

Bounded Model Checking of ANSI-C Programs with CBMC

Ramón Béjar - DIEI - UdL

March 21, 2024

Introduction

CBMC is an automatic bounded model checker for ANSI-C programs. That is, a program for checking that certain properties hold (true) in a program. The idea is that a program that is *bug-free* should satisfy certain properties, and these are the properties we would like to check with CBMC. The kind of properties that we can check with this program are boolean expressions that we want to be true at certain points of the program, **for any possible execution path** (execution trace) of the program. A property is true for an execution path $\pi = s_0, s_1, \dots, s_n$ if it is true at any state s_i of π . However, most of the properties we will specify with CBMC, that we call *assertions*, refer to relations between variables that we are only interested to check them at such states where the variables change. So, we insert assertions in such specific points of the program where we are really worried about whether they are satisfied. Then, the properties that CBMC will check will be linked to such states associated with points of the program where assertions have been introduced.

CBMC works by creating two formulas:

- The formula $CNF(C, k)$ associated with all the execution traces for the ANSI-C program C and an *upper bound* on the maximum length of any execution trace to be considered (k).
- The formula $CNF(C, k, P)$ associated with the set of all the assertions P that need to be checked in the program C in all the execution traces of maximum length k to be considered.

Capturing execution traces with $CNF(C, k)$

The formula $CNF(C, k)$ is created from C by first unwinding loops, up to deep k or less when possible, and then renaming variables so that each variable is assigned only once (in only one instruction of the unwounded program).

For example, consider the following simple program:

```
x = 3;
y = 4;
r = 0;
m = 0;
while (r < x) {
    m = m + y;
    r = r + 1;
}
```

First, CBMC performs loop unwinding for the unique loop of the program. For this program, observe that we need to unwind the loop 3 times in order to catch all the possible execution traces¹. So, the

¹ However, as we will see later, when CBMC is not able to automatically detect the needed number of unwinds for a loop, to check whether an user specified number of unwinds U is enough for a loop, CBMC needs to unwind the loop one more time in order to be able to **prove** that U unwinds are enough. Can you guess why?

resulting program is:

```
x = 3;
y = 4;
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;
    }
  }
}
```

Next, we have to perform the renaming of all the variables as previously indicated:

```
x0 = 3;
y0 = 4;
r0 = 0;
m0 = 0;
if (r0 < x0) {
  m1 = m0 + y0;
  r1 = r0 + 1;
  if (r1 < x0) {
    m2 = m1 + y0;
    r2 = r1 + 1;
    if (r2 < x0) {
      m3 = m2 + y0;
      r3 = r2 + 1;
    }
  }
}
```

Observe that in this final transformed program, we can consider simultaneous assignments to **all its variables**, and check whether the assignment satisfies all the sentences of the program as if they were a set of equations. CBMC extracts from this final transformed program a finite state model. If we further transform the integer variables to bit vectors, is when we can finally get our desired boolean formula even in CNF format. This is the $CNF(C, k)$ formula that CBMC builds, although with many optimizations to reduce its size.

What is the minimum execution depth (k) needed for a general program in order to catch any possible bug? This is a very difficult question, and actually there is no general method to answer it for any possible input program C . This is an easy consequence of the result of *Alan Turing* about the **Halting Problem** ², as you should know. That is, there are even programs where for some inputs they never stop and no algorithm is able to correctly detect whether a program will ever stop when executed with a given input. And even for programs that always stop, there is no way to automatically discovering an upper bound on the number of needed execution steps for every input ³

So, that means that although the default running mode of CBMC is to try to discover the amount of unwinding needed for any loop of the program, so that CBMC will discover the value for k needed for a program C , it will not be able to discover this information **for each program**. Actually, we can even have very *simple* loops where it can be very difficult to bound their running time, if it depends on some inputs that come from external sources (user inputs, data files, ...). Also, some times the worst case number of unwindings needed for a loop can be so high, that the resulting transformed formula can be too large to be handled by CBMC.

