) {
    while (...) {
        x = f(x);
        y = g(y);
    }
    ...
    assert(x != y);
}
```

Background

- Test Generation
- Bounded Verification

```
Heap-based data structures: List, Stack, Tree, Graph, ...
```

```
void fun1(T o, int y) {
    T u = o.m1();
    int v = g(y);
    ...
    if (u.m2(v) < 0) {
        // ERROR!
    }
}
```

```
void fun2(T o, int y) {
    while (...) {
        o = o.m1();
        y = g(y);
    }
    ...
    assert(o.m2(y) != 0);
}
```

How to determine the existence of, and further construct, an input heap state that satisfies a given specification?

A Simple Java Class

```
class Node {
   private Node next;
   private int value;
   private Node(Node n, int v) {
      this.next = n;
      this.value = v;
   }

   public static Node create(int v, boolean b) {
      if (b == true)
         return new Node(null, v * 2);
      else return new Node(null, v * 2 + 1);
   }
   ...
```

fields

constructor

static factory method

```
   ...
   public Node getNext() {
      return this.next;
   }
   public int getValue() {
      return this.value;
   }
   public void addAfter(int v) {
      this.next = new Node(null, v);
   }
   public Node addBefore(int v) {
      return new Node(this, v);
   }
}
```

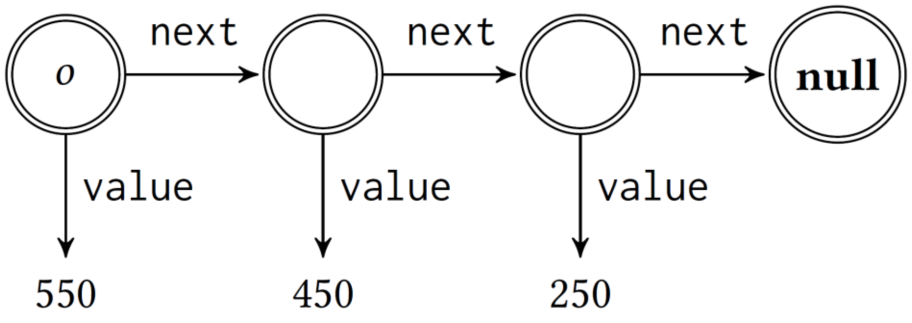
instance methods

A Simple Specification

```
class Node {
    private Node next;
    private int value;
    ...
}
```

```
boolean TEST(Node o) {
   return (o.value – o.next.value == 100) &&
      (o.next.value – o.next.next.value == 200) &&
      (o.value + o.next.next.value == 800);
}
```

A specification



A solution

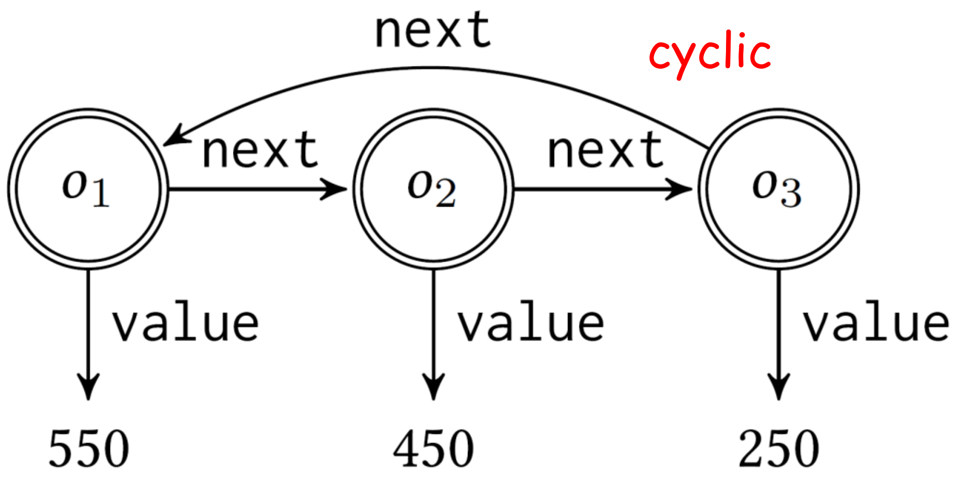
```
550 – 450 = 100
450 – 250 = 200
550 + 250 = 800
```

Existing Work

```
class Node {
    private Node next;
    private int value;
    ...
}
```

- Direct construction approaches
  - directly assign values to the fields of the heap objects
  - *Korat* [Boyapati et al. 2002], *JBSE* [Braione et al. 2015], ...

```
Node o1 = new Node(...);
Node o2 = new Node(...);
Node o3 = new Node(...);
o1.next = o2; o1.value = 550;
o2.next = o3; o2.value = 450;
o3.next = o1; o3.value = 250;
```



violate the accessibility rules

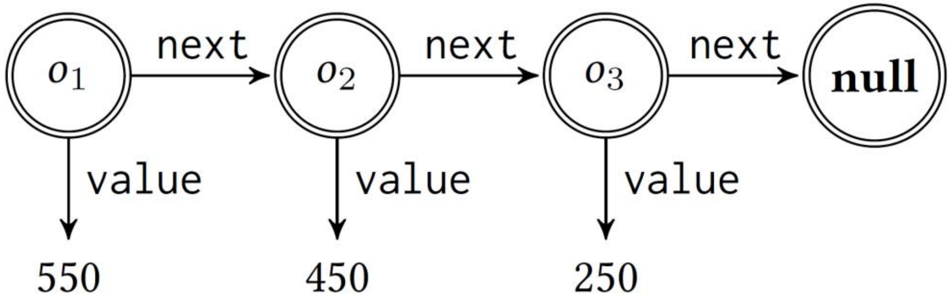
produce invalid/unreachable heap states

Existing Work

```
class Node {
    ...
    public static Node create(int v, boolean b);
    public Node addBefore(int v);
}
```

- Sequence generation approaches
  - synthesize and execute a sequence of calls to the public methods
  - *Seeker* [Thummalapenta et al. 2011], *SUSHI* [Braione et al. 2017], ...

```
Node o3 = Node.create(125, true);
Node o2 = o3.addBefore(450);
Node o1 = o2.addBefore(550);
```



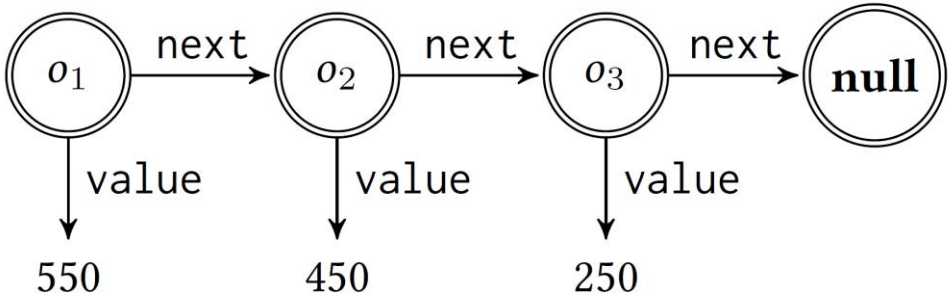
SUSHI, the previous state-of-the-art approach, fails to synthesize such a method call sequence within 10 hours

Existing Work

```
class Node {
   ...
   public static Node create(int v, boolean b);
   public Node addBefore(int v);
}
```

- Sequence generation approaches
  - synthesize and execute a sequence of calls to the public methods
  - *Seeker* [Thummalapenta et al. 2011], *SUSHI* [Braione et al. 2017], ...

