Static Analysis of Ansi-C Programs in CBMC

Ramón Béjar Torres

Departament d'Informàtica
Universitat de Lleida

BMC for Linear Temporal Logic (LTL) properties

What can we verify for a system specified trough its Kripke representation $M = (S, S_0, R, AP, L)$?

We are going to express and verify properties over its set of valid execution paths:

- Properties satisfied by particular states (e.g. ending state of the executions)
- Properties satisfied by general subsets of states (initial, intermediate, all the states...)

BMC for Linear Temporal Logic (LTL) properties

Given a Kripke model $M = (S, S_0, R, AP, L)$ that represents a system we want to verify, we are going to focus on checking a property formula f that we want to be satisfied by any (A) possible execution path of our system. In symbols:

$$M \models A\, f$$

By execution path, we mean any sequence of states:

$$s_0, s_1, \ldots, s_n$$

that is a valid execution path in M:

- $s_0 \in S_0$
- $(s_i, s_{i+1}) \in R$

BMC for Linear Temporal Logic (LTL) properties

Typical propositional properties that can be expressed in LTL for a Krikpe model $M = (S, S_0, R, AP, L)$:

- Global properties: a property p satisfied in any state for any valid execution path
- Eventual properties: a property p satisfied in some state for any valid execution path

The propositional properties are expressed over the set AP used to label the states

BMC for Linear Temporal Logic (LTL) properties

However, with the goal to understand faulty systems, we are more interested in obtaning a counterexample when our Kripke model does not satisfy a property:

Counterexample

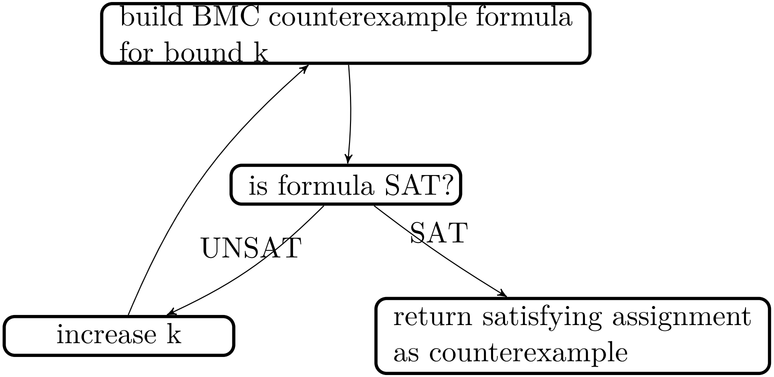
If $M \models A\,f$ is not true, then this formula

$$M \models E\neg f$$

is true, meaning that $\neg f$ is true for some execution path $s_0, s_1, \ldots, s_n$ valid for M
We call such path a counterexample (it shows f is not true for any execution)

Because such paths can have a potentially infinite number of states, we are going to consider finding such counterexamples but with a bounded number of states

BMC for Linear Temporal Logic (LTL) properties


build BMC counterexample formula for bound k
is formula SAT?
UNSAT
SAT
increase k
return satisfying assignment as counterexample

But this is a semidecision procedure:

- if there is a counterexample, it will be found for some k
- If M satisfies the property formula, it will increase k indefinitely (no counterexample is ever found)

BMC as Boolean Satisfiability (SAT)

We can encode the problem of finding a counterexample execution path (bounded by k) as a SAT problem:

1. Encoding execution paths:

$$\mathrm{path_k}(s_0, s_1, \ldots, s_k) = S_0(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1})$$

2. Each state s_i in the path is a vector with a copy of the propositional variables that represent the state i in the path. Any assignment to the variables of the vectors $s_0, s_1, \ldots, s_k$ that satisfies the previous formula is a valid execution path of length k.

BMC as Boolean Satisfiability (SAT)

3. Add constraints to encode the existence of a counterexample. For example, for a global property p(s), we say that there is a path where p is not true always by:

$$\mathrm{path}_k(s_0, s_1, \ldots, s_k) \wedge \bigvee_{i=0}^{k} \neg p(s_i)$$

where $p(s_i)$ is the expression of boolean property p expressed with the variables of state i.