For example, consider the following modified version of our first program:

```
x = get_integer_number();
y = 4;
r = 0;
m = 0;
while (r < x) {
    m = m + y;
    r = r + 1;
}
```

where the value of x is asked to the user. In that case, the loop will have also an upper bound on its number of iterations, if we consider a realistic program model with variables implemented as finite size bit vectors. But of course, the upper bound can be quite large, if we consider for example integers implemented as 32 or 64 bit vectors. For such cases, CBMC allows the user to transform a program but using a particular upper bound for each loop of the program, so CBMC will not try to check if more unwinding is necessary to catch any possible execution trace. However, as an optional safety measure, CBMC allows to introduce a security mechanism to check whether the user introduced upper bound for loop unwinding is really enough for all the loops. This is done through what is called unwinding assertions,

² Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2, 42:230–265, 1936

³ Except may be for programs with a bounded possible number of inputs where we can always execute the program with all the inputs, and obtain the length of the longest execution path. But this is not a feasible way for discovering such upper bounds.

that will be introduced in the next subsection.

Once loop unwinding and renaming of variables has been performed for the program C , the formula $CNF(C, k)$, that represents the constraints that execution paths of depth up to k must satisfy to be valid execution paths for C , can be build. In the formula the possible values of any variable after executing a particular instruction of the program (call it the state s_i of the program) are linked (constrained) by equations to the valid values of the variable just **before** the execution of that instruction (so, at state s_{i-1}). Then, the formula $CNF(C, k)$ represents a **finite state model** of the program for execution traces up to depth k .

Because we use different variables to store the value of a same variable at different states of the program, we can simulate, **and analyze**, the behaviour of the program without really executing it. So, we say that the formula allows us to perform an *static analysis* (SA) of the program (no need to execute it). More formally, consider that from the transformed program we derive, for every state s_i , the boolean expressions that must be satisfied by consistent assignments between variables in the previous state s_{i-1} and the current state s_i by a boolean formula $R(s_{i-1}, s_i)$:

The formula $R(s_{i-1}, s_i)$ will be true if the value assigned to variables at state s_{i-1} is consistent to the value assigned to variables at state s_i , considering the instruction executed when going from state s_{i-1} to state s_i .

Observe that in a deterministic program, for every assignment to the variables at s_{i-1} there will be an unique assignment to the variables at s_i that is consistent.⁴ Then, any valid execution trace s_0, s_1, \dots, s_k must satisfy (make true) the formula:

$$\bigwedge_{i=1}^k R(s_{i-1}, s_i)$$

where we assume that each state s_i is represented by the set of variables of the program at that state. If we represent any variable of the program in any state s_i as a vector of boolean variables, is when we can get the final CNF representation of the formula $CNF(C, k)$:

$$CNF(C, k) \equiv \bigwedge_{i=1}^k CNF(R(s_{i-1}, s_i))$$

where $CNF(R(s_{i-1}, s_i))$ represents the formula $R(s_{i-1}, s_i)$ but rewritten in CNF form using only boolean variables.

However, the SA process performed by CBMC is much more complex, as dynamic memory, arrays and pointers receive a particular treatment in order to be able to perform SA with them. See⁵ for more details about the whole SA process performed by CBMC.

⁴ Regular C programs are of course deterministic, even if we use pseudo-random generators, given that functions where all their inputs are known are deterministic.

⁵ Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, 2003

Filtering Buggy Execution Traces with $CNF(C, k, P)$

The previous formula, $CNF(C, k)$, captures **all** the possible execution traces (up to execution length k) of C . That is, any truth assignment that satisfies $CNF(C, k)$ encodes a valid execution trace of the program C of length at most k . However, if we are interested in checking whether they satisfy a given set of assertions, we need our second formula, the assertions formula $CNF(C, k, P)$, that will be used to impose certain restrictions on the execution paths we want to capture. The idea is that we want to focus on the possible *buggy* execution traces of the program (if any), that is, execution traces where the program is not showing the expected behaviour. The expected behaviour will be modelled as a set of assertions P .

