Revisiting SMTs and BMCs

Mooly Sagiv
Plan

• From Propositional Logic to SMT
• BMC Using SMT
• Alloy (Shahar Maoz)
SAT Solvers

Propositional Formula

SAT Solver (DPLL)

YES

Satisfying Assignment

NO

A Proof of Unsatisfiability
Project with the Minisat solver

- Generate proof trees for Unsat formulas
Limitations of propositional logic

• Hard to deal with unbounded nature of systems/code
  – Integers
  – Arrays
  – Dynamically allocated data
  – Sets
  – Relations
  – ...
• The size of input formulas can be huge
First Order Logic

- Vocabulary $V=\langle R, F, C \rangle$
  - Set of relation symbols $R$ each with a fixed arity
  - Set of function symbols $F$ each with a fixed arity
  - Set of constant symbols $C$

- $F ::= \exists X. F \mid F \lor F \mid \neg F \mid r(t) \mid t_1 = t_2$
- $t ::= c \mid X \mid f(t)$
- $F_1 \land F_2 \equiv \neg(\neg F_1 \lor \neg F_2)$
- $\forall X. F \equiv \neg \exists X. \neg F$
Example 1: Graph Coloring

- $\forall u. \neg\text{edge}(u, u)$
- $\forall u. \text{node}(u) \rightarrow \exists \text{cl. color(cl)} \land \text{cl}(u, \text{cl})$
- $\forall u_1, u_2, c. \text{node}(u_1) \land \text{node}(u_2) \land \text{edge}(u_1, u_2) \land \text{cl}(u_1, c) \rightarrow \neg \text{cl}(u_2, c)$
Example 2: Array Program

- $P \Rightarrow \forall i: f(i) \geq f(\text{succ}(i))$
Example 3: Hidden Integers

- \( \forall X. \ r(X, X) \)
- \( \forall X, Y. \ r(X, Y) \land r(Y, X) \implies X = Y \)
- \( \forall X, Y, Z. \ r(X, Y) \land r(Y, Z) \implies r(X, Z) \)
- \( \forall X. \ \exists Y. \ r(X, Y) \land X \neq Y \)
Model $M = \langle U, \iota \rangle$

- 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 $k$,
  \[ \iota(r) \subseteq U^k \]
Model for Graph Coloring

- \( \forall u. \neg \text{edge}(u, u) \)
- \( \forall u. \text{node}(u) \rightarrow \exists cl: \text{color}(cl) \land \text{cl}(u, cl) \)
- \( \forall u_1, u_2, c. \text{node}(u_1) \land \text{node}(u_2) \land \text{edge}(u_1, u_2) \land \text{cl}(u_1, c) \rightarrow \neg \text{cl}(u_2, c) \)

Vocabulary

- \( C = \{\} \)
- \( F = \{\} \)
- \( R = \{\text{edge}(*, *), \text{node}(*), \text{color}(*), \text{cl}(*, *)\} \)

U = \{a, b, c, green, blue\}

node = \{a, b, c\}

color = \{green, blue\}

e边 = \{<a, b>, <b, a>, <a, c>, <c, a>\}

color = \{<a, blue>, <b, green>, <c, blue>\}

Diagram: [Description of the diagram is not included in the text.]
Model for Hidden Integers

- $\forall X. \ r(X, X)$
- $\forall X, Y. \ r(X, Y) \land r(Y, X) \implies X = Y$
- $\forall X, Y, Z. \ r(X, Y) \land r(Y, Z) \implies r(X, Z)$
- $\forall X. \ \exists Y. \ r(X, Y) \land X \neq Y$
Model for Array Programs

- $P \Rightarrow \forall i. f(i) \geq f(\text{succ}(i))$
Formula Satisfaction/Assignments

• A first order formula over vocabulary V
• A model $M = \langle U, \iota \rangle$ for V
• An assignment $A: \text{Var} \rightarrow U$
• $[A]: \text{Term} \rightarrow U$ is inductively defined
  – $[A](X) = A(X)$
  – $[A](c) = \iota(c)$
  – $[A](f(t_1, t_2, ..., t_k)) = \iota(f([A](t_1), [A](t_2), ..., [A](t_k)))$
Formula Satisfaction