```
Node o3 = Node.create(125, true);
Node o2 = o3.addBefore(450);
Node o1 = o2.addBefore(550);
```

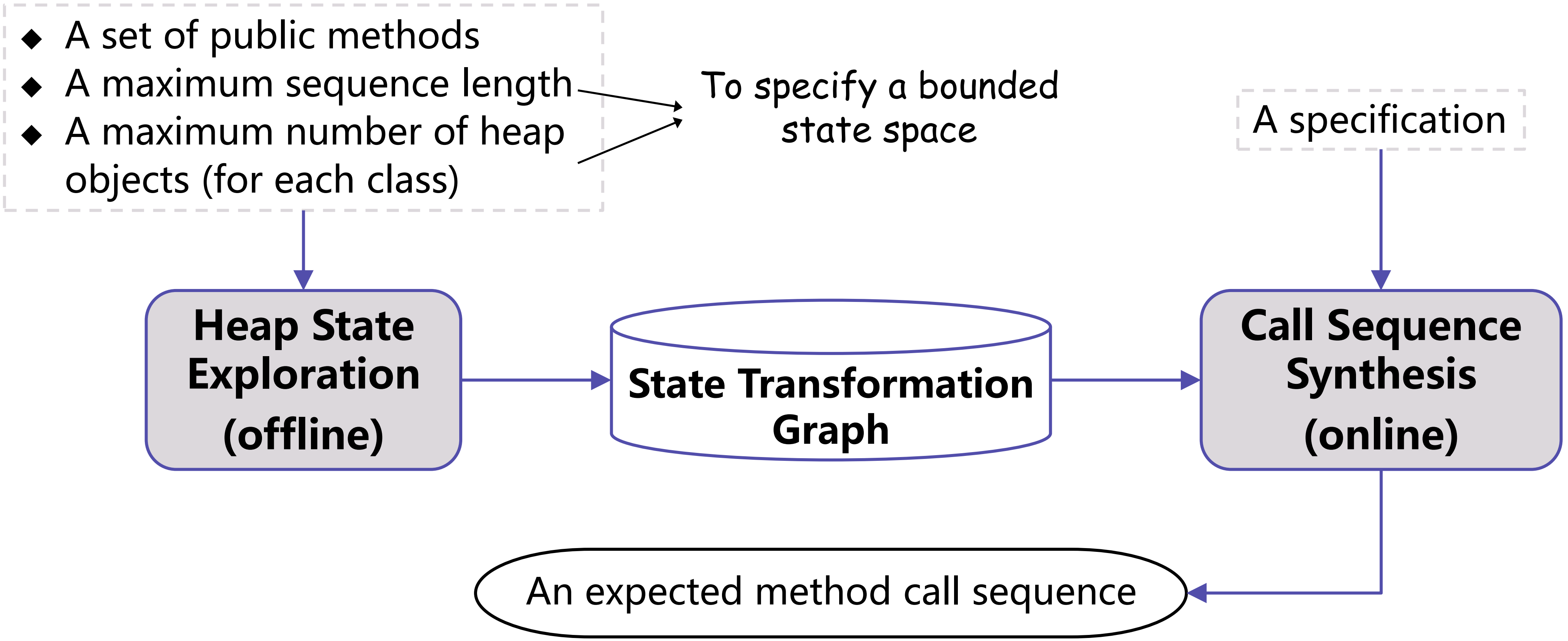


Motivation & Contribution: developing an efficient synthesis algorithm for method call sequences

Outline

1. Background and motivation
2. Algorithm with a running example
3. Evaluation

Workflow

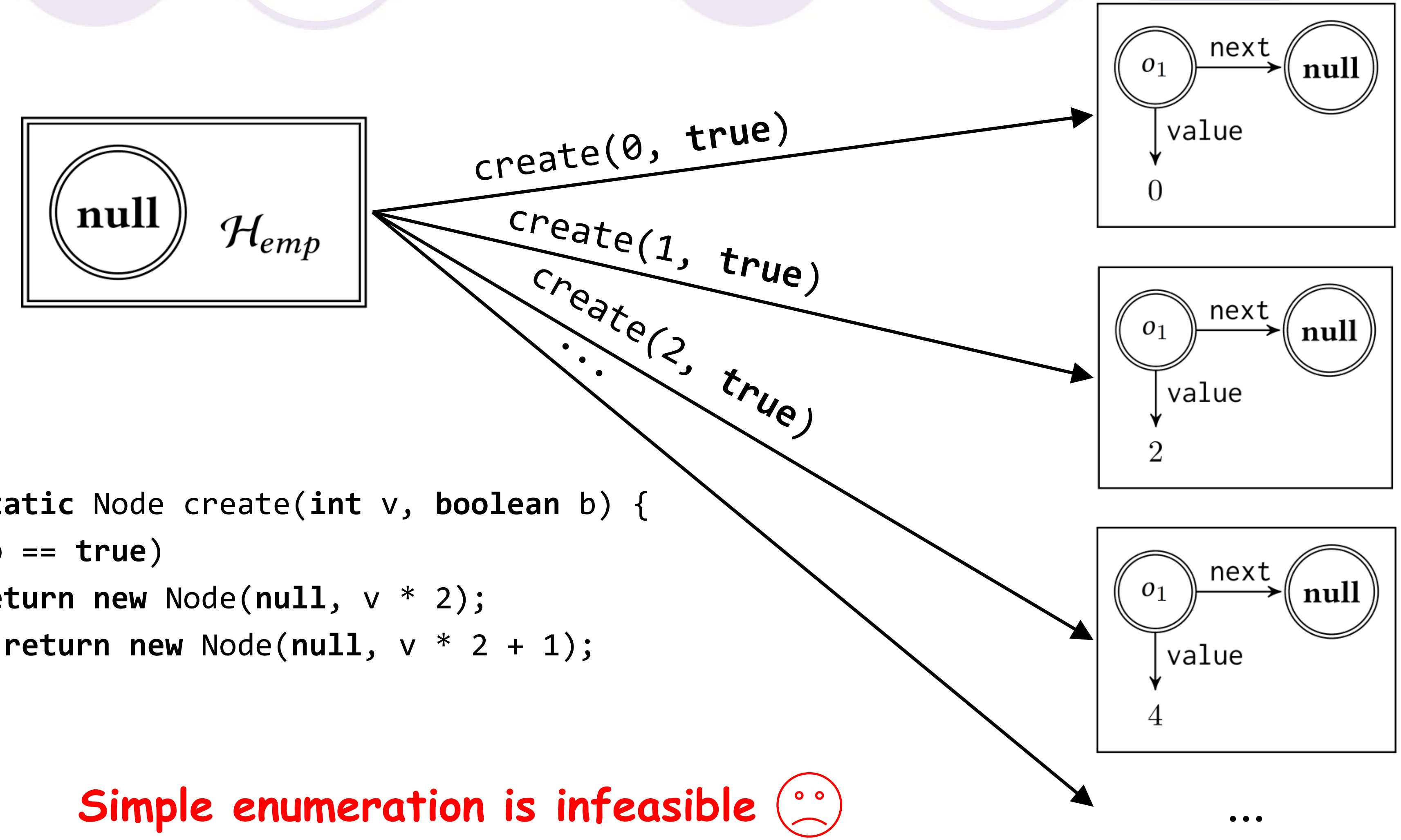


A set of public methods
A maximum sequence length
A maximum number of heap objects (for each class)
To specify a bounded state space
A specification
Heap State Exploration (offline)
State Transformation Graph
Call Sequence Synthesis (online)
An expected method call sequence

Heap State Exploration


```
class Node {
   ...
   public static Node create(int v, boolean b)
      { ... }
   public Node getNext() { ... }
   public int getValue() { ... }
   public void addAfter(int v) { ... }
   public Node addBefore(int v) { ... }
}
```

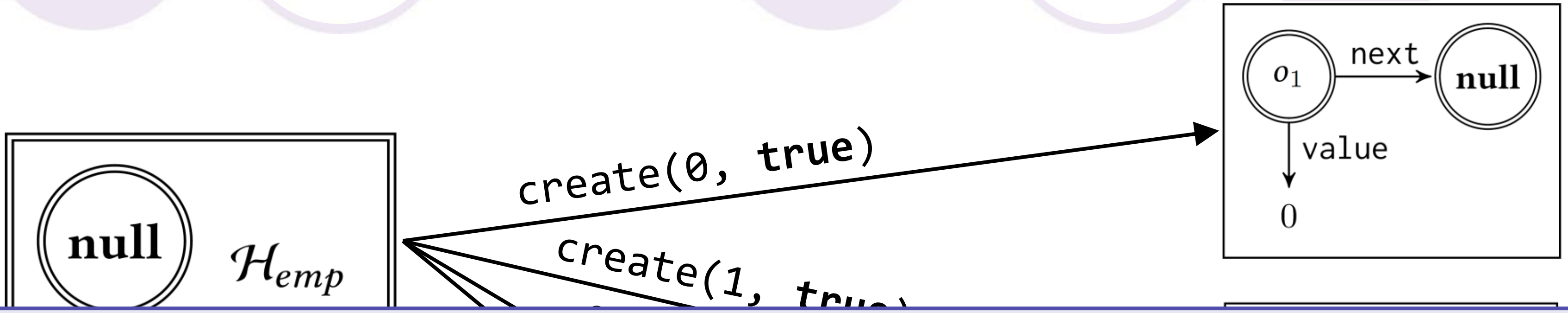
Heap State Exploration




```
public static Node create(int v, boolean b) {
    if (b == true)
       return new Node(null, v * 2);
    else return new Node(null, v * 2 + 1);
}
```

Simple enumeration is infeasible ☹

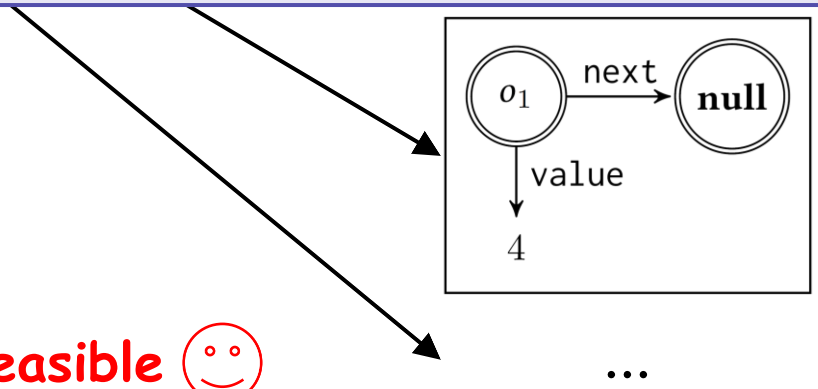
Heap State Exploration



A heap state = a heap structure (objects & references) + primitive values

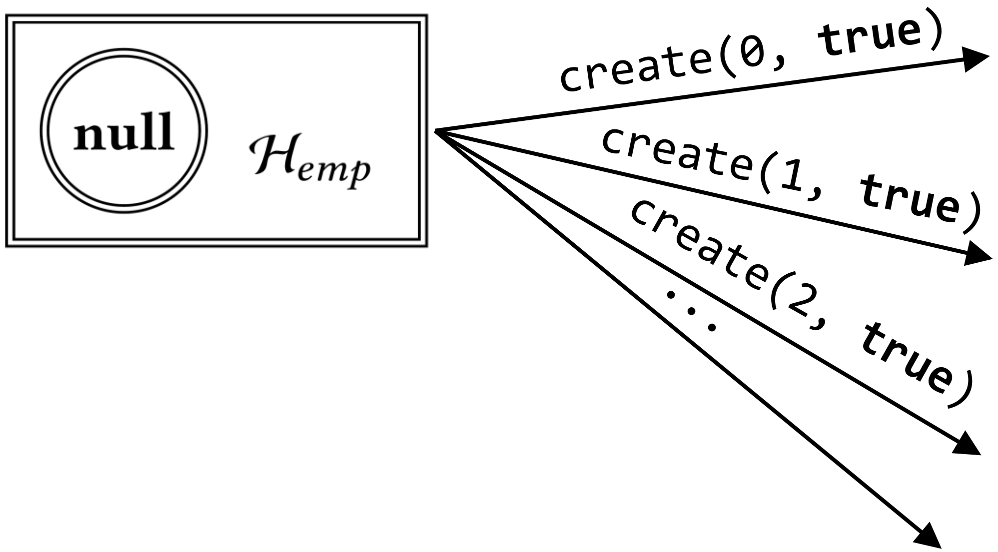
- The space of the former is relatively <u>small</u>, and can be enumerated.
- The space of the latter is large, but can be inferred using a <u>constraint solver</u>.

