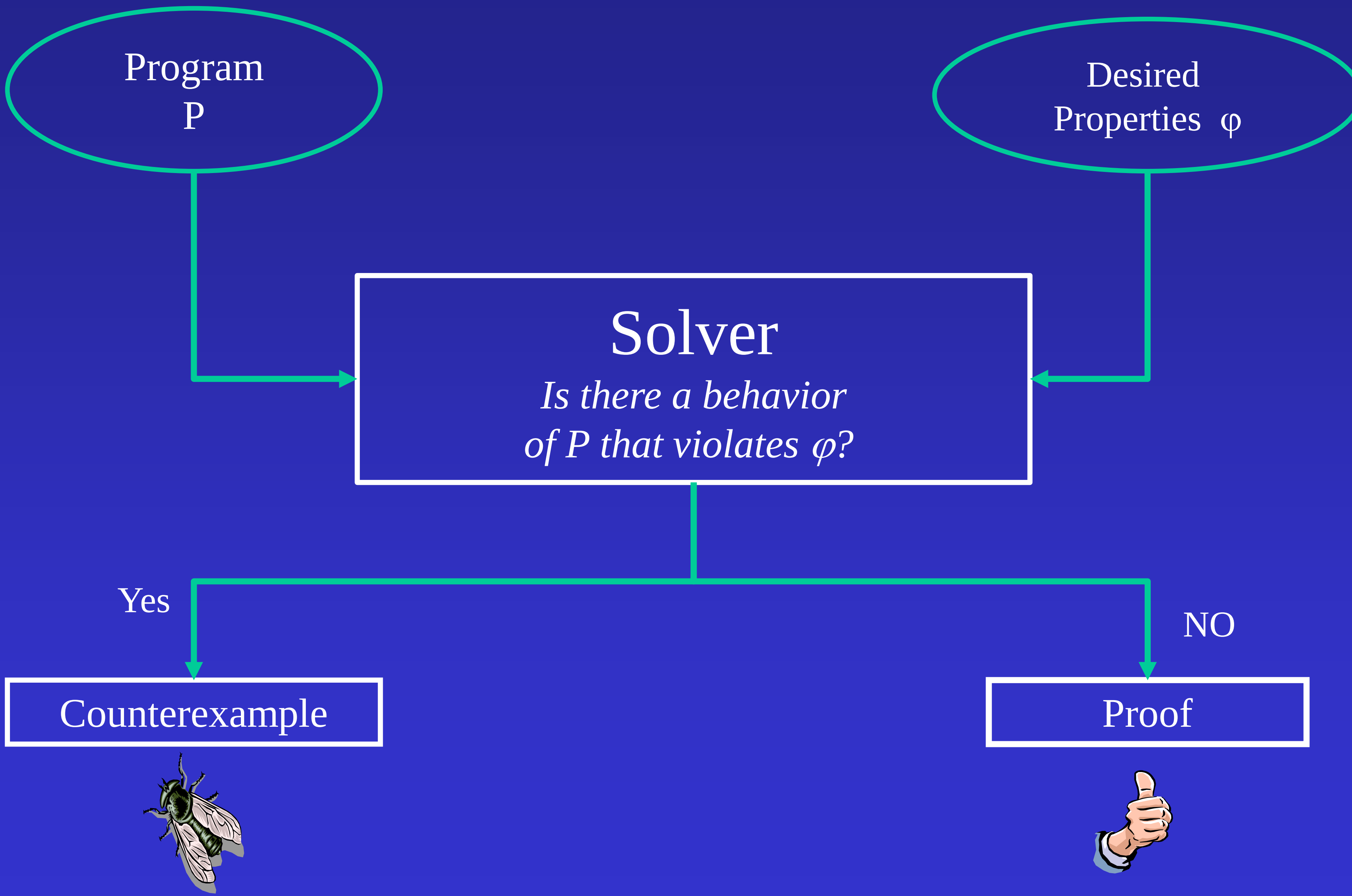
Bounded Model Checking

Mooly Sagiv

Slides from Arie Gurfinkel & Sagar Chaki, Daniel Jackson, Shahar Maoz

Automatic Program Verification



Simple Bug

```
scanf("%d", &n);
for (c = 0; c < n; c++)
  scanf("%d", &array[c]);
 for (c = 0 ; c < ( n - 1 ); c++)
{
  for (d = 1 ; d < n - c - 1; d++)
  {
    if (array[d] > array[d+1])
    {
      swap       = array[d];
      array[d]   = array[d+1];
      array[d+1] = swap;
    }
  }
}
```

Program Properties

- User defined assertions
- General cleanliness properties
  - Absence of buffer overruns
  - No null dereference
  - No double free
  - No overflow

Jackson’s Thesis

- If a program has a bug $\Rightarrow$ it also occurs on small input k
  - True in many cases

Model Checking

- Does a given model M satisfy a property P, $M \models P$
  - M is usually a finite directed graph
  - P is usually a formula in temporal logic
- Examples:
  - Is every request to this bus arbiter eventually acknowledged?
  - Does this program every dereference a null pointer?

Bounded Model Checking

- Given
  - A finite transition system M
  - A property P
- Determine
  - Does M allow a counterexample to P of *k transitions of fewer*?

