Symbolic Execution of Alloy Models

Junaid Haroon Siddiqui and Sarfraz Khurshid
University of Texas at Austin

ICFEM
Durham, UK
27.10.11

Overview

Alloy is a first-order, relational language for writing declarative models [Jackson+2000]

- Alloy tool-set uses SAT for automatic analysis
  - "Execution" of Alloy models: generation of satisfying instances w.r.t. constraints in model

Symbolic execution is a well-known, path-based analysis technique for imperative programs

Our insight: solving engine of SAT can enable symbolic execution of Alloy models

This paper presents our technique and demonstrates

Outline

Overview

Background

- Alloy
- Symbolic execution

Symbolic execution of Alloy models

Related work

Conclusions

Alloy language

Designed by Daniel Jackson's group at MIT

- "Analyzable models for software design"

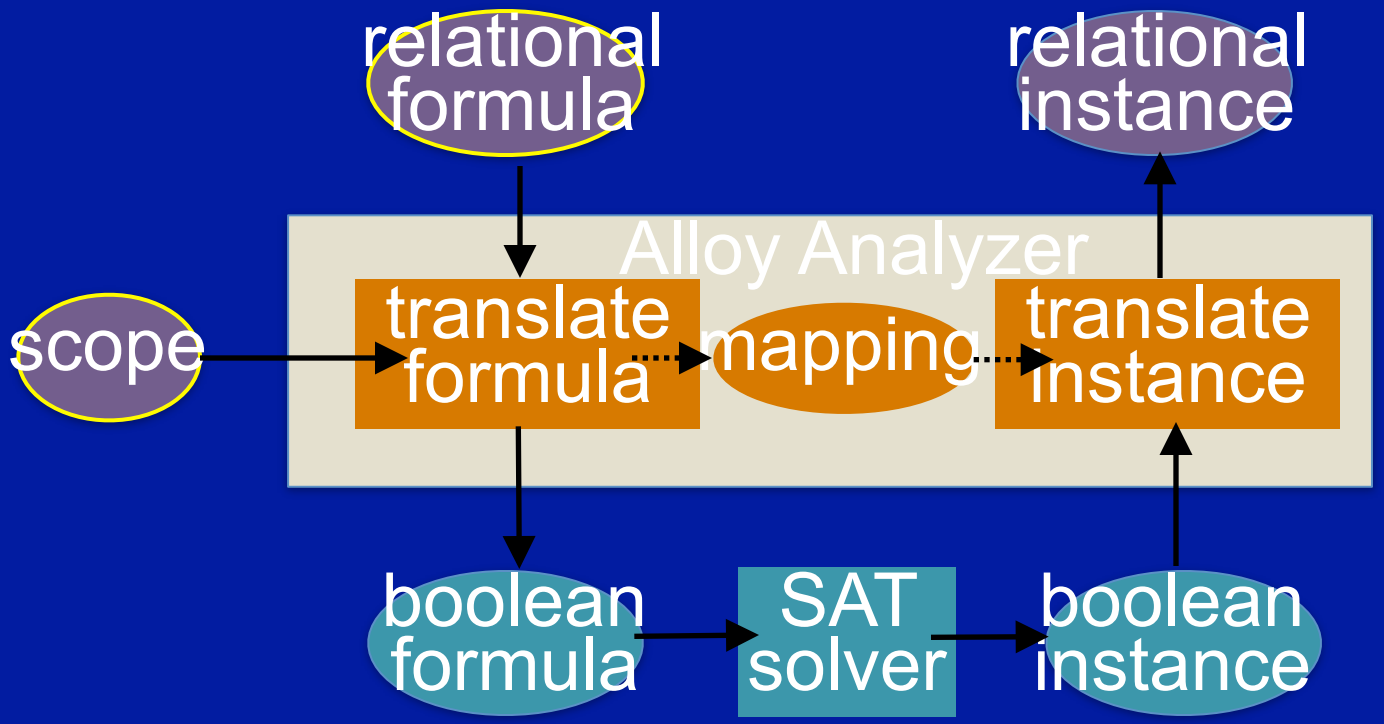
Suitable for software modeling

- Conceptually similar to OCL for UML
- First-order logic formulas with transitive closure
- Declarative, relational