```
publi
    if (b == true)
        return new Node(null, v * 2);
    else return new Node(null, v * 2 + 1);
}
```

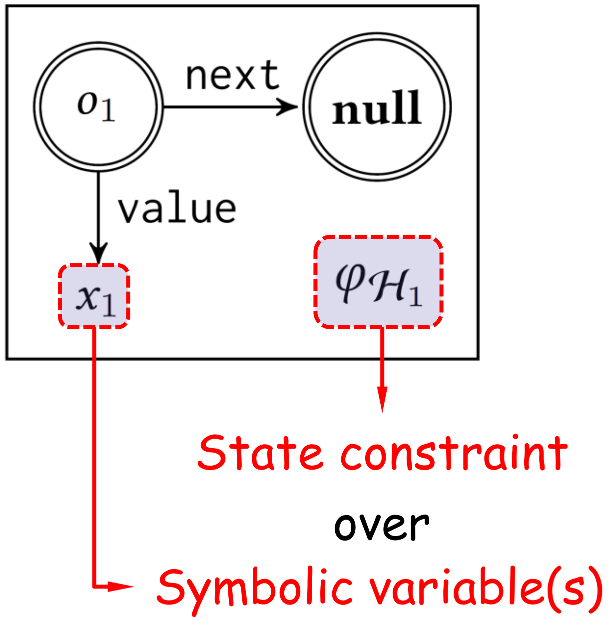


Enumeration of the heap structures is feasible ☺

State Abstraction

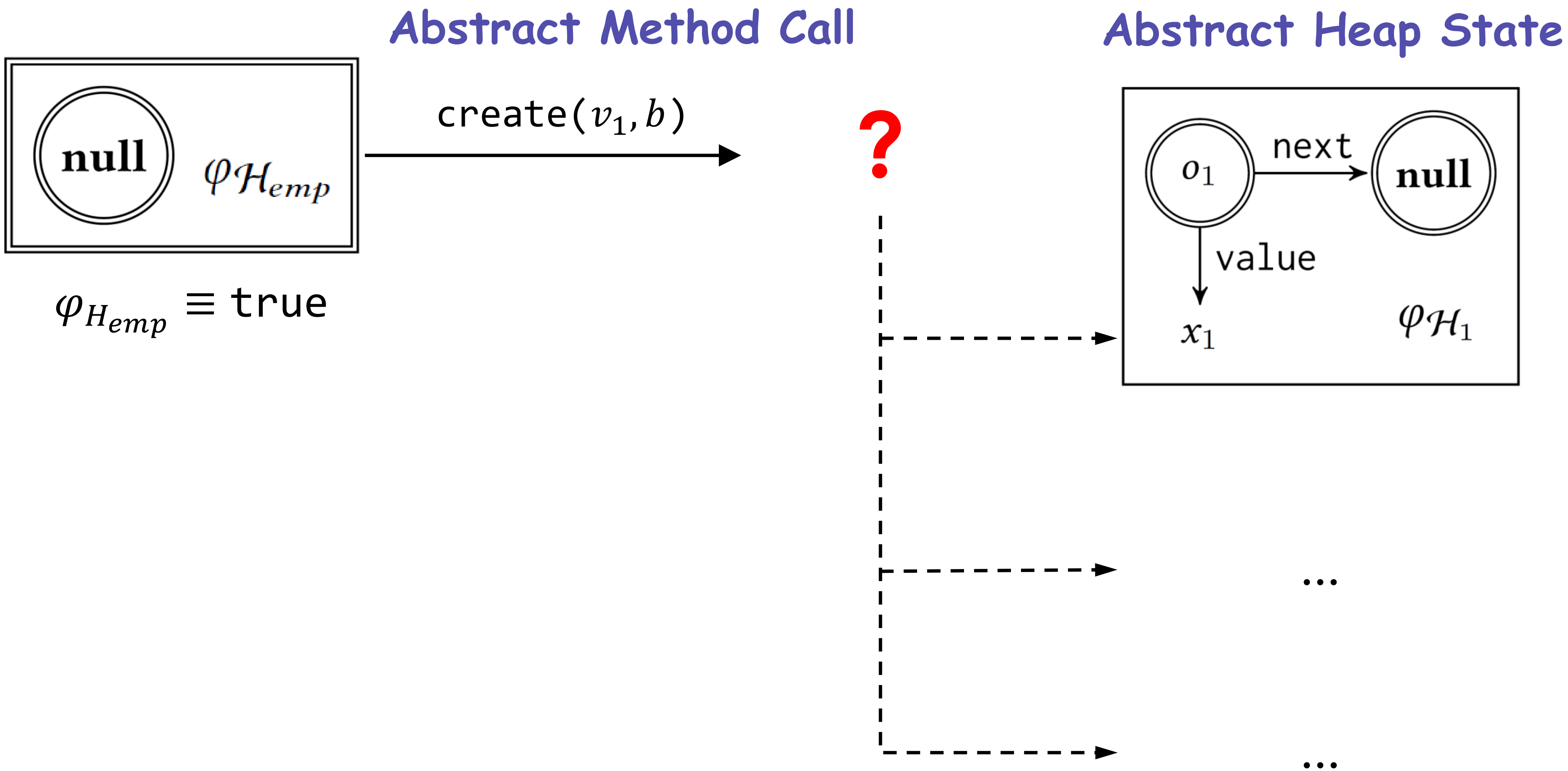


Abstract Heap State

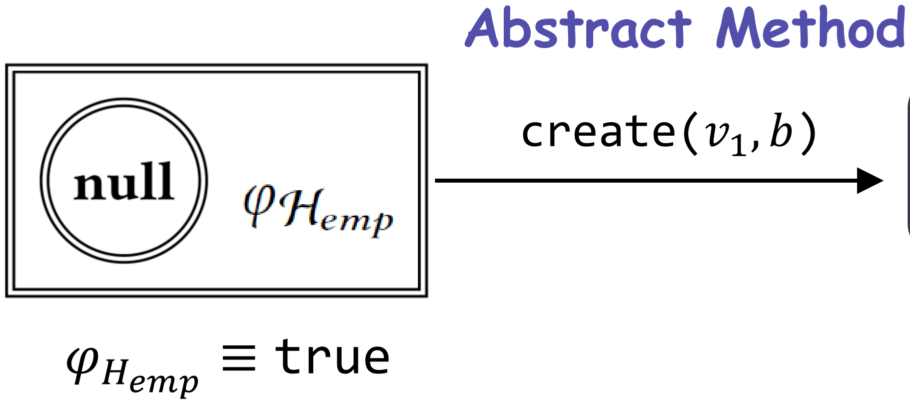


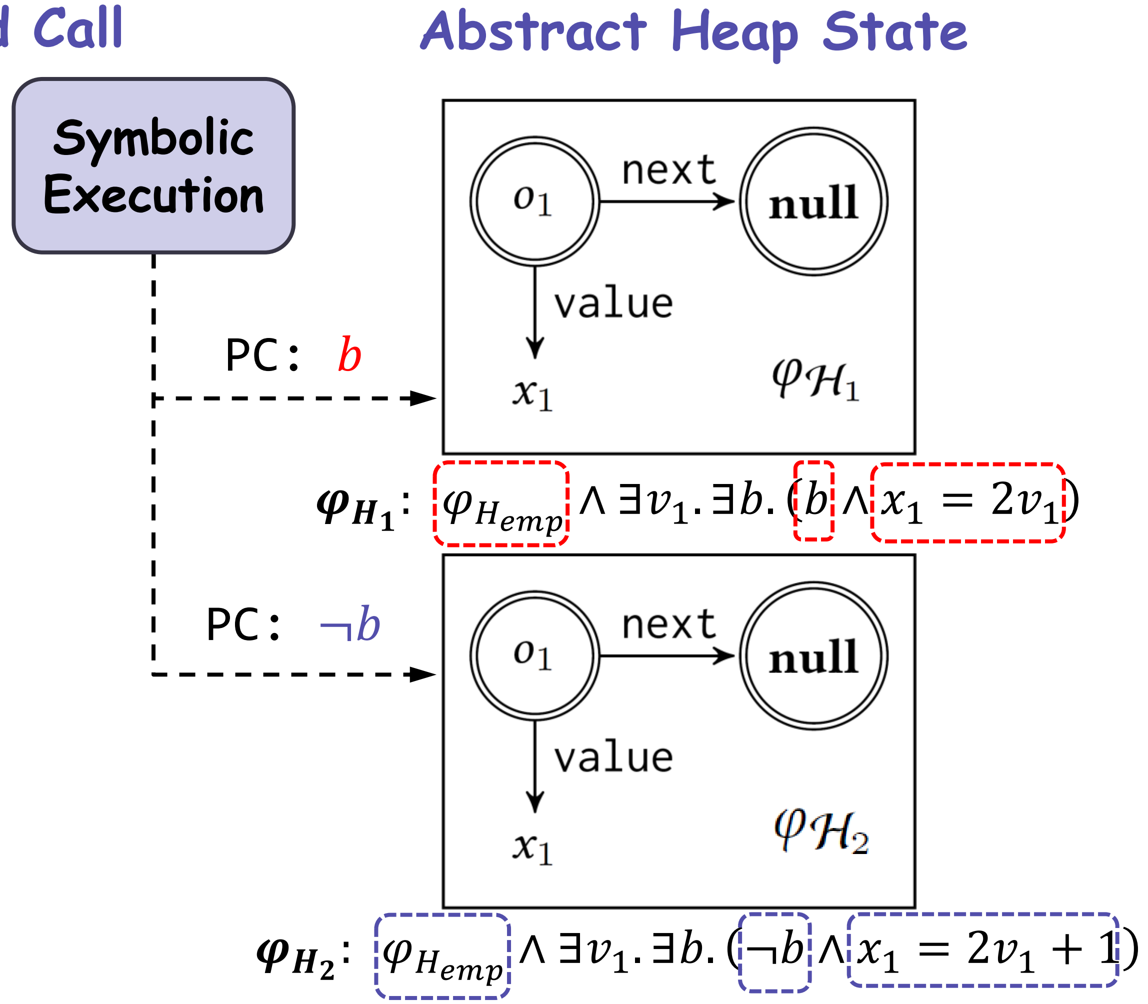
e.g., $\exists v_1 . \exists b . (b \wedge x_1 = 2v_1)$

State Abstraction



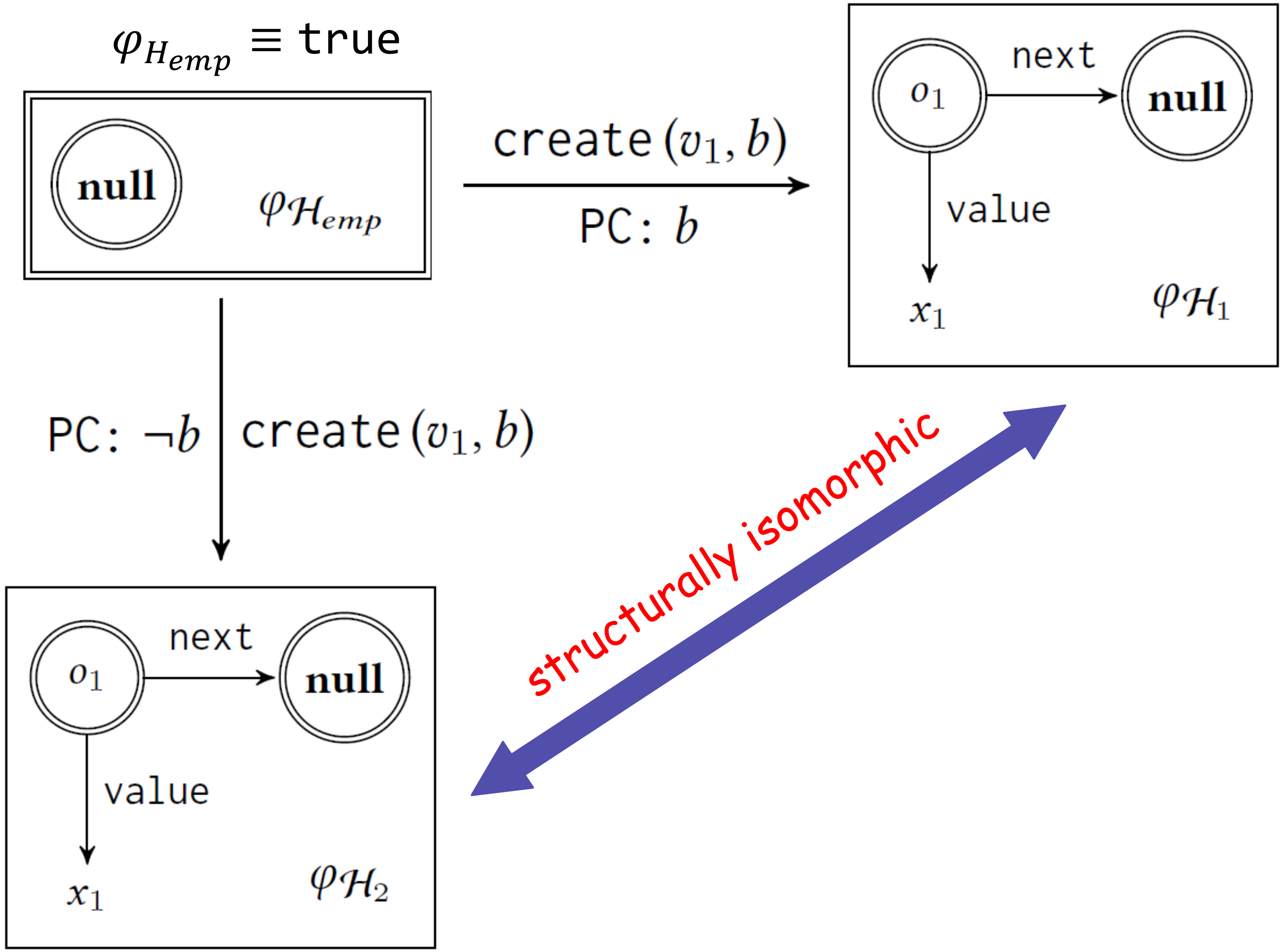
State Abstraction



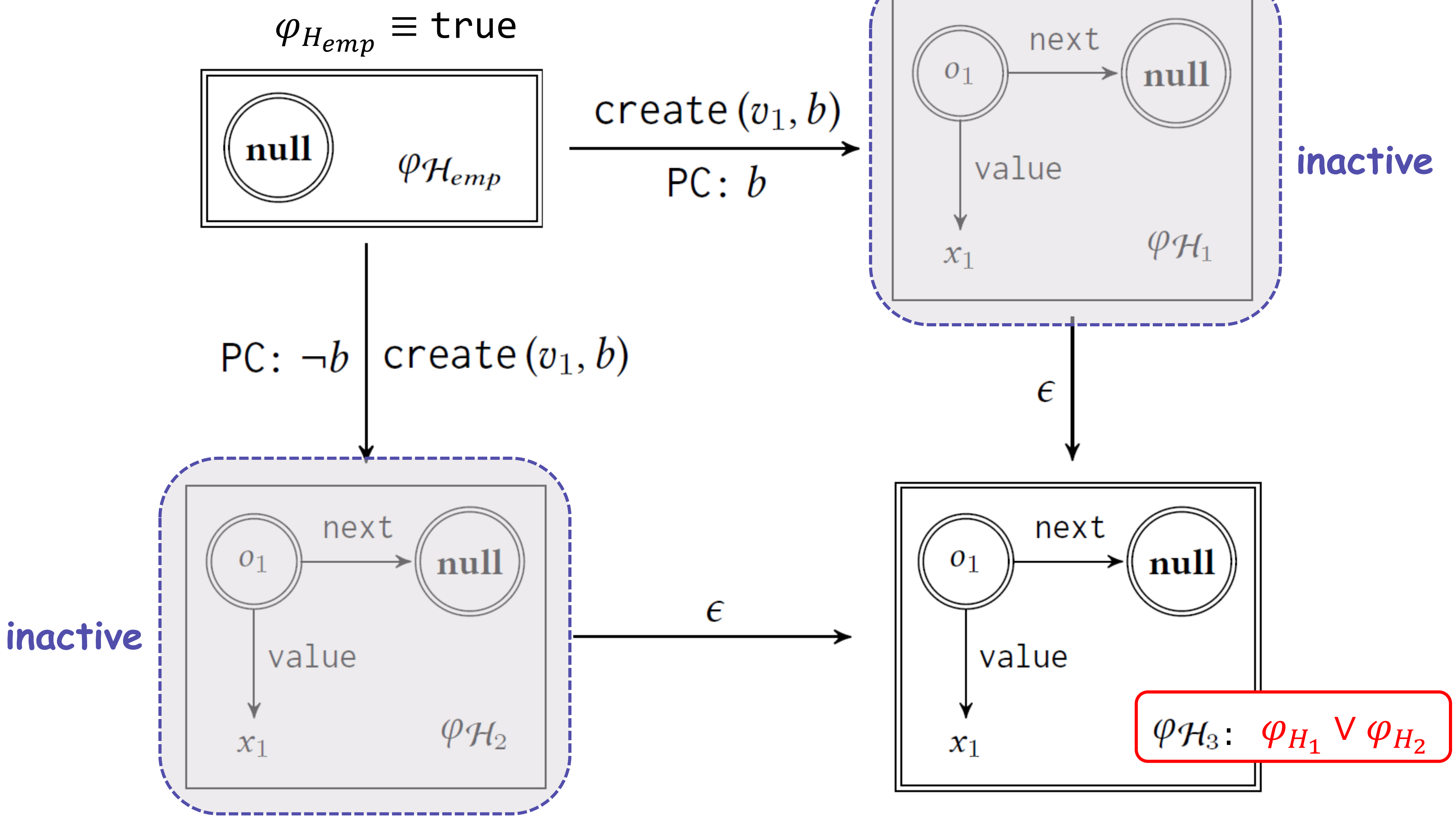