Definition 1 *Any execution trace that violates some of the assertions in P will be called a **counterexample** for the assertions in P . A program C such that all its execution traces satisfy all the assertions in P will be considered bug free (with respect to the assertions in P).*

However, that definition implies that the set of assertions P should be complete enough in order to not miss any buggy behaviour that may cause troubles. An assertion is inserted in a specific point of the program using the following macro:

```
__CPROVER_assert( Boolean Expression, "Description of the assertion
");
```

What we are telling to CBMC is that the boolean expression of the assertion, that is build using program variables, should be true **every time** the execution of the program reaches the line of code **just before** the assertion. Observe that the same line of code can be reached several times in a same execution (for programs with loops or with functions that are called more than once). So, although an assertion is inserted in a particular point of the program, in the final finite state model of the program resulting from the transformation to a BMC problem the assertion may appear linked to several states of the program. For example, consider that in the previous example program we insert an assertion at the end of the body loop for expressing that after each iteration that equality $m = y * r$ should be true. That is:

```
x = 3;
y = 4;
r = 0;
m = 0;
while (r < x) {
    m = m + y;
```

```

    r = r + 1;
    __CPROVER_assert( m == y*r, "Invariant true after loop iteration");
}

```

Then, after the unwinding (three times) of the loop and the renaming of the program variables, the resulting transformed program (ready to extract from it a finite state model plus a set of assertions P) is the next one:

```

x0 = 3;
y0 = 4;
r0 = 0;
m0 = 0;
if (r0 < x0) {
    m1 = m0 + y0;
    r1 = r0 + 1;
    __CPROVER_assert( m1 == y0*r1, "Invariant true after loop iteration");
    if (r1 < x0) {
        m2 = m1 + y0;
        r2 = r1 + 1;
        __CPROVER_assert( m2 == y0*r2, "Invariant true after loop iteration");
        if (r2 < x0) {
            m3 = m2 + y0;
            r3 = r2 + 1;
            __CPROVER_assert( m3 == y0*r3, "Invariant true after loop iteration");
        }
    }
}
}

```

So, in the final transformed program the assertion is replicated in three different points, each one using the appropriate renamed variables. Observe that for any complete assignment to the variables of the transformed program it is possible to check whether any of the three renamed copies of the assertion are satisfied. So, from now on, when we talk about a set of assertions P linked with a transformed program C , we mean a set of **duplicated and renamed assertions**, where each assertion has been duplicated (if necessary) and renamed in the appropriate manner in the transformed program.

To build the final assertions formula $CNF(C, k, P)$, we need not only to duplicate and rename the assertions in the appropriate points of the program, but to evaluate each assertion p_i in P only when the execution trace considered (by the value assigned to the variables of the formula) reaches the assertion p_i . For that reason, the general form of an evaluation of an assertion p_i in $CNF(C, k, P)$ is a **guarded assertion**:

$$g_i \rightarrow p_i$$

where g_i denotes the boolean expression that is true whenever the execution trace reaches the point where p_i must be evaluated. For example, in the previous example program the resulting guarded assertions would be:

$$\begin{aligned} (r0 < x0) &\rightarrow (m1 = y0 * r1) && \wedge \\ ((r0 < x0) \wedge (r1 < x0)) &\rightarrow (m2 = y0 * r2) && \wedge \\ ((r0 < x0) \wedge (r1 < x0) \wedge (r2 < x0)) &\rightarrow (m3 = y0 * r3) \end{aligned}$$

So, the final $CNF(C, k, P)$ would be that formula, but rewritten using vectors of boolean variables for representing the variables of the program. That is, the same boolean variables used in the formula $CNF(C, k)$.

Unwinding Assertions

As we have said before, because for some loops it may be imposible, or not desirable, to consider the minimum number of unwindings needed for any possible execution trace, CBMC allows the user to specify the number of unwindings U we want to consider for a particular loop.

However, in that case CBMC also allows the user to check if that specified number of unwindings is enough to capture all the possible execution traces through the loop. This is done using *unwinding assertions*. An unwinding assertion is an assertion that says that the loop guard condition b is false at some of the U loop unwindings, that is:

$$UA = \neg b_1 \vee \neg b_2 \vee \dots \vee \neg b_U$$

where b_i is the loop guard condition over the renamed variables for iteration i . So, if the assertion UA is true for any possible execution trace we are saying that the loop will always perform at most $U - 1$ iterations. Or in other words, if CBMC finds an execution trace that violates the unwinding assertion, that execution trace is a counterexample for the property of not needing more than $U - 1$ iterations. For example, consider the following program:

```
x = nondet_posinteger();
y = nondet_posinteger();
while (x != y) {
  if (x > y) x = x-y;
  else y = y-x;
}
```

