Kiasan/KUnit:
Automatic Test Case Generation and Analysis Feedback for Open Object-Oriented Systems

SAnToS Laboratory, Kansas State University, USA

William (Xianghua) Deng   Robby   John Hatcliff

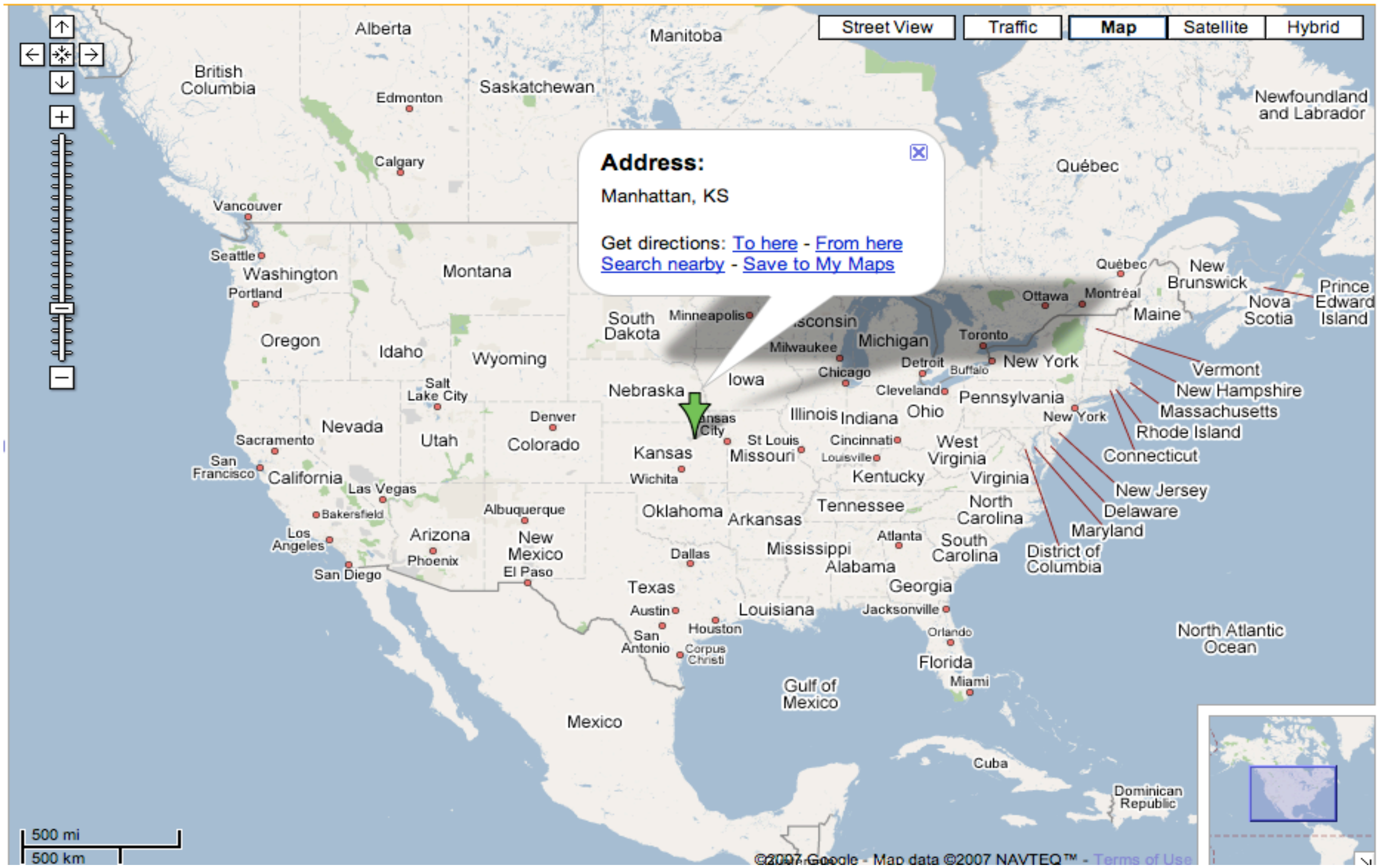
Support

US Air Force Office of Scientific Research (AFOSR)
US Army Research Office (ARO)
US National Science Foundation (NSF)

Lockheed Martin ATL (Cherry Hill, NJ)
Rockwell Collins Advanced Technology Center
IBM Eclipse

Kansas State University

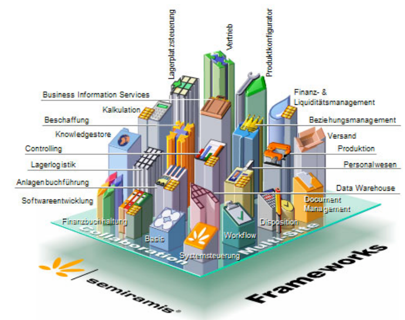
In the interest of full disclosure…

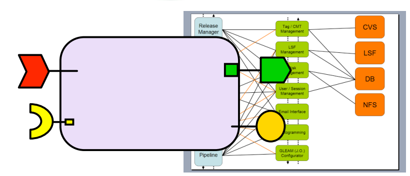


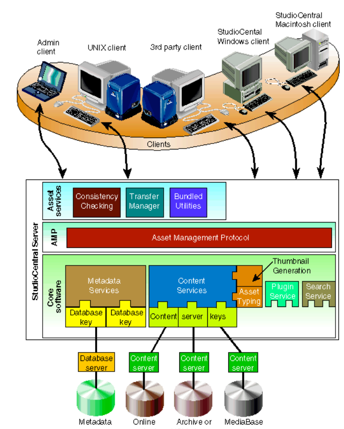
Trends In Software Development

Building Software from Reusable Units

- Frameworks
  - collection of units targeted to a particular application domain
  - Apache Struts, JavaServerFaces, CLSA
- Component Middleware
  - EJB, CCM, nesC, Bonobo
  - dictates a structure notion of reusable component
  - provides extensive infrastructure and services
- Software Product Lines
  - families of similar systems
  - reduce development time and costs through systematic reuse
  - carefully managed set of assets with clear component boundaries







Specification Language?

Java Modeling Language (JML) -- Software Contracts for Java

- Developed by Gary Leavens and colleagues
- Standard specification language for Java within the research community
  - over 100 research papers related to JML in the last seven years
- Tool support -- multiple research tools
  - static analysis, theorem proving, and runtime checking

(also see Spec# for C# from Microsoft Research)

