Bogor/Kiasan:
A Contract-based Verification and Test-Case Generation Framework

SAnToS Laboratory, Kansas State University, USA

William (Xianghua) Deng

Jooyong Lee
(BRICS, DK)

Robby

John Hatcliff

Support

US Army Research Office (ARO)
US National Science Foundation (NSF)
US Air Force Office of Scientific Research (AFOSR)
Lockheed Martin ATL (Cherry Hill, NJ)
Rockwell Collins Advanced Technology Center
IBM Eclipse

Implementation Level Specification

Implementation level specification & checking plays an important role in developing high-assurance systems

Simple pre-condition…

```
function FindSought
 (A: Table; Sought: Integer) return Index;
--# pre for some M in Index => ( A(M) = Sought );
--# return Z => (( A(Z) = Sought) and
--#     (for all M in Index range 1 .. (Z - 1) =>
--#          (A(M) /= Sought)));
```

Post-condition constraining return value to inputs…

"SPARK Examiner with Run-time Checker…", p. 22

Implementation Level Specification

Implementation level specification & checking plays an important role in developing high-assurance systems

Spark information flow annotations…

```
procedure Operate;
--# global out KeyStore.RotorValue,Encrypted;
--# in out KeyStore.SymmetricKey;
--# in Clear;
--# derives
--#   KeyStore.SymmetricKey, KeyStore.RotorValue
--# from
--#   KeyStore.SymmetricKey
--# &
--#   Encrypted
--# from
--#   Clear, KeyStore.SymmetricKey
--# ;
```

Information flows from `Clear`, `KeyStore.SymmetricKey` *to* `Encrypted`

"Enforcing Security and Safety Models with an Information Flow Analysis Tool"

Our Interests

- Implementation level specification and checking for languages with rich object-oriented features
  - focusing on properties of heap-allocated data
  - specification and checking of both
    - safety properties
    - information flow/partitioning (not covered in this talk)
  - deep integration with other quality assurance methods such as testing

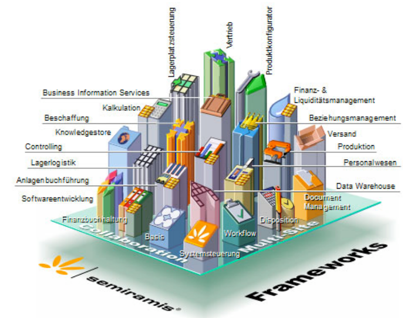
Due to the complexities of languages like Java, we don't aim to provide all the soundness guarantees of Spark/Praxis tools, but we do provide a rigorously justified formal foundation and soundness on a bounded portion of a program's state space.

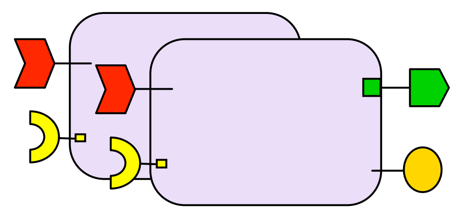
…let's now look at some of the issues that we aim to address

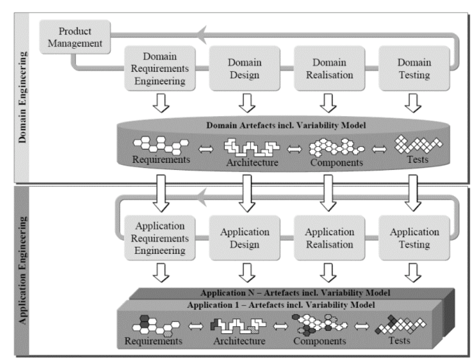
Trends In Software Development

Building Software from Reusable Units

- Frameworks
  - collection of units targeted to a particular application domain
  - Apache Struts, JavaServerFaces, CLSA
- Component Middleware
  - dictates a structure notion of reuseable component
  - provides extensive infrastructure and services
  - CCM, EJB, nesC, Bonobo
- Software Product Lines
  - Drive down development time and costs through systematic reuse of a managed set of assets across families of similar platforms

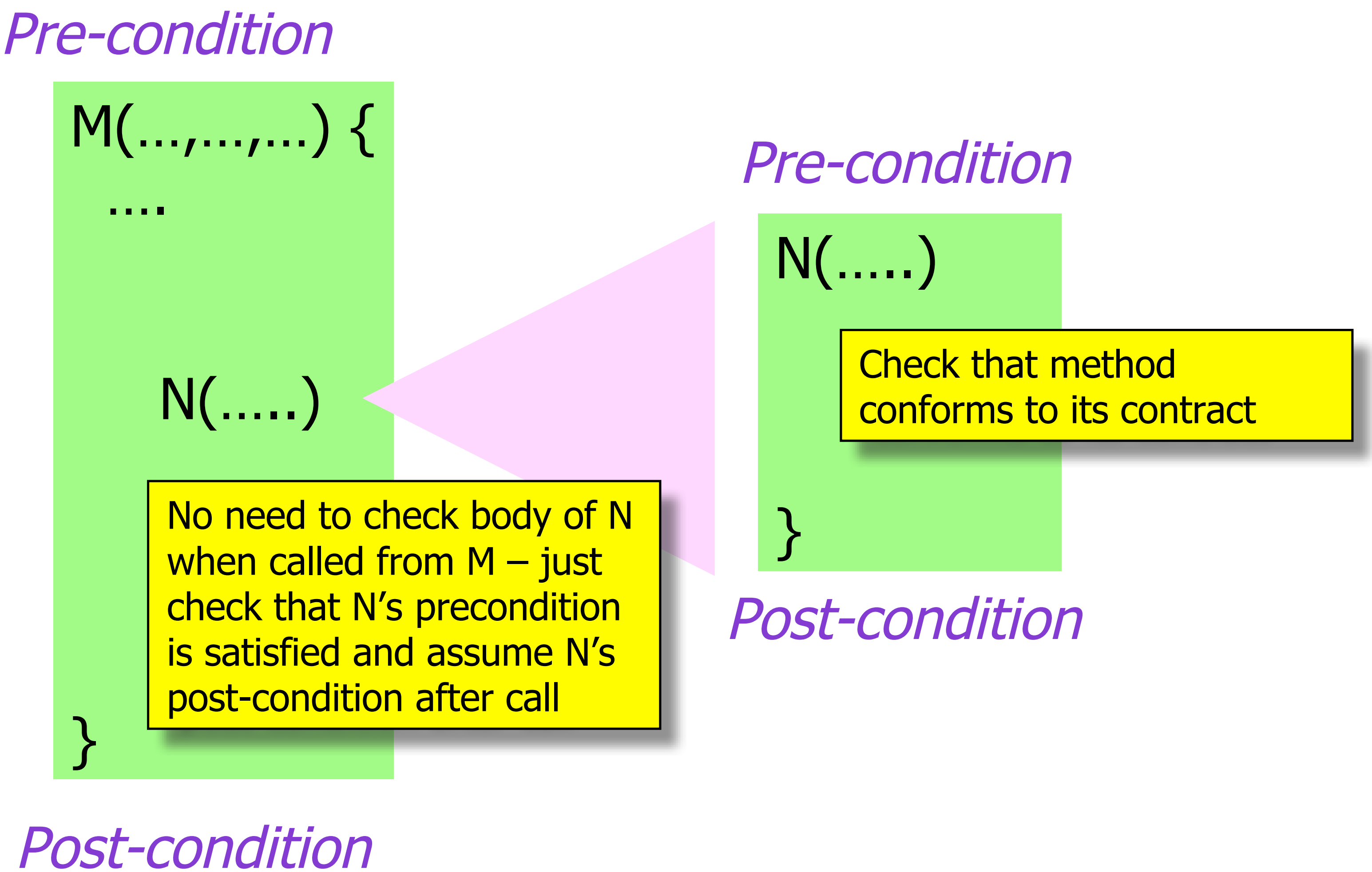






Benefits of Contracts

Contracts enable compositional checking



Compositional Checking

- Compositional checking is the key to scalability
  - Allows each method to be checked in isolation
  - If a method is changed, only need to recheck that one method (not the entire code base)
  - Enables checking to be carried out in parallel

Software Contracts

Lightweight Contracts

Simple pre-condition…

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

```
/*@ 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;
  }
}
```

linked list from java.util.concurrent

Software Contracts

Strong Properties of Heap-allocated Data

…moving beyond ESC/Java

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

```
/*@ 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;
 }
```