Fully automatic analysis tool: Alloy Analyzer

- Performs scope-bounded analysis
- Compiles to SAT

Alloy Analyzer



Symbolic execution

Technique for executing a program on symbolic input values [Clarke'76, King'76]

Performs a systematic exploration of program paths

- For each (bounded) path, builds a path condition and checks its satisfiability

Traditionally used for test generation and verification

- Originally developed for code with primitive types
- More recently generalized to arbitrary

Concrete execution path (example)

```
int x, y; // inputs

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```

Concrete execution path (example)

```
int x, y; // inputs        x = 1, y = 0

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```

Concrete execution path (example)

```
int x, y; // inputs        x = 1, y = 0
if (x > y) {                1 >? 0
    x = x + y;
    y = x – y;
    x = x – y;
    if (x – y > 0)
        assert(false);
}
```

Concrete execution path (example)

```
int x, y; // inputs        x = 1, y = 0
if (x > y) {                1 >? 0
    x = x + y;                  x = 1 + 0 = 1
    y = x – y;                  y = 1 – 0 = 1
    x = x – y;                  x = 1 – 1 = 0
    if (x – y > 0)
        assert(false);
}
```

Concrete execution path (example)

```
int x, y; // inputs        x = 1, y = 0
if (x > y) {                1 >? 0
    x = x + y;                  x = 1 + 0 = 1
    y = x – y;                  y = 1 – 0 = 1
    x = x – y;                  x = 1 – 1 = 0
    if (x – y > 0)              0 – 1 >? 0
        assert(false);
}
```

Symbolic execution tree* (example)

```
int x, y; // inputs

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```

*Assuming no arithmetic overflow

Symbolic execution tree* (example)

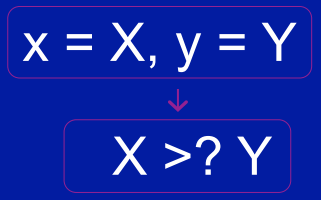
```
int x, y; // inputs

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```

x = X, y = Y

*Assuming no arithmetic overflow

Symbolic execution tree* (example)

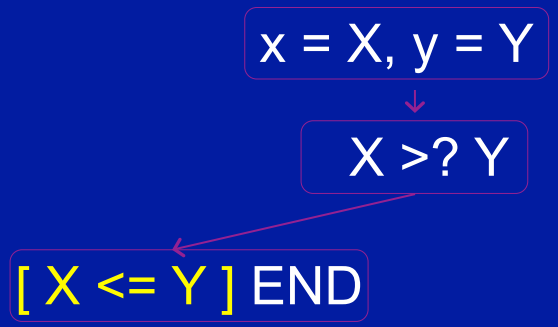
```
int x, y; // inputs

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```



*Assuming no arithmetic overflow

Symbolic execution tree* (example)

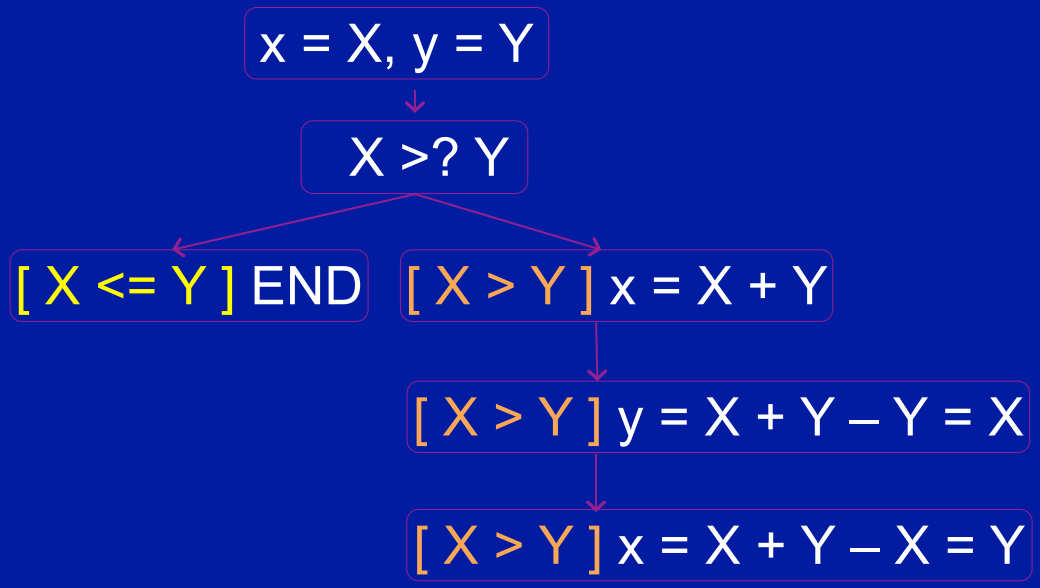
```
int x, y; // inputs

