

Topics in Software Engineering (CS527): Software Testing and Debugging **Spec-based Testing**

Spring 2023
Lingming Zhang



Reminder

- Check your presentation schedule on course website
 - The presenters for next week are: YH, EC, YS, XS
- Proposal presentation (Feb 16, in class)
- Proposal submission (Feb 18, 11:59pm)
 - Only one team member of each team needs to submit the proposal on campuswire

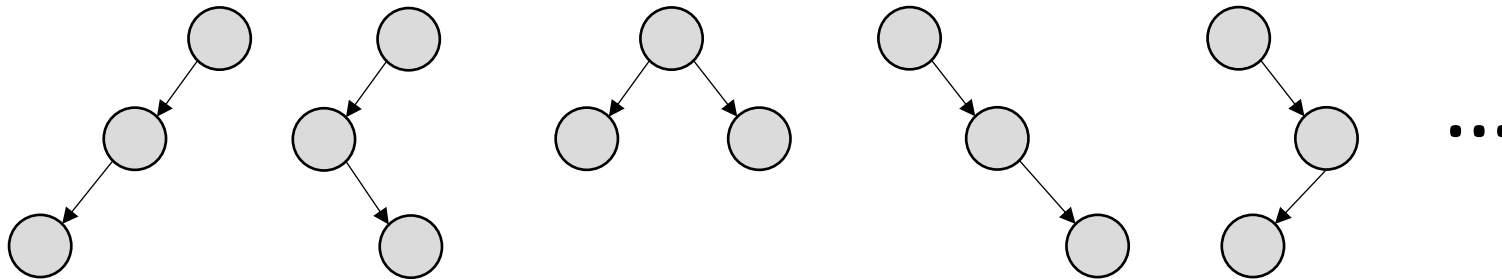
Spec-based test generation

```
// specification for removing from binary tree  
/*@ public normal_behavior  
  @ requires has(n); // precondition  
  @ ensures !has(n); // postcondition */
```

Generate

Program

Execute



Generated tests

This class

- Korat: Automated Testing Based on Java Predicates (ISSTA'02)
- TestEra: A Novel Framework for Automated Testing of Java Programs (ASE'01)

What specifications to use?

- Formal specifications in specifically designed languages (e.g., Z and Alloy)
 - Precise and concise
 - Hard to write (steep learning curve)
- Korat directly utilizes **Java predicates** for encoding the specifications
 - Some existing formal specifications (e.g., JML) can be automatically transformed to Java
 - **Programmers can also use the full expressiveness of the Java language to write specifications!**

Korat predicate

```
class BinaryTree {  
    private Node root; // root node  
    static class Node {  
        private Node left; // left child  
        private Node right; // right child  
    }  
    ...  
}
```

BinaryTree Program

```
public boolean repOk() {  
    if (root == null) return true; // empty tree  
    Set visited = new HashSet();  
    visited.add(root);  
    LinkedList workList = new LinkedList();  
    workList.add(root);  
    while (!workList.isEmpty()) {  
        Node current = (Node)workList.removeFirst();  
        if (current.left != null) {  
            if (!visited.add(current.left)) return false; // tree has no cycle  
            workList.add(current.left);  
        }  
        if (current.right != null) {  
            if (!visited.add(current.right)) return false; // tree has no cycle  
            workList.add(current.right);  
        }  
    }  
    return true; // valid non-empty tree  
}
```

Korat RepOK predicate

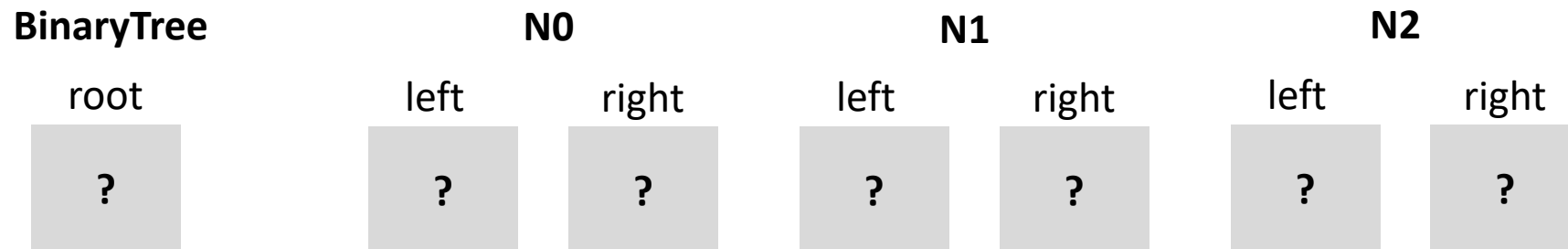
Finitization

- **Finitization**: a set of bounds that limits the size of the inputs
 - Specifies the number of objects for each used class
 - A set of objects of one class forms a **class domain**
 - Specifies the set of classes whose objects each field can point to
 - The set of values a field can take forms its **field domain**
 - Note that a field domain is a union of some class domains

```
public static Finitization finBinaryTree(int NUM_Node) {  
    Finitization f = new Finitization(BinaryTree.class);  
    ObjSet nodes = f.createObjectSet("Node", NUM_Node);  
    nodes.add(null);  
    f.set("root", nodes);  
    f.set("Node.left", nodes);  
    f.set("Node.right", nodes);  
    return f;  
}
```