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*

linked list from java.util.concurrent

A Skeptic's Questions

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

Progress

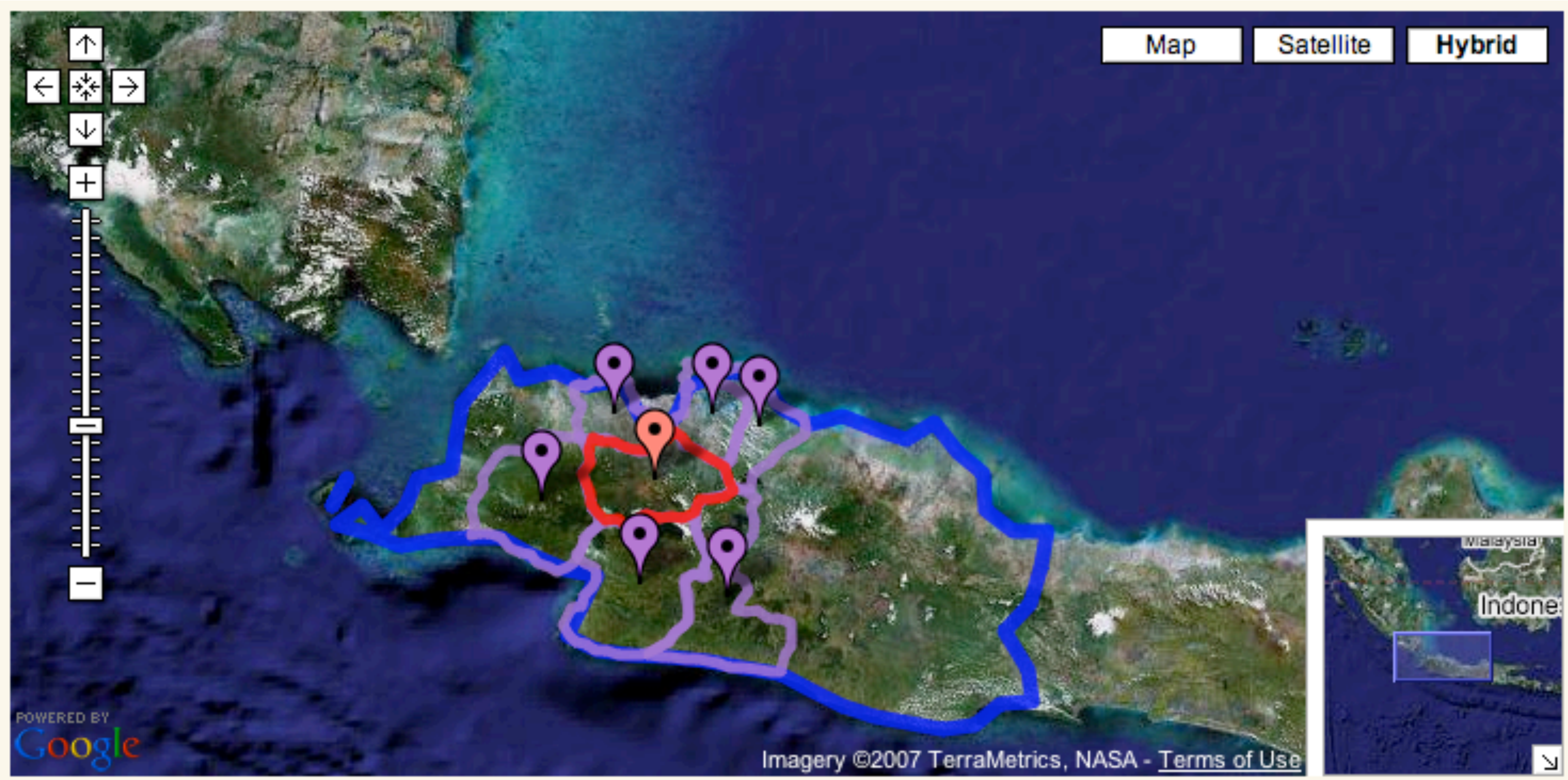
Tools like ESC/Java have made good progress toward answering the skeptic's questions…

- Practical contract checking technology for Java
- Supports automated checking of lightweight method contracts
- Effective for statically eliminating many common run-time errors such as null-pointer exceptions, array bounds checking

But a number of limitations remain…

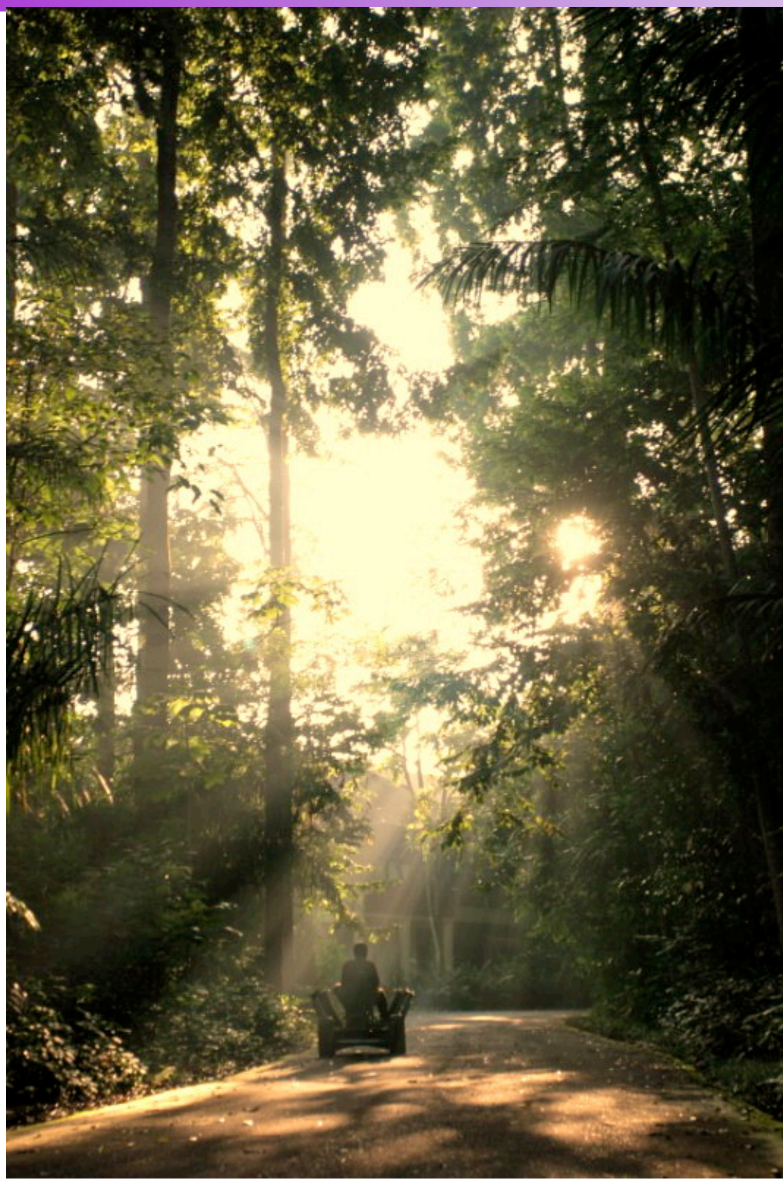
- Don't handle heap-allocated data very well
- Error messages are hard to decipher
- No direct connection to other quality assurance techniques

Bogor, West Java, Indonesia



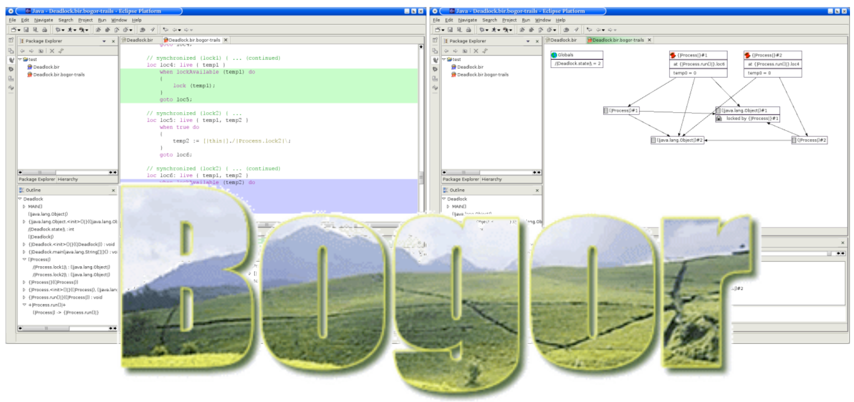
Bogor, West Java, Indonesia

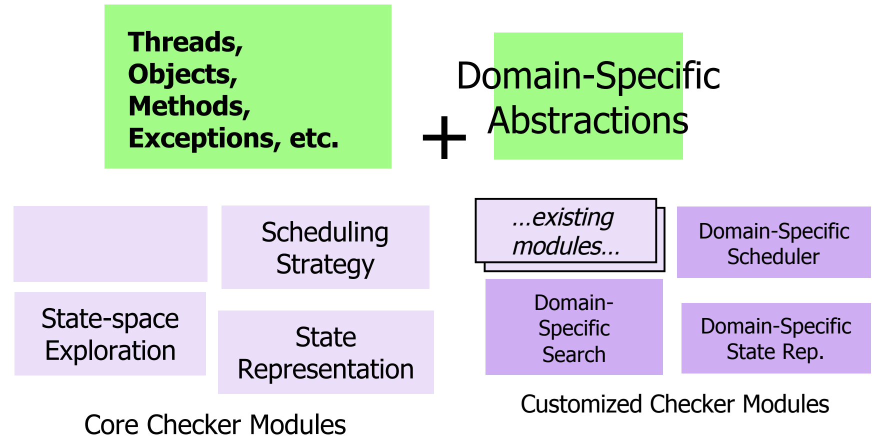
Bogor, West Java, Indonesia



Bogor, West Java, Indonesia

Bogor Model Checking Framework





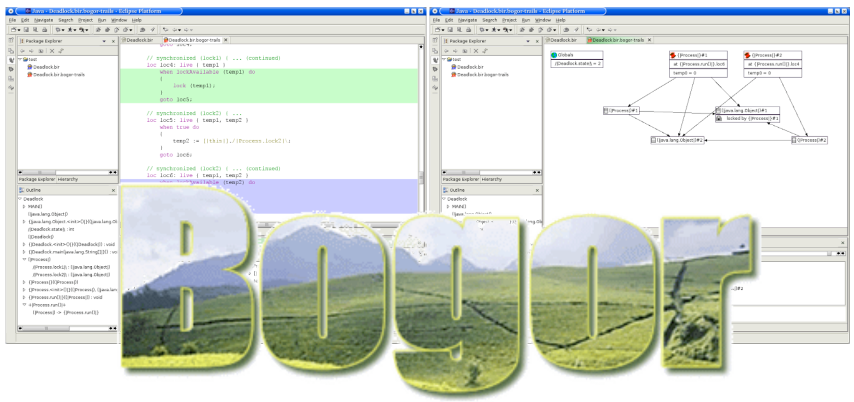
Direct support for…

- *unbounded* dynamic creation of threads and objects
- automatic memory management (garbage collection)
- virtual methods, …
- …, exceptions, etc.
- *supports virtually all of Java*

