An Empirical Study of Structural Constraint Solving Techniques

Junaid Haroon Siddiqui Sarfraz Khurshid
University of Texas at Austin

ICFEM, Rio de Janeiro
9 December 2009

Overview

Structural constraint solving enables systematic analyses

- E.g., scope-bounded testing, symbolic/concolic execution

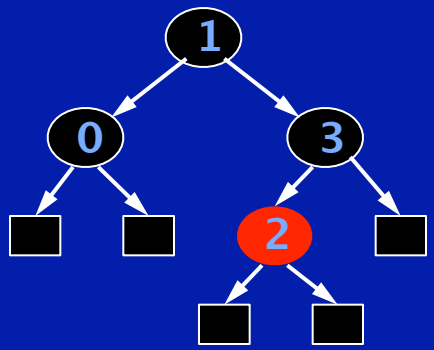
Several existing tools can solve structural constraints

We evaluate four such tools

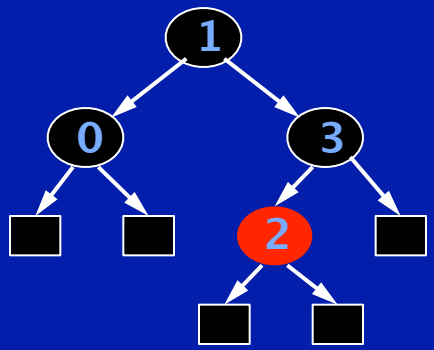
- Alloy tool-set – SAT-based analysis
- CUTE – concolic execution engine
- Java PathFinder (JPF) – model checker
- Korat – specialized solver

