Karlsruhe Institute of Technology

Static Program Checking

Bounded Verification – Jalloy

Automated Software Analysis Group, Institute of Theoretical Informatics

Jun.-prof. Mana Taghdiri

June 5, 2014

Modeling dynamic behavior in Alloy

- Dynamic attributes
  - Those parts of the model that change in the lifetime of the system
  - E.g. the spanning tree algorithm
    - Whether a node of the graph is already in the tree or not
- Alloy has no built-in notion of “state” or “time”
  - Provides flexibility
  - Users can pick the right formulation, and the most intuitive one
- Some common idioms
  - Local state
  - Global state

Global vs. local state

- Alloy is a side-effect free declarative language.
  - Cannot say that time advances (or state changes)
  - Instead, we define an order over all time ticks (or states) in order to talk about the order in which events happen.
- In local-state models, history is local to objects, but in global-state models, state is a snapshot of the whole system at each time
  - Local-state: parent: Node → Time → lone Node
  - Global-state: parent: State → Node → lone Node
  - By shifting the time notion, in local-state models, we can maintain all the attributes of an entity in a single place, i.e. the declaration of that entity.
- Global state distinguishes between static and dynamic attributes
  - A state can be added on top of an existing model of static attributes
  - Separation of concerns
- Local-state modeling results in better modularity.
- Simpler? More intuitive?

Jalloy – problem statement

- Checking deep user-defined properties of object-oriented code

- Properties are about the functionality of the code:
  - Pre-condition => post-condition
  - Include linked data structures
    - Can get arbitrarily complex

- Most tools target “temporal safety properties”
  - Represented by a finite state machine
  - Good for checking properties that describe event sequences
  - Example? Lock acquire/release

Jalloy

- Inputs
  - A Java procedure (method)
  - A description of pre and post conditions – property – (in Alloy)
  - Finite bounds (number of objects, loop iterations)
- Outputs
  - A sound bug (no false alarms)

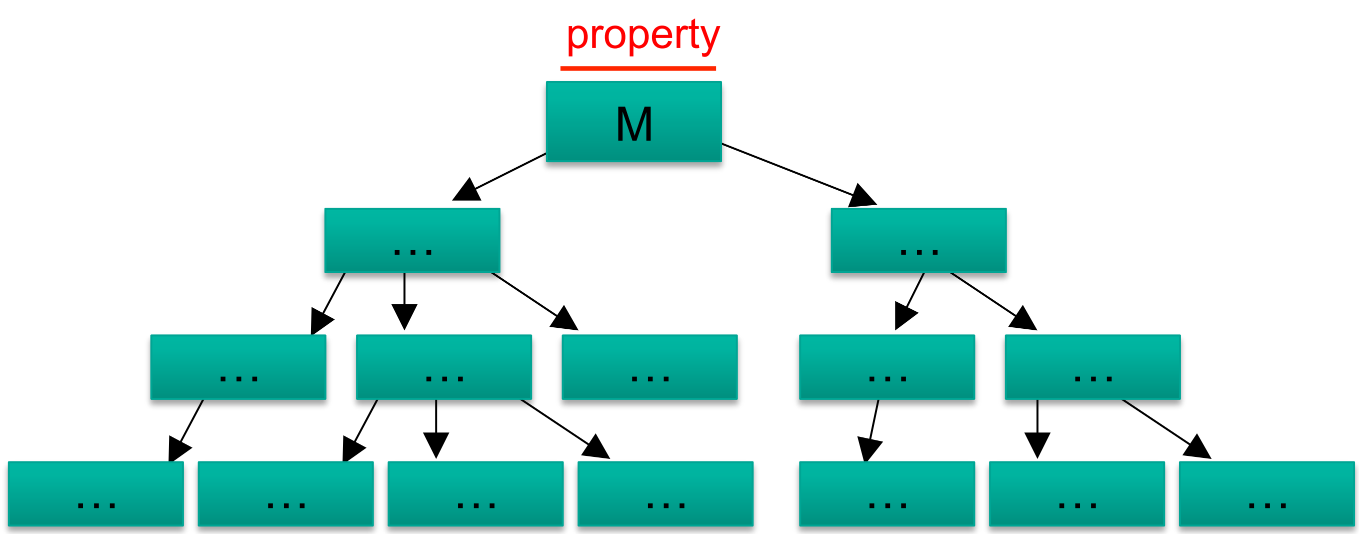
Other verification tools for structural properties

- Verification tools
  - Prove that the code is correct
  - Examples
    - Shape analysis (TVLA)
    - Theorem proving (KeY)
  - Scalability is a big problem
  - A lot of annotations should be provided by the user
- Bounded verification
  - Look for a bug statically – lack of bug is not conclusive
  - Examples
    - Based on Alloy (Jalloy, Forge, Karun)
    - Based on SMT (InspectJ)
    - Based on Simplify (ESC/Java)
  - Scale better than verification
  - Amount of user-provided annotations depends on the tool