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```



*Assuming no arithmetic overflow

Symbolic execution tree* (example)

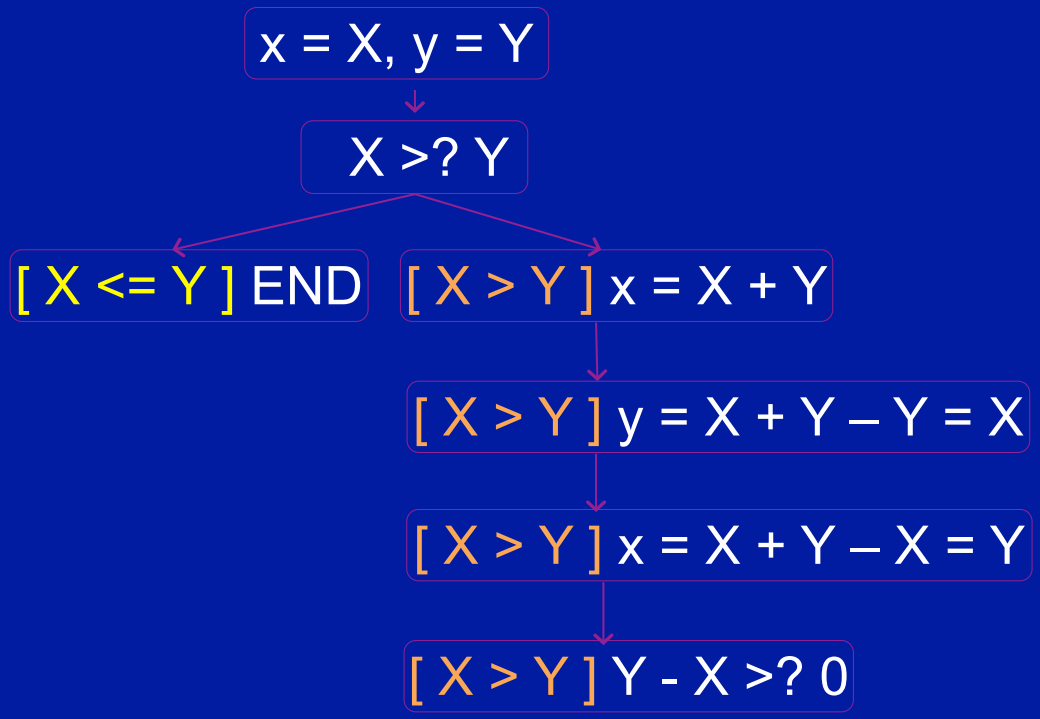
```
int x, y; // inputs

