Systematic Software Analysis Using SAT

Sarfraz Khurshid
University of Texas at Austin
khurshid@utexas.edu

SAT/SMT/AR Summer School
Lisbon, Portugal
July 5, 2019

Overview

SAT solvers have many uses, e.g., model finding, model enumeration, and model counting

This lecture focuses on **model enumeration**

It has many applications in software (and hardware) engineering

- **Testing**: create high quality test suites
- Analysis: illustrate different counterexamples
- Synthesis: create alternative implementations
- Repair: create alternative fixes

An application of enumeration

Systematic testing of code using **specs** [ASE’01]

- Idea: create **all** “small” inputs, and test against them
    - High quality suites with non-equivalent inputs
        - **Symmetry breaking** [SAT’03]
- Enabling technology: Alloy tool-set [Jackson-FSE’00]
    - Alloy: relational first-order logic + transitive closure
    - Alloy analyzer: SAT-based tool for automatic analysis
    - http://alloy.mit.edu

Outline

Basics of software testing

- Focus: programs with structurally complex inputs

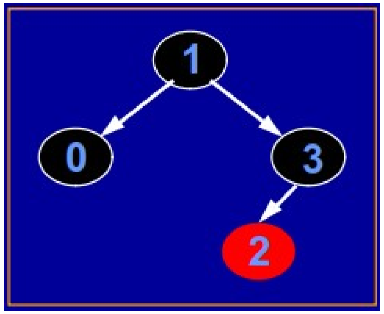
Basics of Alloy

Basics of systematic testing

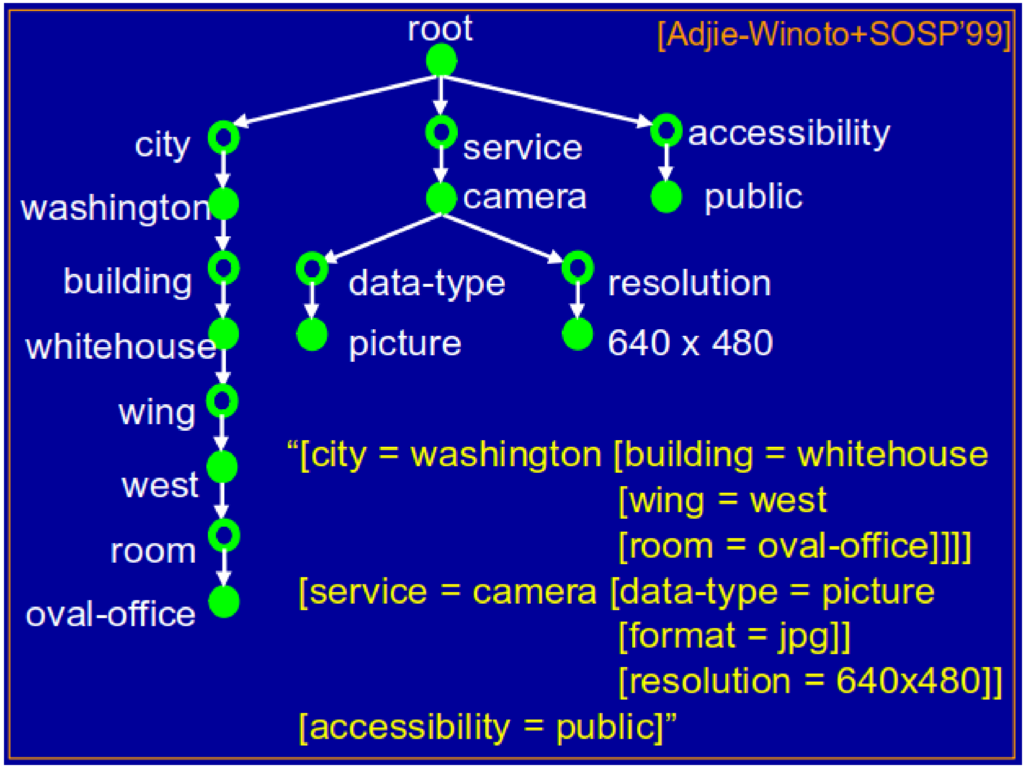
- Create non-equivalent tests using symmetry breaking

