Automated SAT-based Analysis of Relational Models and Code

Marcelo Frias mfrias@dc.uba.ar

University of Buenos Aires

Argentina

(Joint work with Juan Galeotti and Nicolas Rosner)

Tutorial. RelMiCS/AKA 2009, Doha, Qatar, 1-5 November, 2009.

Contents

- SAT-solving
- Alloy and the Alloy Analyzer
- KodKod
- TACO: Translation of Annotated COde
- Conclusions and further work

SAT-Solving

- The SAT problem: given a propositional formula A, find a satisfying valuation v : Vars -> {T, F}.
- First problem to be known as NP-complete.

SAT-Solving

- A *literal* is a variable *v* or its negation (not *v*)
- A literal is *pure* if it appears always with the same sign.
- A clause is a disjunction of literals:

  *v*1 or not *v*2 or ...or not *v*k {*v*1, not *v*2,..., not *v*k}
- A *unit* clause contains a single literal.
- A formula is in *conjunctive normal form* (CNF) if it has the form

  f = *c*1 and *c*2 and ... and *c*n

  where the *c*i are clauses. *f* = {*c*1, *c*2,..., *c*n}

SAT-Solving

The Davis-Putnam-Logemann-Loveland algorithm (1960, 1962):

```
DPLL(Φ) =
  if Φ is a consistent set of literals then return
  true;

  if Φ contains an empty clause then
  return false;

  for every unit clause l in Φ
  Φ := unit-propagate(l, Φ);

  for every literal l that occurs pure in Φ
  Φ := pure-literal-assign(l, Φ);

  l := choose-literal(Φ);
  return DPLL(Φ∧l) OR DPLL(Φ∧not(l));
```

SAT-Solving: Examples

$\phi_{1 \wedge v1} = \{\}$

```
DPLL(ϕ) =
  if ϕ is a consistent set of literals then return
  true;

  if ϕ contains an empty clause then
  return false;

  for every unit clause l in ϕ
  ϕ := unit-propagate(l, ϕ);

  for every literal l that occurs pure in ϕ
  ϕ := pure-literal-assign(l, ϕ);

  if there are literals left then
    l := choose-literal(ϕ);
    return DPLL(ϕ∧l) OR DPLL(ϕ∧not(l));
```

The Alloy Modeling Language (Jackson)

- Allows to describe data domains, and operations on such domains.
- The Alloy Analyzer allows to analyze wether properties hold in the models (but within bounded sizes for data domains).

A Simple Alloy Model

```
sig A {
one sig
fact reflexive {       iden in  el.r }
fact transitive { (Rel.r).(Rel.r) in Rel.r }

assert rEqualsItsClosure { Rel.r = A<:*(Rel.r)}
check rEqualsItsClosure for 5
```

a field containing a binary relation

defines data

only "one"

binary identity relation on A

facts are axioms

composition of *n*-ary relations

property to be verified using the Alloy Analyzer

reflexive-transitive closure

gives instructions to the Alloy Analyzer on the sizes of data domains

Alloy: Relational Semantics

```
sig A { }
one sig Rel { r : A -> A }
fact reflexive { A<:iden in Rel.r }
fact transitive { (Rel.r).(Rel.r) in Rel.r }

assert rEqualsItsClosure { Rel.r = A<:*(Rel.r)}
check rEqualsItsClosure for 5
```

`A` is a set, unary relation

`Rel` is a singleton {`Rel`}

`r` is a ternary relation

`r` in `Rel` x `A`

“.” is composition.

`Rel.r`

= {`Rel`}.{(`Rel`,*a*1,*a*2): (`Rel`,*a*1,*a*2) in `r`}

= {(*a*1,*a*2): (`Rel`,*a*1,*a*2) in `r`}.

