Korat: Use implementation language
[ISSTA'02: Boyapati, Khurshid, Marinov]

Problem origin

- Darko presented TestEra at a group meeting
- Chandra asked if Java could be used instead of Alloy for writing constraints
  - The name repOk is from Barbara Liskov's book/class

Advantages

- Familiar language
- Existing development tools
- Predicates often already present

Challenge: generate tests from imperative constraints

Korat test generator

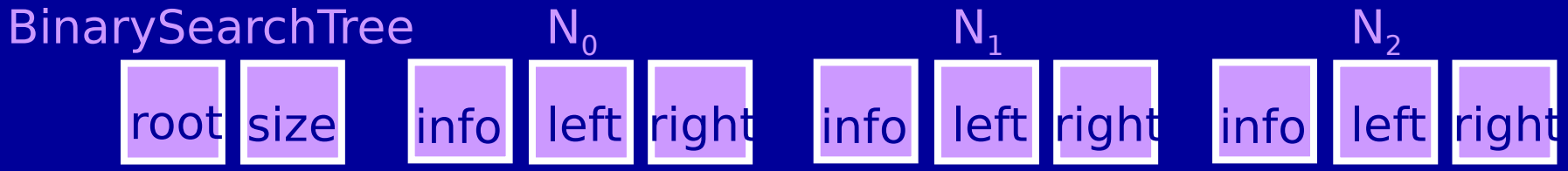
[ISSTA'02: Boyapati, Khurshid, Marinov]

Key insight: repOk executions can help prune input space

- Monitor accesses of object fields

Algorithm

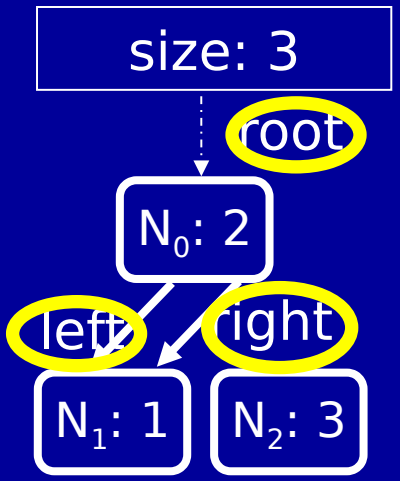
- Explores bounded input space defined by a finitization
- Represents structures using candidate vectors, e.g.,



  - For size = 3, #candidates = $4 * 1 * (3 * 4 * 4)^3 \sim 400K$
- Executes repOk on a candidate to check its validity and to determine which candidate to check next
- Provides isomorph-free generation

Execution of repOk: Field accesses

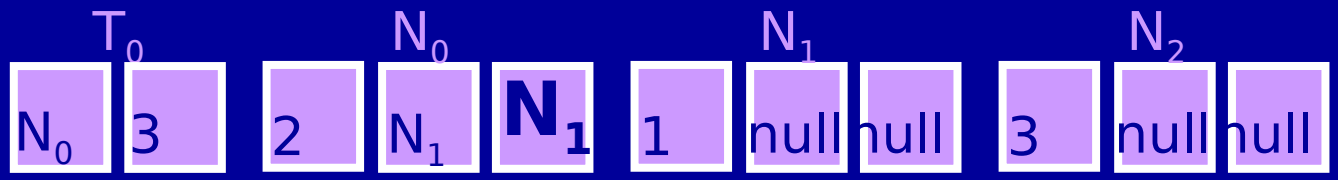
```
boolean repOk() {
  if (root == null) return size == 0; // empty tree
  Set visited = new HashSet();
  LinkedList workList = new LinkedList();
  visited.add(root);
  workList.add(root);
  while (!workList.isEmpty()) {
     Node current = (Node)workList.removeFirst();
     if (current.left != null) {
       if (!visited.add(current.left)) return false; // sharing
       workList.add(current.left);
     }
     if (current.right != null) {
       if (!visited.add(current.right)) return false; // sharing
       workList.add(current.right);
     }
  }
  if (visited.size() != size) return false; // inconsistent size
  // check binary search properties
  return true;
}
```

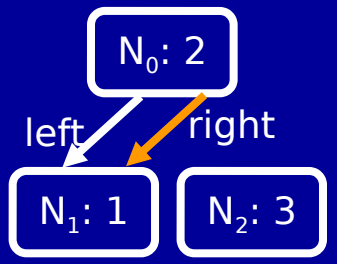


[T_0.root, N_0.left N_0.right]