```
public static Node create(int v, boolean b) {
    if (b == true)
        return new Node(null, v * 2);
    else return new Node(null, v * 2 + 1);
}
```

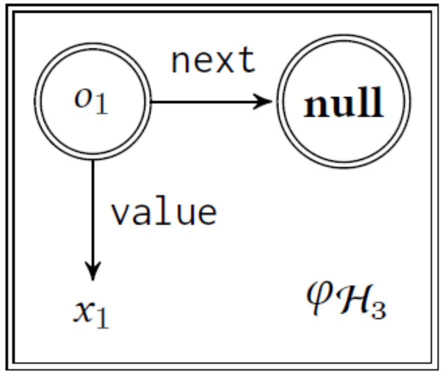
Structural Isomorphism



State Merging

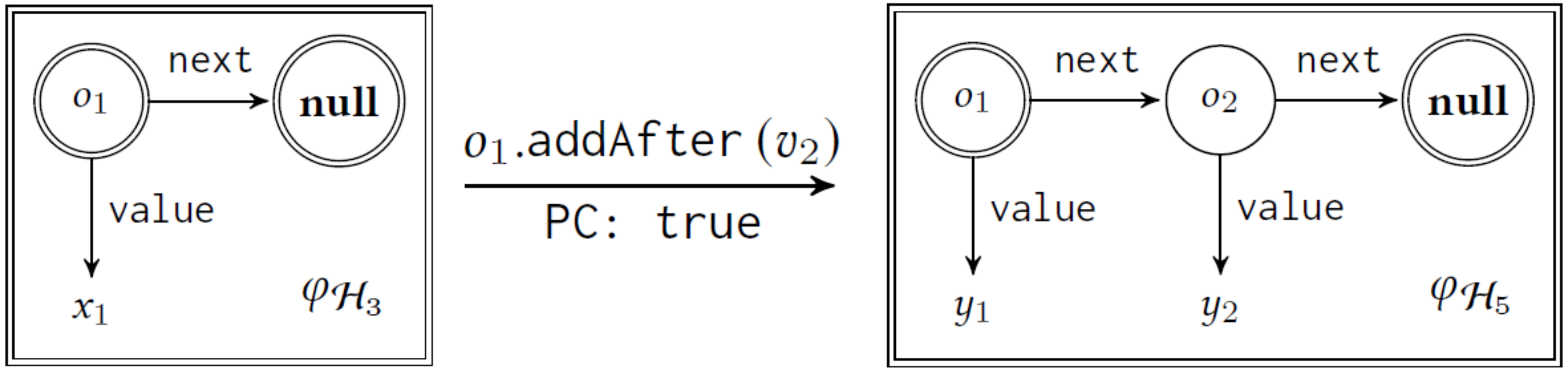


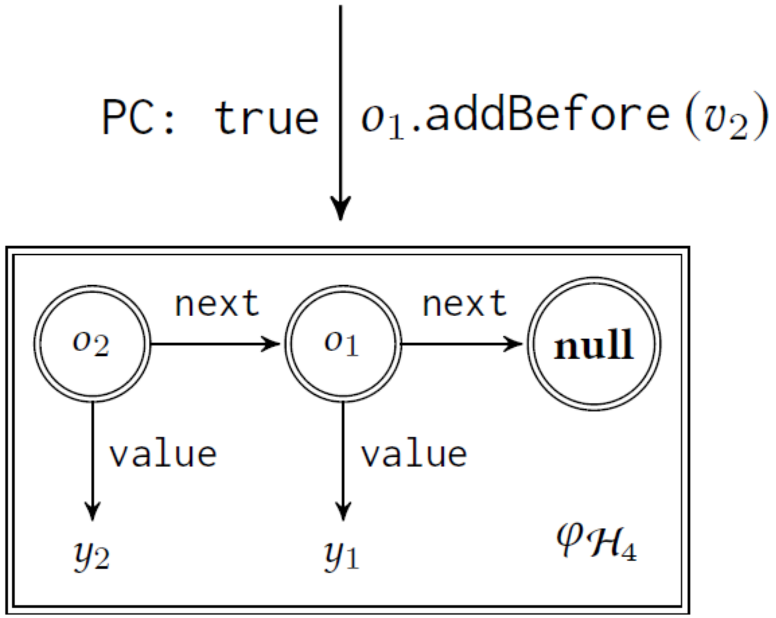
Heap State Exploration