Static Analysis in CBMC

So, if we are able to obtain a Kripke representation of a Ansi-C program (for execution traces of bounded length), we can analyze if the valid execution traces of an Ansi-C program satisfy properties expressed over the set of propositional variables used to label the states

Using the previous encoding scheme, we have an implicit model for the execution of the program, as we do not generate all the possible execution traces (paths) of the program. Instead, we use vectors of state variables that represent the possible states in a possible execution trace

For doing this, we have to represent the execution of the program in a transition relation $R(s_{i-1}, s_i)$ to represent it as a Kripke structure, and then create the boolean encoding of $R(s_{i-1}, s_i)$ as a set of constraints, for each value of $i \in [1, k]$

Static Analysis for Ansi-C in CBMC

From a C program to an equivalent set of equations for analysing execution traces of deep at most k:

1. Get an equivalent version that uses only while, if, goto, and assignments
2. All the while loops are unwounded k times, so a loop is replaced by k nested if (conditional) sentences
3. Rename each variable so that each renamed variable is assigned only once
4. Transform the resulting program as a set of equations with the operator $C(\text{program}, \text{guard})$.

We will introduce the operator $C(\text{program}, \text{guard})$ through the following examples

Static Analysis for Ansi-C in CBMC - Example 1

Original program:

```
x   =   1;
y   =   2 * x;
y   =   y + 4;
```

Static Analysis for Ansi-C in CBMC - Example 1

After renaming variables (step 3):

$$\begin{aligned} x_0 &= 1; \\ y_0 &= 2 * x_0; \\ y_1 &= y_0 + 4; \end{aligned}$$

As we have only a sequence of assignments, in this case the recursive application of the operator $\mathrm{C(program, guard)}$ is:

$$\begin{array}{l} C(x_0 = 1; y_0 = 2 * x_0; y_1 = y_0 + 4; \mathrm{true}) = \\ (x_0 = 1) \wedge C(y_0 = 2 * x_0; y_1 = y_0 + 4; \mathrm{true}) = \\ (x_0 = 1) \wedge (y_0 = 2 * x_0) \wedge C(y_1 = y_0 + 4; \mathrm{true}) = \\ (x_0 = 1) \wedge (y_0 = 2 * x_0) \wedge (y_1 = y_0 + 4) \end{array}$$

The value for the boolean expression guard is set to true in the first recursive call to the operator

Static Analysis in CBMC - Example 1

A valid assignment to the resulting set of equations represents a valid execution trace: $(x_0 = 1, y_0 = 2, y_1 = 6)$

Any assignment that does not satisfy some of the equations does not represent a valid execution trace: $(x_0 = 1, y_0 = 8, y_1 = 12)$

In this example, there is only one valid assignment (so only one valid execution trace)

Static Analysis in CBMC - Example 2

Original program:

```
if (x > 10)
  y = 2*x;
else
  y = 3*x;
```

Static Analysis in CBMC - Example 2

After renaming variables (step 3):

```
if  (x_0 > 10)
   y_1 = 2 * x_0;      (pi_0)
else
   y_2 = 3 * x_0;      (pi_1)
```

As we have a conditional sentence, the operator $C(\text{program}, \text{guard})$ splits into two applications, one for the true branch and other for the false branch:

$$\begin{aligned}
&C(\text{if } (x_0 > 10)\ \pi_0 \text{ else } \pi_1; \text{true}) = \\
&C(\pi_0; (x_0 > 10)) \wedge C(\pi_1; !(x_0 > 10)) = \\
&y_1 = ((x_0 > 10)\ ?\ 2 * x_0 : y_0) \wedge y_2 = (!(x_0 > 10)\ ?\ 3 * x_0 : y_1)
\end{aligned}$$

Static Analysis in CBMC - Example 2

In this example, there are as many valid execution traces as possible values for the variable x (if we omit the particular value for y_0 that is irrelevant in the execution traces):

1. $(x_0 = 1, y_1 = y_0, y_2 = 3)$
2. $(x_0 = 2, y_1 = y_0, y_2 = 6)$
3. ...
4. $(x_0 = 11, y_1 = 22, y_2 = 22)$
5. $(x_0 = 12, y_1 = 24, y_2 = 24)$
6. ...

Static Analysis in CBMC - Guarded assignment sentences

Guarded assignment sentences

So, a guarded assignment sentence $\mathrm{C}(\mathrm{v_i} = \mathrm{e}; \mathrm{b})$ is transformed into a conditional assignment expression

$$\mathrm{v_i} = (\mathrm{b}\ ?\ \mathrm{e} : \mathrm{v_{i-1}})$$

where $\mathrm{v_i}$ is assigned e if $\mathrm{b} = \mathrm{true}$ and $\mathrm{v_{i-1}}$ if e is false (where $\mathrm{v_{i-1}}$ is the previously renamed version of v)