Conclusions

Structurally complex data



```
.ClassName4{
        -webkit-transform: rotateY( 180deg );}
.ClassName12{
        -webkit-perspective: 800;
        -webkit-backface-visibility: hidden;}
```



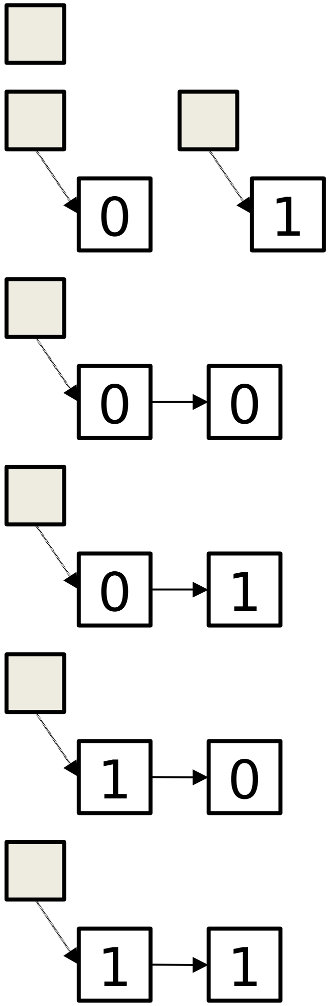
[Adjie-Winoto+SOSP'99]

```
"[city = washington [building = whitehouse
                     [wing = west
                     [room = oval-office]]]]
 [service = camera [data-type = picture
                     [format = jpg]]
                     [resolution = 640x480]]
 [accessibility = public]"
```

```
<html>
<head>
<link rel="stylesheet"
type="text/css"  href="file.css">
</link></head>
<body>
<div class="ClassName4">
<h1>This is some text
<div class="ClassName12">
<h1>This is some text</h1>
</div></h1></div></body></html>
```

Acyclic singly-linked list

```
class SLList {
    // class invariant: acyclic and size-okay
    Node header;
    int size;

    static class Node {
        int elem;
        Node next; }

    void add(int x) {
        // pre-cond: class invariant (this)
        // post-cond: class invariant (this)
        //          and x is added at the head

        Node n = new Node();
        n.elem = x;
        n.next = header;
        header = n;
        size++; }

    void remove(int x) { /*... */ }
```



How to create an input list?

Write a test (by hand)

Two basic ways: at *abstract* level or at *concrete* level

```
@Test public void abst() {
    // create receiver object state
    SLList l = new SLList();
    l.add(0);



    // execute method to test
    l.remove(1);

    // (partially) check output
    assertEquals(0, l.header.elem);
}
```

```
@Test public void conc() {
    // create receiver object state
    SLList l = new SLList();
    Node n0 = new Node();
    l.header = n0; l.size = 1;
    n0.elem = 0; n0.next = null;

    // execute method to test
    l.remove(1);

    // (partially) check output
    assertEquals(0, l.header.elem);
}
```

How to create many lists?

Can write a test generator (by hand)

Can automate using **non-deterministic choice**, e.g., with the Java PathFinder [https://github.com/javapathfinder]

```
static List abstractGen() {
    List l = new List();
    int length = Verify.getInt(0, 2);
    for (int i = 0; i < length; i++) {
        boolean method = Verify.getBoolean();
        int arg = Verify.getInt(0, 1);
        if (method) {
            l.add(arg);
        } else {
            l.remove(arg);
        }
    }
    return l; }
```

Abstract-level generation

Advantage: simple to automate

Disadvantage:

- Hard to test partial implementations
  - To test *remove*, must implement *add* first
- Hard to avoid equivalent tests
  - E.g., naive exploration creates 21 method sequences:

    ε, “add(0)”, “add(1)”, “remove(0)”, “remove(1)”,

    “add(0); add(0)”, “add(0); add(1)”, “add(0); remove(0)”, “add(0); remove(1)”, …

How to create many lists – at the concrete level?

