CBMC
(C Bounded Model Checker)

Daniel Kroening et al,
CMU

Presentation by: Emilia Katz
Motivation

• Checking hardware circuits versus a functional specification written as a program in C [original motivation]
  – The specification program is assumed to be thoroughly tested and debugged
  – Need to show that all the computations of the circuit are also possible computations of the specification program

• Checking properties of C programs [additional motivation]
Usages of CBMC

On its own:
• Hardware verification
• Checking Linux device drivers

As part of other tools:
• Finding bugs in library files of Microsoft, in Win 32
• Locating the source of an error in the code, using a counterexample that was found by some model checking
• Filtering out spurious counterexamples that appear as a result of abstraction-based verification techniques application
• Simultaneous checks of several configurations of the same program (2 bugs in a Linux kernel found)
• Test generation
• Proving equivalence of programs (recursive procedures) – CBMC is used as a back-end
Workflow – circuit verification

program \rightarrow P \rightarrow \varphi(P) \rightarrow \neg \varphi(P) \land \varphi(C) \rightarrow \neg \varphi(P) \land \varphi(C) \rightarrow SAT Solver

C \rightarrow \varphi(C) \rightarrow \neg \varphi(P) \land \varphi(C) \rightarrow \neg \varphi(P) \land \varphi(C) \rightarrow SAT Solver

CBMC

bit vector equations

don’t know

BUG!!!

counterexample

unsat.

sat.

no bug found (yet)

“OK”

crash
Today: Verification of programs

- Verifying arbitrary ANSI-C programs
  - pointers, pointer arithmetic
  - dynamic memory allocation (malloc/free) [=> data structures such as lists, graphs, …]
  - side effects (i++, etc.)
  - bit vector operators (shifting, and, or, …)
  - floating point arithmetic
  - dynamic data types (char s[n]), type casts
  - non-determinism
  - etc.
Properties checked

• Built-in safety checks:
  – array bounds checks (buffer overflow)
  – division by zero
  – pointer checks (null pointer dereference)
  – arithmetic overflow

• User-specified assertions
Workflow – C program verification

- Program: P
- Property: q

1. Translation to a formula: φ(P)
2. Negation: ¬q
3. CBMC
   - There exists a computation of P that violates q
4. SAT Solver
   - unsat. → “OK”
   - sat. → crash
     - don’t know
     - counterexample

- No bug found (yet)
Example 1

Program:
```
int x;
int y=8; z=0; w=0;
if (x)
  z = y-1;
else
  w = y+1;
```

Property checked:
```
(values – from the assertion point)
```
```
y=8,
z= x ? y-1: 0,
w= x ? 0: y+1,
z != 7,
w != 9
```

```
constraints from the computation
negation of asserted property (¬q)
```

No counterexample exists. Assertion always holds!
Example 1 – version 2

Program:

```c
int x;
int y=8; z=0; w=0;
if (x)
    z = y-1;
else
    w = y+1;
assert (z == 5 || w == 9)
```

Constraints:

At the assertion point,

\[
y=8,
z= x \ ? \ y-1: 0,
w= x \ ? \ 0: y+1,
z \neq 5,
w \neq 9
\]

Counterexample found!
\[y = 8, x = 1, w = 0, z = 7\]
What is Bounded Model Checking?

• Program model = transition system (as usual in M.C.)
• “Unwinding” computations of the program to a tree
• Analyzing the computations tree of the program
• Bound = depth of the counterexample search in the tree (maximal number of states in path prefixes analyzed) = depth of computations unwinding
Bounds are not bad!

• Minimal counterexamples possible: tries to find a counterexample in computations of length 0, 1, …, n sequentially (where n is the bound)

• Special, “bounded” semantics is defined for finite prefixes. Two cases are distinguished: prefix without loop and prefix containing all the loop states => loops can be analyzed

• Thus, bounded model checking can analyze infinite computations! (There is a finite number of states in the model, thus infinite computations are obtained by loops)

• If all the possible bounds are considered, the bounded semantics is equivalent to the unbounded one
C program verification - details

- **Program (P)**: Translation to a formula
  - $\varphi(P)$
  - $\varphi(P) \land \neg q$
  - SAT Solver
    - sat,
    - assignment
      - conversion
      - counterexample
    - crash
    - unsat.
  - no bug found (yet)
- **Property (q)**: Negation
  - $\neg q$

Phase 1:
- CBMC
- phase 2:
  - BUG!!!
Translating a program into a formula

1. Simplify control flow
   - remove side effects \( j = i++ \Rightarrow j=i; i=i+1 \)
   - make control flow explicit ("continue", "break" \( \Rightarrow \) goto)
   - simplify the loops ("for", "do while" \( \Rightarrow \) while)