This problem can be translated to a SAT problem

Bounded Model Checking of Loops

- Does the program reach an error within at most k unfolding of the loop
- Special kind of symbolic evaluation

Bounded Model Checking Tools

- CBMC: Bounded Model Checker for C and C++
  - Developed at CMU/Oxford
  - Supports C89, C99, most of C11
  - Verifies array bounds (buffer overflows), absence of null dereferences, assertions
- Alloy: Bounded model checking for program designs
  - Developed at MIT
  - Rich specification language
    - First order logic, transitive closure, arithmetics

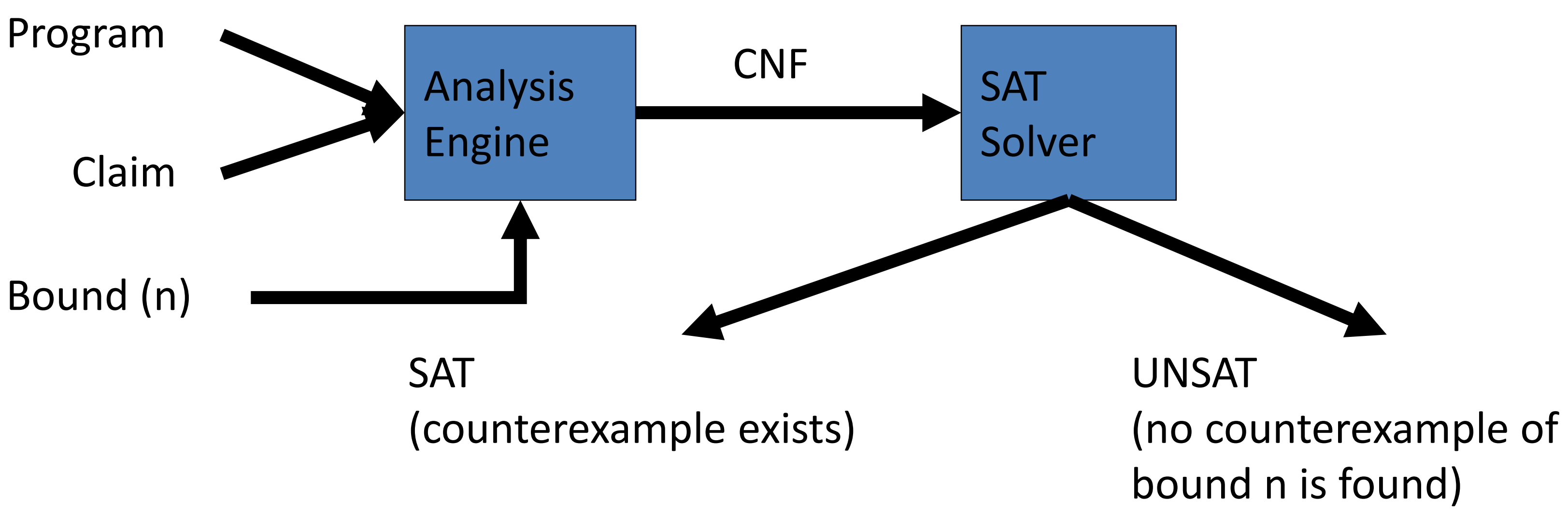
CBMC: C Bounded Model Checker

- Developed at CMU by Daniel Kroening et al.
- Available at: http://www.cs.cmu.edu/~modelcheck/cbmc/
- Supported platafoms: Windows (requires VisualStudio’s` CL), Linux
- Provides a command line and Eclipse-based interfaces

- Known to scale to programs with over 30K LOC
- Was used to find previously unknown bugs in MS Windows device drivers

What about loops?!

- SAT Solver can only explore finite length executions!
- Loops must be bounded (i.e., the analysis is incomplete)



How does it work?

- Transform a programs into a set of equations

1. Simplify control flow
2. Unwind all of the loops
3. Convert into Single Static Assignment (SSA)
4. Convert into equations
5. Bit-blast
6. Solve with a SAT/SMT Solver
7. Convert SAT assignment into a counterexample

Control Flow Simplifications

- All side effect are removal
  - `e.g., j=i++` becomes `j=i;i=i+1`
- Control Flow is made explicit
  - `continue`, `break` replaced by `goto`
- All loops are simplified into one form
  - `for`, `do while` replaced by `while`

Loop Unwinding

- All loops are unwound
  - can use different unwinding bounds for different loops
  - to check whether unwinding is sufficient special “unwinding assertion” claims are added
- If a program satisfies all of its claims and all unwinding assertions then it is correct!
- Same for backward `goto` jumps and recursive functions

Loop Unwinding

```
void f(...) {
  ...
  while(cond) {
    Body;
  }
  Remainder;
}
```

- while() loops are unwound iteratively
- Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    while(cond) {
      Body;
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Loop Unwinding

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond) {
        Body;
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        while(cond) {
          Body;
        }
      }
    }
  }
  Remainder;
}
```

while() loops are unwound iteratively

Break / continue replaced by goto

Assertion inserted after last iteration: violated if program runs longer than bound permits

Unwinding assertion

```
void f(...) {
  ...
  if(cond) {
    Body;
    if(cond) {
      Body;
      if(cond) {
        Body;
        assert(!cond);

