

Bogor/Kiasan

Combining Symbolic Execution, Model Checking, Theorem Proving, and Constraint Solving for Reasoning about Open Systems

ESF Workshop on Challenges in Java Program Verification
October 16 - 18, 2006, Nijmegen, Netherlands

Robby

Xianghua Deng

Jooyong Lee
(BRICS, DK)

SAnToS Laboratory, Kansas State University

Supports

IBM Eclipse Innovation Grant
US Army Research Office (ARO)
US National Science Foundation (NSF)

Lockheed Martin Advanced Technology Laboratories
US Air Force of Scientific Research (AFOSR)

Outline

- Features of Bogor/Kiasan
- Underlying techniques used in Bogor/
Kiasan
- Challenges and open research problems

Outline

- Features of Bogor/Kiasan
- Underlying techniques used in Bogor/
Kiasan
- Challenges and open research problems

Overview of Bogor/Kiasan

- “Kiasan” = “symbolic”; it combines several analysis techniques
- Trading-off *full* soundness (bounding) for *automatic* checking
- Focus: *improving* code quality, automating and *aiding* some tasks of software developers
- Features / capabilities
 - can check strong (heap-oriented) properties
 - bytecode-level analysis (*our* Java memory model)
 - quantifiable control over analysis cost/coverage
 - helpful analysis feedback to help justify/evaluate Kiasan

Checking Unspecified Code

- Similar to ESC/Java, Kiasan can detect possible uncaught runtime exceptions

```
class Node {  
    Node next;  
    int x;  
}  
  
void foo(Node n) {  
    if (n.x > 0) {  
        n.x++;  
        return;  
    }  
    n.x = 0;  
}
```

Checking Unspecified Code

- Similar to ESC/Java, Kiasan can detect possible uncaught runtime exceptions

```
class Node {  
    Node next;  
    int x;  
}  
  
void foo(Node n) {  
    if (n.x > 0) {  
        n.x++;  
        return;  
    }  
    n.x = 0;  
}
```

Possible null dereference

Checking “Unspecified” Code

- Consider the following example:

```
void bar(Node n1, Node n2) {  
    if (n1 != null && n2 != null) {  
        n1.x = 2;  
        n2.x = 3;  
        assert n1.x == 2 && n2.x == 3;  
    }  
}
```

Checking “Unspecified” Code

- Consider the following example:

```
void bar(Node n1, Node n2) {  
    if (n1 != null && n2 != null) {  
        n1.x = 2;  
        n2.x = 3;  
        assert n1.x == 2 && n2.x == 3;  
    }  
}
```

Possible assertion violation

Checking “Unspecified” Code

- Not only that Kiasan can detect assertion violations, it can even generate the error scenarios under which they happen

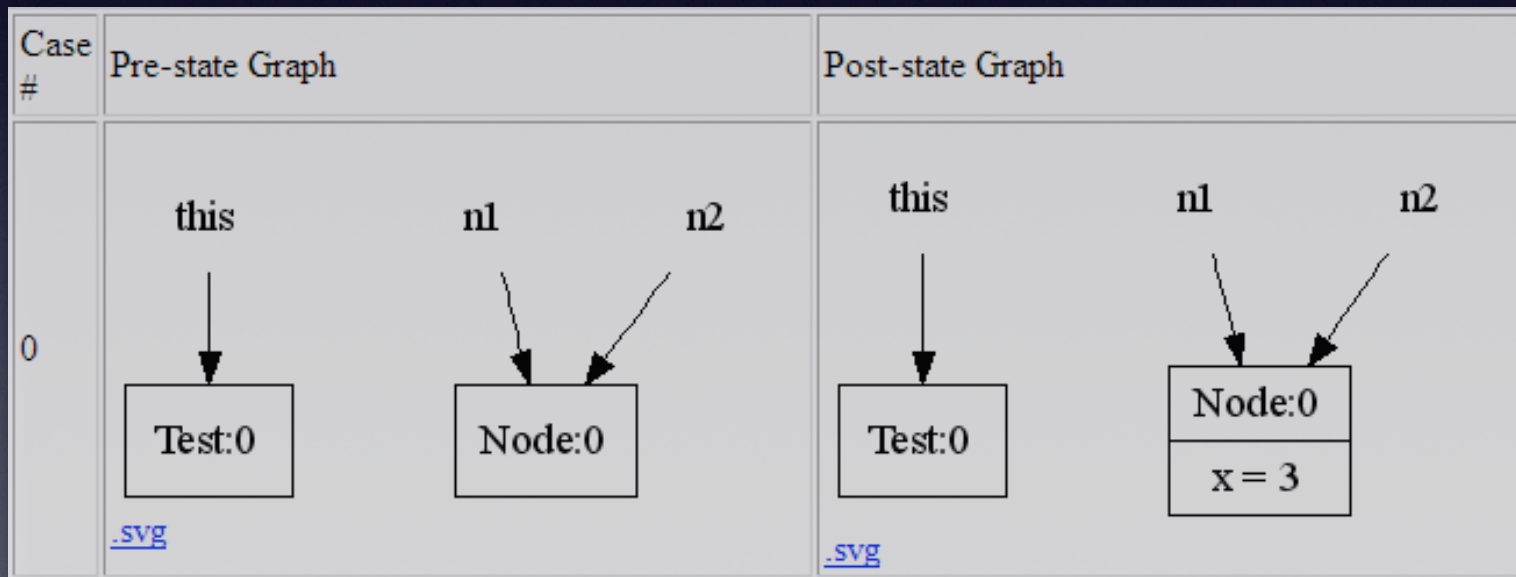
```
void bar(Node n1, Node n2) {  
    if (n1 != null && n2 != null) {  
        n1.x = 2;  
        n2.x = 3;  
        assert n1.x == 2 && n2.x == 3;  
    }  
}
```

Error Scenario:



Kiasan's Visualization

- Kiasan generates object graphs (useful for aiding code inspection/understanding)