General approach

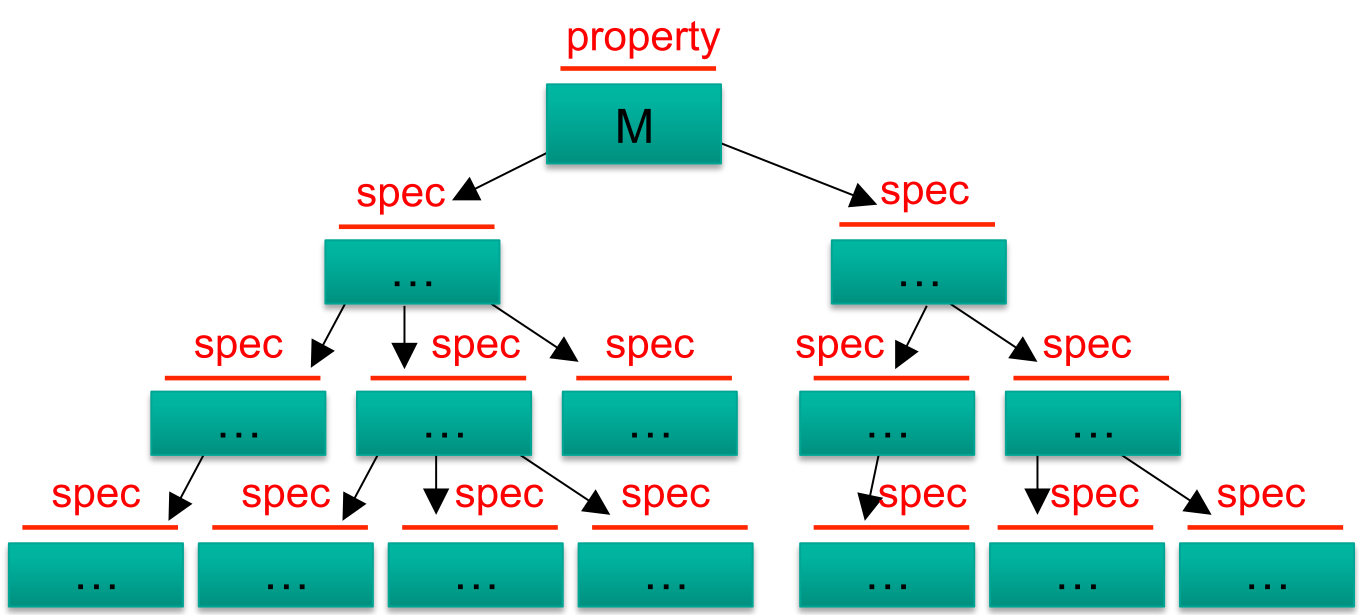
- Translate the code to a logical formula (c)
- Translate the property to a logical formula (p)
- Use a constraint solver on ($c \wedge \neg p$)
- Any satisfying solution is a code execution violating the property

Either translate the code precisely, or ..



Modularity

- Replace a procedure with its specification
- Assume-guarantee (done bottom up)
- Makes the technique scale better
- (like a divide-and-conquer approach)