Extensible Framework…

- new commands and expressions can be added to the modeling language to create *domain specific modeling languages*
- modular architecture allows core algorithms to easily be plugged and unplugged
- … becoming a generic *state-space exploration framework*

Bogor Model Checking Framework



…thank you for Eclipse support!

University Research…

- a number of external research projects
  - MPI, BPEL (ICSE 2007), UML State Charts, .NET
- over external 1300 downloads

Educational Material…

- wide collection of pedagogical material…
  - lecture slides
  - streaming video lectures
  - projects, exams, labs, quizzes
- used by at least five universities at both the undergraduate & graduate level during the past year

Industrial Use…

- Funded in 2006-2007 by Lockheed Martin Software Technology Initiative
- Bogor is the core of LM’s Thimble framework for verification / visualization of threading properties of .NET systems
- Primary testbed is LM’s Horizon satellite mission control system software product line

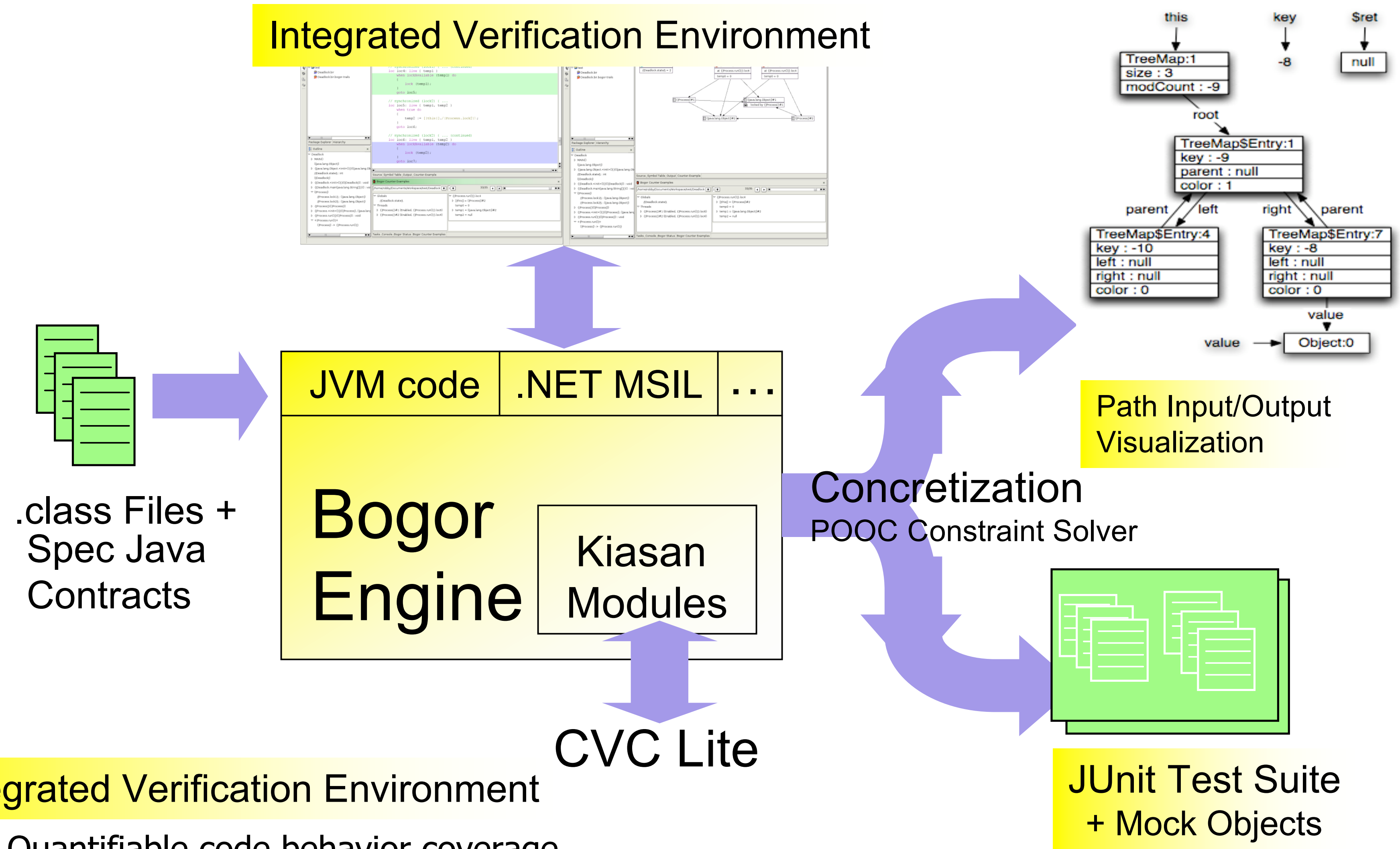
Bogor / Kiasan

Kiasan – A Bogor Extension for Symbolic Execution

"kiasan"
=
"symbolic"

- Combines symbolic execution with…
  - model checking
  - theorem proving
  - constraint solving
- Formal operational semantics
  - Relative soundness and completeness proofs
- Quantifiable code behavior coverage
- Adjustable analysis cost/coverage
- Static (compositional/non-compositional) checking of
  - unspecified code
  - light-weight specifications
  - strong statements about heap properties
  - e.g., exceeding capabilities of ESC/Java
- Provides helpful analysis feedback
  - counter examples, visualization using object graphs
- Automates some of developers' tasks
  - JUnit test case generation

Bogor / Kiasan Architecture



Integrated Verification Environment

- Quantifiable code behavior coverage
- Adjustable analysis cost/coverage

Outline

Bogor / Kiasan

Foundations

- Basic concepts
- Dealing with the heap
- Correctness results and distinguishing features

Tool Capabilities

- Lightweight property checking
- Input/Output Visualizations
- Strong contract checking
- Test case generation for open systems

Symbolic Execution [King:ACM76]