Obviously, in this loop the number of loop iterations depends on the size of the integer variables x and y ⁶. But suppose that we assume

⁶ Exercise: Can you give an upper bound on the number of loop iterations for this algorithm ?

that $x > 0 \wedge x < 5$ and $y > 0 \wedge y < 5$. Then, if in this case we think, but we are not sure, that the maximum number of loop iterations will be three so we need to evaluate the loop guard condition at most four times to be sure that the loop never needs more than three iterations. So, we should say to CBMC that we want to unwind the loop 4 times, but that we want to assert that the loop guard condition fails always at some of these 4 loop guard condition evaluations (at least it should fail when evaluated for the fourth time). That is, we will have the following unwound program after renaming variables:

```
x0 = nondet_posinteger();
y0 = nondet_posinteger();
if (x0 != y0) {
  if (x0 > y0) x1 = x0-y0; else y1 = y0-x0;
  if (x1 != y1) {
    if (x1 > y1) x2 = x1-y1; else y2 = y1-x1;
    if (x2 != y2) {
      if (x2 > y2) x3 = x2-y2; else y3 = y2-x2;
      if (x3 != y3) {
        if (x3 > y3) x4 = x3-y3; else y4 = y3-x3;
      }
    }
  }
}
```

The corresponding unwinding assertion for this unwound loop is:

$$UA \equiv (x_0 = y_0) \vee (x_1 = y_1) \vee (x_2 = y_2) \vee (x_3 = y_3)$$

and it will always be satisfied by any execution trace that satisfies the precondition $x > 0 \wedge x < 5 \wedge y > 0 \wedge y < 5$, so concluding that the loop never needs to iterate more than three times for this case.

Because in some cases we will specify deliberately an insufficient number of loop unwindings for a general execution context, but that it is enough for a particular case we are interested to analyze ⁷, CBMC allows us to control the insertion of unwinding assertions. If we want CBMC to insert unwinding assertions for every loop in our program, we must use the option `--unwinding-assertions`.

⁷ Consider for example a case where for some integer variables of the loop we know that in a particular execution context their possible values range in a much smaller interval.

The Final SAT Formula

Once we have the formula $CNF(C, k)$, with satisfying assignments being execution traces of length at most k , and the formula $CNF(C, k, P)$ that is satisfied only for such execution traces that satisfy all the assertions in P , we can verify if C satisfies all the assertions in P by verifying if the boolean formula:

$$CNF(C, k) \rightarrow CNF(C, k, P)$$

is valid (any execution trace that satisfies $CNF(C, k)$ satisfies also $CNF(C, k, P)$). Or what it is the same, we can check whether the formula

$$F \equiv CNF(C, k) \wedge \neg CNF(C, k, P)$$

is satisfiable. That is, to check if there exists an execution trace for C that violates (does not satisfy) some of the assertions in P . The formula $\neg CNF(C, k, P)$ is satisfied by an execution trace if the execution trace violates at least one of the assertions in P , so it can be rewritten as the disjunction of the negations of the individual guarded assertions P_i in the set P :

$$\neg CNF(C, k, P) \equiv (\neg CNF(P_1) \vee \neg CNF(P_2) \vee \dots \vee \neg CNF(P_n))$$

Remember that as we are considering an *unwounded and renamed* version of C for considering execution traces of length at most k , where we use different variables every time a variable is changed in a line of code, the actual formula $\neg CNF(C, k, P)$ will contain the disjunction of the negations of **all** the guarded assertions resulting after performing unwinding and renaming of variables.

If F is satisfiable, that means that there exists at least one execution trace for C where some of the guarded assertions in P are false. That is, an execution trace that is a *counterexample*.

By default, CBMC uses the SAT solver minisat (version 2), but it can output the final SAT formula and then the formula be solved with any desired SAT solver.

In the most recent versions of CBMC there is also the option of using SMT encodings, with integer or other types of variables, instead of using a boolean encoding of the formulas for C and P . This way, we can use less variables (but with bigger domains) and/or higher level constraints that help to produce smaller final formulas for C and P , such that in some cases may help to solve better some complex problems.