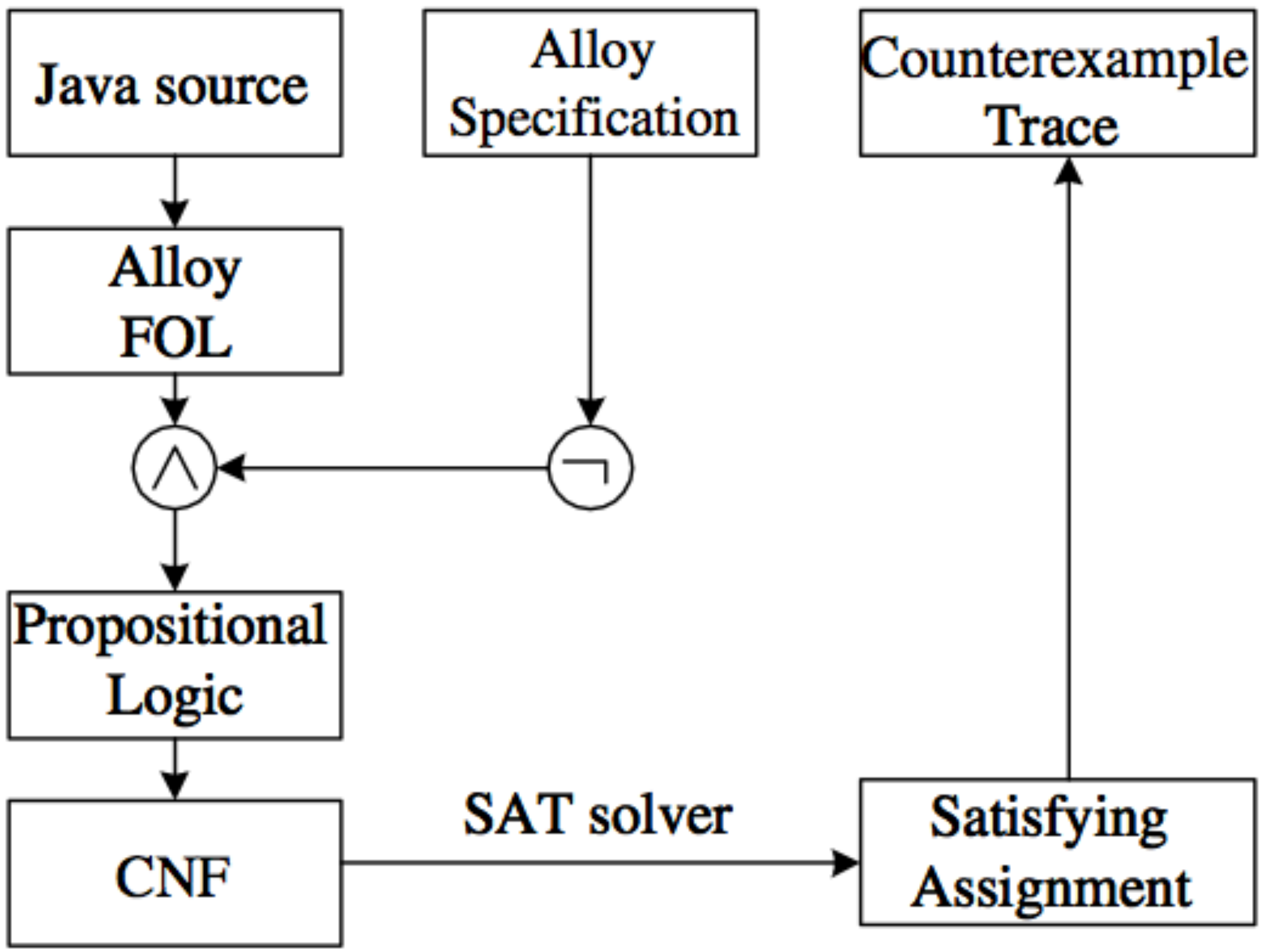


Modularity

- The user must provide all these intermediate specifications
- Costly for users:
  - Proportional to the size of code
  - ESC/Java: annotations can be 10% of the implementation size
  - Hob: annotations can be 40% of the implementation size

- Jalloy is modular
  - Can substitute specifications for procedures
- But, doesn't have to
  - If no specifications provided, inlines procedure calls

Jalloy architecture



Jalloy's algorithm

- Uses a 3 step translation:
  - From code to an Alloy formula
  - From Alloy to propositional logic
  - From propositional logic to CNF
- A SAT solver solves the generated CNF
  - A solution is a counterexample to the property being checked
- Jalloy came out at the time of Alloy 3
  - Alloy had no well-defined API
    - Jalloy had to produce Alloy files and parse them using the Alloy Analyzer
  - Many optimizations were absent from Alloy
    - Alloy 3 is overall much slower than Alloy 4
- (Alloy 4 has a very well-written engine API: Kodkod)

Scalability

- Two possible approaches
  - Top-down:
    - Look at the constraint solver as a black box
    - Optimize the process to scale to larger code
    - Examples: Forge, Karun
  - Bottom-up:
    - Develop an efficient, domain-specific constraint solver
    - Example: Jalloy

- Jalloy employs a set of optimizations for all translation levels
  - Suitable in the context of code analysis
    - For Java → Alloy (Java fields are dynamic attributes)
    - For Alloy → Propositional logic (Java fields are functional relations)
    - For Propositional logic → CNF (for functional relations)
  - The size of the generated CNF reduced exponentially
    - Better analysis time
    - Scales to larger programs

Modeling the heap

- Relational vs. scalar variables
  - Relational requires an expressive logic
  - Relational can support expressing data structure properties
    - reachability
    - Acyclicity
- Example:

```
class ListElem {
  int val;
  ListElem next;
}
class List {
  ListElem first;
```

Modeling field updates – Local state – example

Swaps the tails of the two given linked lists

```
class ListElem {
  int val;
  ListElem next;
}
class List {
  ListElem first;
  static void swapTail(List l, List m){
    if (l.first != null && m.first != null) {
      ListElem temp = l.first.next;
      l.first.next = m.first.next;
      m.first.next = temp;
    }
  }
}
```

Local state – example

```
class ListElem {
  int val;
  ListElem next;
}
class List {
  ListElem first;
```

```
sig Time {}
sig ListElem {
  val: Time → one int,
  next: Time → lone ListElem
}

sig List {
   first: Time → lone ListElem
}
```

Local state – example

```
static void swapTail(List l, List m){
  if (l.first != null && m.first != null) {
    ListElem temp = l.first.next;
    l.first.next = m.first.next;
    m.first.next = temp;
  }
}
```

```
pred swapTail(l, m: List, t0, t1, t2: Time) {
  (some l.first[t0]) && (some m.first[t0]) => {
     let temp = l.first[t0].next[t0] | {
        l.first[t0].next[t1] = m.first[t0].next[t0]
        all o: ListElem-l.first[t0] | o.next[t1] = o.next[t0]
        all o: ListElem | o.val[t1] = o.val[t0]
        all o: List | o.first[t1] = o.first[t0]
        … // m.first.next = temp
     }
  } else
     t2 = t0
}
```

t0 = initial time
t2 = final time

Null modeled as empty value

t1 = time after first field update

Frame conditions

Local/global state modeling

- Frame conditions are necessary
  - We can’t leave the fields unconstrained
  - Frame conditions say which values stay the same
  - Writing those can be tedious
    - Every time a field is updated, one must say that other fields stay the same
- Almost every single program statement requires a new state
  - The scope of “state” or “time” is in the order of hundreds for a small Java method
  - All relations that have “state” or “time” as a column become huge
  - Alloy can’t handle this
- Good for hand-written Alloy models where the number of states is small

Jalloy translation of Java to Alloy

- After each statement, only duplicate the relation that was modified
  - Don’t allow any other changes

- Steps:
  - Build a computation graph
  - Introduce correctly-named variables
  - Encode data flow
  - Encode control flow