Static Analysis in CBMC - Example 3

Original program:

```
if (x > 10)
  y = 2*x;
else
  y = 3*x;
x = 2*y;
```

Static Analysis in CBMC - Example 3

After renaming variables (step 3):

```
if  ($x_0 > 10$)
    $y_1 = 2 * x_0$;        ($\pi_0$)
else
    $y_2 = 3 * x_0$;        ($\pi_1$)
$x_1 = 2 * y_2$;        ($\pi_2$)
```

$$
\begin{aligned}
&C((\text{if } (x_0 > 10)\ \pi_0 \text{ else } \pi_1; \pi_2); \text{true}) = \\
&C(\pi_0; (x_0 > 10)) \wedge C(\pi_1; !(x_0 > 10)) \wedge C(\pi_2; \text{true}) = \\
&y_1 = ((x_0 > 10)\ ?\ 2 * x_0 : y_0) \wedge \\
&y_2 = (!(x_0 > 10)\ ?\ 3 * x_0 : y_1) \wedge \\
&x_1 = 2 * y_2
\end{aligned}
$$

Static Analysis in CBMC - Example 3

Observe that the value for x_1 is always computed, considering the last equation, from the value of y_2:

1. $(x_0 = 1, y_1 = y_0, y_2 = 3, x_1 = 6)$
2. $(x_0 = 2, y_1 = y_0, y_2 = 6, x_1 = 12)$
3. ...
4. $(x_0 = 11, y_1 = 22, y_2 = 22, x_1 = 44)$
5. $(x_0 = 12, y_1 = 24, y_2 = 24, x_1 = 48)$
6. ...

This works because y_2 is always equal to the last value assigned to y in the execution trace (up to the third step)

Static Analysis in CBMC - Example 4

Original program:

```
if (x > 10)
   if (y > 10) y = x*y; else y = 2*x*y;
else
  y = 3*x;
x = 2 * y;
```

Static Analysis in CBMC - Example 4

After renaming variables (step 3):

```
if  ($x_0 > 10$)
   if  ($y_0 > 10$)  $y_1 = x_0 * y_0$;  else  $y_2 = 2 * x_0 * y_1$;    ($\pi_0$)
else
   $y_3$ = $3 * x_0$;     ($\pi_1$)
$x_1$ = $2 * y_3$;      ($\pi_2$)
```

$$C((\text{if } (x_0 > 10)\ \pi_0 \text{ else } \pi_1; \pi_2); \text{true}) =$$

$$C(\pi_0; (x_0 > 10)) \wedge C(\pi_1; !(x_0 > 10)) \wedge C(\pi_2; \text{true}) =$$

$$(C((\text{if } (y_0 > 10)\ y_1 = x_0 * y_0;\ \text{else } y_2 = 2 * x_0 * y_1; (x_0 > 10)) \wedge$$
$$y_3 = (!(x_0 > 10)\ ?\ 3 * x_0 : y_2) \wedge$$
$$x_1 = 2 * y_3)$$

Static Analysis in CBMC - Example 4

Continuation of the application of operator C:

$\mathrm{C}((\mathrm{if}\ (x_0 > 10)\ \pi_0\ \mathrm{else}\ \pi_1; \pi_2); \mathrm{true}) =$

$\mathrm{C}(\pi_0; (x_0 > 10)) \wedge \mathrm{C}(\pi_1; !(x_0 > 10)) \wedge \mathrm{C}(\pi_2; \mathrm{true}) =$

$(\mathrm{C}((\mathrm{if}\ (y_0 > 10)\ y_1 = x_0 * y_0;\ \mathrm{else}\ y_2 = 2 * x_0 * y_1; (x_0 > 10)) \wedge$
$y_3 = (!(x_0 > 10)\ ?\ 3 * x_0 : y_2) \wedge$
$x_1 = 2 * y_3) \ =$

$(y_1 = (((x_0 > 10) \wedge (y_0 > 10))\ ?\ x_0 * y_0 : y_0) \wedge$
$y_2 = (((x_0 > 10) \wedge !(y_0 > 10))\ ?\ 2 * x_0 * y_1 : y_1) \wedge$
$y_3 = (!(x_0 > 10)\ ?\ 3 * x_0 : y_2) \wedge$
$x_1 = 2 * y_3)$

Static Analysis in CBMC - Example 4

Consider the different groups of execution traces:

1. Execution traces for $(x_0 > 10) \wedge (y_0 > 10)$. In these cases, the equations force:
   $y_1 = x_0 * y_0 \wedge y_2 = y_1 \wedge y_3 = y_2 \wedge x_1 = 2 * y_3$
2. Execution traces for $(x_0 > 10) \wedge !(y_0 > 10)$. In these cases, the equations force:
   $y_1 = y_0 \wedge y_2 = 2 * x_0 * y_1 \wedge y_3 = y_2 \wedge x_1 = 2 * y_3$
3. Execution traces for $!(x_0 > 10)$. In these cases, the equations force:
   $y_1 = y_0 \wedge y_2 = y_1 \wedge y_3 = 3 * x_0 \wedge x_1 = 2 * y_3$

In any case, y_3 is always equal to the last value assigned to y
But the equational representation of the program obtained with static analysis allows us to reason about the values of the variables in any intermediate state of the program !

Static Analysis in CBMC - Example 5

Original program:

```
x = 2;
r = 0;
m = 0;
while (r < x) {
   m = m + y;
   r = r + 1;
}
```

Static Analysis in CBMC - Example 5

After unwinding the loop k = 3 times (step 2):

```
x = 2;
r = 0;
m = 0;
if (r < x) {
  m = m + y;
  r = r + 1;
  if (r < x) {
    m = m + y;
    r = r + 1;
    if (r < x) {
      m = m + y;
      r = r + 1;
    }
  }
}
```

Static Analysis in CBMC - Example 5

After renaming variables (step 3):

```
$x_1 = 2$; $r_1 = 0$; $m_1 = 0$;   $(\pi_0)$
if ($r_1 < x_1$) {
  $m_2 = m_1 + y_0$; $r_2 = r_1 + 1$;   $(\pi_1)$
  if ($r_2 < x_1$) {
    $m_3 = m_2 + y_0$; $r_3 = r_2 + 1$;   $(\pi_2)$
      if ($r_3 < x_1$) {
        $m_4 = m_3 + y_0$; $r_4 = r_3 + 1$;   $(\pi_3)$
      }
  }
}
```

Static Analysis in CBMC - Example 5

Application of the operator C:

$$C((\pi_0;\ \mathrm{if}(r_1 < x_1)\{\pi_1;\ \mathrm{if}(r_2 < x_1)\{\pi_2;\ \mathrm{if}(r_3 < x_1)\pi_3;\ \}\}); \mathrm{true}) =$$

$$((x_1 = 2) \wedge (r_1 = 0) \wedge (m_1 = 0) \wedge$$
$$C(\pi_1; (r_1 < x_1)) \wedge$$
$$C(\pi_2; (r_1 < x_1) \wedge (r_2 < x_1)) \wedge$$
$$C(\pi_3; (r_1 < x_1) \wedge (r_2 < x_1) \wedge (r_3 < x_1)) =$$

$$((x_1 = 2) \wedge (r_1 = 0) \wedge (m_1 = 0) \wedge$$
$$m_2 = ((r_1 < x_1)\ ?\ m_1 + y_0 : m_1) \wedge r_2 = ((r_1 < x_1)\ ?\ r_1 + 1 : r_1) \wedge$$
$$m_3 = (((r_1 < x_1) \wedge (r_2 < x_1))\ ?\ m_2 + y_0 : m_2) \wedge$$
$$r_3 = (((r_1 < x_1) \wedge (r_2 < x_1))\ ?\ r_2 + 1 : r_2) \wedge$$
$$m_4 = (((r_1 < x_1) \wedge (r_2 < x_1)) \wedge (r_3 < x_1)\ ?\ m_3 + y_0 : m_3) \wedge$$
$$r_4 = (((r_1 < x_1) \wedge (r_2 < x_1) \wedge (r_3 < x_1))\ ?\ r_3 + 1 : r_3)$$

Static Analysis in CBMC - Example 6

Exercise: Perform the static analysis transformation of the following program:

```
x = 2*y;
y =  y+x;
if (x > y)
   x = x + 1
else
   y = y + 1
m = x*y;
```

Static Analysis in CBMC - Arrays and Dynamic Memory

This is not the end of the story: What about other features of Ansi-C like arrays and dynamic memory ?