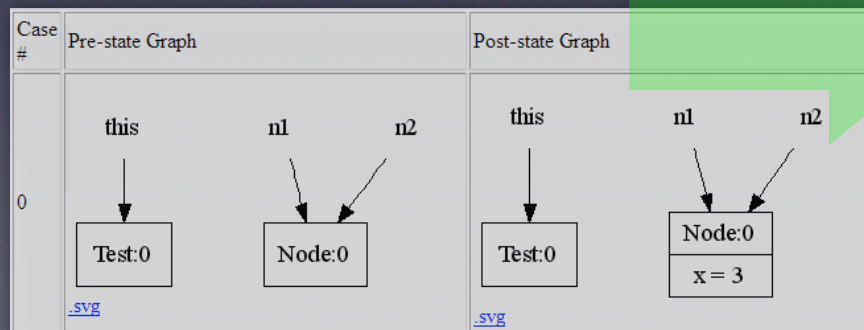


Kiasan's Test Case Generation

- Kiasan generates (JUnit) test cases (useful for regression testing, white-box testing)

```
void bar(Node n1, Node n2) {  
    if (n1 != null && n2 != null) {  
        n1.x = 2;  
        n2.x = 3;  
        assert n1.x == 2 && n2.x == 3;  
    }  
}
```

```
import junit.framework.*;  
import java.lang.reflect.Field;  
  
public class BarTest0 extends TestCase {  
  
    public void testBar() throws Exception {  
  
        Test this_;  
        Class clazzTest = Class.forName("Test");  
        this_ = (Test) clazzTest.newInstance();  
  
        Node n1;  
        Class clazzNode = Class.forName("Node");  
        n1 = (Node) clazzNode.newInstance();  
  
        Node n2;  
        n2 = n1;  
        this_.bar(n1, n2);  
    }  
}
```



Leveraging Design Intentions

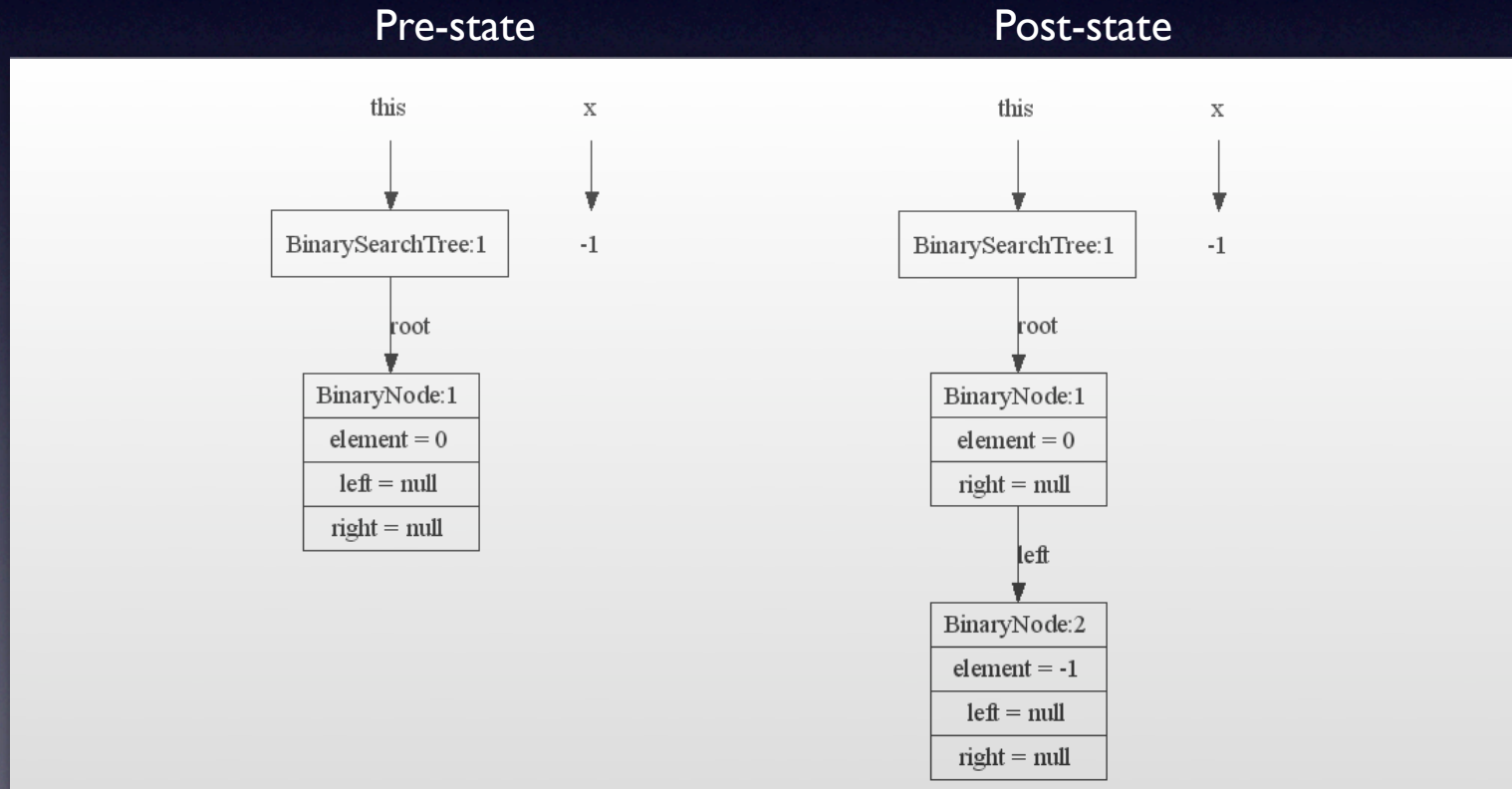
- Kiasan can leverage design intentions that are codified in Java (black-box testing)

```
@Assertion(@Case(  
    pre = "repOK(t)",  
    post = "repOK($ret)")  
private BinaryNode myins(int x,  
                          BinaryNode t) {  
    if (t == null)  
        t = new BinaryNode(x,  
                            null, null);  
    else if (x < t.element)  
        t.left = myins(x, t.left);  
    else if (x > t.element)  
        t.right = myins(x, t.right);  
    else  
        ; // Duplicate; do nothing  
  
    return t;  
}
```

```
boolean repOK(BinaryNode t) {  
    return repOK(t, new Range());  
}  
  
boolean repOK(BinaryNode t,  
              Range range) {  
    if (t == null)  
        return true;  
  
    if (!range.inRange(t.element))  
        return false;  
  
    boolean ret = repOK(t.left,  
                       range.setUpper(t.element));  
    ret = ret && repOK(t.right,  
                      range.setLower(t.element));  
    return ret;  
}
```

