CBMC

A Bounded Model Checker for C programs

Maria João Frade

HASLab - INESC TEC
Dep. Informática, Universidade do Minho

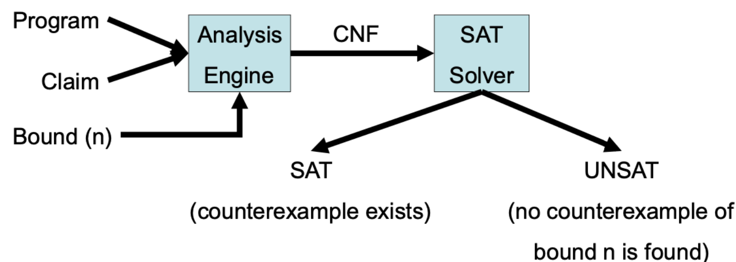
(First part is based on slides by Arie Gurfinkel.)

CBMC — C Bounded Model Checker

- Developed at CMU by Daniel Kroening (Oxford) and Edmund Clarke (CMU).
- Available at: http://www.cprover.org/cbmc/
- CBMC demonstrates the violation of assertions in C programs, or proves safety of the assertions under a given bound.
- CBMC implements a bit-precise translation of an input C program, annotated with assertions and with loops unrolled to a given depth, into a formula. If the formula is satisfiable, then an execution leading to a violated assertion exists.
- CBMC is not able to prove correctness for programs with unbounded loops in general, but is very useful for bug catching.

Bug catching with CBMC

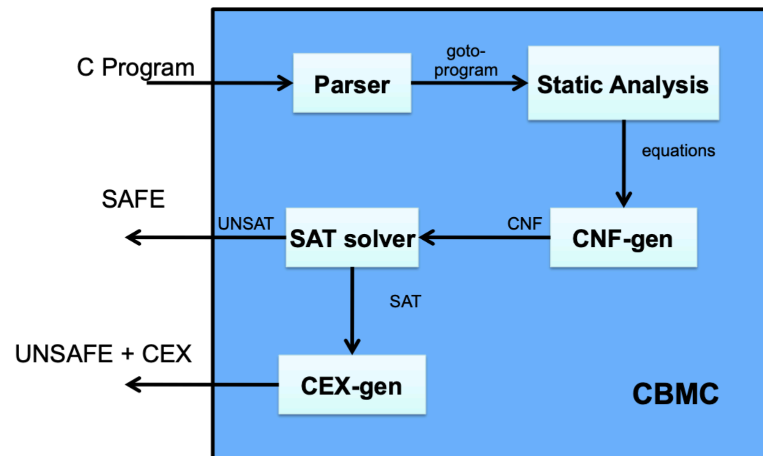
- **Main Ideia:** Given a program and a claim use a SAT solver to find whether there exists an execution that violates the claim.
  - Arbitrary ANSI-C programs (with bitvector arithmetic, dynamic memory, pointers, type casts, etc).
  - Simple Safety Claims (such as, array bound checks, pointer checks, division by zero, arithmetic overflow, user supplied assertions).
- Loops must be bounded (i.e., the analysis is incomplete).
- The analysis is completely automated.



How does CBMC work

- Transform a program into a set of equations.
  1. Simplify control flow
  2. Unwind all the loops
  3. Convert into Static Single Assignment (SSA) form
  4. Convert into equations
  5. Bit-blast
- Solve the equations with a SAT solver.
- Convert the SAT assignment (if any) into a counterexample.

How does CBMC work



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`

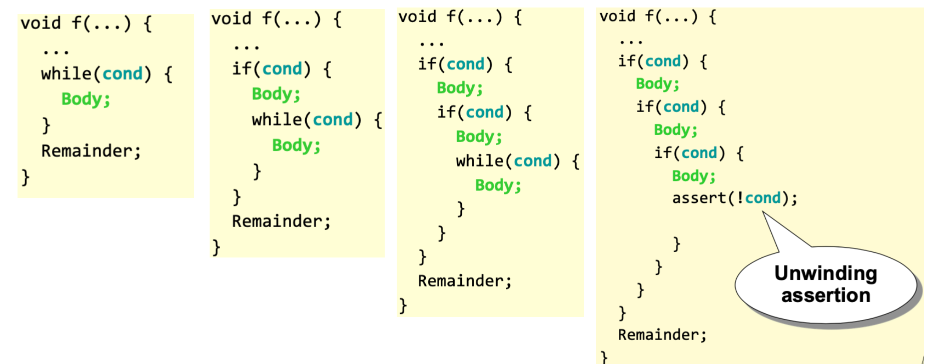
- All loops are simplified into one form