The Alloy Analyzer

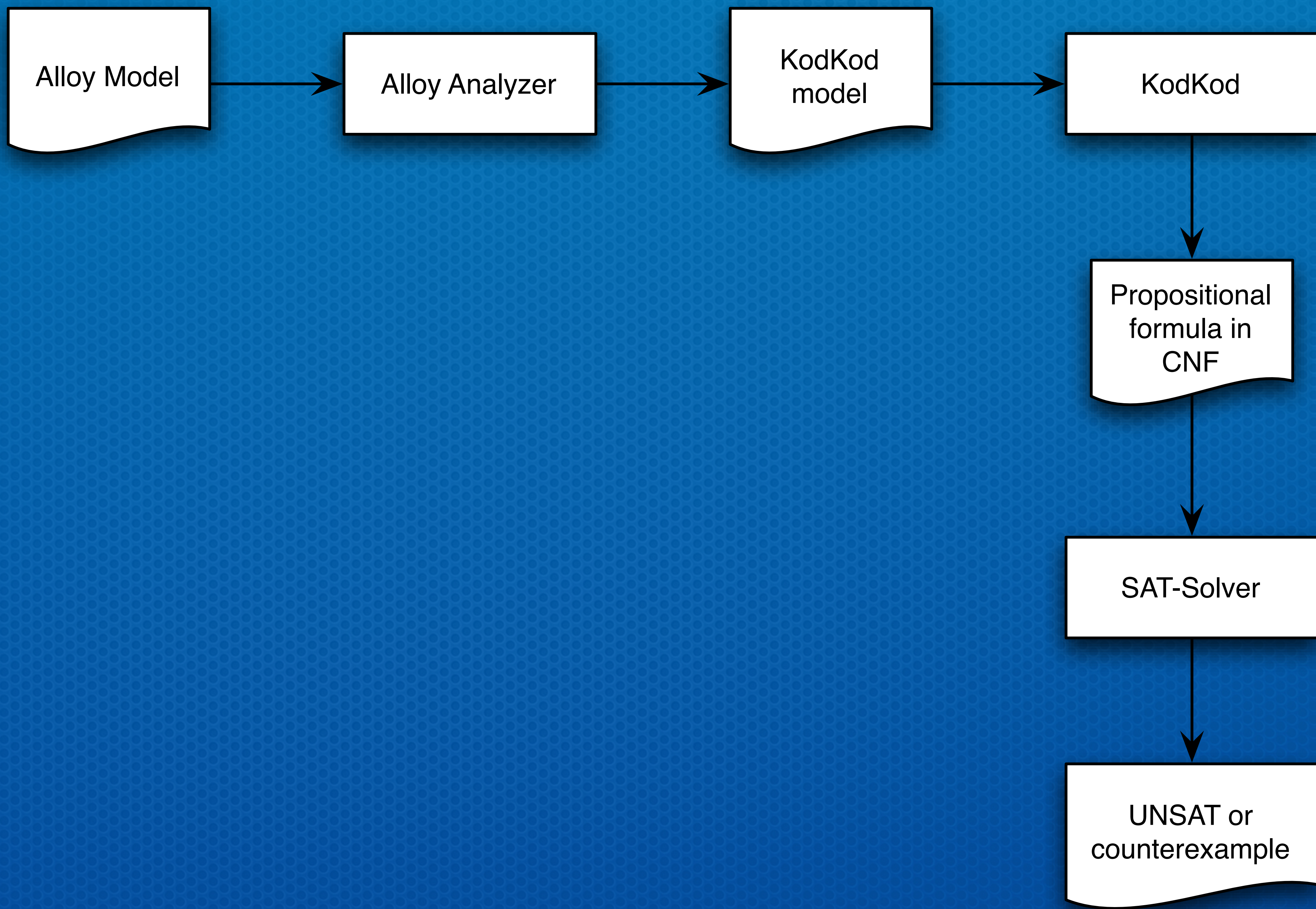
- open rEqualsItsClosure

The Alloy Analyzer

- For increasing scopes we get the following analysis times (TO means > 48hours).
-

8	9	10	11	12	13	14
00:00:04	00:05:22	00:58:58	04:05:41	36:34:13	TO	TO

The Alloy Analyzer

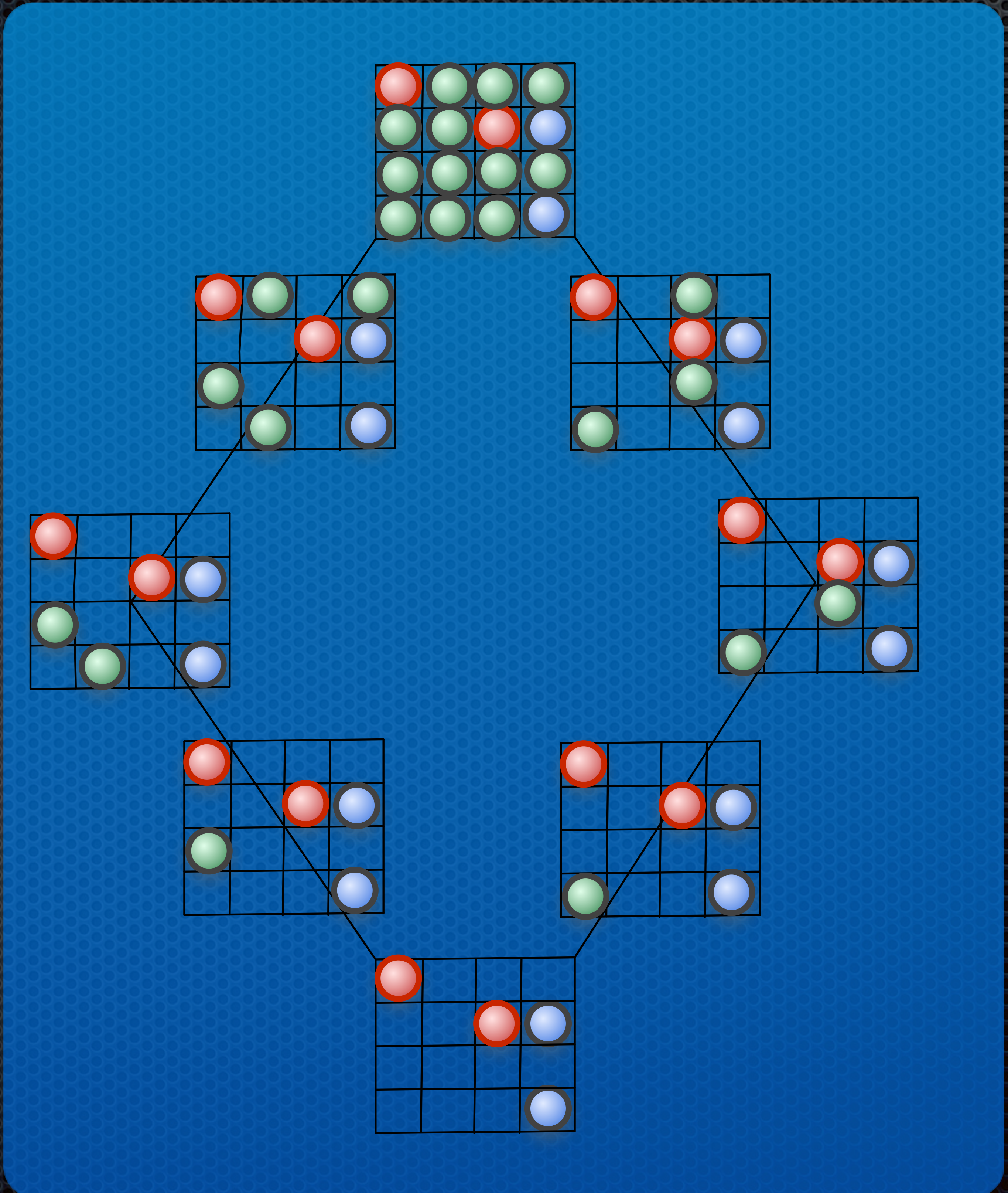


KodKod (Torlak, Jackson)

- For each relation symbol R, there are lower and upper bounds l_R and u_R.
- If a tuple *t* in l_R, then *t* must occur in every interpretation for R.
- If *t* does not occur in u_R, then *t* cannot occur in any intepretation for R.

KodKod

Intuitively,



KodKod: From Relational to Propositional

- Let R and S be binary relations on a set A. Let A's scope be 3. Then:

propositional variables

$$R \rightsquigarrow \begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} \qquad S \rightsquigarrow \begin{pmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{pmatrix}$$

r_{ij} is a propositional variable modeling wether pair (i,j) is in *R*. Similar for s_{ij}.