Jalloy – example

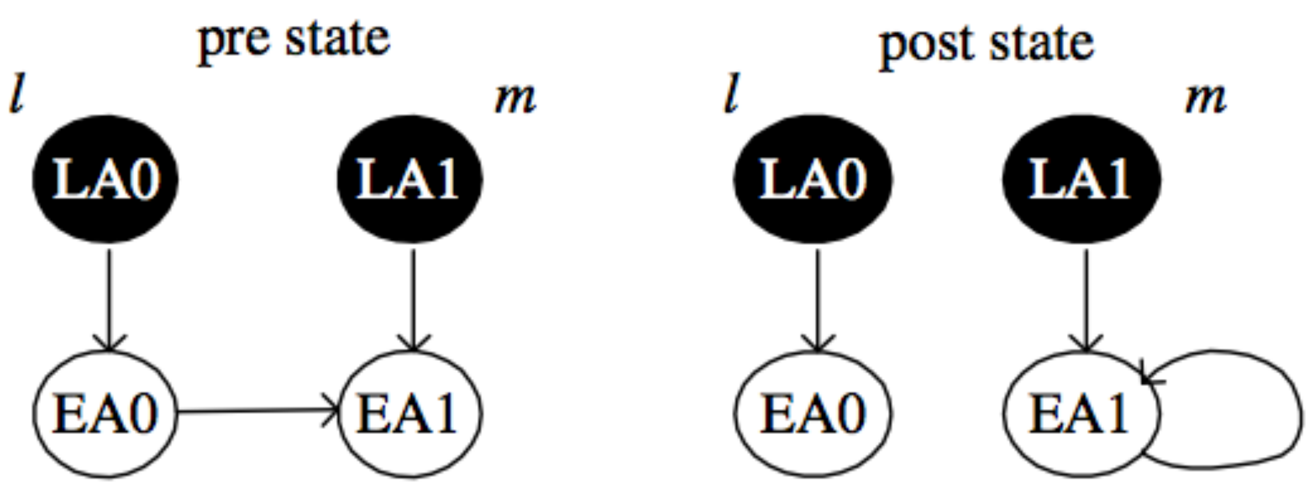
```
  class ListElem {
    int val;
    ListElem next;
  }
  class List {
    ListElem first;
    static void swapTail(List l, List m){
0     if (l.first != null
1                && m.first != null) {
        ListElem temp = l.first.next;
2       l.first.next = m.first.next;
3       m.first.next = temp;
4     }
  }}
```

l, m, first, next, val : pre-state
ret, first’, next’, val’ : post-state

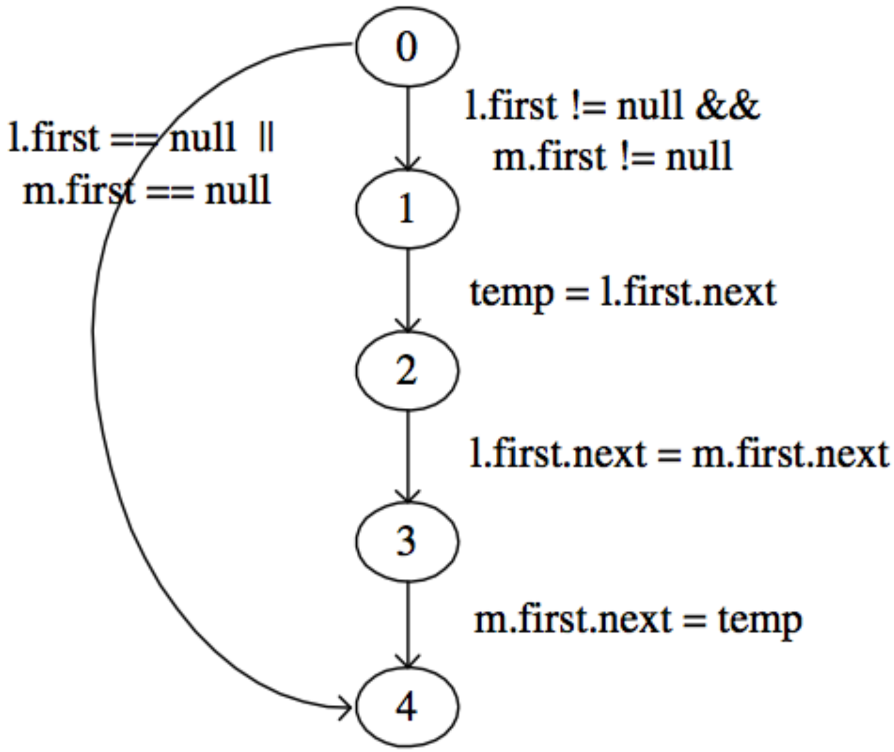
Property:
acyclic(l, first, next) **and**
acyclic(m, first, next) **implies**
acyclic(l, first’, next’) **and**
acyclic(m, first’, next’)

Is this property valid?

Jalloy counterexample



Computation graph



- ✓ Is a CFG with unrolled loops:
- ✓ Is a DAG
- ✓ Nodes = program points
- ✓ Edges = stmts and conditions

Single static assignment (SSA)

- SSA makes dataflow information explicit
- Usually used for compiler optimizations
- In every assignment to a variable v, it generates a fresh name for v
- Every time v is used, it is obvious which v it is.

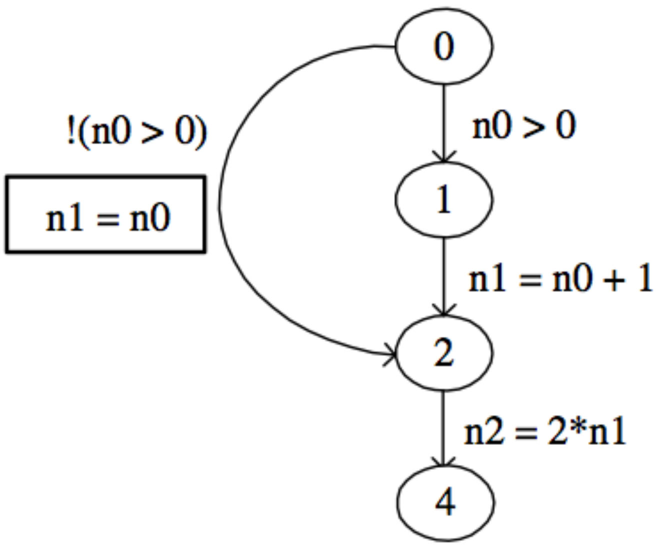
```
if (n > 0)
   n  = n + 1
n = 2*n
```