  `for`, `do-while` replaced by `while`

Loop unwinding

- All loops are unwound.
  - different unwinding bounds for different loops can be used
  - 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



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

Example of a sufficient loop unwinding

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

unwind = 3

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

Unwinding assertion

Example of a insufficient loop unwinding

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

unwind = 3

```
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;
}
```

Unwinding assertion

Transforming loop-free programs into equations

When a variable is assigned multiple times, use a new variable for the LHS of each assignment.

This is called Static Single Assignment (SSA) form.

Program

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

SSA Program

$x_1 = x_0 + y_0;$
$x_2 = x_1 * 2;$
$a_1[i_0] = 100;$

Transforming loop-free programs into equations

What about conditionals?

For each join point, add new variables with selectors.

Program

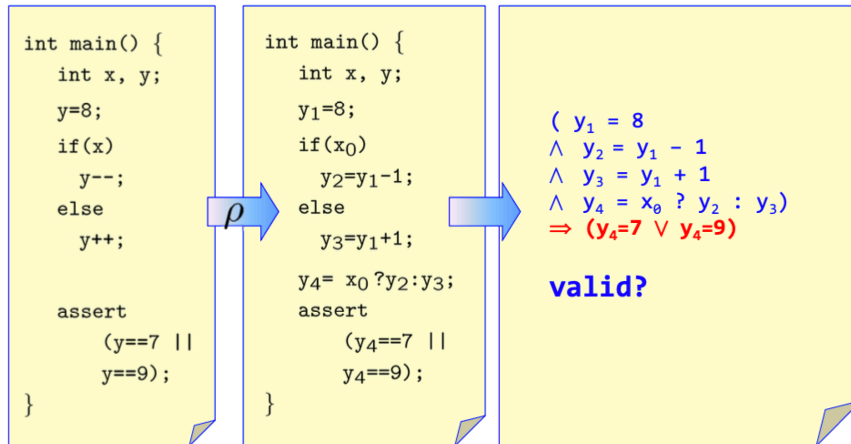
```
if (v)
  x = y;
else
  x = z;

w = x;
```

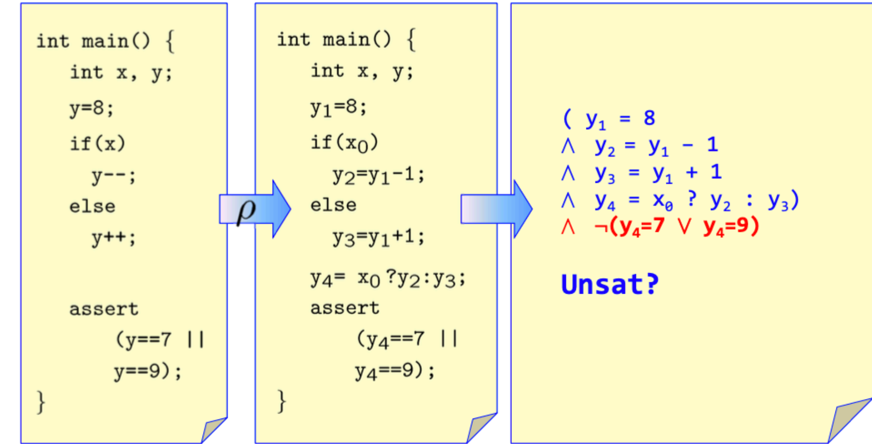
SSA Program

$\text{if } (v_0)$
$\quad x_1 = y_0;$
else
$\quad x_2 = z_0;$
$x_3 = v_0 \ ? \ x_1 : x_2;$
$w_1 = x_3;$

Example



Example



Bit-basting

- To check satisfiability of the formulas the usual approach is to replace the arithmetic operators by circuit equivalents to obtain a propositional formula, which is then passed to a SAT solver.
- This approach is called 'bit-blasting' or 'bit-flattening', as the word-level structure is lost.
- CBMC also supports SMT solvers as back-ends. In this case a bit-vector theory is used.

