Introduction to CBMC

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA  15213

Arie Gurfinkel
December 5, 2011

based on slides by
Daniel Kroening
Main Idea: Given a program and a claim use a SAT-solver to find whether there exists an execution that violates the claim.
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
A (very) simple example (1)

Program

```c
int x;
int y=8,z=0,w=0;
if (x)
    z = y - 1;
else
    w = y + 1;
assert (z == 7 ||
       w == 9)
```

Constraints

```c
y = 8,
z = x ? y - 1 : 0,
w = x ? 0 : y + 1,
z != 7,
w != 9
```

UNSAT
no counterexample
assertion always holds!
### A (very) simple example (2)

<table>
<thead>
<tr>
<th>Program</th>
<th>Constraints</th>
<th>SAT</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x;</td>
<td><strong>y = 8,</strong></td>
<td>counterexample found!</td>
</tr>
<tr>
<td>int y=8,z=0,w=0;</td>
<td>z = x ? y – 1 : 0,</td>
<td>y = 8, x = 1, w = 0, z = 7</td>
</tr>
<tr>
<td>if (x)</td>
<td>w = x ? 0 : y + 1,</td>
<td></td>
</tr>
<tr>
<td>z = y – 1;</td>
<td>z != 5,</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w = y + 1;</td>
<td>w != 9</td>
<td></td>
</tr>
<tr>
<td>assert (z == 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>w == 9)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
What about loops?!

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

- Developed at CMU by Daniel Kroening et al.
- Available at: http://www.cprover.org/cbmc
  - On Ubuntu: apt-get install cbmc
- Supported plataforms: 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
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
  
  \texttt{cbmc --show-claims -I include file.c}

- To check a single claim
  
  \texttt{cbmc --unwind n --claim x -I include file.c}

- For help
  
  - \texttt{cbmc --help}
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 Solver
7. Convert SAT assignment into a counterexample
CBMC: Bounded Model Checker for C
A tool by D. Kroening/Oxford

C Program → Parser → Static Analysis

SAFE → UNSAT → SAT solver

UNSAFE + CEX → CNF-gen

goto-program → equations

SAT → CEX-gen

CBMC
Control Flow Simplifications

- All side effect are removed
  - 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

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

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

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

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

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;
    }
    assert(! (j <= 2));
}

unwind = 3
```

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

```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;
}

unwind = 3
```

```c
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

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

Constraints

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

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

w = x;
```

SSA Program

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

w_1 = x??;
```

What should ‘x’ be?
What about conditionals?

Program

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

w = x;
```

SSA Program

```c
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

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

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

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:

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

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

```c
int main() {
    int x, y;
    y=8;
    if(x)
        y--;
    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);
}
```

( y1 = 8
∧ y2 = y1 - 1
∧ y3 = y1 + 1
∧ y4 = x0 ? y2 : y3 )

⇒ ( y4 = 7 ∨ y4 = 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

Pointer data type assumed to be part of bit-vector logic

- Consists of pair <object, offset>
void *p;
int i;
char c;
int main (void) {
    int input1, input2, z;
    p = input1 ? (void*)&i : (void*) &c;
    if (input2)
        z = *(int*)p;
    else
        z = *(char*)p; }

Pointer Typecast Example
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
Modeling with CBMC
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); }
```
Example

```c
int x, y;
void main (void)
{
    x = nondet_int ();
    assume (x > 10);
    y = x + 1;
    assert (y > x);
}
```

possible overflow assertion fails
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; }
}

void 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

This program is passed by CMBMC!

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

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

```c
x = nondet_int ();
y = nondet_int ();
__CPROVER_assume (x < y);
```
Example: Prophecy variables

```c
int x, y, v;
void main (void)
{
    v = nondet_int ();
    x = v;
    x = x + 1;
    y = nondet_int ();
    assume (v == y);
    assert (x == y + 1);
}
```

v is a *prophecy* variable:
- it guesses the future value of y

assume *blocks* executions with a wrong guess:
- syntactically: x is changed *before* y
- semantically: x is changed *after* y
Context-Bounded Analysis with CBMC
Context-Bounded Analysis (CBA)

Explore all executions of TWO threads that have at most R context-switches (per thread)
CBA via Sequentialization

1. Reduce concurrent program P to a sequential (non-deterministic) program P’ such that “P has error” iff “P’ has error”
2. Check P’ with CBMC
Key Idea

1. Divide execution into rounds based on context switches
2. Execute executions of each context separately, starting from a symbolic state
3. Run all parts of Thread 1 first, then all parts of Thread 2
4. Connect executions from Step 2 using assume-statements
Sequentialization in Pictures

Guess initial value of each global in each round

Execute task bodies

- $T_1$
- $T_2$

Check that initial value of round $i+1$ is the final value of round $i$
CBA Sequentialization in a Nutshel

Sequential Program for execution of R rounds (i.e., context switches):

1. for each global variable $g$, let $g[r]$ be the value of $g$ in round $r$
2. execute thread bodies sequentially
   - first thread 1, then thread 2
   - for global variables, use $g[r]$ instead of $g$ when running in round $r$
   - non-deterministically decide where to context switch
   - at a context switch jump to a new round (i.e., inc $r$)
3. check that initial value of round $r+1$ is the final value of round $r$
4. check user assertions
CBA Sequentialization

```c
var
int round; // current round
int g[R], i_g[R]; // global and initial global
Bool saved_assert = 1; // local assertions

void main ()
initShared ();
initGlobals ();
for t in [0,N) : // for each thread
    round = 0;
    T'(t);
checkAssumptions ();
checkAssertions ();

initShared ()
    for each global var g, g[0] = init_value (g);

initGlobals ()
    for r in [1,R): //for each round
        for each global g: g[r] = i_g[r] = nondet();

checkAssumptions ()
    for r in [0,R-1):
        for each global g:
            assume (g[r] == i_g[r+1]);

checkAssertions ()
    assert (saved_assert);
```
void \( T'_t() \)
\begin{align*}
\text{Same as } T_t, \text{ but each statement } & \text{‘st’ is replaced with:} \\
& \text{contextSwitch(t); st[g} \leftarrow g[\text{round}]]; \\
\text{and ‘assert(e)’ is replaced with:} \\
& \text{saved_assert} = e;
\end{align*}

void contextSwitch ()
\begin{align*}
\text{int oldRound;} \\
\text{if (nondet ()) return; } & \text{ // non-det do not context switch} \\
\text{oldRound} = \text{round;} \\
\text{round} = \text{nondet_int ();} \\
\text{assume (oldRound} & \text{ < round} \leq R-1);\end{align*}

For more details, see
Akash Lal and Tom Reps. “Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis”,