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

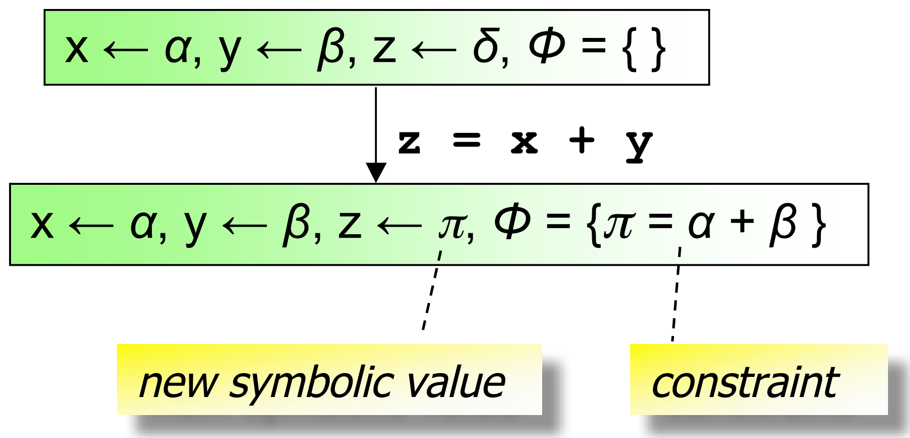
symbolic values

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++;
   }
}
```



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`

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

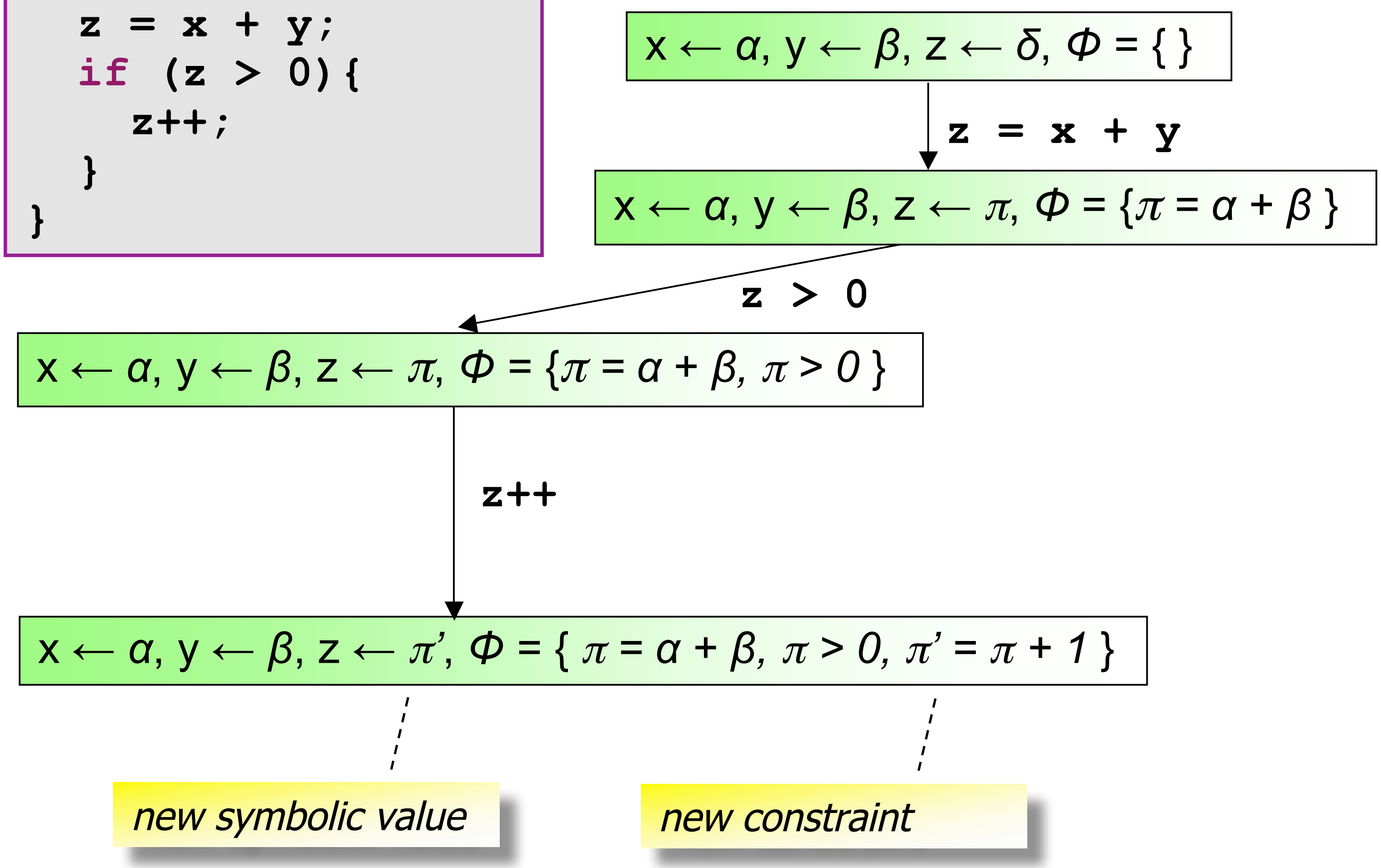
`z > 0`

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

new constraint for conditional

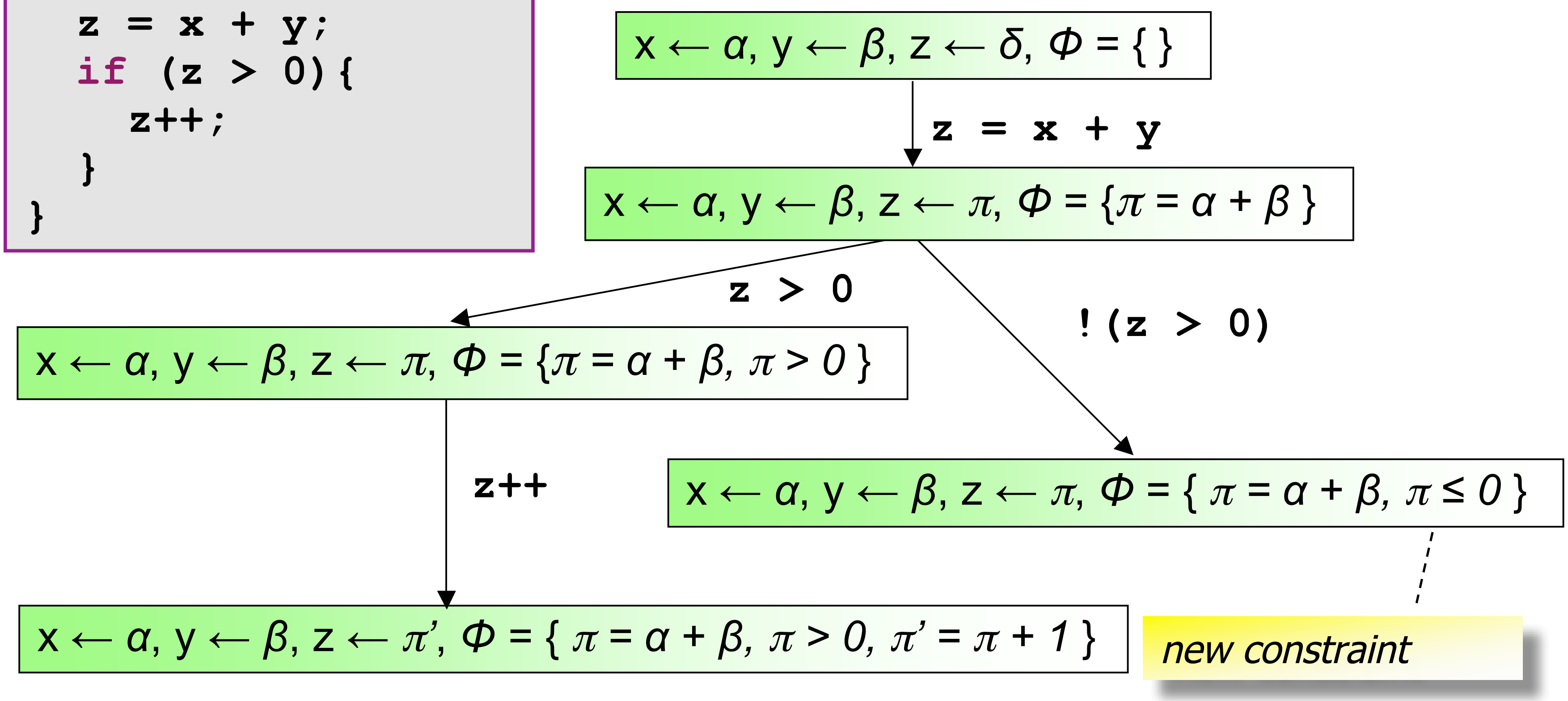
Symbolic Execution [King:ACM76]

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



Symbolic Execution [King:ACM76]

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



…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 > 0)`

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

`z++`

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

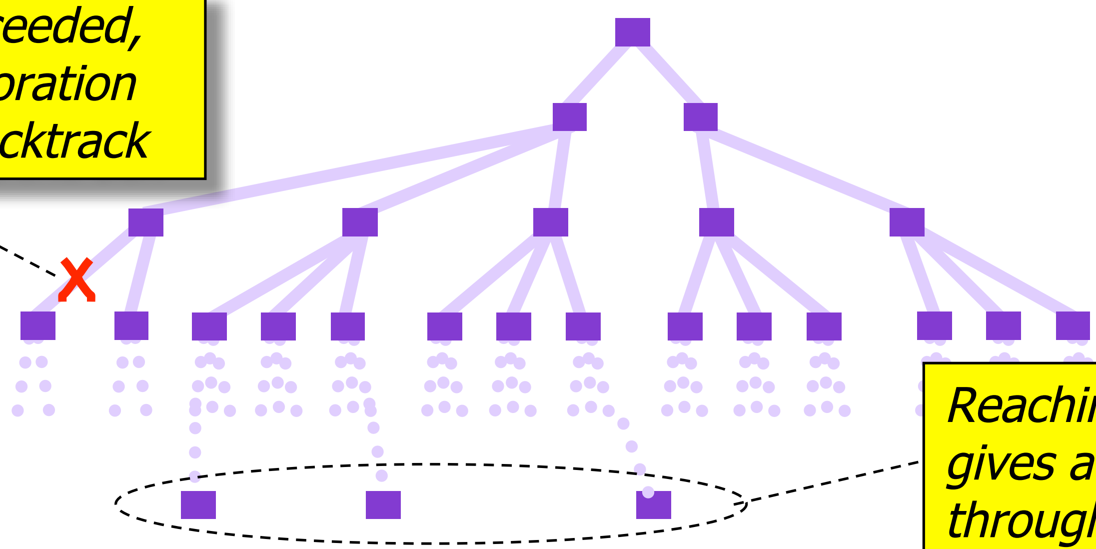
The path condition *characterizes the set of program states that flow to this point in the path.*

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

Issue: Handling Loops

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

If bound is exceeded, then stop exploration of path and backtrack



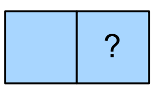
Reaching a method exit gives a complete path through the method

Representing Heap Data [Khurshid-al:TACAS03]

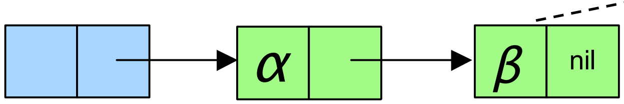
How should dynamically allocated heap data be represented in symbolic execution?

…model checker maintains a representation of the heap *…take advantage of that*

Combined Concrete & Symbolic Representation



…abstract heap location; represents an arbitrary heap structure



…use conventional symbolic constraints on scalars in heap

$\alpha > \beta$

…lazily expand symbolic representation as program interacts with the heap

Representing Heap Data — Kiasan's *k*-bounding

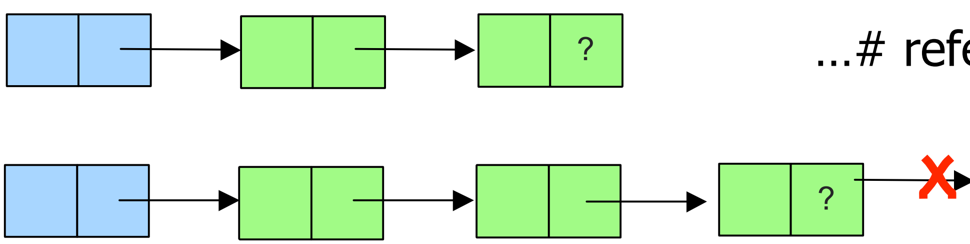
How should dynamically allocated heap data be represented in symbolic execution?

…model checker maintains a representation of the heap *…take advantage of that*

Note: Kiasan uses an improved algorithm -- lazier# initialization

Bound search by bounding length of reference chains

length limit k = 3



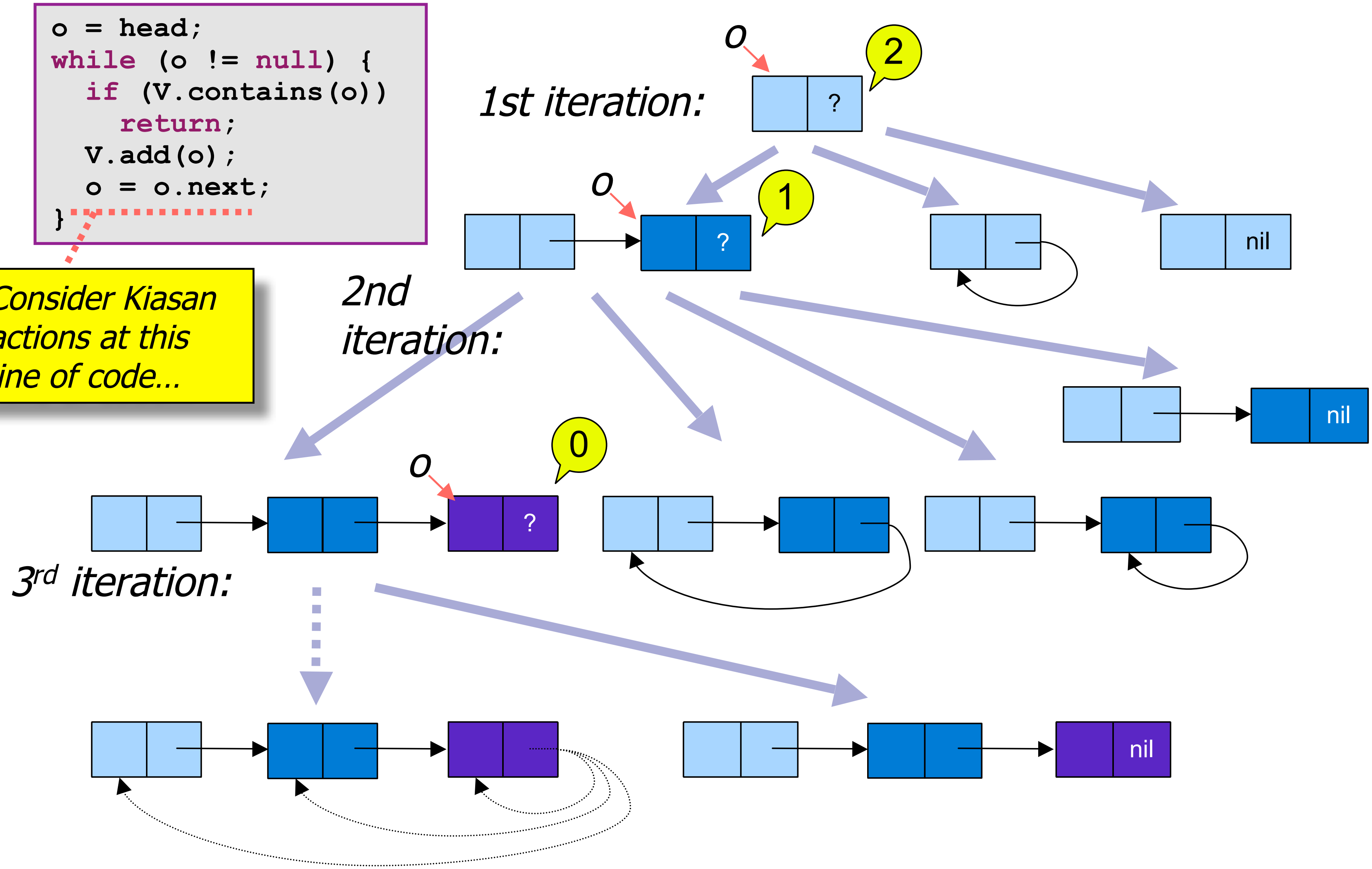
…# references is 2; keep expanding

…backtrack when execution generates a chain longer 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…



Correctness Results and Distinguishing Features

- Formal semantics of Kiasan’s static analysis
  - proofs: relatively sound and complete
    - found an unsoundness (bug) in NASA’s JPF symbolic execution implementation
- Significantly more efficient algorithms
  - orders of magnitude reduction in analysis cost
- A method to quantify the behavior coverage analyzed by Kiasan
- Fully supports Design-by-Contract paradigm