2. Unwind all the loops

3. Convert into SSA (Single Static Assignment)

4. Convert into a set of equations
Loop unwinding

- A bound on the number of unwindings is needed
- Built-in simplifier can calculate non-runtime-dependent bounds
- In other cases an explicit bound should be given
- Unwinding assertion: automatically created to verify that enough loop unwindings were performed
- Nested loops efficient treatment exists (unwinding into a single loop)

Program to formula: Step 2
Loop unwinding - example

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

iterative unwinding: while (c) B ➔
if (c) {B; while (c) B}

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

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

Program to formula: Step 2
void f(…) {
    ...
    if (cond) {
        Body;
        if (cond) {
            Body;
            if (cond) {
                Body;
                assert (!cond);
            }
        }
    }
    ...
}

unwinding assertion
• inserted after the last iteration
• violated if the program runs more iterations than the bound permits

“if” replaces “while”

unwinding assertion
Example: sufficient unwinding

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

bound = 3
Example: insufficient unwinding

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

bound = 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));
            }  
        }
    }
    
    ...  
}

assertion does not hold
Loop-free programs to SSA

• SSA = Single Static Assignment
• Idea: It is easy to convert a program into a formula if every variable is assigned only once in the program
• Example:

<table>
<thead>
<tr>
<th>program</th>
<th>constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = a;</td>
<td>x = a &amp; &amp;</td>
</tr>
<tr>
<td>y = x + 1;</td>
<td>y = x + 1 &amp; &amp;</td>
</tr>
<tr>
<td>z = y - 1;</td>
<td>z = y - 1</td>
</tr>
</tbody>
</table>
Loop-free programs to SSA-2

- The algorithm:
  - If a variable is assigned multiple times, use a new variable for the left hand side of each assignment
  - Initial values of the variables are marked by index 0
- We denote the translation function by $\rho$

Program to formula: Step 3

<table>
<thead>
<tr>
<th>program</th>
<th>SSA program</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = x + y;$</td>
<td>$x_1 = x_0 + y_0;$</td>
</tr>
<tr>
<td>$x = x \ast 2;$</td>
<td>$x_2 = x_1 \ast 2;$</td>
</tr>
<tr>
<td>$a[i] = 100;$</td>
<td>$a_1[i_0] = 100;$</td>
</tr>
</tbody>
</table>
Loop-free programs to SSA-3

- What about conditionals?

**Program**

```c
int x;
int y=8; z=0; w=0;
if (x)
  z = y-1;
else
  w = y+1;
assert (z==5 || w==9)
```

**SSA program**

```c
y_1 = 8;
z_1 = 0;
w_1 = 0;
if (x_0)
  z_2 = y_1 - 1;
else
  w_2 = y_1 + 1;
assert (z_2 == 5 || w_2 == 9)
```

what should “z” and “w” be?
Loop-free programs to SSA-3

• What about conditionals?

program

```c
int x;
int y=8; z=0; w=0;
if (x)
    z = y-1;
else
    w = y+1;
assert (z==5 || w==9)
```

SSA program

```c
int x;
y_1 = 8;
z_1 = 0;
w_1 = 0;
if (x_0)
    z_2 = y_1 - 1;
else
    w_2 = y_1+1;
w_3 = x_0 ? w_1 : w_2;
```z_3 = x_0 ? z_2 : z_1;
assert (z_3 == 5 || w_3 == 9)
From SSA to a formula

- All the assignments are program constraints
- Take the conjunction of all the program constraints
- Take the conjunction of all the assertions
- The constraints should imply the assertions (or, in other words, the conjunction of the constraints with the negation of the assertions should be unsatisfiable)
From SSA to a formula - 2

Program to formula: Step 4

SSA program + assertion

```
y_1 = 8;
z_1 = 0;
w_1 = 0;
if (x_0)
    z_2 = y_1 - 1;
else
    w_2 = y_1 + 1;
w_3 = x_0 ? w_1 : w_2;
z_3 = x_0 ? z_2 : z_1;
assert (z_3 == 5 $|$ w_3 == 9)
```

Formula

```
( y_1 = 8 $\land$
  z_1 = 0 $\land$
  w_1 = 0 $\land$
  z_2 = y_1 - 1 $\land$
  w_2 = y_1 + 1 $\land$
  w_3 = x_0 ? w_1 : w_2 $\land$
  z_3 = x_0 ? z_2 : z_1 )
\land
\neg (z_3 == 5 \lor w_3 == 9)
```
Performing the verification