JML Software Contracts

Lightweight Contracts

Simple pre-condition…

Post-condition requires that object bound to `last` not exist in the pre-state

…insert method for linked queue

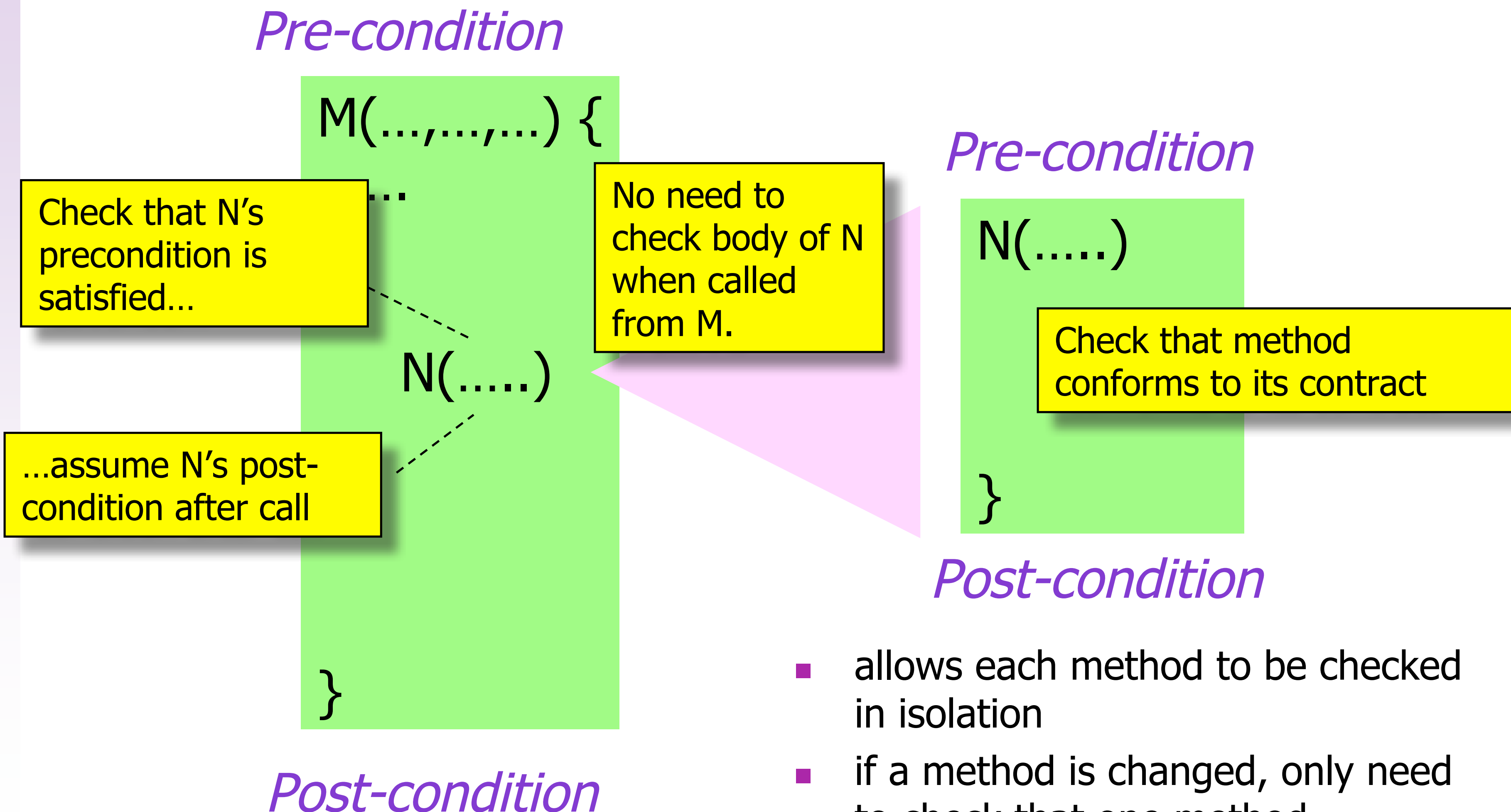
```
/*@ requires x != null;
  @ ensures last.value == x && \fresh(last);
  @*/
protected void insert(Object x) {
  synchronized (putLock) {
    LinkedNode p = new LinkedNode(x);
    synchronized (last) refactoredInsert(p);
    if (waitingForTake > 0) putLock.notify();
      return;
  }
}
```

`x` gets inserted properly

linked queue from java.util.concurrent (actually, older version from Doug Lea)

Benefits of Contracts

Contracts enable compositional checking



- allows each method to be checked in isolation
- if a method is changed, only need to check that one method

JML Checking Tools - ESC/Java

Tools like ESC/Java have made good progress toward automatic checking of lightweight JML contracts…

- Applied to large code bases
- Supports automated checking of *lightweight* method contracts
- Effective for statically eliminating many common run-time errors such as null-pointer exceptions, array bounds violations

But a number of limitations remain…

- Don’t handle heap-allocated data very well
- Feed back (e.g., error messages) provided to the user is quite poor
- No direct connection to other quality assurance techniques

JML Software Contracts

Strong Properties of Heap-allocated Data

…moving beyond ESC/Java

Frame conditions *-- only these cells can be modified.*

existential quantification over elements of a class

n's value is returned as the result

`n` *is reachable from* `head` *of the list in pre-state*

`n` *is NOT reachable from* `head` *of the list in the post-state*

```
/*@ behavior
  @   assignable head, head.next.value;
  @   ensures \result == null || (\exists LinkedNode n;
  @                              \old(\reach(head).has(n));
  @                               n.value == \result
  @                               && !(\reach(head).has(n)));
  @*/
 protected Object extract() {
   Object x = null;
   LinkedNode first = head.next;
   if (first != null) {
     x = first.value;
     first.value = null;
     head = first;
   }
   return x;
 }
```

linked list from java.util.concurrent

A Skeptic's Questions

The "hard-nosed" manager

- It takes effort to write these contracts -- what's the payoff?
  - please give me more than one way to leverage a contract!
- How can your tool and methodology be incrementally introduced into my development workflow?
- How does your approach integrate with other QA techniques my team is already trained for?
- Does this stuff scale?

Kiasan -- In a Nutshell

Contract checking for Java units with extensive heap data

Use contracts for compositional checking

Pre-condition

Pre

P(…..)

}

Post

M(…,…,…) {

….

Pre

P(…..)

Post

Q(….)

}

Q(…..)

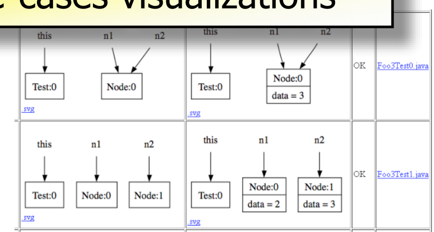
}

Post-condition

…or check using implementation directly when no contract exists

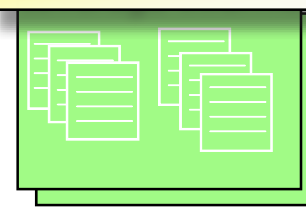
path information

Use-cases visualizations

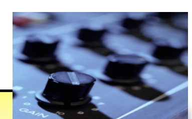


…illustrating heap shapes at input & output for each path through method

Junit unit test suite + coverage information



Controllable coverage & costs



…automatic generation of Junit tests for each (bytecode) path giving very high-levels of branch & heap configuration coverage.

Symbolic Execution [King:ACM76]

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

symbolic values

constraints

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

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

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

new symbolic value

constraint

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`

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

`z > 0`

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

new constraint for conditional

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`

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

`z > 0`

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

`z++`

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

new symbolic value

new constraint

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`

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

`z > 0`

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

`z++`

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

`!(z > 0)`

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

new constraint

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

Solving Constraints

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

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

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

`z = x + y`

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

`z > 0`

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

`z++`

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

`!(z > 0)`

$x \leftarrow \alpha,\ y \leftarrow \beta,\ z \leftarrow \pi,\ \Phi = \{\ \pi = \alpha + \beta,\ \pi \le 0\ \}$

The path condition *characterizes the set of program states that flow to this point in the path.*

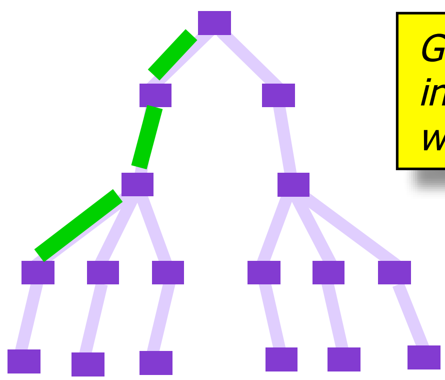
Solving constraints on input variables yields input values (a test case) that drives execution down the current path.

