CBMC: Bounded Model Checking for ANSI-C

Version 1.0, 2010
Outline

Preliminaries

BMC Basics

Completeness

Solving the Decision Problem
Preliminaries

- We aim at the analysis of programs given in a commodity programming language such as C, C++, or Java.

- As the first step, we transform the program into a control flow graph (CFG).
Example: SHS

```c
if ( (0 <= t) && (t <= 79) )
    switch ( t / 20 )
    {
        case 0:
            TEMP2 = ( (B AND C) OR (~B AND D) );
            TEMP3 = ( K_1 );
            break;
        case 1:
            TEMP2 = ( (B XOR C XOR D) );
            TEMP3 = ( K_2 );
            break;
        case 2:
            TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
            TEMP3 = ( K_3 );
            break;
        case 3:
            TEMP2 = ( B XOR C XOR D );
            TEMP3 = ( K_4 );
            break;
        default:
            assert(0);
    }
```
Example: SHS

```c
if ( (0 <= t) && (t <= 79) )
switch ( t / 20 )
{
  case 0:
    TEMP2 = ( (B AND C) OR (~B AND D) );
    TEMP3 = ( K_1 );
    break;
  case 1:
    TEMP2 = ( (B XOR C XOR D) );
    TEMP3 = ( K_2 );
    break;
  case 2:
    TEMP2 = ( (B AND C) OR (B AND D) OR (C AND D) );
    TEMP3 = ( K_3 );
    break;
  case 3:
    TEMP2 = ( B XOR C XOR D );
    TEMP3 = ( K_4 );
    break;
  default:
    assert(0);
}
```

Goal: check properties of the form $AG_p$, say assertions.

Idea: follow paths through the CFG to an assertion, and build a formula that corresponds to the path.
Example

Example:

if

0 ≤ t ≤ 79

switch

case-0

t/20 ≠ 0

case-1

t/20 ≠ 1

case-2

t/20 ≠ 2

case-3

t/20 ≠ 3

default

0 ≤ t ≤ 79

t/20 ≠ 0

t/20 ≠ 1

t/20 ≠ 2

t/20 ≠ 3
Example

if

0 ≤ t ≤ 79

switch

case-0

t/20 ≠ 0

case-1

t/20 ≠ 1

case-2

t/20 ≠ 2

case-3

t/20 ≠ 3

default

0 ≤ t ≤ 79
t/20 ≠ 0
t/20 ≠ 1
t/20 ≠ 2
t/20 ≠ 3
Example

```
if
  0 ≤ t ≤ 79
switch
  case-0
    t/20 ≠ 0
  case-1
    t/20 ≠ 1
  case-2
    t/20 ≠ 2
  case-3
    t/20 ≠ 3
  default
```

\[ 0 ≤ t ≤ 79 \quad ∧ \quad t/20 ≠ 0 \quad ∧ \quad t/20 = 1 \quad ∧ \quad TEMP2 = B ⊕ C ⊕ D \quad ∧ \quad TEMP3 = K_2 \]
Example

We pass

\[ 0 \leq t \leq 79 \]
\[ \land \ t/20 \neq 0 \]
\[ \land \ t/20 = 1 \]
\[ \land \ TEMP2 = B \oplus C \oplus D \]
\[ \land \ TEMP3 = K \_2 \]

to a decision procedure, and obtain a satisfying assignment, say:

\[ t \mapsto 21, \ B \mapsto 0, \ C \mapsto 0, \ D \mapsto 0, \ K\_2 \mapsto 10, \]
\[ TEMP2 \mapsto 0, \ TEMP3 \mapsto 10 \]

✓ It provides the values of any inputs on the path.
Which Decision Procedures?

- We need a decision procedure for an appropriate logic
  - Bit-vector logic (incl. non-linear arithmetic)
  - Arrays
  - Higher-level programming languages also feature lists, sets, and maps

- Examples
  - Z3 (Microsoft)
  - Yices (SRI)
  - Boolector
Enabling Technology: SAT

number of variables of a typical, practical SAT instance that can be solved by the best solvers in that decade
Enabling Technology: SAT

- propositional SAT solvers have made enormous progress in the last 10 years

- Further scalability improvements in recent years because of efficient word-level reasoning and array decision procedures
Let’s Look at Another Path

if

0 ≤ t ≤ 79

switch

case-0

\( t/20 \neq 0 \)

case-1

\( t/20 \neq 1 \)

case-2

\( t/20 \neq 2 \)

case-3

\( t/20 \neq 3 \)

default

That is UNSAT, so the assertion is unreachable.