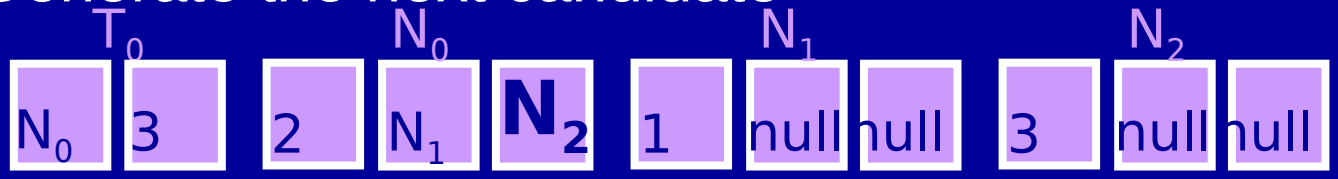
Korat search step

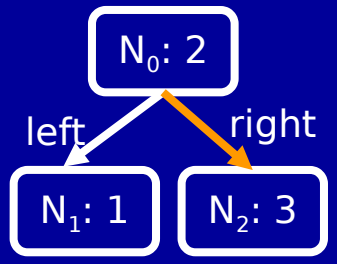
Backtrack using field access list [T_0.root, N_0.left, N_0.right]





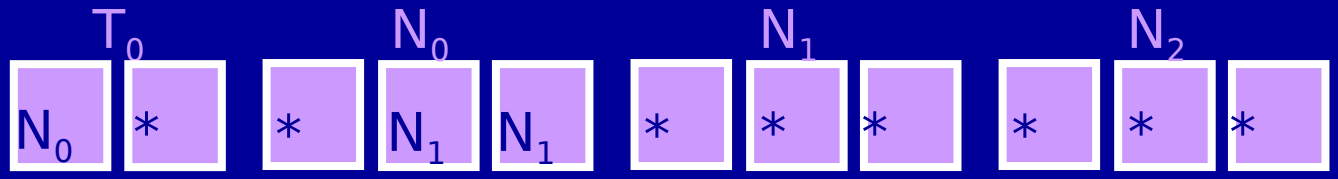
Generate the next candidate





- which satisfies repOk

Prune from the search all $3^3.4^4$ = 6,912 candidates of the form

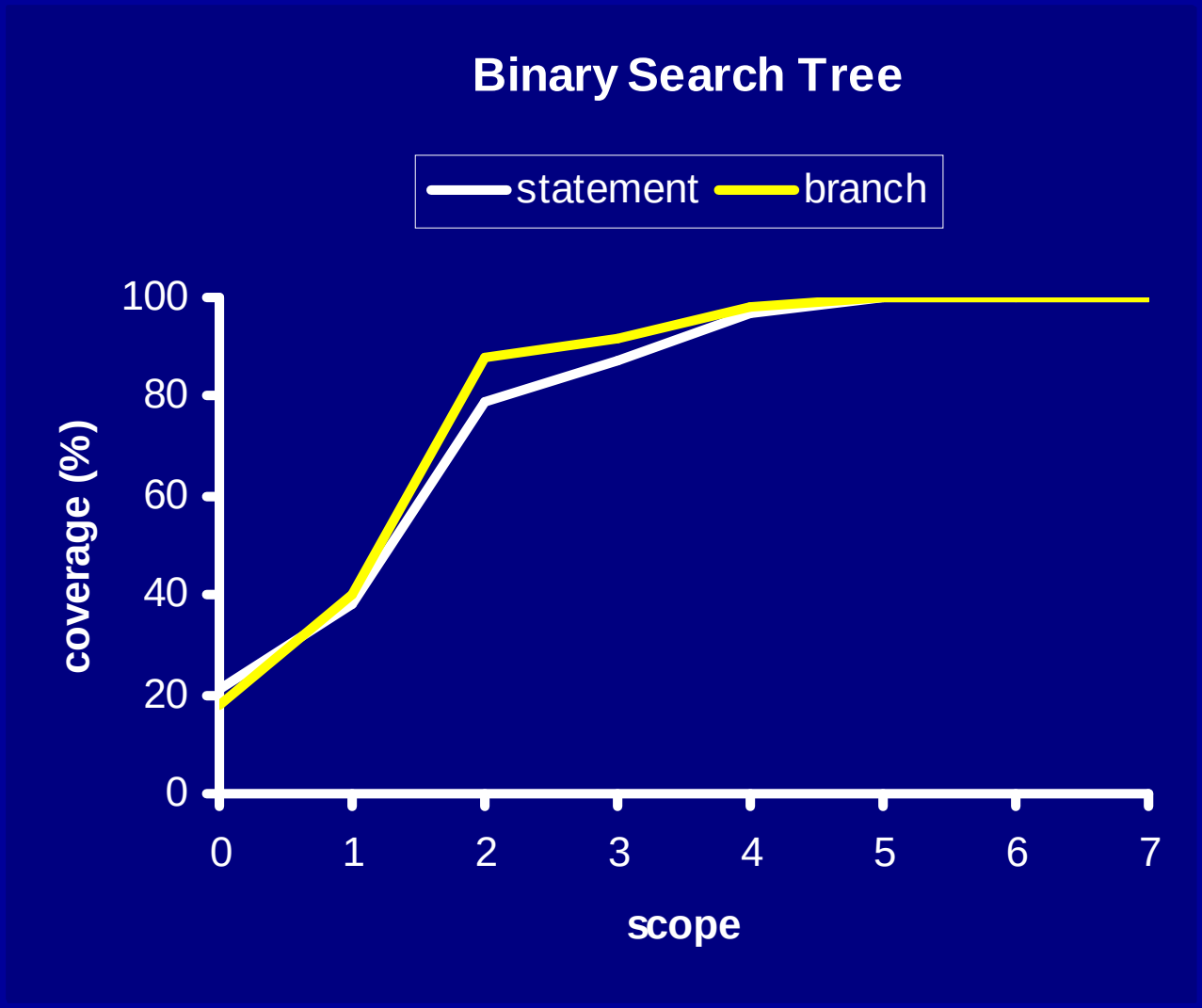


Korat performance (input generation)

benchmark	*size*	*structures generated**	*time (sec)*	*state space*
BinaryTree	8 12	1430 208012	2 234	2^{53} 2^{92}
HeapArray	6 8	13139 1005075	2 43	2^{20} 2^{29}
java.util.LinkedList	8 12	4140 4213597	2 690	2^{91} 2^{150}
java.util.TreeMap	7 9	35 122	9 2149	2^{92} 2^{130}
java.util.HashSet	7 11	2386 277387	4 927	2^{119} 2^{215}
AVTree (INS)	5	598358	63	2^{50}