KodKod: From Relational to Propositional

- For transposition (converse), we have:

$$\breve{S} \leadsto \begin{pmatrix} s_{11} & s_{21} & s_{31} \\ s_{12} & s_{22} & s_{32} \\ s_{13} & s_{23} & s_{33} \end{pmatrix}$$

Relational terms are mapped to matrices of propositional formulas

- For join (union), we have:

$$R + \breve{S} \leadsto \begin{pmatrix} r_{11} \vee s_{11} & r_{12} \vee s_{21} & r_{13} \vee s_{31} \\ r_{21} \vee s_{12} & r_{22} \vee s_{22} & r_{23} \vee s_{32} \\ r_{31} \vee s_{13} & r_{32} \vee s_{23} & r_{33} \vee s_{33} \end{pmatrix}$$

KodKod: From Relational to Propositional

- For equalities between terms:

It is extended to connectives and quantifiers

$$R + \breve{S} = T$$

$$\Downarrow$$

$$\begin{pmatrix} r_{11} \vee s_{11} & r_{12} \vee s_{21} & r_{13} \vee s_{31} \\ r_{21} \vee s_{12} & r_{22} \vee s_{22} & r_{23} \vee s_{32} \\ r_{31} \vee s_{13} & r_{32} \vee s_{23} & r_{33} \vee s_{33} \end{pmatrix} = \begin{pmatrix} t_{11} & t_{12} & t_{13} \\ t_{21} & t_{22} & t_{23} \\ t_{31} & t_{32} & t_{33} \end{pmatrix}$$

$$\Downarrow$$

$$(r_{11} \vee s_{11} \Leftrightarrow t_{11}) \wedge (r_{12} \vee s_{21} \Leftrightarrow t_{12}) \wedge (r_{13} \vee s_{31} \Leftrightarrow t_{13}) \wedge \ \ldots$$

DynAlloy (Frias et al.)

- Is an extension of Alloy to model behavior.
- Semantics inspired on Dynamic logic.
- Allows to define atomic and composite actions.

DynAlloy: Atomic Actions

action Increment[*x* : Int] {

pre { **gt**[*x*,0] }

post { *x*’ = **add**[*x*,1] }

}

Precondition
to be satisfied by input

Postcondition.
Primed variables denote values in the final state

DynAlloy: Complex actions

- A1 + A2 : Nondeterministic choice
- A1 ; A2 : sequential composition
- alpha ? : test action (alpha is an Alloy formula)
- *A : reflexive transitive closure

DynAlloy: Analyzability

- We can analyze partial correctness assertions within domain scopes.
- We bound the number of iterations of the *.

```
assertCorrectness IncrementTwiceAdds2[x : Int] {
    pre = { gt[x,0] }
    program = {
        Increment[x];
        Increment[x]
    }
    post = { x' = add[x,2] }
}
```

Automated Analysis of Java Code

- Map the Java class hierarchy to the Alloy signatures hierarchy.
- Map Java atomic sentences to atomic actions.
- Map Java programs to DynAlloy.

Java to DynAlloy: Atomic

```
act NewC[o : C]
   pre = { true }
   post = {o’ !in ObjectsC and o’ in ObjectsC’}
```

```
act Setf[o : C, v : C’, f : C -> C’]
   pre = { o in ObjectsC }
   post = { f’ = f ++ (o -> v) }
```

```
act VarAssign[v1, v2 : C]
   pre = { true }
   post = { v1’ = v2 }
```

(abbreviated v1 := v2)

Java to DynAlloy: Code

stmt1 ; stmt2	$\mapsto$	$\mathbf{stmt1}; \mathbf{stmt2},$
if (pred) stmt1 else stmt2	$\mapsto$	$(\text{pred}?; \mathbf{stmt1}) + ((!\text{pred})?; \mathbf{stmt2})$
while (pred) {stmt}	$\mapsto$	$*(\text{pred}?; \mathbf{stmt}); (!\text{pred})?,$

Java to DynAlloy: Example

Ensures: property to be established by the method.
“result = true iff x is the value of a node in the list”

Invariant: all instances must satisfy it.
“Lists are acyclic structures”.
JML syntax.

```
public class LNode extends Object {
   LNode next;
   int key;
}

public class List extends Object {
   /*@
   @ invariant (\forall LNode n;
   @   \reach(this.head, LNode, next).has(n);
   @   !\reach(n.next, LNode, next).has(n));
   @*/
   LNode head;
```

```
/*@
@ ensures (\exists LNode n;
@   \reach(this.head, LNode, next).has(n) &&
@   n.val==x) <==> \result == true;
@*/
boolean find(int x) {
   LNode current;
   boolean output;
   current = this.head;
   output = false;
   while (output==false && current!=null) {
      if (current.val == x) {
         output = true; }
      current = current.next;
   }
   return output;
}
}
```

Class Hierarchy and Code

```
sig Object {}
one sig null {}
sig List extends Object {}
sig LNode extends Object {}
sig Throwable extends Object {}
sig Exception extends Throwable {}
sig RuntimeException extends Exception {}
one sig NullPointerException extends RuntimeException {}
```