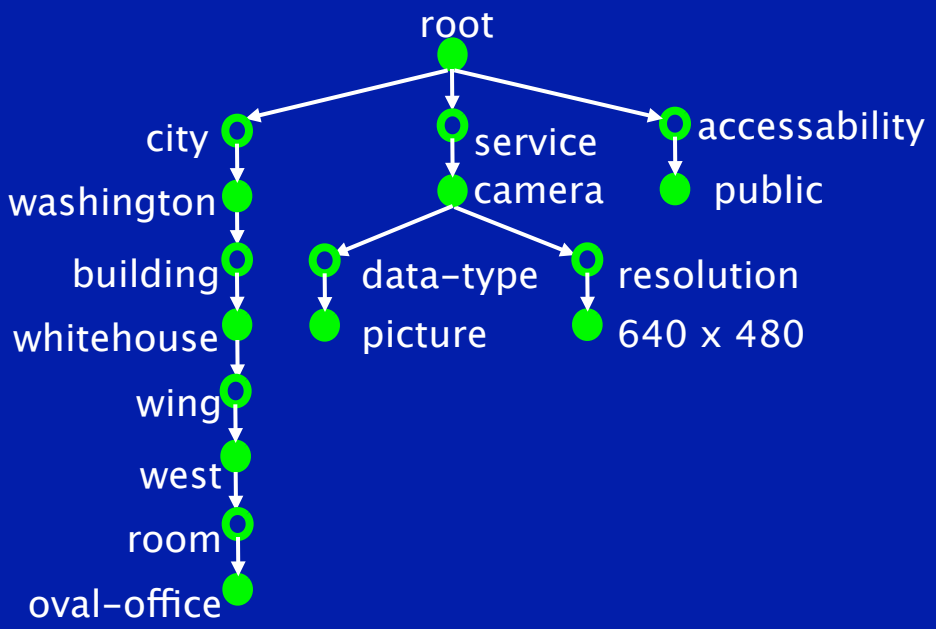
Examples of structural constraints

Examples of structural constraints

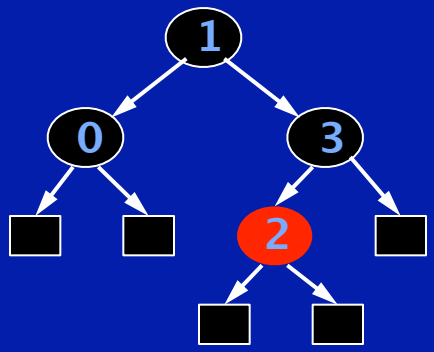


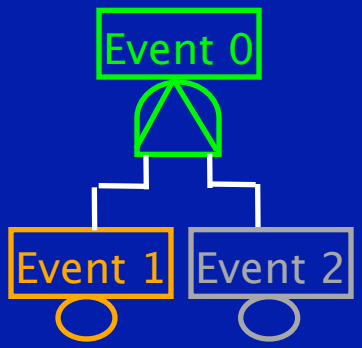
Examples of structural constraints

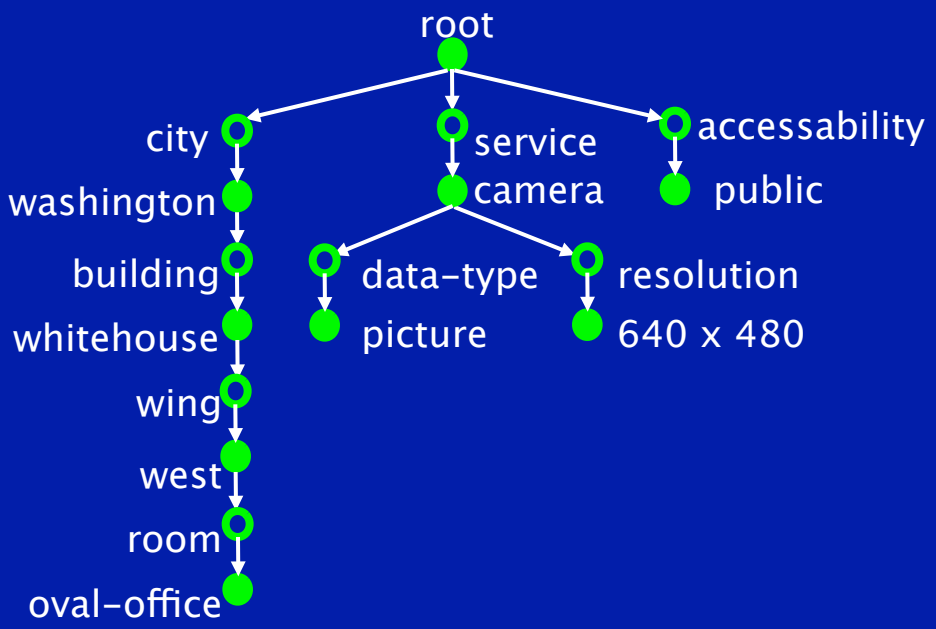




Examples of structural constraints

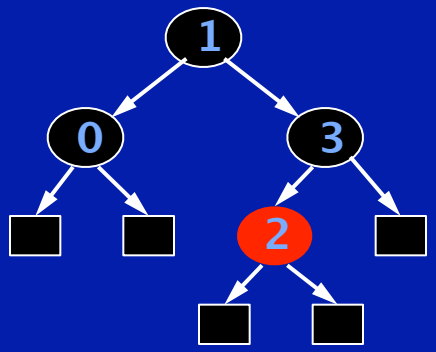


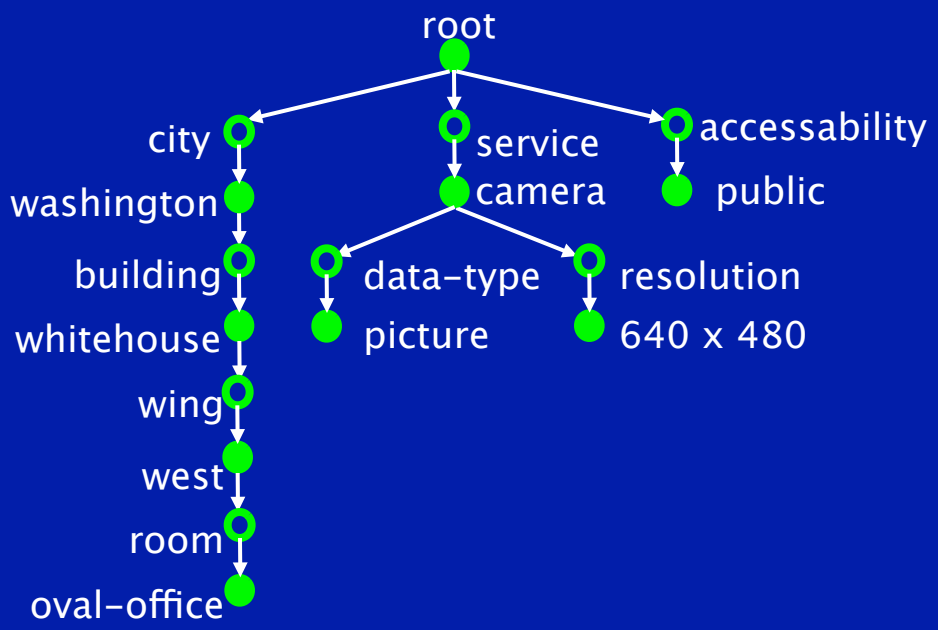


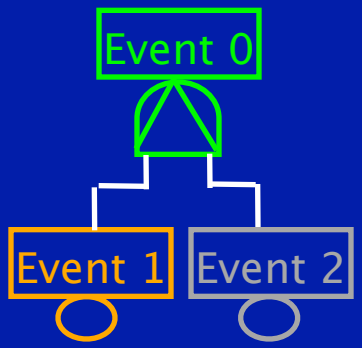



```
toplevel Event_0 ;
Event_0 pand Event_1 Event_2 ISeq_0 ISeq_1 FDep_0 FDep_1 ;
Event_1 be replication = 1 ;
Event_2 be replication = 1 ;
ISeq_0 seq Event_0 ;
ISeq_1 seq Event_1 ;
FDep_0 fdep trigger = Event_0 Event_1 ;
FDep_1 fdep trigger = Event_1 Event_2 ;
Event_1 dist=exponential rate=.0004 cov=0 res=.5 spt=.5
dorm=0 ;
Event_2 dist=exponential rate=.0004 cov=0 res=.5 spt=.5 dorm=.
```

Examples of structural constraints








```
toplevel Event_0 ;
Event_0 pand Event_1 Event_2 ISeq_0 ISeq_1 FDep_0 FDep_1 ;
Event_1 be replication = 1 ;
Event_2 be replication = 1 ;
ISeq_0 seq Event_0 ;
ISeq_1 seq Event_1 ;
FDep_0 fdep trigger = Event_0 Event_1 ;
FDep_1 fdep trigger = Event_1 Event_2 ;
Event_1 dist=exponential rate=.0004 cov=0 res=.5 spt=.5
dorm=0 ;
Event_2 dist=exponential rate=.0004 cov=0 res=.5 spt=.5 dorm=.
```

```
module meta_spec

sig Signature
sig Test


static sig S1 extends Test
static sig S0 extends