**Sloane's online encyclopedia of integer sequences*

“Small scope hypothesis” for code



Trivia: Name origin

Considered names for testing with Alloy

- TestAlloy, AlloyTest, ATest, TestA…
- TestEra
  - Testing tool (Tester) using Alloy
  - Precursor of CheckEra or VerifyEra
  - Also: “saw” (the tool for cutting wood) in Darko’s native language

Natural progression to testing with Java

- Korat
  - “Saw” in one of Chandra’s native languages
  - Not a breed of cats

Outline

Overview

Example

Systematic testing using logical constraints

A few more recent projects

A bit of history

Some lessons learned at grad school

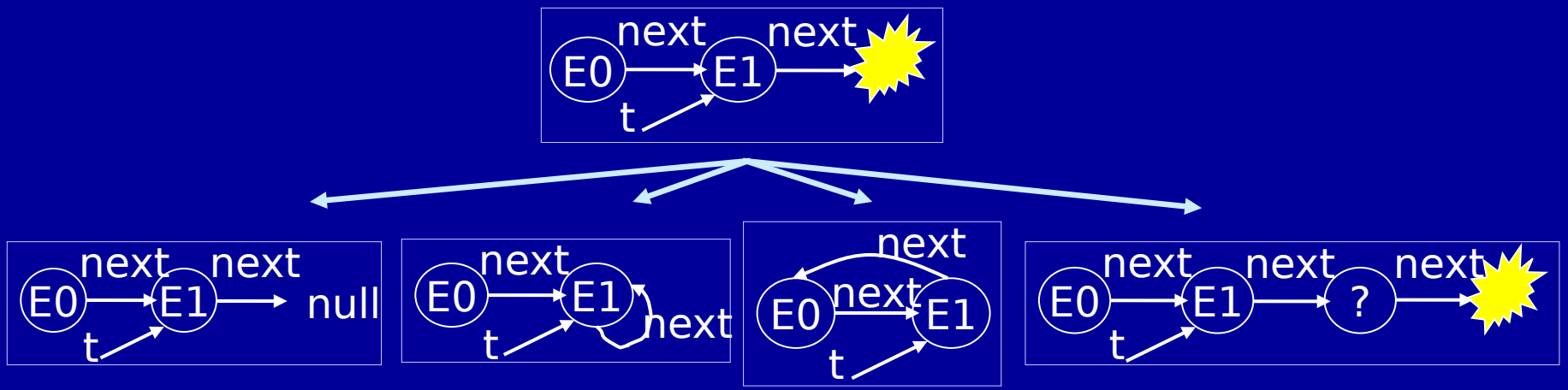
Conclusions

Generalized symbolic execution

[TACAS'03: Khurshid, Pasareanu, Visser]