Dealing with Heap Data

Lazily "discover" possible shapes of data in the heap

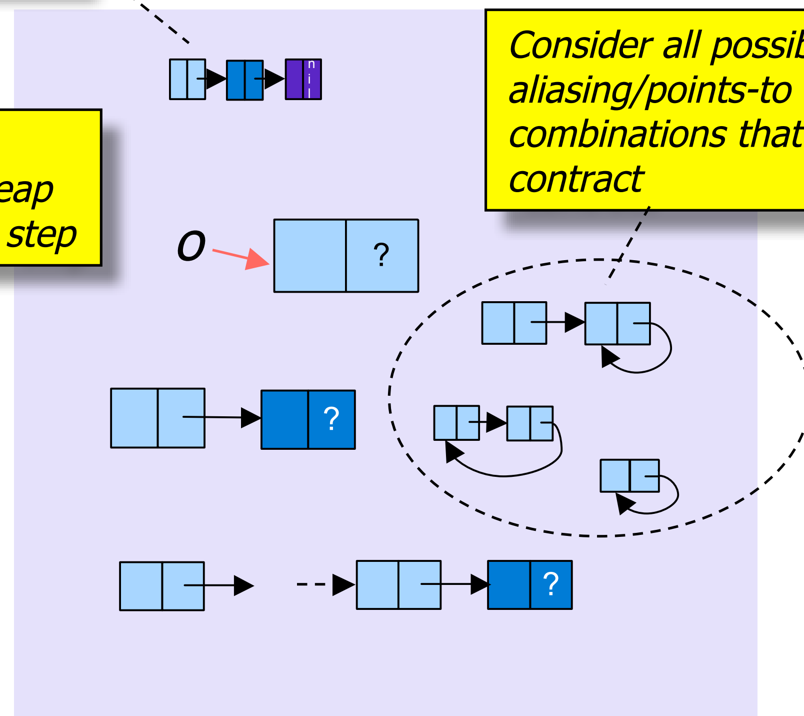
Pre-condition

Gather initial properties about the heap



Gradually uncover information about heap with each execution step

Consider all possible aliasing/points-to combinations that satisfy contract

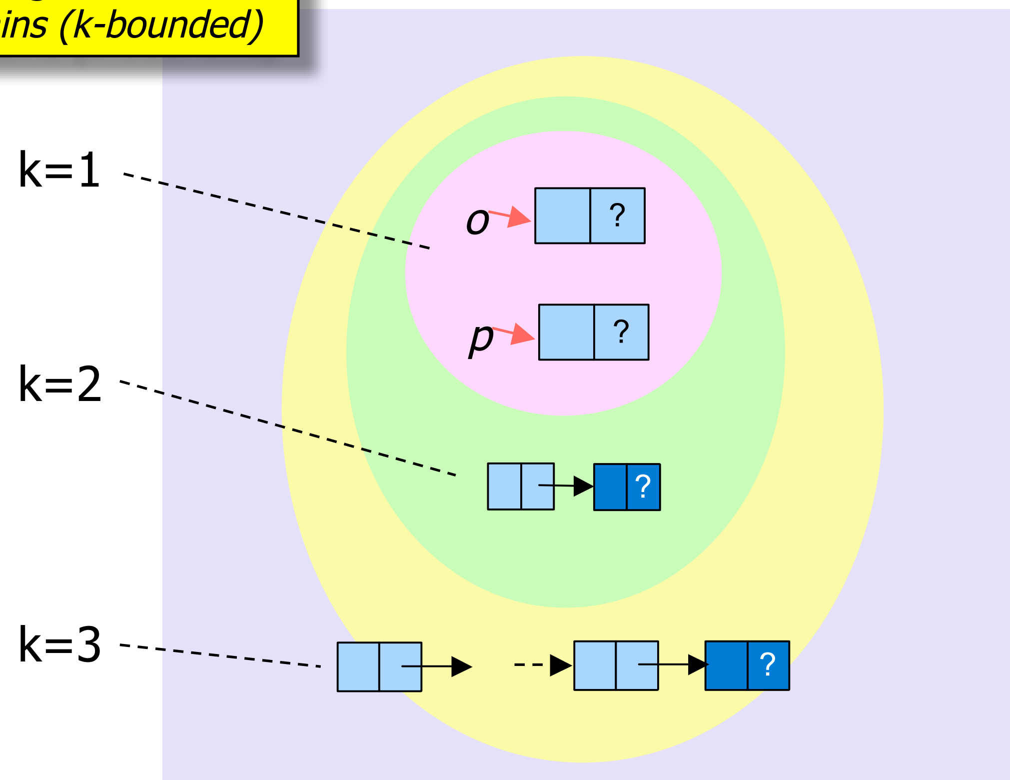


...based on Khurshid, Pasareanu, and Visser (TACAS 2003)
-- NASA JPF symbolic execution algorithm

Dealing with Heap Data

Tuneable bounds on the state-space explored

Controllable length of reference chains (k-bounded)

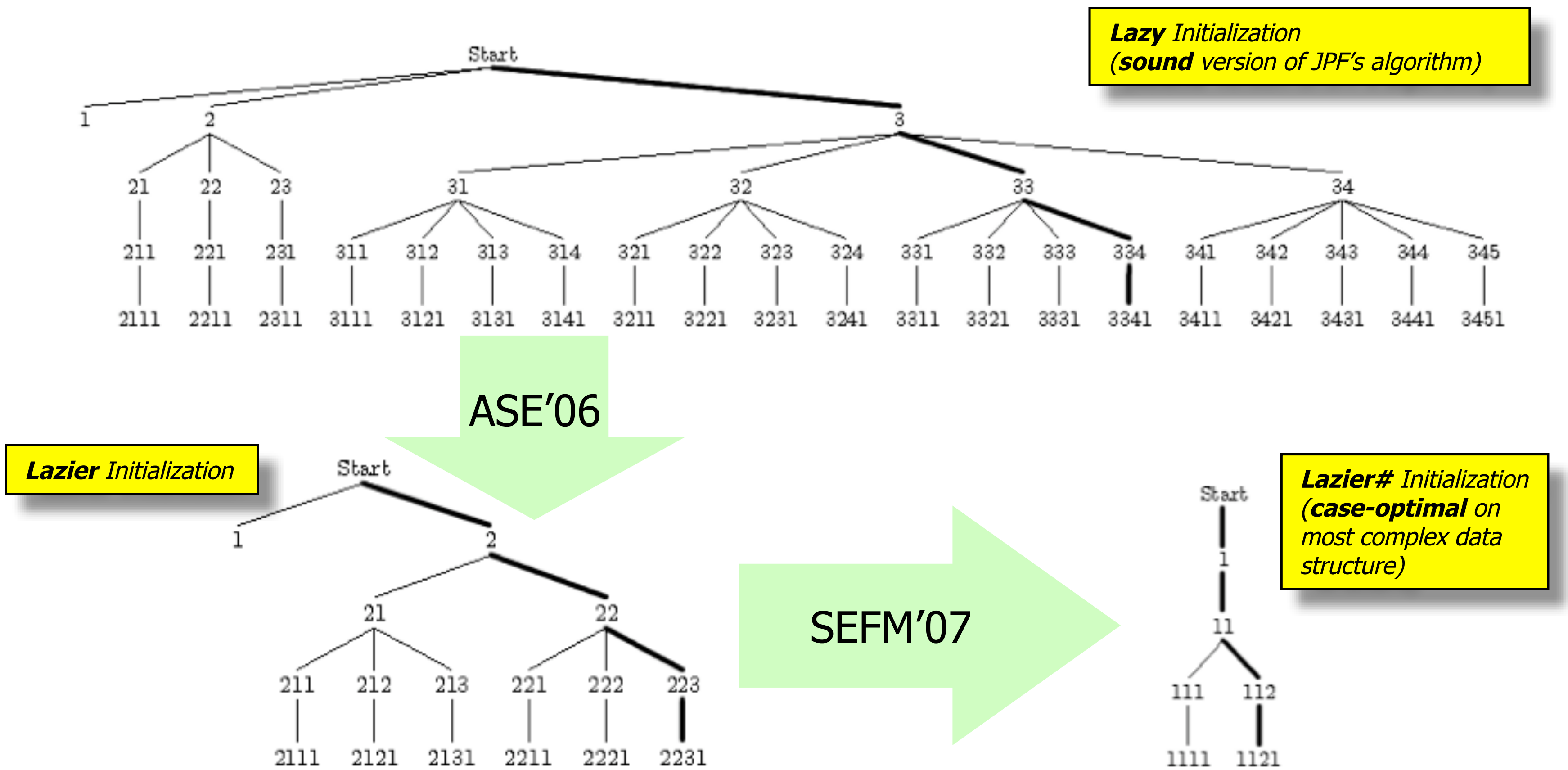


increasing heap coverage

Within the chosen bound, the analysis is complete (no false alarms) and sound (all errors that can be exposed with data size will be found)

Efficiency of Kiasan's Algorithms

```
public class Node<E> {
  //@ ensures data == \old(n.data) && n.data == \old(data);
  public void swap(@NonNull Node<E> n)
  { E e = data; data = n.data; n.data = e; }
  private Node<E> next; E data;
}
```



…state spaces of Kiasan's algorithms

Kiasan without Contracts

My developers are not going to be inclined to use this tool if they have to start out writing a bunch of complicated contracts.

How can you introduce Kiasan gradually into their workflow?

Example

```
void sort(int[] data) {
   boolean isSorted;
   int numberOfTimesLooped = 0;

  do {
    isSorted = true;

    for (int i = 1; i <= data.length - numberOfTimesLooped; i++) {
       if (data[i] < data[i - 1]) {
          int tempVariable = data[i];
          data[i] = data[i - 1];
          data[i - 1] = tempVariable;

          isSorted = false;
          }
     }

     numberOfTimesLooped++;
  } while (!isSorted);
}
```

Example

```
void sort(int[] data) {
   boolean isSorted;
   int numberOfTimesLooped = 0;

  do {
    isSorted = true;

    for (int i = 1; i <= data.length - numberOfTimesLooped; i++) {
       if (data[i] < data[i - 1]) {
          int tempVariable = data[i];
          data[i] = data[i - 1];
          data[i - 1] = tempVariable;

          isSorted = false;
          }
     }

     numberOfTimesLooped++;
  } while (!isSorted);
}
```

Kiasan detects possible null-dereference

Example

```
void sort(int[] data) {
   boolean isSorted;
   int numberOfTimesLooped = 0;

  do {
    isSorted = true;

    for (int i = 1; i <= data.length - numberOfTimesLooped; i++) {
       if (data[i] < data[i - 1]) {
          int tempVariable = data[i];
          data[i] = data[i - 1];
          data[i - 1] = tempVariable;

          isSorted = false;
          }
     }

     numberOfTimesLooped++;
  } while (!isSorted);
}
```

Kiasan detects array index out of bounds (i.e., i can be equal to data.length)

Reasoning about Heap Data

```
void foo3(Node n1, Node n2) {
  if (n1 != null && n2 != null) {
    n1.x = 2;
    n2.x = 3;
    assert (n1.x == 2 && n2.x == 3);
  }
}
```

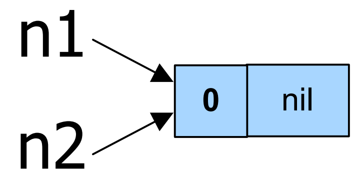
This assertion is ***obviously*** true!!! There is no way it can fail!