  - the most powerful compositional static analyzer for strong heap-oriented properties
- Formalized generation of analysis feedback
  - test cases, input/output object graphs

Experiment Data

Kiasan's algorithm (Lazier#) dramatically improves over competitors.

Class	Method	k	States Lazy	States Lazier	States Lazier#	Cases Lazy	Cases Lazier	Cases Lazier#	Total Time Lazy	Total Time Lazier	Total Time Lazier#	Theorem Prover Time Lazy	Theorem Prover Time Lazier	Theorem Prover Time Lazier#
AvlTree	find	1	3271	2420	1864	5	4	4	1.2s	1.5s	0.8s	0.4s	0.5s	0.1s
		2	48244	23807	18800	29	21	21	8.9s	7.2s	6.9s	2.8s	3.9s	2.5s
		3	10944306	459718	351798	275	190	190	1.7h	2.4m	3.7m	1.1h	1.9m	3.2m
	insert	1	4719	3841	3053	5	4	4	2.1s	2.6s	1.5s	0.3s	1.6s	0.7s
		2	56832	31905	25702	29	21	21	10.3s	7.0s	7.5s	4.5s	3.1s	4.0s
		3	11036507	542929	422049	275	190	190	2.1h	2.8m	3.9m	1.4h	2.2m	3.5m
BinarySearchTree	insert	1	6097	5521	1621	13	12	4	3.5s	2.1s	1.1s	0.9s	0.5s	0.1s
		2	91691	63931	12551	112	94	21	22.7s	17.1s	5.0s	9.5s	7.9s	1.7s
		3	3349343	1855571	234595	2161	1668	236	50.1m	16.4m	1.5m	39.1m	12.6m	1.1m
	remove	1	4146	3693	1001	13	12	4	2.4s	2.5s	1.3s	1.2s	0.7s	0.5s
		2	74896	49422	9254	112	94	21	22.4s	14.5s	5.1s	12.8s	5.6s	1.9s
		3	3031511	1599087	197738	2161	1668	236	43.0m	13.8m	1.3m	35.1m	10.9m	1.0m
	find	1	4890	4301	1162	13	12	4	2.1s	2.9s	0.8s	0.3s	1.0s	0.2s
		2	89819	57292	10443	126	98	21	23.1s	16.3s	4.9s	10.9s	8.4s	1.8s
		3	3822839	1808683	212296	2873	1788	236	55.5m	15.7m	1.4m	42.5m	12.4m	1.1m
StackList	push	1	758	758	374	4	4	2	0.7s	0.7s	0.6s	0.0s	0.0s	0.0s
		2	1466	1390	687	6	6	3	0.9s	0.8s	0.5s	0.0s	0.1s	0.1s
		3	2450	2260	1119	8	8	4	2.1s	1.7s	0.7s	0.4s	0.1s	0.0s
	pop	1	196	196	189	2	2	2	0.2s	0.2s	0.2s	0.0s	0.0s	0.1s
		2	425	387	377	3	3	3	0.4s	0.3s	0.5s	0.0s	0.0s	0.1s
		3	770	675	662	4	4	4	0.4s	0.6s	0.5s	0.0s	0.2s	0.0s
java.util.TreeMap	get	1	4309	2009	1199	8	6	4	4.2s	2.1s	1.6s	3.0s	1.2s	1.0s
		2	85601	27489	17440	62	40	28	16.0s	10.3s	7.7s	8.5s	4.3s	4.4s
		3	20707094	774545	470913	782	482	331	7.0h	3.1m	2.0m	5.0h	2.3m	1.4m
	remove	1	2247	1721	1110	7	5	4	1.4s	1.4s	1.4s	0.1s	0.4s	0.9s
		2	74892	37832	17081	73	43	28	16.0s	12.2s	5.8s	7.9s	6.2s	2.3s
		3	17631620	1166311	472985	1075	579	331	5.1h	7.6m	1.9m	3.7h	6.4m	1.4m
	lastKey	1	1219	664	657	2	2	2	0.7s	0.4s	0.6s	0.1s	0.0s	0.3s
		2	15680	7658	7614	6	6	6	7.7s	2.5s	3.6s	3.5s	0.5s	1.2s
		3	3524450	205430	204738	31	31	31	27.0m	27.9s	30.3s	21.5m	17.3s	19.3s
java.util.Vector	add	1	986	818	354	6	6	3	2.0s	1.4s	0.7s	1.0s	0.8s	0.5s
		2	2932	1514	472	20	14	5	6.5s	2.8s	1.2s	4.2s	1.4s	0.7s
		3	10990	2906	590	74	30	7	21.0s	5.6s	0.9s	15.8s	3.3s	0.4s
	indexOf	1	644	588	438	7	6	6	0.9s	1.6s	0.9s	0.2s	0.3s	0.4s
		2	1195	1135	486	17	16	7	2.1s	2.0s	1.1s	0.5s	1.0s	0.7s
		3	2686	2339	486	44	38	7	4.5s	4.1s	0.5s	2.5s	1.7s	0.1s