Signature

fun Main() {}
run Main for 3
```

Background: Scope-bounded testing

Tests against all inputs within a given bound on input size

- Inspired by model checking

To test a Java method:

- Use the method precondition to enumerate valid inputs
- Run the method on each input
- Check each output using the method postcondition

Example: binary search tree

```
class BinarySearchTree {
  Node root;
  int size;

  static class Node {
    int info;
    Node left, right;
  }


}
```

Example: binary search tree

```
class BinarySearchTree {
  Node root;
  int size;

  static class Node {
    int info;
    Node left, right;
  }

  void remove(int i) { … }
}
```

Example: binary search tree

```
class BinarySearchTree {
  Node root;
  int size;

  static class Node {
    int info;
    Node left, right;
  }

  void remove(int i) { … }
}
```

precondition: isTree() && isOrdered()

Example: binary search tree

```
class BinarySearchTree {
  Node root;
  int size;

  static class Node {
    int info;
    Node left, right;
  }

  void remove(int i) { … }
}
```

precondition: isTree() && isOrdered()
postcondition: isTree() && isOrdered()

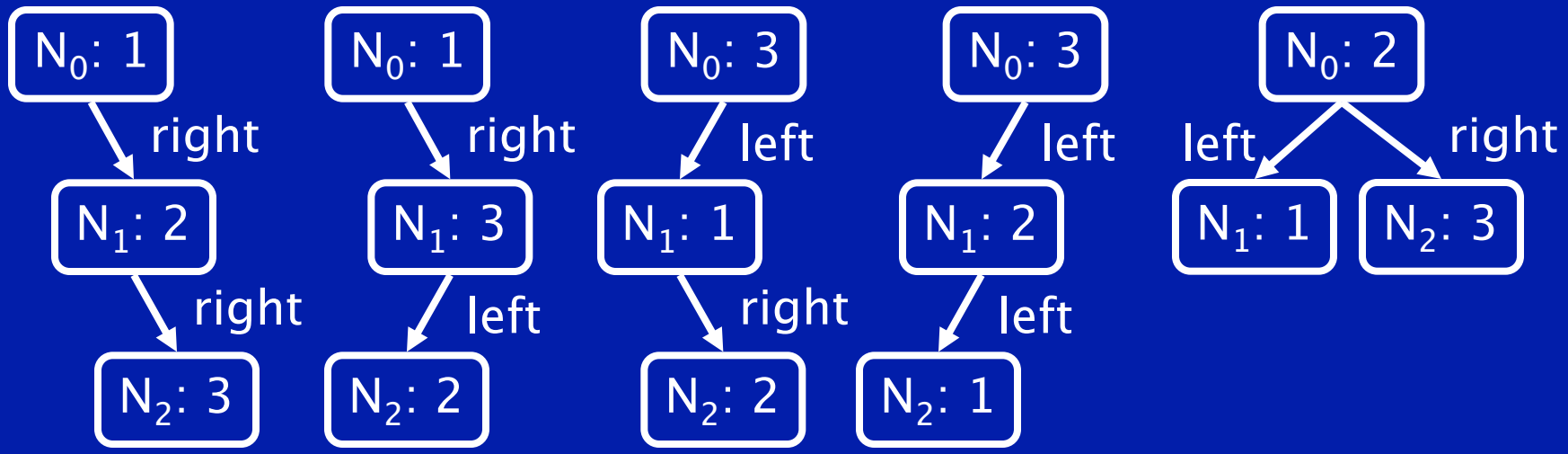
Example: binary search tree

```
class BinarySearchTree {
  Node root;
  int size;

  static class Node {
    int info;
    Node left, right;
  }

  void remove(int i) { … }
}
```