Again, can write a test generator (by hand), or automate using non-deterministic choice

Advantage: efficient, high quality test generation

Disadvantage:

- Different structures require different generators
- Writing the generators can be hard
  - No textbook methods
  - Cannot simply sample at random: *#valid/#all* → 0
- Generators need to account for symmetry breaking

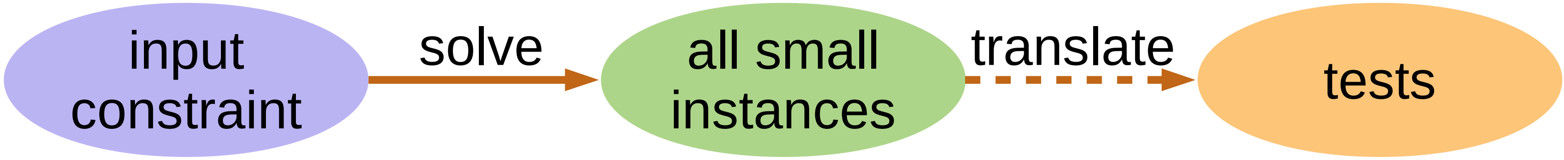
Idea: use logical constraints and model enumeration!

Constraint-based generation

Observe: each input must be a **valid** structure

- **Acyclic**, singly-linked list

Approach: characterize validity properties as logical constraints, and solve them [ASE'01]



input constraint

solve

all small instances

translate

tests

Two key questions:

- How to write the constraints?
- How to solve the constraints to find one, many, or all solutions?

How to write constraints?

Use a declarative language, e.g., Alloy

```
pred Acyclic(l: List) {
  all n: l.header.*link | n !in n.^link
}
```

Use an **imperative** language, e.g., Java

```
boolean repOk() {
    if (header == null) return size == 0;
    Set<Node> visited = new HashSet<Node>();
    Node current = header;
    while (current != null) {
        if (!visited.add(current)) return false;
        current = current.next;
    }
    return size == visited.size();
}
```

How to solve constraints?

For Alloy, its analyzer provides fully automatic solving using off-the-shelf SAT technology

- Kodkod back-end [TorlakJackson-TACAS’07]
- Supports several SAT solvers

For Java, there are four basic approaches:

- Translate to SAT, a la bounded model checking [Biere+TACAS’99, JacksonVaziri-ISSTA’00]
- Use symbolic execution [King-CACM’76, TACAS’03]
    - For each path that returns true, create input(s)
- Filter (naively) all candidates using *repOk*
- Use a dedicated solver for Java, e.g., Korat [ISSTA’02]

Non-det. choice and filtering

```
static void concreteGen() {
    // allocate objects
    SLList l = new SLList();
    Node n1 = new Node();
    Node n2 = new Node();

    // build domain(s)
    Node[] nodes = new Node[]{ null, n1, n2 };

    // initialize fields
    l.header = nodes[Verify.getInt(0, nodes.length - 1)];
    l.size = Verify.getInt(0, 2);
    n1.elem = Verify.getInt(0, 1);
    n1.next = nodes[Verify.getInt(0, nodes.length - 1)];
    n2.elem = Verify.getInt(0, 1);
    n2.next = nodes[Verify.getInt(0, nodes.length - 1)];

    // check validity
    if (l.repOk()) {
        // output list
    }
}
```

Solving imperative constraints

repOk is a logical constraint written in an imperative language, hence termed *imperative constraint*

Solving *repOk* using naive filtering is infeasible

- Checks every candidate in the state space (e.g., 324)
- Creates too many solutions that are redundant
  - E.g., 68 valid lists (instead of 7 that we expect)

However, *repOk* can be used to **prune** the search and make it feasible [ISSTA'02]

- Korat prunes **and** checks only **non-isomomorphic** candidates (e.g., 31)
  - Creates non-equivalent solutions (e.g., 7 valid lists)

Alloy

Alloy demo

An Alloy specification

```
module list

one sig List { // set of list atoms
  header: lone Node } // header: List x Node

sig Node { // set of node atoms
  link: lone Node } // link: Node x Node

pred RepOk(l: List) { all n: l.header.*link | n !in n.^link }
```