State space

- **finBinaryTree** with **NUM_Node=3**



- Each field with type Node includes 4 possible choices:
 - {null, N0, N1, N2}
- Total number of possible tests for a tree with 3 nodes:
 - $4 * (4 * 4)^3 = 2^{14} = 16,384$
- Total number of possible tests for a tree with **n** nodes:
 - $(n+1) * ((n+1) * (n+1))^n = (n+1)^{2n+1}$

State space: more examples

- The number of “trees” explodes rapidly!
 - $n=3$: over 16,000 “tests”
 - $n=4$: over 1,900,000 “tests”
 - $n=5$: over 360,000,000 “trees”
- Limit us to only very small input sizes

Are they all valid tests?

State space: examples

- **finBinaryTree** with **NUM_Node=3**



BinaryTree

root

N0

N0

left

N1

right

N2

N1

left

null

right

null

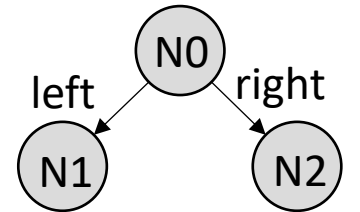
N2

left

null

right

null



BinaryTree

root

N0

N0

left

N1

right

N1

N1

left

null

right

null

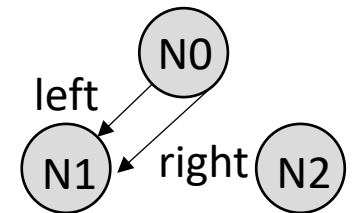
N2