From programming to modelling

Extend C programming language with 3 modelling features.

Assertions

`assert(e)` – aborts an execution when **e** is false, no-op otherwise.

Non-determinism

`nondet_int()` – returns a non-deterministic integer value.

Assumptions

`__CPROVER_assume(e)` – “ignores” execution when **e** is false.

Restricts the program traces that are considered and allows **assume-guarantee** reasoning.

Assume-Guarantee reasoning

(A) Is `foo` correct assuming `p` is not NULL?

```
int foo (int* p) { __CPROVER_assume(p!=NULL); … }
```

(G)Is `foo` guaranteed to be called with a non-NULL argument?

```
void main(void) {
  …
  assert (x!=NULL);// foo(x);
  …
  assert (y!=NULL); //foo(y);
  …}
```


Dangers of unrestricted assumptions

Assumptions can lead to vacuous satisfaction.

This program is passed by CBMC!

```
if (x > 0) {
  __CPROVER_assume (x < 0);
  assert (0); }
```

Assumptions can be used to restrict the program traces that are considered.

```
x = nondet_int ();
y = nondet_int ();
__CPROVER_assume (x < y);
```


CBMC by example

Assertions [ex1.c]

CBMC checks assertions as defined by the ANSI-C standard.
The assert statement takes a Boolean condition, and CBMC checks that this condition is true for all runs of the program.

```
void main (void)
{
  int x;
  int y=8, z=0, w=0;

  if (x)
    z = y - 1;
  else
    w = y + 1;

  assert (z == 7 || w == 9);
}
```

```
$ cbmc ex1.c
$ cbmc ex1.c --show-vcc
```

ex1.c outcome

$ cbmc ex1.c

```
CBMC version 5.10 (cbmc-5.10) 64-bit x86_64 macos
Parsing ex1.c
Converting
Type-checking ex1
file ex1.c line 11 function main: function `assert' is not declared
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
size of program expression: 43 steps
simple slicing removed 2 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
141 variables, 39 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime decision procedure: 0.00294518s

** Results:
[main.assertion.1] assertion z == 7 || w == 9: SUCCESS

** 0 of 1 failed (1 iteration)
VERIFICATION SUCCESSFUL
```


ex1.c outcome

$ cbmc ex1.c --show-vcc

```
(..)
Generated 1 VCC(s), 1 remaining after simplification

VERIFICATION CONDITIONS:

file ex1.c line 11 function main
assertion z == 7 || w == 9
(…)
{-12} y!0@1#2 == 8
{-13} z!0@1#2 == 0
{-14} w!0@1#2 == 0
{-15} \guard#1 == !(x!0@1#1 == 0)
{-16} z!0@1#3 == 7
{-17} z!0@1#4 == 0
{-18} w!0@1#3 == 9
{-19} w!0@1#4 == (\guard#1 ? 0 : 9)
{-20} z!0@1#5 == (\guard#1 ? 7 : 0)
|--------------------------
{1} w!0@1#4 == 9 || z!0@1#5 == 7
```


Alternatively: use SMT solver

$ cbmc -z3 ex1.c

```
(…)
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to SMT2 QF_AUFBV using Z3
converting SSA
Running SMT2 QF_AUFBV using Z3
Runtime decision procedure: 0.0279709s

** Results:
[main.assertion.1] assertion z == 7 || w == 9: SUCCESS

** 0 of 1 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

$ cbmc -cvc4 ex1.c

```
(…)
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to SMT2 QF_AUFBV using CVC4
converting SSA
Running SMT2 QF_AUFBV using CVC4
Runtime decision procedure: 0.0147195s

** Results:
[main.assertion.1] assertion z == 7 || w == 9: SUCCESS

** 0 of 1 failed (1 iteration)
VERIFICATION SUCCESSFUL
```


ex2.c

```
void main (void)
{
  int x;
  int y=8, z=0, w=0;

  if (x)
    z = y - 1;
  else
    w = y + 1;

  assert (z == 5 || w == 9);
}
```

$ cbmc ex2.c

$ cbmc ex2.c --trace

ex2.c outcome

$ cbmc ex2.c