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```



*Assuming no arithmetic overflow

Symbolic execution tree* (example)

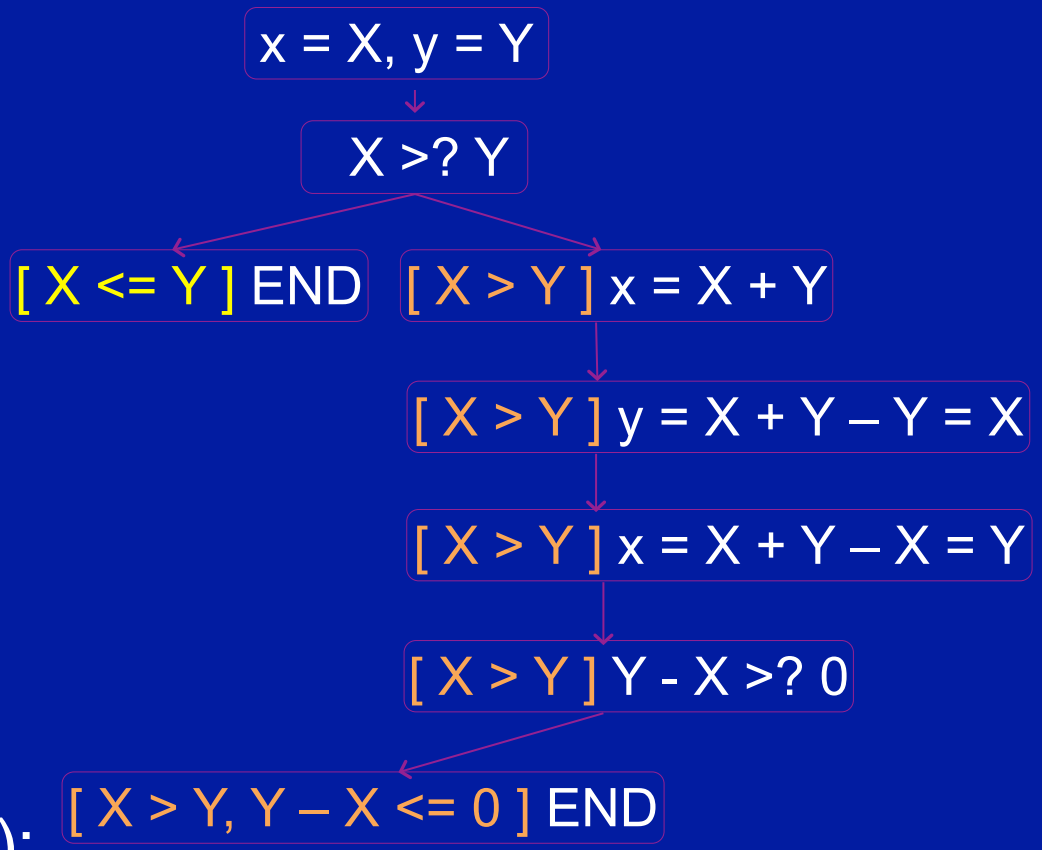
```
int x, y; // inputs

