Bounded Model Checking for Software

COMP6210 - Automated Software Verification

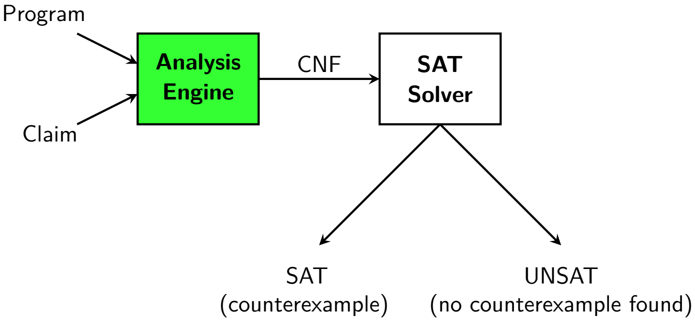
ECS, University of Southampton

3rd November 2020

Previously...

CBMC: the **C** **B**ounded **M**odel **C**hecker

Main idea: Given a C program and a claim, use a SAT solver to check of there is an execution that violates the claim.

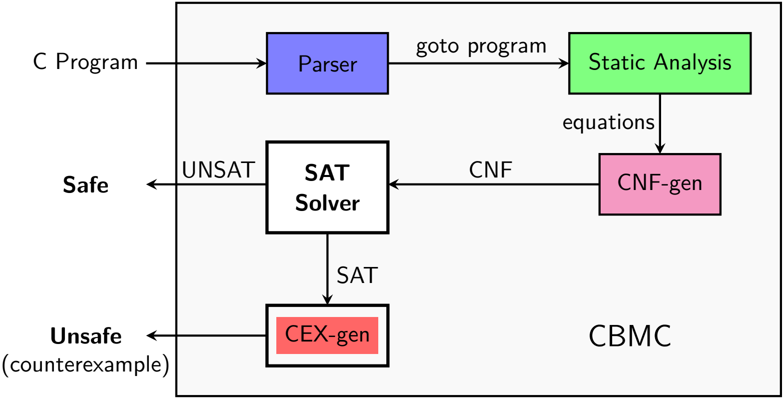


CBMC: How Does It Work?

Transforms a C program into a *set of equations*.

1. Simplify control flow
2. Unwind all the loops
   展开
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: How Does It Work?



Control Flow Simplification

- All side effects are removed

  e.g. `j=i++;` *is transformed into* `j=i; i=i+1;`

- Control flow is made *explicit*

  `continue` *and* `break` *are replaced by* `goto`

- All loops are simplified into *one form*

  e.g. `for`, `do`, `while` *are replaced by* $\boxed{\textit{just}\ \texttt{while}}$

Loop Unwinding

- All loops are *unwound*

  can use different unwinding bounds for different loops

  can check whether unwinding is sufficient using a special unwinding assertion

- If a program satisfies all of its claims *and* all unwinding assertions, then it is *correct*.
- Recursive functions and backward `goto` are similar (use inlining).

Loop Unwinding

`while` loops are unwound *iteratively*; `break`/`continue` replaced by `goto`.

重复的，迭代的

```
void f(...) {
  ... // some code
  while(cond){
    Body;
  }
  Remainder;
}
```

Loop Unwinding 将循环展开为if

while loops are unwound *iteratively*; break/continue replaced by goto.

```
void f(...) {
  ... // some code
  if(cond){
    Body;
    while(cond){
      Body;
    }
  }
  Remainder;
}
```

Loop Unwinding

`while` loops are unwound *iteratively*; `break`/`continue` replaced by `goto`.

```
void f(...) {
  ... // some code
  if(cond){
    Body;
    if(cond){
      Body;
      while(cond){
        Body;
      }
    }
  }
  Remainder;
}
```

Loop Unwinding

`while` loops are unwound *iteratively*; `break`/`continue` replaced by `goto`.

```
void f(...) {
  ... // some code
  if(cond){
    Body;
    if(cond){
      Body;
      if(cond){
        Body;
        while(cond){
          Body;
        }
      }
    }
  }
  Remainder;
}
```

Loop Unwinding

Assertion inserted after last iteration: violated if the program runs longer than bound permits.

```
void f(...) {
  ... // some code
  if(cond){
    Body;
    if(cond){
      Body;
      if(cond){
        Body;
        assert(!cond); //Unwinding assertion
      }
    }
  }
  Remainder;
}
```

未按期望终止循环

Loop Unwinding

Assertion inserted after last iteration: violated if the program runs longer than bound permits. Positive correctness result!

```
void f(...) {
  ... // some code
  if(cond){
    Body;
    if(cond){
      Body;
      if(cond){
        Body;
        assert(!cond); //Unwinding assertion
      }
    }
  }
  Remainder;
}
```

Example: Sufficient Loop Unwinding

`unwind = 3`

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

```
void f(...){
  j = 1;
  if(j<=2){
   j = j + 1;     2
   if(j<=2){
    j = j + 1; 3
    if(j<=2){
     j = j + 1;
     assert(!(j<=2));
    }
   }
  }
  Remainder;
}
```

Example: Insufficient Loop Unwinding

`unwind = 3`

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

```
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;
}
```

Southampton

Transforming Loop-Free Programs Into Equations

不重要的

It is trivial to translate a program into a set of equations if each variable is only assigned once!

```
x = a;
y = x+1;
z = y-1;
```

This program is directly transformed into

$$x = a \wedge y = x + 1 \wedge z = y - 1\,.$$

Transforming Loop-Free Programs Into Equations

Static Single Assignment (SSA) form.