CBMC Command Line Syntax

The general execution syntax of CBMC is:

```
cbmc options file1.c file2.c ...
```

Where `file1.c file2.c ...` is the list of source files where the program that we want to check can be found. It is assumed that the `main()` function will be located in one of such source files, because this is the entry point that CBMC assumes for the program. But we can also specify that the entry point of the program will be a different function, using the option `--function funcname`:

```
cbmc --function funcname file1.c file2.c ...
```

Specifying the right entry point of the program, or part of the code, that we want to check is important, because the static analysis that CBMC performs to build the formula $CNF(C, k)$ starts at the first line of the entry function, and only covers lines of code from `file1.c` `file2.c` ... that are reachable, step by step, starting from the entry point up to execution depth k . By default, CBMC automatically tries to find a high enough value for k such that any **complete** execution path (an execution path that ends in an exit state of the program) starting from the entry point will have depth at most k . For complex programs where this may not be possible, we have two different ways of upper bounding the execution path depth:

1. The most useful is probably the option `--unwind nr`, that is used for upper bounding the amount of unwinding for any loop (or recursive function) of the program to nr , but the maximum execution depth k is still not directly bounded by the user, but will depend on the total number of loop unwindings and sequential pieces of code traversed starting from the entry point.
2. We can also set directly the upper bound on the execution path depth k , with the option `--depth k`. This option, when used without using also the previous one, only limits the total execution depth, so for any loop traversed CBMC still tries to find the right amount of unwinding. So, we can also use both options at the same time. If we use both options, any loop will be unwound up to nr times but in any case the maximum execution path depth will be at most k .

The option `--function funcname` is also helpful when we are interested in checking only an individual function from a set of functions, as we will discuss at the end of this manual.

Another useful option, that will be used for example at the final section of this manual, is the option `-property ID` to indicate that we only want to check the assertion with identifier `ID` and ignore the rest of them:

```
cbmc --property ID file1.c file2.c ...
```

See the last section for an example of use of this option.

Checking General Assertions

Let's study how to use general assertions for checking that our program behaves as we expect. Suppose we want to check whether the following program correctly computes the distance between two integer numbers stored in `a` and `b`:

```

1 int main( void ) {
2   int a, b;
3   int abs;
4
5   if ( a >= b )
6     abs = a-b;
7   else
8     abs = b-a;
9
10  __CPROVER_assert((abs >= 0) &&
11                  ((abs == (a-b)) || (abs == (b-a))),
12                  " abs stores the absolute value of a-b" );
13
14  printf( " %i\n", abs );
15 }
```

examples/userassertions-ex1.c

We have included in such program an assertion at line 10 that is expressing that the value that is stored in variable `abs` represents the distance between `a` and `b`. However, if we check this program with CBMC we obtain a counterexample that shows that the program not always works correctly. Here there is the relevant part of such counterexample⁸:

```

State 14 file userassertions-ex1.c line 2 function main
-----
a=101449728

State 15 file userassertions-ex1.c line 2 function main
-----
b=-2114387968

...

State 18 file userassertions-ex1.c line 6 function main
-----
abs=-2079129600

Violated property:
file userassertions-ex1.c line 10 function main
abs stores the absolute value of a-b
abs >= 0 && (abs == a - b || abs == b - a)
```

⁸ Example found with CBMC version 4.5 and default SAT solver minisat2.

In this counterexample what we have is two integer numbers, a positive and b negative, such that the distance between them is so big that when we calculate $a - b$ a signed arithmetic overflow is produced⁹, so we end up with a negative value, instead of the expected positive value given that a is bigger than b.

So, if there are no restrictions on the values of a and b this program is not always correct. CBMC allows us to filter out some of the possible counterexamples by using *assumptions*. Assumptions are conditions that can be used to restrict the values that certain variables can take when the assertions that follow the assumption are checked.