```
(…)
Solving with MiniSAT 2.2.1 with simplifier
141 variables, 50 clauses
SAT checker: instance is SATISFIABLE
Runtime decision procedure: 0.000694225s

** Results:
[main.assertion.1] assertion z == 5 || w == 9: FAILURE

** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```


ex2.c outcome

$ cbmc ex2.c --trace

```
(…)
Solving with MiniSAT 2.2.1 with simplifier
141 variables, 50 clauses
SAT checker: instance is SATISFIABLE
Runtime decision procedure: 0.000671996s

** Results:
[main.assertion.1] assertion z == 5 || w == 9: FAILURE

Trace for main.assertion.1:

State 17 file ex2.c line 3 function main thread 0
----------------------------------------------------
  x=33554432 (00000010 00000000 00000000 00000000)

State 18 file ex2.c line 4 function main thread 0
----------------------------------------------------
  y=0 (00000000 00000000 00000000 00000000)

State 19 file ex2.c line 4 function main thread 0
----------------------------------------------------
  y=8 (00000000 00000000 00000000 00001000)
```


ex2.c outcome

$ cbmc ex2.c --trace

```
(…)
State 23 file ex2.c line 4 function main thread 0
----------------------------------------------------
  w=0 (00000000 00000000 00000000 00000000)

State 25 file ex2.c line 7 function main thread 0
----------------------------------------------------
  z=7 (00000000 00000000 00000000 00000111)

Violated property:
  file ex2.c line 11 function main
  assertion z == 5 || w == 9
  z == 5 || w == 9


** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```


ex3.c

CBMC can ignore user assertions.

```
void main (void)
{
  int x, y;

  x = x + y;
  if (x != 3) x = 2;
  else  x++;

  assert (x <= 3);
}
```

$ cbmc ex3.c --show-vcc

$ cbmc ex3.c

$ cbmc ex3.c --no-assertions

$ cbmc ex3.c --no-assertions --show-vcc

Checking overflow

But the code can be automatically instrumented.

$ cbmc ex3.c --signed-overflow-check --no-assertions --trace

```
State 17 file ex3.c line 3 function main thread 0
----------------------------------------------------
  x=-1610612735 (10100000 00000000 00000000 00000001)

State 18 file ex3.c line 3 function main thread 0
----------------------------------------------------
  y=-2147483648 (10000000 00000000 00000000 00000000)

Violated property:
  file ex3.c line 5 function main
  arithmetic overflow on signed + in x + y
  !overflow("+", signed int, x, y)

** 1 of 2 failed (2 iterations)
VERIFICATION FAILED
```


Workflow

- Internally CBMC runs `goto-cc` to produce a representation of the control flow graph of the program.

  $ goto-cc ex3.c -o ex3.gb

- Then the instrumentation tool `goto-instrument` automatically add assertions to be checked.

  $ goto-instrument --signed-overflow-check ex3.gb ex3.instr.gb

- And finally the assertions are checked.

  $ cbmc ex3.instr.gb

Seeing the properties

$ cbmc ex3.c --signed-overflow-check --show-properties

```
Property main.overflow.1:
  file ex3.c line 5 function main
  arithmetic overflow on signed + in x + y
  !overflow("+", signed int, x, y)

Property main.overflow.2:
  file ex3.c line 7 function main
  arithmetic overflow on signed + in x + 1
  !overflow("+", signed int, x, 1)

Property main.assertion.1:
  file ex3.c line 9 function main
  assertion x <= 3
  x <= 3
```


Seeing the instrumented code

$ cbmc ex3.c --signed-overflow-check --show-goto-functions