left

null

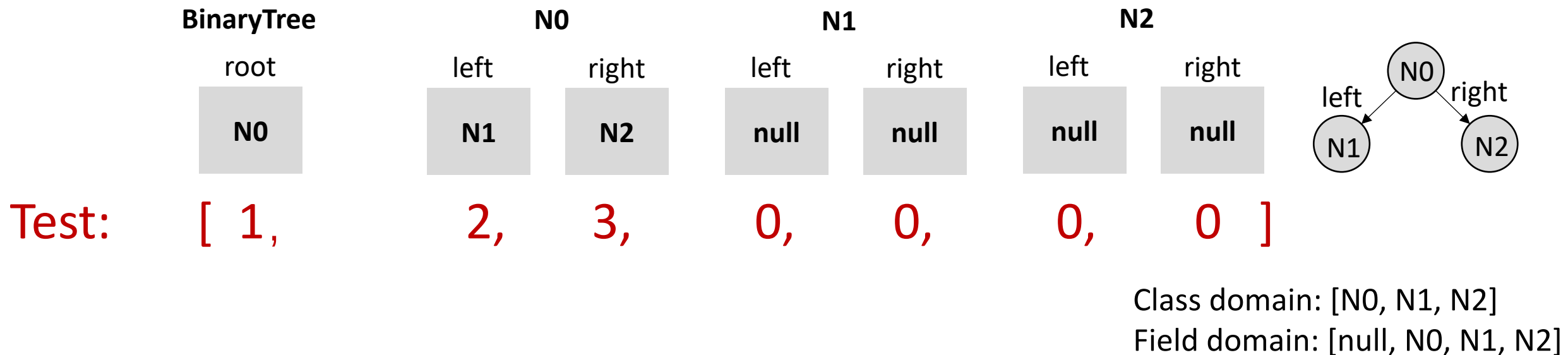
right

null



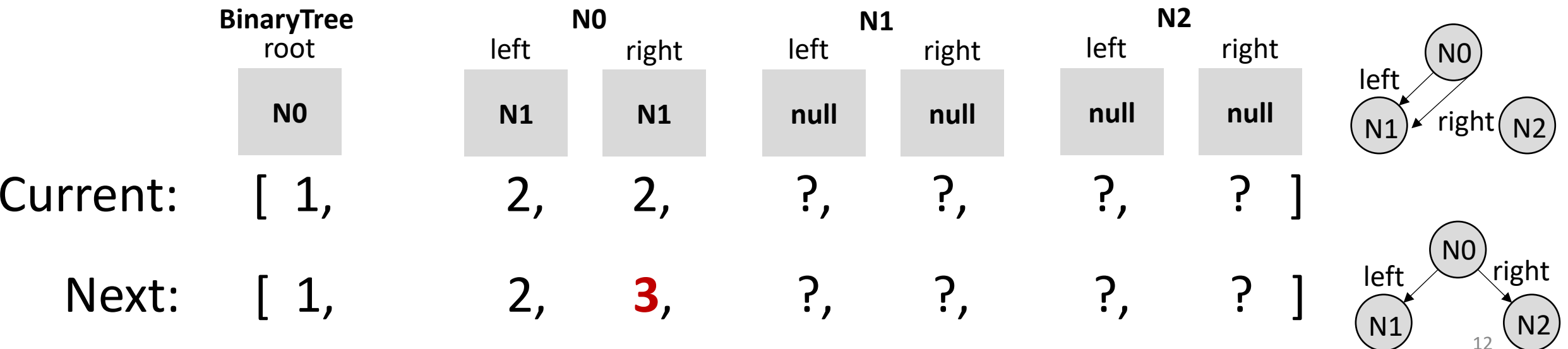
State space: vector representation

- To systematically explore the state space, Korat orders all the elements in every class domain and every field domain
 - The ordering in each field domain is consistent with the orderings in the class domains
- Each candidate input is then a vector of field domain indices!

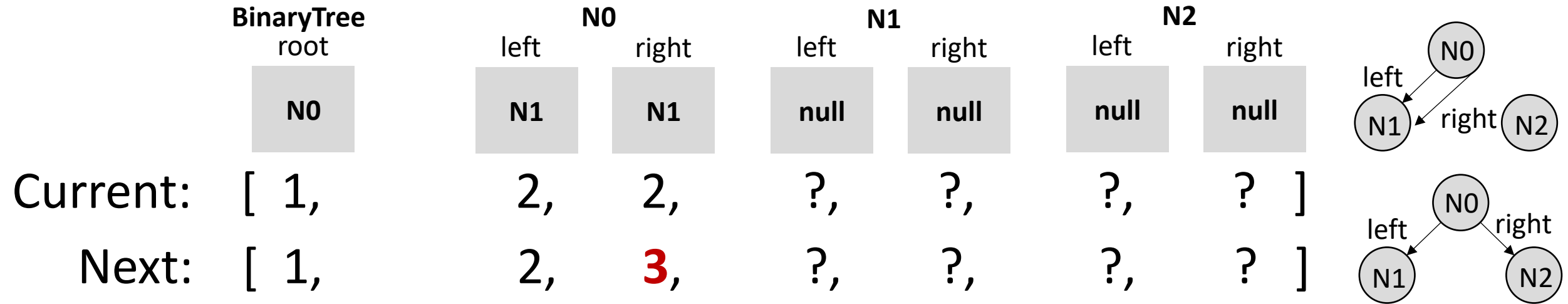


Search

- The search starts with the candidate vector set to all zeros
- Then, iterate through the following steps:
 - Construct the actual test based on the current vector
 - Invoke **repOK()** to check the test validity and record accessed field ordering
 - Increment the field domain index for the last field in the **recorded field ordering**
 - If the index exceeds the limit, reset it to 0 and increment the previous field in field ordering