1. (Convert the formula to CNF and) solve with a SAT-Solver
2. Convert SAT assignment into counterexample
   • In the counterexample we have all the assignments to the variables in all the steps of the program, thus the reconstruction of the trace is possible
Example

Formula
\[ y_1 = 8 \land z_1 = 0 \land w_1 = 0 \land z_2 = y_1 - 1 \land w_2 = y_1 + 1 \land (w_3 = x_0 \ ? w_1 : w_2) \land (z_3 = x_0 \ ? z_2 : z_1) \land \neg (z_3 = 5 \lor w_3 = 9) \]

Sat. Assignment
\[
\begin{align*}
x_0 &= 1, & y_1 &= 8, & z_1 &= 0, & w_1 &= 0, & z_2 &= 7, & w_2 &= 9, & w_3 &= 0, & z_3 &= 7
\end{align*}
\]

trace

```c
int x;
int y=8; z=0; w=0;
if (x)
  z = y-1;
else
  w = y+1;
```

state1: x=1
state2: x=1, y=8, z=0, w=0
state3: x=1, y=8, z=0, w=0
state4: x=1, y=8, z=7, w=0
end values: x=1, y=8, z=7, w=0
Modeling with CBMC

Built-in modeling primitives:

• `xxx nondet_xxx ()`
  – non-deterministically returns a value of type `xxx`
  – modeling external input, unknown environment, library function stub, etc.

• `__CPROVER_assume (expr)`
  – restricts program traces to those satisfying the assumption
  – if (expr) is true, continue with the execution
  – if (expr) is false, abort the program successfully

• `__CPROVER_assert (expr, str)`
  – exists along with the usual `assert(…)`, allowing to add comments to assertions. E.g.,
    `__CPROVER_assert(!(x&1), “x divisible by 2”)`
Question: How to define a function that randomly chooses a number between 1 and 10?

```c
int one_to_ten() {
    int val = nondet_int();
    __CPROVER_assume (1<= val && val <= 10)
    return val;
}
```
Assumptions - example

```c
int array[10]; //defined somewhere (global)
int sum(int k) {
    unsigned i, res;
    res = 0;
    __CPROVER_assume (0<= k && k <= 10)
    for (i=0; i < k; i++)
        res+=array[i];
    return res;
}
```

possible problem: 
k might be out of bounds of the array!

possible solution: 
assume the right value for k

is it enough?
Assumptions example (contd.)

```c
int array[10]; //defined somewhere
int sum(int k) {
    unsigned i, res;
    res = 0;
    assume (0<= k && k<= 10)
    for (i=0; i < k; i++)
        res+=array[i];
    return res;
}

int main(int argc, char ** argv) {
    ...
    int count;
    ...
    assert (0<= count && count <= 10)
    int s = sum(count);
    ...
    return 0;
}
```

ensures that the assumption was possible
Unchecked assumptions danger

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

Passes in CBMC!!!

there are no computations satisfying (x<0) here!

=> The assertion holds vacuously!
Running CBMC

• Support for: Windows, Linux
• Has both command line interface and GUI (Eclipse 3.2 plugin)
• Both working!
CBMC vs. BLAST

• Both tools:
  – Are used to verify C programs
  – Perform assertions checking
  – Support non-determinism

• Verification performed using:
  – BLAST: BDDS, Simplify (theorem prover, using SAT checking), reachability checks
  – CBMC: SAT-Solver (no BDDS)
CBMC vs. BLAST – contd.

• BLAST advantages:
  – Separate “spec” files possible
  – Abstraction – refinement mechanism (there is the SATABS tool developed by CBMC developers, that performs abstraction-refinement, but it is an independent verification tool, not part of CBMC)

• CBMC advantages:
  – Can be used also to verify consistency of hardware designs with a functional specification (written as C program)
  – “Assumptions” mechanism => assume-guarantee specification
  – Can verify modules, and not only whole programs
  – Treats recursive functions
  – Has GUI
CBMC vs. ESC/Java

- ESC/Java is based on Simplify theorem prover (using SAT checking)
- It also translates code to SSA, and then into verification conditions (in FOL)
- ESC/Java supports assume-guarantee reasoning (for JAVA programs, of course)
- But the assume-guarantee statements are on methods and method calls, whereas in CBMC they can appear in any place in the program
- Many features of ESC/Java are relevant only to object-oriented languages (which C is not), and thus incomparable with the capabilities of CBMC
Thank you!