- A first order formula over vocabulary V
- A model M=<U, τ> for V
- An assignment A: Var → U
- A formula φ over V
- M, A ⊨ φ is defined inductively
  - M, A ⊨ r(t₁, t₂, ..., tₖ) if <[A](t₁), [A](t₂), ..., [A](tₖ)> ∈ τ(r)
  - M, A ⊨ t₁ = t₂ if [A](t₁)=[A](t₂)
  - M, A ⊨ ¬φ if not M, A ⊨ φ
  - M, A ⊨ φ₁ ∨ φ₂ if M, A ⊨ φ₁ or M, A ⊨ φ₂
  - M, A ⊨ ∃X. φ if there exists u ∈ U such that M, A[X ← u] ⊨ ∃X. φ
Model for Graph Coloring

- $\forall u. \neg \text{edge}(u, u)$
- $\forall u. \text{node}(u) \rightarrow \exists \text{cl}. \text{color}($cl$) \land \text{cl}(u, \text{cl})$
- $\forall u_1, u_2, c. \text{node}(u_1) \land \text{node}(u_2) \land \text{edge}(u_1, u_2) \land \text{cl}(u_1, c) \rightarrow \neg \text{cl}(u_2, c)$

Vocabulary

- $C = \{\}$
- $F = \{\}$
- $R = \{\text{edge}(*, *), \text{node}(*), \text{color}(*), \text{cl}(*, *)\}$

$U = \{a, b, c, \text{green}, \text{blue}\}$

- $\text{node} = \{a, b, c\}$
- $\text{color} = \{\text{green}, \text{blue}\}$

$\text{edge} = \{<a, b>, <b, a>, <a, c>, <c, a>\}$

$\text{color} = \{<a, \text{blue}>, <b, \text{green}>, <c, \text{blue}>\}$
The SAT problem for first order logic

• Given a first order formula $\varphi$ do there exist a model $M$ and assignment such that $M, A \models \varphi$
SAT Solvers for First Order Logic

First Order Formulas

FO SAT Solver (MACE, Z3)

YES

Model+Assignment

NO

A Proof of Unsatisfiability
The SAT problem for first order logic

• Given a first order formula $\varphi$ do there exist a model $M$ and assignment such that $M, A \models \varphi$

• Example 1:
  
  $\forall u: \text{node}(u) \rightarrow \exists \text{cl. color}(\text{cl}) \land \text{cl}(u, \text{cl})$
  $\forall u_1, u_2, c: \text{node}(u_1) \land \text{node}(u_2) \land \text{edge}(u_1, u_2) \land \text{cl}(u_1, c) \rightarrow \neg \text{cl}(u_2, c)$
The SAT problem for first order logic

• Given a first order formula $\varphi$ do there exist a model $M$ and assignment such that $M, A \models \varphi$

• Example 2:
  - $\forall X. r(X, X)$
  - $\forall X, Y. r(X, Y) \land r(Y, X) \rightarrow X = Y$
  - $\forall X, Y, Z. r(X, Y) \land r(Y, Z) \rightarrow r(X, Z)$
  - $\forall X. \exists Y. r(X, Y) \land X \neq Y$
The SAT problem for first order logic

• Given a first order formula $\varphi$ do there exist a model $M$ and assignment such that $M, A \models \varphi$
• Undecidable in general
• Decidable cases
  – Unary relations
  – Quantifier free formulas
  – Two variable formulas
  – EPR formulas [$Z3$]
  – The size of $M$ is known (Alloy, $Z3$)
Beyond First Order Logic

• Transitive Closure
• Many Sorted Logics
• Theories
First Order Logic +TC

• Vocabulary \( V=\langle R, F, C \rangle \)
  – Set of relation symbols \( R \) each with a fixed arity
  – Set of function symbols \( F \) each with a fixed arity
  – Set of constant symbols \( C \)

• \( F ::= \text{TC}(X, Y)(W, Z). F \mid \exists X. F \)
  \mid F \lor F \mid \neg F \mid r(t) \mid t_1 = t_2 \)

• \( t ::= f(t) \mid c \mid X \)