Tool says:
"Assertion can be violated."

Providing Diagnostic Information

```
void foo3(Node n1, Node n2) {
  if (n1 != null && n2 != null) {
    n1.x = 2;
    n2.x = 3;
    assert (n1.x == 2 && n2.x == 3);
  }
}
```

Error Case



I'm sure that the ***tool is wrong***! There is nothing that can cause the violation!!!

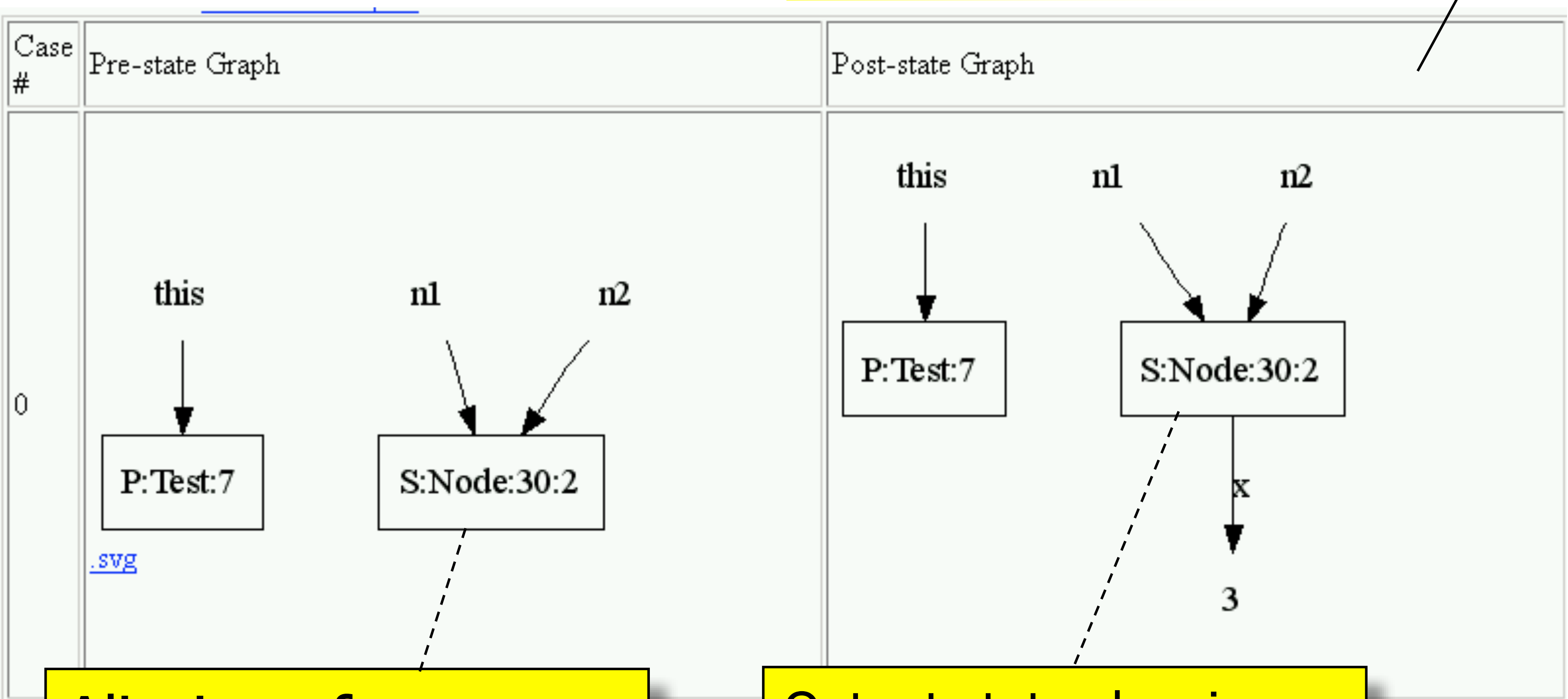
Not only does Kiasan tell you that there is an error, it gives you an example execution trace that leads to the error.

Providing Diagnostic Information

Pre-state Graph

Post-state Graph

auto-generated by Kiasan



Aliasing of n1, n2 in the inputs

Output state showing condition giving rise to assertion violation

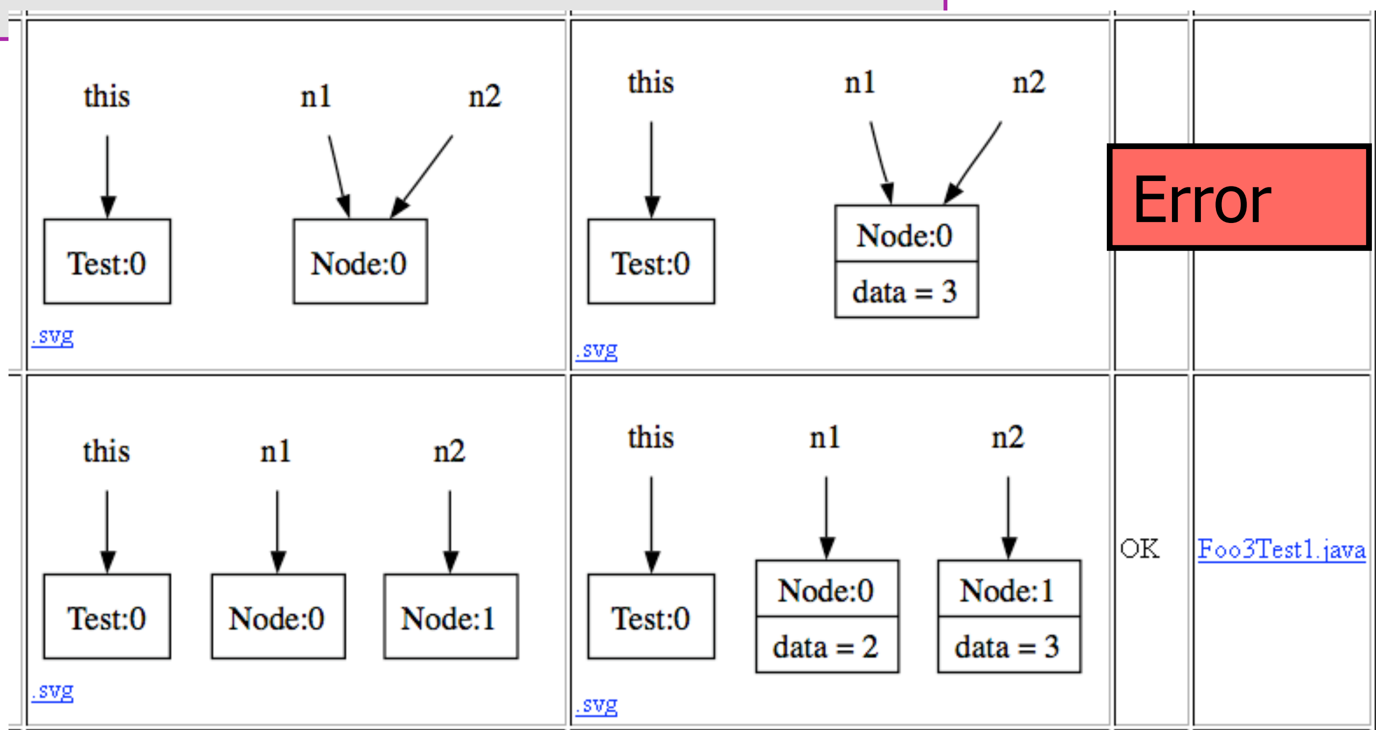
Kiasan provides pairs of states (pre,post) associated with a path leading to the error state

All Paths for Foo3 Example

```
void foo3(Node n1, Node n2) {
  if (n1 != null && n2 != null) {
    n1.x = 2;
    n2.x = 3;
    assert (n1.x == 2 && n2.x == 3);
  }
}
```

1.