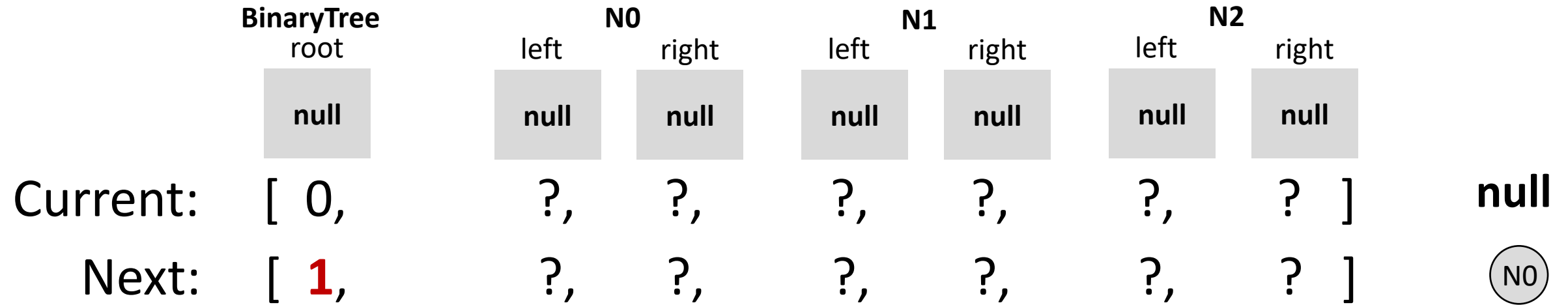


Search: why field ordering accessed matters



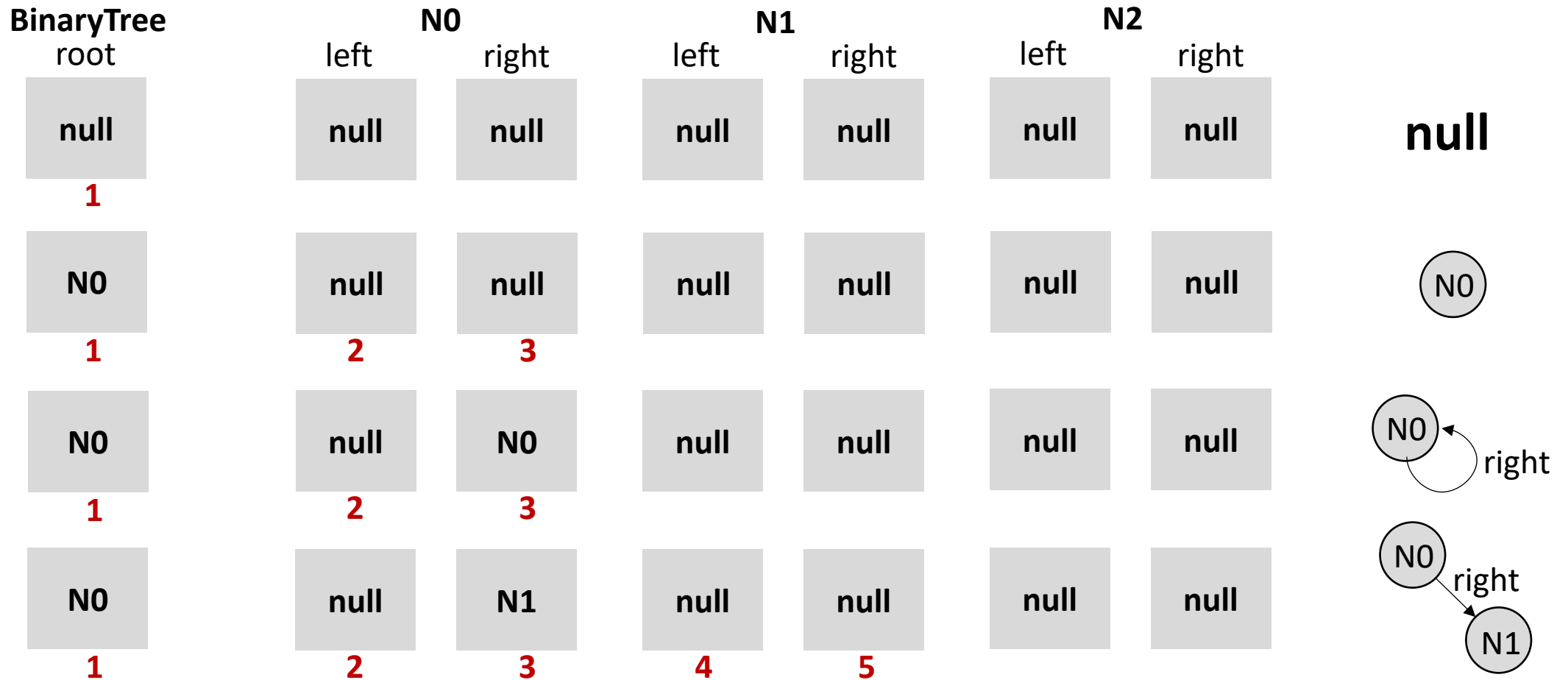
- Any test vectors of the form $[1, 2, 2, ?, ?, ?, ?]$ are invalid!
- Keeping the accessed field ordering enables us to prune all such tests
 - 4^4 tests pruned for this single step!

Search: why field ordering accessed matters (cont.)



- Only the root is accessed since it is **null**
- Any test vectors of the form $[0,?, ?, ?, ?, ?, ?]$ do not need to be repeated!
- Keeping the accessed field ordering enables us to prune all such tests
 - 25% of all tests pruned by this single test input!

Search: example

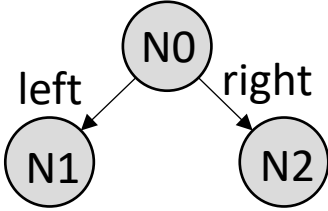
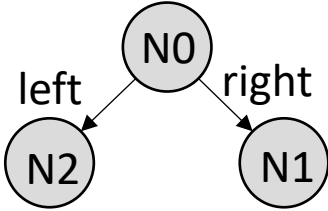