This makes sense, for example, in cases where the value of such variables come from external inputs to the program, but we assume that certain conditions will be true and restrict the possible values we are interested to study. For example, if in the previous example we know for sure that a and b will be always in the range $[-200000000, 200000000]$, we can modify the previous program with an assumption to restrict the domain of possible values in that range:

⁹ Here we are executing CBMC such that it represents integer variables as bit vectors of size 32. This can be changed with an option of CBMC for emulating other possible architecture dependent implementations of integers.

```

1 int main( void ) {
2     int a, b;
3     int abs;
4
5     __CPROVER_assume( (a >= -200000000) &&
6                       (a <=  200000000) &&
7                       (b >= -200000000) &&
8                       (b <=  200000000) );
9
10    if ( a >= b )
11        abs = a-b;
12    else
13        abs = b-a;
14
15    __CPROVER_assert((abs >= 0) &&
16                    ((abs == (a-b)) || (abs == (b-a))),
17        " abs stores the absolute value of a-b" );
18
19    printf( " %i\n", abs );
20 }
```

examples/userassertions-ex2.c

In this case, CBMC does not find any counterexample, as assuming that the domain of values for a and b is within the range $[-200000000, 200000000]$ makes not possible to have a signed arithmetic overflow with the integer variables in the expression $a - b$.

In the next example, we analyze a program for performing integer division, that should be able to compute the quotient and the remainder, but we are going to see that is not a good algorithm for all the

possible inputs.

```

1 // undet_int may return any possible integer
2 int nondet_int(void);
3
4 int main( void ) {
5     int D, d, q, r;
6
7     // Simulate D and d come from some external input
8     D = nondet_int();
9     d = nondet_int();
10    r = D;
11    q = 0;
12    __CPROVER_assert( (D == (q*d + r)) && (r >= 0),
13                      "I true before entering" );
14    while ( r >= d ) {
15        q = q + 1;
16        r = r - d;
17        __CPROVER_assert( (D == (q*d + r)) && (r >= 0),
18                          "I true at the end of iteration" );
19    }
20    // Post-condition
21    __CPROVER_assert( (D == (q*d + r)) && (r >= 0) && (r < d),
22                      " q and r are the quotient and the remainder of D div d" );
23 }

```

examples/userassertions-ex3.c

If we ask CBMC to check the assertions of this program with the default command line ¹⁰:

```
cbmc --trace --stop-on-fail userassertions-ex3.c
```

We get a very long output from CBMC given that it unwinds the loop of the program for a big number of iterations and it seems not to stop:

```

...
Unwinding loop main.0 iteration 69 file userassertions-ex3.c line
14
Unwinding loop main.0 iteration 70 file userassertions-ex3.c line
14
Unwinding loop main.0 iteration 71 file userassertions-ex3.c line
14
Unwinding loop main.0 iteration 72 file userassertions-ex3.c line
14
Unwinding loop main.0 iteration 73 file userassertions-ex3.c line
14
Unwinding loop main.0 iteration 74 file userassertions-ex3.c line
14
...

```

¹⁰ Starting from cbmc version 5.7, in order to show on screen a counterexample (when one is found), we have to use the option `--trace`. Also, if we want to search a counterexample for any assertion, and then stop searching counterexamples for other assertions, we use the option `--stop-on-fail`.

So we decide to interrupt the execution of CBMC . Because the unwinding performed by CBMC seems to have not a clear upper bound from the input parameters, or at least CBMC is having a long time to get it, we can run CBMC in a different way. This time, we ask CBMC to unwind loops up to certain number of times we provide with the option `--unwind U`, where `U` is the unwinding upper bound we want to apply for any loops in the program. That means that we are going to analyze the correctness of our assertions only for such cases where the number of iterations of the loop is upper bounded by `U`. With this new execution:

```
cbmc --unwind 51 --stop-on-fail --trace userassertions-ex3.c
```

This time, we get a counterexample that violates the assertion inside of the loop (the invariant is not true at the end of the iteration):

$$D = 1787112374, d = -1582541451$$

The problem is that after the first iteration is performed, we end up with `q=1` and `r=1787112374 - (-1582541451)` that causes an integer overflow and `r` becomes a negative value (`r = -925313471`). This time, it seems that the problem is that the way of updating the remainder is only valid if `d` is not negative. So, we insert an assumption, to consider only executions with `d` non-negative, just after the line where the value of `D` is obtained with the `undet_int()` function call:

```
__CPROVER_assume( d >= 0 );
```

But if we run again CBMC with this assumption added, we get a new counterexample, but now for the first assertion (the invariant is not true before entering the loop):

$$D = -1834472784, d = 218642508$$

This counterexample shows that now the problem is that the remainder is not greater or equal to zero before entering the loop, because at that point `r=D` and `D` is negative, so the invariant does not hold true at that point.

