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

Yunfan Zhang, Ruidong Zhu, Yingfei Xiong, Tao Xie
Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University)
School of Computer Science, Peking University
Beijing, PR China
{zyf0726,zhurd,xiongyf,taoxie}@pku.edu.cn

By 高宏艳

Outline

- Background & Motivation
- Algorithm with a running example
- Evaluation

Outline

- Background & Motivation
- Algorithm with a running example
- 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);
}
```

Specification:d. Typically, we can obtain the specifications for reaching different paths or triggering errors by symbolic execution or by manual writing of specifications.

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 satisfy a given specification?

Background

```java
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 + 1);
        else return new Node(null, v * 2);
    }
    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);
    }
}
```

field

constructor

Static factory method

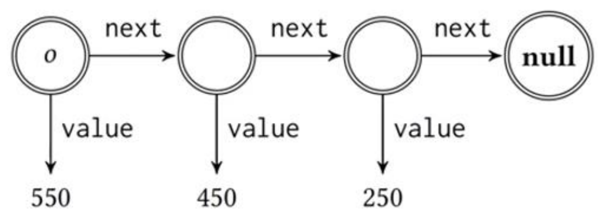
Instance method

Figure 1: A sample Java class implementing a list node

A simple specification

```java
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 solution



Background

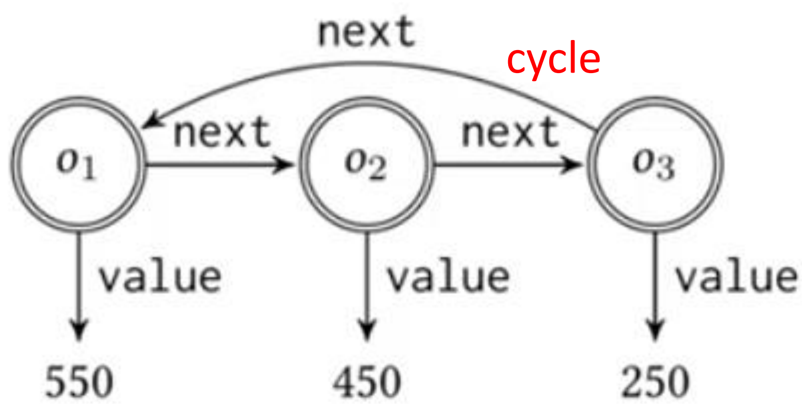
Existing Work

- Direct construction

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

Directly assign values to the fields of the heap objects

```
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 valid/unreachable heap states

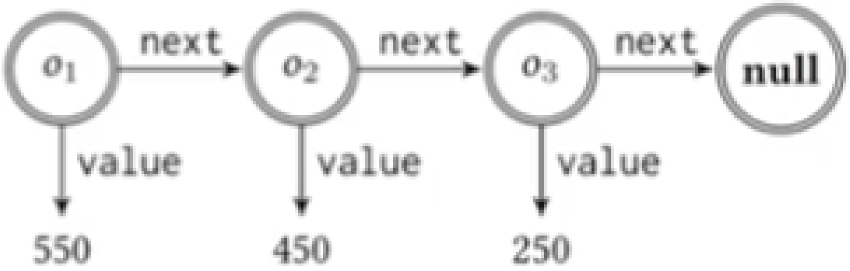
Background

Existing Work

- Sequence generation

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);
```



However, even the state-of-the-art approach, SUSHI, fails to generate a method call sequence for constructing a heap state to satisfy the specification within 10 hours

Motivation

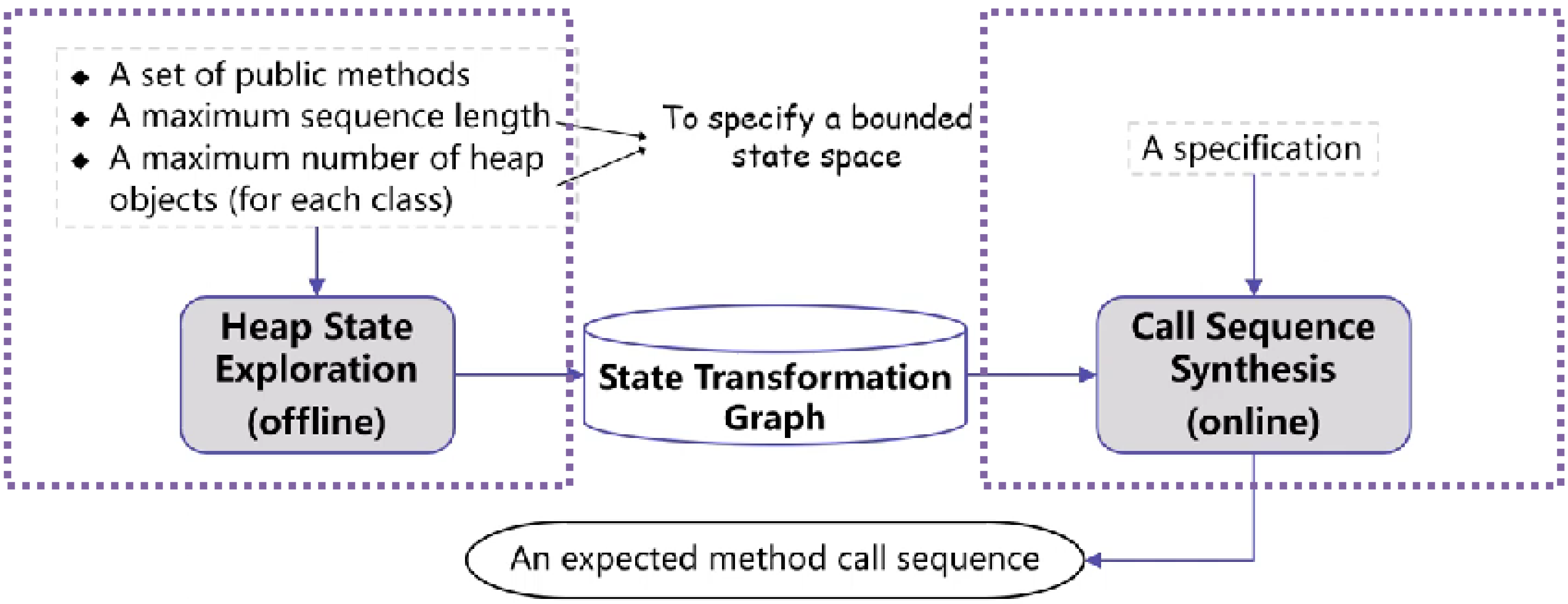
Developing an efficient synthesis algorithm for method call sequences