- Every variable is assigned *exactly once*.
- Every variable is defined *before it is used*.

When a variable is assigned multiple times, we use a new variable for each assignment.

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

```
x1 = x0 + y0;
x2 = x1*2;
a1[i0] = 100;
```

Transforming Loop-Free Programs Into Equations

Static Single Assignment (SSA) form.

- Every variable is assigned *exactly once*.
- Every variable is defined *before it is used*.

When a variable is assigned multiple times, we use a new variable for each assignment.

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

```
x1 = x0 + y0;
x2 = x1*2;
a1[i0] = 100;
```

What about conditionals?

Transforming Loop-Free Programs Into Equations

Converting conditionals to SSA.

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

```
if(v0)
  x0 = y0;
else
  x1 =  z0;
w1 = x?? // which x?
```

Transforming Loop-Free Programs Into Equations

Converting conditionals to SSA.

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

```
if(v0)
  x0 = y0;
else
  x1 =  z0;
x2 = v0 ? x0 : x1;
w1 = x2;
```

For each joint point and new variables with *selectors*.

Loop-Free Example

Starting from the following C code:

```
int y;
int x;
x=x+y;
if(x!=1)
  x=2;
else
  x++;
assert(x<=3);
```

Loop-Free Example

Simplify control flow

```
int y;
int x;
x=x+y;
if(x!=1)
  x=2;
else
  x=x+1;
assert(x<=3);
```

Loop-Free Example

Convert to SSA (Static Single Assignment form)

```
x1 = x0+y0;
if(x1 != 1)
  x2 = 2;
else
  x3 = x1 + 1;
x4 = (x1 != 1) ? x2 : x3;
assert(x4<=3);
```

Loop-Free Example

Convert to SSA (Static Single Assignment form)

```
x1 = x0+y0;
if(x1 != 1)
  x2 = 2;
else
  x3 = x1 + 1;
x4 = (x1 != 1) ? x2 : x3;
assert(x4<=3);
```

$\text{Init} \wedge (\bigwedge \text{Trans}) \wedge (\bigvee \text{Target})$

Generate constraints (if SAT, then assertion is false):

$$
\begin{aligned}
& x_1 = x_0 + y_0 \wedge x_2 = 2 \wedge x_3 = x_1 + 1 \\
\wedge\ & ((x_1 \neq 1 \wedge x_4 = x_2) \vee (x_1 = 1 \wedge x_4 = x_3)) \quad \text{[selector]} \\
\wedge\ & \neg(x_4 \leq 3) \quad \text{[negated assertion]}
\end{aligned}
$$

Example with a Loop

Starting from the following C code:

```
int i;
int p;
p=5;
for (i=0; i<=n; i++) {
  p = p * m;
}
assert(p>=5);
```

Example with a Loop

Transform the for loop into a while loop

```
int i;
int p;
p=5; i=0;
while (i<=n) {
  p = p * m;
  i = i + 1;
}
assert(p>=5);
```

Example with a Loop

展开

Unroll the loop twice and add an assume statement to exit the loop

```
int i;
int p;
p=5; i=0;
if(i<=n) {
  p = p * m;
  i = i + 1;
  while (i<=n) {
    p = p * m;
    i = i + 1;
  }
assert(p>=5);
```

Example with a Loop

Unroll the loop twice and add an assume statement to exit the loop

```
int i;
int p;
p=5; i=0;
if(i<=n)
  p = p *
  i = i +
  if (i<
    p = p
    i = i
    assume(!(i<=n));
  }
assert(p>=5);
```

Example with a Loop

Assign all variables exactly once, compute guards for conditionals and add conditionals for merging values.

缩进敏感?

```
p1=5;
i1=0;
g1=i1<=n1;
```

布尔条件变量

```
  p2=p1*m1; // g1
  i2=i1+1;  // g1
  g2=(i2<=n1);
    p3 = p2*m1 //g1 && g2
    i3 = i2 + 1; //g1 && g2
    assume( !(i3<=n1));
```