→

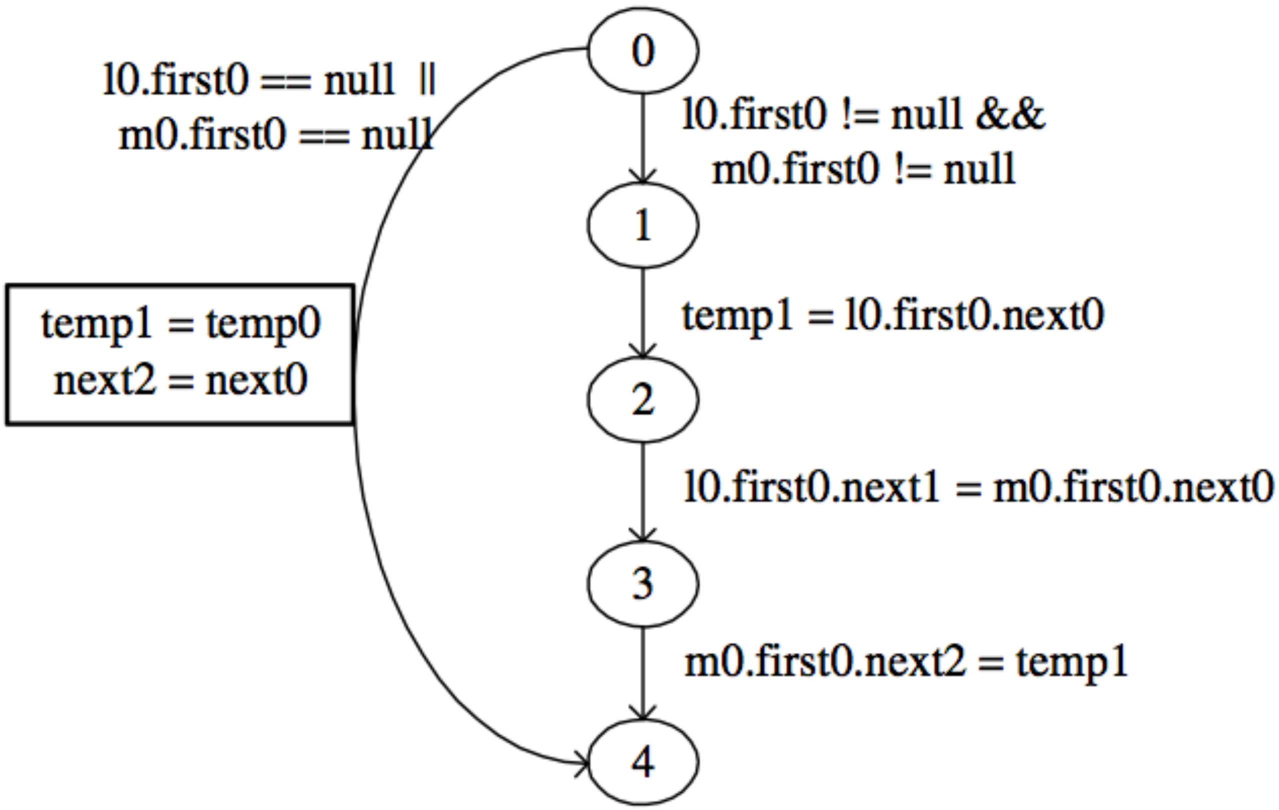
```
if (n0 > 0)
   n1  = n0 + 1
n2 =  φ(n0, n1)
n3 = 2*n2
```

Jalloy's renamings

- Same as SSA
  - Not only for variables, but also for fields (relations)
- No use of phi function
  - At each branch reuse the names.
  - Before the join point, constrain the shorter path s.t. variable name at the end of longer path = variable name at the end of shorter path