Outline

- Background & Motivation
- Algorithm with a running example
- Evaluation

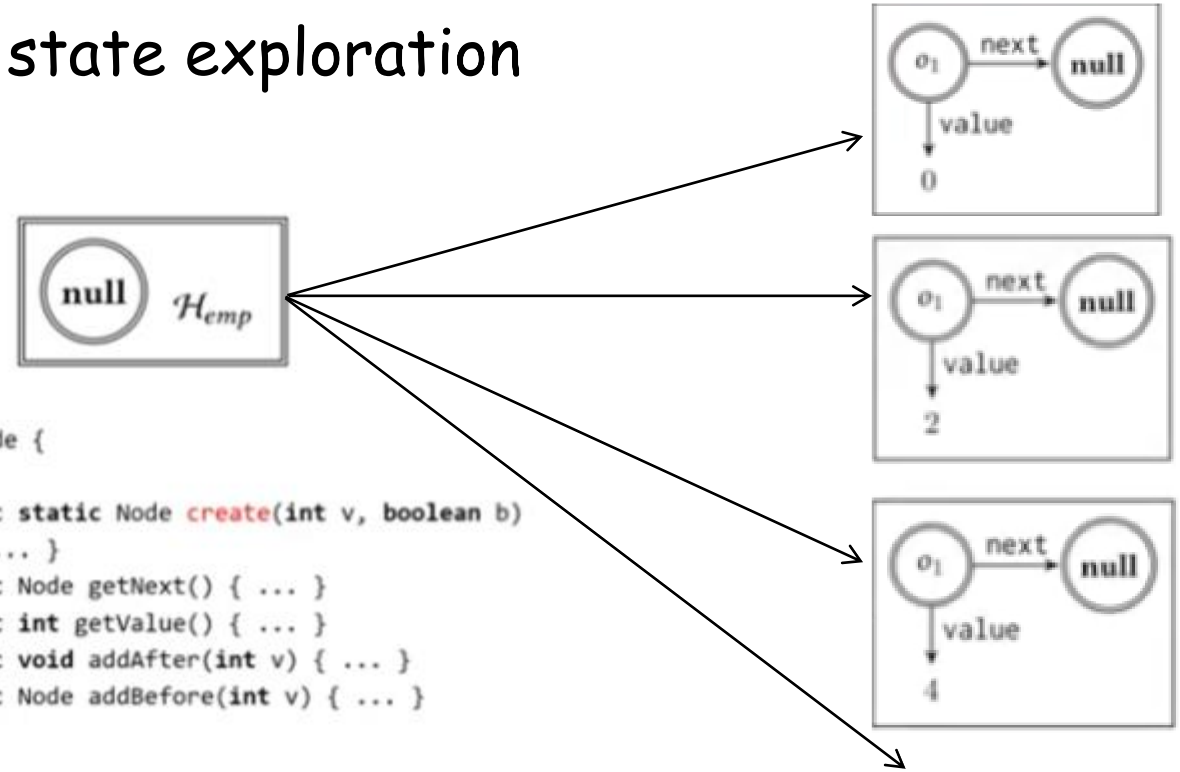
Algorithm

Workflow



Example

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) { ... }
}
```

......

Simple enumerate is infeasible 😑……

Example

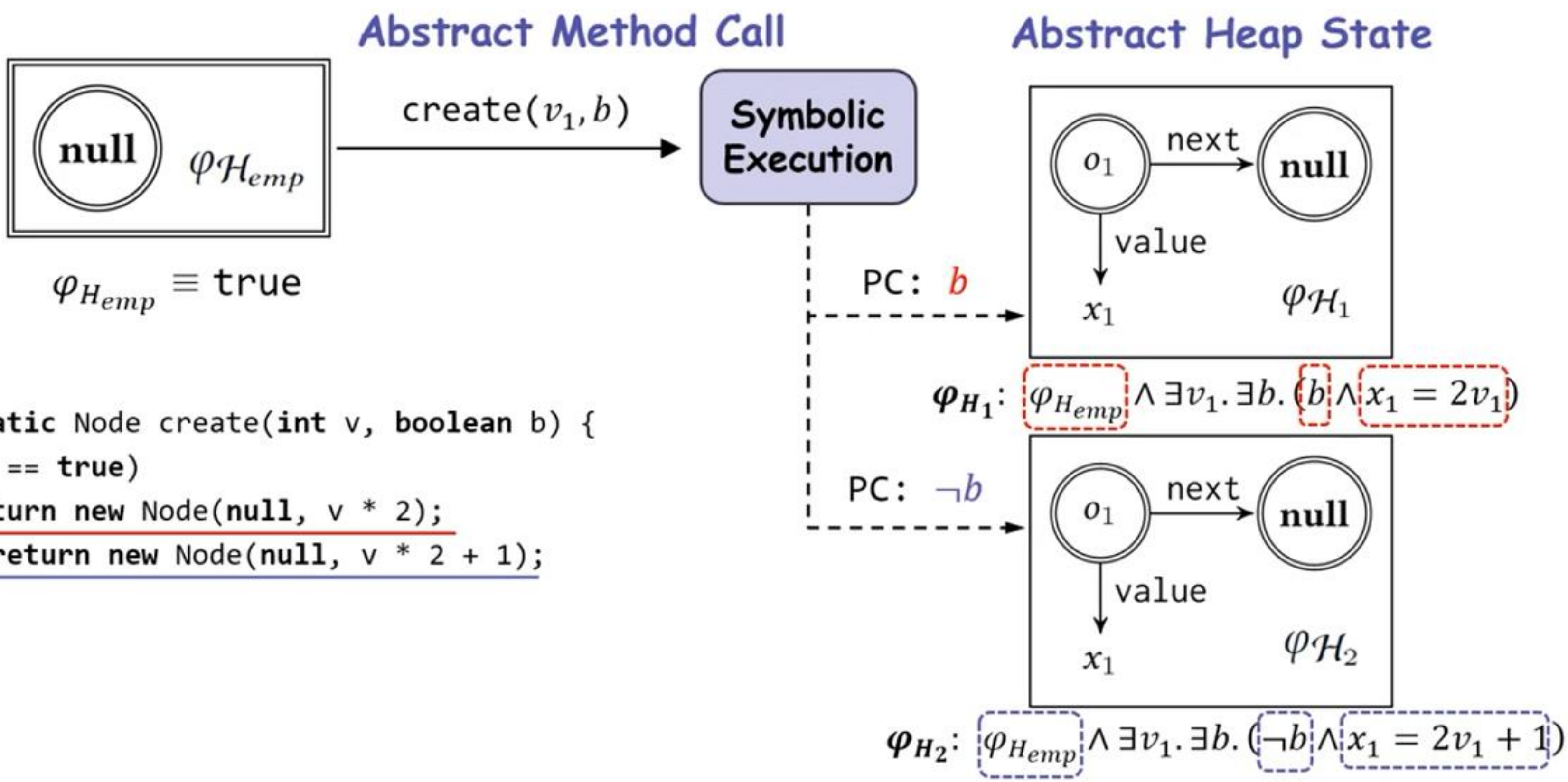
A heap state = a heap structure(objects & references) + primitive values

- The space of the former is relatively small, and can be enumerate
- The space of the latter is large, but can be inferred using a constraint solver

Enumeration of the heap structure is feasible😀~

Example

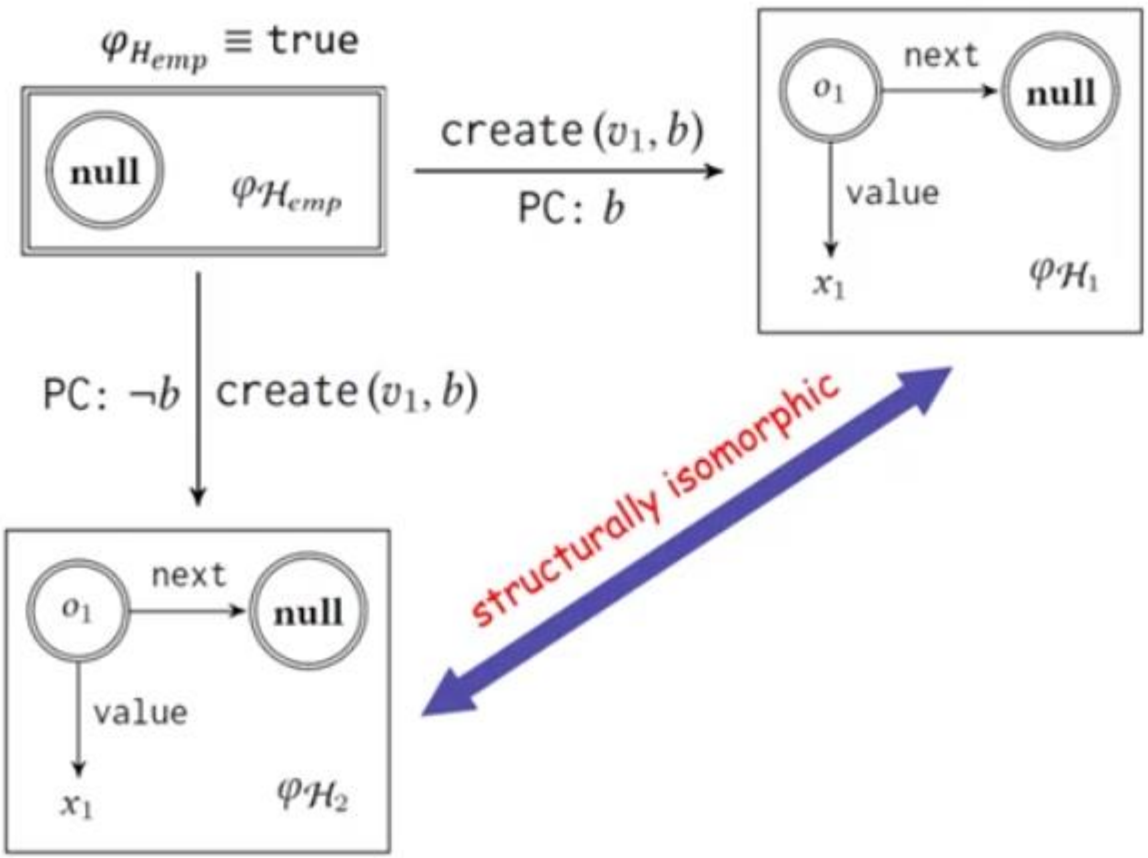
State abstraction