Symbolic execution for primitives

Concrete execution for references using lazy initialization on access, e.g., consider "t.next"



- Included in UC-KLEE [Ramos+CAV'11]

Abstract symbolic execution for library class java.util.String

Data structure repair using assertions

[SPIN'05: Khurshid, Garcia, Suen]

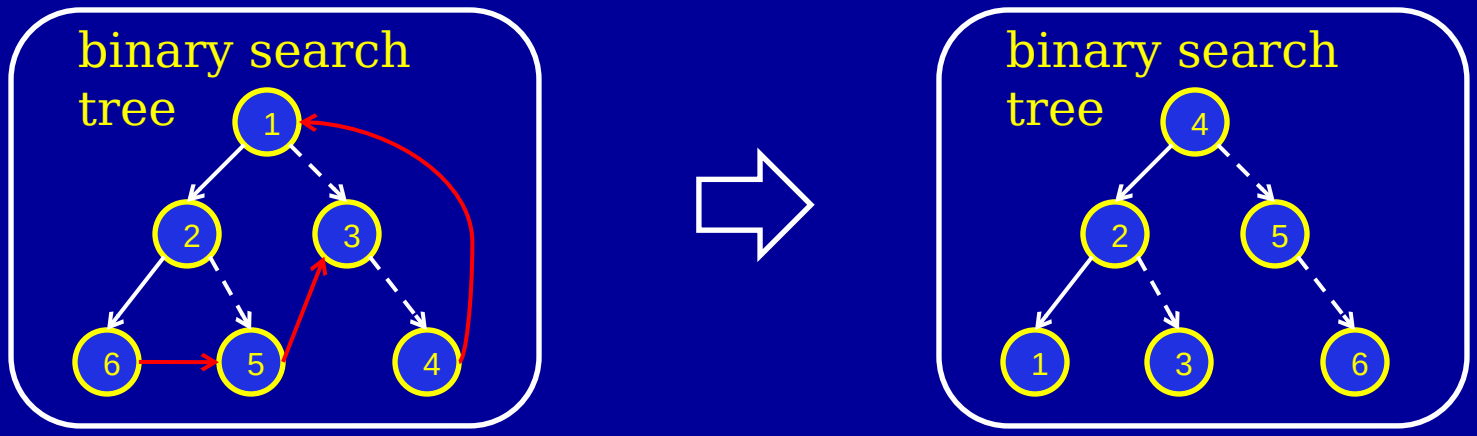
[ASE'07: Elkarablieh, Garcia, Suen, Khurshid]

[OOPSLA'07: Elkarablieh, Khurshid, Vu, McKinley]

[ISSTA'08: Elkarablieh, Marinov, Khurshid]

How to recover from runtime errors?

- Repair corrupt structure w.r.t. the violated repOk
  - Korat + symbolic execution