• Example:
  – \( \forall X, Y. \text{edge}^*(X, Y) \leftrightarrow \text{TC}(X,Y)(W, Z).\text{edge}(W,Z) \)
traverse(Node x, Node y) {
    for (t = x; t != y; t = t.n) {
        ...
    }
}

\{n^*(x, y)\}
Disjoint Parallelism

\( \{ \forall \alpha: \alpha \neq \text{null} \rightarrow \neg (h<n^*>\alpha \land k<n^*>\alpha) \} \)

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

```
for (y = k;
    y != null;
    y = y.n) {
    ...
}
```
Many Sorted First Order Logic

• Vocabulary $V=\langle S, R, F, C \rangle$
  – Finite set of Sorts
  – Set of relation symbols $R$ each with a fixed signature in $S^*$
  – Set of function symbols $F$ each with a fixed arity $S^* \rightarrow S$
  – Set of constant symbols $C$

• $F ::= \exists X: s. F \mid F \lor F \mid \neg F \mid r(t) \mid t_1 = t_2$

• $t ::= f(t) \mid c \mid X$
Example 1: Graph Coloring with sorts

- $\forall u : N. \neg \text{edge}(u, u)$
- $\forall u : N. \exists cl : C. \text{color}(cl) \land \text{cl}(u, cl)$
- $\forall u_1, u_2, c. \text{node}(u_1) \land \text{node}(u_2) \land \text{edge}(u_1, u_2) \land \text{cl}(u_1, c) \rightarrow \neg \text{cl}(u_2, c)$
Adding Theories

• Some properties are hard or awkward to describe in first order logic
• The SAT solver can be specialized for interesting domains called theories
• A theory includes
  – A special sort
  – A set of predefined functions and relations
• The SAT solver finds models in the theory
Presburger Arithmetic

• First order formulas over one sort with addition and equality

• Axioms
  – $\forall x. \neg (0 = x + 1)$
  – $\forall x, y. x + 1 = y + 1 \rightarrow x = y$
  – $\forall x. x + 0 = x$
  – $\forall x, y. x + (y + 1) = (x + y) + 1$

  – For every first order formula $P$ with one free variable $x$
    $$(P(0) \land \forall x(P(x) \rightarrow P(x + 1))) \rightarrow \forall y P(y)$$
Uninterpreted Functions (EUF)

\[ X = Y \implies f(X) = f(Y) \]
Theory of Arrays (Stores)

• read(write(v, i, e), j) =
  if i=j then e else read(v, j)
• write(v, i, read(v, i)) = v
• write(write(v, i, e), i, f) = write(v, i, f)
• i ≠j ⇒ write (write (v, i, e), j, f) =
  write (write (v, j, f), i, e)
Axioms of lists

\[
\begin{align*}
\text{car}(\text{cons}(X, Y)) &= X \\
\text{cdr}(\text{cons}(X, Y)) &= Y \\
\neg \text{atom}(X) &\Rightarrow \text{cons}(\text{car}(X), \text{cdr}(X)) = X \\
\neg \text{atom}(\text{cons}(X, Y)) &
\end{align*}
\]
Bounded Model Checking
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
What about loops?!  

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

Program → Analysis Engine → CNF → SAT/SMT Solver

- SAT (counterexample exists
- UNSAT (no counterexample of bound n is found)
How does it work?

- Transform a program 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 \texttt{goto} jumps and recursive functions
Loop Unwinding

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

while() loops are unwound iteratively

Break / continue replaced by goto
Loop Unwinding

```c
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;
}
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

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

unwind = 3
```

```c
void f(...) {
    j = 1
    if (j <= 2) {
        j = j + 1;
        if (j <= 2) {
            j = j + 1;
            assert (!(j <= 2));
        }
    }
    Remainder;
}
```
Example: Insufficient Loop Unwinding

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

```c
unwind = 3
```

```c
void f(...) {
    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

\[
\begin{align*}
x &= a; \\
y &= x + 1; \\
z &= y - 1; \\
\end{align*}
\]

Constraints

\[
\begin{align*}
x &= a \\
y &= x + 1 \\
z &= y - 1
\end{align*}
\]
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
\[
\begin{align*}
x &= x + y; \\
x &= x \times 2; \\
a[i] &= 100;
\end{align*}
\]

SSA Program
\[
\begin{align*}
x_1 &= x_0 + y_0; \\
x_2 &= x_1 \times 2; \\
a_1[i_0] &= 100;
\end{align*}
\]
What about conditionals?

**Program**

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

w = x;
```

**SSA Program**

```plaintext
if (v_0)
    x_0 = y_0;
else
    x_1 = z_0;

w_1 = x_1;
```

What should ‘x’ be?
What about conditionals?