假设此刻退出

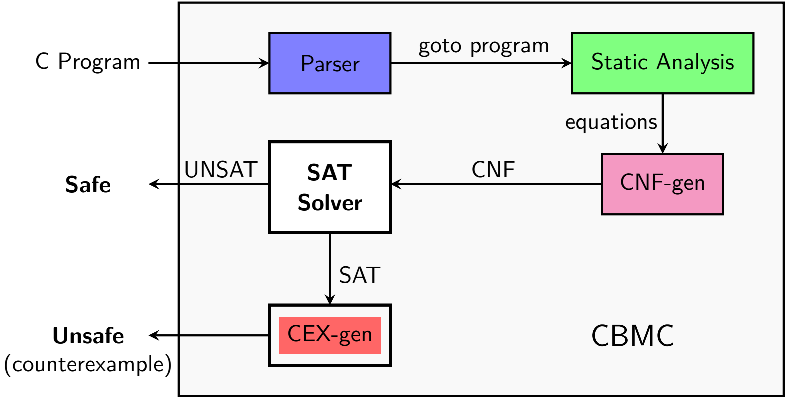
```
p4=g1 ? (g2 ? p3 : p2) : p1;
i4=g1 ? (g2 ? i3 : i2) : i1; // i4 unused
assert(p4 >= 5);
```

Example with a Loop

Convert to logical expression (if UNSAT, then assertion holds).

$$
\begin{aligned}
& p_1 = 5 \\
& \wedge\ i_1 = 0 \\
& \wedge\ g_1 = (i_1 \leq n_1) \\
& \wedge\ p_2 = p_1 * m_1 \\
& \wedge\ i_2 = i_1 + 1 \\
& \wedge\ g_2 = (i_2 \leq n_1) \\
& \wedge\ p_3 = p_2 * m_1 \\
& \wedge\ i_3 = i_2 + 1 \\
& \wedge\ \neg(i_3 \leq n_1) \quad \text{[assume statement]} \\
& \wedge\ p_4 = g_1\ ?\ (g_2\ ?\ p_3\ :\ p_2)\ :\ p_1 \\
& \wedge\ i_4 = g_1\ ?\ (g_2\ ?\ i_3\ :\ i_2)\ :\ i_1 \\
& \wedge\ \neg(p_4 \geq 5) \quad \text{[assert statement]}
\end{aligned}
$$

CBMC: How Does It Work?



Bit Blasting

So far, formulas such as $x_2 = x_1 + 1 \wedge y_2 = x_2$ are *not* stated in propositional logic!

The operations are performed on *bit vectors*.

In order to convert these formulas into a format acceptable to a SAT solver, one needs to apply *flattening*/*bit blasting*.

Bit Blasting

So far, formulas such as $x_2 = x_1 + 1 \wedge y_2 = x_2$ are *not* stated in propositional logic!

The operations are performed on *bit vectors*.

In order to convert these formulas into a format acceptable to a SAT solver, one needs to apply *flattening*/*bit blasting*.

Intuitively, we can build Boolean circuits for the bit-vector operations; these can be described by Boolean formulas (*unfortunately with a lot more variables*).

Bit Blasting

So far, formulas such as $x_2 = x_1 + 1 \wedge y_2 = x_2$ are *not* stated in propositional logic!

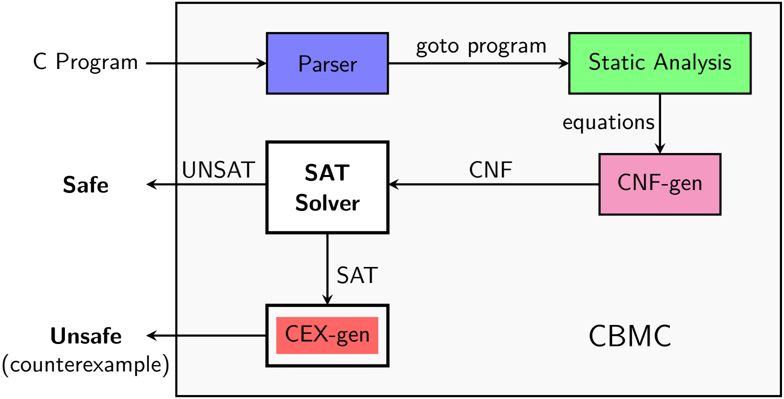
The operations are performed on <u>*bit vectors*</u>.

In order to convert these formulas into a format acceptable to a SAT solver, one needs to apply *flattening*/*bit blasting*.

Intuitively, we can build Boolean circuits for the bit-vector operations; these can be described by Boolean formulas (*unfortunately with a lot more variables*).

A Boolean formula can be (cheaply) brought into CNF using Tseytin transformation (at the cost of yet more new Boolean variables).

CBMC: How Does It Work?



Further Reading:

- CBMC Tutorial:
  http://www.cprover.org/cprover-manual/cbmc/tutorial/
- **Edmund Clarke, et al.** "*Behavioral consistency of C and Verilog programs using bounded model checking.*" " Proceedings 2003. Design Automation Conference. IEEE, 2003.
  CMU-CS-03-126.pdf

Optional :

- **Lucas Cordeiro, et al.** "*SMT-based bounded model checking for embedded ANSI-C software.*" IEEE Transactions on Software Engineering 38.4 (2011): 957-974.
  https://core.ac.uk/download/pdf/59348834.pdf

Acknowledgements: Partly based on material by Dr Corina Cîrstea, Dr Gennaro Parlato, University of Southampton, Dr Daniel Kroening, University of Oxford, and Dr Paul Jackson, University of Edinburgh.