So, it seems that this time, the problem is that this algorithm does not work with `D` negative. Then, we can add this assumption in the same position as before:

```
__CPROVER_assume( d >= 0 && D >= 0 );
```

This time, if we execute CBMC again, it may seem that the execution of `cbmc` never ends. What happens is that `cbmc` seems to not find any errors for the huge domain of values that D can have. To bound the possible domain to check, we are going to check the program again, but with this modified assumption:

```
__CPROVER_assume( d >= 0 && D >= 0 && D <= 50 );
```

With this restricted domain for D , it seems that the program is working OK. However, remember that we are analyzing the program for a bounded number of iterations of the loop. So, how can we be sure that there are no **longer execution traces that are buggy**? It seems obvious that for $D \leq 50$, a number of maximum iterations equal to 50 should be enough, so if we unwind the loop 51 times, we expect that in any execution, some of these 51 loop conditions should be false (for an execution that needs 50 iterations, the copy 51 of the loop condition should be false). So, we check again our program, but asking to insert an unwinding assertion (as we have explained in a previous section) in every loop of it:

```
cbmc --unwind 51 --stop-on-fail --unwinding-assertions --trace
userassertions-ex3.c
```

That is, to insert an assertion that expresses that there is no execution trace where the loop condition would be true more than 50 times (and to check this we have to unwind the loop 51 times). This time, we get a counterexample for the unwinding assertion:

$$D = 0, d = 0$$

The final problem is that with $d=0$ this algorithm does not work, as in that case the remainder never decreases, so any finite upper bound for the number of iterations of the loop is not enough ($d = 0$ gives an infinite loop, so obviously any finite bound on the number of unwindings will not satisfy the unwinding assertion). This indicates that we should modify the assumption in our program in this way:

```
__CPROVER_assume( d > 0 & D >= 0 && D <= 50 );
```

with this final assumption, CBMC does not find any counterexample, so it seems that the program is correct. But because we are checking the program for values of D between 0 and 50, we cannot ensure that it is valid for any positive value of D . If we increase the domain of values for D (increasing the value of `--unwind U` accordingly) we will always get that the program is correct, but the bigger the domain of D the bigger the BMC problem that CBMC needs to solve, and so the bigger the execution time of the verification process. But observe that the analysis we have performed has helped us to discover that the program was not valid for the division of negative integers. So even if we do not have a total correctness proof for positive integers, we know at least some cases where the program is not valid.

Let's study one more example program. Consider the following program for finding the biggest number from an array of integers:

```

1 int main( void ) {
2   int i, max, a[10];
3
4   max = a[0];
5   for ( i = 1 ; i < 10 ; i++ ) {
6     if ( a[i] > max )
7       max = a[i];
8   }
9   __CPROVER_assert( ??? ,
10    " max stores the biggest value from a[]" );
11 }
```

examples/userassertions-ex4-1.c

The expression that is missing in the assertion, should express that the value stored in `max` is the biggest from the array `a[]`. That is, that `max` is bigger than or equal to any number in `a[]` but that there is at least one number in `a[]` equal to `max`. From these two conditions, the one we can express as a set of individual assertions is the first one. We are going to assert that for any i between 0 and 9, $\text{max} \geq a[i]$ must be true with the following modified version of the program:

```

1 int nondet_int(void);
2
3 int main( void ) {
4   int i, max, a[10];
5
6
7   max = a[0];
8   for ( i = 1 ; i < 10 ; i++ ) {
9     if ( a[i] > max )
10       max = a[i];
11   }
12
13   i = nondet_int();
```



```

14 __CPROVER_assume( i >= 0 && i < 10 );
15 __CPROVER_assert( max >= a[i] ,
16     " max >= a[i]" );
17 }

```

examples/userassertions-ex4-2.c

In this program we use a function that starts with a special prefix: `nondet_` and that returns an integer. Any declared function with such prefix in its name is treated by CBMC as a function that can return any possible value for its return type (integer in this case), so CBMC does not need to know the code of such function, because it is used to simulate the input of a non determined integer number in a specific point of the program. The combination of the call to this function to get a value for i and then check, for values between 0 and 9, whether they satisfy the assertions, produces that CBMC will try to find a counterexample whith a particular value of i within that range where the assertion is false. If this is not possible, that means the assertion holds for all the elements of the array.