	removeElementAt	1	202	200	197	3	3	3	0.8s	0.3s	0.3s	0.6s	0.1s	0.1s
		2	382	320	257	6	5	4	1.0s	0.7s	0.6s	0.2s	0.1s	0.4s
		3	999	566	318	16	9	5	2.0s	0.9s	0.6s	0.7s	0.5s	0.2s

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

Experiment Data

Class	Method	k	States Lazy	States Lazier	States Lazier#	Cases Lazy	Cases Lazier	Cases Lazier#	Total Time Lazy	Total Time Lazier	Total Time Lazier#	Theorem Prover Time Lazy	Theorem Prover Time Lazier	Theorem Prover Time Lazier#
AvlTree	find	1	3271	2420	1864	5	4	4	1.2s	1.5s	0.8s	0.4s	[illegible]	
		2	48244	23807	18800	29	21	21	8.9s	7.2s	6.9s	2.8s	3.9s	2.5s
		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	1001	13	12	4						
		2	74896	49422	9254	112	94	21						
		3	3031511	1599087	197738	2161	1668	236						
	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	7614	6	6	6						
		3	3524450	205430	204738	31	31	31						
java.util.Vector	add	1	986	818	354	6	6	3						
		2	2932	1514	472	20	14	5						
		3	10990	2906	590	74	30	7						
	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						

		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

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

Outline

Bogor / Kiasan

Foundations

- Basic concepts
- Dealing with the heap
- Correctness results and distinguishing features

Tool Capabilities

- Lightweight property checking
- Input/Output Visualizations
- Strong contract checking
- Test case generation for open systems

Kiasan without Contracts

What's all this "contract" rubbish – they're just a big waste of time. **The code is the only thing that matters anyway**.

So what can Kiasan do for me?

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

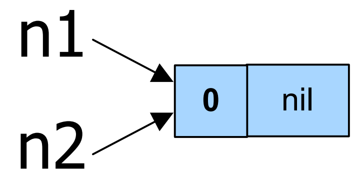
```
void foo3(Node n1, Node n2) {
  if (n1 != null && n2 != null) {
    n1.x = 2;
    n2.x = 3;
    assert (n1.x == 2 && n2.x == 3);
  }
}
```

Yes, it can! Aliasing issues often cause faults in even very simple code.

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!!!

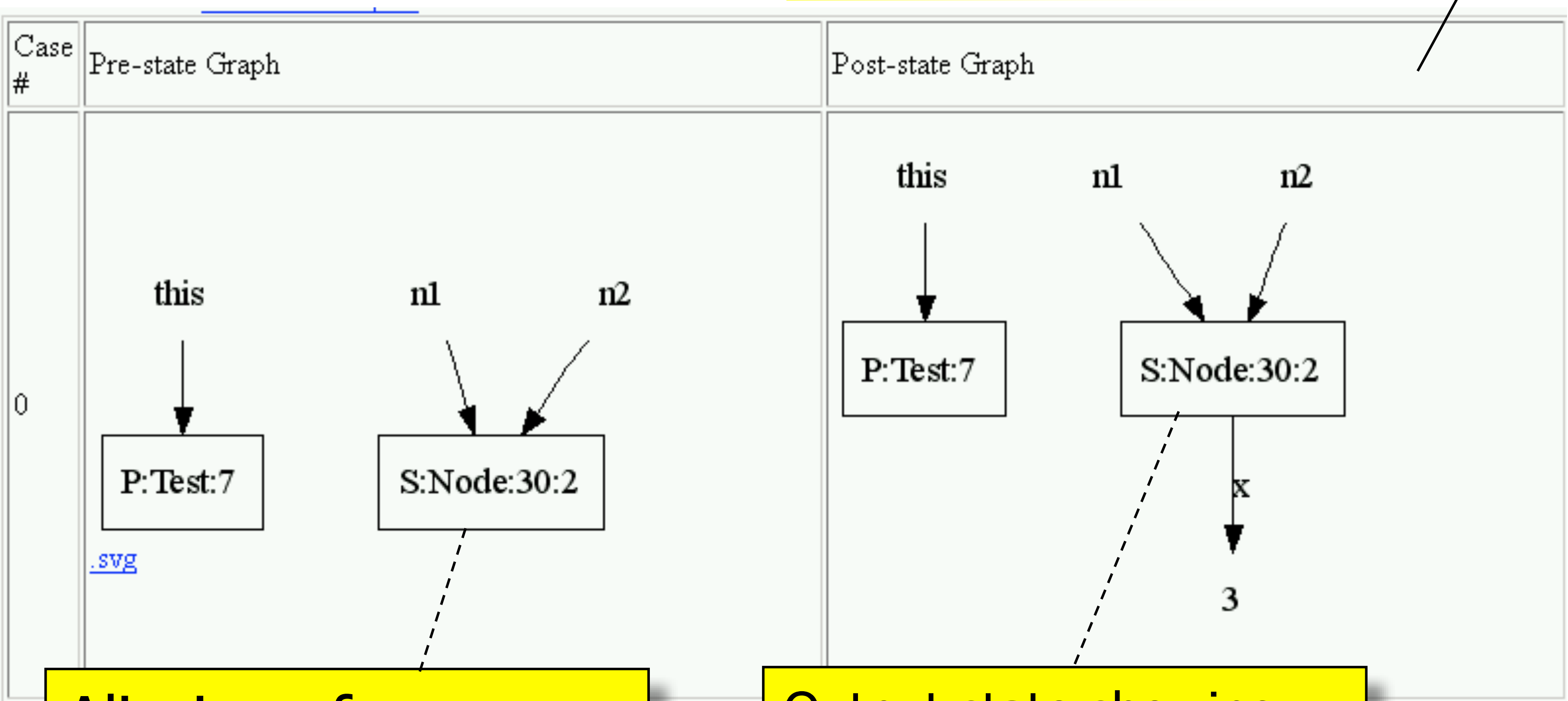
Not only does Kiasan tell you that there is an error, it gives you an example execution traces that leads to the error.

Providing Diagnostic Information

Pre-state Graph

Post-state Graph

auto-generated by Kiasan



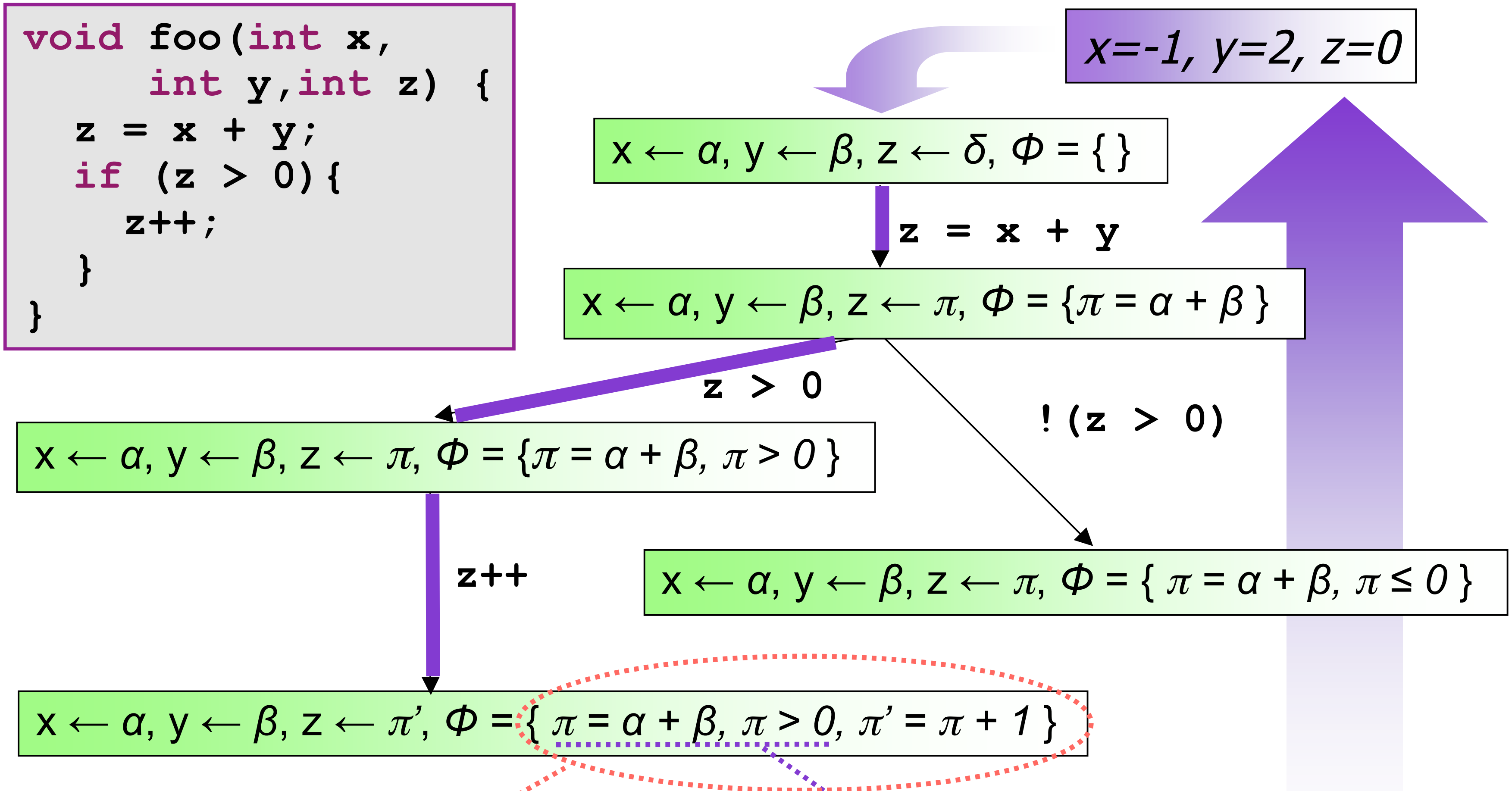
Aliasing of n1, n2 in the inputs

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

Solving Constraints

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



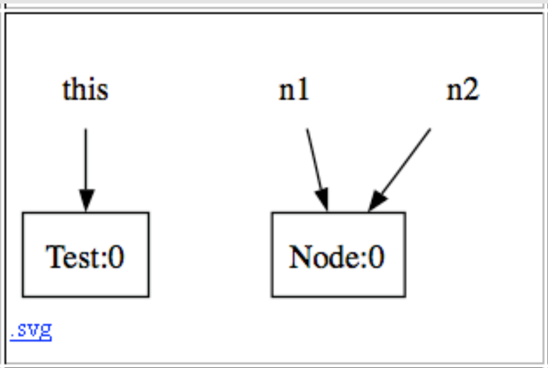
The path condition *characterizes the set of program states that flow to this point in the path.*

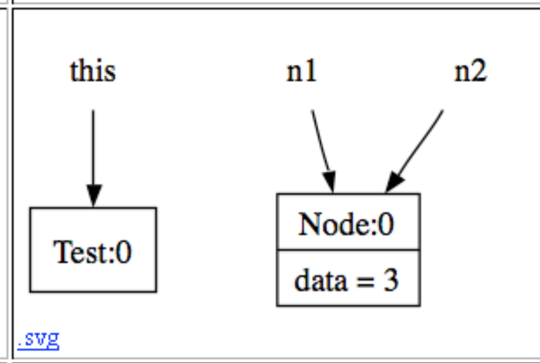
Solving constraints on input variables yields input values (a test case) that drives execution down the current path.

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);
  }
}
```