```
public void addAfter(int v) {
   this.next = new Node(null, v);
}
public Node addBefore(int v) {
   return new Node(this, v);
}
```

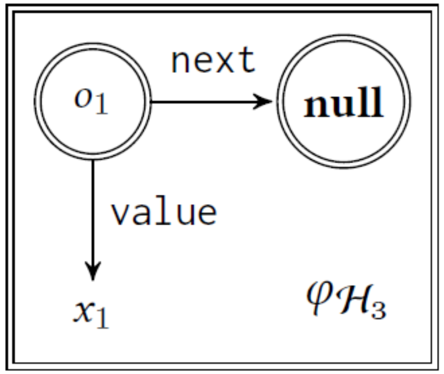
Heap State Exploration





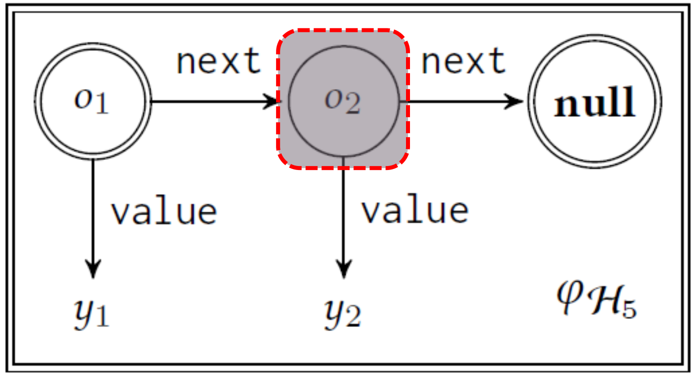

```java
public void addAfter(int v) {
   this.next = new Node(null, v);
}
public Node addBefore(int v) {
   return new Node(this, v);
}
```

Heap State Exploration



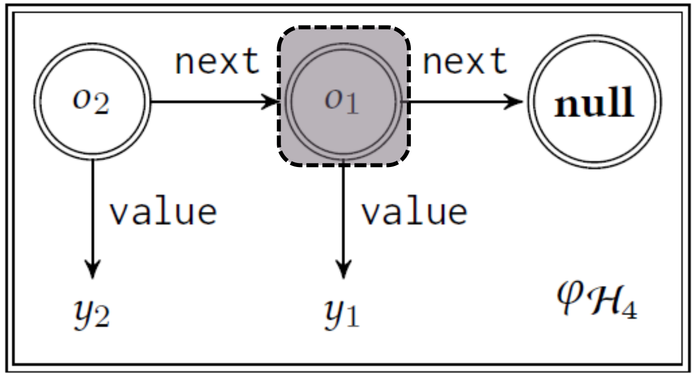
o_1.addAfter (v_2)

PC: true



not stack-accessible

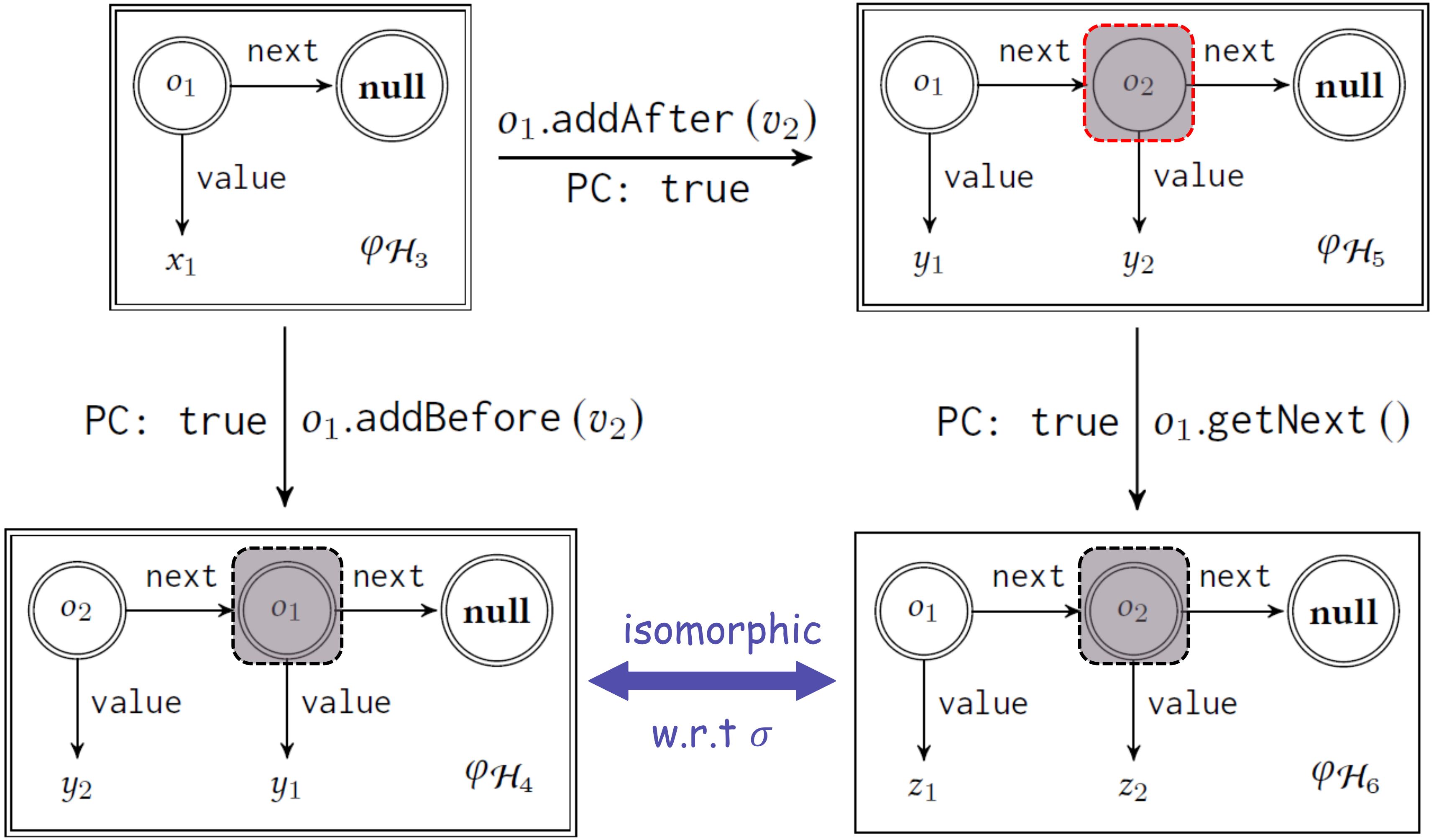
PC: true | o_1.addBefore (v_2)



stack-accessible

```
public void addAfter(int v) {
    this.next = new Node(null, v);
}
public Node addBefore(int v) {
    return new Node(this, v);
}
```

Heap State Exploration



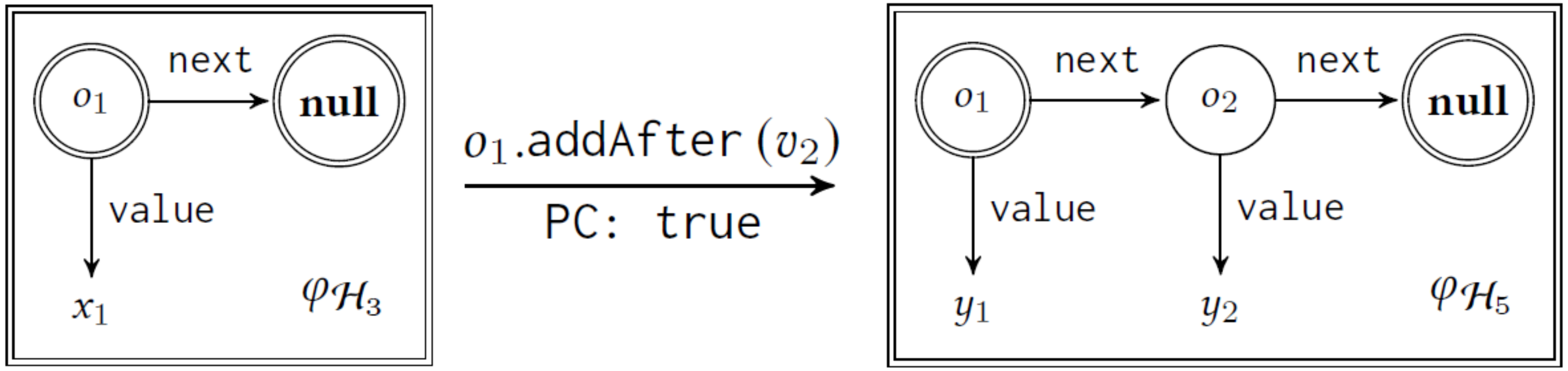
not stack-accessible