Java to DynAlloy: Checking Correctness

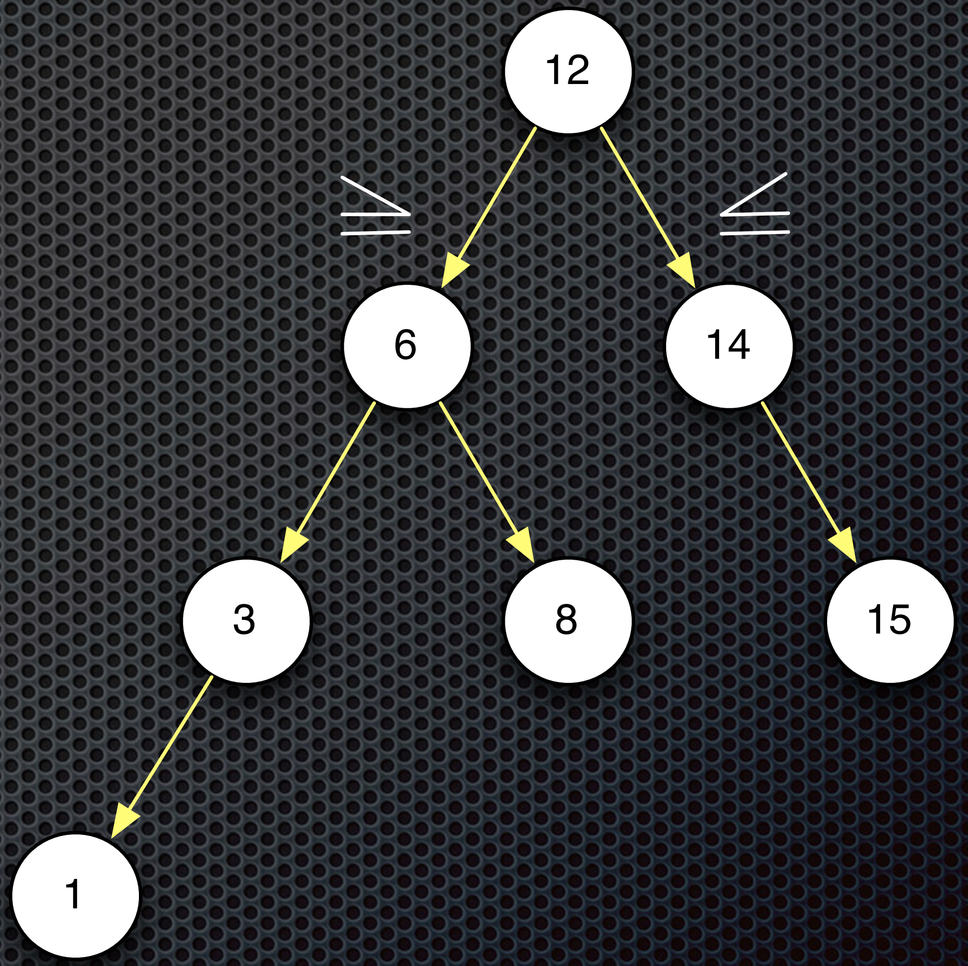
```
assertCorrectness find[this_L:List, result:Boolean,
   x:Int,
   head : List -> one (LNode + null),
   next : LNode -> one (LNode + null),
   key : LNode -> one Int ]{
pre { List_Inv[this_L, head, next] }
program { find[this_L, result, x, head, next, key] }
post {ensures_find[this_L,head,next,key,x,result']
      && List_Inv[this_L, head, next] }
}

check find for 5
```

TACO: Efficient Analysis of Java Code

- TACO: Translation of Annotated COde.
- Uses an efficient technique for reducing KodKod upper bounds.
- Analysis speeds up by several orders of magnitude.
- Experiments show that it improves over state-of-the-art tools based on model checking or SMT-solving.

A Sample Problem

Generating an instance of AVL-tree.



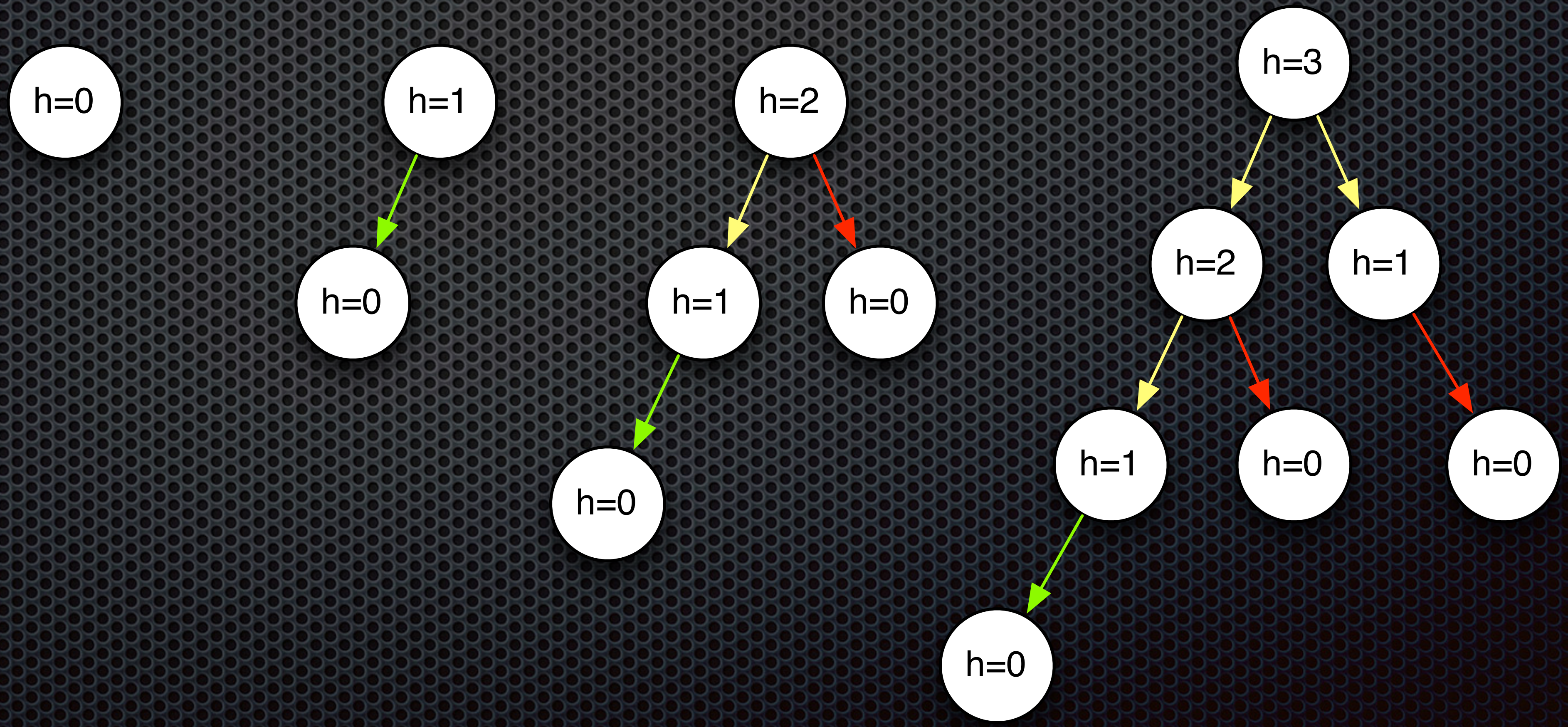
1. Binary tree,
2. Ordered,
3. Balanced: |h(left(n)) - h(right(n))| <= 1

Technique: Fully Automated Bound Refinement

- To find an instance the SAT-solver attempts to find (using strategies for pruning the state space) a tree thiz, and functions for fields root, h, left and right such that the invariant is satisfied.