Check the paper of the CBMC tool to read about how the tool handles these features using a representation of array and pointer expressions based on uninterpreted functions

Checking Boolean Properties

Once we have a way to enumerate all the valid execution traces (up to bounded deep k) of a C program with a set of equations, the next step is to add properties we would like to check in particular states

We will introduce the properties into the C program using the macro assert(condition)

The intended meaning of the assert macro is that we are expresing that any execution trace that reaches the point where is located the macro should satisfy the condition

Checking Boolean Properties

The set of assertions in the program will generate a property formula similar to the one obtained with the operator $\mathrm{C(program, guard)}$, but this time equations will be generated only each time we reach an assert macro

The property formula is generated by the application of the $\mathrm{P(program, guard)}$ operator:

- Sequential composition:
  $\mathrm{P}(\pi_1;\pi_2\ ;\ \mathrm{guard}) = \mathrm{P}(\pi_1\ ;\ \mathrm{guard}) \wedge \mathrm{P}(\pi_2\ ;\ \mathrm{guard})$
- Conditional: $\mathrm{P}(\mathrm{if\ (b)}\ \pi_1\ \mathrm{else}\ \pi_2\ ;\ \mathrm{guard}) =$
  $\mathrm{P}(\pi_1\ ;\ \mathrm{b} \wedge \mathrm{guard}) \wedge \mathrm{P}(\pi_2\ ;\ \neg\mathrm{b} \wedge \mathrm{guard})$
- Assert: $\mathrm{P}(\mathrm{assert(b)}\ ;\ \mathrm{guard}) = (\mathrm{guard} \Rightarrow \mathrm{b})$

Observe that the only instruction that generates code for P is assert. The other cases simply accumulate the guard conditions that appear in different subbranches of the execution tree

Checking Boolean Properties - Example

Original program:

```
if (x > 10) {
  y = 2*x;
  assert( y > x+1 ); }
else {
  y = 3*x;
  assert( y != x );   }
assert( y > x );
```

Checking Boolean Properties - Example

After renaming variables (step 3):

```
if  ($x_0 > 10$)
 $y_1 = 2 * x_0;$  assert($y_1 > x_0 + 1$);    ($\pi_0$)
else
   $y_2 = 3 * x_0;$  assert($y_2 != x_0$);    ($\pi_1$)
  assert(  $y_2 > x_0$  );
```

Checking Boolean Properties - Example

As we have a conditional sentence, the operator $\mathrm{P}(\mathrm{program}, \mathrm{guard})$ splits into two applications, one for the true branch and other for the false branch:

$$\mathrm{P}((\mathrm{if}\ (x_0 > 10)\ \pi_0\ \mathrm{else}\ \pi_1; \mathrm{assert}(y_2 > x_0)); \mathrm{true}) =$$

$$\mathrm{P}(\pi_0; (x_0 > 10)) \wedge \mathrm{P}(\pi_1; !(x_0 > 10)) \wedge (\mathrm{true} \Rightarrow (y_2 > x_0)) =$$

$$((x_0 > 10) \Rightarrow (y_1 > x_0 + 1))\wedge$$
$$(!(x_0 > 10) \Rightarrow (y_2! = x_0))\wedge$$
$$(\mathrm{true} \Rightarrow (y_2 > x_0))$$

Checking Boolean Properties - Final Formula

Once we get the formula that encodes the execution traces (the one obtained with $\mathrm{C}(\mathrm{program}, \mathrm{true})$) and the formula that encodes the guarded properties to check ($\mathrm{P}(\mathrm{program}, \mathrm{true})$), we combine them with the global formula:

$$\mathrm{C}(\mathrm{program}, \mathrm{guard}) \Rightarrow \mathrm{P}(\mathrm{program}, \mathrm{true})$$

Then, the program will satisfy all the properties encoded in $\mathrm{P}(\mathrm{program}, \mathrm{true})$ if and only if that formula is valid: any valid execution trace (that satisfies $\mathrm{C}(\mathrm{program}, \mathrm{guard})$) satisfies also the property formula $\mathrm{P}(\mathrm{program}, \mathrm{true})$

Checking Boolean Properties - Final Formula

But remember that we explained at the begining that if we are interested on finding an explanation of the causes of a buggy system, we can instead work with the contrary (negated) version of the previous formula, to get a counterexample (execution trace) that DOES NOT satisfy some of the assertions when the system is buggy. That is, we check the satisfiability of the formula:

$$\neg(\mathrm{C}(\mathrm{program}, \mathrm{guard}) \Rightarrow \mathrm{P}(\mathrm{program}, \mathrm{true})) \equiv \\ \mathrm{C}(\mathrm{program}, \mathrm{guard}) \wedge \neg\mathrm{P}(\mathrm{program}, \mathrm{true})$$

Then, if this formula is:

- unsatisfiable, it means that there are no valid execution traces that violate the property formula P (our program works as expected up to executions traces with bound k).
- satisfiable, the solution represents a valid execution trace that violates at least one of the assertions of the program.