Generating larger structures

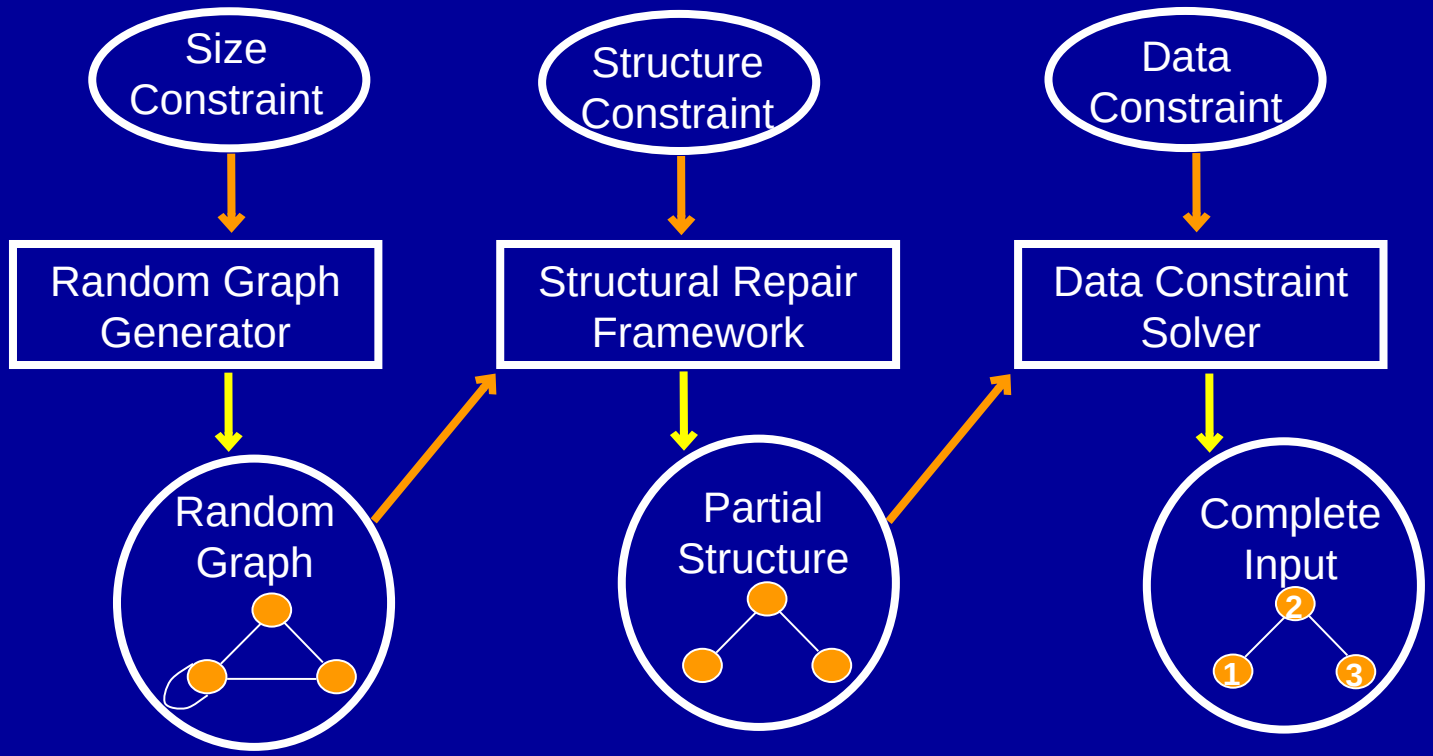
[ECOOP'07: Elkarablieh, Zayour, Khurshid]

A constraint specifies different kinds of properties

- E.g., properties of size, structure, or data

Each such property defines a generation concern

Idea: solve for the concerns separately

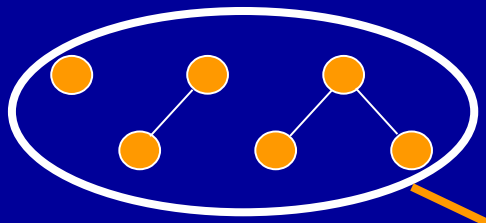


Deryaft: Dynamic constraint discovery

[TACAS'07: Malik, Pervaiz, Khurshid]

How to facilitate writing structural constraints?

- Synthesize constraints from example structures
  - In the spirit of Ernst's Daikon



Java

Deryaft

Alloy

Invariant database

Acyclic(...), Circular(...),
Two-cycle(...), <(...), ...