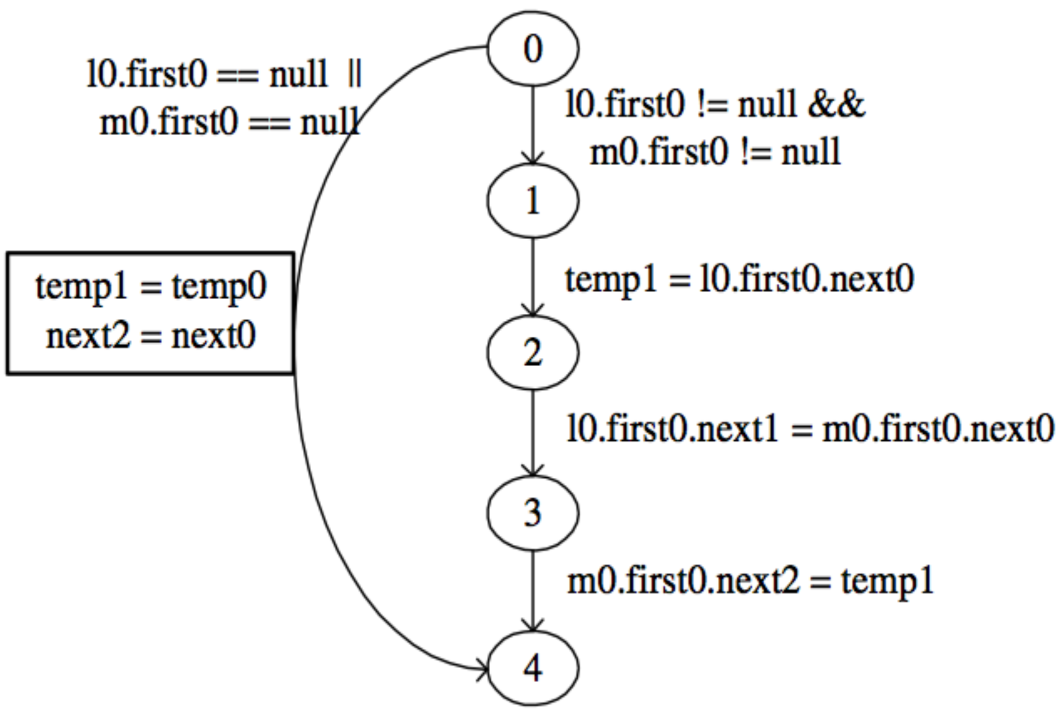


Example – swap tails



Encoding control flow

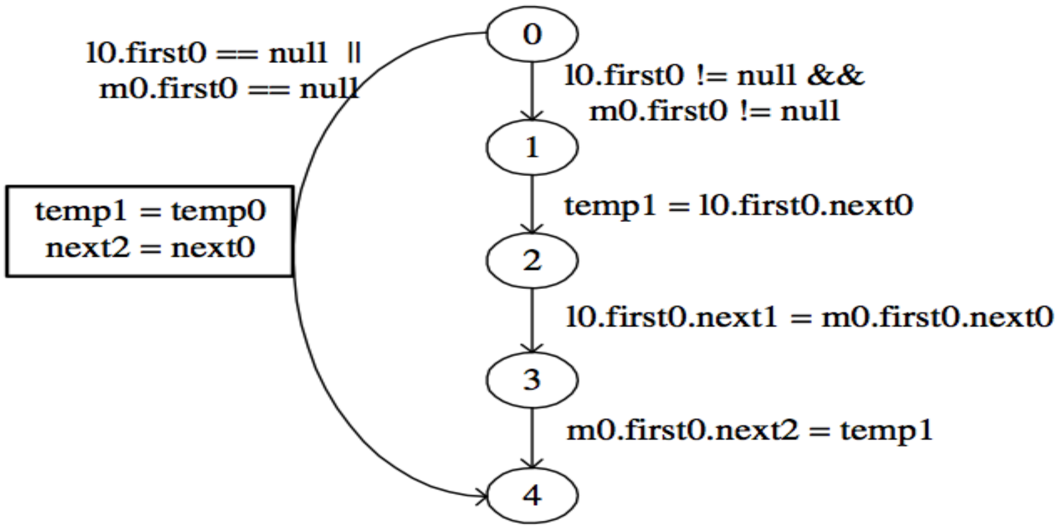
- The Alloy model includes a boolean variable for every edge of the computation graph.
  - An edge from node 0 to node 1 is modeled by variable E01
  - The value of this variable is true if and only if the edge is traversed




```
E_01 || E_04 &&
E_01 => E_12 &&
E_12 => E_23 &&
E_23 => E_34
```

Encoding data flow

- For every edge, express how relations are changed along that edge




```
E_01 => some l0.first0 && some m0.first0
E_04 => no l0.first0 || no m0.first0
E_12 => temp1 = l0.first0.next0
E_23 => l0.first0.next1 = m0.first0.next0 &&
          all o: ListElem - l0.first0 | o.next1 = o.next0
E_34 => m0.first0.next2 = temp1 &&
          all o:ListElem-m0.first0 | o.next2 = o.next1
```

Frame codition (→ `all o: ListElem - l0.first0 | o.next1 = o.next0`)

Frame codition (→ `all o:ListElem-m0.first0 | o.next2 = o.next1`)

Loop unrolling

```
{ …
  stmt1;
  while (cond) {
    stmt2;
  }
  stmt3;
  …
}
```

```
{ …
  stmt1;
  if (cond) {
    stmt2;
    if (cond) {
      stmt2;
    }
  }
  assume (!cond);
  stmt3;
  …
}
```

Jalloy only checks those executions that don't go over the loop more that the specified bound.

What happens to a for-loop with a fixed iteration number?

Program constructs

- Method calls
  - If a specification is provided, it will be used. Otherwise, the method is inlined
- Object allocation
  - x = new Type();
  - (x = T0) **and** (T0 !**in** usedType0) **and** (usedType1 = useType0 + T0)
- Dynamic dispatch
  - The actual type of an atom (representing an object) is determined by set membership test in Alloy
  - Dynamic dispatch becomes a switch statement
- Arrays and integers
  - Very limited support – due to Alloy's limited support for numbers
- Java API
  - Common library classes and methods are manually specified in Alloy

Discussions

- Advantages of this translation:
  - No explicit state atoms,
    - Instead of v: Time → Type, and v[t1], we have v1
  - Smaller CNF
    - Local/global state replicates all relations after every statement
  - Small frame conditions
    - They only concern the field being updated. Other fields can never be changed
- Treatment of null:
  - Null is represented by empty set
    - Can't express sets containing null (for the java set data structure)
  - Null can be represented by a special atom of each type
    - Makes relations bigger by one
    - Type hierarchy becomes hard to manage

From Alloy to propositional logic – review

- Represent relations by bit vectors
  - A unary relation r: A (signature, scalar, etc.)
    $[r_1\ r_2\ ..\ r_n]$ where
    r_i is a boolean variable and n = scope(A)
    (a vector)
  - A binary relation r: A → B
    $[r_{11}\ r_{12}\ ..\ r_{1n},\ r_{21}\ r_{22}\ ..\ r_{2n},\ ..,\ r_{m1}\ r_{m2}\ ..\ r_{mn}]$ where
    r_{ij} is a boolean variable and n = scope(B) and m = scope(A)
    (an m*n matrix)