Let’s Look at Another Path

if
0 ≤ t ≤ 79
switch
case-0
  t/20 ≠ 0
case-1
  t/20 ≠ 1
case-2
  t/20 ≠ 2
case-3
  t/20 ≠ 3
default

That is UNSAT, so the assertion is unreachable.
Let’s Look at Another Path

\begin{align*}
\text{if} & \quad 0 \leq t \leq 79 \\
\text{switch} & \\
\text{case-0} & \quad t/20 \neq 0 \\
\text{case-1} & \quad t/20 \neq 1 \\
\text{case-2} & \quad t/20 \neq 2 \\
\text{case-3} & \quad t/20 \neq 3 \\
\text{default} & \\
\end{align*}

0 \leq t \leq 79
\wedge t/20 \neq 0
\wedge t/20 \neq 1
\wedge t/20 \neq 2
\wedge t/20 \neq 3

That is UNSAT, so the assertion is unreachable.
Let’s Look at Another Path

if
0 ≤ t ≤ 79
switch
case-0
   t/20 ≠ 0
case-1
   t/20 ≠ 1
case-2
   t/20 ≠ 2
case-3
   t/20 ≠ 3
default

0 ≤ t ≤ 79 ∧ t/20 ≠ 0 ∧ t/20 ≠ 1 ∧ t/20 ≠ 2 ∧ t/20 ≠ 3

That is UNSAT, so the assertion is unreachable.
What If a Variable is Assigned Twice?

```c
x=0;
if (y>=0)
    x++;  
```

Rename appropriately:

\[
\begin{align*}
    x &= 0 \\
    \land y &\geq 0 \\
    \land x &= x + 1 \\
\end{align*}
\]
What If a Variable is Assigned Twice?

```
x = 0;
if (y >= 0)
    x++;  
```

Rename appropriately:

```
x_1 = 0
∧ y_0 ≥ 0
∧ x_1 = x_0 + 1
```

This is a special case of SSA (static single assignment)
How do we handle dereferencing in the program?

```c
int *p;
p = malloc(sizeof(int) * 5);
...
p[1] = 100;
```

Track a 'may-point-to' abstract state while simulating!
Pointers

How do we handle dereferencing in the program?

```c
int *p;
p = malloc(sizeof(int) * 5);
...
p[1] = 100;
```

Track a ‘may-point-to’ abstract state while simulating!

\[ p_1 = \& DO1 \land DO1_1 = (\lambda i. i = 1?100 : DO1_0[i]) \]
Let’s consider the following CFG:

This is a loop with an `if` inside.
Scalability of Path Search

Let’s consider the following CFG:

```
L1
L2 L3
L4
```

This is a loop with an \texttt{if} inside.

Q: how many paths for \( n \) iterations?
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
Definition: A transition system is a triple \((S, S_0, T)\) with

- set of states \(S\),
- a set of initial states \(S_0 \subset S\), and
- a transition relation \(T \subset (S \times S)\).

The set \(S_0\) and the relation \(T\) can be written as their characteristic functions.
Unwinding a Transition System

Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation $T$:

$$S_0$$
Unwinding a Transition System

Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation $T$:

$$S_0 \land T$$

\[ \bullet \xrightarrow{T} \bullet \]
Q: How do we avoid the exponential path explosion?

We just ”concatenate” the transition relation $T$:

$$S_0 \land T \land T \cdots$$

[Diagram showing state transitions]
Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation $T$:

$$S_0 \wedge T \wedge T \wedge \ldots \wedge T$$
Q: How do we avoid the exponential path explosion?

We just "concatenate" the transition relation $T$:

$$\begin{align*}
S_0 \land T & \land T & \land T & \cdots & \land T \\
\bullet_{s_0} & \rightarrow \bullet_{s_1} & \rightarrow \bullet_{s_2} & \cdots & \rightarrow \bullet_{s_k} 
\end{align*}$$
Unwinding a Transition System

As formula:

\[ S_0(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \]

Satisfying assignments for this formula are traces through the transition system.
Example

\[ T \subseteq \mathbb{N}_0 \times \mathbb{N}_0 \]

\[ T(s, s') \iff s'.x = s.x + 1 \]

...and let \( S_0(s) \iff s.x = 0 \lor s.x = 1 \)
Example

\[ T \subseteq \mathbb{N}_0 \times \mathbb{N}_0 \]

\[ T(s, s') \iff s'.x = s.x + 1 \]

... and let \( S_0(s) \iff s.x = 0 \lor s.x = 1 \)

An unwinding for depth 4:

\[ (s_0.x = 0 \lor s_0.x = 1) \]

\[ \land s_1.x = s_0.x + 1 \]

\[ \land s_2.x = s_1.x + 1 \]

\[ \land s_3.x = s_2.x + 1 \]

\[ \land s_4.x = s_3.x + 1 \]
Suppose we want to check a property of the form $\forall G p$. Satisfying assignments are counterexamples for the $\forall G p$ property.
Checking Reachability Properties

Suppose we want to check a property of the form $\text{AG}_p$.