```
main /* main */
        // 0 file ex3.c line 3 function main
        signed int x;
        // 1 file ex3.c line 3 function main
        signed int y;
        // 2 file ex3.c line 5 function main
        ASSERT !overflow("+", signed int, x, y) // arithmetic overflow on signed + in x + y
        // 3 file ex3.c line 5 function main
        x = x + y;
        // 4 file ex3.c line 6 function main
        IF !(x != 3) THEN GOTO 1
        // 5 file ex3.c line 6 function main
        x = 2;
        // 6 file ex3.c line 6 function main
        GOTO 2
        // 7 file ex3.c line 7 function main
     1: ASSERT !overflow("+", signed int, x, 1) // arithmetic overflow on signed + in x + 1
        // 8 file ex3.c line 7 function main
        x = x + 1;
        // 9 file ex3.c line 9 function main
     2: ASSERT x <= 3 // assertion x <= 3
        // 10 file ex3.c line 10 function main
        dead y;
        // 11 file ex3.c line 10 function main
        dead x;
        // 12 file ex3.c line 10 function main
        END_FUNCTION
```

Entrypoints [ex4.c]

```
int fun (int a, int b)
{
  int c = a+b;

  if (a>0 || b>0)
    c = 1/(a+b);
  return c;
}
```

$ cbmc ex4.c

$ cbmc ex4.c --function fun

$ cbmc ex4.c --function fun --div-by-zero-check

$ cbmc ex4.c --function fun --div-by-zero-check --trace

Checking division by zero

$ cbmc ex4.c --function fun --div-by-zero-check --trace

```
** Results:
[fun.division-by-zero.1] division by zero in 1 / (a + b): FAILURE

Trace for fun.division-by-zero.1:
(…)
State 23 file ex4.c line 1 thread 0
----------------------------------------------------
  a=-1073741808 (11000000 00000000 00000000 00010000)

State 24 file ex4.c line 1 thread 0
----------------------------------------------------
  b=1073741808 (00111111 11111111 11111111 11110000)

State 25 file ex4.c line 3 function fun thread 0
----------------------------------------------------
  c=0 (00000000 00000000 00000000 00000000)

State 26 file ex4.c line 3 function fun thread 0
----------------------------------------------------
  c=0 (00000000 00000000 00000000 00000000)

Violated property:
  file ex4.c line 6 function fun
  division by zero in 1 / (a + b)
  !(a + b == 0)

** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```


ex5.c

```
void main ()
{
  char c;
  long l;
  int i;

  l = c = i;
  assert (l==i);
}
```

$ cbmc ex5.c

$ cbmc ex5.c --trace

ex5.c outcome

$ cbmc ex5.c --trace

```
** Results:
[main.assertion.1] assertion l == (signed long int)i: FAILURE
Trace for main.assertion.1:

State 17 file ex5.c line 3 function main thread 0
----------------------------------------------------
  c=0 (00000000)

State 18 file ex5.c line 4 function main thread 0
----------------------------------------------------
  l=0l (00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000)

State 19 file ex5.c line 5 function main thread 0
----------------------------------------------------
  i=262144 (00000000 00000100 00000000 00000000)

State 20 file ex5.c line 7 function main thread 0
----------------------------------------------------
  c=0 (00000000)

State 21 file ex5.c line 7 function main thread 0
----------------------------------------------------
  l=0l (00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000)

Violated property:
  file ex5.c line 8 function main
  assertion l == (signed long int)i
  l == (signed long int)i
```

Array bounds [ex6.c]

```
int puts (const char *s);

int main (int argc, char **argv)
{
  int i;

  if (argc >= 1)
    puts (argv[2]);
}
```

$ cbmc ex6.c

$ cbmc ex6.c --bounds-check --pointer-check

$ cbmc ex6.c --bounds-check --pointer-check --trace

ex6.c outcome

$ cbmc ex6.c --bounds-check --pointer-check --trace

```
(…)
Trace for main.pointer_dereference.6:
State 17  thread 0
----------------------------------------------------
  INPUT argc: 1 (00000000 00000000 00000000 00000001)

State 18  thread 0
----------------------------------------------------
  argv'[1]=((char *)NULL) (00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000)
State 21 file ex6.c line 3 thread 0
----------------------------------------------------
  argc=1 (00000000 00000000 00000000 00000001)

State 22 file ex6.c line 3 thread 0
----------------------------------------------------
  argv=argv' (00000010 00000000 00000000 00000000 00000000 00000000 00000000 00000000)

State 23 file ex6.c line 5 function main thread 0
----------------------------------------------------
  i=0 (00000000 00000000 00000000 00000000)

Violated property:
  file ex6.c line 8 function main
  dereference failure: pointer outside object bounds in argv[(signed long int)2]
  16l + POINTER_OFFSET(argv) >= 0l && OBJECT_SIZE(argv) >= 24ul + (unsigned long
int)POINTER_OFFSET(argv) || DYNAMIC_OBJECT(argv)

** 1 of 7 failed (2 iterations)
VERIFICATION FAILED
```