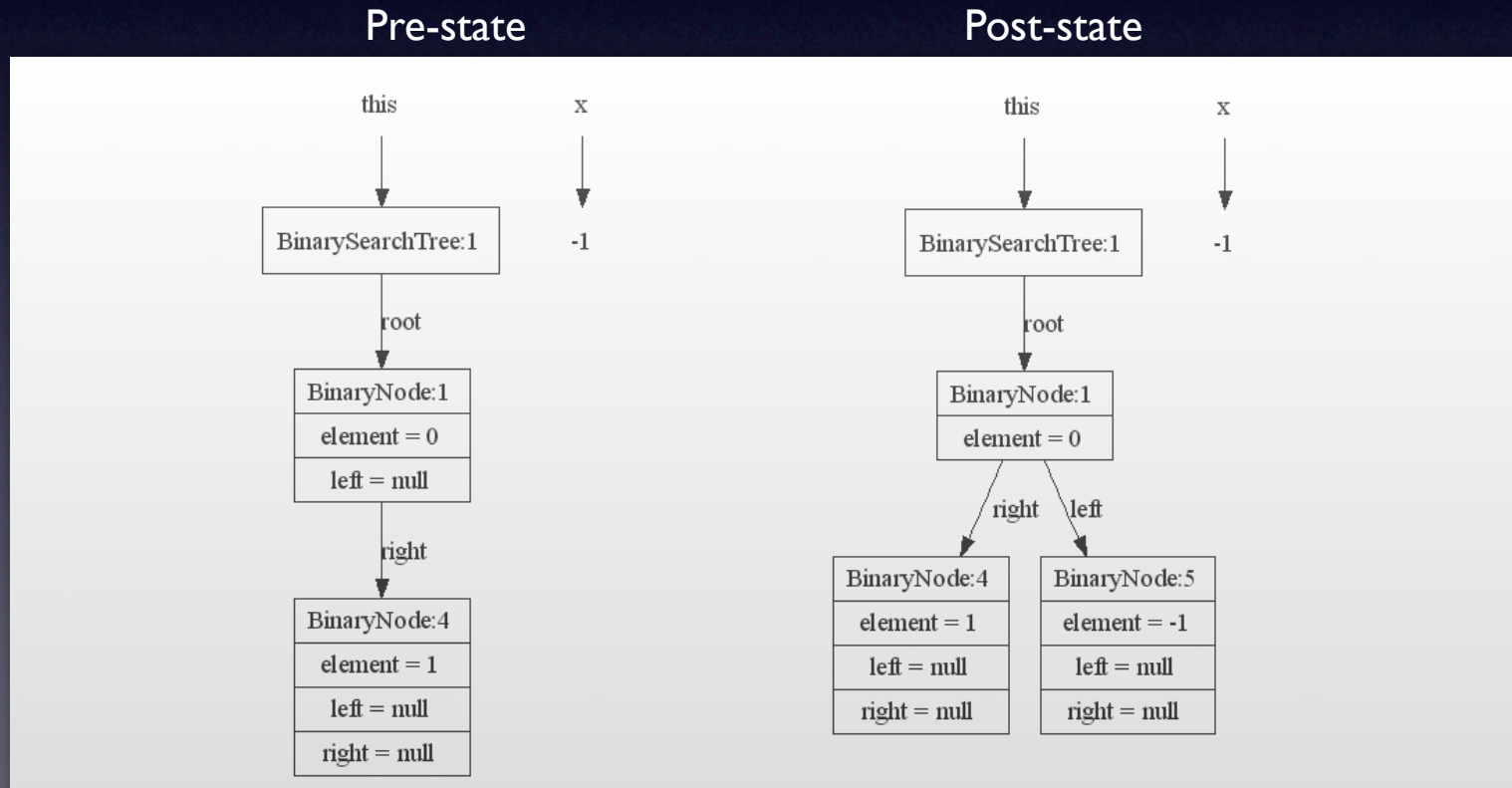
Leveraging Design Intentions

- Kiasan generates test cases based on the given repOK for the insert method



Leveraging Design Intentions

- Kiasan generates test cases based on the given repOK for the insert method



Checking Strong Properties

- Kiasan can check strong heap-oriented properties that even relate pre-/post-states

```
public class LinkedList<E> {
    //@ inv: isAcyclic();

    /*@ pre:  isSorted(c) && other.isSorted(c);
       @ post: isSorted(c)
       @      && size() = \old(size()) + other.size()
       @      && (\forall E e;
       @          elements.contains(e);
       @          \old(this).contains(e)
       @          || other.contains(e))
       @*/
    void merge(@NonNull LinkedList<E> other,
               @NonNull Comparator<E> c) {
        ...
    }
}
```

Measuring Behavior Coverage

- During analysis, Kiasan computes coverage metrics (statement, branch)
 - this includes coverage on specifications
- Its analysis can even be driven by the coverage metrics
 - i.e., stop the analysis as soon as the desirable coverage is achieved
 - reasonable cost/coverage trade-off

Outline

- Features of Bogor/Kiasan
- Underlying techniques used in Bogor/
Kiasan
- Challenges and open research problems

Symbolic Execution

[King:ACM76]

```
void foo(int x,  
         int y,int z)  
{  
    z = x + y;  
    if (z > 0){  
        z++;  
    }  
}
```

Symbolic Execution

[King:ACM76]

```
void foo(int x,  
         int y,int z)  
{  
  z = x + y;  
  if (z > 0){  
    z++;  
  }  
}
```

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

Symbolic Execution

[King:ACM76]

```
void foo(int x,  
         int y,int z)  
{  
  z = x + y;  
  if (z > 0){  
    z++;  
  }  
}
```

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

$z = x + y;$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta\}$

Symbolic Execution

[King:ACM76]

```
void foo(int x,  
         int y,int z)  
{  
  z = x + y;  
  if (z > 0){  
    z++;  
  }  
}
```

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta\}$

$z > 0$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma > 0\}$

Symbolic Execution

[King:ACM76]

```
void foo(int x,  
         int y,int z)  
{  
  z = x + y;  
  if (z > 0){  
    z++;  
  }  
}
```

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma > 0\}$

$z++;$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma', \Phi = \{\gamma = \alpha + \beta, \gamma > 0, \gamma' = \gamma + 1\}$

Symbolic Execution [King:ACM76]

```
void foo(int x,  
         int y,int z)  
{  
  z = x + y;  
  if (z > 0){  
    z++;  
  }  
}
```

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta\}$

$!(z > 0)$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma > 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma', \Phi = \{\gamma = \alpha + \beta, \gamma > 0, \gamma' = \gamma + 1\}$

Symbolic Execution [King:ACM76]

```
void foo(int x,  
         int y,int z)  
{  
  z = x + y;  
  if (z > 0){  
    z++;  
  }  
}
```

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma > 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma', \Phi = \{\gamma = \alpha + \beta, \gamma > 0, \gamma' = \gamma + 1\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

skip

Symbolic Execution

[King:ACM76]

```
void foo(int x,  
         int y,int z)  
{  
  z = x + y;  
  if (z > 0){  
    z++;  
  }  
}
```

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma > 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma', \Phi = \{\gamma = \alpha + \beta, \gamma > 0, \gamma' = \gamma + 1\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

...symbolic execution characterizes (theoretically) infinite number of real executions!