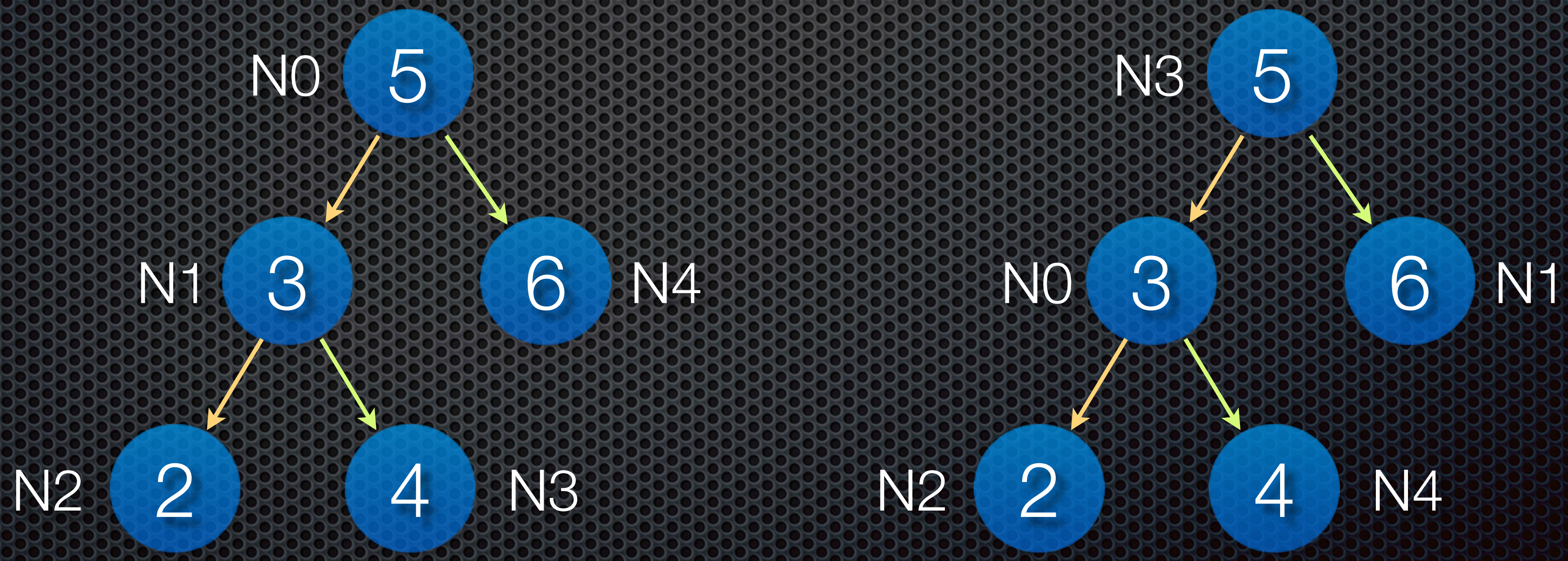
For AVL trees...

- Regarding field h, notice that all leaves have h = 0. Besides, since these are balanced trees, for up to 7 nodes no node satisfies h(n) greater than 3.



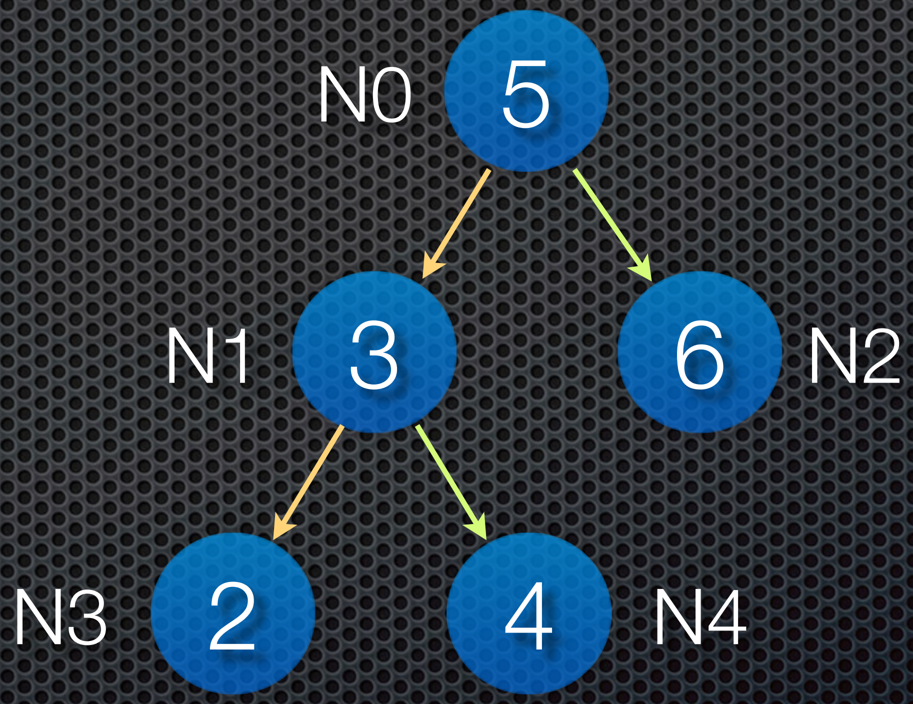
For AVL trees...