```
public Node getNext() {
    return this.next;
}
```

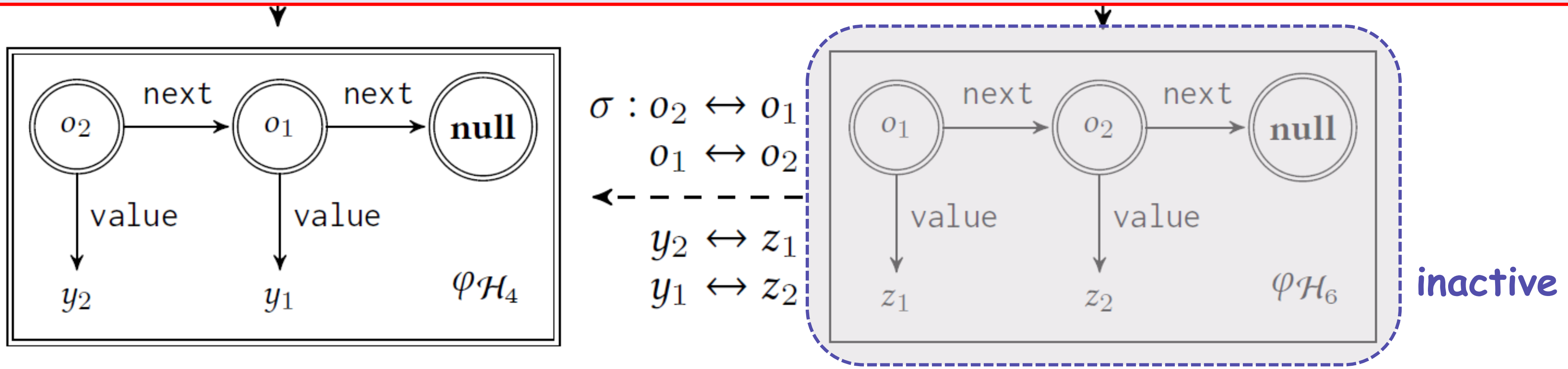
stack-accessible

stack-accessible

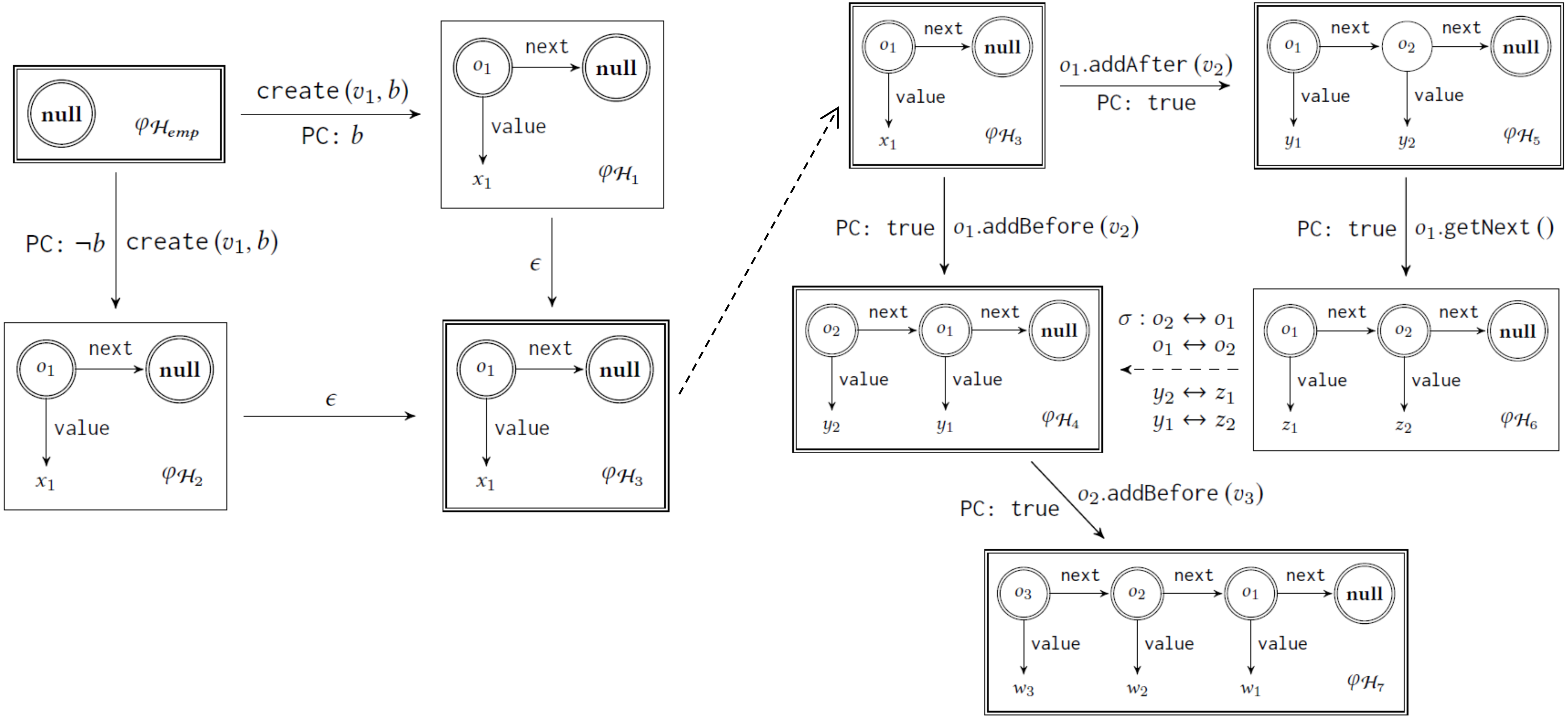
Heap State Exploration



All concrete heap states represented by H_6 are included in those represented by H_4!
(by checking $\varphi_{H_6} \rightarrow \varphi_{H_4}[y_2 \coloneqq z_1, y_1 \coloneqq z_2]$ holds for all z_1, z_2)

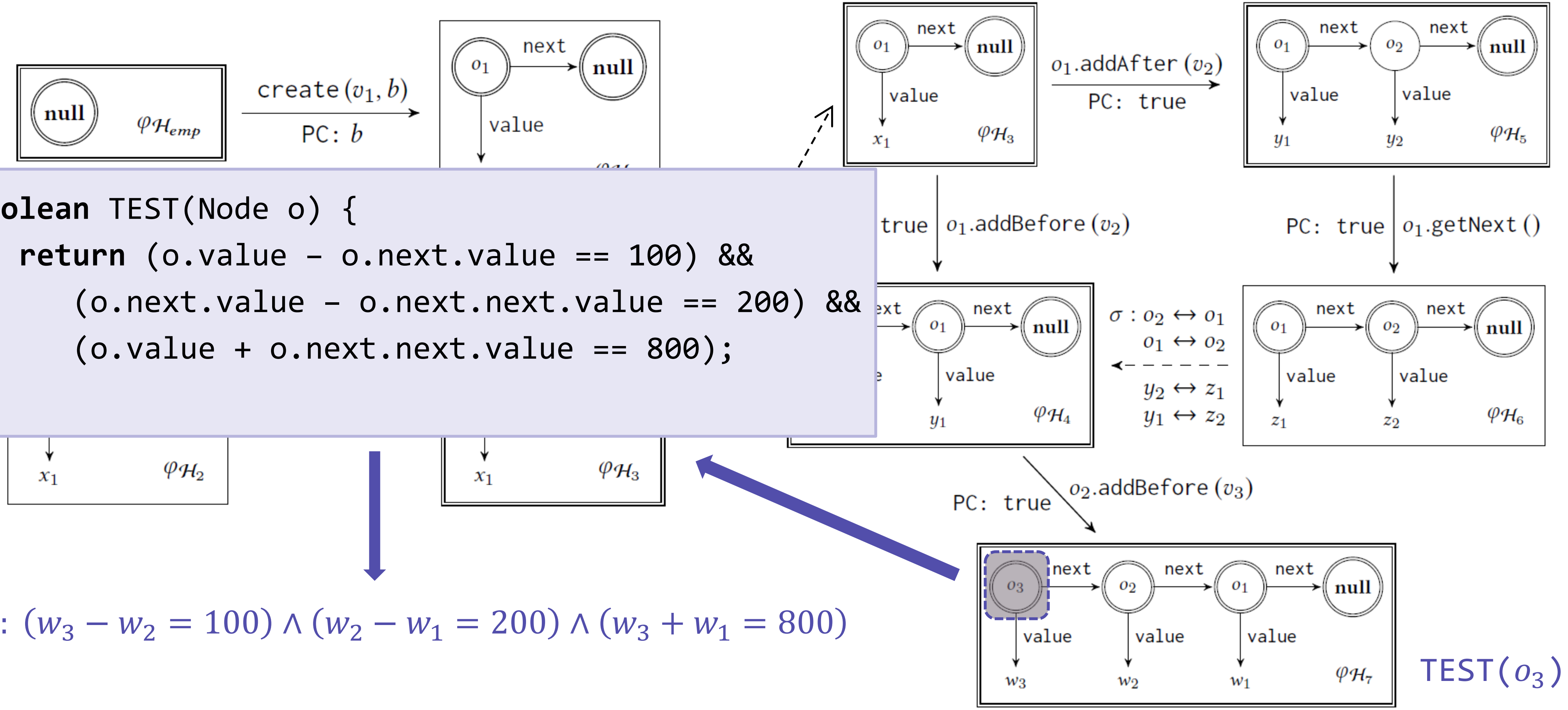


State Transformation Graph



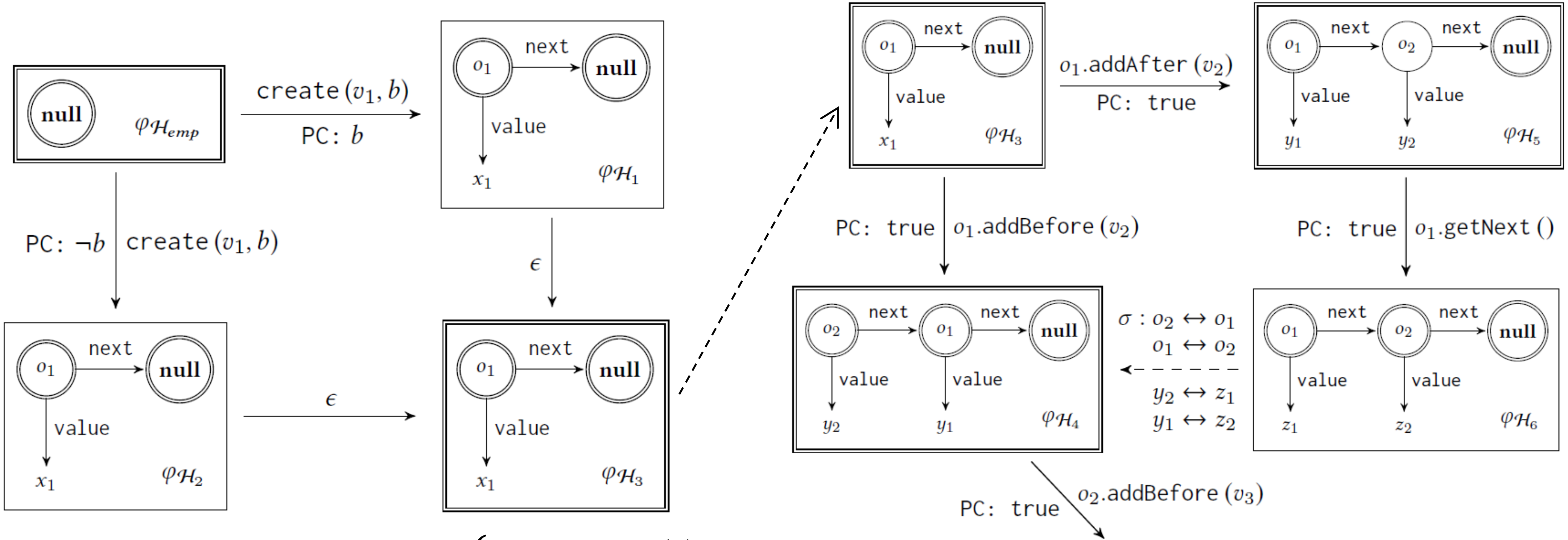
Call Sequence Synthesis

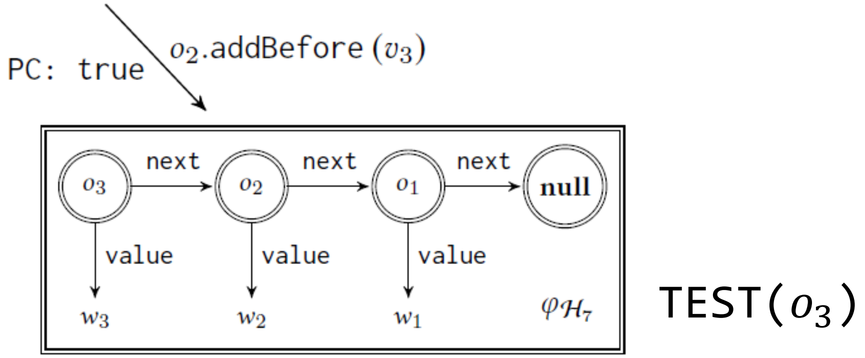
```
boolean TEST(Node o) {
   return (o.value – o.next.value == 100) &&
      (o.next.value – o.next.next.value == 200) &&