Alloy: simulation

```
module list

one sig List { // set of list atoms
  header: lone Node } // header: List x Node

sig Node { // set of node atoms
  link: lone Node } // link: Node x Node

pred RepOk(l: List) { all n: l.header.*link | n !in n.^link }

run RepOk // default scope is 3

fact Reachability { List.header.*link = Node }
```

Alloy: checking

```
sig List {  header: lone Node }

sig Node { link: lone Node }

pred RepOk(l: List) { all n: l.header.*link | n !in n.^link }

pred RepOk2(l: List) {
  no l.header or some n: l.header.*link | no n.link }

assert Equivalence {
  all l: List | RepOk[l] <=> RepOk2[l] }

check Equivalence // for 1, 2, 3, 4, 5, 6, ...
```

Symmetry breaking (SB)

Alloy adapts Crawford's symmetry breaking predicates to remove some, but not all, symmetries [Shlyakhter-SAT'01]

We can remove all symmetries – for a class of structures – by writing additional constraints **in Alloy** [SAT'03]

- For example:
  - Define a linear order on nodes
  - Add constraints to define a "traversal" and require the nodes to be "visited" w.r.t. the linear order

Full symmetry breaking: lists

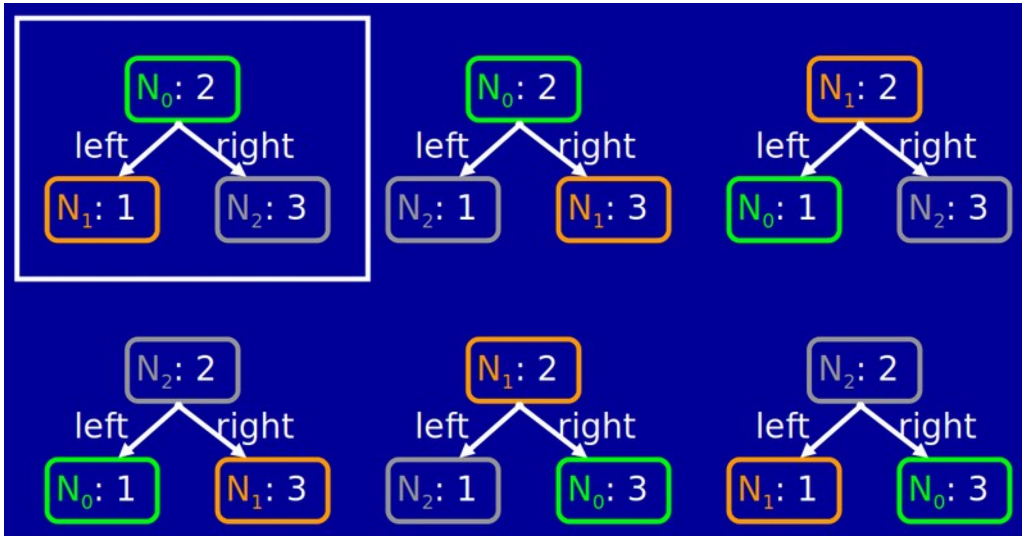
```
open util/ordering[Node]
module list

one sig List { header: lone Node }

sig Node {link: lone Node }

pred RepOk(l: List) { all n: l.header.*link | n !in n.^link }

fact SymmetryBreaking {
  List.header in first[]
  all n: List.header.*link | n.link in next[n]
}
```

Full symmetry breaking: binary search trees

```
fact SymmetryBreaking { // pre-order
  Tree.root in first[]
  all n: Tree.root.*(left + right) {
    some n.left implies n.left in next[n]
    no n.left implies n.right in next[n]
    some n.right and some n.left implies
      n.right in next[max[n.left.*(left + right)]]
  }
}
```

Full symmetry breaking: illustration

For exactly 3 nodes (and integer keys {1, 2, 3}), there are 3! = 6 trees in each isomorphism class, e.g.,