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```



*Assuming no arithmetic overflow

Symbolic execution tree* (example)

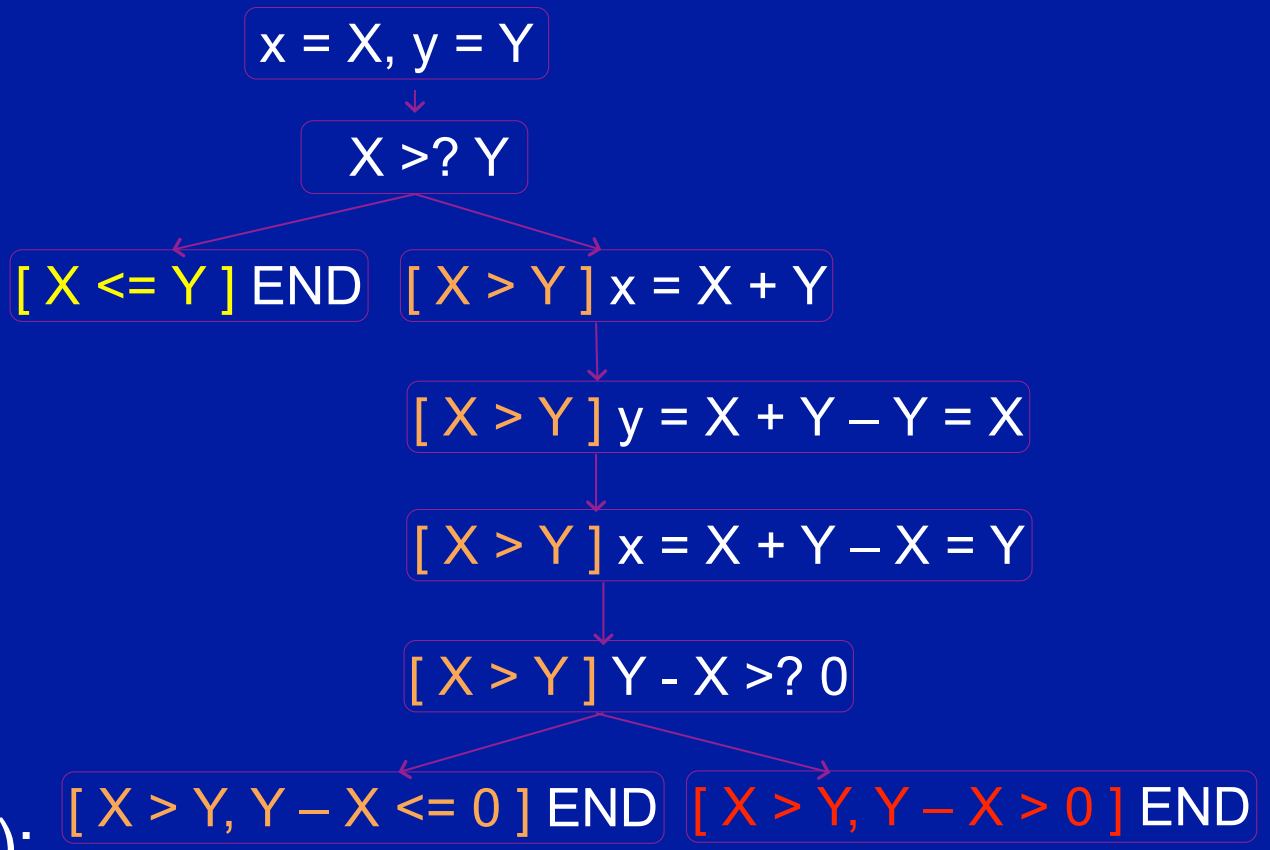
```
int x, y; // inputs

if (x > y) {
    x = x + y;
    y = x – y;
    x = x – y;
    if (x – y > 0)
        assert(false);
}
```



*Assuming no arithmetic overflow

Symbolic execution tree* (example)

```
int x, y; // inputs

if (x > y) {

    x = x + y;

    y = x – y;

    x = x – y;

    if (x – y > 0)

        assert(false);

}
```