- Since nodes are objects, a node can hold different values (at different times). For instance:

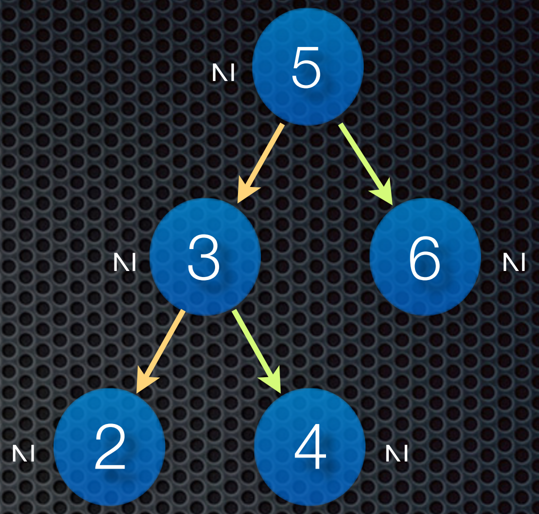


For AVL trees...

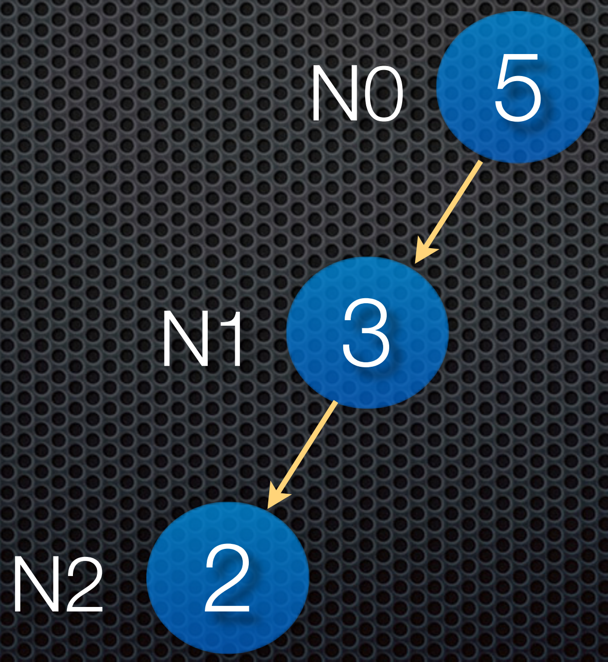
- But if we force nodes to be traversed in BFS order...



For AVL trees...



- Is it possible for N0 to point to a node that is neither N1 nor N2?
- Is it possible for N2 to be pointed to by a node other than N0?



Is not AVL!

Therefore, there are infeasible values...

- For instance, for a tree with up to 7 nodos, h(*n*)<=3 for all node *n*.
- Left(N0) is either N1 or null (but not N2, N3,...)
- Right(N0) is N1, N2 or null (but not N3, N4,...)
- Right(Ni) != N2 for i != 0.

Therefore, there are infeasible values...

- These values correspond to tuples in fields, and therefore, correspond to propositional variables in the KodKod translation.
- If we can remove these infeasible variables, the SAT-solver has fewer assignements to try.

Refining bounds reduces to:

- Forcing nodes to be allocated using a BFS traversal.
- Establishing the infeasible variables for each field.
- Doing all this in a fully automatic manner.

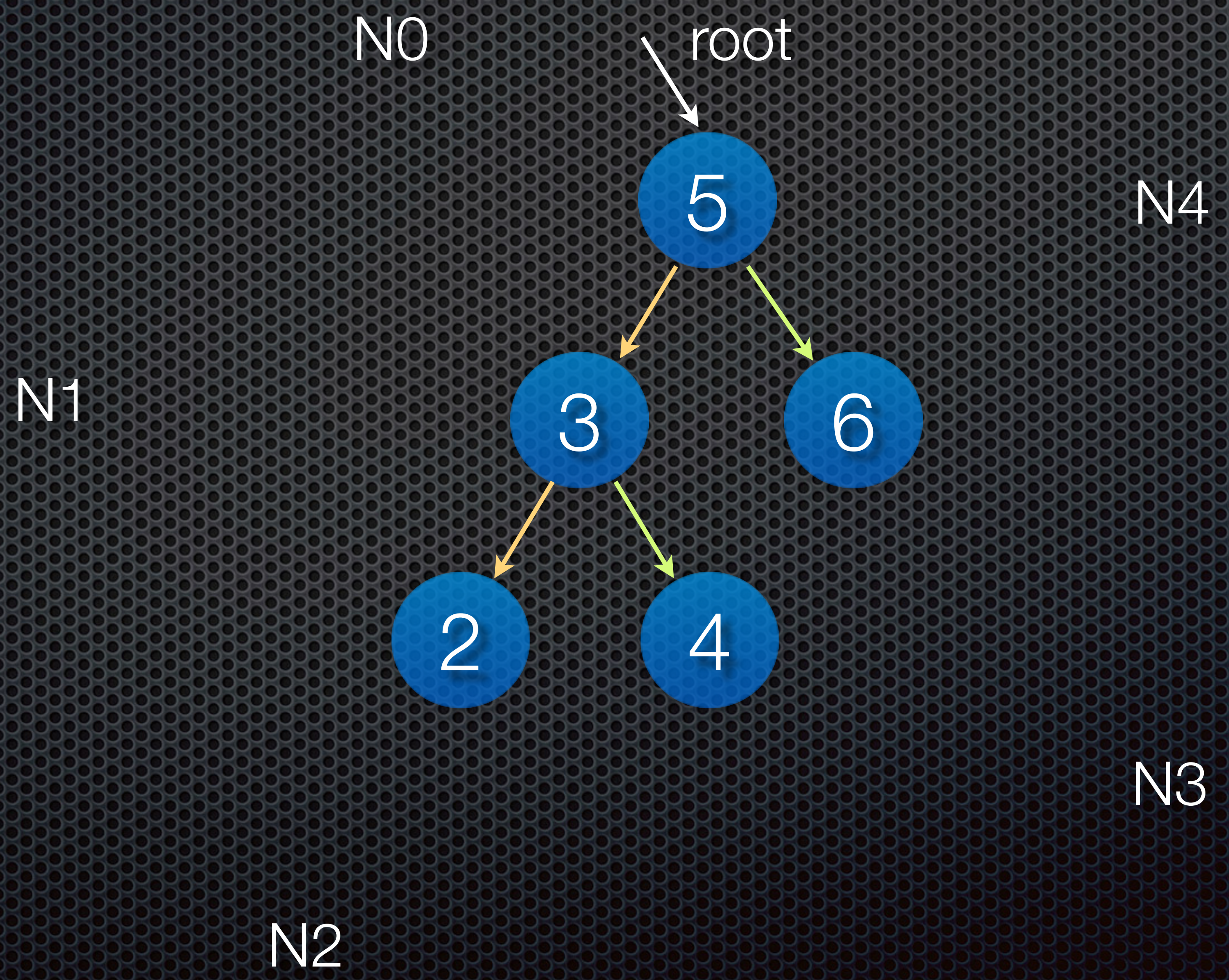
Hints:

- Instrument the relational model with new formulas forcing nodes to be allocated using a BFS ordering.
- Check feasibility for each pair in a class field.

Instrumenting the model

Applying once again Rule 2,

Rule2: Two nodes with the same parent are labeled from left to right.



Testing feasibility

Naive approach: use a cluster to analyze all pair in fields in parallel.

N0.left = N0,
N0.left = N1,
...
N0.h = 0,
N0.h = 1,
...

For 20 nodes, there are 2120 analyses to be performed.

Problem...
generating an AVL tree with 20 nodes does not finish.
No refined bounds yet!

An Effective Approach

p11 p10 p9 p8 p7 p6 p5 p4 p3 p2 p1 p0

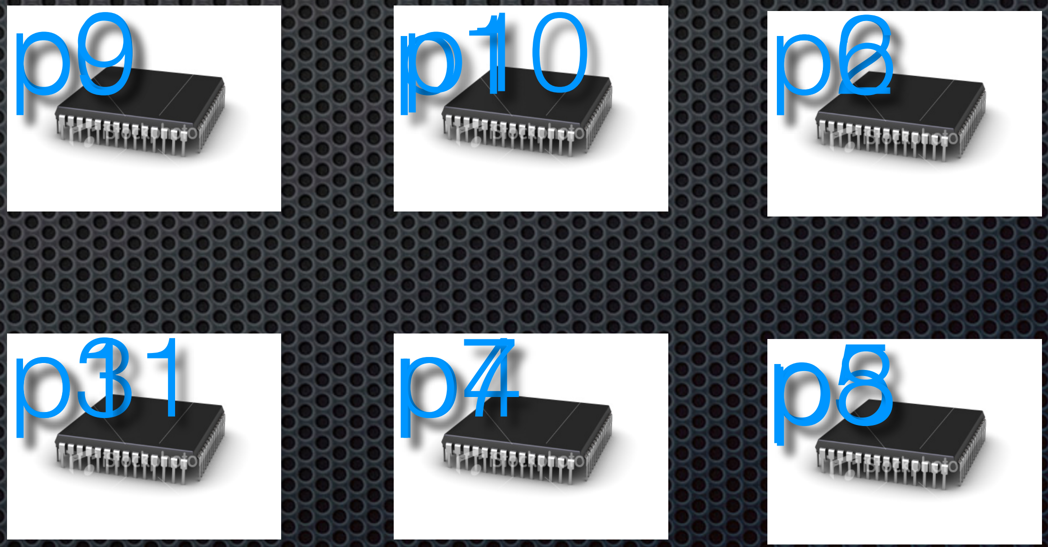
SATs

UNSATs

TIMEOUTS

Bound

p0	p1	p2
p3	p4	p5
p6	p7	p8
p9	p10	p11



Demo 2: Instances from refined bounds.

open generateAVL10Nodes.als (aprox. 1 minute)

open instGenerateAVL15Nodes.als

Code Analysis: Experimental Results

We compare with:

- JForge (MIT)
- Java Pathfinder (NASA)
- KIASAN (Kansas State University)
- ESC/Java2 (University College Dublin)
- Jahob (ETH - Zurich)