Isomorphism

- **O**: $\mathbf{O}_1 \cup \dots \cup \mathbf{O}_n$, the sets of objects from **n** classes
- **P**: the set consisting of **null** and all values of primitive types that objects in **O** can reach
- Two candidates, **C** and **C'**, are *isomorphic* iff there is a permutation π on **O**, mapping objects from \mathbf{O}_i to objects from \mathbf{O}_i for all $1 \leq i \leq n$, such that: $\forall o, o' \in \mathbf{O}. \forall f \in \text{fields}(o). \forall p \in \mathbf{P}.$
 - $o.f == o'$ in **C** iff $\pi(o).f == \pi(o')$ in **C'** AND
 - $o.f == p$ in **C** iff $\pi(o).f == p$ in **C'**

Two data structures are isomorphic if a permutation exists between the two that preserves structure

Isomorphism: examples

	BinaryTree	N0		N1		N2		
	root	left	right	left	right	left	right	
	N0	N1	N2	null	null	null	null	
Test1:	[1,	2,	3,	0,	0,	0,	0]	
	N0	N2	N1	null	null	null	null	
Test2:	[1,	3,	2,	0,	0,	0,	0]	
	N1	null	null	N0	N2	null	null	
Test3:	[2,	0,	0,	1,	3,	0,	0]	

They are isomorphic!

We just need one of them...

Nonisomorphism

- **Algorithm:** only allow an index into a given class domain to exceed previous indices into that domain by 1
 - Initial prior index: -1
- **Example:** assume we are generating tests with three fields from the same class domain

0 0 0

0 0 1

0 0 ~~X~~

0 1 0

0 1 1

0 1 2

0 ~~X~~ 0

0 ~~X~~ 1

0 ~~X~~ 2

~~X~~ 0 0

~~X~~ 0 1

~~X~~ 0 2

~~X~~ 1 0

~~X~~ 1 1

~~X~~ 1 2

~~X~~ 2 0

~~X~~ 2 1

~~X~~ 2 2

~~X~~ 0 0

~~X~~ 0 1

~~X~~ 0 2

~~X~~ 1 0

~~X~~ 1 1

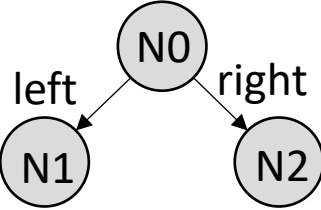
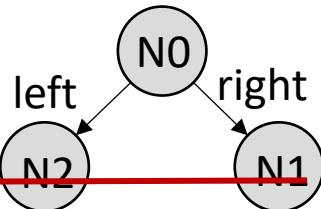
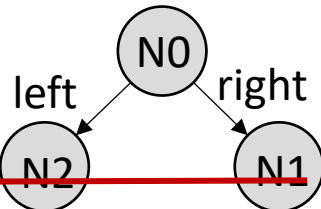
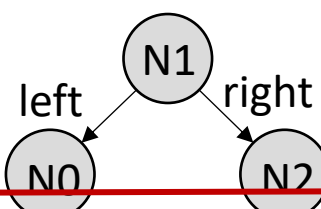
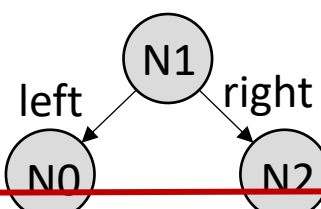
~~X~~ 1 2