```
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);
}
```

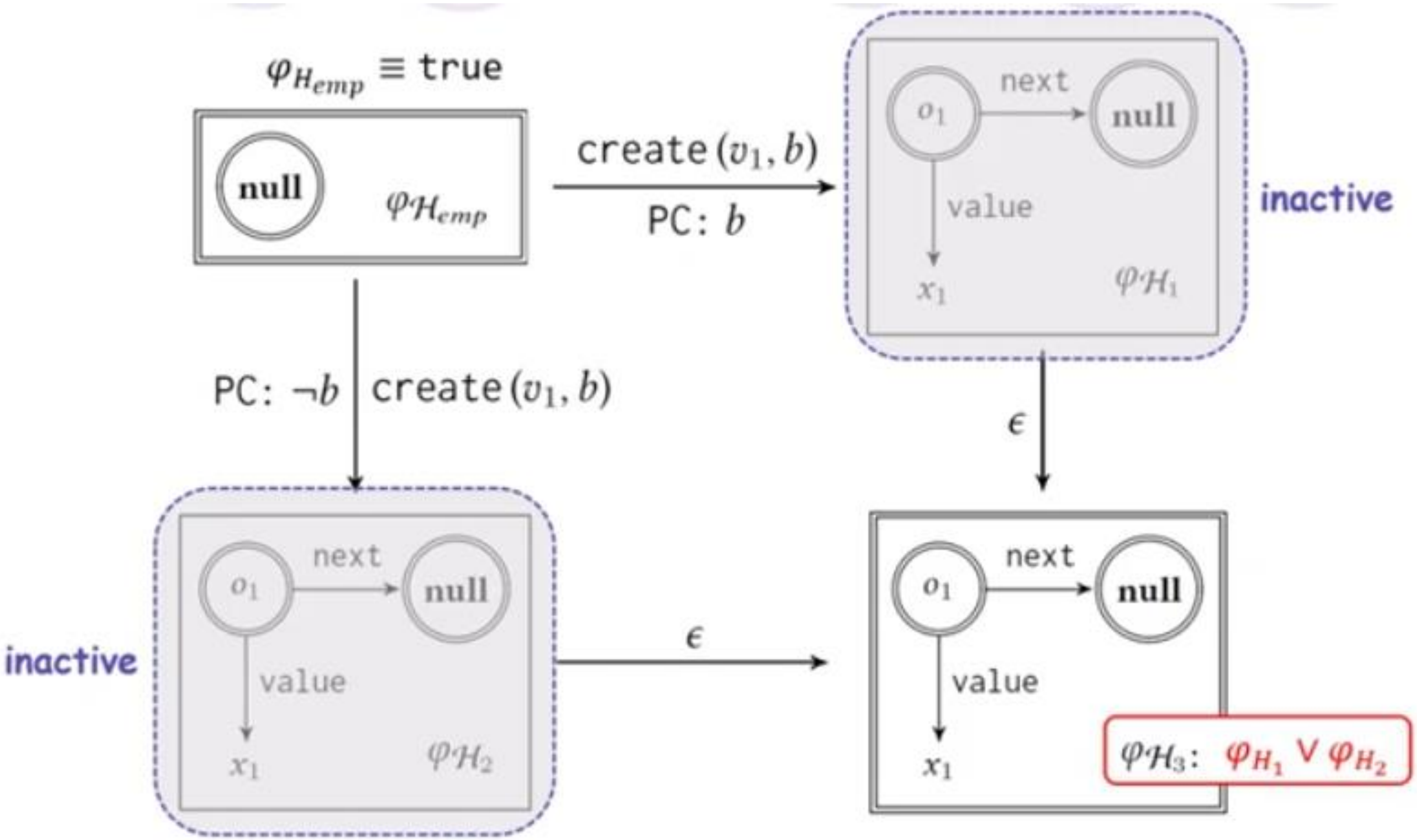
Example

Structural isomorphism



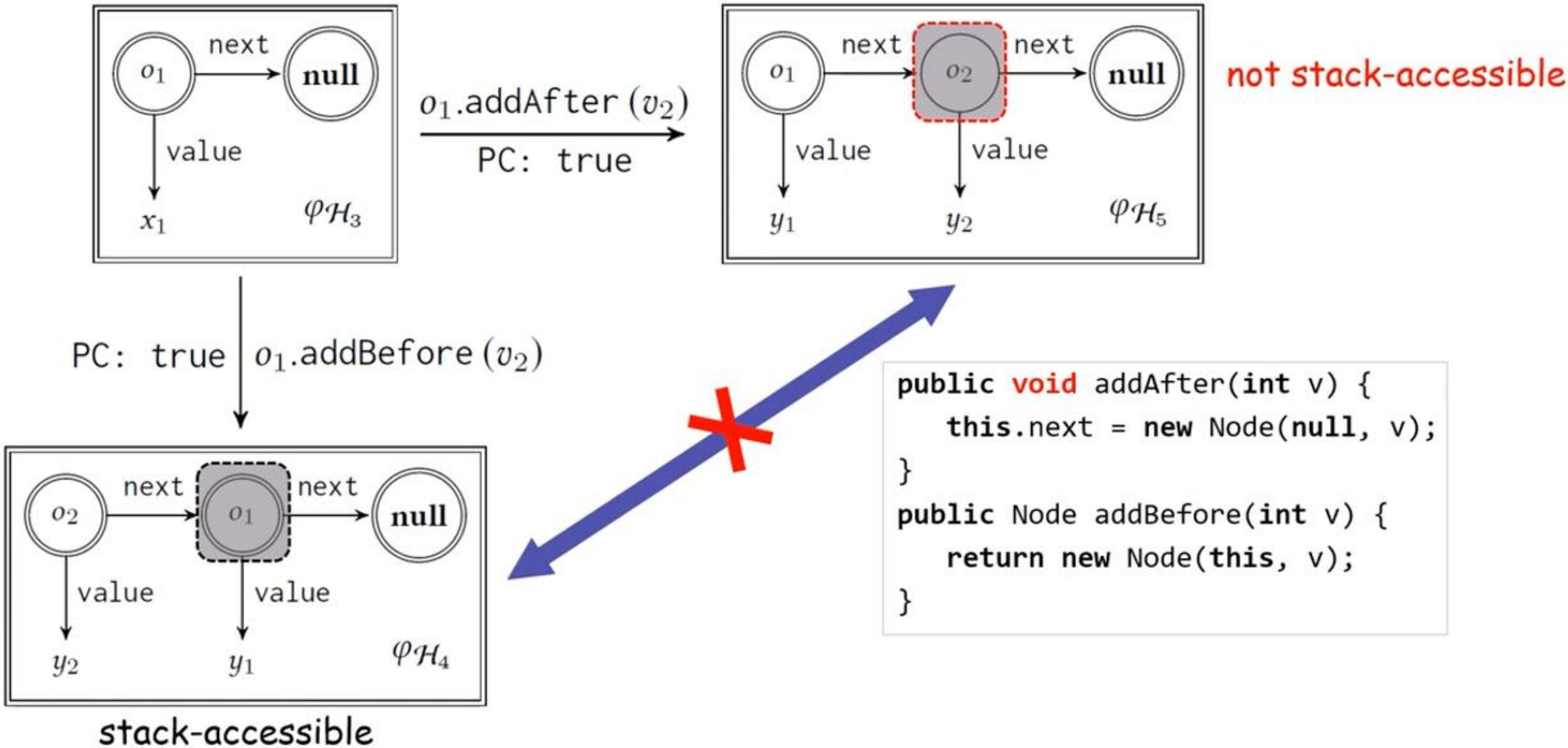