2.



All Paths for Foo3 Example

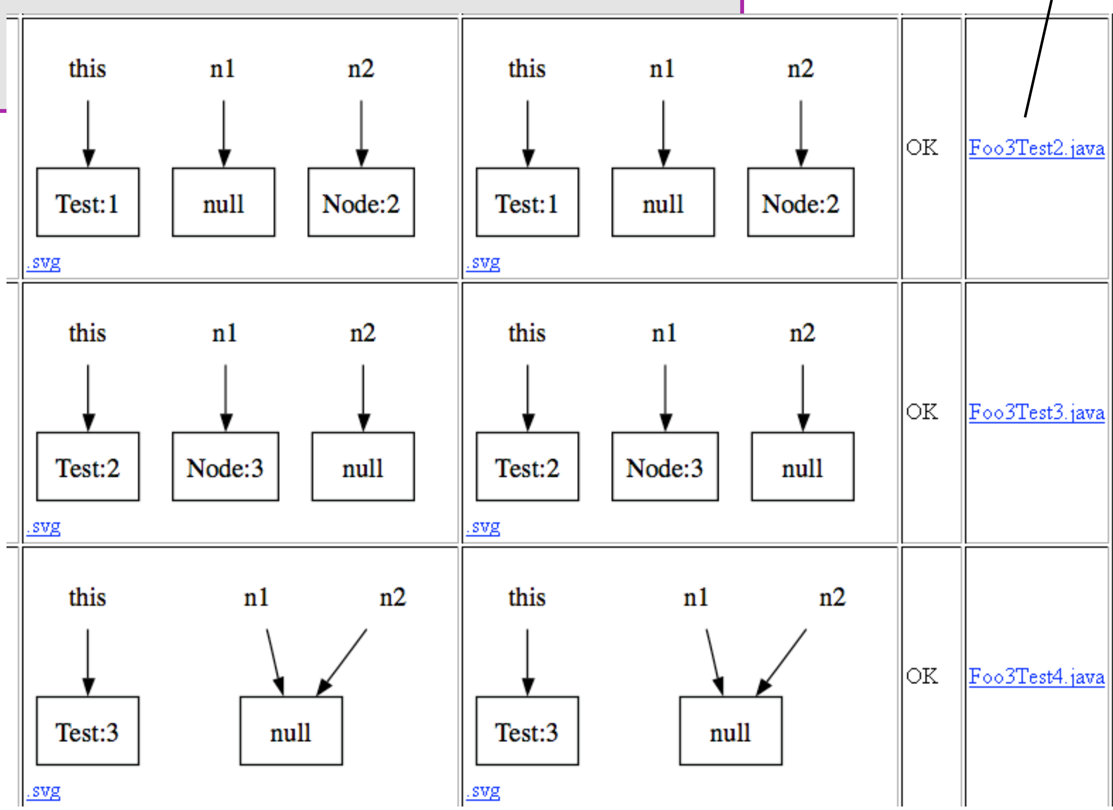
```
void foo3(Node n1, Node n2) {
  if (n1 != null && n2 != null) {
    n1.x = 2;
    n2.x = 3;
    assert (n1.x == 2 && n2.x == 3);
  }
}
```

JUnit test cases auto-generated by Kiasan for each case

3.

4.

5.



Kiasan with Contracts

"***Without specifications***,
the ***code*** is trivially ***correct*** !

I don't use anyone's service
unless they provide a contract"

Strong Property Checking

Kiasan has the technology to check strong properties in specification languages like JML

```
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) {
    …
  }
}
```

…merging of two sorted lists

Strong Property Checking

Kiasan has the technology to check strong properties in specification languages like JML

every linked-list is acyclic

this list is sorted and the other list is sorted based on the Comparator c

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

  /*@      :  isSorted(c) && other.isSorted(c);
           t: isSorted(c)
    @         && size() = \old(size()) + other.size()
    @         && (\forall E e;
    @              elements.contains(e);
    @              \old(this.contains(e))
                   || other.contains(e))

             NonNull LinkedList<E> other
             NonNull Comparator<E> c) {
```

this list is sorted

the size is equal to the other size plus this list's old size

all the elements are from the previous two lists

...merging of two sorted lists

Heavyweight & Lightweight

Actually, there are number of reasons why you might be willing to write specs like that, but for now I'll simply point out that one can also have useful ***lightweight specifications*.**

I can see how that would be helpful in high-assurance applications, but what about when I don't need that effort?

Samples of Design Intentions

Specifying common patterns

- Null-ness

```
class LinkedList { @NonNull LinkedNode head; }

class LinkedList { @MaybeNull LinkedNode head; }
```

- Null-ness of a container's element

```
class TreeNode {
 @NonNull @NonNullElements Set<TreeNode> children;
}
```

Samples of Design Intentions

Specifying common patterns

- Cyclic/Acyclic

```
class LinkedList { @Acyclic LinkedNode head; }
```

OR

```
@Acyclic(“head") class LinkedList { ... }
```

- Tree/Graph

```
@Tree("children") class TreeNode {
  Set<TreeNode> children;
}
```

Executable Specifications

If you don't like JML, you can write your own specification predicate directly as a pure (no non-local side effects) Java method…

```
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;

  return repOK(t.left,range.setUpper(t.element));
      && repOK(t.right,range.setLower(t.element));
}
```

…invariant for binary search tree

…not elegant, but it was effective for the author

Dealing with Heap Data

II. Specify that invariant should be checked on input & output

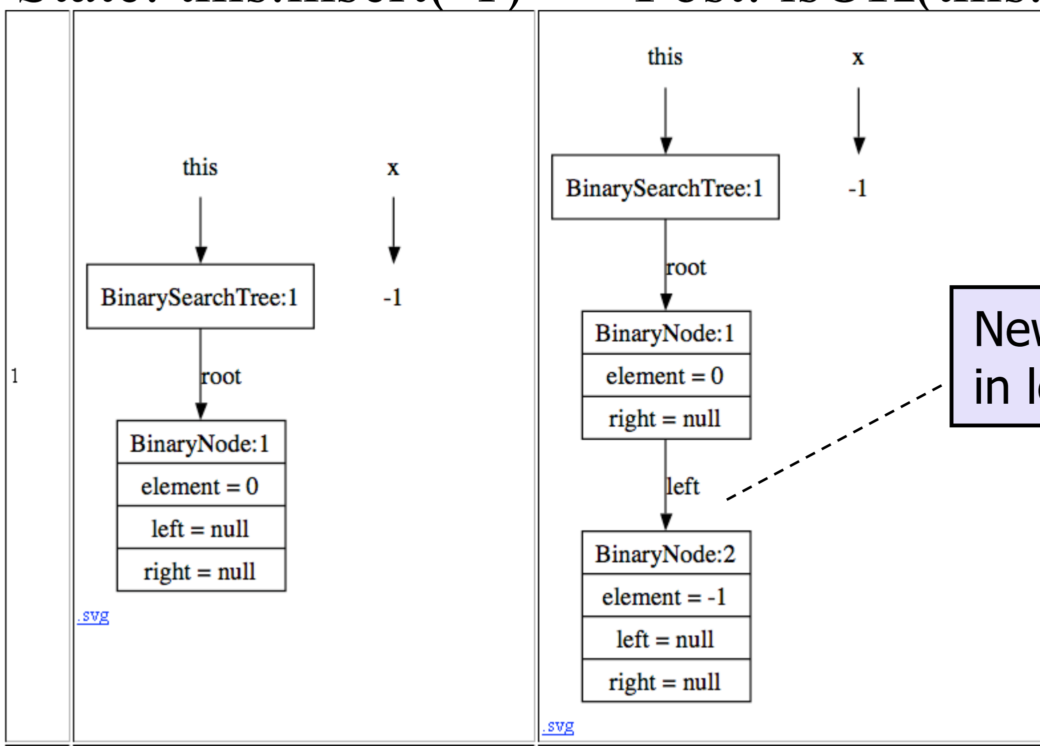
```
@Assertion(@Case(
      pre = "repOK(root)",
      post = "repOK(root)"))
public void insert( int x ) {root = myins( x, root ); }