Array bounds [ex7.c]

```
int puts (const char *s);

int main (int argc, char **argv)
{
  int i;

  if (argc >= 2)
    puts (argv[2]);
}
```

$ cbmc ex7.c --bounds-check --pointer-check

```
(…)
[main.pointer_dereference.7] dereference failure: invalid integer address
in argv[(signed long int)2]: SUCCESS

** 0 of 7 failed (1 iteration)
VERIFICATION SUCCESSFUL
```


ex8.c

```
int array[10];

int sum ()
{
  unsigned i, sum;

  sum = 0;
  for (i = 0; i <= 10; i++)
    sum += array [i];
}
```

$ cbmc ex8.c --function sum

$ cbmc ex8.c --function sum --bounds-check

$ cbmc ex8.c --function sum --bounds-check --trace

ex8.c outcome

$ cbmc ex8.c --function sum --bounds-check --trace

```
(…)

State 59 file ex8.c line 9 function sum thread 0
----------------------------------------------------
  sum$$1$$sum=0u (00000000 00000000 00000000 00000000)

State 60 file ex8.c line 8 function sum thread 0
----------------------------------------------------
  i=10u (00000000 00000000 00000000 00001010)

Violated property:
  file ex8.c line 9 function sum
  array `array' upper bound in array[(signed long int)i]
  !((signed long int)i >= 10l)

** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```


Loop unwinding [ex9.c]

```
int binsearch (int x)
{
  int a[16];
  signed low = 0, high = 16;

  while (low < high) {
    signed middle = low + ((high - low) >> 1);
    if (a[middle] < x)  high = middle;
    else if (a[middle] > x)  low = middle + 1;
    else return middle;
  }
 return -1;
}
```

$ cbmc ex9.c --function binsearch
--bounds-check --pointer-check

ex9.c outcome

$ cbmc ex9.c --function binsearch
--bounds-check --pointer-check

CBMC does not stop! The loop is being infinitely unwound.
We must provide the number of iterations to be unwound.

$ cbmc ex9.c --function binsearch
--bounds-check --pointer-check
--unwind 4

```
(…)
[binsearch.array_bounds.4] array `a' upper bound in a[(signed long int)middle]: SUCCESS

** 0 of 4 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

The above verification simply means that no array bounds are violated in the first 4 iterations os the loop!

Unwinding assertion

To see if the entire set of possible executions is being covered, we must generate ***unwinding assertions***.

$ cbmc ex9.c --function binsearch
--bounds-check --pointer-check
--unwind 4 --unwinding-assertions

```
(…)

[binsearch.array_bounds.4] array `a' upper bound in a[(signed long int)middle]: SUCCESS
[binsearch.unwind.0] unwinding assertion loop 0: FAILURE

** 1 of 5 failed (2 iterations)
VERIFICATION FAILED
```

Unwinding assertion

The failure of the "unwinding assertion" means that it is not guaranteed that the number k of iterations given as parameter will be sufficient, i.e. some execution path may run through n>k iterations.

In this case it suffices to increase k.

```
$ cbmc ex9.c --function binsearch
             --bounds-check --pointer-check
             --unwind 6 --unwinding-assertions
```

```
(…)
** Results:
[binsearch.array_bounds.1] array `a' lower bound in a[(signed long int)middle]: SUCCESS
[binsearch.array_bounds.2] array `a' upper bound in a[(signed long int)middle]: SUCCESS
[binsearch.array_bounds.3] array `a' lower bound in a[(signed long int)middle]: SUCCESS
[binsearch.array_bounds.4] array `a' upper bound in a[(signed long int)middle]: SUCCESS
[binsearch.unwind.0] unwinding assertion loop 0: SUCCESS

** 0 of 5 failed (1 iteration)
VERIFICATION SUCCESSFUL
```


Bounded loops [ex10.c]