*Assuming no arithmetic overflow

Outline

Overview

Background

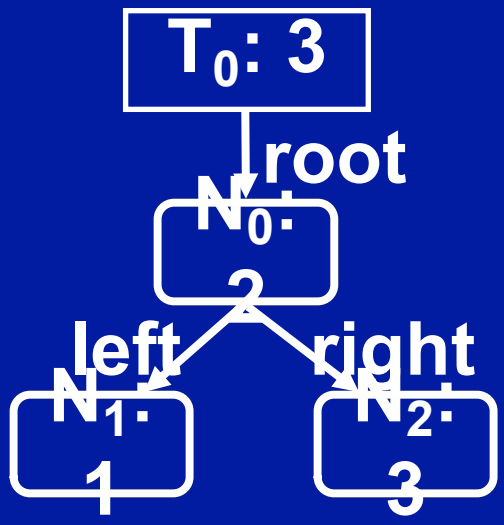
Symbolic execution of Alloy models

- Example
- Approach
- Demonstration

Related work

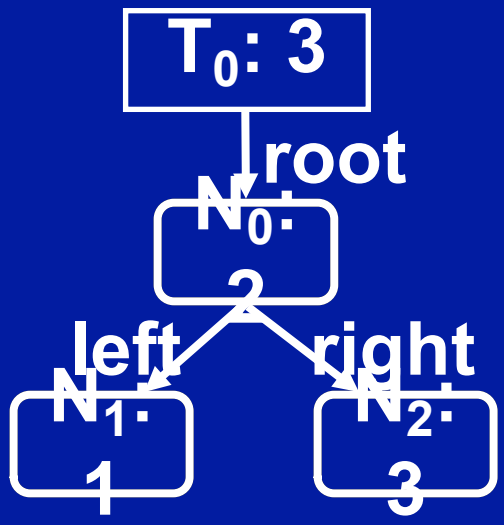
Conclusions

Example: binary search trees



Example: binary search trees

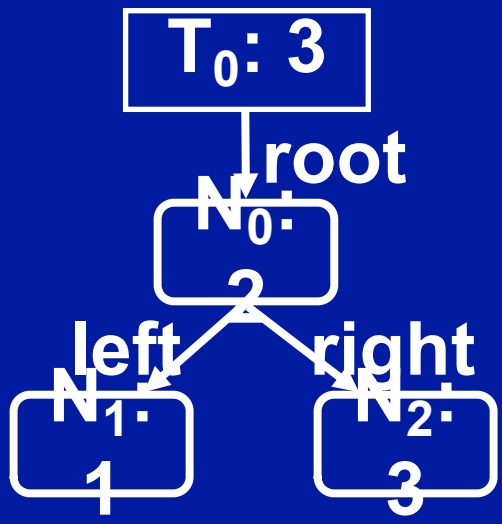
```
sig Tree {
  root: lone Node,
  size: Int
}
sig Node {
  left: lone Node,
  right: lone Node,
  element: Int
}
```



Example: binary search trees

```
sig Tree {
  root: lone Node,
  size: Int
}
sig Node {
  left: lone Node,
  right: lone Node,
  element: Int
}
```

```
pred repOk(t: Tree) {
  isTree[t] and search[t] }
pred isTree(t: Tree) {
  // consistency of size
  # t.root.*(left + right) = t.size
  all n: root.*(left + right) {
    n !in n.^(left + right) // no directed cycles
    lone n.~(left + right) // at most one parent
    no n.left & n.right  // left and right child not the same
  } }
pred search(t: Tree) { ... }

run repOk
```



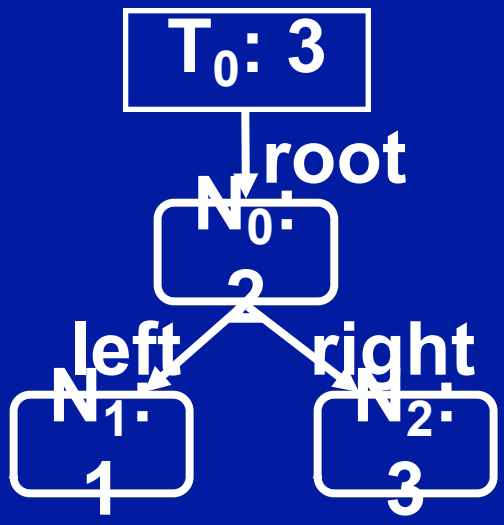
Example: concrete solving (execution)

Concrete instance generated by SAT

Tree = { T0 }
Node = { N0, N1, N2 }
Int = { 1, 2, 3 }

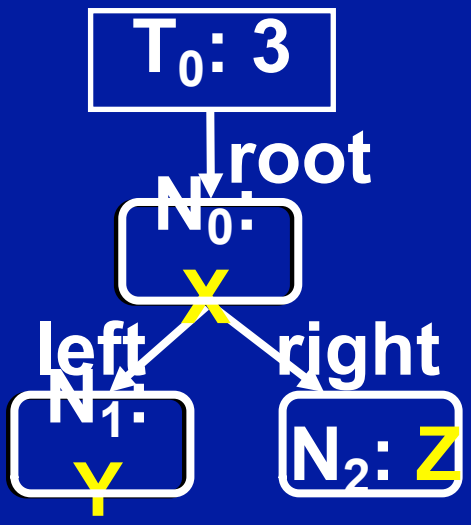
root = { (BST0, N0) }
size = { (BST0, 3) }

element = { (N0, 2), (N1, 1), (N2, 3) }
left = { (N0, N1) }
right = { (N0, N2) }



Example: symbolic solving (execution)