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

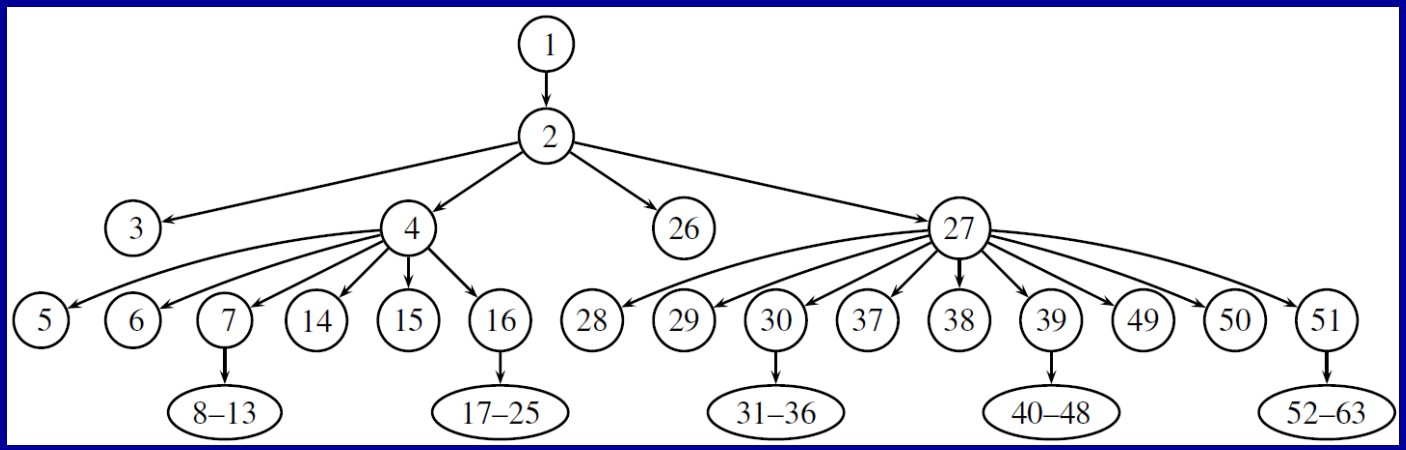
```
boolean isTree() {
  all n: root.*(left + right) {
    n !in n.^(left + right) // no directed cycles
    sole n.~(left + right)  // at most one parent
    no n.left & n.right     // left and right child are not the same
  }
}
```

Parallel Korat

[FSE'07: Misailovic, Milicevic, Petrovic, Khurshid, Marinov]
[ICST'09: Siddiqui, Khurshid]

Issue: Korat search is mostly sequential

- Input space exploration tree is highly imbalanced



Solutions for load balancing

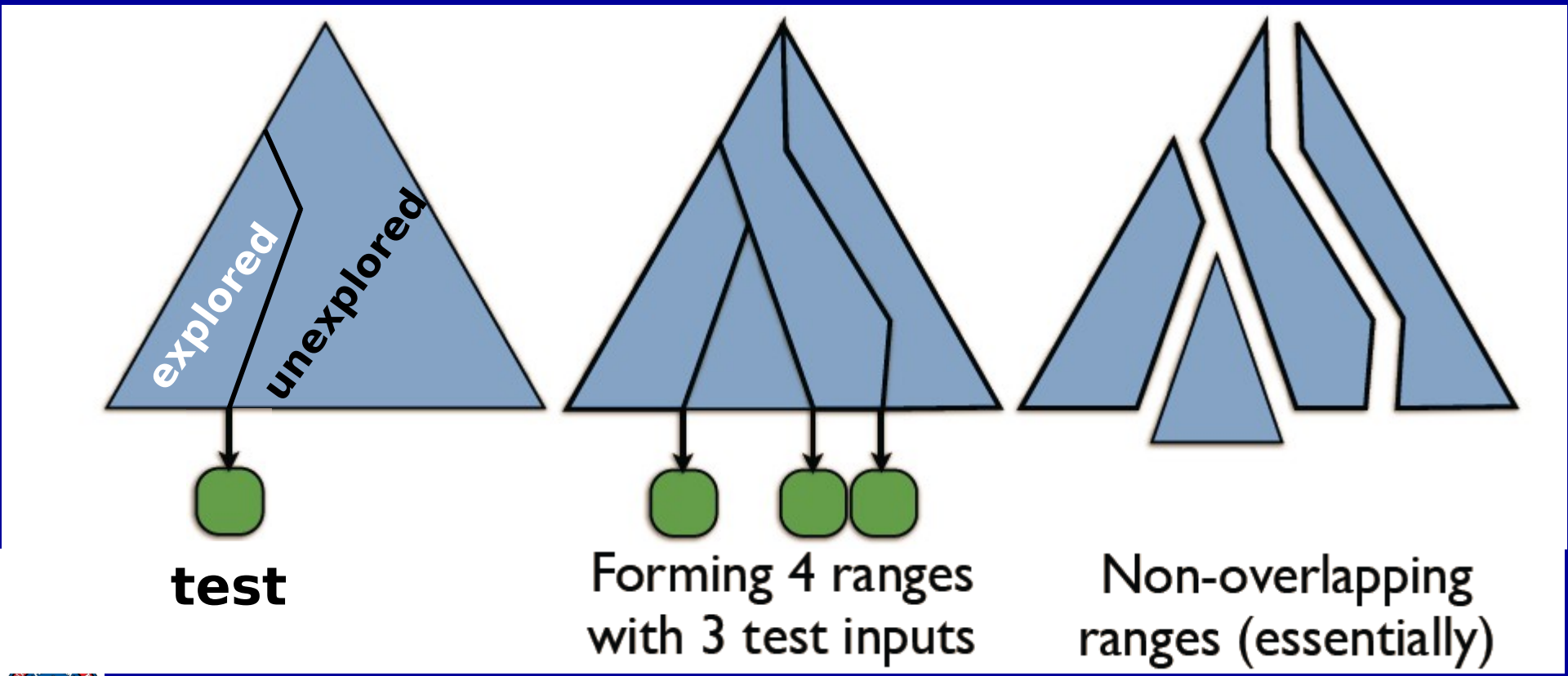
- Randomized candidate selection
- Dynamic work stealing