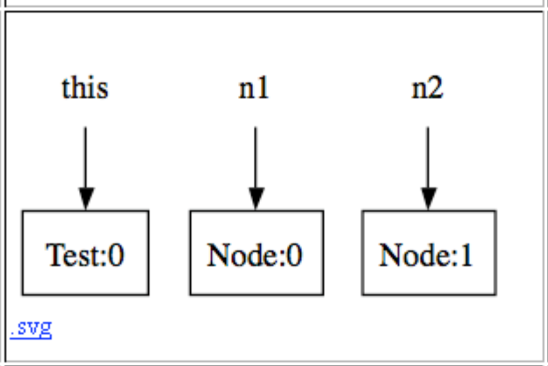
1.

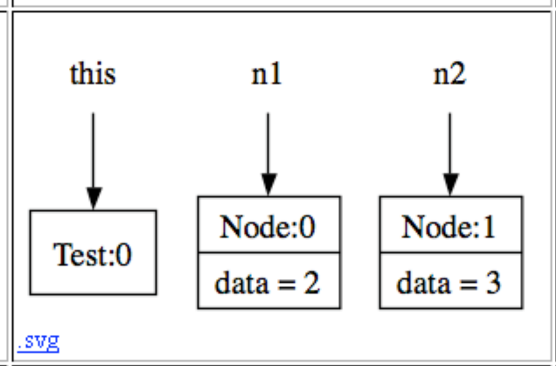




Error

2.





OK

Foo3Test1.java

All Paths for Foo3 Example

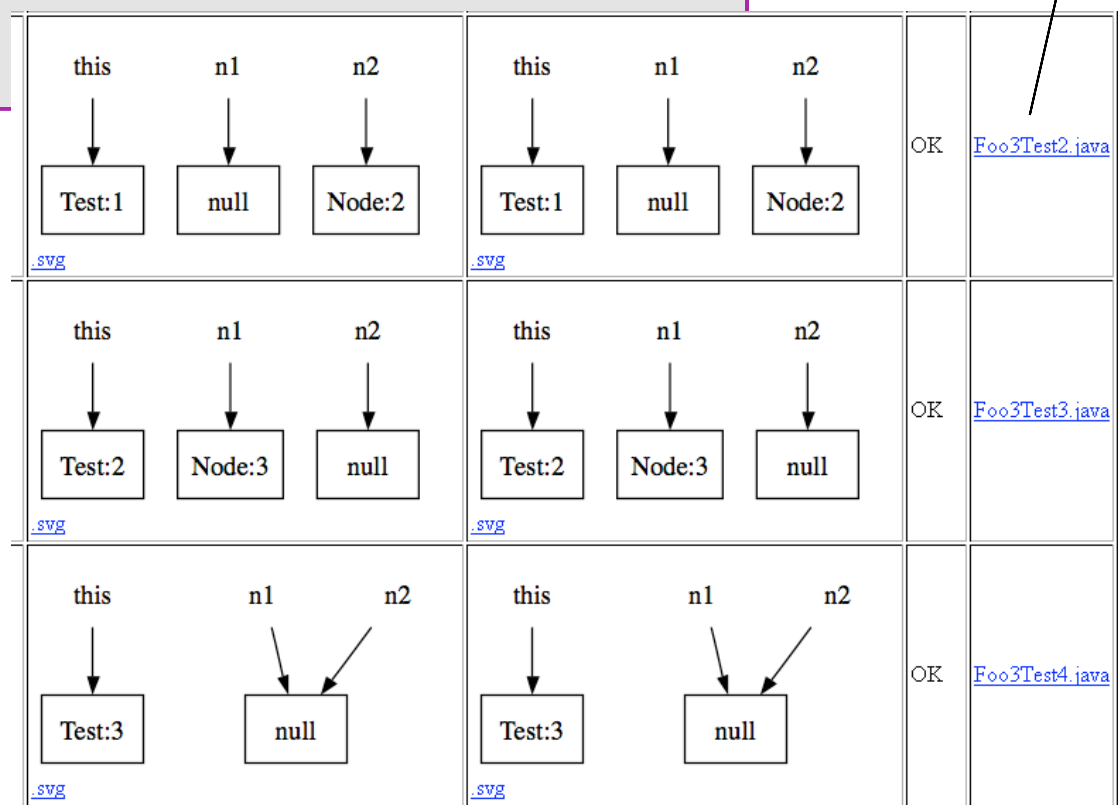
```
void foo3(Node n1, Node n2) {
  if (n1 != null && n2 != null) {
    n1.x = 2;
    n2.x = 3;
    assert (n1.x == 2 && n2.x == 3);
  }
}
```

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) {
    …
  }
}
```

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

```
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 list

Heavyweight & Lightweight

Geez, that's a ***huge*** ***contract***! Who is going to write all that contract rubbish?