Test Case Generation

```
void foo(int x,  
         int y,int z)  
{  
  z = x + y;  
  if (z > 0){  
    z++;  
  }  
}
```

$x=-1, y=2, z=0$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

Solving the constraint
of each path's Φ to
generate a test case

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma > 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma', \Phi = \{\gamma = \alpha + \beta, \gamma > 0, \gamma' = \gamma + 1\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

$x=-1, y=2, z=2$

... the explored computation tree can be directly leveraged for test case generation!

Test Case Generation

```
void foo(int x,  
         int y,int z)  
{  
    z = x + y;  
    if (z > 0){  
        z++;  
    }  
}
```

$x=-1, y=0, z=0$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \delta, \Phi = \{\}$

Solving the constraint
of each path's Φ to
generate a test case

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma > 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma', \Phi = \{\gamma = \alpha + \beta, \gamma > 0, \gamma' = \gamma + 1\}$

$x \leftarrow \alpha, y \leftarrow \beta, z \leftarrow \gamma, \Phi = \{\gamma = \alpha + \beta, \gamma \leq 0\}$

$x=-1, y=0, z=-1$

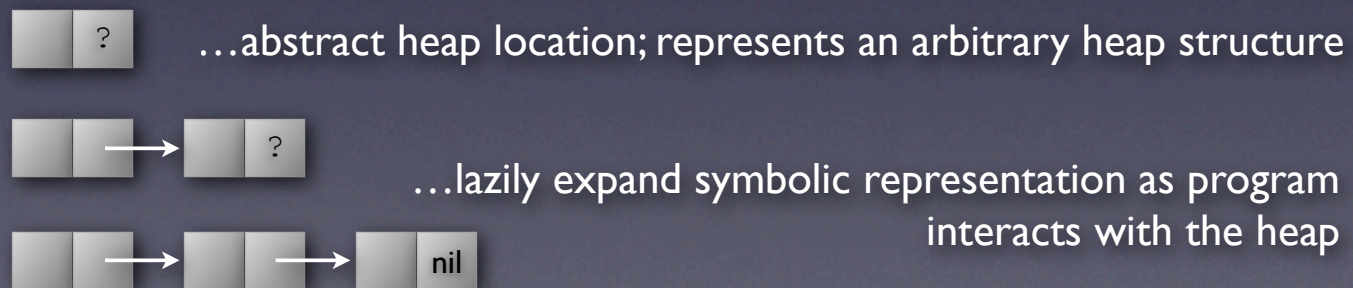
... the explored computation tree can be directly leveraged for test case generation!

Issues in Symbolic Execution: Loops

- How do we know when to quit going around a loop?
 - could leverage loop invariants, but they are difficult to obtain for several reasons
 - common strategy is to use different forms of bounds
 - bound the total number of steps, or
 - bound the number of loop iterations

Issues in Symbolic Execution: Objects [Khurshid-al:TACAS03]

- How should dynamically allocated heap data be represented in symbolic execution?
- ...leverage recent advancements on model checking OO-systems strong properties, i.e., maintains a precise representation of the heap
- hybrid concrete and symbolic representation



Issues in Symbolic Execution: Objects [Khurshid-al:TACAS03]

- How should dynamically allocated heap data be represented in symbolic execution?
- ...leverage recent advancements on model checking OO-systems strong properties, i.e., maintains a precise representation of the heap
- hybrid concrete and symbolic representation

α ? ...abstract heap location; represents an arbitrary heap structure

α \rightarrow β ? ...lazily expand symbolic representation as program interacts with the heap

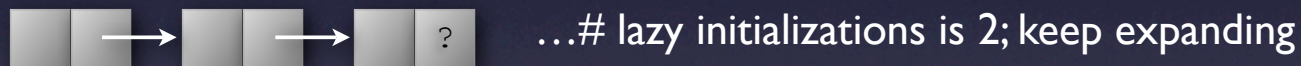
α \rightarrow β \rightarrow γ nil

...use conventional symbolic constraints on scalar values in heap

Kiasan's k -bounding

- Bound search by bounding the number of “fresh” lazy initializations

Limit $k = 3$



Kiasan's k -bounding

Limit $k = 2$

```
o = head;
while (o != null) {
  if (V.contains(o))
    break;
  V.add(o);
  o = o.next;
}
```

Kiasan's k -bounding

Limit $k = 2$

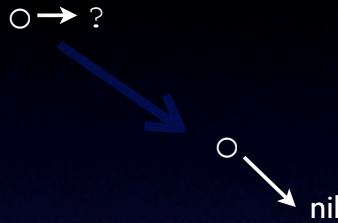
$o \rightarrow ?$

```
o = head;
while (o != null) {
  if (V.contains(o))
    break;
  V.add(o);
  o = o.next;
}
```

Kiasan's k -bounding

Limit $k = 2$

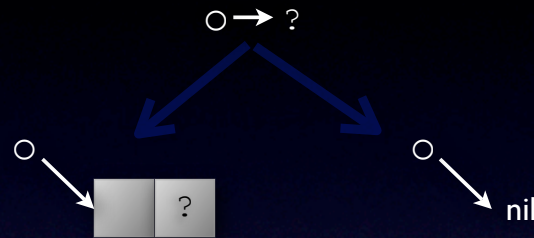
```
o = head;  
while (o != null) {  
  if (V.contains(o))  
    break;  
  V.add(o);  
  o = o.next;  
}
```