@Helper
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;
}
```

Dealing with Heap Data: Results

III. Invoke Kiasan to check method and/or generate tests

21 cases for k=2

Pre-State: this.insert(-1) Post: isOK(this.root)



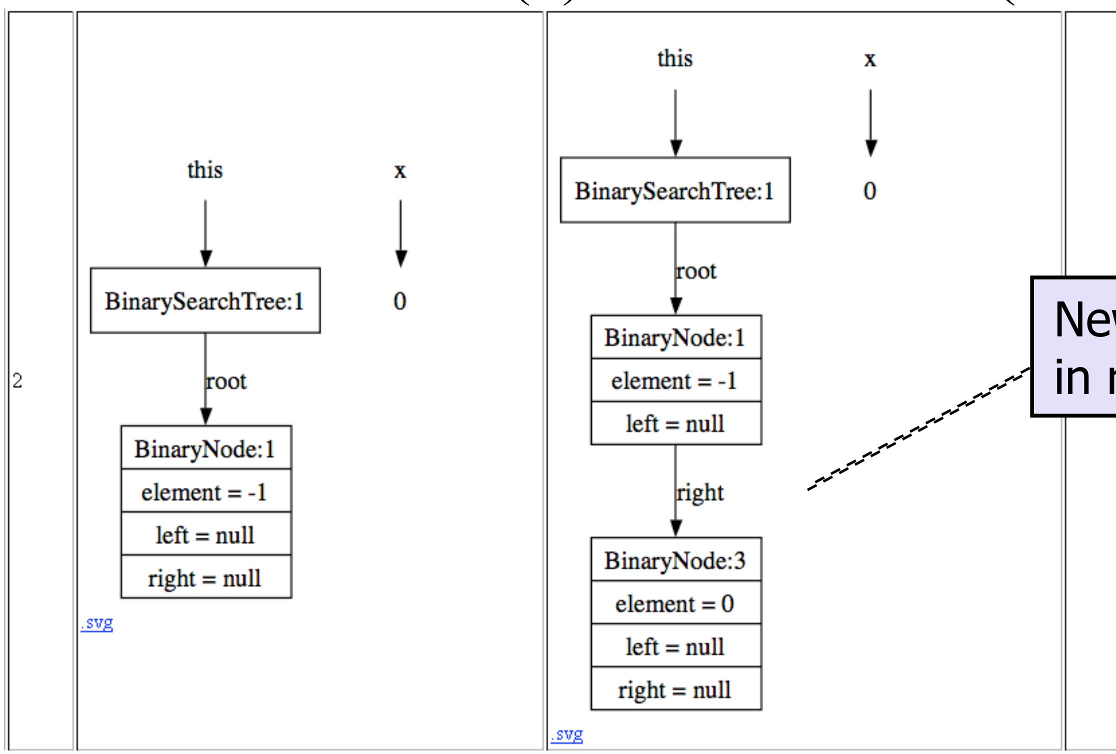
New element goes in left child

Tool verifies that pre/post conditions are satisfied and gives pre/post-state pairs for each path through the method

Dealing with Heap Data: Results

III. Invoke Kiasan to check method and/or generate tests

Pre-State: this.insert(0)   Post: isOK(this.root)



New element goes in right child

Think about the effort if one has to generate these test scenarios manually!

Scalability & Performance

Short answer:

Significantly better than related (publicly available) techniques that use symbolic execution technology

Example Code Bases

Object-based examples

- AVL Tree
- Binary Search Tree
- Red-black Tree
- Double Linked List
- Linked List
- GC
- Stack (List Impl)

…plus additional examples of data structures containing scalar data

Array-based examples

- Binary Heap
- Insertion Sort
- Shell Sort
- Stack (Array Impl)
- Array Partition
- Disjoint Set (original/fast)
- Vector
- Triangle classification
- Absolute value

…the largest collection of examples considered for OO symbolic execution

Research Questions

- Can this method obtain high values of (branch) coverage?
- What value of k-bound is typically needed to obtain 100% branch coverage?
- What sizes of test suites are generated for different values of k?
- What's typical time required on runs where 100% branch coverage is reached?

Data

2.4 GHz Opteron Linux
w/ 512 MB Java heap

Data for the most complicated examples…

Class	Method	k	Test Cases Generated	Branch Coverage	Bytecode Coverage	Total (-dot)	CVC Lite	$\Rightarrow_A^{-1}$ & POOC	JUnit Gen.	GraphViz dot (+)
BinarySearchT	insert	1	4	6/6=100%	63/63=100%	1.7s	0.1s	0.3s	0.2s	1.2s
		2	21	6/6=100%	63/63=100%	5.8s	1.3s	0.9s	0.5s	4.3s
		3	236	6/6=100%	63/63=100%	1.8m	1.2m	15.6s	1.1s	31.6s
	remove	1	4	8/16=50%	68/113=60%	1.2s	0.0s	0.2s	0.0s	1.3s
		2	21	14/16=87%	104/113=92%	7.4s	1.7s	1.2s	0.3s	3.9s
		3	236	15/16=93%	111/113=98%	1.7m	1.1m	13.8s	1.1s	34.1s
AvlTree	find	1	4	10/10=100%	60/60=100%	1.8s	0.0s	0.4s	0.2s	1.3s
		2	21	10/10=100%	60/60=100%	8.9s	3.6s	1.3s	0.1s	3.9s
		3	190	10/10=100%	60/60=100%	2.8m	1.9m	25.4s	1.7s	25.5s
	findMax	1	2	7/8=87%	38/42=90%	1.1s	0.0s	0.3s	0.0s	0.3s
		2	5	8/8=100%	41/42=97%	2.7s	0.3s	0.3s	0.0s	0.9s
		3	20	8/8=100%	41/42=97%	21.3s	10.3s	2.2s	0.7s	4.1s
	insert	1	4	12/18=66%	119/270=44%	2.0s	0.3s	0.3s	0.0s	0.5s
		2	21	18/18=100%	270/270=100%	8.9s	2.1s	1.4s	0.3s	3.4s
		3	190	18/18=100%	270/270=100%	3.3m	2.2m	31.7s	2.2s	23.7s
java.util.Vector	add	1	8	6/6=100%	73/73=100%	1.5s	0.4s	0.3s	0.0s	1.7s
		2	8	6/6=100%	73/73=100%	1.2s	0.2s	0.1s	0.1s	1.5s
		3	8	6/6=100%	73/73=100%	1.3s	0.2s	0.4s	0.2s	1.6s
	indexOf	1	6	10/10=100%	41/41=100%	1.0s	0.1s	0.2s	0.1s	1.0s
		2	7	10/10=100%	41/41=100%	1.2s	0.2s	0.2s	0.2s	1.2s