~~X~~ 2 0

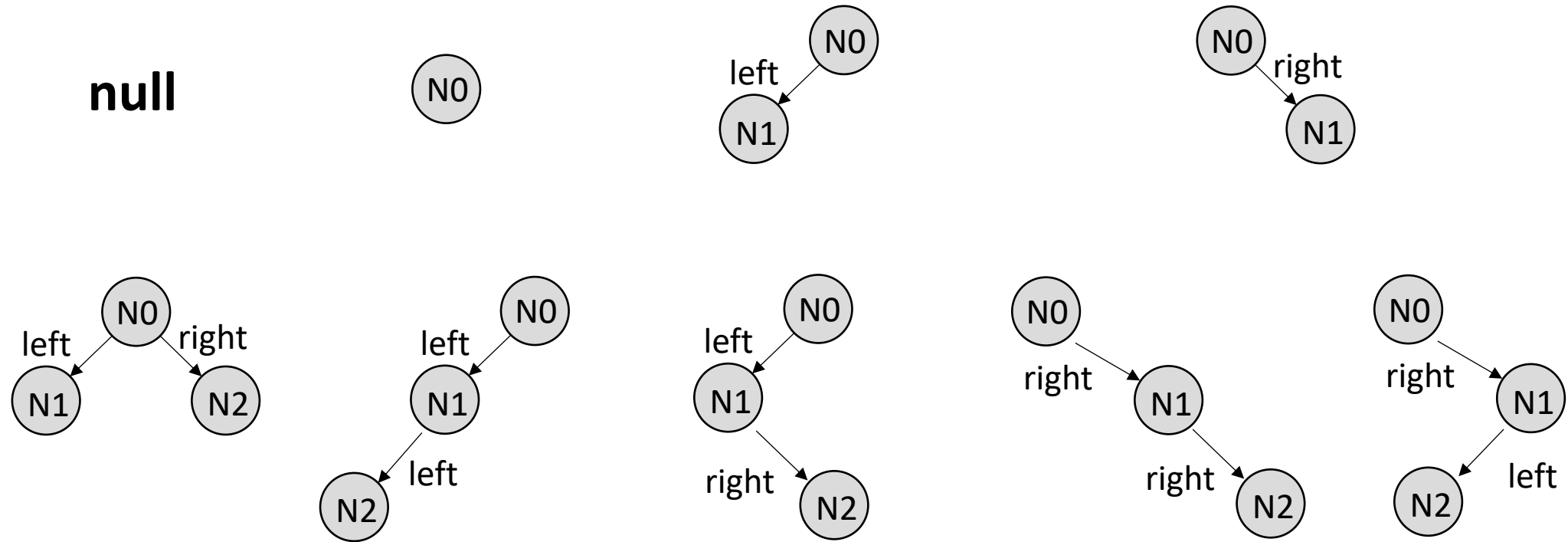
~~X~~ 2 1

~~X~~ 2 2

Nonisomorphism: more examples

	BinaryTree	N0		N1		N2		
	root	left	right	left	right	left	right	
	N0	N1	N2	null	null	null	null	
Test1:	[1,	2,	3,	0,	0,	0,	0]	
	N0	N2	N1	null	null	null	null	
Test2:	[1,	2,	2,	0,	0,	0,	0]	 
	N1	null	null	N0	N2	null	null	
Test3:	[2,	0,	0,	1,	3,	0,	0]	 

Korat results for BinaryTree with up to 3 nodes



- Only 9 valid tests out of 2^{14} possibilities!

Test generation

- Valid test cases for a method must satisfy its precondition
- Korat uses a class that represents method's inputs:
 - One field for each parameter of the method (including the implicit *this*)
 - A **repOk** predicate that uses the precondition to check the validity of method's inputs
- Given a finitization, Korat then generates all inputs with **repOk=true**

```
class BinaryTree_remove {  
    BinaryTree This; // the implicit "this"  
    BinaryTree.Node n; // the Node parameter  
    //@ invariant repOk();  
    public boolean repOk() {  
        return This.has(n);  
    }  
}
```

```
public static Finitization  
    finBinaryTree_remove(int NUM_Node) {  
        Finitization f = new Finitization(BinaryTree_remove.class);  
        Finitization g = BinaryTree.finBinaryTree(NUM_Node);  
        f.includeFinitization(g);  
        f.set("This", g.getObjects(BinaryTree.class));  
        f.set("n", /**/);  
        return f;  
    }
```

Test generation for *remove(Node n)*

Test oracle

- To check partial correctness of a method, a simple test oracle could just invoke **repOk** in the post-state to check the class invariant
- The current Korat implementation uses the JML tool-set to automatically generate test oracles from method postconditions

```
//@ public invariant repOk(); //class invariant
/*@ public normal_behavior
    @ requires has(n); // precondition
    @ ensures !has(n); // postcondition @*/
public void remove(Node n) {
    ...
}
```

JML specification for removing from binary tree

Testing activity	JUnit	JML+JUnit	Korat
Generating tests			X
Generating oracle		X	X
Running tests	X	X	X

Benchmark subjects

benchmark	package	initization parameters
BinaryTree	<code>korat.examples</code>	NUM_Node
HeapArray	<code>korat.examples</code>	MAX_size, MAX_length, MAX_elem
LinkedList	<code>java.util</code>	MIN_size, MAX_size, NUM_Entry, NUM_Object
TreeMap	<code>java.util</code>	MIN_size, NUM_Entry, MAX_key, MAX_value
HashSet	<code>java.util</code>	MAX_capacity, MAX_count, MAX_hash, loadFactor
AVTree	<code>ins.namespace</code>	NUM_AVPair, MAX_child, NUM_String

Overall results

benchmark	size	time (sec)	structures generated	candidates considered	state space
BinaryTree	8	1.53	1430	54418	2^{53}
	9	3.97	4862	210444	2^{63}
	10	14.41	16796	815100	2^{72}
	11	56.21	58786	3162018	2^{82}
	12	233.59	208012	12284830	2^{92}
HeapArray	6	1.21	13139	64533	2^{20}
	7	5.21	117562	519968	2^{25}
	8	42.61	1005075	5231385	2^{29}
LinkedList	8	1.32	4140	5455	2^{91}
	9	3.58	21147	26635	2^{105}
	10	16.73	115975	142646	2^{120}
	11	101.75	678570	821255	2^{135}
	12	690.00	4213597	5034894	2^{150}
TreeMap	7	8.81	35	256763	2^{92}
	8	90.93	64	2479398	2^{111}
	9	2148.50	122	50209400	2^{130}
HashSet	7	3.71	2386	193200	2^{119}
	8	16.68	9355	908568	2^{142}
	9	56.71	26687	3004597	2^{166}
	10	208.86	79451	10029045	2^{190}
	11	926.71	277387	39075006	2^{215}
AVTree	5	62.05	598358	1330628	2^{50}