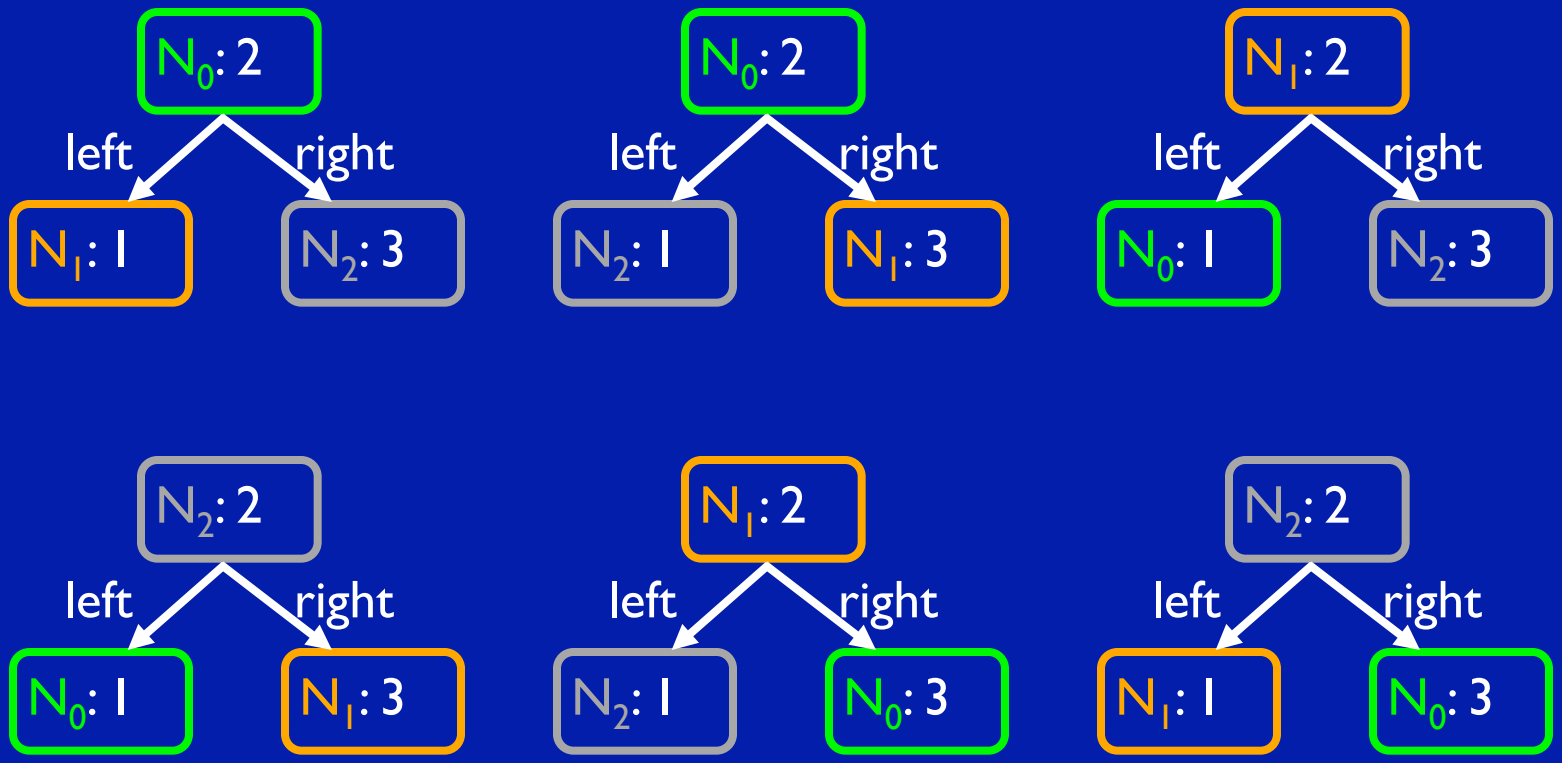
precondition: isTree() && isOrdered()

postcondition: isTree() && isOrdered() && “removes only i”

