CBMC
Main Idea: Given a program and a claim use a SAT-solver to find whether there exists an execution that violates the claim.

Program → Analysis Engine → CNF → SAT Solver

SAT (counterexample exists) → UNSAT (no counterexample found)
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., 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
What about loops?!

- SAT Solver can only explore finite length executions!
- Loops must be bounded (i.e., the analysis is incomplete)
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`
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. Solve with a SAT Solver
  6. Convert SAT assignment into a counterexample
CBMC: Bounded Model Checker for C
A tool by D. Kroening/Oxford and Ed Clarke/CMU
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

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

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

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

while() loops are unwound iteratively
Break / continue replaced by goto
void f(...) {
    if(cond) {
        Body;
        if(cond) {
            Body;
            if(cond) {
                Body;
                while(cond) {
                    Body;
                }
            }
        }
    }
} 

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!
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**

```
x=x+y;
x=x*2;
a[i]=100;
```

**SSA Program**

```
x₁=x₀+y₀;
x₂=x₁*2;
a₁[i₀]=100;
```
What about conditionals?

**Program**

\[
\begin{align*}
\text{if } (v) & \\
& x = y; \\
\text{else} & \\
& x = z; \\
w & = x;
\end{align*}
\]

**SSA Program**

\[
\begin{align*}
\text{if } (v_0) & \\
& x_0 = y_0; \\
\text{else} & \\
& x_1 = z_0; \\
w_1 & = x_??;
\end{align*}
\]

*What should ‘x’ be?*
What about conditionals?

- For each join point, add new variables with selectors

**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
```