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.***

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;
}
```

Samples of Design Intentions

Specifying specific patterns

- Units

```
class Rod { @Meter double length;
            @Celcius double temperature; }
```

- One can define domain-specific annotations that can be checked by Kiasan

Benefits of Strong Specs?

Why don’t we actually step through the ***methodology/workflow*** for constructing and leveraging stronger specifications….

OK, I can see how codified design intentions could be useful, but what about ***heavyweight*** specs?

Executable Specifications

I. Write invariants, pre/post-conditions

- Kiasan will eventually support checking of specifications written in JML.
- Currently specifications must be written as executable (pure) boolean-valued Java methods.

Executable Specifications

Invariant of a binary search tree

```
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));
}
```

Dealing with Heap Data

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

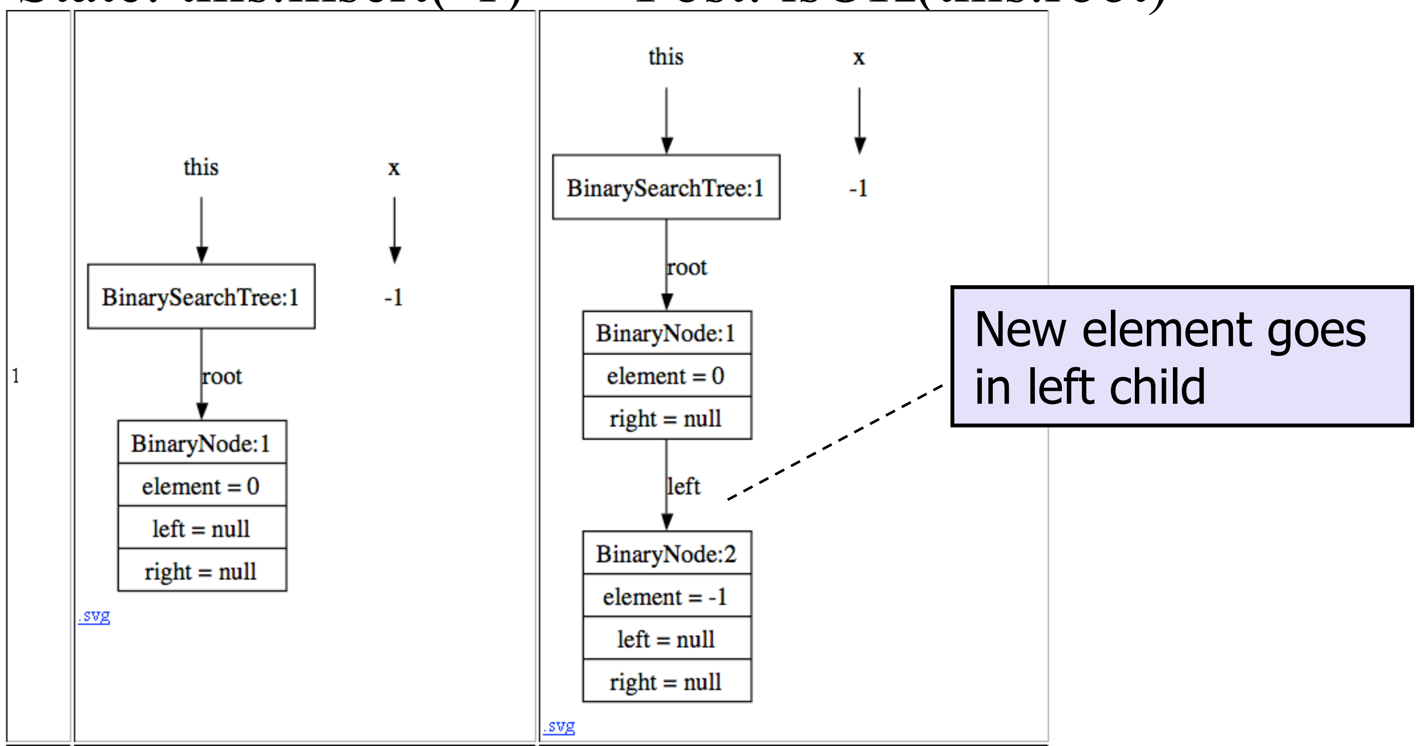
```
@Assertion(@Case(
      pre = "repOK(root)",
      post = "repOK(root)"))
public void insert( int x ) {root = myins( x, root ); }

@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

20 cases for k=3

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

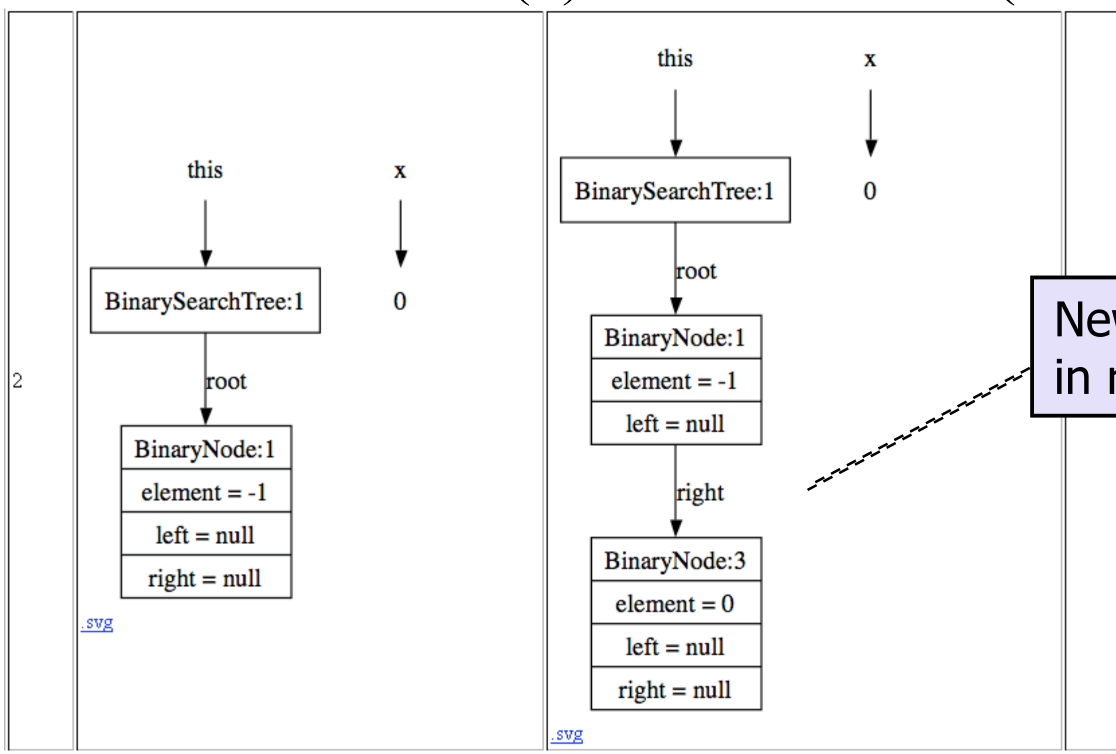


Tool verifies that pre/post conditions are satisfied and gives pre/post-state pairs for each path through the method

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 do this manually!

Need External Evidence and Automated Evidence Checker

That's great, but why should I ***trust*** your tool? Are you telling me that ***my*** developers should check the scenarios manually?

What is the ***external evidence*** that they are correct, and how to check them ***automatically***?

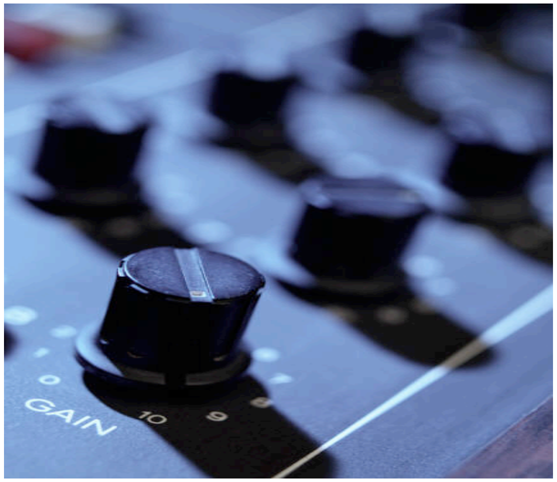
Automatic Test Case Generators and Assisting Code Inspection

- Extends the generation of error scenarios to generate test cases
  - generate cases for "good" behaviors
  - while test generations should not based on code alone, this is valuable for regression testing
- This can be used for code inspection
  - the generated input/output (side-effects) of a method give some clue about the method's behavior
  - generalize to any statement block

Connecting With SE QA Tech

- 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

Kiasan Methodology



- 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.

Brief Summary of Capabilities

- Static checker for common runtime errors
  - run in background for low bounds
  - run parallelizing checks at night with high bounds
  - similar Java checking tools such as ESC/Java, with focus on
    - supports heap data
    - provides error trace & input/output pairs
- Test-case generation with complete path coverage up to bounds – more powerful than commercial tools
  - Run in background in Eclipse, and update test suite with changes
- Gentle introduction to the inclusion of specifications (from light-weight to heavy-weight)
  - Support checking directly with controllable coverage
  - Generate tests as *evidence* for either bugs found or to illustrate coverage via a test suite
  - Argue that writing specs is easier than writing a high-coverage unit test suite – plus, specs can be leveraged in multiple ways

Kiasan Future Work

- Significant engineering effort to create easy-to-use tool that can be dropped to developers
  - specification language and methodology
    - next generation (extensible) JML
  - expressing properties/design intentions (e.g., regions)
  - usability in configuring the analysis
  - integration with various theorem provers (SMT-LIB)
  - IDE integration
- Library models/abstractions
- Parallel/distributed solutions
- Integrating abstract interpretation and algebraic specification
- Concurrency, secure information flow, etc.

For More Information…

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

Bogor/Kiasan Project
http://bogor.projects.cis.ksu.edu

Indus Project
http://indus.projects.cis.ksu.edu