Bounded Model Checking

Mooly Sagiv

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

• Arbitrary ANSI-C programs
  – With bitvector arithmetic, dynamic memory, pointers, ...

• Simple Safety Claims
  – Array bound checks (i.e., buffer overflow)
  – Division by zero
  – Pointer checks (i.e., NULL pointer dereference)
  – Arithmetic overflow
  – User supplied assertions (i.e., \texttt{assert (i > j)})
  – etc
Why use a SAT Solver?

• SAT Solvers are very efficient

• Analysis is completely automated

• Analysis as good as the underlying SAT solver

• Allows support for many features of a programming language
  – bitwise operations, pointer arithmetic, dynamic memory, type casts
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 few? 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 and Ed Clarke
• Available at: http://www.cprover.org/cbmc
  • On Ubuntu: apt-get install cbmc
  • with source code
• Supported platforms: Windows, Linux, OSX
• Has a command line, Eclipse CDT, and Visual Studio interfaces

• Scales to programs with over 30K LOC
• Found previously unknown bugs in MS Windows device drivers
CBMC: Supported Language Features

ANSI-C is a low level language, not meant for verification but for efficiency

Complex language features, such as

- Bit vector operators (shifting, and, or,...)
- Pointers, **pointer arithmetic**
- Dynamic memory allocation: malloc/free
- Dynamic data types: `char s[n]`
- Side effects
- `float`/`double`
- Non-determinism
Using CBMC from Command Line

• To see the list of claims
  
  cbmc --show-claims -I include file.c

• To check a single claim
  
  cbmc --unwind n --claim x -I include file.c

• For help
  
  cbmc --help
What about loops?!  

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

<table>
<thead>
<tr>
<th>Program</th>
<th>Analysis Engine</th>
<th>CNF</th>
<th>SAT Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Claim</td>
<td>SAT (counterexample exists)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bound (n)</td>
<td>UNSAT (no counterexample of bound n is found)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CBMC: Bounded Model Checker for C

A tool by D. Kroening/Oxford and Ed Clarke/CMU

C Program → Parser → goto-program → Static Analysis

SAFE → UNSAT → SAT solver → CNF

UNSAFE + CEX → CEX-gen → CNF

CNF-gen equations → CBMC
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

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

while() loops are unwound iteratively

Break / continue replaced by goto
Loop Unwinding

while() loops are unwound iteratively

Break / continue replaced by goto
Loop Unwinding

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

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

code:

```c
void f(...) {
  if (cond) {
    Body;
    if (cond) {
      Body;
      if (cond) {
        Body;
        assert (!cond);
      }
    }
  }
  Remainder;
}
```
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
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_?;
```

What should ‘x’ be?
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;
x_2 = v_0 ? x_0 : x_1;
w_1 = x_2
```

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

\[ v_\alpha[a] = e \quad \rho \]

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

- Arrays are updated “whole array” at a time

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

Examples:

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

Uses only as much space as there are uses of the array!
int main() {
    int x, y;
    y=8;
    if(x)
        y--;  
    else
        y++;
    assert
        (y==7 || y==9);
}

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);
}

( y1 = 8
∧ y2 = y1 - 1
∧ y3 = y1 + 1
∧ y4 = x0 ? y2 : y3 )
⇒ (y4 = 7 ∨ y4 = 9)
CBMC: Supported Language Features

ANSI-C is a low level language, not meant for verification but for efficiency

Complex language features, such as

- Bit vector operators (shifting, and, or, ...)
- Pointers, pointer arithmetic
- Dynamic memory allocation: malloc/free
- Dynamic data types: char s[n]
- Side effects
- float/double
- Non-determinism
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
Dynamic Objects

• Dynamic Objects:
  – `malloc` / `free`
  – Local variables of functions

• Auxiliary variables for each dynamically allocated object:
  – Size (number of elements)
  – Active bit
  – Type

• `malloc` sets size (from parameter) and sets active bit
• `free` asserts that active bit is set and clears bit
• Same for local variables: active bit is cleared upon leaving the function
From Programming to Modeling

• Extend C programming language with 3 modeling features

• Assertions
  – assert(e) – aborts an execution when e is false, no-op otherwise

```c
void assert (_Bool b) { if (!b) exit(); }
```

• Non-determinism
  – nondet_int() – returns a non-deterministic integer value

```c
int nondet_int () { int x; return x; }
```

• Assumptions
  – assume(e) – “ignores” execution when e is false, no-op otherwise

```c
void assume (_Bool e) { while (!e); }
```
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;
}

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 \texttt{foo} correct assuming \texttt{p} is not NULL?

\begin{verbatim}
int foo (int* p) { __CPROVER_assume(p!=NULL); ... }
\end{verbatim}

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

\begin{verbatim}
void main(void) {
  ...
  assert (x!=NULL); // foo(x);
  ...
  assert (y!=NULL); //foo(y);
  ...}
\end{verbatim}
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 $V=\langle R, F, C \rangle$
  – Set of relation symbols $R$ each with a fixed arity
  – Set of function symbols $F$ each with a fixed arity
  – Set of constant symbols $C$

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

• $t ::= f(t) \mid c \mid X$

• Example:
  – $\forall u: \neg\text{edge}(u, u)$
  – $\forall u: \text{node}(u) \rightarrow \exists \text{cl}: \text{color}(\text{cl}) \land \text{cl}(u, \text{cl})$
  – $\forall u_1, u_2, c: \text{node}(u_1) \land \text{node}(u_2) \land \text{edge}(u_1, u_2) \land \text{cl}(u_1, c) \rightarrow \neg\text{cl}(u_2, c)$
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 $k$
  $\iota(f) \subseteq U^k \to U$
- 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 $A$: $\text{Var} \to U$
- $[A] : \text{Term} \to U$ 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 $M=\langle U, \iota \rangle$ for V
- An assignment $A: \text{Var} \rightarrow U$
- A formula $\varphi$ 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 $M, A \models \varphi$
  - $M, A \models \varphi_1 \lor \varphi_2$ 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 $\varphi$ do there exist a model $M$ and assignment such that $M, A \models \varphi$

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

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

• 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
  – 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(*, *, *)

∀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
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/addressBook

sig Name, Addr {}

sig Book {
    addr: Name->\texttt{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
module tour/addressBook1

sig Name, Addr {}

sig Book {
    addr: Name->\textit{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

\[\text{assert delUndoesAdd} \{\]
\[\quad \text{all } b, b', b'': \text{Book}, n: \text{Name}, a: \text{Addr} \mid\]
\[\quad \text{add } (b, b', n, a) \text{ and del } (b', b'', n) \text{ implies } b.\text{addr} = b''.\text{addr} \}\]

\[\text{check delUndoesAdd for } 3\]
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

```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') }

Selected references Alloy


Summary Bounded Model Checking

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