      }
    }
  }
  Remainder;
}
```

Unwinding assertion

while() loops are unwound iteratively

Break / continue replaced by goto

Assertion inserted after last iteration: violated if program runs longer than bound permits

Positive correctness result!

Example: Sufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 2)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 2) {
    j = j + 1;
    if(j <= 2) {
      j = j + 1;
      if(j <= 2) {
        j = j + 1;
        assert(!(j <= 2));
      }
    }
  }
  Remainder;
}
```

Example: Insufficient Loop Unwinding

```
void f(...) {
  j = 1
  while (j <= 10)
    j = j + 1;
  Remainder;
}
```

unwind = 3

```
void f(...) {
  j = 1
  if(j <= 10) {
    j = j + 1;
    if(j <= 10) {
      j = j + 1;
      if(j <= 10) {
        j = j + 1;
        assert(!(j <= 10));
      }
    }
  }
  Remainder;
}
```

Transforming Loop-Free Programs Into Equations (1)

- Easy to transform when every variable is only assigned once!

Program

```
x = a;
y = x + 1;
z = y - 1;
```

Constraints

x = a &&
y = x + 1 &&
z = y – 1 &&

Transforming Loop-Free Programs Into Equations (2)

- When a variable is assigned multiple times,
- use a new variable for the RHS of each assignment

Program		SSA Program

`x=x+y;`
`x=x*2;`
`a[i]=100;`
	$\xrightarrow{\rho}$	$x_1 = x_0 + y_0;$
		$x_2 = x_1 * 2;$
		$a_1[i_0] = 100;$

What about conditionals?

Program

```
if (v)
  x = y;
else
  x = z;

w = x;
```

$\xrightarrow{\rho}$

SSA Program

$\texttt{if (}v_0\texttt{)}$

$\quad x_0 = y_0;$

$\texttt{else}$

$\quad x_1 = z_0;$

$w_1 = x??;$

What should ‘x’ be?

What about conditionals?

Program

```
if (v)
  x = y;
else
  x = z;

w = x;
```

$$\xrightarrow{\rho}$$

SSA Program

$$
\begin{array}{l}
\texttt{if } (v_0) \\
\quad x_0 = y_0; \\
\texttt{else} \\
\quad x_1 = z_0; \\
x_2 = v_0 \;?\; x_0 : x_1; \\
w_1 = x_2
\end{array}
$$

- For each join point, add new variables with selectors

Adding Unbounded Arrays

$$v_\alpha[a] = e \quad \xrightarrow{\rho} \quad v_\alpha = \lambda i : \begin{cases} \rho(e) & : \ i = \rho(a) \\ v_{\alpha-1}[i] & : \ \text{otherwise} \end{cases}$$

- Arrays are updated “whole array” at a time

A[1] = 5;	$A_1 = \lambda\ i : i == 1\ ?\ 5 : A_0[i]$
A[2] = 10;	$A_2 = \lambda\ i : i == 2\ ?\ 10 : A_1[i]$
A[k] = 20;	$A_3 = \lambda\ i : i == k\ ?\ 20 : A_2[i]$

Examples:

$A_2[2] == 10$ $\quad$ $A_2[1]==5$ $\quad$ $A_2[3] == A_0[3]$

$A_3[2] == (k==2\ ?\ 20 : 10)$

Uses only as much space as there are uses of the array!

Example

```
int main() {
  int x, y;
  y=8;
  if(x)
    y--;
  else
    y++;

  assert
     (y==7 ||
     y==9);
}
```

ρ

```
int main() {
  int x, y;
  y1=8;
  if(x0)
    y2=y1-1;
  else
    y3=y1+1;
  y4= x0 ?y2:y3;
  assert
     (y4==7 ||
     y4==9);
}
```

$$
\begin{aligned}
(\quad & y_1 = 8 \\
\wedge \quad & y_2 = y_1 - 1 \\
\wedge \quad & y_3 = y_1 + 1 \\
\wedge \quad & y_4 = x_0 \,?\, y_2 : y_3 \;) \\
\Longrightarrow \quad & (y_4 = 7 \vee y_4 = 9)
\end{aligned}
$$

Pointers

- While unwinding, record right hand side of assignments to pointers
- This results in very precise points-to information
  - Separate for each pointer
  - Separate for each <u>instance</u> of each program location
- Dereferencing operations are expanded into case-split on pointer object (not: offset)
  - Generate assertions on offset and on type

Deciding Bit-Vector Logic with SAT

- Pro: all operators modeled with their precise semantics
- Arithmetic operators are flattened into circuits
  - Not efficient for multiplication, division
  - Fixed-point for `float`/`double`
- Unbounded arrays
  - Use uninterpreted functions to reduce to equality logic
  - Similar implementation in UCLID
  - But: <u>Contents</u> of array are interpreted
- Problem: SAT solver happy with first satisfying assignment that is found. <u>Might not look nice</u>.