- Each permutation of node identities (N_0, N_1, N_2) gives an isomorphic tree

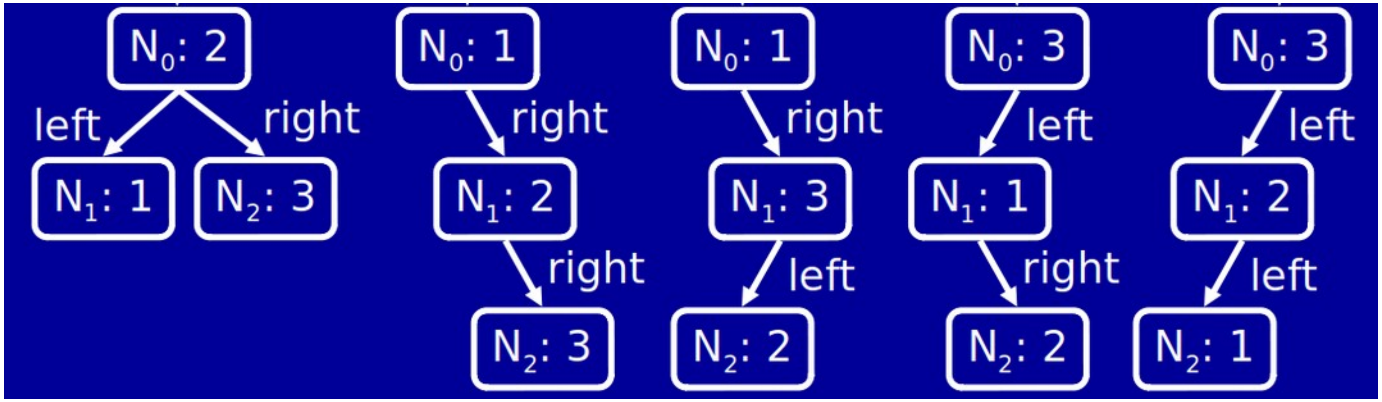
With the *SymmetryBreaking* fact only 1 tree (that respects the pre-order traversal constraint) per class is generated

Importance of symmetry breaking

With no symmetry breaking the number of solutions goes up by a factor that is exponential in the number of nodes