Kiasan's k -bounding

Limit $k = 2$

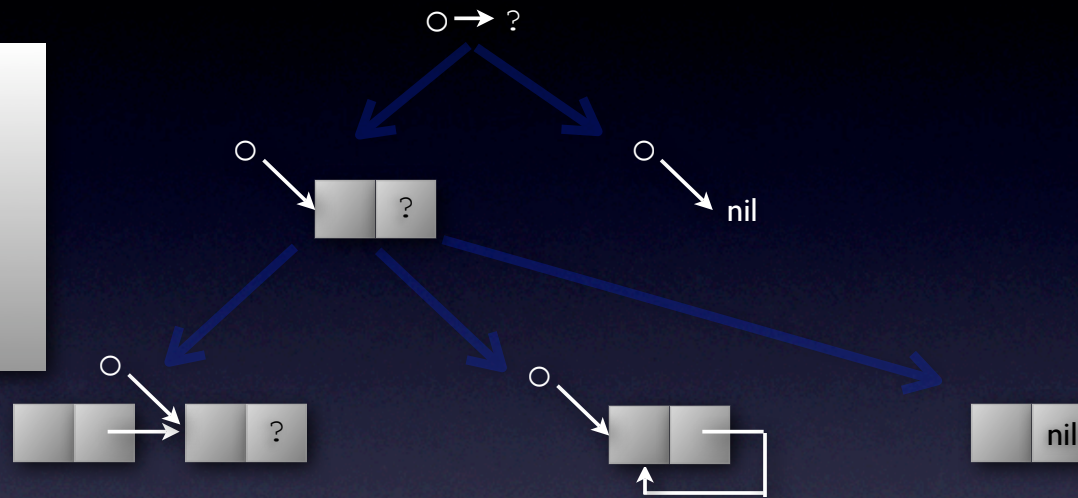
```
o = head;
while (o != null) {
  if (V.contains(o))
    break;
  V.add(o);
  o = o.next;
}
```



Kiasan's k -bounding

Limit $k = 2$

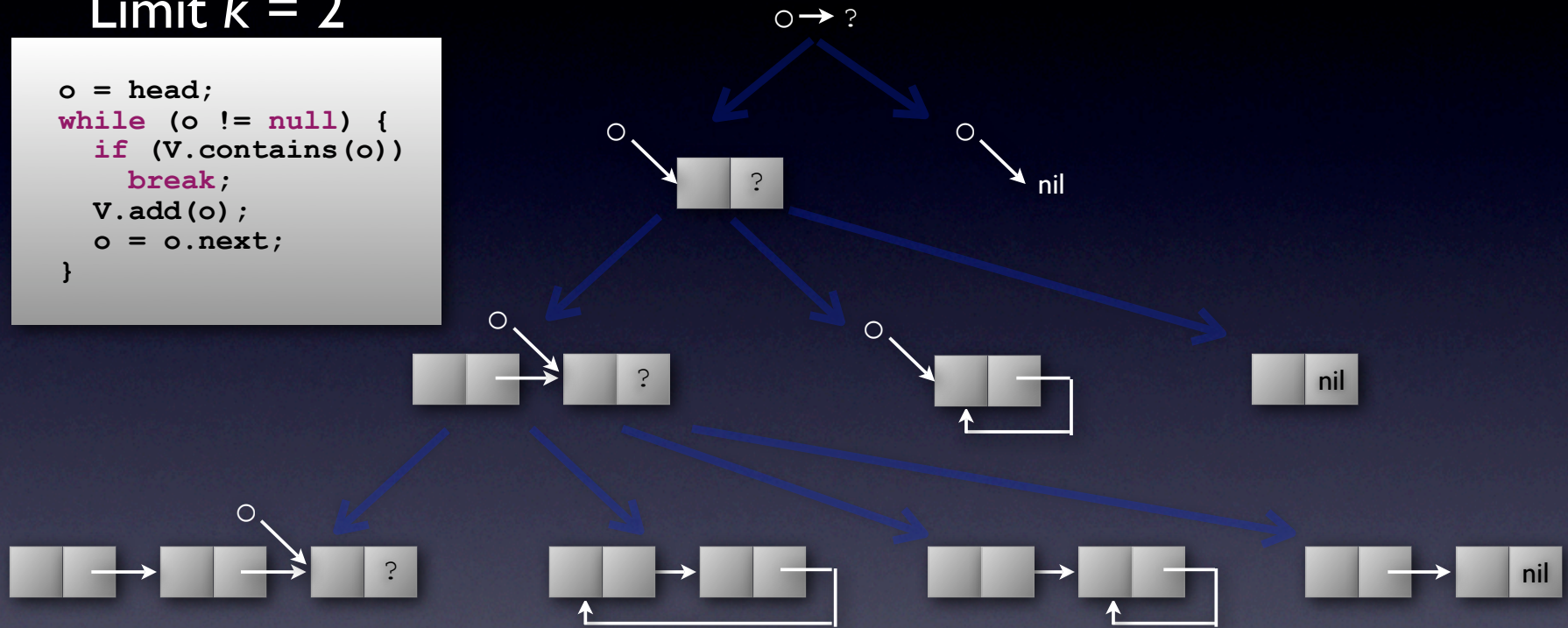
```
o = head;
while (o != null) {
  if (V.contains(o))
    break;
  V.add(o);
  o = o.next;
}
```



