The CBMC bounded model checker for C

Paul Jackson
Paul.Jackson@ed.ac.uk

University of Edinburgh

Formal Verification
Autumn 2023

Bounded Model Checking

- Bounded Model Checking (BMC) is the most successful formal validation technique in the *hardware* industry
- Advantages:
  - ✔ Fully automatic
  - ✔ Robust
  - ✔ Lots of subtle bugs found
- Idea: only look for bugs up to specific depth
- Good for many applications, e.g., embedded systems
- CBMC and related tools apply BMC ideas to *software*

Encoding straight line code and conditionals

Adopt Symbolic Execution strategy:

- Introduce new variable name for each re-assignment
- At control-flow join points, use conditional guards to select variable values

Encoding straight line code and conditionals

Adopt Symbolic Execution strategy:

- Introduce new variable name for each re-assignment
- At control-flow join points, use conditional guards to select variable values

```
int abs (int x) {
  int y = x;
  if (x < 0) {
    y = -x;
  }
  return y;
}
```

Encoding straight line code and conditionals

Adopt Symbolic Execution strategy:

- Introduce new variable name for each re-assignment
- At control-flow join points, use conditional guards to select variable values

```
int abs (int x) {
  int y = x;
  if (x < 0) {
    y = -x;
  }
  return y;
}
```

```
int abs (int x1) {
  int y2 = x1;
  int guard1 = (x1 < 0);
    int y3 = -x1;
  int y4 = (guard1) ? y3 : y2;
  return y4;
}
```

Unrolling Loops

This essentially amounts to unwinding loops:

```
while(cond)
    Body;
```

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
  Body;
  while(cond)
    Body;
}
```

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
    Body;
    if(cond) {
        Body;
        while(cond)
            Body;
    }
}
```

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    if(cond) {
      Body;
      while(cond)
        Body;
    }
  }
}
```

Unrolling Loops

This essentially amounts to unwinding loops:

```
if(cond) {
   Body;
   if(cond) {
      Body;
      if(cond) {
         Body;
         assume(!cond);

      }
   }
}
```

Completeness

BMC, as discussed so far, is incomplete.
It only refutes, and does not prove.

How can we fix this?

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
while(cond)
    Body;
```

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
if(cond) {
    Body;
    while(cond)
        Body;
}
```

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
if(cond) {
    Body;
    if(cond) {
        Body;
        while(cond)
            Body;
    }
}
```

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
if(cond) {
   Body;
   if(cond) {
      Body;
      if(cond) {
         Body;
         while(cond)
            Body;
      }
   }
}
```

Unwinding Assertions

Let's revisit the loop unwinding idea:

```
if(cond) {
  Body;
  if(cond) {
    Body;
    if(cond) {
      Body;
      assert(!cond);

    }
  }
}
```

CBMC VC derivation 1

Q. Given program

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

What VC might CBMC generate, if loop is unrolled two times and we assume loop will not execute a third time?

A. Transform first to while loop, since easier to unroll

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

CBMC VC derivation 2

Unroll loop 2 times and add assume statement for loop exiting at that point

```
p = 1;
i = 0;
if (i <= n) {
  p = p * m;
  i = i + 1;
  if (i <= n) {
    p = p * m;
    i = i + 1;
    assume( !(i <= n) );
  }
}
assert(p >= 1);
```

CBMC VC derivation 3

Assign all variables exactly once. Compute guards for conditional statements. Add conditional expressions for merging values.

```
p1 = 1;
i1 = 0;
g1 = i1 <= n1;
  p2 = p1 * m1; // g1
  i2 = i1 + 1;  // g1
  g2 = (i2 <= n1);
    p3 = p2 * m1;  // g1 & g2
    i3 = i2 + 1;   // g1 & g2
    assume( !(i3 <= n1) );
p4 = g1 ? (g2 ? p3 : p2) : p1;
i4 = g1 ? (g2 ? i3 : i2) : i1; // Optional, since i4 unused
assert(p4 >= 1);
```

Comments track conditions under which assignments hold and help with computing value merge expressions.

CBMC VC derivation 4

Convert to logical expression.

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

If this is found unsatisfiable, then assertion holds.

Inlining function calls

- A standard compiler transformation
- Recursive definitions handled in similar way to loops

Inlining function calls

- A standard compiler transformation
- Recursive definitions handled in similar way to loops

Library calls

- Assumed to have non-deterministic behaviour

Pointers

How do we handle dereferencing in the program?

Pointers

How do we handle dereferencing in the program?

```
int *p;
p=malloc(sizeof(int)*5);
 ...

p[1]=100;
```

$$p_1 = \& DO1$$
$$\wedge \quad DO1_1 = (\lambda i. \; i = 1 ? 100 : DO1_0[i])$$

Here *DO1* is an *uninterpreted function* and the formulas on the right are in the theory of *equality and uninterpreted functions* (EUF)

Pointers

How do we handle dereferencing in the program?

```
int *p;
p=malloc(sizeof(int)*5);
...

p[1]=100;
```

$$
\begin{array}{ll}
 & p_1 = \&DO1 \\
\wedge & DO1_1 = (\lambda i. \\
 & i = 1 ? 100 : DO1_0[i])
\end{array}
$$

Here *DO1* is an *uninterpreted function* and the formulas on the right are in the theory of *equality and uninterpreted functions* (EUF)

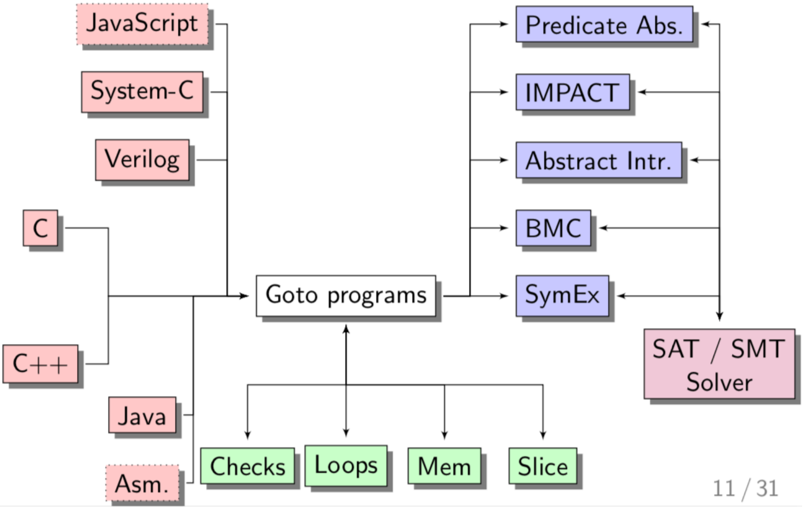
EUF handled by either SMT techniques or reduction to SAT.

Automatic property checks

Include

- Buffer overflows: For each array access, check whether the upper and lower bounds are violated.
- Pointer safety: Search for NULL-pointer dereferences or dereferences of other invalid pointers.
- Division by zero: Check whether there is a division by zero in the program.
- Not-a-Number: Check whether floating-point computation may result in NaNs.
- Uninitialised local Check whether the program uses an uninitialised local variable.
- Data race: Check whether a concurrent program accesses a shared variable at the same time in two threads.

CProver Tool Suite



Sources

CBMC: Bounded Model Checking for ANSI-C
Introductory slides on CBMC from CBMC website: `http://www.cprover.org/cbmc/`

The CProver Suite of Verification Tools.
Martin Brain. 2016.
First part of a tutorial on CBMC and related tools given at the FM 2016 conference.