Symbolic instance generated by SAT



Tree = { T0 }
Node = { N0, N1, N2 }
Int = { 1, 2, 3 }

root = { (BST0, N0) }
size = { (BST0, 3) }

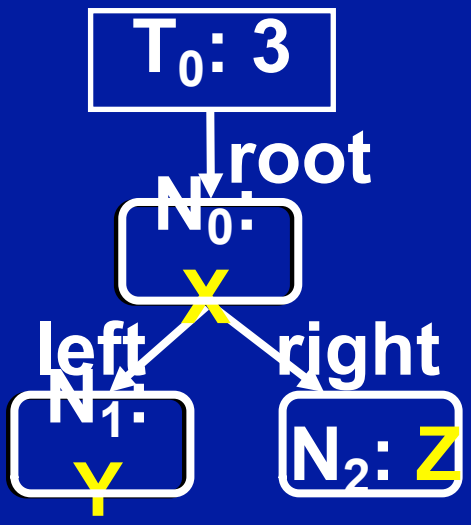
element = { (N0, X), (N1, Y), (N2, Z) }
left = { (N0, N1) }
right = { (N0, N2) }

Clause = { C0, C1 }

C0 = “Y < X”, C1 = “X < Z”

Example: symbolic solving (execution)

Symbolic instance generated by SAT



Tree = { T0 }
Node = { N0, N1, N2 }
Int = { 1, 2, 3 }

root = { (BST0, N0) }
size = { (BST0, 3) }

element = { (N0, X), (N1, Y), (N2, Z) }
left = { (N0, N1) }
right = { (N0, N2) }

Clause = { C0, C1 }
C0 = “Y < X”, C1 = “X < Z”

Instance condition

Overview of our approach

Our approach has four key elements:

1. Introducing symbolic values in Alloy
   - Library modules
2. Declaring symbolic values in an Alloy model and bounding the size of the instance condition
3. Adding constraints to ensure consistency
4. Techniques for avoiding redundancy in symbolic solution
   - Iterative deepening and skolemization

Approach: library support

Library module “symbolic” contains signatures and predicates to enable symbolic execution for Alloy

- `SymbolicInt` and `SymbolicBool` sigs
- Operators
  - Comparison: `lt`, `gt`, `lte`, `gte`, `eq`, `neq`
  - Arithmetic: `plus`, `minus`
- Binary expressions: clauses and arithmetic
- Predicates to represent binary operations

```
pred lt(e1: Expr+Int, e2: Expr+Int) {
  some c: Clause | c.LHS = e1 and c.OP = LT and c.RHS = e2 }
```

Approach: library usage and consistency

Library usage has three key steps:

1. Declare fields that have symbolic values, e.g.,
   "sig Node { element: SymbolicInt, ... }"
2. Use library predicates in expressions, e.g.,
   "pred p(...) { ...
   n.left.element.lt[n.element] ... }"
   - This style of using predicates is already used in Alloy 4.2 RC – for concrete values
3. Define scope for binary expressions

To ensure consistency of symbolic execution, new constraints are mechanically introduced

Approach: avoiding redundant clauses

Alloy analysis requires a scope, so we need to provide scopes for the library sigs used

Determining an appropriate scope is important

- Instance condition can contain redundancy if scope permits more expressions than necessary