CBMC checks if enough unwinding is done.

```
int sumq (void)
{
  short int i, s;

  s = 0;
  for (i = 0; i <= 10; i++)
    s *= i*i;
  return s;
}
```

```
$ cbmc ex10.c --function sumq --signed-overflow-check
```

```
[sumq.overflow.3] line 6 arithmetic overflow on signed + in i + 1: SUCCESS
[sumq.overflow.1] line 7 arithmetic overflow on signed * in (signed int)i * (signed int)i: SUCCESS
[sumq.overflow.2] line 7 arithmetic overflow on signed * in s * (signed short int)((signed int)i * (signed int)i): SUCCESS

** 0 of 3 failed (1 iteration)
VERIFICATION SUCCESSFUL
```


Unbounded loops [ex11.c]

CBMC can also be used for programs with unbounded loops.

```
int sumqq (int x)
{
  short int i, s;

  s = 0;
  for (i = 0; i <= x; i++)
    s += i+i;
  return s;
}
```

```
$ cbmc ex11.c --function sumqq
              --signed-overflow-check --unwind 100
```


Unbounded loops [ex11.c]

```
$ cbmc ex11.c --function sumqq --signed-overflow-check
              --unwind 100
```

```
** Results:
(…)
[sumqq.overflow.3] arithmetic overflow on signed + in i + 1: SUCCESS

** 0 of 3 failed (1 iteration)
VERIFICATION SUCCESSFUL
```

In this case CBMC is used for bug hunting only. CBMC does not attempt to find all bugs. If you increase the bound you can find a bug.

```
$ cbmc ex11.c --function sumqq --signed-overflow-check
              --unwind 200
```

```
[sumqq.overflow.3] line 6 arithmetic overflow on signed + in i + 1: SUCCESS
[sumqq.overflow.1] line 7 arithmetic overflow on signed + in (signed int)i + (signed int)i: SUCCESS
[sumqq.overflow.2] line 7 arithmetic overflow on signed + in s + (signed short int)((signed int)i + (signed int)i): FAILURE

** 1 of 3 failed (2 iterations)
VERIFICATION FAILED
```

Inlining [ex12.c]

CBMC supports functions by inlining, and preserves the locality of the parameters and the non-static local variables by renaming.

```
int f() {
  static int s=0;
  s++;
  return s;
}

int g() {
  int l=0;
  l++;
  return l;
}
```

```
int h() {
  int x=10;
  x += f() + g();
  return x;
}

void main(void)
{
 assert(f()==1); // first call to f
 assert(f()==2); // second call to f
 assert(g()==1); // first call to g
 assert(g()==1); // second call to g
 assert(h()==14);
}
```

$ cbmc ex12.c

```
** 0 of 1 failed (1 iteration)
VERIFICATION SUCCESSFUL
Generated 5 VCC(s), 0 remaining after simplification
VERIFICATION SUCCESSFUL
```


Recursion [ex13.c]

Recursion is implemented by finite unwinding, as done for while loops.

```
int fact(int n)
{
  if (n==0)
    return 1;
  else return n*fact(n-1);
}

void main(void)
{
  fact(20);
}
```

$ cbmc ex13.c --function fact –signed-overflow-check

```
Unwinding recursion fact iteration 1
Unwinding recursion fact iteration 2
Unwinding recursion fact iteration 3
Unwinding recursion fact iteration 4
Unwinding recursion fact iteration …..
```

CBMC does not stop! The loop is being infinitely unwound.
We must provide the number of iterations to be unwound.

Recursion [ex13.c]

$ cbmc ex13.c --function fact --signed-overflow-check --unwind 5

```
[fact.overflow.1] line 5 arithmetic overflow on signed - in n - 1: FAILURE
[fact.overflow.2] line 5 arithmetic overflow on signed * in n * return_value_fact: SUCCESS
** 1 of 2 failed (2 iterations)

VERIFICATION FAILED
```

If called from main fact will be inlined and unwound.
There is no need to provide `--unwind k` switch.