- Also, the solver can suffer a substantial slowdown

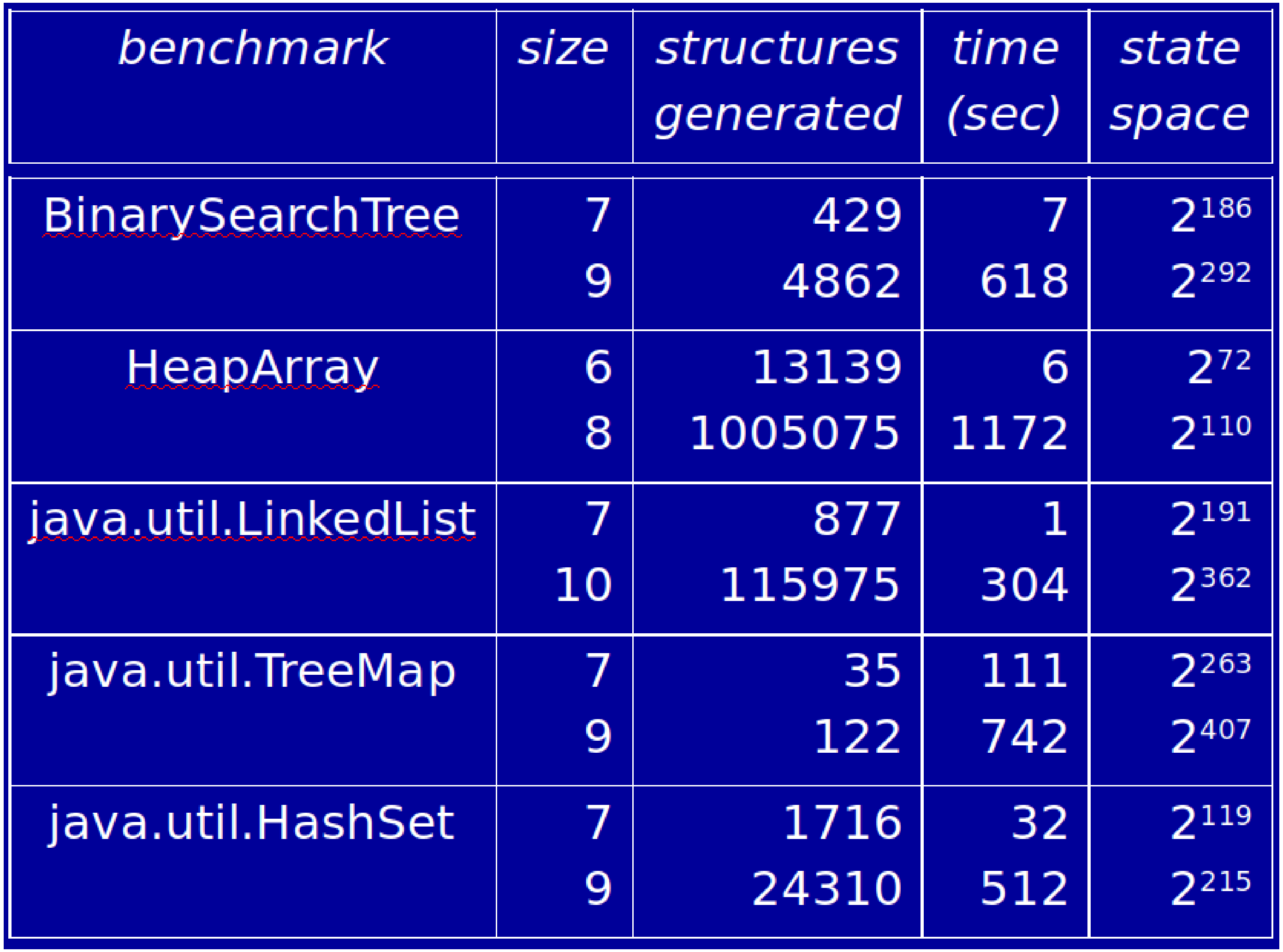
E.g., with full symmetry breaking, there are 5 trees (with 3 nodes and keys {1, 2, 3}):



- With no symmetry breaking, there are 5 x 3! = 30 trees

For red-black trees with 9 nodes, solving time is >5x less for full symmetry breaking vs. Alloy's default SB [SAT'03]

Results (historic context)



benchmark	*size*	*structures generated*	*time (sec)*	*state space*
BinarySearchTree	7	429	7	2^{186}
	9	4862	618	2^{292}
HeapArray	6	13139	6	2^{72}
	8	1005075	1172	2^{110}
java.util.LinkedList	7	877	1	2^{191}
	10	115975	304	2^{362}
java.util.TreeMap	7	35	111	2^{263}
	9	122	742	2^{407}
java.util.HashSet	7	1716	32	2^{119}
	9	24310	512	2^{215}

Using Alloy with mChaff back in the early 2000’s [SAT’03]

Related work (a few pointers)

Solution enumeration

Alloy

Symmetry [Shlyakhter-SAT'01][KMSJ-SAT'03]

Minimality [Nelson+ICSE'13]

Field exhaustiveness [Ponzio+FSE'16]

Coverage [SPIN'14][Porncharoenwase+FM'18]

Alternative formulation [Trippel+MICRO'18]

Other systems

Dedicated search [BKM-ISSTA'02][KPV-TACAS'03]

Mixing solvers and dedicated generators [GGJKKM-ICSE'10][Kuraj+OOPSLA'15]

Solver-aided languages [Ringer+OOPSLA'17]

Sampling [Meel+AAAI-Workshop'16][Dutra+ICCAD'18]

Work funded in part by the National Science Foundation

Conclusions

Model enumeration has many applications in software (and hardware) engineering

- E.g., in testing, analysis, synthesis, and repair

Symmetry breaking is vital for scalability!

- Without it, too many redundant solutions and much higher time cost

Designing SAT solvers for faster/better enumeration is very important!

CNF benchmarks for enumeration and symmetries:

http://projects.csail.mit.edu/mulsaw/alloy/sat03

khurshid@utexas.edu