Ranged symbolic execution

[OOPSLA'12: Siddiqui, Khurshid]

A concrete input encodes the state of a run of symbolic execution analysis

Two (in-order) inputs range the analysis run



Dynamic programming for input generation

[FSE'12: Zaeem, Khurshid]

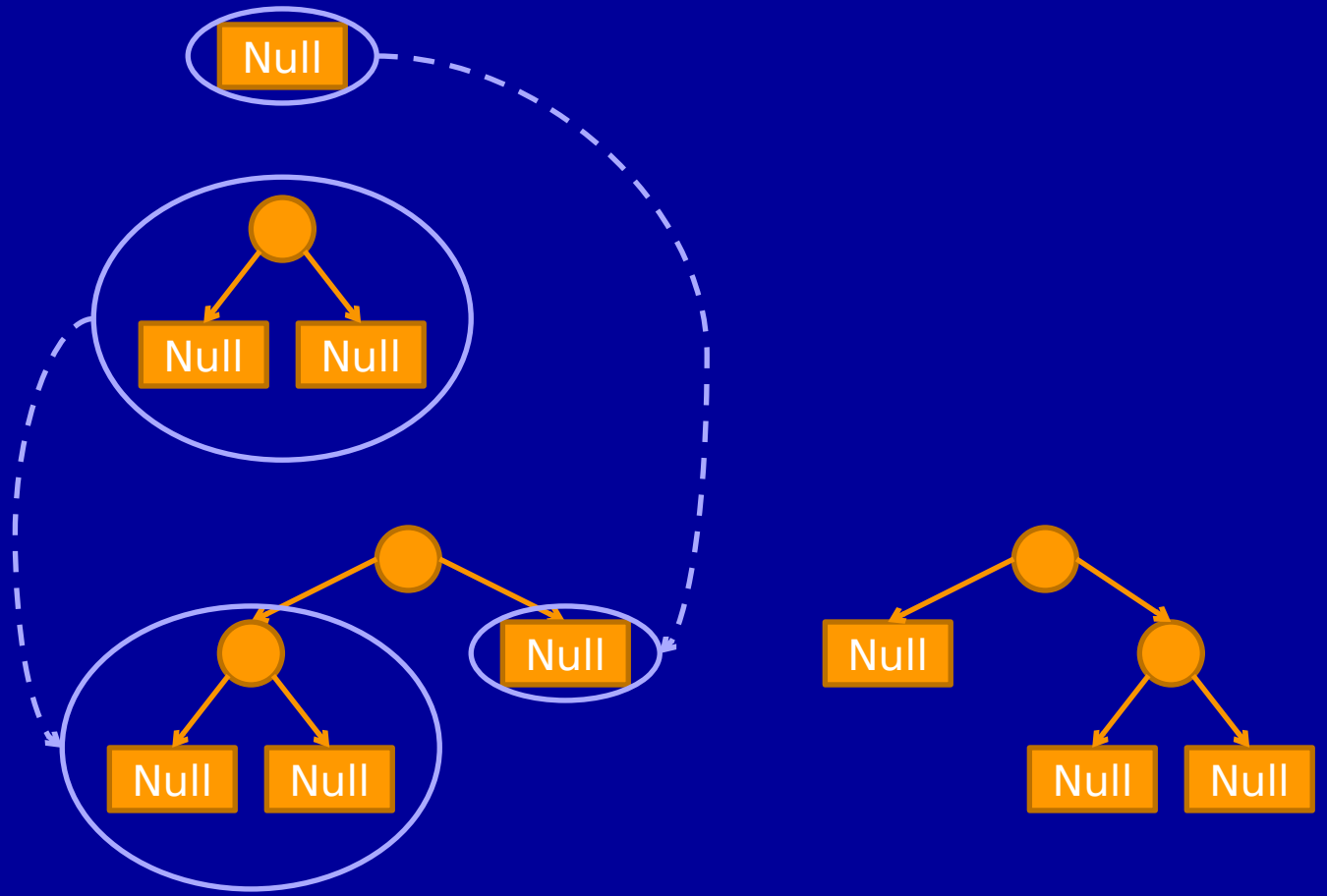
Write constraints using recursive repOk's