Example

State Merging



Example

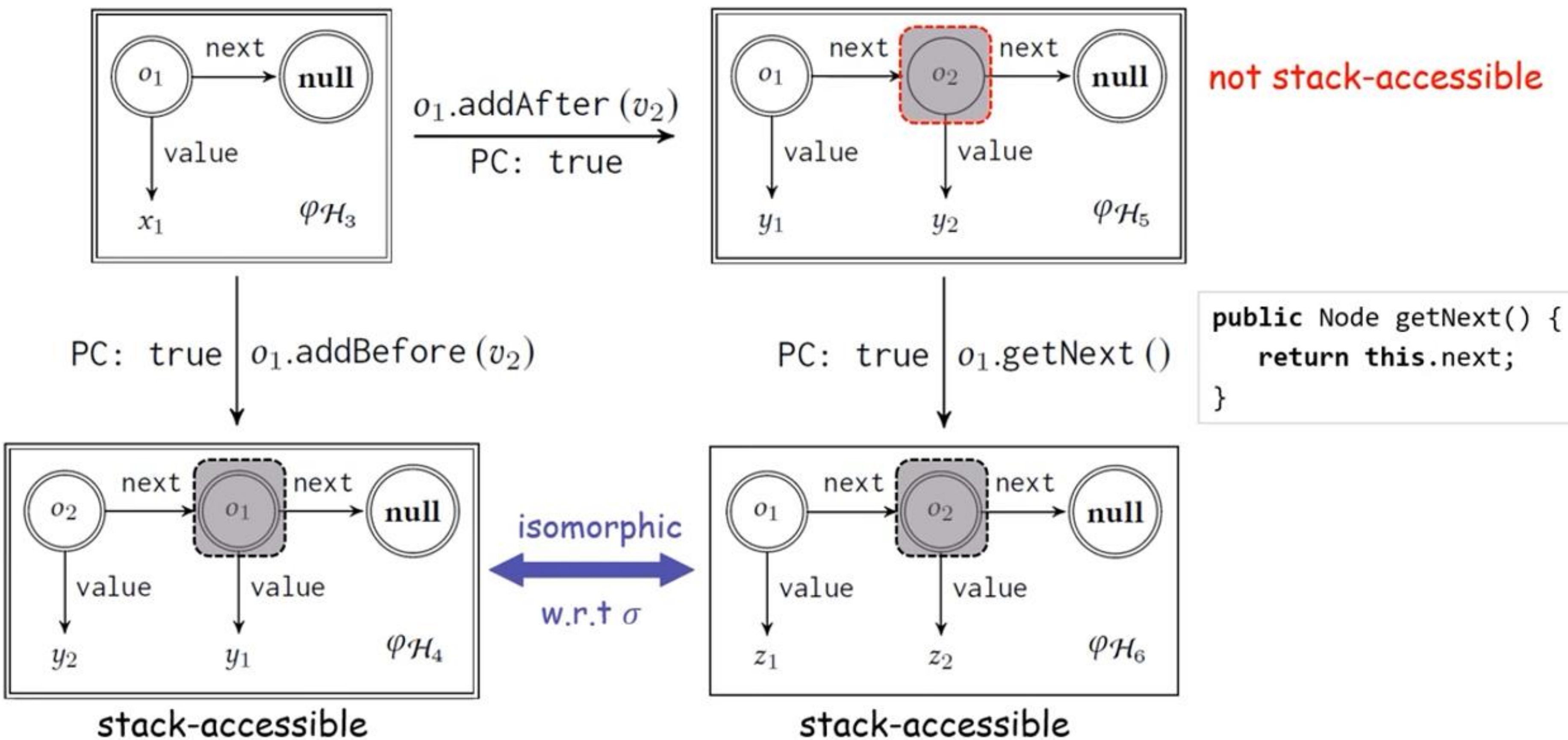
Heap state exploration



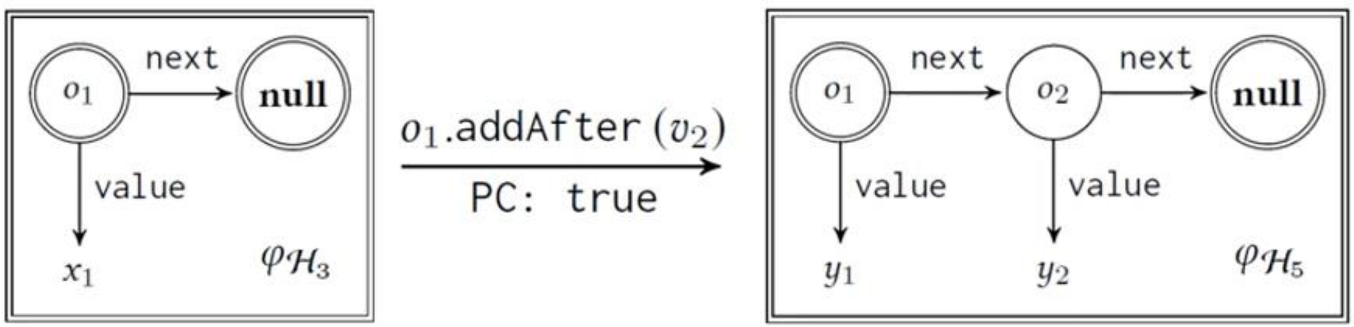