- All relational operations are performed on these matrices
- Operations are done bottom up on the abstract syntax tree (AST)
- When we get to the root, we are left with a single boolean formula

Alloy to boolean – review

$r : A \rightarrow B$, $s : A \rightarrow B$

- r + s
  - A matrix of (r_{ij} or s_{ij})
- r & s
  - A matrix of (r_{ij} and s_{ij})
- r.s
  - Matrix multiplication
- r in s
  - A formula of and { (r_{ij} implies s_{ij}) }
- r = s
  - A formula of and { (r_{ij} implies s_{ij}) and (s_{ij} implies r_{ij}) }

From Alloy to CNF – Jalloy optimizations

- Fields declared in programs are functional
  - They map each (non-null) object to exactly one object (including null)
  - Can be modeled more efficiently and thus, reduce the CNF size

- Reducing the size of CNF is not necessarily good
  - The behavior of the SAT solver depends on the structure of the formula rather than its size
  - Symmetry-breaking in Alloy adds more clauses to the boolean formula

- However,
  - State-of-the-art solvers can handle formulas up to a certain size, beyond that, in most cases, requires either too long or too much memory
  - But still experiments are needed to see Jalloy CNF reductions are good or not

Representing functional relations

- Default of Alloy: represent relations by bit vectors
  - A binary relation r: A → B, scope = 3 (+null = 4 columns)

    [r_{00} r_{01} r_{02} r_{03}, r_{10} r_{11} r_{12} r_{13}, r_{20} r_{21} r_{22} r_{23}]

    (r_{ij} = true => <a_i, b_j> in r)
  - For a functional relation, in each row, exactly one boolean variable will be true
- Optimization: a logarithmic representation suffices
  - Encode the index of that one atom in binary form
  - [r_{00} r_{01}, r_{10} r_{11}, r_{20} r_{21}]
  - After a solution is found, read the values of the variables of each row as a binary number:
  - r_{00} = false, r_{01} = false (00) => <a0, b0> in r
  - r_{00} = false, r_{01} = true (01) => <a0, b1> in r
  - r_{00} = true, r_{01} = false (10) => <a0, b2> in r
  - r_{00} = true, r_{01} = true (11) => <a0, b3> in r

Field dereference

- In Java (Alloy): x.f = y
  - x, y scalar
  - f functional relation
- x and y represented by (log n) bits, f by n(log n) bits
  - $x = [x_1\ x_2\ ..\ x_k]$, $f = [f_{11}\ f_{12}\ ..f_{1k},\ ..,\ f_{n1}\ f_{n2}\ f_{nk}]$
- Construct 1*n representation of x
  - $[!x_1 \wedge !x_2 \wedge .. \wedge !x_k \quad !x_1 \wedge !x_2 \wedge .. \wedge x_k \quad \ldots]$, call this $[A_1\ A_2\ ..\ A_n]$
- x.f = y is given by
  - $A_1 \Rightarrow (f_{11} \Leftrightarrow y_1 \wedge f_{12} \Leftrightarrow y_2 \wedge \ldots \wedge f_{1k} \Leftrightarrow y_k)$
  - $A_2 \Rightarrow (f_{21} \Leftrightarrow y_1 \wedge f_{22} \Leftrightarrow y_2 \wedge \ldots \wedge f_{2k} \Leftrightarrow y_k)$
  - …
  - $A_n \Rightarrow (f_{n1} \Leftrightarrow y_1 \wedge f_{n2} \Leftrightarrow y_2 \wedge \ldots \wedge f_{nk} \Leftrightarrow y_k)$

Summary

- Jalloy optimizations
  - Reduced the size of the final CNF dramatically
  - Can check the code in a higher scope with more loop iterations

- Jalloy applications
  - Red-black tree
  - A garbage collection algorithm
  - A method in Jalloy implementation

- Looks like the analyzed method in each case is around 50LOC
- But very data structure intensive

- The first tool ever that used Alloy for static program analysis!

Can we do better than Jalloy?

- How else would you model the Java code in Alloy?

```
class ListElem {
  int val;
  ListElem next;
}
class List {
  ListElem first;
  static void swapTail(List l, List m){
    if (l.first != null && m.first != null) {
      ListElem temp = l.first.next;
      l.first.next = m.first.next;
      m.first.next = temp;
    }
  }
}
```

Alloy Analyzer as a backend engine

- Up until the end of Alloy 3, a clean API, as a standalone linkable piece of code was never the concern.
  - Alloy was a Desktop CAD application where the Analyzer would only parse the formulas and produce boolean SAT problems
  - Tools like Jalloy would generate Alloy text files and feed it to the parser
    - Awkward, and
    - Slow (these were usually just a single big formula, with no predicate, function, or let structures)
- Kodkod
  - Designed as a plug-in API
  - Clean and well-documented Java API
  - Kodkod logic is the core subset of the Alloy logic
- Alloy Analyzer 4
  - Is just a parser + kodkod

Kodkod

- Designed with focus on partial instances
  - What is a partial instance?
  - Example: Sudoku
  - Alloy doesn't support partial instance
    - Should model them as singleton signatures
    - The Analyzer has to re-discover the partial instance, thus slower analysis
- Better sharing detection mechanism
  - To avoid duplicate boolean variable generation in e.g. ground out quantifier
  - Alloy expressions were internally represented as a tree to simplify the algorithms
- Orders of magnitude better performance than Alloy 3
  - Especially when partial instances involved
  - Allows sharing sub-expressions and sub-formulas by a DAG data structure