		3	7	10/10=100%	41/41=100%	0.8s	0.1s	0.3s	0.0s	1.7s
	removeElementAt	1	3	7/8=87%	70/71=98%	0.5s	0.1s	0.1s	0.2s	0.2s
		2	5	8/8=100%	71/71=100%	1.1s	0.3s	0.1s	0.2s	0.5s
		3	9	8/8=100%	71/71=100%	1.5s	0.3s	0.3s	0.1s	1.4s
java.util.TreeMap	lastKey	1	2	5/6=83%	32/35=91%	0.5s	0.0s	0.1s	0.1s	0.4s
		2	6	6/6=100%	35/35=100%	3.2s	0.6s	0.4s	0.4s	1.1s
		3	31	6/6=100%	35/35=100%					
	put	1	10	12/52=23%	133/500=27%					
		2	78	40/52=76%	470/500=94%					
		3	1023	42/52=80%	478/500=95%					
	remove	1	5	13/86=15%	105/684=15%					
		2	43	45/86=52%	347/684=51%					
		3	579	70/86=81%	640/684=93%					
	GC.mark	1	306	12/12=100%	65/65=100%					

k=2 is enough to give 100% branch coverage for most examples, with time required under 9sec (but usually only 2-3secs)

Experiment Data

Class	Method	k	States Lazy	States Lazier	States Lazier#	Cases Lazy	Cases Lazier	Cases Lazier#
AvlTree	find	1	3271	2420	1864	5	4	4
		2	48244	23807	18800	29	21	21
		3	10944306	459718	351798	275	190	190
	insert	1	4719	3841	3053	5	4	4
		2	56832	31905	25702	29	21	21
		3	11036507	542929	422049	275	190	190
BinarySearchTree	insert	1	6097	5521	1621	13	12	4
		2	91691	63931	12551	112	94	21
		3	3349343	1855571	234595	2161	1668	236
	remove	1	4146	3693	[illegible]	[illegible]	[illegible]	[illegible]
		2	74896	49422	[illegible]	[illegible]	[illegible]	[illegible]
		3	3031511	1599087	[illegible]	[illegible]	[illegible]	[illegible]
	find	1	4890	4301	1162	13	12	4
		2	89819	57292	10443	126	98	21
		3	3822839	1808683	212296	2873	1788	236
StackList	push	1	758	758	374	4	4	2
		2	1466	1390	687	6	6	3
		3	2450	2260	1119	8	8	4
	pop	1	196	196	189	2	2	2
		2	425	387	377	3	3	3
		3	770	675	662	4	4	4
java.util.TreeMap	get	1	4309	2009	1199	8	6	4
		2	85601	27489	17440	62	40	28
		3	20707094	774545	470913	782	482	331
	remove	1	2247	1721	1110	7	5	4
		2	74892	37832	17081	73	43	28
		3	17631620	1166311	472985	1075	579	331
	lastKey	1	1219	664	657	2	2	2
		2	15680	7658	[illegible]	[illegible]	[illegible]	[illegible]
		3	3524450	205430	[illegible]	[illegible]	[illegible]	[illegible]
java.util.Vector	add	1	986	818	[illegible]	[illegible]	[illegible]	[illegible]
		2	2932	1514	[illegible]	[illegible]	[illegible]	[illegible]
		3	10990	2906	[illegible]	[illegible]	[illegible]	[illegible]
	indexOf	1	644	588	438	7	6	6
		2	1195	1135	486	17	16	7
		3	2686	2339	486	44	38	7
	removeElementAt	1	202	200	197	3	3	3
		2	382	320	257	6	5	4
		3	999	566	318	16	9	5

BinarySearchTree

NASA/JPF		Kiasan
3.5s	2.1s	1.1s
22.7s	17.1s	5.0s
50.1m	16.4m	1.5m
2.4s	2.5s	1.3s
22.4s	14.5s	5.1s
43.0m	13.8m	1.3m
2.1s	2.9s	0.8s
23.1s	16.3s	4.9s
55.5m	15.7m	1.4m
4.2s	2.1s	1.6s
16.0s	10.3s	7.7s
7.0h	3.1m	2.0m
1.4s	1.4s	1.4s
16.0s	12.2s	5.8s
5.1h	7.6m	1.9m
0.7s	0.4s	0.6s
7.7s	2.5s	3.6s
27.0m	27.9s	30.3s

util.TreeMap
(RedBlack Tree)

Table 1. Experiment Data (excerpts); s – seconds; m – minutes; h – hours

jCute Comparison (excerpts)

Binary Search (remove)

- Time: 2.5 mins to achieve 15/16 branch coverage compared to 1.3 mins for Kiasan