Korat vs. TestEra

benchmark	size	Korat			Alloy Analyzer		
		struc. gen.	total time	first struc.	inst. gen.	total time	first inst.
BinaryTree	3	5	0.56	0.62	6	2.63	2.63
	4	14	0.58	0.62	28	3.91	2.78
	5	42	0.69	0.67	127	24.42	4.21
	6	132	0.79	0.66	643	269.99	6.78
	7	429	0.97	0.62	3469	3322.13	12.86
HeapArray	3	66	0.53	0.58	78	11.99	6.20
	4	320	0.57	0.59	889	171.03	16.13
	5	1919	0.73	0.63	1919	473.51	39.58
LinkedList	3	5	0.58	0.60	10	2.61	2.39
	4	15	0.55	0.65	46	3.47	2.77
	5	52	0.57	0.65	324	14.09	3.51
	6	203	0.73	0.61	2777	148.73	5.74
	7	877	0.87	0.61	27719	2176.44	10.51
TreeMap	4	8	0.75	0.69	16	12.10	6.35
	5	14	0.87	0.88	42	98.09	18.08
	6	20	1.49	0.98	152	1351.50	50.87
AVTree	2	2	0.55	0.65	2	2.35	2.43
	3	84	0.65	0.61	132	4.25	2.76
	4	5923	1.41	0.61	20701	504.12	3.06

Korat for methods

benchmark	method	max. size	test cases generated	gen. time	test time
BinaryTree	remove	3	15	0.64	0.73
HeapArray	extractMax	6	13139	0.87	1.39
LinkedList	reverse	2	8	0.67	0.76
TreeMap	put	8	19912	136.19	2.70
HashSet	add	7	13106	3.90	1.72
AVTree	lookup	4	27734	4.33	14.63

Discussion

- Strengths
- Limitations
- Future work

This class

- Korat: Automated Testing Based on Java Predicates (ISSTA'02)
- TestEra: A Novel Framework for Automated Testing of Java Programs (ASE'01)

TestEra vs Korat

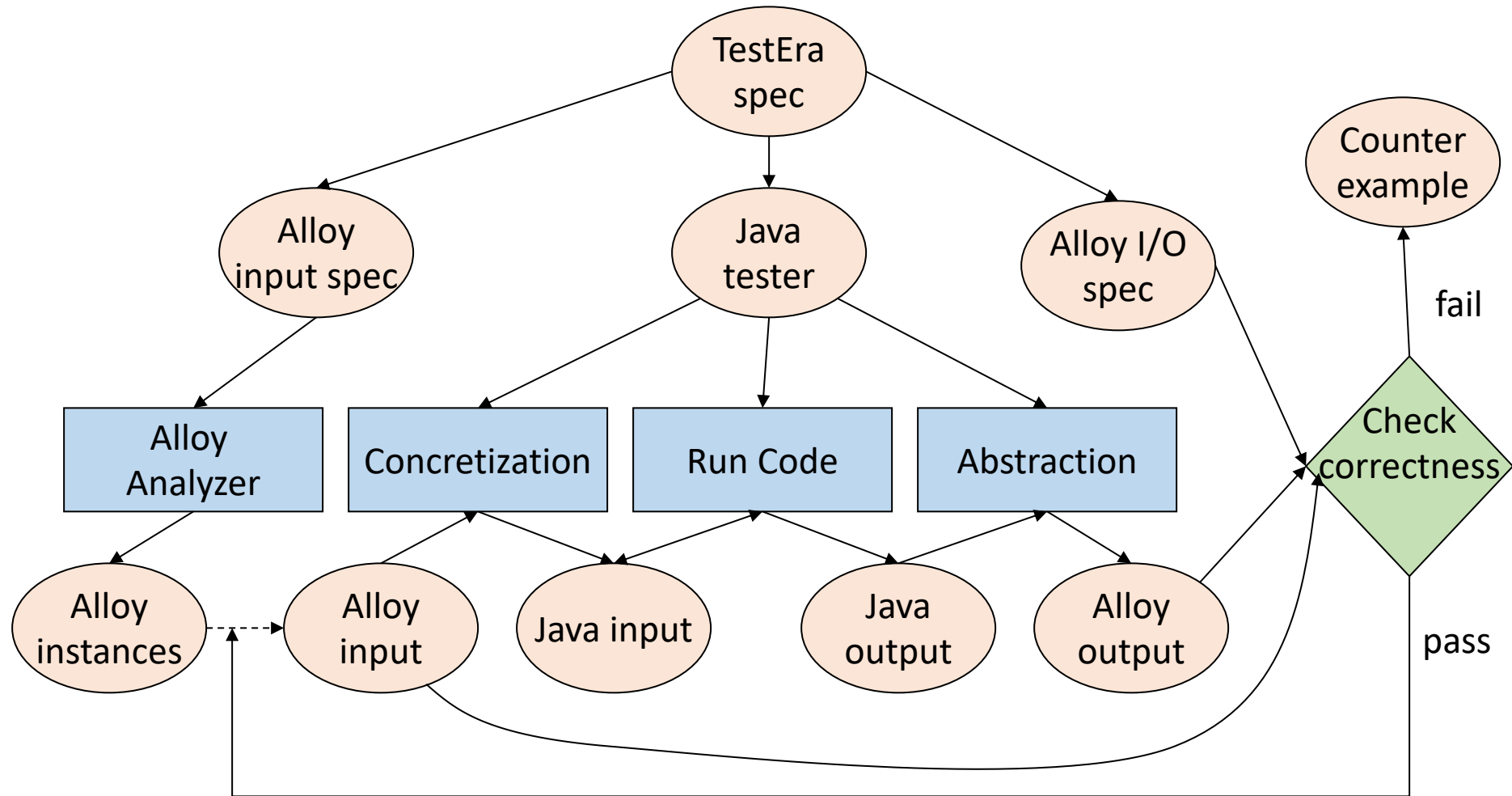
- Similarities:
 - Both target structurally complex test input generation based on specifications
 - Both automatically generate all non-isomorphic tests within a given input size
- Differences
 - TestEra uses Alloy¹ as the specification language
 - Alloy is a simple declarative language based on first-order logic
 - TestEra uses Alloy and Alloy Analyzer to generate the tests and to evaluate the correctness criteria
 - TestEra produces concrete Java inputs as counterexamples to violated correctness criteria