```java
public void addAfter(int v) {
    this.next = new Node(null, v);
}
public Node addBefore(int v) {
    return new Node(this, v);
}
```

Example

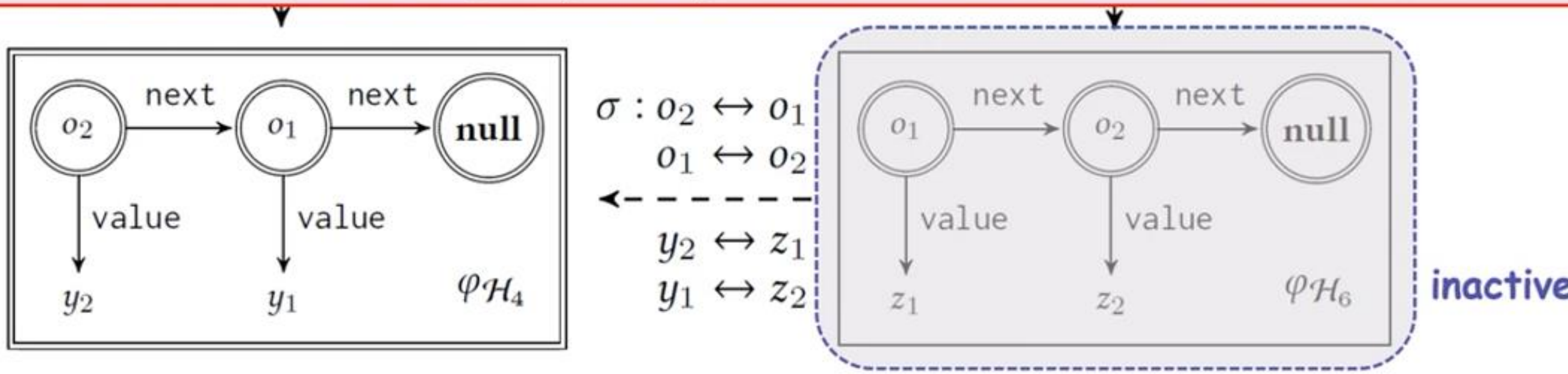
Heap state exploration



Example

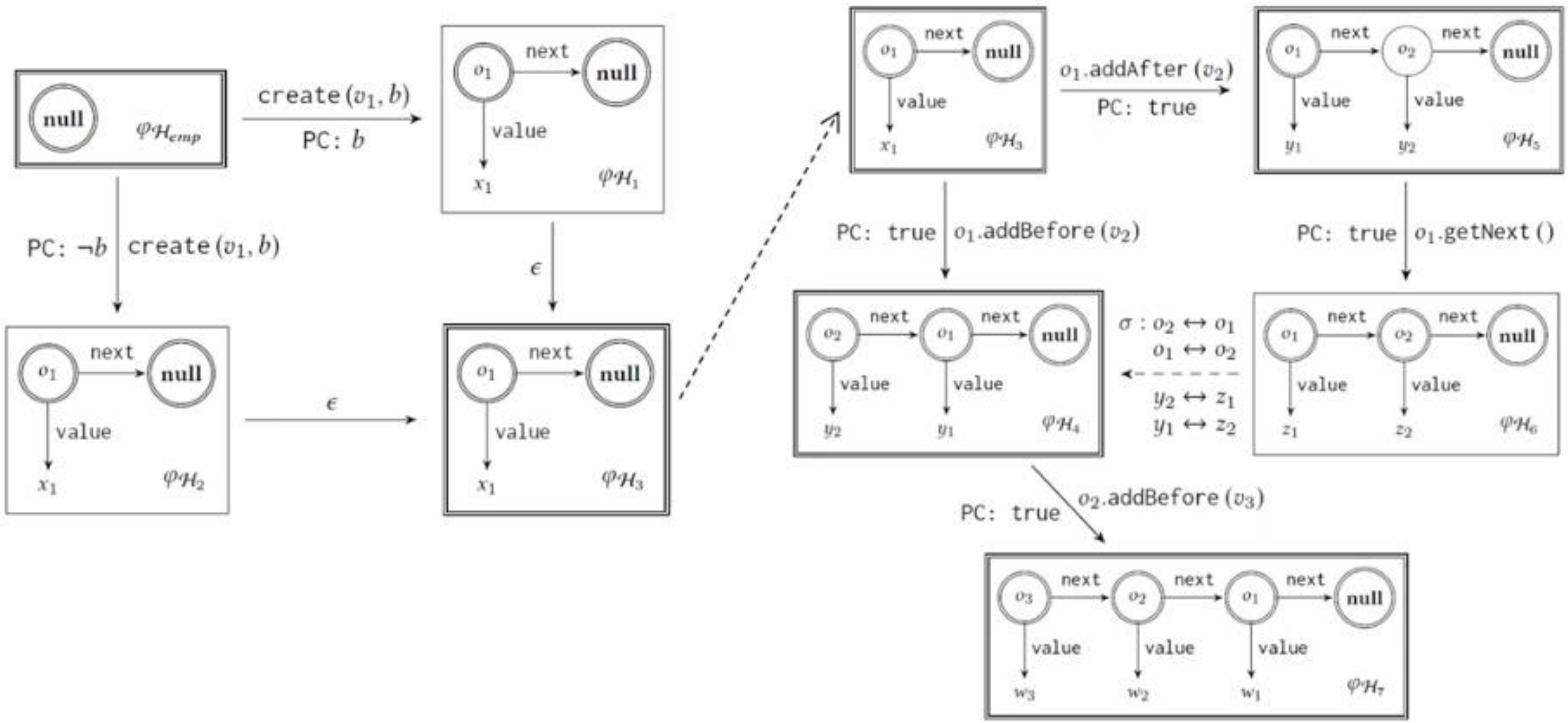


All concrete heap states represented by H_6 are included in those represented by H_4!
(by checking $\varphi_{H_6} \to \varphi_{H_4}[y_2 := z_1, y_1 := z_2]$ holds for all z_1, z_2)



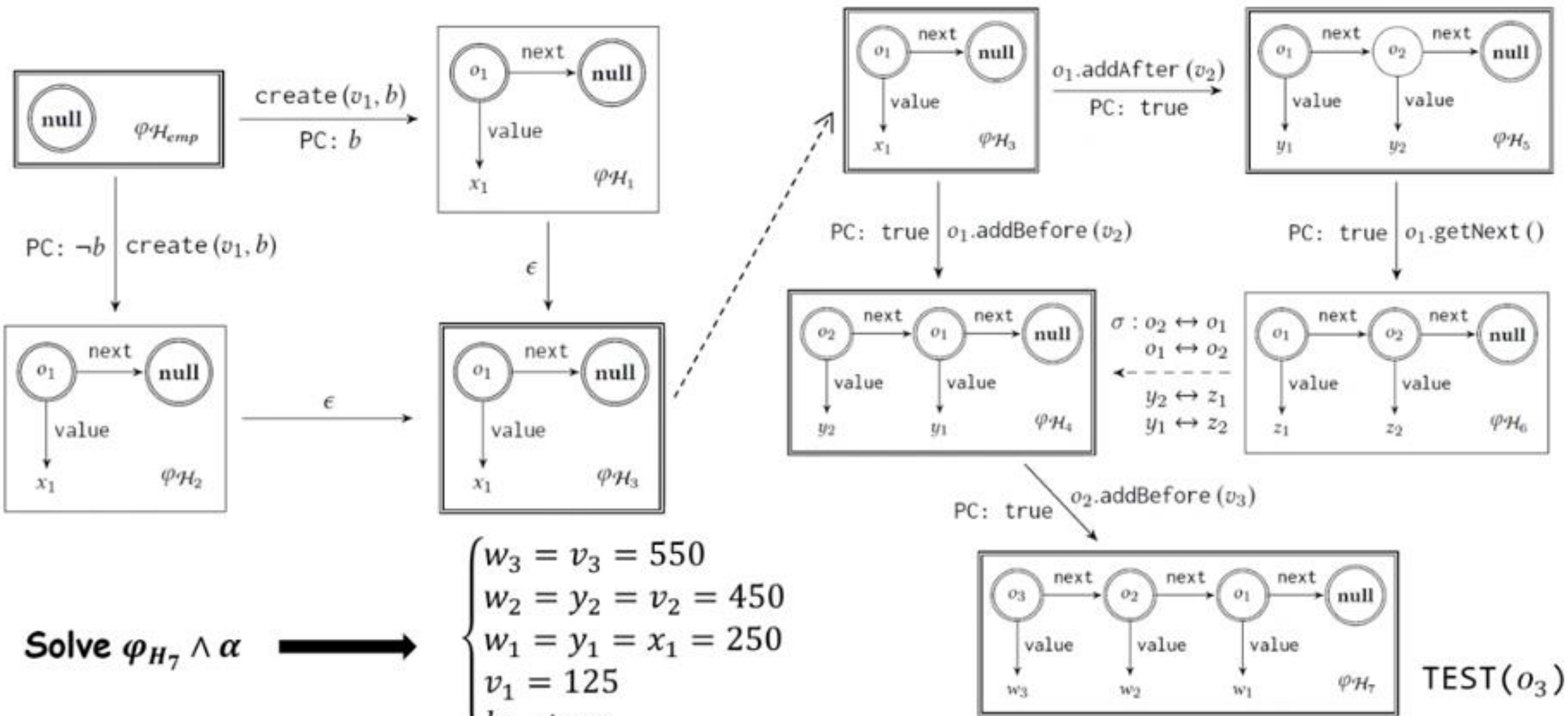
Example

State transformation graph



Example

Call sequence synthesis



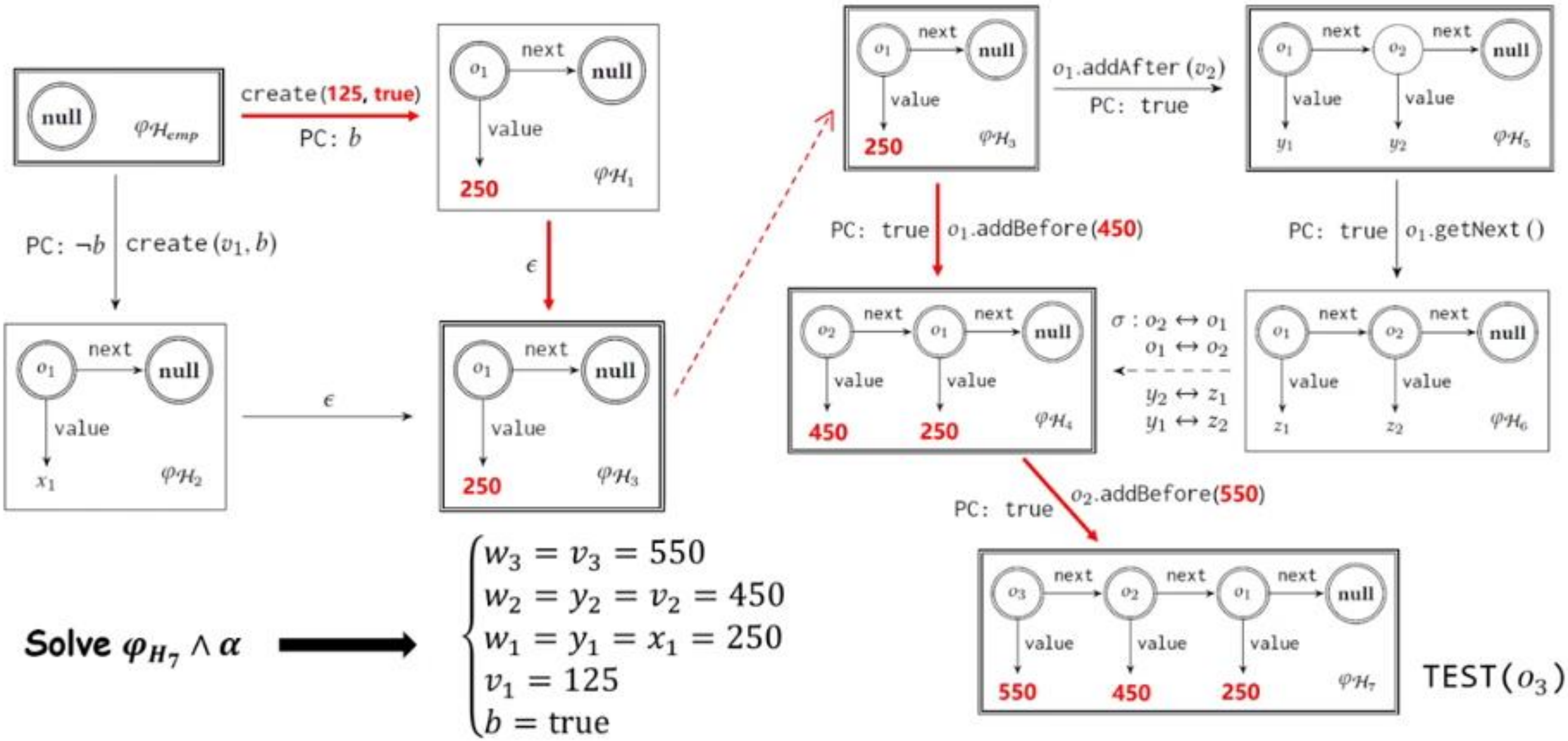
Solve $\varphi_{H_7} \wedge \alpha$ ⟶

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

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

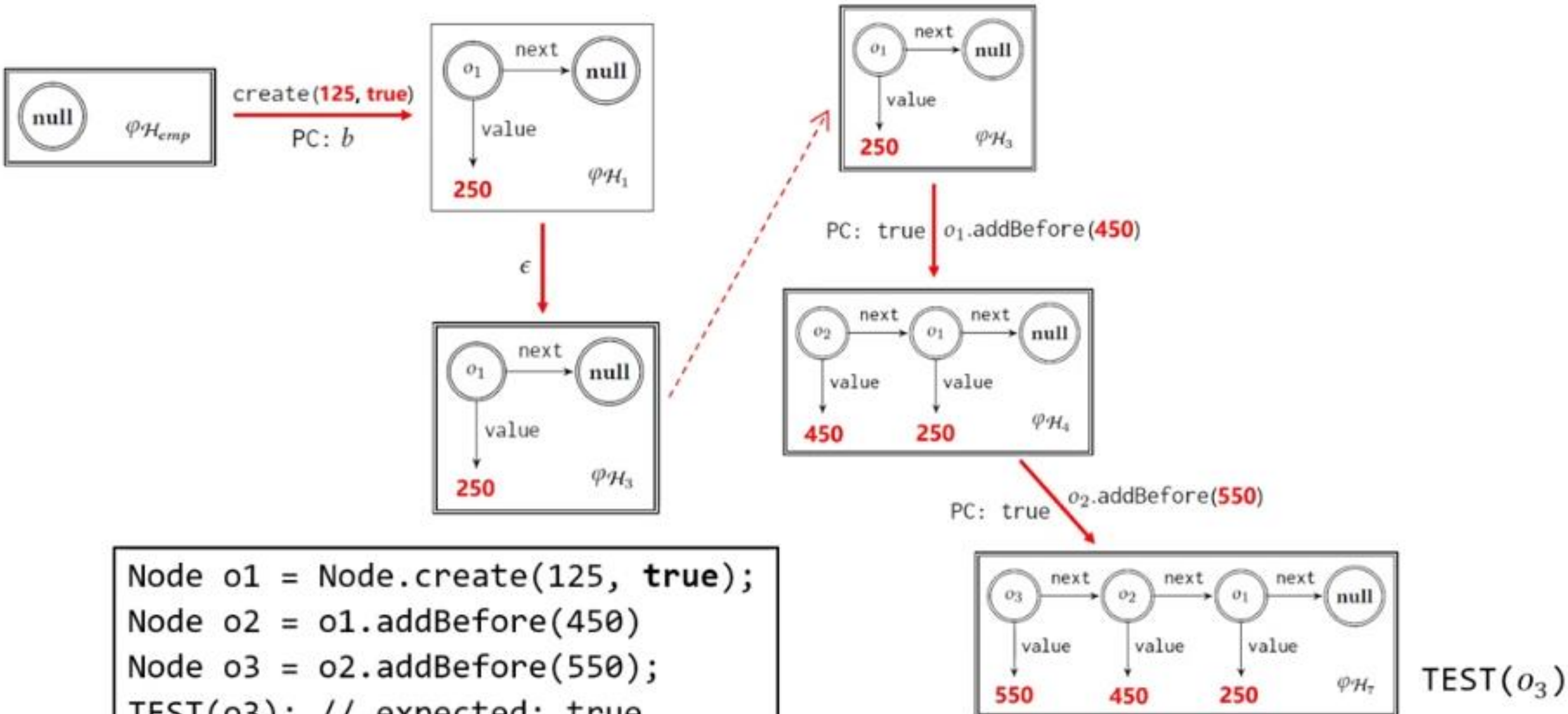
Example

Call sequence synthesis



Example

Call sequence synthesis




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

Algorithm

Algorithm 1: The pseudo-code of `BuildGraph`.

Input: A set of methods $\mathcal{M}$, a mapping *hs* specifying the heap scope, and a maximum sequence length *maxL*
Output: A state transformation graph $\mathcal{G} = (V_{act}, V, E)$

$\mathcal{H}_{emp} \leftarrow (\{\texttt{null}\}, \{\texttt{null}\}, \emptyset, \emptyset, \texttt{true})$;
$V \leftarrow \{\mathcal{H}_{emp}\}$; $V_{act} \leftarrow \{\mathcal{H}_{emp}\}$; $E \leftarrow \emptyset$;
Function `AddState`$(\mathcal{H}_{new}, \mathcal{H}_{old}, c, p)$:
  if $\mathcal{H}_{new}$ *is out of the heap scope hs* **then return** $(\perp, \perp)$;
  $V \leftarrow V \cup \{\mathcal{H}_{new}\}$; $E \leftarrow E \cup \{(\mathcal{H}_{old}, c, p, \mathcal{H}_{new})\}$;
  for *each abstract state* $\mathcal{H} \in V_{act}$ **do**
    $\sigma \leftarrow$ `DecideIsomorphism`$(\mathcal{H}, \mathcal{H}_{new})$;
    if $\sigma \neq \perp$ **then**
      if $\varphi_{\mathcal{H}_{new}} \to \varphi_{\mathcal{H}}[v := \sigma(v)]$ *always holds* **then**
        return $(\perp, \perp)$
      $\mathcal{H}_u \leftarrow \mathbf{merge}_\sigma(\mathcal{H}, \mathcal{H}_{new})$;
      $V \leftarrow V \cup \{\mathcal{H}_u\}$;
      $E \leftarrow E \cup \{(\mathcal{H}, \epsilon, \sigma, \mathcal{H}_u)\} \cup \{(\mathcal{H}_{new}, \epsilon, \sigma_{id}, \mathcal{H}_u)\}$;
      $V_{act} \leftarrow V_{act} \setminus \{\mathcal{H}\} \cup \{\mathcal{H}_u\}$;
      return $(\mathcal{H}_u, \mathcal{H})$;
  $V_{act} \leftarrow V_{act} \cup \{\mathcal{H}_{new}\}$;
  return $(\mathcal{H}_{new}, \perp)$;
Function `ExploreStates`$(oldStates)$:
  $newStates \leftarrow \emptyset$;
  while $oldStates \neq \emptyset$ **do**
    $\mathcal{H}_{old} \leftarrow$ pop an abstract state from $oldStates$;
    for *each public method* $m \in \mathcal{M}$ **do**
      for *each* $c \in$ `GetAbstractCalls`$(\mathcal{H}_{old}, m)$ **do**
        perform symbolic execution for c and obtain a set of path descriptors $\{(\alpha_p, O_p, r_p, \tau_p)\}$;
        for *each path p* **do**
          if $\varphi_{\mathcal{H}_{old}} \wedge \alpha_p$ *is unsatisfiable* **then**
            continue
          $\mathcal{H}_{new} \leftarrow \mathbf{PostState}(\mathcal{H}_{old}, c, p)$;
          $\mathcal{H}_u, \mathcal{H} \leftarrow$ `AddState`$(\mathcal{H}_{new}, \mathcal{H}_{old}, c, p)$;
          if $\mathcal{H}_u = \perp$ **then continue**;
          if $\mathcal{H}_u = \mathcal{H}_{new}$ **then**
            $newStates \leftarrow newStates \cup \{\mathcal{H}_{new}\}$;
            continue;
          if $\mathcal{H} \in oldStates$ **then**
            $oldStates \leftarrow oldStates \setminus \{\mathcal{H}\} \cup \{\mathcal{H}_u\}$;
          else
            $newStates \leftarrow newStates \setminus \{\mathcal{H}\} \cup \{\mathcal{H}_u\}$;
  return $newStates$;
$oldStates \leftarrow \{\mathcal{H}_{emp}\}$;
for $L \leftarrow 1 \ldots maxL$ **do**
  $oldStates \leftarrow$ `ExploreStates`$(oldStates)$;
  if $oldStates = \emptyset$ **then break**;
return $\mathcal{G} = (V_{act}, V, E)$;