Example

```c
void f (int a, int b, int c)
{
  int temp;
  if (a > b) {
    temp = a; a = b; b = temp;
  }
  if (b > c) {
    temp = b; b = c; c = temp;
  }
  if (a < b) {
    temp = a; a = b; b = temp;
  }
  assert (a<=b && b<=c);
}
```

CBMC

```
State 1-3
  a=-8193 (11111111111111111101111111111111)
  b=-402 (11111111111111111111111001101110)
  c=-2080380800 (10000011111111111110100010…)
  temp=0 (00000000000000000000000000000000)

State 4 file sort.c line 10
  temp=-402 (11111111111111111111111001101110)

State 5 file sort.c line 11
  b=-2080380800 (10000011111111111110100010…)

State 6 file sort.c line 12
  c=-402 (11111111111111111111111001101110)

Failed assertion: assertion file
sort.c line 19
```

Problem (I)

- Reason: SAT solver performs DPLL backtracking search
- Very first satisfying assignment that is found is reported
- Strange values artifact from bit-level encoding
- Hard to read
- Would like nicer values

Problem (II)

- Might not get shortest counterexample!
- Not all statements that are in the formula actually get executed
- There is a variable for each statement that decides if it is executed or not (conjunction of `if`-guards)
- Counterexample trace only contains assignments that are actually executed
- The SAT solver picks some…

Example

```
void f (int a, int b,
        int c)
{
  if(a)
  {
    a=0;
    b=1;
  }

  assert(c);
}
```

CBMC

```
{-1} b_1#2 == (a_1#0?b_1#1:b_1#0)
{-2} a_1#2 == (a_1#0?a_1#1:a_1#0)
{-3} b_1#1 == 1
{-4} a_1#1 == 0
{-5} \guard#1 == a_1#0
{-6} \guard#0 == TRUE
|-----------------------
{1} c_1#0
```

from SSA

assign-
ments

Example

```
void f (int a, int b,
        int c)
{
  if(a)
  {
    a=0;
    b=1;
  }

  assert(c);
}
```

CBMC

```
State 1-3
  a=1  (00000000000000000000000000000001)
  b=0  (00000000000000000000000000000000)
  c=0  (00000000000000000000000000000000)

State 4 file length.c line 5
  a=0  (00000000000000000000000000000000)

State 5 file length.c line 6
  b=1  (00000000000000000000000000000001)

Failed assertion: assertion
file length.c line 11
```

Basic Solution

- Counterexample length typically considered to be most important
  - e.g., SPIN iteratively searches for shorter counterexamples
- Phase one: Minimize length

$$min \sum_{g \in G} l_g \cdot l_w$$

- l_g: Truth value (0/1) of guard,
  l_w: Weight = number of assignments

- Phase two: Minimize values

Pseudo Boolean Solver (PBS)

- Input:
  - CNF constraints
  - Pseudo Boolean constraints
    - 2x + 3y + 6z <= 7, where x, y, z are Boolean variables
  - Pseudo Boolean objective function
- Output:
  - Decision (SAT/UNSAT)
  - Optimizatioin (Minimize/Maximize an objective function)
- Some implementations:
  - PBS http://www.eecs.umich.edu/~faloul/Tools/pbs
  - MiniSat+ (from MiniSat web page)

Example

```
void f (int a, int b, int c)
{
  int temp;
  if (a > b) {
    temp = a; a = b; b = temp;
  }
  if (b > c) {
    temp = b; b = c; c = temp;
  }
  if (a < b) {
    temp = a; a = b; b = temp
  }
  assert (a<=b && b<=c);
}
```

CBMC

+

Mini-
mization

```
State 1-3
  a=0  (00000000000000000000000000000000)
  b=0  (00000000000000000000000000000000)
  c=-1  (11111111111111111111111111111111)
  temp=0  (00000000000000000000000000000000)

State 4 file sort.c line 10
  temp=0  (00000000000000000000000000000000)

State 5 file sort.c line 11
  b=-1  (11111111111111111111111111111111)

State 6 file sort.c line 12
  c=0  (00000000000000000000000000000000)

Failed assertion: assertion file
sort.c line 19
```

Modeling with CBMC (1)

- CBMC provides 2 modeling (not in ANSI-C) primitives

- `xxx nondet_xxx ()`
- Returns a non-deterministic value of type `xxx`
- `int nondet_int (); char nondet_char ();`

- Useful for modeling external input, unknown environment, library functions, etc.

Using nondet for modeling

- Library spec:
- "foo is given non-deterministically, but is taken until returned"
- CMBC stub:

```
int nondet_int ();
int is_foo_taken = 0;
int grab_foo () {
  if (!is_foo_taken)
    is_foo_taken = nondet_int ();
  return is_foo_taken; }
```

```
int return_foo ()
{ is_foo_taken = 0; }
```

Assume-Guarantee Reasoning (1)

- Is `foo` correct?

Check by splitting on the argument of `foo`

```
int foo (int* p) { … }
void main(void) {
  …
  foo(x);
  …
  foo(y);
  …
}
```

Assume-Guarantee Reasoning (2)

- (A) Is `foo` correct assuming `p` is not NULL?

```
int foo (int* p) { __CPROVER_assume(p!=NULL); … }
```

(G)Is `foo` guaranteed to be called with a non-NULL argument?

```
void main(void) {
  …
  assert (x!=NULL);// foo(x);
  …
  assert (y!=NULL); //foo(y);
  …}
```

Dangers of unrestricted assumptions

- Assumptions can lead to vacuous satisfaction

```
if (x > 0) {
  __CPROVER_assume (x < 0);
  assert (0); }
```

This program is passed by CMBMC!