¹ <https://www.csail.mit.edu/research/alloy>

TestEra components

- A specification of inputs to a Java program written in Alloy
 - Class invariant and precondition
- A correctness criterion written in Alloy
 - Class invariant and post-condition
- An concretization function
 - Which maps instances of Alloy specifications to concrete Java objects
- An abstraction function
 - Which maps the concrete Java objects to instances of Alloy specifications

TestEra big picture



TestEra: example

```
class List {  
  int elem;  
  List next;  
  static List mergeSort(List l) { ... }  
}
```

Java code

- A recursive method for performing merge sort on acyclic singly linked lists

```
module list  
import integer  
sig List {  
  elem: Integer,  
  next: lone List  
}
```

Alloy model

- Signature declaration introduces the List type with functions:
 - **elem: List \rightarrow Integer**
 - **next: List \rightarrow List**

Input specification

```
module list
import integer
```

```
sig List {
  elem: Integer,
  next: lone List }
```

```
fun Acyclic(l: List) {
  all n: l.*next | lone n.~next // at most one parent
  no l.~next }                // head has no parent
```

```
one sig Input in List {}
```

```
fact GenerateInputs {
  Acyclic(Input) }
```

- \sim : transpose (converse relation)
- $*$: reflexive transitive closure
- Subsignature **Input** is a subset of **List** and it has exactly one atom which is indicated by the keyword **one**

Correctness specification

```
fun Sorted(l: List) {  
  all n: l.*next | some n.next => n.elem <= n.next.elem }  //?
```

```
fun Perm(l1: List, l2: List)  
  all e: Integer | #(e.~elem & l1.*next) =  
    #(e.~elem & l2.*next) }  //?
```

```
fun MergeSortOK(i: List, o: List) {  
  Acyclic(o)  
  Sorted(o)  
  Perm(i, o) }
```

one sig Output in List {}

```
fact OutputOK {  
  MergeSortOk(Input, Output) }
```

- #: cardinality of sets

Counter-examples

- If an error is inserted in the method for merging where (**`l1.elem <= l2.elem`**) is changed to (**`l1.elem >= l2.elem`**)
- Then TestEra generates a counter example:

Counterexample found:

Input List: 1 -> 1 -> 3 -> 2

Output List: 3 -> 2 -> 1 -> 1

TestEra: case studies

- Red-Black trees
 - Tested the implementation of Red-Black trees in `java.util.TreeMap`
 - Introduced some bugs and showed that they can catch them with TestEra framework
- Intentional Naming System
 - A naming architecture for resource discovery and service location in dynamic networks
 - Found some bugs
- Alloy Analyzer
 - Found some bugs in the Alloy Analyzer using TestEra framework

Discussion

- Strengths
- Limitations
- Future work

Thanks and stay safe!