Heap state exploration

Algorithm

Call sequence synthesis

Algorithm 2: The pseudo-code of `SynthCallSeq`.

Input: A state transformation graph $\mathcal{G} = (V_{act}, V, E)$, and a specification Φ in the form of a Boolean function

Output: a sequence of method calls with their return values S, and two lists of arguments $\overline{o}$ and $\overline{a}$ such that $\Phi(\overline{o}, \overline{a})$ returns `true`

Function `Traverse`$(\mathcal{H}, \overline{o}, \overline{x}, \pi)$:
  $S \leftarrow [\,]$; $\overline{a} = \overline{\pi(x)}$;
  while $\mathcal{H} \neq \mathcal{H}_{emp}$ **do**
    if *$\mathcal{H}$ is a union abstract state* **then**
      $(\mathcal{H}_1, \epsilon, \sigma_1, \mathcal{H}), (\mathcal{H}_2, \epsilon, \sigma_2, \mathcal{H}) \leftarrow$ the two incoming ϵ edges of $\mathcal{H}$;
      if $\varphi_{\mathcal{H}_1}[v := \sigma_1(v)]$ *is satisfied under* π **then**
        $\mathcal{H} \leftarrow \mathcal{H}_1$; $\sigma \leftarrow \sigma_1$;
      else
        // $\varphi_{\mathcal{H}_2}[v := \sigma_2(v)]$ is satisfied under π
        $\mathcal{H} \leftarrow \mathcal{H}_2$; $\sigma \leftarrow \sigma_2$;
      $S \leftarrow \sigma^{-1}(S)$; $\overline{o} \leftarrow \sigma^{-1}(\overline{o})$; $\pi \leftarrow \sigma^{-1}(\pi)$;
    else
      $(\mathcal{H}', c, p, \mathcal{H}) \leftarrow$ the incoming edge of $\mathcal{H}$;
      insert $(c[\pi], r_p)$ at the front of S;
      $\mathcal{H} \leftarrow \mathcal{H}'$;
  return $S, \overline{o}, \overline{a}$;

backtracking

for *each abstract heap state* $\mathcal{H} \in V_{act}$ **do**
  for *each* $c = (\Phi, \overline{o}, \overline{x}) \in$ `GetAbstractCalls`$(\mathcal{H}, \Phi)$ **do**
    perform symbolic execution for c and obtain a set of path descriptors $\{(\alpha_p, O_p, r_p, \tau_p)\}$;
    for *each path* p **do**
      if $r_p \neq$ `true` **then continue**;
      $\pi \leftarrow$ `CheckSat`$(\varphi_{\mathcal{H}} \wedge \alpha_p)$;
      if $\pi \neq \perp$ **then**
        return `Traverse`$(\mathcal{H}, \overline{o}, \overline{x}, \pi)$;
return `UNSAT`;

Outline

- Background & Motivation
- Algorithm with a running example
- Evaluation

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

used in the bounded verification experiment

Evaluation

RQ1: Effectiveness on Test Generation

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

Table 1: Comparative evaluation of MSeqSynth and SUSHI.

	Subject	$\lvert\mathcal{M}\rvert$	B_{all}	MSeqSynth T_{all}	$T_{explore}$	N_{solve}	N_{fail}	T_{solve}	T_{fail}	B_{cov}	SUSHI 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

For each subject program, we report the number of public methods ($\lvert\mathcal{M}\rvert$) and the number of all program branches to cover (B_{all}). The number of test generation tasks that are successfully solved or failed to solve is respectively reported as N_{solve} or N_{fail}. Note N_{fail} contains the tasks that are unsolvable. The average generation time for the solved tasks or failed tasks is reported as T_{solve} or T_{fail}. The elapsed time of MSeqSynth for (possibly offline) state exploration is $T_{explore}$. The overall elapsed time for each subject program is T_{all}, and the number of covered program branches is B_{cov}. All the time statistics are reported in seconds. An asterisk (*) indicates the execution timed out.

manually write partial invariants to discard only a part of unreachable paths

Evaluation

RQ2: Effectiveness on Bounded Verification

Table 2: Comparative evaluation of MSeqSynth and $\mathsf{SE_{seq}}$**.**

		maxL = 7		maxL = 8	
Subject	**Property**	T_{synth}	T_{SE}	T_{synth}	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

The verification time of MSeqSynth and $\mathsf{SE_{seq}}$ for each property is reported as T_{synth} and T_{SE} in seconds (N/A means out of time budget or memory exhausted). T_{synth} includes the state exploration time.

we construct a baseline SEseq that extends symbolic execution to (partially) address our problem by writing *driver programs* for each target class