Assume must either be checked with assert or used as an idiom:

```
x = nondet_int ();
y = nondet_int ();
__CPROVER_assume (x < y);
```

Summary CBMC

- Bounded model checking is effective for bug finding
- Tricky points
  - PL semantics
  - Procedure Summaries
  - Pointers
  - Loops

Alloy Analyzers

Alloy in one slide

- Invented at MIT by Daniel Jackson (starting around 2000)
- Textual, object-oriented modeling language based on first-order relational logic
- “Light-weight formal methods” approach, fully automated bounded analysis using SAT
- Hundreds of case studies, taught in many universities

Alloy Goals

- Apply bounded model checking to software designs
  - UML
  - Z
- A user friendly modeling language
  - First order logic + transitive closure + many syntactical extensions
  - Graphical user interface
    - Displays counterexamples in a user friendly way

First Order Logic

- Vocabulary V=<R, F, C>
  - Set of relation symbols R each with a fixed arity
  - Set of function symbols F each with a fixed arity
  - Set of constant symbols C
- $F ::= \exists X.\ F \mid \forall X.\ F \mid F \vee F \mid \neg F \mid r(\underline{t}) \mid t_1 = t_2$
- $t ::= f(\underline{t}) \mid c \mid X$
- Example:
  - $\forall u: \neg edge(u, u)$
  - $\forall u: node(u) \rightarrow \exists cl: color(cl) \wedge cl(u,cl)$
  - $\forall u_1, u_2, c: node(u_1) \wedge node(u_2) \wedge edge(u_1, u_2) \wedge cl(u_1, c) \rightarrow \neg cl(u_2, c)$

Model $M = <U, \iota>$

- A set of elements (universe) U
- For each constant $c \in C$, $\iota(c) \in U$
- For each function $f \in F$ of arity k
  $\iota(f) \subseteq U^k \rightarrow U$
- For each relation $r \in R$ of arity k,
  $\iota(r) \subseteq U^k$

Formula Satisfaction

- A first order formula over vocabulary V
- A model $M = \langle U, \iota \rangle$ for V
- An assignment $A: Var \rightarrow U$
- $[A] : Term \rightarrow U$ is inductively defined
  - $[A](X) = A(X)$
  - $[A](c) = \iota(c)$
  - $[A](f(t_1, t_2, \ldots, t_k) = \iota(f)([A](t_1), [A](t_2), \ldots, [A](t_k))$

Formula Satisfaction

- A first order formula over vocabulary V
- A model M=<U, ι> for V
- An assignment A: Var $\rightarrow$ U
- A formula φ over V
- M, A $\models \varphi$ is defined inductively
  - M, A $\models r(t_1, t_2, \ldots, t_k)$ if $<[A](t_1), [A](t_2), \ldots, [A](t_k)> \in \iota(r)$
  - M, A $\models t_1 = t_2$ if $[A](t_1)=[A](t_2)$
  - M, A $\models \neg \varphi$ if not M, A $\models \varphi$
  - M, A $\models \varphi_1 \vee \varphi_2$ if M, A $\models \varphi_1$ or M, A $\models \varphi_2$
  - M, A $\models \exists X.\ \varphi$ if there exists $u \in U$ such that M, $A[X \mapsto u] \models \exists X.\ \varphi$

The SAT problem for first order logic

- Given a first order formula φ do there exist a model M and assignment such that $M, A \models \varphi$
- Example 1:
  - $\forall u$: node(u) $\rightarrow \exists cl$: color(cl) $\wedge$ cl(u,cl)
  - $\forall u_1, u_2, c$: node(u_1) $\wedge$ node(u_2) $\wedge$ edge(u_1, u_2) $\wedge$ cl(u_1, c) $\rightarrow \neg$ cl(u_2, c)

The SAT problem for first order logic

- Given a first order formula φ do there exist a model M and assignment such that $M, A \vDash \varphi$
- Example 2:
  - $\forall X.\ r(X, X)$
  - $\forall X, Y.\ r(X, Y) \wedge r(Y, X) \rightarrow X = Y$
  - $\forall X, Y, Z.\ r(X, Y) \wedge r(Y, Z) \rightarrow r(X, Z)$
  - $\forall X.\ \exists Y.\ r(X, Y) \wedge X \neq Y$

The SAT problem for first order logic

- Given a first order formula φ do there exist a model M and assignment such that $M, A \vDash \varphi$
- Undecidable in general
- Decidable cases
  - Unary relations
  - EPR formulas
  - Presburger formulas
  - The size of M is known (Alloy)

A Tour of Alloy

Shahar Maoz

Statics: exploring states

module tour/addressBook1

sig Name, Addr {}

sig Book {

addr: Name->**lone** Addr }

Name(*), Addr(*), Book(*)

disjoint Name, Addr, Book

addr(*, *, *)

$\forall$X, Y, Z: X.addr(Y, Z) → Book(X) /\ Name(Y) /\ Addr(Z)