Kiasan's k -bounding

Limit $k = 2$

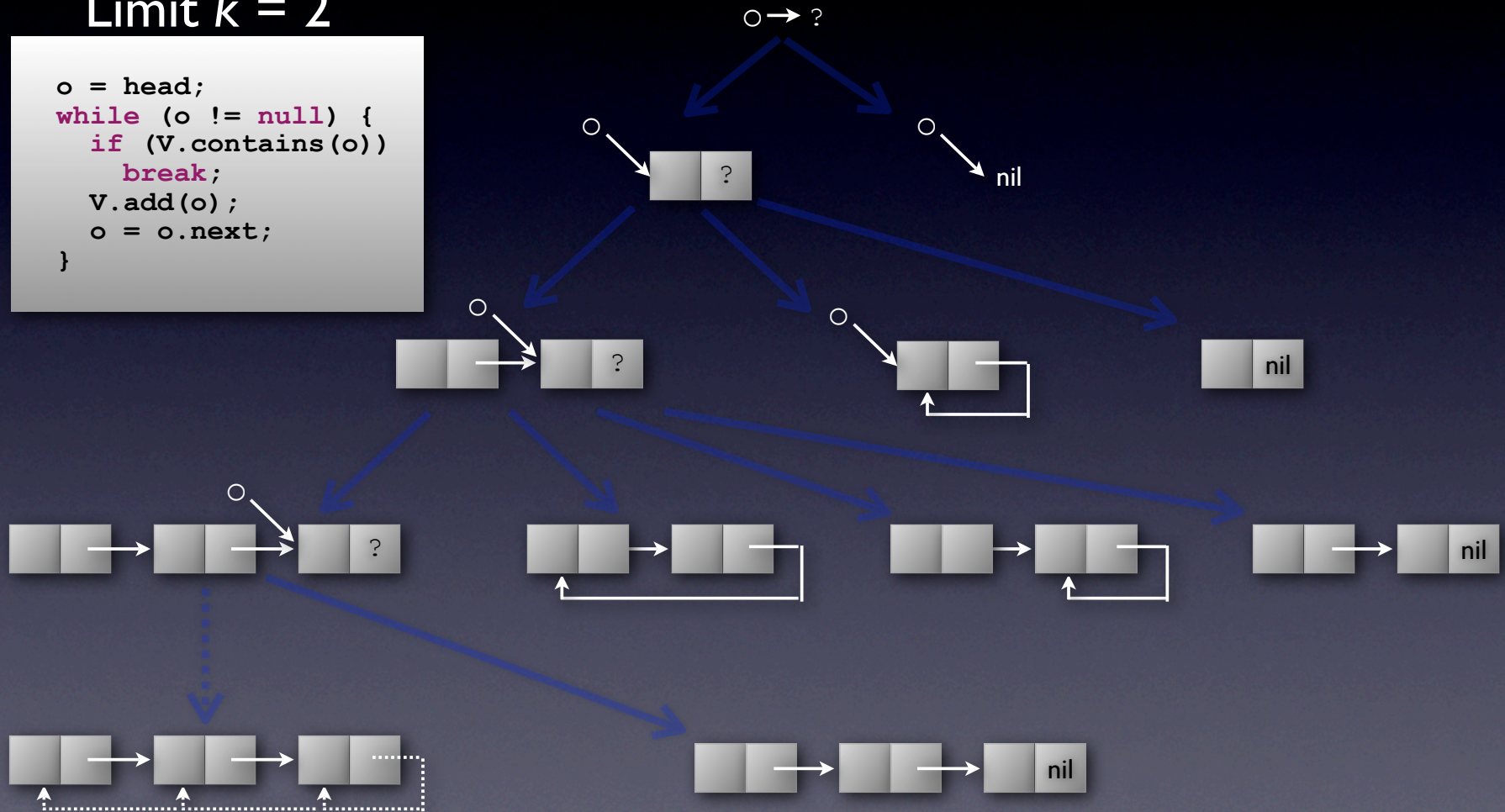
```
o = head;  
while (o != null) {  
  if (V.contains(o))  
    break;  
  V.add(o);  
  o = o.next;  
}
```



Kiasan's k -bounding

Limit $k = 2$

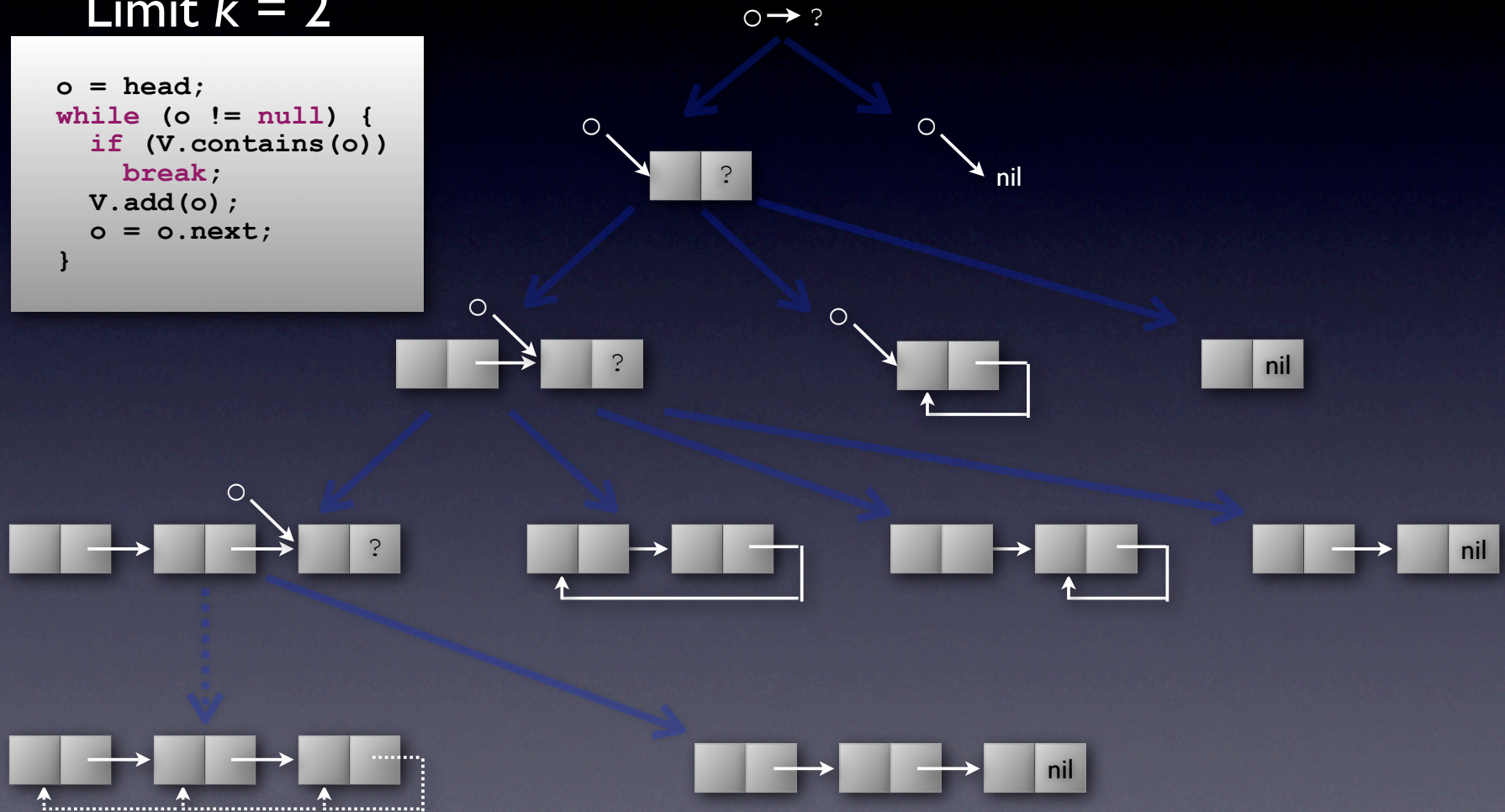
```
o = head;  
while (o != null) {  
  if (V.contains(o))  
    break;  
  V.add(o);  
  o = o.next;  
}
```



Kiasan's k -bounding

Limit $k = 2$

```
o = head;  
while (o != null) {  
  if (V.contains(o))  
    break;  
  V.add(o);  
  o = o.next;  
}
```



... the generated object graphs are directly leverage-able for test case generations!