Partial instance example – sudoku

- A 9x9 table divided into nine 3x3 sub-tables
- All rows must contain all numbers 1 to 9
- All columns must contain all numbers 1 to 9
- All 3x3 sub-tables must contain all numbers 1 to 9
- Some cells already have numbers (shaded cells)

1	4	5	2	8	9	3	7	6
7	2	6	5	3	1	8	4	9
9	8	3	7	6	4	1	2	5
6	1	9	4	2	7	5	3	8
3	7	4	1	5	8	9	6	2
2	5	8	3	9	6	4	1	7
8	6	2	9	4	3	7	5	1
4	9	7	6	1	5	2	8	3
5	3	1	8	7	2	6	9	4

Sudoku in Alloy

```
abstract sig Number { data: Number -> Number }

abstract sig Region1, Region2, Region3 extends Number {}

one sig N1, N2, N3 extends Region1 {}
one sig N4, N5, N6 extends Region2 {}
one sig N7, N8, N9 extends Region3 {}

pred complete(rows: set Number, cols: set Number) {

  Number in cols.(rows.data) }

pred rules() {
  all x, y: Number { lone y.(x.data) }
  all row: Number { complete(row, Number) }
  all col: Number { complete(Number, col) }
  complete(Region1, Region1)
  complete(Region1, Region2)
  complete(Region1, Region3)
  complete(Region2, Region1)
  complete(Region2, Region2)
  complete(Region2, Region3)
  complete(Region3, Region1)
  complete(Region3, Region2)
  complete(Region3, Region3)
}

pred puzzle() {
  N1->N1->N1 + N1->N4->N2 + N1->N7->N3 +
  N2->N2->N2 + N2->N5->N3 + N2->N8->N4 +
  N3->N3->N3 + N3->N6->N4 + N3->N9->N5 +
  N4->N1->N6 + N4->N4->N4 + N4->N7->N5 +
  N5->N2->N7 + N5->N5->N5 + N5->N8->N6 +
  N6->N3->N8 + N6->N6->N6 + N6->N9->N7 +
  N7->N1->N8 + N7->N4->N9 + N7->N7->N7 +
  N8->N2->N9 + N8->N5->N1 + N8->N8->N8 +
  N9->N3->N1 + N9->N6->N2 + N9->N9->N4 in data
}

pred game() { rules() && puzzle() }

run game
```

- The *N1-N9* declarations ensure that any solution contains exactly nine Number atoms.

- Field *data* maps (row, column) to the number in that cell

- Because Alloy lacks support for partial instances, given cells must be encoded as constraints on the data field

- For example, the constraint N1–>N7–>N3 in data ensures that the solution maps the cell (1, 7) to the number 3.

Kodkod vs. Alloy

- In Alloy,
  - Relational variables are divided into
    - Signatures (unary relations)
    - Fields (non-unary relations)
  - Signatures form a type hierarchy
  - Signatures are bound by an integer limit and that limits the relations too
- In kodkod,
  - All relations are interpreted the same
  - All relations are untyped
  - Has none of the Alloy's syntactic sugar (pred, func, fact)
  - Relations are bound from both above and below by relational constants
    - A fixed set of tuples drawn from a universe of atoms
    - Represent "may" and "must" values
  - Has no parser – no textual input

Sudoku in Kodkod – parts of the program

```
private final Relation Number = Relation.unary("Number");
private final Relation data = Relation.ternary("data");
private final Relation[] regions = new Relation[] {
  Relation.unary("Region1"), Relation.unary("Region2"),
  Relation.unary("Region3") };

public Bounds puzzle() {
  final Set<Integer> atoms = new LinkedHashSet<Integer>(9);
  for(int i = 1; i <= 9; i++) { atoms.add(i); }

  final Universe u = new Universe(atoms);
  final Bounds b = new Bounds(u);
  final TupleFactory f = u.factory();

  b.boundExactly(Number, f.allOf(1));
  b.boundExactly(regions[0], f.setOf(1, 2, 3));
  b.boundExactly(regions[1], f.setOf(4, 5, 6));
  b.boundExactly(regions[2], f.setOf(7, 8, 9));

  final TupleSet givens = f.noneOf(3);
  givens.add(f.tuple(1, 1, 1));
  givens.add(f.tuple(1, 4, 2));
  ...
  givens.add(f.tuple(9, 9, 4));
  b.bound(data, givens, f.allOf(3));

  return b;
}
```

- Universe is a user-provided Collection of Objects.
- Each Universe provides a TupleFactory for creating constants
- Relations have upper and lower bound TupleSets
- Unlike their Alloy equivalents, these relations are untyped
- Unlike its Alloy equivalent, the puzzle method encodes the partial instance in the Bounds rather than as constraints

Kodkod optimizations

- In kodkod, symmetry breaking is different because
  - Relations are untyped
  - Partial instance makes atoms distinct
- In kodkod, sharing detection is at the boolean level
  - In Alloy, it is done at the problem level
  - Kodkod uses compact boolean circuits
- Kodkod is a free open-source API
  - http://alloy.mit.edu/kodkod/

Kodkod vs. Alloy 3

	Sudoku (9 × 9)		
solver	time	vars	clauses
AA	3	11,618	44,152
KK	0	1,833	2,398

	Ceilings and Floors					
scope	6 men, 6 platforms			10 men, 10 platforms		
solver	time	vars	clauses	time	vars	clauses
AA	1	2,723	11,704	10	9,987	46,740
KK	0	1,749	3,289	4	6,477	12,449

	Mutex Ordering					
scope	30 atoms			45 atoms		
solver	time	vars	clauses	time	vars	clauses
AA	65	74,818	722,236	> 300	-	-
KK	2	20,080	120,097	15	67,695	543,597