Evaluation

RQ3: Improvement by State Merging

Table 3: Experimental results for evaluating the efficiency improvement of the state merging strategy.

Subject	maxL = 5		maxL = 6		maxL = 7	
	T_{merge}	T_{not}	T_{merge}	T_{not}	T_{merge}	T_{not}
Avl	13	27	13	748	18	N/A
RBT	88	N/A	90	N/A	98	N/A
DList	141	N/A	178	N/A	251	N/A
CList	16	633	22	N/A	29	N/A
Avl	18	26	18	638	20	N/A
RBT	47	N/A	48	N/A	54	N/A
BST	11	31	12	1253	18	N/A
AATree	69	107	126	N/A	733	N/A
Leftist	9	15	9	611	13	N/A
Stack	9	17	9	246	9	N/A
DList	60	N/A	77	N/A	116	N/A
SList	25	400	35	N/A	56	N/A
Skew	7	8	8	20	9	190
Binom	143	189	148	N/A	176	N/A

T_{merge} is the elapsed time of state exploration with state merging, and T_{not} is the elapsed time without state merging, all in seconds (N/A means out of time budget or memory exhausted).

Evaluation

Summary

- Contribution: developing an efficient synthesis algorithm for method call sequences

- An offline procedure for exploring reachable heap states
  - Based om (isomorphic) state abstraction and state merging
- An online procedure for synthesizing method call sequences
  - Combining enumerative techniques and symbolic techniques

- Evaluation results show that this algorithm performs efficiently in both test generation tasks and bounded verification tasks

Thanks for your attention!

Q&A

Definition 4.1 (abstract heap state). An abstract heap state $\mathcal{H}$ is a 5-tuple $(O_{\mathcal{H}}, AO_{\mathcal{H}}, Var_{\mathcal{H}}, \delta_{\mathcal{H}}, \varphi_{\mathcal{H}})$ where:

- $O_{\mathcal{H}}$ and $AO_{\mathcal{H}}$ are sets of all heap objects and stack-accessible heap objects, respectively;
- $Var_{\mathcal{H}}$ is a set of symbolic variables;
- $\delta_{\mathcal{H}} : (O_{\mathcal{H}} \times \mathcal{F}) \rightarrow (O_{\mathcal{H}} \cup Var_{\mathcal{H}})$ is a mapping that maps a field of a heap object to an abstract value, which is a reference to either another heap object or a symbolic variable;
- $\varphi_{\mathcal{H}}$ is a first-order constraint with existential quantification, where the free variables in $\varphi_{\mathcal{H}}$ are the variables in $Var_{\mathcal{H}}$.

Q&A

Definition 4.5 (abstract method call). An abstract method call on an abstract heap state $\mathcal{H}$ is a 3-tuple $(m, \bar{o}, \bar{x})$ where $m \in \mathcal{M}$ is a method, $\bar{o}$ is a list of object arguments $o \in AO_{\mathcal{H}}$, and $\bar{x}$ is a list of different symbolic variables.

Formally, the result of symbolic execution for an abstract method call $c = (m, \bar{o}, \bar{x})$ on an abstract pre-state $\mathcal{H}$ is a set of path descriptors $(\alpha_p, O_p, r_p, \tau_p)$, where for each path p:

(1) α_p is the path condition, which is a constraint over the arguments x and the variables $v \in Var_{\mathcal{H}}$;
(2) O_p is the set of all heap objects when execution terminates;
(3) r_p is the return value, which is $\perp$ for no return value, a heap object $o \in O_p$, or a symbolic expression over x and $v \in Var_{\mathcal{H}}$;
(4) τ_p is the symbolic store, which maps fields $f \in \mathcal{F}$ of heap objects $o \in O_p$ to other heap objects (for reference fields) or symbolic expressions (for primitive fields).

For each path p, the abstract post-state $\mathcal{H}' = \textbf{PostState}(\mathcal{H}, c, p)$ can be constructed as follows, according to the result of symbolic exeuction:

(1) $O_{\mathcal{H}'} = O_p$ containing all existent heap objects;
(2) $AO_{\mathcal{H}'} = AO_{\mathcal{H}} \cup \{r_p\}$ if the return value $r_p \in O_p$ is a heap object, otherwise $AO_{\mathcal{H}'} = AO_{\mathcal{H}}$;
(3) for all objects $o \in O_p$ and fields $f \in \mathcal{F}$:
   - $\delta_{\mathcal{H}'}(o, f) = o'$ if $\tau_p(o, f) = o' \in O_p$ is an object, or
   - $\delta_{\mathcal{H}'}(o, f) = v_{o,f}$ if $\tau_p(o, f)$ is an expression, where $v_{o,f}$ is a fresh variable;
(4) $Var_{\mathcal{H}'} = \{v_{o,f} : \tau_p(o, f) \notin O_p\}$ containing all fresh variables;