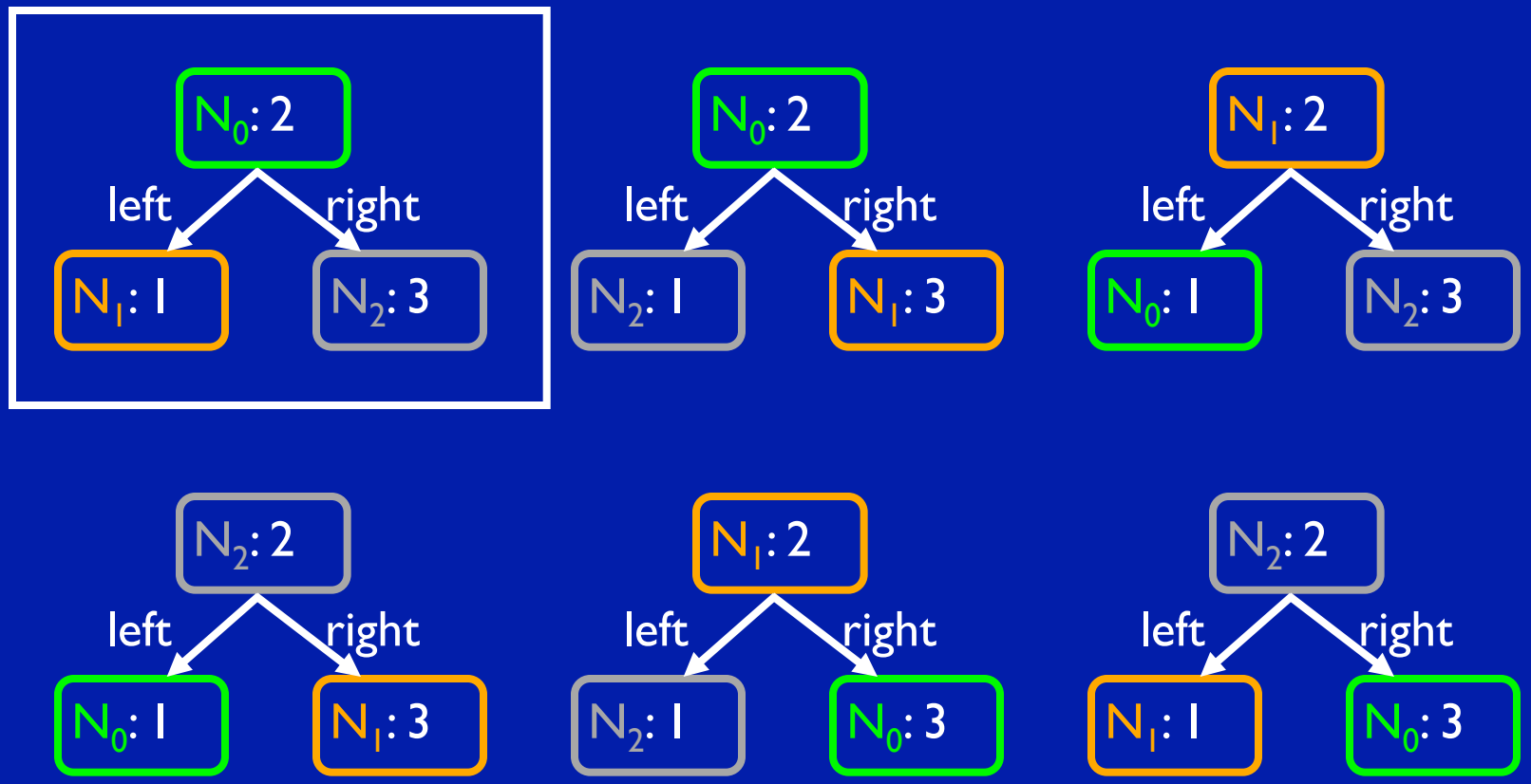
Example: non-isomorphic trees of size 3



Example: non-isomorphism



Example: non-isomorphism



Outline

Four tools

Research Questions

The Experiment

Results

Summary

Alloy tool-set [Jackson+2000]