$ cbmc ex13.c --signed-overflow-check

```
[fact.overflow.1] line 5 arithmetic overflow on signed - in n - 1: SUCCESS
[fact.overflow.2] line 5 arithmetic overflow on signed * in n * return_value_fact: FAILURE

** 1 of 2 failed (2 iterations)
VERIFICATION FAILED
```


Low level properties [ex14.c]

Nondeterminism can be introduced explicitly into the program by means of functions that begin with the prefix **nondet_**

```
int nondet_int();
int *p;
int global;

void f (void)
{
  int local = 10;
  int input = nondet_int();

  p = input ? &local : &global;
}
```

```
int main (void)
{
  int z;

  global = 10;
  f ();
  z = *p;
  assert (z==10);
}
```

$ cbmc ex14.c VERIFICATION FAILED **Why?**

$ cbmc ex14.c --pointer-check --no-assertions

ex15.c

```
int fun (int n)
{
  int *p, i, s=0;

  p = malloc(sizeof(int)*n);
  for (i=0; i<++n; i++)
    p[i] = 10*i;
  for (i=0; i<n; i++)
    s += p[i];
  return s;
}
```

```
int main(void)
{
  printf("%d",fun(8) + 100);
  return 0;
}
```

$ cbmc ex15.c --bounds-check --pointer-check

The loop is being infinitely unwound. **Why?**

$ cbmc ex15.c --bounds-check –pointer-check –unwind 5

`VERIFICATION SUCCESSFUL` **Why?**

$ cbmc ex15.c --bounds-check –pointer-check –unwind 10

`VERIFICATION FAILED` **Why?**

ex16.c

```
char *p = "abc";

void fun(unsigned int i)
{
  char ch;
  ch = p[i];
}
```

$ cbmc ex16.c --bounds-check --pointer-check --function fun

```
(…)
[fun.pointer_dereference.6] dereference failure: pointer outside object bounds in p[(signed long int)i]: FAILURE
[fun.pointer_dereference.7] dereference failure: invalid integer address in p[(signed long int)i]: SUCCESS

** 1 of 7 failed (2 iterations)
VERIFICATION FAILED
```


ex17.c

```
void f (unsigned int n)
{
  int *p;
  p = malloc(sizeof(int)*n);
  p[n-1] = 0;
  free(p);
}
```

$ cbmc ex17.c --function f `VERIFICATION SUCCESSFUL`

$ cbmc ex17.c --function f --bounds-check --pointer-check

`VERIFICATION FAILED` **Why?**

ex18.c

```
void f (int i)
{
  int *p, y;

  p = malloc(sizeof(int)*10);
  if (i) p = &y;
  free(p);
}
```

$ cbmc ex18.c --function f

`VERIFICATION FAILED` **Why?**

Assume-guarantee reasoning

- In addition to the assert statement, CBMC provides the **__CPROVER_assume** statement.
- As an assertion, `__CPROVER_assume` takes a Boolean expression.
- The `__CPROVER_assume` statement restricts the program traces that are considered and allows assume-guarantee reasoning.
- Intuitively, one can consider the `__CPROVER_assume` statement to abort the program successfully if the condition is false. If the condition is true, the execution continues.

ex19.c

```
int nondet_int();
int x, y;

void main (void)
{
  x = nondet_int();
  y = x+1;
  assert (y>x);
}
```

$ cbmc ex19.c

VERIFICATION FAILED **Why?**

$ cbmc ex19.c --show-vcc

```
VERIFICATION CONDITIONS:
(…)
assertion y > x
(…)
{-12} x#1 = 0
{-13} y#1 = 0
{-14} x#2 = nondet_symbol identifier="symex::nondet0"
{-15} y#2 = 1 + x#2
├──────────────────────────
{1} ¬(x#2 ≥ y#2)
```


ex20.c

```
int nondet_int();
int x, y;

void main (void)
{
  x = nondet_int();
  __CPROVER_assume (x<10);
  y = x+1;
  assert (y>x);
}
```

$ cbmc ex20.c

VERIFICATION SUCCESSFUL

$ cbmc ex20.c --show-vcc

```
VERIFICATION CONDITIONS:
(…)
assertion y > x
(…)
{-12} x#1 = 0
{-13} y#1 = 0
{-14} x#2 = nondet_symbol identifier="symex::nondet0"
{-15} ¬(x#2 ≥ 10)
{-16} y#2 = 1 + x#2
├──────────────────────────
{1} ¬(x#2 ≥ y#2)
```


Lab session