(5) $\varphi_{\mathcal{H}'} = \exists \bar{v}. \left(\varphi_{\mathcal{H}} \wedge \exists \bar{x}. \left(\alpha_p \wedge \bigwedge_{v_{o,f} \in Var_{\mathcal{H}'}} v_{o,f} = \tau_p(o, f)\right)\right)$ formed by introducing existential quantification on the variables $v \in Var_{\mathcal{H}}$ and the call arguments x, and conjoining the constraint $\varphi_{\mathcal{H}}$ of the pre-state $\mathcal{H}$, the path condition α_p, and a set of equality constraints that characterizes the expected values of the variables $v_{o,f} \in Var_{\mathcal{H}'}$.

Note that the non-object return value of a method call is non-essential, since we can always use a literal value to replace it.

The connection between execution on concrete heap states and symbolic execution on abstract heap heaps is depicted in the following lemma, where the notation $c[\pi]$ indicates substitution of primitive value $\pi(x)$ for symbolic arguments x in c.

Lemma 4.6 (Symbolic execution on abstract states). *Given a path p of an abstract method call c, an abstract pre-state $\mathcal{H}$, and an abstract post-state $\mathcal{H}' = \textbf{PostState}(\mathcal{H}, c, p)$, we hold that*

$$\textbf{Inst}(\mathcal{H}') = \{H' : \pi \models \varphi_{\mathcal{H}} \wedge \pi \models \alpha_p \wedge \mathcal{H}[\pi] \xrightarrow{c[\pi]} H'\}$$

Q&A

Definition 4.4 (structural isomorphism). For two abstract heap states $\mathcal{H}_1$ and $\mathcal{H}_2$, we say that $\mathcal{H}_1$ is *structurally isomorphic* to $\mathcal{H}_2$ with regard to a bijection $\sigma : (O_{\mathcal{H}_1} \cup Var_{\mathcal{H}_1}) \to (O_{\mathcal{H}_2} \cup Var_{\mathcal{H}_2})$, if for all objects $o \in O_{\mathcal{H}_1}$ and fields $f \in \mathcal{F}$, it holds that:

同构

- $o \in AO_{\mathcal{H}_1}$ iff $\sigma(o) \in AO_{\mathcal{H}_2}$, and $\sigma(\texttt{null}) = \texttt{null}$;
- $\delta_{\mathcal{H}_1}(o, f) = o' \in O_{\mathcal{H}_1}$ iff $\delta_{\mathcal{H}_2}(\sigma(o), f) = \sigma(o') \in O_{\mathcal{H}_2}$;
- $\delta_{\mathcal{H}_1}(o, f) = v \in Var_{\mathcal{H}_1}$ iff $\delta_{\mathcal{H}_2}(\sigma(o), f) = \sigma(v) \in Var_{\mathcal{H}_2}$.

For two isomorphic abstract states $\mathcal{H}_1$ and $\mathcal{H}_2$ with regard to bijection σ, their union abstract state $\mathcal{H} = \mathbf{merge}_\sigma(\mathcal{H}_1, \mathcal{H}_2)$ can be obtained by creating a new abstract state $\mathcal{H}$ identical to $\mathcal{H}_2$ but only the state constraint $\varphi_\mathcal{H}$ to be the disjunction of $\varphi_{\mathcal{H}_1}[v := \sigma(v)]$ and $\varphi_{\mathcal{H}_2}$. The notation $\varphi_{\mathcal{H}_1}[v := \sigma(v)]$ indicates substitution of $\sigma(v)$ for free variable $v \in Var_{\mathcal{H}_1}$ in formula $\varphi_{\mathcal{H}_1}$. For example, assume that $\varphi_{\mathcal{H}_1}$ is $\exists x.\, u = 2x$ while $\varphi_{\mathcal{H}_2}$ is $\exists y.\, v = 2y + 1$, and the bijection σ maps variable u to v. The constraint $\varphi_\mathcal{H}$ of the union abstract state can be computed as follows:

$$\varphi_\mathcal{H} = \varphi_{\mathcal{H}_1}[u := v] \vee \varphi_{\mathcal{H}_2} = (\exists x.\, v = 2x) \vee (\exists y.\, v = 2y + 1)$$

4.1.2 Symbolic Execution. Since we introduce the notion of abstract heap state, we need to know how to obtain the abstract post-state of executing an (abstract) method call on an abstract pre-state. This problem can be solved by means of *symbolic execution*.

Symbolic execution is a program analysis technique that determines what inputs would cause each control flow path of a program to execute. Symbolic execution engines regard the inputs of a program as symbolic variables, explore multiple paths simultaneously, and maintain for each explored path: (1) a *path condition* that describes the conditions satisfied by the branches taken along this path, and (2) a *symbolic store* that maps variables to symbolic expressions or values, which is updated by assignments [5].