$\forall$X, Y, Z1, Z2: X.addr(Y, Z1) /\ X.addr(Y, Z2) → Z1 = Z2

Statics: exploring states

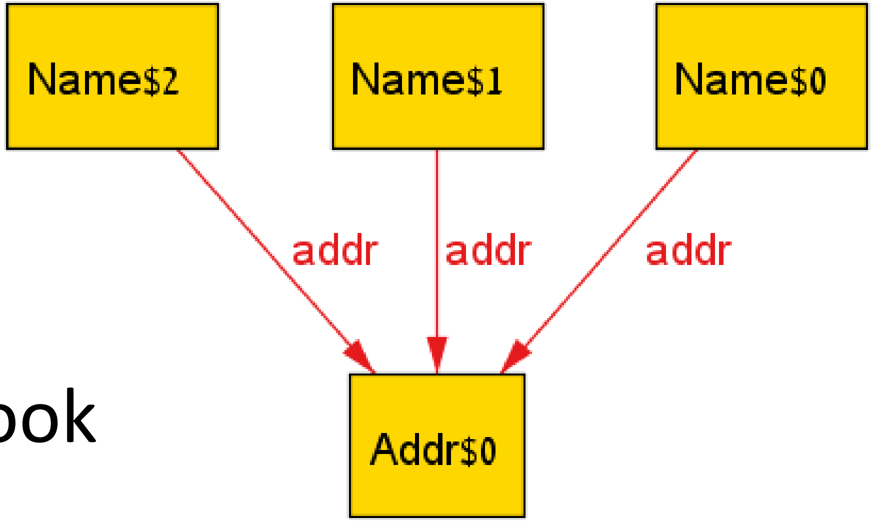
```
module tour/addressBook1
sig Name, Addr {}
sig Book {
    addr: Name->lone Addr }

pred show () {}
run show for 3 but 1 Book
```

Statics: exploring states

module tour/addressBook1
sig Name, Addr {}
sig Book {
    addr: Name->**lone** Addr }

pred show (b: Book) {
    #b.addr > 1}

run show **for** 3 **but** 1 Book

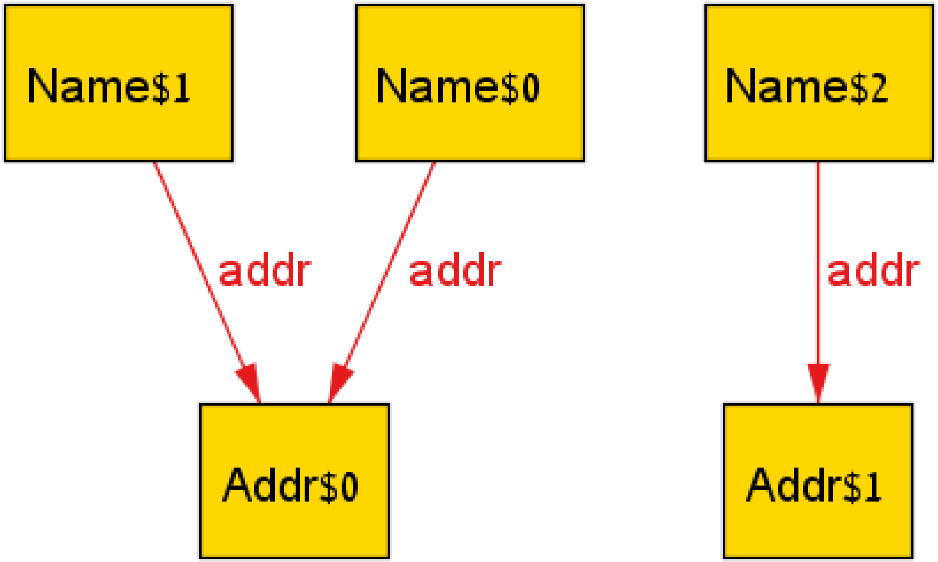


Statics: exploring states

```
module tour/addressBook1
sig Name, Addr {}
sig Book {
    addr: Name->lone Addr }

pred show (b: Book) {
    #b.addr > 1
    some n: Name | #n.(b.addr) > 1 }


run show for 3 but 1 Book
```

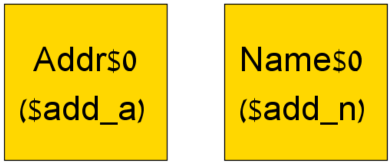
Statics: exploring states

```
module tour/addressBook1
sig Name, Addr {}
sig Book {
    addr: Name->lone Addr }

pred show (b: Book) {
    #b.addr > 1
    // some n: Name | #n.(b.addr) > 1
    #Name.(b.addr) > 1 }

run show for 3 but 1 Book
```



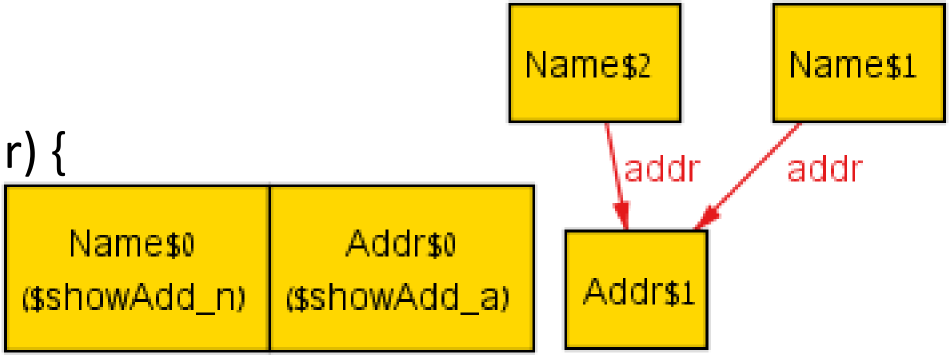
Dynamics: adding operations

```
module tour/addressBook1
sig Name, Addr {}
sig Book {
    addr: Name->lone Addr }

pred add (b, b': Book, n: Name, a: Addr) {
    b'.addr = b.addr + n -> a }

run add for 3 but 2 Book
```