Alloy is a first-order, relational language

The Alloy Analyzer is a fully automatic, SAT-based tool

- Kodkod model finder optimizes the analysis [Torlak +2007]

Originally motivated by checking of designs

Applications to code

- Static checking of code [Vaziri+2000]
- Systematic testing [Marinov+2001]

To run Alloy

- Iteratively set scope, run analysis, fix design/code, ...
  - Until solver times out

Java PathFinder (JPF) [Visser+2000]

General purpose model checker for Java programs

- Bounded depth search (iterative deepening)

Implements customized JVM to enable model checking

- Systematic thread interleaving
- Non-deterministic assignment

Originally designed for checking concurrent reactive systems

Generalized symbolic execution using lazy initialization allows
structural constraint solving [Khurshid+2003]

Korat [Boyapati+2002]

Korat is a tool for automated generation of structurally complex test inputs

Given a Java predicate, Korat enumerates inputs for which the predicate returns true

Korat performs a backtracking search of a bounded space of candidate inputs

- Prunes the space by monitoring field accesses

CUTE [Sen+2005]

Combines concrete execution with symbolic execution

- Supports pointers as well as primitives

Motivated by a basic limitation of symbolic execution:

- Complexity of solving and undecidability of path conditions

Uses concrete values to replace some symbolic values

- Reduces complexity of path conditions

Example: acyclicity constraint in Alloy

precondition

```
boolean isTree() {
    # root.*(left + right) = size      // consistency of size
    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 not the same node

boolean isOrdered() {                   // binary search
    …
}
```

Example: acyclicity constraint in Alloy

precondition

```
boolean isTree() {
   # root.*(left + right) = size      // consistency of size
   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 not the same node

boolean isOrdered() {                 // binary search
   …
}
```

Example: acyclicity constraint in Alloy

precondition

```
boolean isTree() {
    # root.*(left + right) = size      // consistency of size
    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 not the same node

boolean isOrdered() {                   // binary search
    …
}
```

Example: acyclicity constraint in Alloy

precondition

```
boolean isTree() {
    # root.*(left + right) = size      // consistency of size
    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 not the same node

boolean isOrdered() {                   // binary search
    …
}
```

Example: acyclicity constraint in Alloy

precondition

```
boolean isTree() {
    # root.*(left + right) = size      // consistency of size
    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 not the same node

boolean isOrdered() {                   // binary search
    …
}
```

Example: acyclicity constraint in Alloy

precondition

```
boolean isTree() {
    # root.*(left + right) = size      // consistency of size
    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 not the same node

boolean isOrdered() {                  // binary search
    …
}
```

Example: acyclicity constraint in Alloy

precondition

```
boolean isTree() {
    # root.*(left + right) = size      // consistency of size
    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 not the same node

boolean isOrdered() {                   // binary search
    …
}
```

Example: acyclicity constraint in Java

precondition

```
boolean isTree() {
  if (root == null) return size == 0; // empty tree has size 0
  Set visited = new HashSet(); visited.add(root);
  List 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; // acyclicity
       workList.add(current.left);
    }
    if (current.right != null) {
      if (!visited.add(current.right)) return false; // acyclicity
      workList.add(current.right);
    }
  }
  if (visited.size() != size) return false;          // consistency of size
  return true;
}
```

Outline

Four tools

Research Questions

The Experiment

Results

Summary

Research Questions

Writing constraints and Defining bounds

Output format

Performance for small sizes

Performance with complex constraints

Time complexity

The Experiment

Subjects of increasing complexity

- Binary Tree, Binary Search Tree, Red Black Tree
- Singly Linked List, Doubly Linked List, Sorted Linked List

Measured

- Quantitative: time and candidates generated
- Qualitative: Input (constraints and bounds) formatting and
  output (test cases) formatting