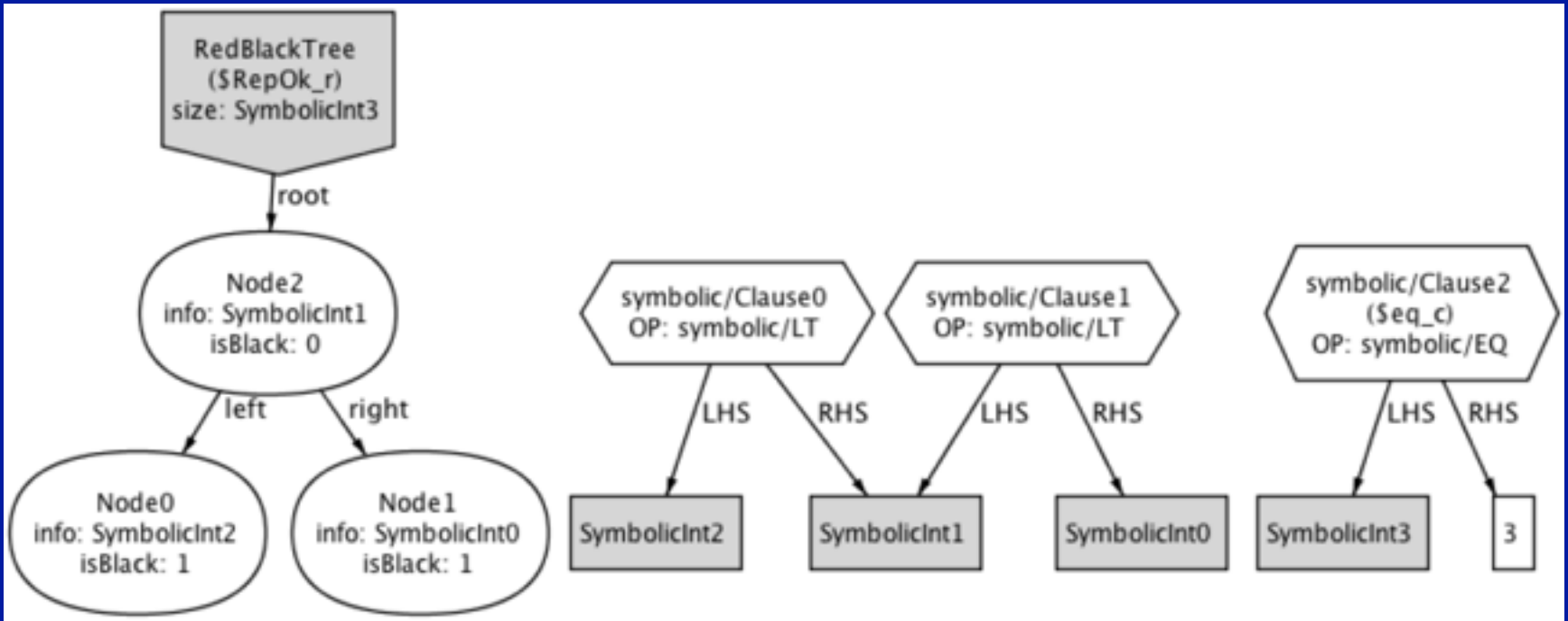
We present two techniques for determining scope

- Iterative deepening – the smallest number of expressions that give a solution is used
- Skolemization – the user predicates are written using skolem constants

Demonstration: red-black trees

```
open symbolic
sig RedBlackTree { root: lone Node, size: SymblicInt }
sig Node { left, right: lone Node, isBlack: Bool, info: SymbolicInt }
…
pred isOrdered(r: RedBlackTree) {
  all n: r.root.*(left+right) { -- binary search order
    some n.left => (n.left.info).lt[n.info]
    some n.right => (n.info).lt[n.right.info] }}
pred isColorOk(r: RedBlackTree) {
  all e: root.*(left + right) | -- red nodes have black children
    e.isBlack = false && some e.left + e.right => (e.left + e.right).isBlack =
     true
  ... }
```

Demonstration: symbolic Alloy instance



Outline

Overview

Background

Symbolic execution of Alloy models

Related work

Conclusions

Related work

Forward symbolic execution for imperative programs

- Pioneered 3 decades ago [Clarke'76, King'76]
- Supported by several current tools, e.g., PREfix, JPF_{SE}, DART, CUTE, KLEE, and Pex

Symbolic execution for other kinds of programs

- Temporal logic and statecharts [ThumsBalser'04]
- Live sequence charts [Wang+'04]

Alloy tool-set and static analysis (concrete values)

- JAlloy [Vaziri'04] and J/Forge [Dennis'09] provide

Conclusions

This paper introduces the idea of symbolic execution
of declarative models written in Alloy

- Library modules for symbolic integers and booleans enable SAT-based analysis

Symbolic execution of Alloy models is useful

- Combining dedicated solvers effectively
- Inspecting representative instances efficiently

Future work

- Further development/evaluation of our approach

Questions/Comments

{jsiddiqui,khurshid}@ece.utexas.edu