Code Analysis: Experimental Results

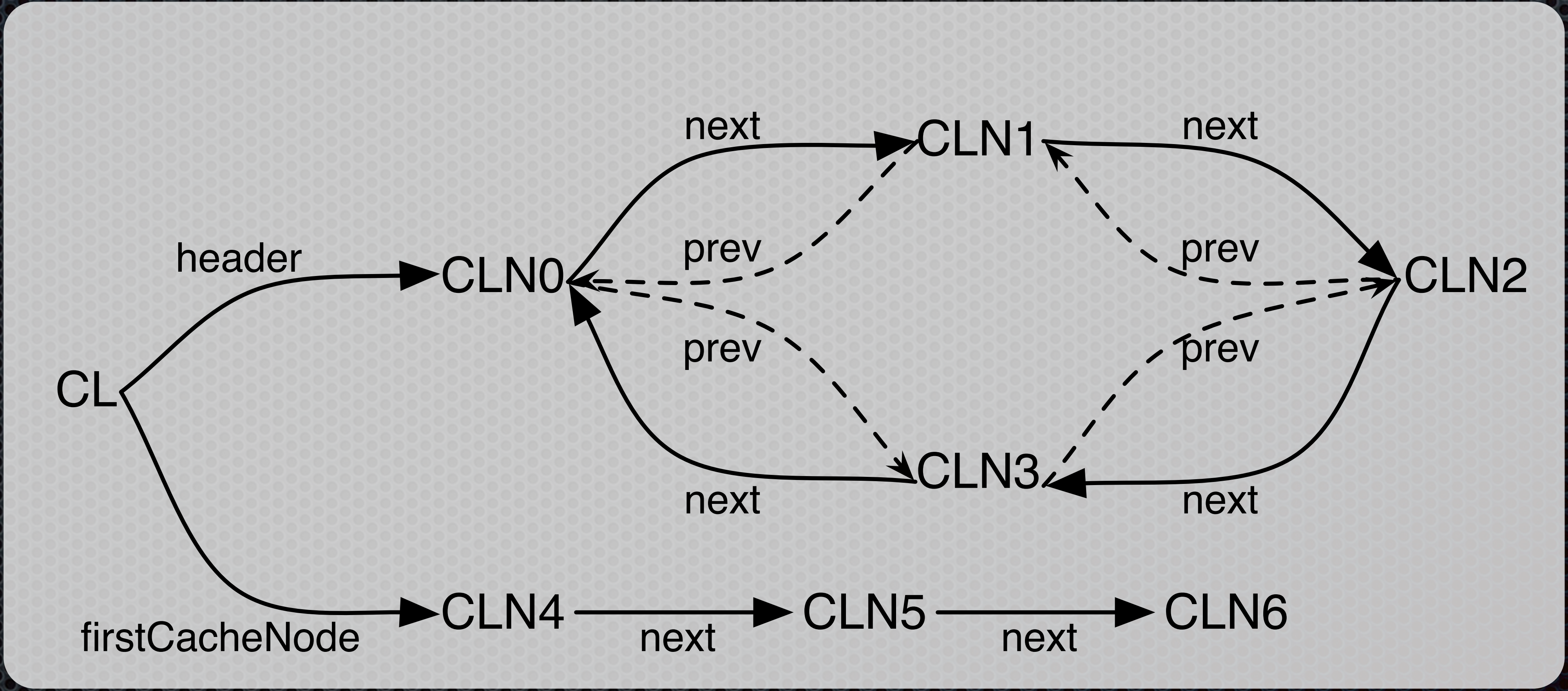
			5	7	10	12	15	17
LList	Cont	NI	00:03	00:05	00:08	00:11	00:13	00:22
		JF	00:01	02:00	TO	TO	TO	TO
		I	00:03	00:04	00:05	00:06	00:07	00:09
	Ins	NI	00:04	00:09	01:14	00:33	04:26	01:25
		JF	00:02	04:56	TO	TO	TO	TO
		I	00:04	00:05	00:07	00:08	00:13	00:26
	Rem	NI	00:05	00:27	TO	TO	TO	TO
		JF	00:04	21:51	TO	TO	TO	TO
		I	00:04	00:06	00:11	00:12	00:17	00:33
AList	Cont	NI	00:05	00:11	00:29	00:38	00:42	01:20
		JF	00:02	05:01	TO	TO	TO	TO
		I	00:04	00:06	00:16	00:22	00:27	00:58
	Ins	NI	00:04	00:05	01:02	26:22	TO	TO
		JF	00:03	11:52	TO	TO	TO	TO
		I	00:04	00:05	00:07	00:08	00:12	00:16
	Rem	NI	00:06	00:14	11:25	05:47:39	TO	TO
		JF	00:18	01:13:27	TO	TO	TO	TO
		I	00:05	00:06	00:17	00:31	01:08	03:13
TreeSet	Find	NI	02:13	04:36:49	TO	TO	TO	TO
		JF	00:42	01:57:49	TO	TO	TO	TO
		I	00:04	00:10	01:56	12:43	58:54	05:05:06
	Ins	NI	21:38	TO	TO	TO	TO	TO
		JF	OofM	OofM	OofM	OofM	OofM	OofM
		I	00:43	08:44	TO	TO	TO	TO

Code Analysis: Experimental Results

AVL	Find	NI	00:14	27:06	TO	TO	TO	TO
		JF	00:26	03:10:10	TO	TO	TO	TO
		I	00:03	00:06	00:36	01:41	08:20	33:06
	FMax	NI	00:02	00:04	46:12	TO	TO	TO
		JF	00:06	49:49	TO	TO	TO	TO
		I	00:01	00:01	00:03	00:04	00:09	00:13
	Ins	NI	01:20	05:35:51	TO	TO	TO	TO
		JF	OofM	OofM	OofM	OofM	OofM	OofM
		I	00:07	00:34	04:47	21:53	02:53:57	TO
BHeap	Min	NI	00:03	00:41	TO	TO	TO	TO
		JF	00:22	01:23:07	TO	TO	TO	TO
		I	00:02	00:04	00:11	00:20	02:29	00:07
	DecK	NI	00:30	38:58	TO	TO	TO	TO
		JF	01:48	TO	TO	TO	TO	TO
		I	00:10	00:59	24:05	02:42:30	TO	00:26
	Ins	NI	01:55	51:22	TO	TO	TO	TO
		JF	01:13:47	TO	TO	TO	TO	TO
		I	00:16	01:05	10:44	21:31	01:20:09	51:55

Finding a Nontrivial Bug

- Cache Lists: include a cache where removed nodes are stored so that they are not garbage collected.



Experimental Results

```
public Object remove(int index) {
  Node node = getNode(index, false);
  Object oldValue = node.getValue();
  super.removeNode(node);
  if (cacheSize >= maximumCacheSize){
     return;
  }
  Node nextCacheNode = firstCacheNode;
  node.previous = null;
  node.next = nextCacheNode;
  firstCacheNode = node;
  return oldValue;
}
```

```
public Object remove(int index) {
  Node node = getNode(index, false);
  Object oldValue = node.getValue();
  super.removeNode(node);
  if (cacheSize > maximumCacheSize){
     return;
  }
  Node nextCacheNode = firstCacheNode;
  node.previous = null;
  node.next = nextCacheNode;
  firstCacheNode = node;
  return oldValue;
}
```

LU	JForge	ESC/Java2	JPF	Kiasan	Jahob	TACO
4	OofM(227)	OofM(206)	TO	OofM(4)	03:03:19	03:52 + 03:56
6	TO	OofM(207)	TO	OofM(4)	05:05:29	03:52 + 31:14
8	OofM(287)	OofM(213)	TO	OofM(4)	07:39:01	03:52 + 33:23
10	05:40:22	OofM(215)	TO	OofM(4)	TO	03:52 + 00:11
12	06:53:04	OofM(219)	TO	OofM(4)	TO	03:52 + 03:30
15	24:08	OofM(219)	TO	OofM(4)	TO	03:52 + 15:00
20	TO	OofM(218)	TO	OofM(4)	TO	03:52 + 00:06

- Thanks!

Marcelo Frias mfrias@dc.uba.ar

University of Buenos Aires

Argentina

(Joint work with Juan Galeotti and Nicolas Rosner)