- *This is the typical comparative behavior for examples that include non-trivial for non-complex heap manipulation.*

AVL (remove)

- Time: only able to achieve 14/18 branch coverage (time out after 1 hour) while Kiasan obtains 18/18 coverage in 8.9 secs.

Red Black Tree (remove)

- Time: only able to achieve 16/73 (feasible) branch coverage (time out after 1 hour) while Kiasan obtains 70/73 coverage in 1.9 mins.

…and recall that Kiasan is giving stronger heap configuration coverage in examples above.

Summary

- Automated contract checking for strong heap properties
  - provides a significant increment to ESC-Java-like checking
  - concise summaries of behavior from which other artifacts (including tests) can be derived
- Several nice methodological approaches
  - controllable costs/coverage
  - gradual transition from light to heavyweight specs
- Integrated unit test case generation
  - leverages contracts to prune tests
  - limitations
    - test cases phrased via reflection, not class APIs
    - engineering issues to be addressed before yielding a deployable tool
- Extends to a variety of different policies regarding heap sharing/partitioning, etc.
- Scalability is reasonable and getting better all the time
- See 72-page tech report for correctness proofs (including minimality results) and all experimental data

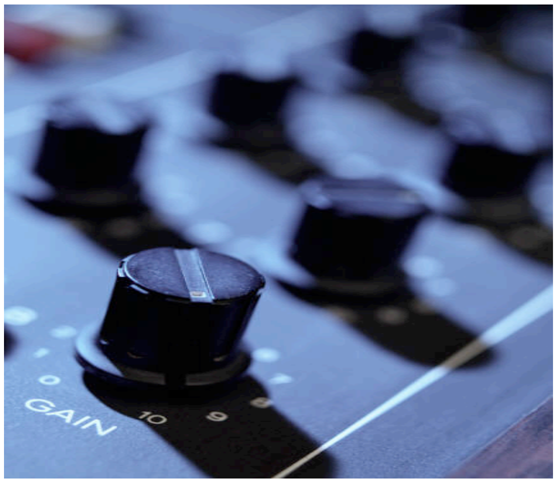
For More Information…

SAnToS Laboratory,
Kansas State University
http://www.cis.ksu.edu/santos

Bogor/Kiasan Project

http://bogor.projects.cis.ksu.edu

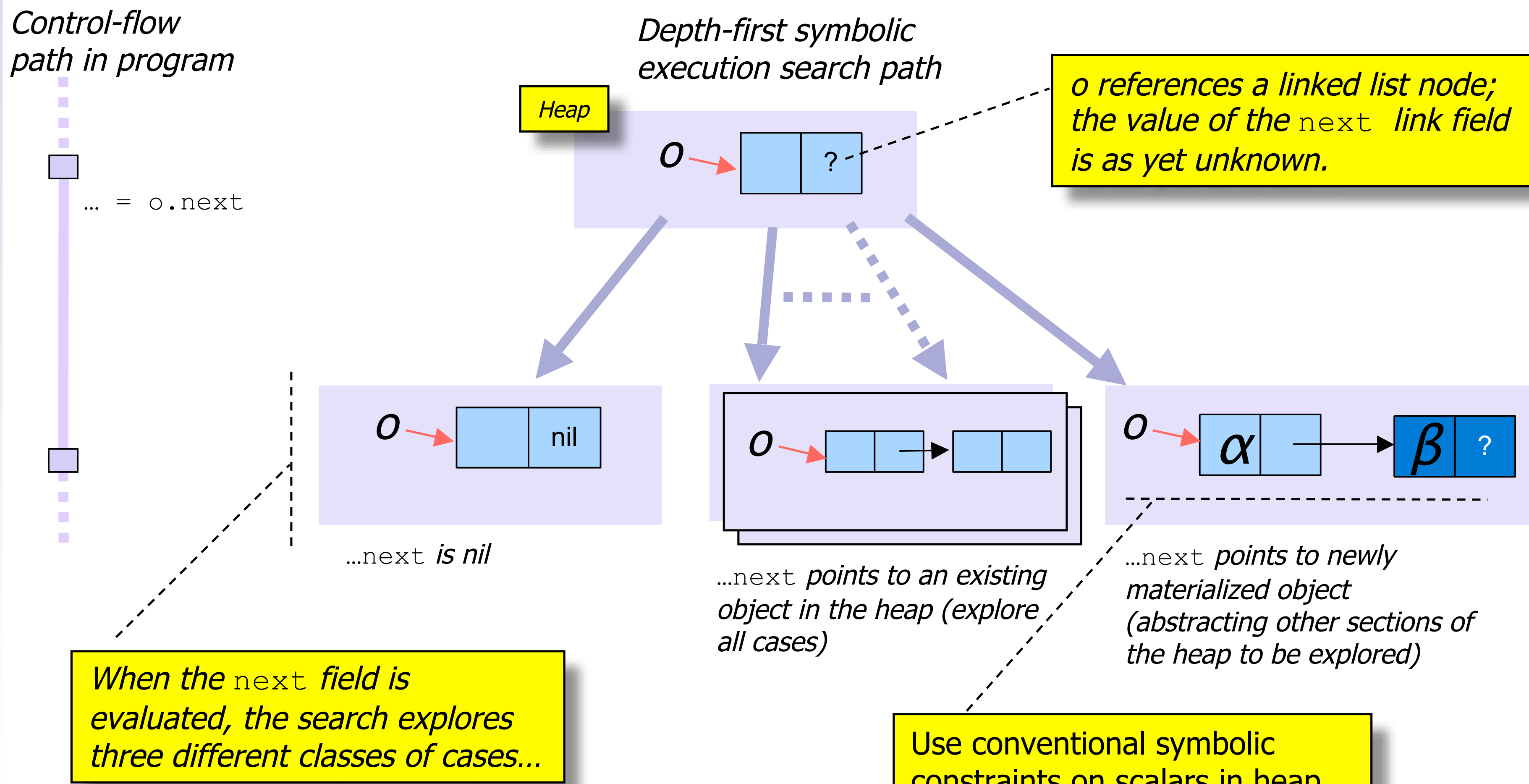
Kiasan Methodology (Vision)



- Checking in IDE
  - start with small bounds
  - incrementally check
  - scenario and test case generation for violations
- More exhaustive checking
  - higher bounds with overnight/parallel checking
  - Kiasan tells you if coverage criteria has been met
- Code understanding
  - select any block of code,
    Kiasan generates flow scenarios giving path coverage
- Test case generation for regression testing
  - automatically generate tests from code
  - incrementally add tests as changes are made
- Specifications are leveraged for static checking, code understanding/inspection, test case generation, and doc.

Handling Heap Data

Basic concepts…

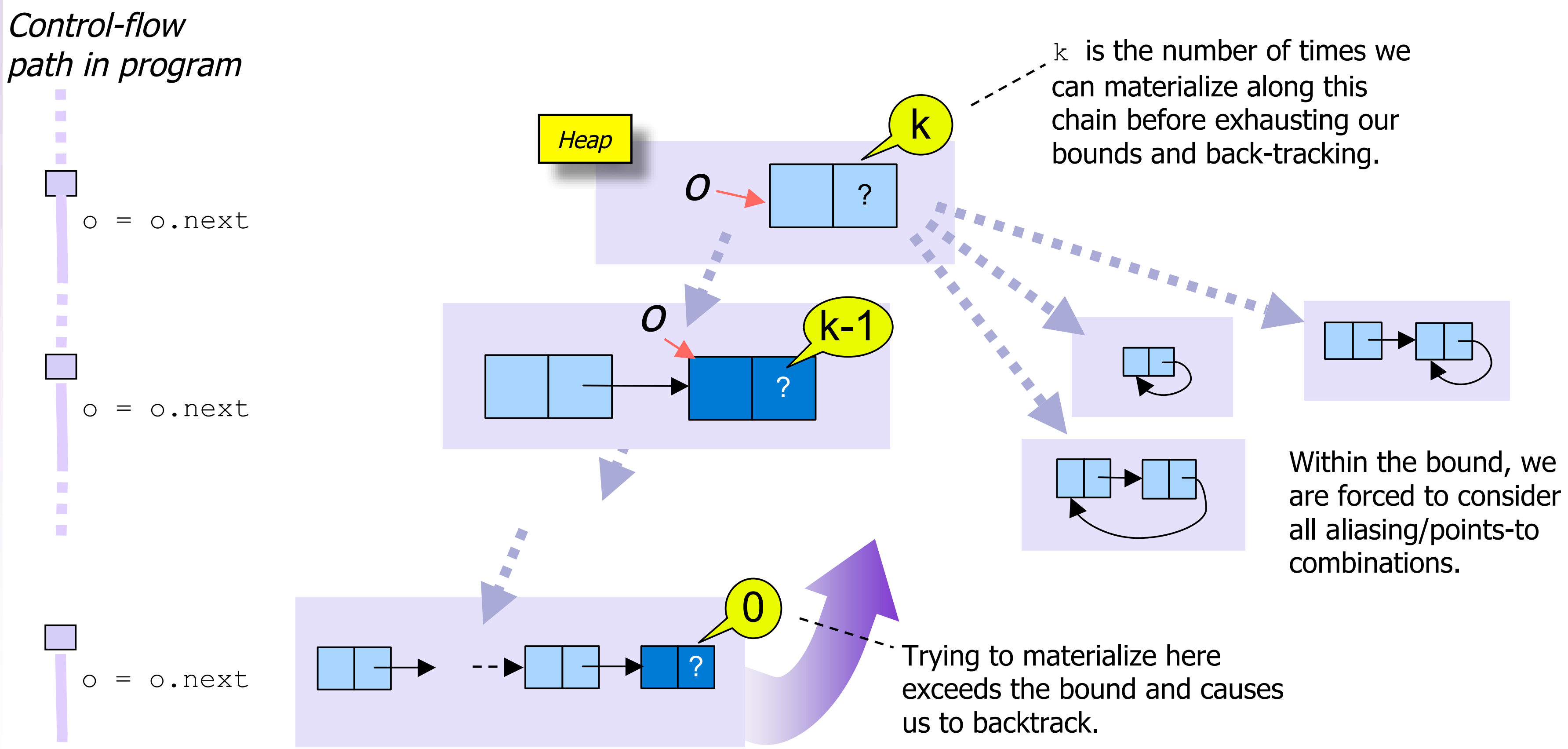


…based on Khurshid, Pasareanu, and Visser (TACAS 2003)
-- NASA JPF symbolic execution algorithm

$\{\alpha > \beta\}$

Bounding for Heap Coverage

Kiasan uses a unique notion of bounding that focuses on achieving coverage of heap configurations

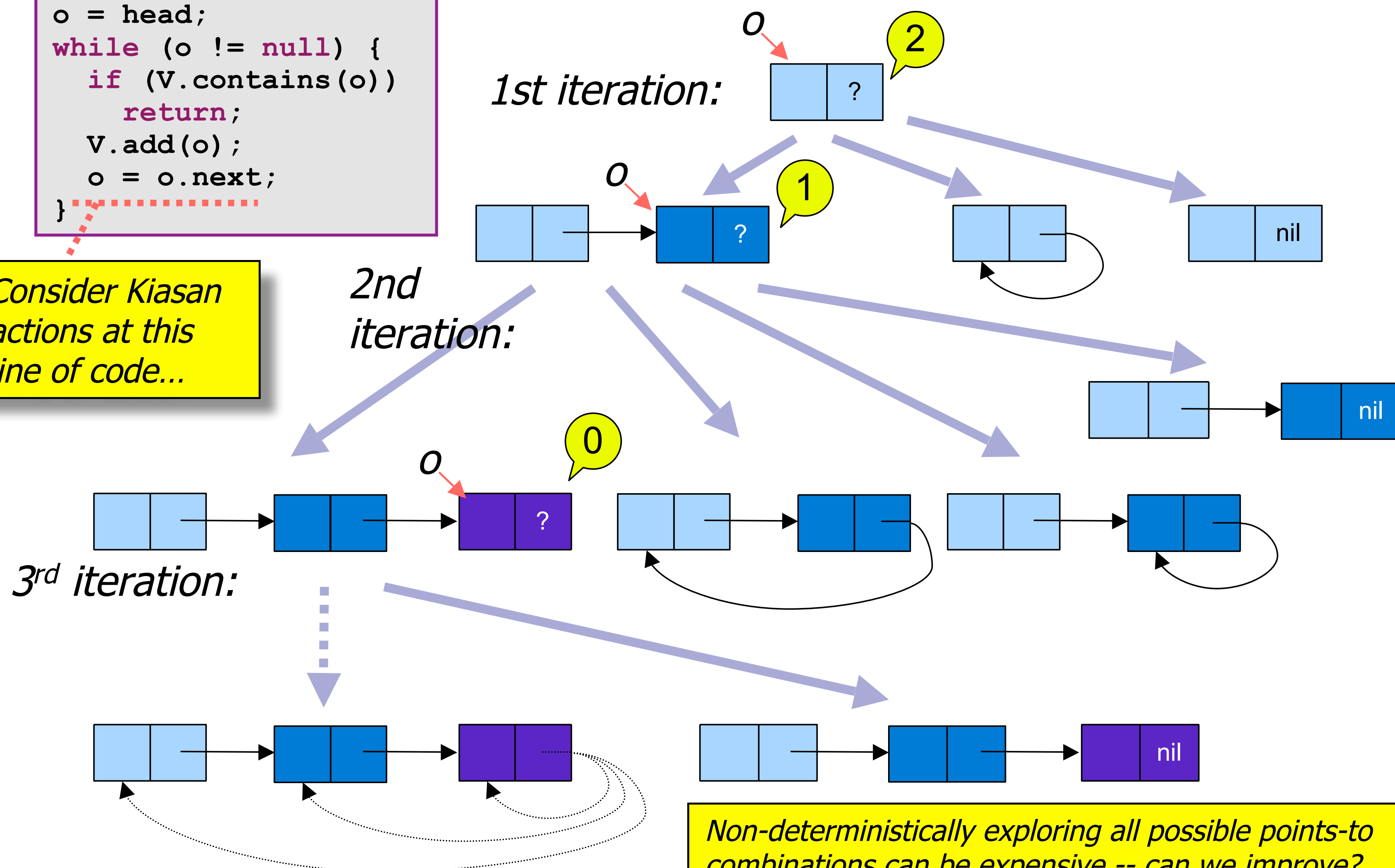


***k*-bound:** *backtrack in search when length of reference chain from original root would be greater than k*

Handling Objects using Lazy Initialization (k = 2): LinkedList

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

Consider Kiasan actions at this line of code…



Non-deterministically exploring all possible points-to combinations can be expensive -- can we improve?