Assessment

- The k -bounding mechanism gives *quantifiable* behavior coverage
- Better *object* coverage than loop bounding (Kiasan uses this whenever it can)
- If the bounds are not exhausted, then we have “full” behavior coverage
- ... reasonable cost/coverage trade-off

More Assessment

- Execution => Kiasan is *aware* whenever its bounds are exhausted (thus, can provide helpful feedback on unexplored cases)
- Bounding => Kiasan does not need to depend on storing state spaces to terminate its algorithm
 - stateless search
 - “embarrassingly” parallel
 - i.e., can be efficiently parallelized/distributed

Kiasan Methodology



- We envision that we use small bounds for spec'n check mode to find shallow errors in a Java IDE
- Increase coverage and use “continuous testing” (Ernst et al.) + distributed computing
- Leverage specification for code inspection/ understanding, debugging, testing, etc.

Outline

- Features of Bogor/Kiasan
- Underlying techniques used in Bogor/
Kiasan
- Challenges and open research problems

Compositional Checking Using Symbolic Execution

Pre-condition

```
M(..., ..., ...) {  
  ....  
  
  N(.....)  
  
}
```

Post-condition

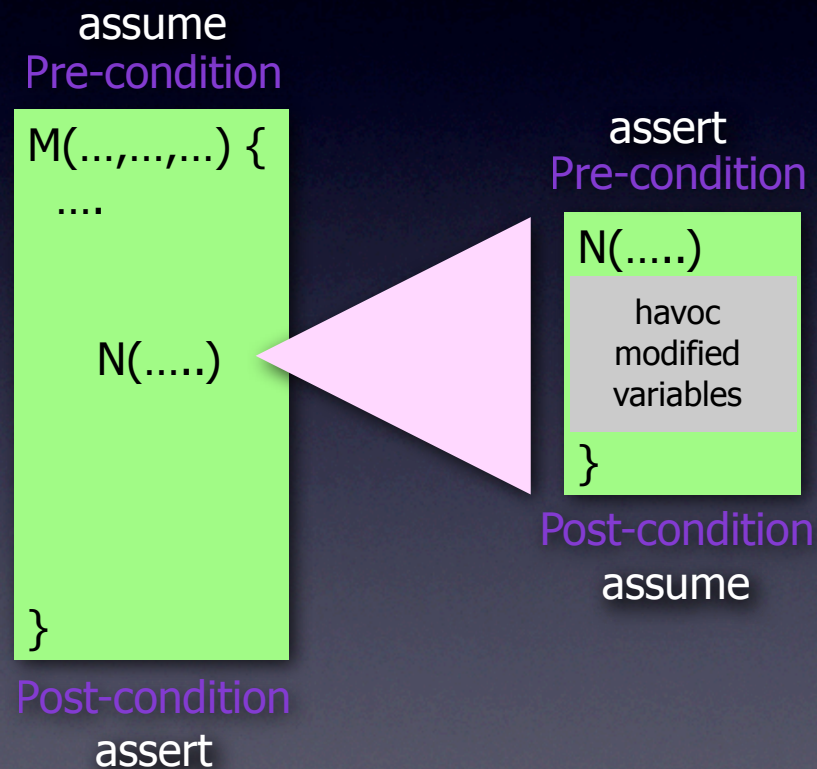
Pre-condition

```
N(.....)  
  
}
```

Post-condition

...we require executable (effective) pre-/post-conditions!

Compositional Checking Using Symbolic Execution



...we require executable (effective) pre-/post-conditions!

Challenges in Checking Strong Properties of Open Systems

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
  
    /*@ pre: isSorted(c)  
    @   && other.isSorted(c);  
    @ post: isSorted(c)  
    @   && size() = \old(size())  
    @   + other.size()  
    @   && (\forall E e;  
    @     elements.contains(e);  
    @     \old(this).contains(e)  
    @     || other.contains(e))  
    @*/  
    void merge(@NonNull  
               LinkedList<E> other,  
               @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- Handling open-ended OO-methods
- Reducing up-front specification cost
- Checking strong heap properties
- Detecting “interference” due to object aliasing
- Object abstraction/refine.

Challenges: Open OO-Systems

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
    /*@ pre: isSorted(c)  
    @   && other.isSorted(c);  
    @ post: isSorted(c)  
    @     && size() = \old(size())  
    @       + other.size()  
    @     && (\forall E e;  
    @       elements.contains(e);  
    @       \old(this).contains(e)  
    @       || other.contains(e))  
    @*/  
    void merge(@NonNull  
              LinkedList<E> other,  
              @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- Comparator is open-ended
- we do not know the actual implementation (or even if it exists at analysis time)
- i.e., its computation structure is incomplete

Challenges: Open OO-Systems

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
    /*@ pre: isSorted(c)  
    @      && other.isSorted(c);  
    @ post: isSorted(c)  
    @      && size() = \old(size())  
    @      + other.size()  
    @      && (\forall E e;  
    @          elements.contains(e);  
    @          \old(this).contains(e)  
    @          || other.contains(e))  
    @*/  
    void merge(@NonNull  
               LinkedList<E> other,  
               @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- This presents greater challenges than systems that
 - only use scalar values or immutable objects, where compare's computation structure is known
 - we do not know what objects in the heap will be used for compare

Challenges: Open OO-Systems

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
    /*@ pre: isSorted(c)  
    @      && other.isSorted(c);  
    @ post: isSorted(c)  
    @      && size() = \old(size())  
    @      + other.size()  
    @      && (\forall E e;  
    @          elements.contains(e);  
    @          \old(this).contains(e)  
    @          || other.contains(e))  
    @*/  
    void merge(@NonNull  
               LinkedList<E> other,  
               @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- Can only depend on the design intention of compare (in API doc.)
- returns negative, zero, or positive
- total order
- pure method

Challenges: Up-front Annotation Cost

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
  
    /*@ pre: isSorted(c)  
       @   && other.isSorted(c);  
       @ post: isSorted(c)  
       @   && size() = \old(size())  
       @   + other.size()  
       @   && (\forall E e;  
       @     elements.contains(e);  
       @     \old(this).contains(e)  
       @     || other.contains(e))  
    @*/  
    void merge(@NonNull  
               LinkedList<E> other,  
               @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- In pure compositional reasoning, we need to supply the specification for addLast (and compare)
- Thus, it imposes an up-front specs (effort)