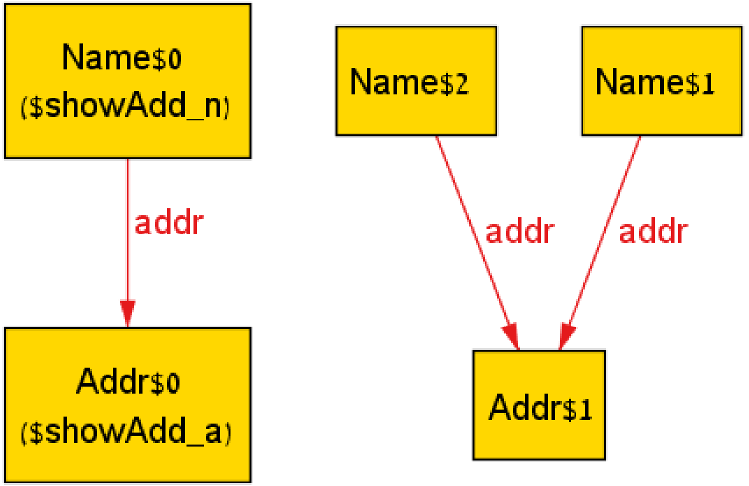


Dynamics: adding operations

```
module tour/addressBook1
...
pred add (b, b': Book, n: Name, a: Addr) {
    b'.addr = b.addr + n -> a }
```




```
pred showAdd (b, b': Book, n: Name, a: Addr) {
    add (b, b', n, a)
    #Name.(b'.addr) > 1 }

run showAdd for 3 but 2 Book
```



Dynamics: adding some more operations

module tour/addressBook1

…

pred add (b, b': Book, n: Name, a: Addr) {
    b'.addr = b.addr + n -> a }

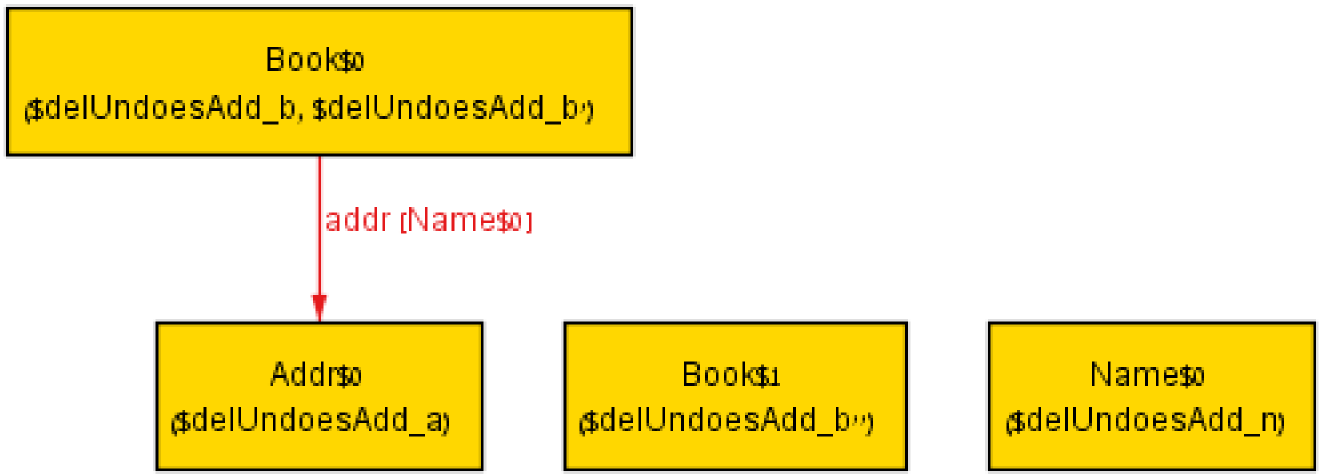
pred del (b, b': Book, n: Name) {
    b'.addr = b.addr - n ->Addr }

fun lookup (b: Book, n: Name): **set** Addr {
    n. (b.addr) }

Adding an assertion

```
module tour/addressBook1
...
pred add (b, b': Book, n: Name, a: Addr) {
    b'.addr = b.addr + n -> a }

pred del (b, b': Book, n: Name) {
    b'.addr = b.addr - n ->Addr }

assert delUndoesAdd {
    all b,b',b": Book, n: Name, a: Addr |
        add (b,b',n,a) and del (b',b",n) implies b.addr = b".addr }

check delUndoesAdd for 3
```

Counterexample found



assert delUndoesAdd {
    all b,b',b": Book, n: Name, a: Addr |
        add (b,b',n,a) **and** del (b',b",n) **implies** b.addr = b".addr }

check delUndoesAdd **for** 3

Assertion fixed

```
assert delUndoesAdd {
    all b,b',b": Book, n: Name, a: Addr |
        no n.(b.addr)  and
        add (b,b',n,a) and del (b',b",n) implies b.addr = b".addr }

check delUndoesAdd for 3
```