Outline

Four tools

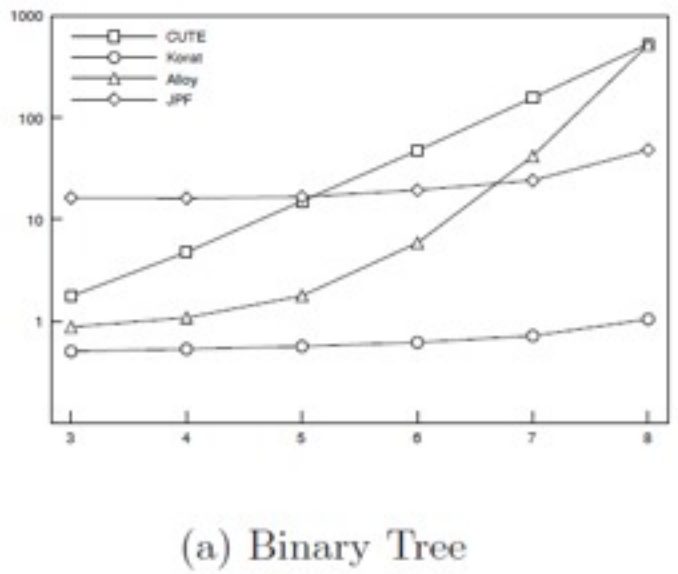
Research Questions

The Experiment

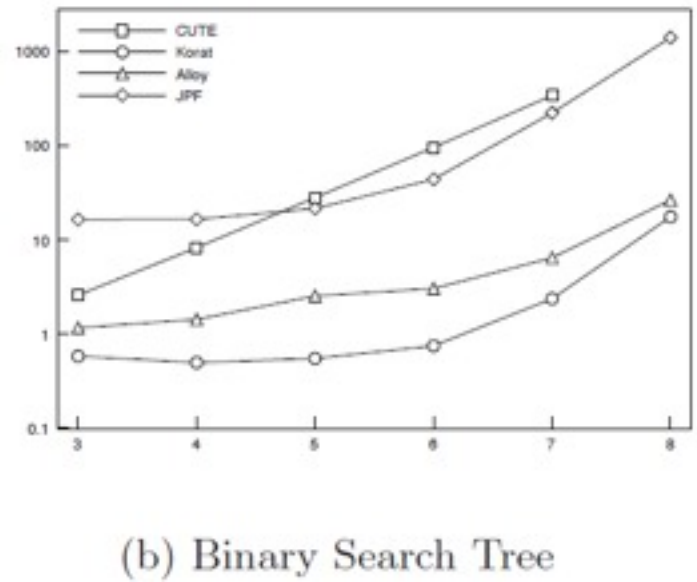
Results

Summary

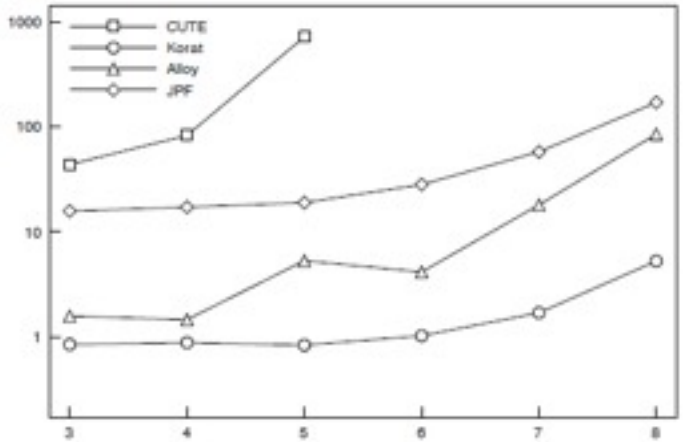
Results: Performance on trees



(a) Binary Tree

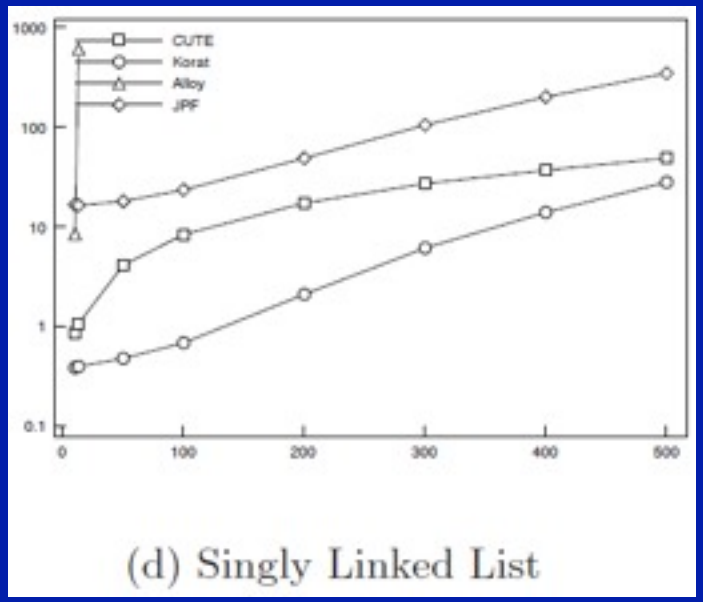


(b) Binary Search Tree

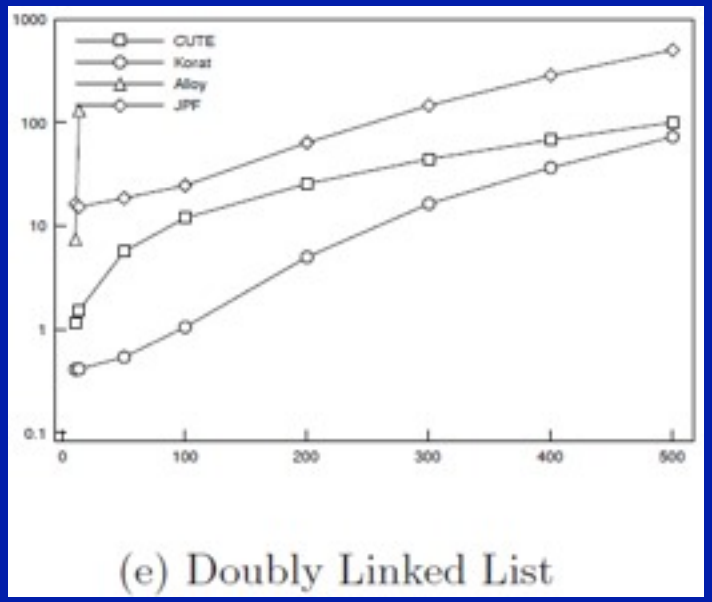


(c) Red Black Tree

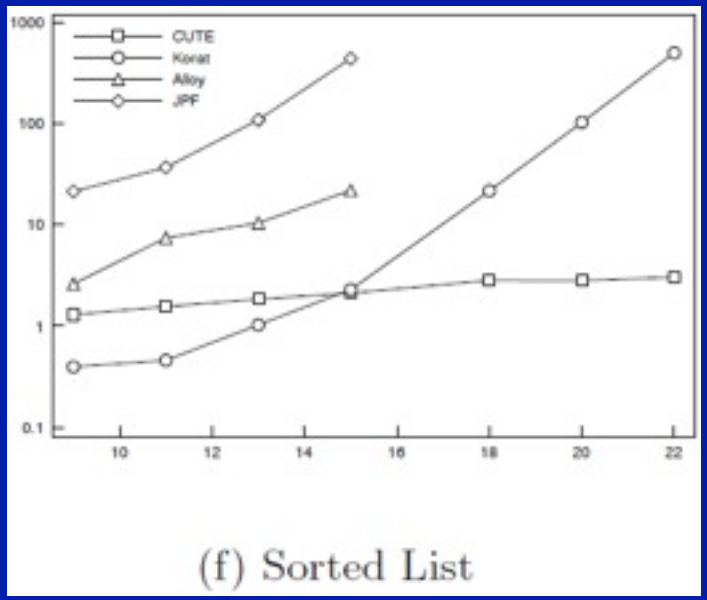
Results: Performance on lists



(d) Singly Linked List



(e) Doubly Linked List



(f) Sorted List

Results: Isomorphism Handling

Subject	CUTE	Korat	Alloy	JPF
Binary Tree	YES	YES	**NO**	YES
Binary Search Tree	YES	YES	YES	YES
Red Black Tree	**NO**	YES	YES	YES
Singly Linked List	YES	YES	**NO**	YES
Doubly Linked List	YES	YES	**NO**	YES
Sorted List	YES	YES	YES	YES

Results: Constraints, Bounds and Output Format

Alloy uses declarative constraints while all other tools use imperative constraints

Korat and Alloy support the most direct bounds specification by design

Alloy results need to be translated into actual structures

- Translation time is insignificant compared to solving time

Results: Summary

Fastest tool for small sizes is Korat

Lazy initialization with JPF is effectively a slower Korat

Declarative constraints are the most concise

CUTE provides better time complexity

CUTE requires minor tweaking of predicates to work

Korat produces no isomorphic solutions

Results: Summary

Fastest tool for small sizes is Korat

Lazy initialization with JPF is effectively a slower Korat

Declarative constraints are the most concise

CUTE provides better time complexity

CUTE requires minor tweaking of predicates to work

Korat produces no isomorphic solutions

jsiddiqui | khurshid@ece.utexas.edu