Challenges: Up-front Annotation Cost

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
  
    /*@ pre: isSorted(c)  
       @   && other.isSorted(c);  
       @ post: isSorted(c)  
       @   && size() = \old(size())  
       @   + other.size()  
       @   && (\forall E e;  
       @     elements.contains(e);  
       @     \old(this).contains(e)  
       @     || other.contains(e))  
    @*/  
    void merge(@NonNull  
              LinkedList<E> other,  
              @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- We would like to be able to substitute implementation with specification and vice versa (and still be scalable)
- Thus, we can focus on specifying more important parts of the system and gradually close out under-approximation

Challenges: Strong Properties

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
  
    /*@ pre: isSorted(c)  
       @   && other.isSorted(c);  
       @ post: isSorted(c)  
       @   && size() = \old(size())  
       @   + other.size()  
       @   && (\forall E e;  
       @     elements.contains(e);  
       @     \old(this).contains(e)  
       @     || other.contains(e))  
    @*/  
    void merge(@NonNull  
               LinkedList<E> other,  
               @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- The size of the resulting list is the sum of the two input lists
- The elements of the resulting list are from the two input lists (and only from those lists)

Challenges: Strong Properties

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
  
    /*@ pre: isSorted(c)  
       @   && other.isSorted(c);  
       @ post: isSorted(c)  
       @   && size() = \old(size())  
       @   + other.size()  
       @   && (\forall E e;  
       @     elements.contains(e);  
       @     \old(this).contains(e)  
       @     || other.contains(e))  
    @*/  
    void merge(@NonNull  
               LinkedList<E> other,  
               @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- Difficult to use abstraction techniques that summarize heap objects
 - need to maintain the elements of lists (at pre-/post-states and whose number may be unbounded)
 - one might be able to manually supply the most precise abstraction to reason about this
 - however, we cannot anticipate the kinds of properties that a user wants to check

Challenges: “Interference”

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
  
    /*@ pre: isSorted(c)  
       @   && other.isSorted(c);  
       @ post: isSorted(c)  
       @   && size() = \old(size())  
       @   + other.size()  
       @   && (\forall E e;  
       @       elements.contains(e);  
       @       \old(this).contains(e)  
       @       || other.contains(e))  
    @*/  
    void merge(@NonNull  
               LinkedList<E> other,  
               @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

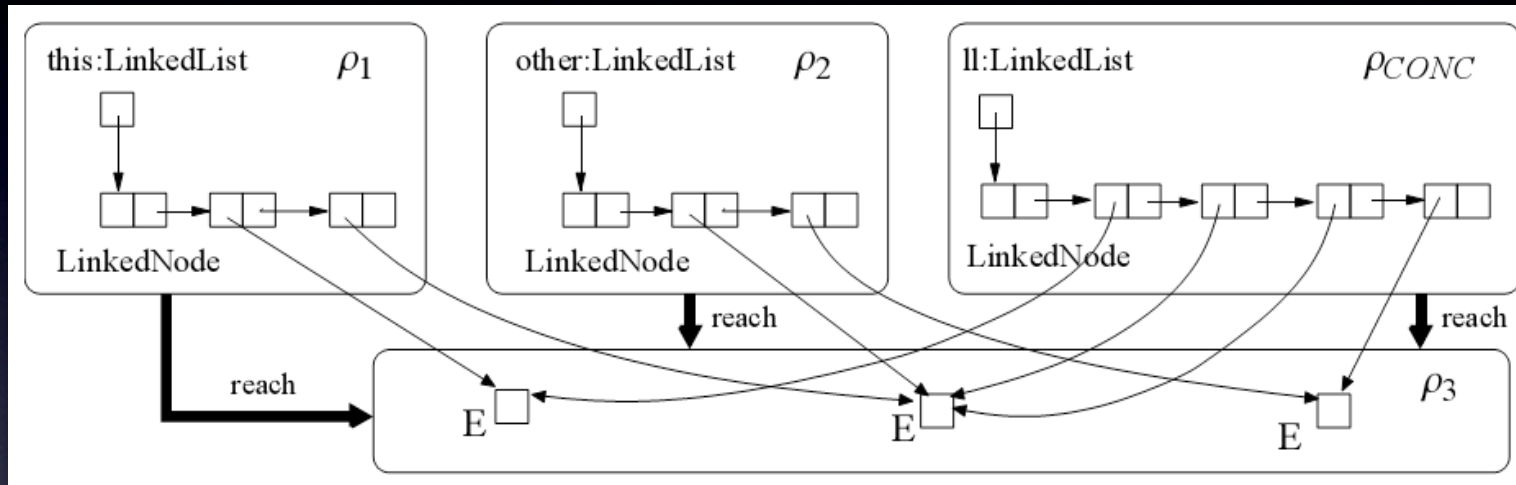
- We need to check that merge does not cause the results of compare to become stale (e.g., by a field write)
- Otherwise, post-condition will not hold

Challenges: “Interference”

```
public class LinkedList<E> {  
    //@ inv: isAcyclic();  
  
    /*@ pre: isSorted(c)  
       @   && other.isSorted(c);  
       @ post: isSorted(c)  
       @   && size() = \old(size())  
       @   + other.size()  
       @   && (\forall E e;  
       @     elements.contains(e);  
       @     \old(this).contains(e)  
       @     || other.contains(e))  
    @*/  
    void merge(@NonNull  
               LinkedList<E> other,  
               @NonNull Comparator<E> c) {  
        ...  
        c.compare(...);  
        ...  
        addLast(...);  
        ...  
    }  
}
```

- One possible approach is to use heap regions
- it is sufficient to show that merge cannot modify the heap regions of where the objects used by compare reside
- However, establishing this requires precise heap analysis that is able to leverage heap region information

Leveraging Region Spec.



- To detect changes (writes in one of the objects) in the regions, we version them
 - this is enough to check that merge does not affect compare's behavior
 - other kinds of properties may need finer-grained regions (e.g., nested) and relationships between transformed regions

Challenges: Object Abstraction/Refinement

```
void foo(Collection c, Object o1, Object o2) {  
  
    int size = c.size();  
    c.add(o1);  
    c.add(o2);  
  
    if (c instanceof LinkedList) {  
        LinkedList l = (LinkedList) c;  
        assert o1 == l.get(size);  
    }  
}
```

- Does the assertion always hold?

Conclusion

- Bogor/Kiasan offers an alternative approach for extended static checking
 - can check strong properties
 - generates helpful analysis feedback, e.g., test cases
 - relatively sound and complete analysis (e.g., under bounds, etc.)
- Big challenge: compositional reasoning of strong properties of OO-systems
- More information about Bogor/Kiasan:
<http://bogor.projects.cis.ksu.edu>