      (o.value + o.next.next.value == 800);
}
```



α: $(w_3 - w_2 = 100) \land (w_2 - w_1 = 200) \land (w_3 + w_1 = 800)$

Call Sequence Synthesis

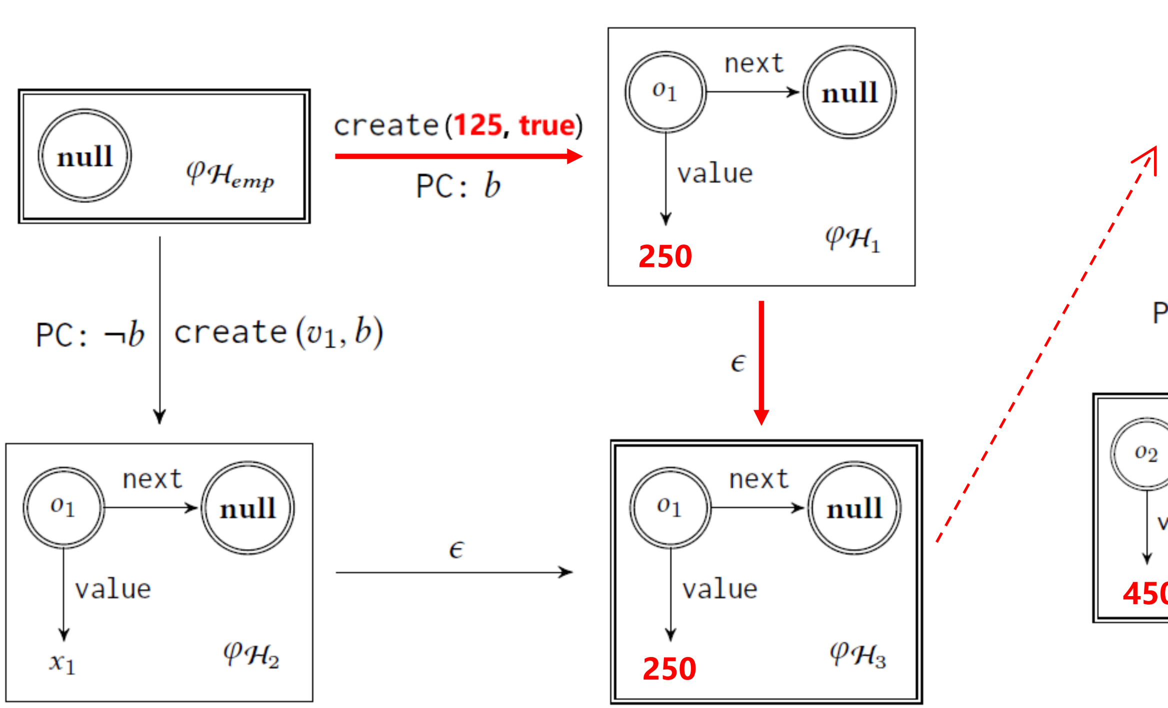


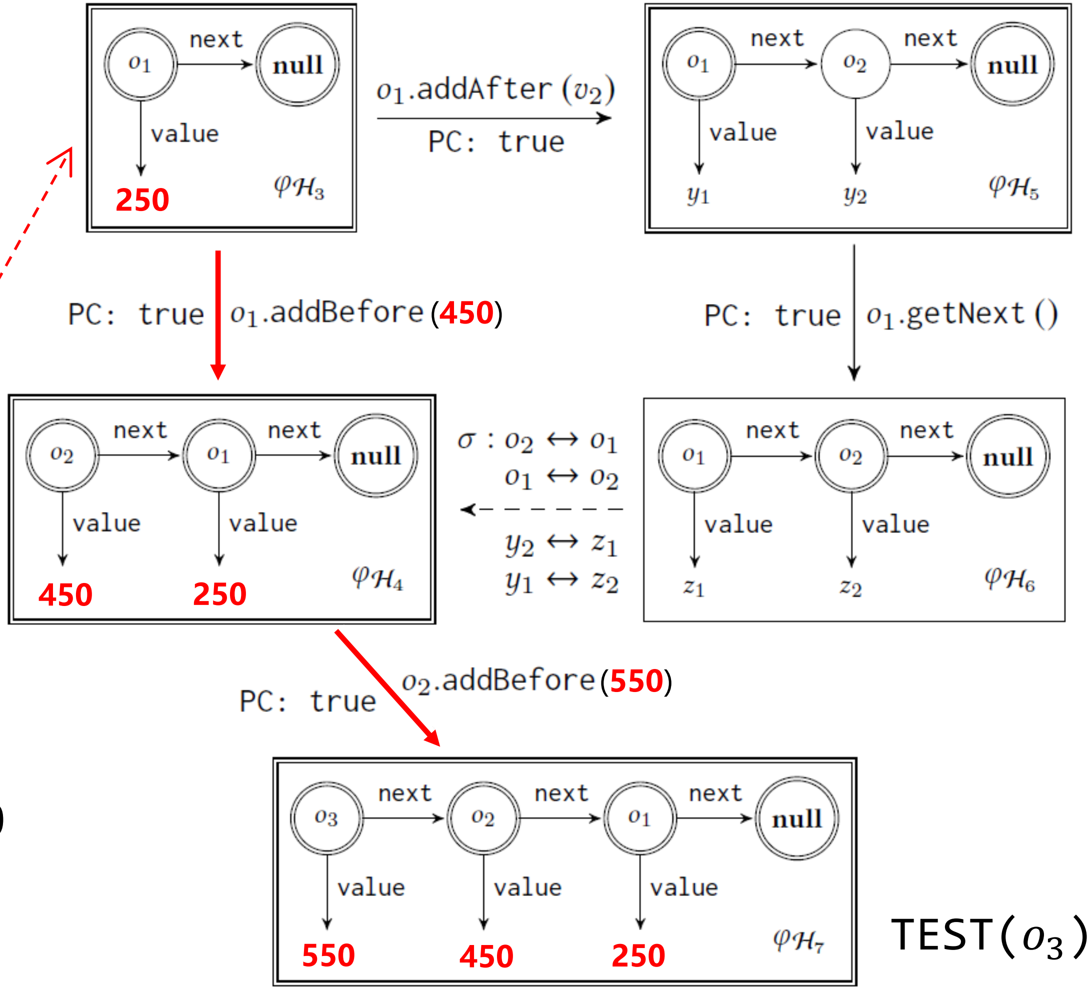


Solve $\varphi_{H_7} \wedge \alpha$ $\longrightarrow$

$$\begin{cases} w_3 = v_3 = 550 \\ w_2 = y_2 = v_2 = 450 \\ w_1 = y_1 = x_1 = 250 \\ v_1 = 125 \\ b = \text{true} \end{cases}$$

Call Sequence Synthesis

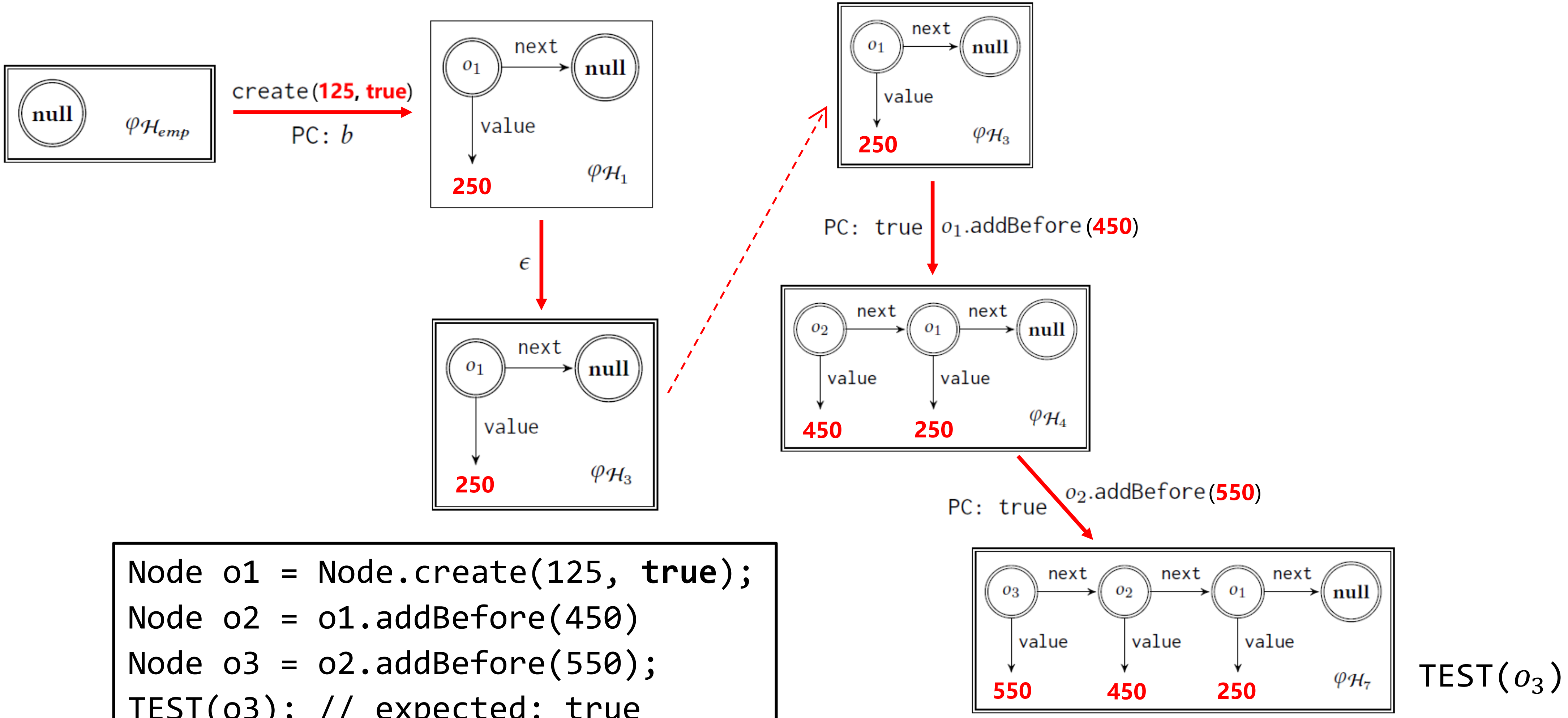




Solve $\varphi_{H_7} \wedge \alpha$ $\longrightarrow$

$$\begin{cases} w_3 = v_3 = 550 \\ w_2 = y_2 = v_2 = 450 \\ w_1 = y_1 = x_1 = 250 \\ v_1 = 125 \\ b = \text{true} \end{cases}$$

Call Sequence Synthesis