Checking the assertion in a larger scope

```
assert delUndoesAdd {
    all b,b',b": Book, n: Name, a: Addr |
        no n.(b.addr)  and
        add (b,b',n,a) and del (b',b",n) implies b.addr = b".addr }

check delUndoesAdd for 10 but 3 Book

check delUndoesAdd for 40 but 3 Book
```

Small scope hypothesis

- We still haven’t proved the assertion to be valid, but intuitively it seems unlikely that, if there is a problem, it can’t be shown in a counterexample with 40 names and addresses

- Small scope hypothesis: Most flaws in models can be illustrated by small instances, since they arise from some shape being handled incorrectly, and whether the shape belongs to a large or a small instance makes no difference. So if the analysis considers **all** small instances, most flaws will be revealed.

- This hypothesis is a fundamental premise that underlies Alloy’s analysis

Some additional assertions

```
assert addIdempotent {
  all b,b',b": Book, n: Name, a: Addr |
    add (b,b',n,a) and add (b',b",n,a)
    implies b'.addr = b".addr }

assert addLocal {
  all b,b': Book, n,n': Name, a: Addr |
    add (b,b',n,a) and n != n'
    implies lookup (b,n') = lookup (b',n') }
```

Summary

- So far we have seen
  - Signatures, fields
  - Predicates, assertions, functions
  - Run and check commands
  - The small scope hypothesis
- Missing Alloy features
  - Object-oriented inheritance
  - Transitive closure
  - Facts

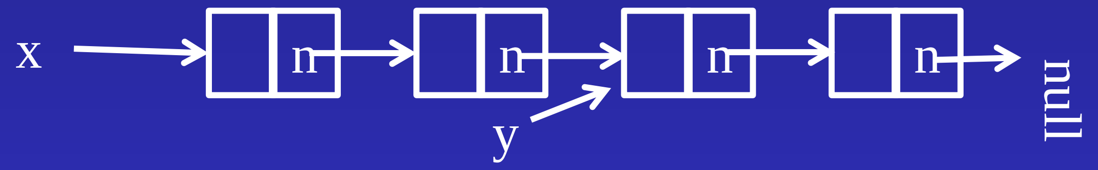
First Order Logic +TC

- Vocabulary V=<R, F, C>
  - Set of relation symbols R each with a fixed arity
  - Set of function symbols F each with a fixed arity
  - Set of constant symbols C
- $F ::= TC(X, Y)(W, Z).\ F \mid \exists X.\ F$
  $\mid F \vee F \mid \neg F \mid r(\underline{t}) \mid t_1 = t_2$
- $t ::= f(\underline{t}) \mid c \mid X$
- Example:
  - $\forall X, Y.\ edge^*(X, Y) \leftrightarrow TC(X,Y)(W, Z).edge(W,Z)$

Program Termination

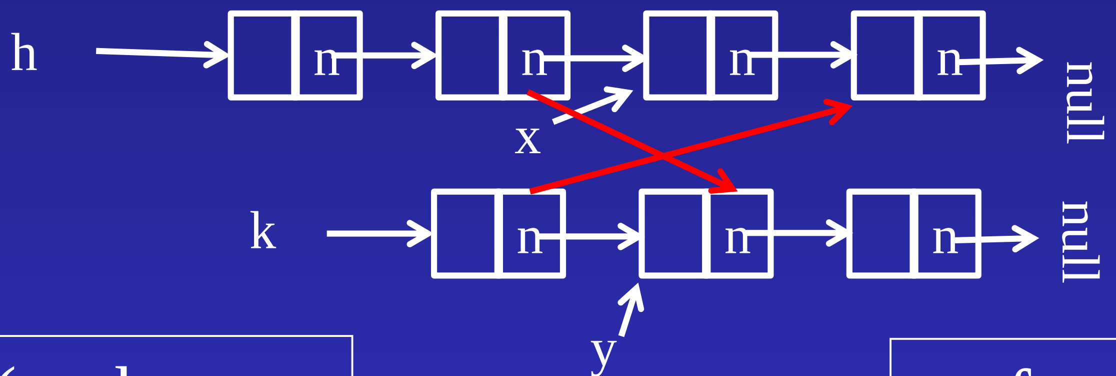
{n*(x, y)}

```
traverse(Node x, Node y) {
    for (t =x; t != y ; t = t.n) {
        ...
    }
}
```



Disjoint Parallelism

$\{\forall\alpha\colon\ \alpha \neq null \rightarrow \neg(n^*(h, \alpha) \wedge n^*(k, \alpha))\}$




```
for (x =h;
      x != null;
      x = x.n) {

   …
}
```

||

```
for (y=k;
      y != null;
      y = y.n) {

   …
}
```

Selected references Alloy

- D. Jackson. “Software Abstractions: Logic, Language, and Analysis”, MIT Press, 2006.

- D. Jackson. “Automating First-Order Relational Logic”, FSE 2000, ACM, pp. 130-139.

Some Suggested Projects

- BMC for a cool language (Python)
- Apply Alloy to an interesting domain
  - Simple distributed protocols
    - Leader election
    - …
- Apply Rosette

Summary Bounded Model Checking

- Effective technique
- Deployed by some companies
- Scaling is an issue