Solve constraints using dynamic programming

iter. 0:

iter. 1:

iter. 2:



Outline

Overview

Example

Systematic testing using logical constraints

A few more recent projects

A bit of history

Some lessons learned at grad school

Conclusions

Constraints in testing

Boyer et al. [1975], Clarke [1976], Howden [1975], King [1976] pioneered core ideas – in the context of symbolic execution

- Constraints based on execution paths – path conditions
- Constraints provided by the user – assertions
  - Focus: numeric constraints

Tools have existed for over 3 decades

- SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution [Boyer+'75]
- EFFIGY
  - Symbolic Execution and Program Testing [King'76]

Constraints in SELECT [Boyer+'75]

G. User Supplied Assertions as an Adjunct to the Program Code

As another mode of operation it is possible to insert assertions, possibly in the form of programs themselves, at various points in the program including the output. These assertions can serve as

(2) constraint conditions that enable a user to define subregions of the input space from which SELECT is to generate the test data, or

(3) specifications for the intent of the program from which it is possible to verify the paths of the program. Note that this does not imply that the program itself is correct, which would require that all program paths are verified.

Path-based verification and need to support debugging [King-PhD-CMU'69]

When a verification condition is found not to be a theorem, one usually is able to exhibit a set of values for the variables which make it evaluate to 'false'. The linear

solver in our prover should be modified to produce a counter-example set of values whenever the proof fails. These values can be used to form a particular state vector for some point in the program where the program and assertions disagree. A verifier which was able to construct such counter-examples for erroneous programs would be an extremely useful debugging aid. Other useful aids would also evolve from careful consideration of the whole process with debugging in mind.

Assertions in EFFIGY [King'76] (1)

8. Program Correctness, Proofs, and Symbolic Execution

That is, one must show, using *any* set of variable values which satisfy the predicate at the beginning of the path, that the values resulting from execution along the path must satisfy the predicate at the end.

One can prove the correctness of each path by executing it symbolically as follows:

1. Change the ASSERT at the beginning of the path to an ASSUME; change the ASSERT at the end of the path to a PROVE.
2. Initialize the path condition to *true* and all the program variables to distinct symbols say, α_1 , α_2 , ...
3. Execute the path symbolically. Whenever an unre-

Assertions in EFFIGY [King'76] (2)

symbolic testing. If one is strictly confined to symbolic execution without the use of any user introduced predicates, *pc* and the expressions requiring proof are syntactically *and semantically* determined by the programming language. However, the predicate semantics in correctness proofs derive from the *problem* area of the program and not the programming language.

It is this difference that convinces us that symbolic execution for *testing programs* is a more exploitable technique in the short term than the more general one of program verification.

Outline

Overview

Example

Systematic testing using logical constraints

A few more recent projects

A bit of history

Some lessons learned at grad school

Conclusions

MulSaw: Some lessons learned (1)

Collaborate (across research groups)

- There would be no TestEra (Korat) without two (three) students from two different groups working together
- Find a good match for collaboration

Communicate

- There would be no Korat without an internal talk

Share

- Don’t be driven by quest to be the first author
- Be fair and share credit

MulSaw: Some lessons learned (2)

Persevere

- Some early criticism: static analysis (in particular shape analysis) can check the same properties
- Other “criticism”: Korat paper was first rejected
- There would be no Korat without a resubmission

The problem you solve can be more important than the solution you present

- Introducing a “new” problem can lead to many future contributions by many researchers
- We were asked “Is this a problem you created or is this really a problem?”

Outline

Overview

Example

Systematic testing using logical constraints

A few more recent projects

A bit of history

Some lessons learned at grad school

Conclusions

Conclusions

Logical constraints play a vital role in effective testing

- Can capture a rich class of input properties
- Allow tester's intuition to guide test generation

Systematic testing is effective at finding bugs in programs with complex inputs

- Non-isomorphic enumeration is key to feasibility

Constraints provide the basis of a unified approach to reliability, which applies before and after deployment

- More reliable software at a lower cost

khurshid@ece.utexas.edu
http://www.ece.utexas.edu/~khurshid