We then want at least one state $s_i$ to satisfy $\neg p$:

$$S_0(s_0) \land \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \land \bigvee_{i=0}^{k} \neg p(s_i)$$

Satisfying assignments are counterexamples for the $\text{AG}_p$ property
Unwinding Software

We can do exactly that for our transition relation for software.

E.g., for a program with 5 locations, 6 unwindings:

```
#0  L1* L2* L3* L4* L5*
#1  L1* L2* L3* L4* L5*
#2  L1* L2* L3* L4* L5*
#3  L1* L2* L3* L4* L5*
#4  L1* L2* L3* L4* L5*
#5  L1* L2* L3* L4* L5*
#6  L1* L2* L3* L4* L5*
```
Problem: obviously, most of the formula is never ’used’, as only few sequences of PCs correspond to a path.
Unwinding Software

Example:

CFG
Unwinding Software

Example:

CFG

L1
L2
L3
L4
L5

#0
L1  L2  L3  L4  L5

#1
L1  L2  L3  L4  L5

#2
L1  L2  L3  L4  L5

#3
L1  L2  L3  L4  L5

#4
L1  L2  L3  L4  L5

#5
L1  L2  L3  L4  L5

#6
L1  L2  L3  L4  L5

unrolling
Unwinding Software

Optimization:
don’t generate the parts of the formula that are not ‘reachable’

CFG

Optimization:
don’t generate the parts of the formula that are not ’reachable’
Unwinding Software

Problem:

CFG unrolling

Unwinding Software

- Unwinding $T$ with bound $k$ results in a formula of size $|T| \cdot k$

- If we assume a $k$ that is only linear in $|T|$, we get a formula with size $O(|T|^2)$

- Can we do better?
Unrolling Loops

Idea: do exactly one location in each timeframe:

```
L1
L2
L3
L4
L5
```

CFG
Unrolling Loops

Idea: do exactly one location in each timeframe:

CFG

unrolling

### CBMC: Bounded Model Checking for ANSI-C

Unrolling Loops

- More effective use of the formula size

- Graph has fewer merge nodes, the formula is easier for the solvers

- Not all paths of length $k$ are encoded → the bound needs to be larger
Unrolling Loops

This essentially amounts to unwinding loops:

```c
while (cond)
    Body;
```
Unrolling Loops

This essentially amounts to unwinding loops:

```c
if (cond) {
    Body;
    while (cond)
        Body;
}
```
Unrolling Loops

This essentially amounts to unwinding loops:

```c
if (cond) {
    Body;
    if (cond) {
        Body;
        while (cond)
            Body;
    }
}
```
Unrolling Loops

This essentially amounts to unwinding loops:

```c
if (cond) {
    Body;
    if (cond) {
        Body;
        if (cond) {
            Body;
            while (cond) {
                Body;
            }
        }
    }
}
```
Unrolling Loops

This essentially amounts to unwinding loops:

```c
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:

```c
if (cond) {
    Body;
    while (cond)
        Body;
}
```
Let’s revisit the loop unwinding idea:

```c
if (cond) {
    Body;
    if (cond) {
        Body;
        while (cond)
            Body;
    }
}
```
Unwinding Assertions

Let’s revisit the loop unwinding idea:

```c
if (cond) {
    Body;
    if (cond) {
        Body;
        if (cond) {
            Body;
            while (cond) {
                Body;
            }
        }
    }
}
```
Unwinding Assertions

Let’s revisit the loop unwinding idea:

```c
if (cond) {
    Body;
    if (cond) {
        Body;
        if (cond) {
            Body;
            assert (!cond);
        }
    }
}
```
Unwinding Assertions

- We replace the assumption we have used earlier to cut off paths by an assertion.

- This allows us to prove that we have done enough unwinding.

- This is a proof of a high-level worst-case execution time (WCET).

- Very appropriate for embedded software.
CBMC Toolflow: Summary

1. Parse, build CFG

2. Unwind CFG, form formula

3. Formula is solved by SAT/SMT
Solving the Decision Problem

Suppose we have used some unwinding, and have built the formula.

For bit-vector arithmetic, the standard way of deciding satisfiability of the formula is *flattening*, followed by a call to a propositional SAT solver.

In the SMT context: SMT-$\mathcal{BV}$
Bit-vector Flattening

- This is easy for the bit-wise operators.

- Denote the Boolean variable for bit $i$ of term $t$ by $\mu(t)_i$.

- Example for $a \mid [y] b$:

$$\bigwedge_{i=0}^{l-1} (\mu(t)_i = (a_i \lor b_i))$$

(read $x = y$ over bits as $x \iff y$)
This is easy for the bit-wise operators.

Denote the Boolean variable for bit $i$ of term $t$ by $\mu(t)_i$.

Example for $a |_{[l]} b$:

$$\bigwedge_{i=0}^{l-1} (\mu(t)_i = (a_i \lor b_i))$$

(read $x = y$ over bits as $x \iff y$)

We can transform this into CNF using Tseitin’s method.
Flattening Bit-Vector Arithmetic

How to flatten $a + b$?
How to flatten $a + b$?

→ we can build a circuit that adds them!

The full adder in CNF:

$(a \lor b \lor \neg o) \land (a \lor \neg b \lor i \lor \neg o) \land (a \lor \neg b \lor \neg i \lor o) \land
\neg a \lor b \lor i \lor \neg o \land (\neg a \lor b \lor \neg i \lor o) \land (\neg a \lor \neg b \lor o)$
Flattening Bit-Vector Arithmetic

Ok, this is good for one bit! How about more?
Flattening Bit-Vector Arithmetic

Ok, this is good for one bit! How about more?

8-Bit ripple carry adder (RCA)

▶ Also called *carry chain adder*
▶ Adds *l* variables
▶ Adds $6 \cdot l$ clauses
Multipliers

- Multipliers result in very hard formulas

- Example:

\[ a \cdot b = c \land b \cdot a \neq c \land x < y \land x > y \]

CNF: About 11000 variables, unsolvable for current SAT solvers

- Similar problems with division, modulo

- Q: Why is this hard?
Multipliers

- Multipliers result in very hard formulas

Example:

\[ a \cdot b = c \land b \cdot a \neq c \land x < y \land x > y \]

CNF: About 11000 variables, unsolvable for current SAT solvers

- Similar problems with division, modulo

- Q: Why is this hard?
- Q: How do we fix this?
Incremental Flattening

\[ \varphi_f := \varphi_{sk}, \quad F := \emptyset \]

\( \varphi_{sk} \): Boolean part of \( \varphi \)
\( F \): set of terms that are in the encoding
Incremental Flattening

\( \varphi_f := \varphi_{sk}, \ F := \emptyset \)

Is \( \varphi_f \) SAT?

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding
Incremental Flattening

\[ \varphi_f := \varphi_{sk}, \ F := \emptyset \]

Is \( \varphi_f \) SAT?

No!

UNSAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding
Incremental Flattening

\[ \varphi_f := \varphi_{sk}, \ F := \emptyset \]

\[ \text{Is } \varphi_f \text{ SAT?} \]

- Yes! compute \( I \)
- No! \( \text{UNSAT} \)

\( \varphi_{sk} \): Boolean part of \( \varphi \)
\( F \): set of terms that are in the encoding
\( I \): set of terms that are inconsistent with the current assignment
Incremental Flattening

\[ \varphi_f := \varphi_{sk}, \quad F := \emptyset \]

Is \( \varphi_f \) SAT?

Yes!

compute \( I \)

\( I = \emptyset \)

No!

UNSAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)

\( F \): set of terms that are in the encoding

\( I \): set of terms that are inconsistent with the current assignment
Incremental Flattening

\[ \varphi_f := \varphi_{sk}, F := \emptyset \]

Pick \( F' \subseteq (I \setminus F) \)
\[ F := F \cup F' \]
\[ \varphi_f := \varphi_f \land \text{CONSTRAINT}(F) \]

Is \( \varphi_f \) SAT?

- Yes!
  - compute \( I \)
  - \( I \neq \emptyset \)
  - \( I = \emptyset \)

- No!
  - UNSAT
  - SAT

\( \varphi_{sk} \): Boolean part of \( \varphi \)
\( F \): set of terms that are in the encoding
\( I \): set of terms that are inconsistent with the current assignment

\[ \text{CBMC: Bounded Model Checking for ANSI-C – http://www.cprover.org/} \]
Incremental Flattening

- Idea: add ‘easy’ parts of the formula first

- Only add hard parts when needed

- $\varphi_f$ only gets stronger – use an incremental SAT solver