Program

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

SSA Program

if (v_0)
    x_0 = y_0;
else
    x_1 = z_0;
x_2 = v_0 ? x_0 : x_1;
w_1 = x_2

• For each join point, add new variables with selectors
Adding Unbounded Arrays

\[ v_\alpha[a] = e \quad \rho \quad v_\alpha = \lambda i : \left\{ \begin{array}{ll} \rho(e) & : i = \rho(a) \\ v_\alpha-1[i] & : \text{otherwise} \end{array} \right. \]

- Arrays are updated “whole array” at a time

\[
\begin{align*}
A[1] &= 5; & A_1 &= \lambda i : i == 1 \ ? \ 5 : A_0[i] \\
A[2] &= 10; & A_2 &= \lambda i : i == 2 \ ? \ 10 : A_1[i] \\
A[k] &= 20; & A_3 &= \lambda i : i == k \ ? \ 20 : A_2[i] \\
\end{align*}
\]

Examples:

\[
\begin{align*}
A_3[2] &= (k==2 \ ? \ 20 : 10) \\
\end{align*}
\]

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

```c
int main() {
    int x, y;
    y=8;
    if(x)
        y--; // ← Simplified
    else
        y++;
    assert
        (y==7 ||
         y==9);
}
```

```c
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);
}
```

\[
\begin{align*}
\land & \quad y_1 = 8 \\
\land & \quad y_2 = y_1 - 1 \\
\land & \quad y_3 = y_1 + 1 \\
\land & \quad y_4 = x_0 \, ? \, y_2 : y_3
\end{align*}
\]

\[\implies (y_4 = 7 \lor y_4 = 9)\]
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 instance 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: Contents of array are interpreted
• Problem: SAT solver happy with first satisfying assignment that is found. Might not look nice.
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:

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

```c
int return_foo ()
{ is_foo_taken = 0; }
```
Assume-Guarantee Reasoning (1)

• Is foo correct?

Check by splitting on the argument of foo

```c
int foo (int* p) { ... }
void main(void) {
    ...
    foo(x);
    ...
    foo(y);
    ...
}
```
Assume-Guarantee Reasoning (2)

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

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

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

```c
void main(void) {
    ...  
    assert (x!=NULL); // foo(x);  
    ...  
    assert (y!=NULL); // foo(y);  
    ...}
```
Dangers of unrestricted assumptions

- Assumptions can lead to vacuous satisfaction

```c
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:

```c
x = nondet_int ();
y = nondet_int ();
__CPROVER_assume (x < y);
```
BMC for Software Synthesis

- Program with "wholes"
- Input/Output Spec
- Program without "wholes"

https://people.csail.mit.edu/asolar/
https://emina.github.io/rosette/
Summary CBMC

• Bounded model checking is effective for bug finding

• Tricky points
  – PL semantics
  – Procedure Summaries
  – Pointers
  – Loops
Projects with BMC

• Implement BMC for a subset of “interesting” programming language
  – Python with comprehensions
  – Javascript
  – Rust
  – Concurrency in Go
  – Concurrent data structures

• Apply synthesis to an interesting domain
  – Configuration languages
    • Puppet and Cheff
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
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(*, *, *)
∀X, Y, Z: X.addr(Y, Z) → Book(X) ∧ Name(Y) ∧ Addr(Z)
∀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**

```plaintext
module tour/addressBook

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/addressBook

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
module tour/addressBook1

sig Name, Addr {}

sig Book {
    addr: Name->\texttt{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 }

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
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

```plaintext
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
Selected references Alloy


Suggested Projects with Alloy

• Model distributed protocols
• Model security protocols
• Concurrent data structures
Summary Bounded Model Checking

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