```
Node o1 = Node.create(125, true);
Node o2 = o1.addBefore(450)
Node o3 = o2.addBefore(550);
TEST(o3); // expected: true
```

Outline

1. Background and motivation
2. Algorithm with a running example
3. Evaluation

Evaluation Setup

- **Implentation**: A prototype named `MSeqSynth`
  - Symbolic Execution Engine: `JBSE`
  - Constraint Solver: `Z3`

- **Subject programs**: 14 data structure classes implemented in Java, including
  - 4 classes from SUSHI's experiments,
  - 6 classes from the Sireum/Kiasan's examples,
  - 2 classes from Software-artifact Infrastructure Repository (SIR),
  - 2 classes from the JavaScan website **(containing programming tutorials with examples)**

benchmarks used in previous publications

RQ1: Effectiveness on Test Generation

- **Baseline**: SUSHI (= a path selector + a search algorithm)

	Subject	$\|\mathcal{M}\|$	B_{all}	MSeqSynth T_{all}	MSeqSynth $T_{explore}$	MSeqSynth N_{solve}	MSeqSynth N_{fail}	MSeqSynth T_{solve}	MSeqSynth T_{fail}	MSeqSynth B_{cov}	SUSHI T_{all}	SUSHI N_{solve}	SUSHI N_{fail}	SUSHI T_{solve}	SUSHI T_{fail}	SUSHI B_{cov}
SUSHI	Avl	7	59	51.5	30	15	0	0.02	-	59	120.8	15	0	6	-	59
	RBT	10	191	399.4	300	34	0	0.22	-	**191**	3600*	30	14	35.1	175.5	162
	DList	38	136	486	328	49	0	0.05	-	**136**	3600*	41	16	11.9	>180	111
	CList	7	80	147	62	11	43	0.02	1.42	78	500.7	11	2	19.3	129.2	**80**
Kiasan	Avl	7	55	64.1	36	15	203	0.01	<0.01	**55**	3600*	10	20	5.6	>180	29
	RBT	10	180	438.7	295	35	983	0.08	0.04	**175**	3600*	20	21	16.2	>180	101
	BST	8	51	68.2	49	14	0	0.01	-	51	100.1	14	0	5.6	-	51
	AATree	8	58	3600*	1800*	16	563	0.02	2.95	**56**	3600*	12	18	5.6	>180	40
	Leftist	7	31	339.1	317	10	5	0.01	0.73	31	1000	10	5	5.5	>180	31
	Stack	8	17	28.3	12	10	0	0.01	-	17	67	10	0	5.5	-	17
SIR	DList	22	81	206	151	33	3	0.03	0.01	81	1018	33	3	8.4	>180	81
	SList	13	41	199.2	167	13	1	0.01	<0.01	41	302.7	13	1	5.5	>180	41
JavaScan	Skew	6	25	43.2	29	8	0	0.01	-	25	54.6	8	0	5.5	-	25
	Binom	9	114	3600*	1298	16	1419	0.05	0.02	**98**	3600*	14	16	5.6	>180	85

The total branches covered by MSeqSynth (1094) is 20% more than those covered by SUSHI (913).

RQ1: Effectiveness on Test Generation

- **Baseline**: SUSHI (= a path selector + a search algorithm)

				MSeqSynth							SUSHI					
	Subject	$\lvert\mathcal{M}\rvert$	B_{all}	T_{all}	$T_{explore}$	N_{solve}	N_{fail}	T_{solve}	T_{fail}	B_{cov}	T_{all}	N_{solve}	N_{fail}	T_{solve}	T_{fail}	B_{cov}
SUSHI	Avl	7	59	51.5	30	15	0	0.02	-	59	120.8	15	0	6	-	59
	RBT	10	191	399.4	300	34	0	0.22	-	**191**	3600*	30	14	35.1	175.5	162
	DList	38	136	486	328	49	0	0.05	-	**136**	3600*	41	16	11.9	>180	111
	CList	7	80	147	62	11	43	0.02	1.42	78	500.7	11	2	19.3	129.2	**80**
Kiasan	Avl	7	55	64.1	36	15	203	0.01	<0.01	**55**	3600*	10	20	5.6	>180	29
	RBT	10	180	438.7	295	35	983	0.08	0.04	**175**	3600*	20	21	16.2	>180	101
	BST	8	51	68.2	49	14	0	0.01	-	51	100.1	14	0	5.6	-	51
	AATree	8	58	3600*	1800*	16	563	0.02	2.95	**56**	3600*	12	18	5.6	>180	40
	Leftist	7	31	339.1	317	10	5	0.01	0.73	31	1000	10	5	5.5	>180	31
	Stack	8	17	28.3	12	10	0	0.01	-	17	67	10	0	5.5	-	17
SIR	DList	22	81	206	151	33	3	0.03	0.01	81	1018	33	3	8.4	>180	81
	SList	13	41	199.2	167	13	1	0.01	<0.01	41	302.7	13	1	5.5	>180	41
JavaScan	Skew	6	25	43.2	29	8	0	0.01	-	25	54.6	8	0	5.5	-	25
	Binom	9	114	3600*	1298	16	1419	0.05	0.02	**98**	3600*	14	16	5.6	>180	85

The total elapsed time of MSeqSynth (9671s) is 61% less than that of SUSHI (24764s).

RQ2: Effectiveness on Bounded Verification

- **Baseline**: a constructed baseline $\texttt{SE}_{\texttt{seq}}$ (based on the symbolic executor `JPF`)

Subject	Property	maxL = 7 T_{synth}	maxL = 7 T_{SE}	maxL = 8 T_{synth}	maxL = 8 T_{SE}
`Avl`	`balanced`	60.5	258	66.8	N/A
	`ordered`	31.1	260	39.5	N/A
	`wellFormed`	44.4	255	52.2	N/A
`BST`	`ordered`	39.1	1743	61.1	N/A
`AATree`	`ordered`	790.8	1630	N/A	N/A
	`wellLevel`	775.7	890	N/A	N/A
	`wellFormed`	1162	1637	N/A	N/A

MSeqSynth can verify heap-based programs and properties more efficiently than the baseline.

Summary

Thank you!

- Contribution: developing an efficient synthesis algorithm for method call sequences
- An offline procedure for exploring reachable heap states
  - Based on (isomorphic) state abstraction and state merging
- An online procedure for synthesizing method call sequences
  - Combining enumerative techniques and symbolic techniques
- Evaluation results show that our algorithm performs efficiently in both test generation tasks and bounded verification tasks.