Next, we have to consider the second condition we have to check: that there is at least one integer $a[i]$ that is equal to max . For checking this second condition, we can introduce at the end of the program a counter for storing the number of elements $a[i]$ that satisfy that condition, and then **assert** that this counter should be bigger than or equal to 1. So, this can be done as follows:

```

1 int nondet_int(void);
2
3 int main( void ) {
4     int i, max, a[10];
5     int b;
6
7     max = a[0];
8     for ( i = 1 ; i < 10 ; i++ ) {
9         if ( a[i] > max )
10             max = a[i];
11     }
12
13     i = nondet_int();
14     __CPROVER_assume( i >= 0 && i < 10 );
15     __CPROVER_assert( max >= a[i] ,
16         " max >= a[i]" );
17     b = 0;
18     for ( i = 0 ; i < 10 ; i++ ) b += (a[i] == max);
19     __CPROVER_assert( b >= 1 ,
20         " max == a[i] for some i" );
21 }

```

examples/userassertions-ex4-3.c

Checking Array Bounds Violations

One of the main causes for incorrect accesses to memory locations is originated by *out of bounds* accesses to arrays. So, for example, in the following program we have included an assertion in order to check whether an access to an array element is performed out of the array bounds:

```

1 #include <stdio.h>
2
3 int main( void ) {
4     int rgNumbers[10];
5     long int a, b;
6     int res;
7
8     a = b % 10;
9     __CPROVER_assert( (a >= 0) && (a < 10),
10         "access to rgNumbers within size bounds" );
11     res = rgNumbers[a];
12
13     printf( " %i\n", res );
14 }

```

examples/abound-ex1.c

If we check this program with CBMC, it will find a counterexample (execution trace) that violates the assertion. For example, an execution trace with 18 states is found in our case ¹¹, where the relevant part of the trace is the final part:

```

...

State 16 file abound-ex1.c line 5 function main
-----
    b=-37871947874308

State 17 file abound-ex1.c line 6 function main
-----
    res=0

State 18 file abound-ex1.c line 8 function main
-----
    a=-8

Violated property:
file abound-ex1.c line 9 function main
    access to rgNumbers within size bounds
    a >= (signed long int)0 && a < (signed long int)10

```

¹¹ Using CBMC version 4.5 and default SAT solver minisat2. The particular counterexample found, when there are many of them, depends on how the search is performed.

That is, this counterexample shows that when $b = -37871947874308$ then $b\%10 = -8$ so $a = -8$ ¹². Then, this counterexample shows that **not for any** possible input values this program will check elements within the bounds of the array `rgNumbers`.

Observe that actually the value shown at state 17 is irrelevant for the violation of the array bound, but CBMC always shows the full execution trace corresponding to the counterexample found, and it does not try to find the *minimum* part of it that explains the violated property.

CBMC allows the automatic insertion of assertions of this kind in every point of the program with an access to an array element using the following option:

```
cbmc --bounds-check file1.c file2.c ...
```

So, with this option CBMC will check, in addition to the user defined assertions, assertions in all the points of the program with an array element access as the one we have presented for the previous example. However, CBMC inserts actually two individual assertions per every array access `a[i]` for an array of size N :

```
__CPROVER_assert( i >= 0,
                  "array 'a' lower bound in a[i]" );
__CPROVER_assert( i < N,
                  "array 'a' upper bound in a[i]" );
```

This way, when a counterexample for an array bounds violation is found, it will explicitly indicate whether it was a lower bound violation or an upper bound violation.

Checking Dynamic Memory Managment

Even if the main use of CBMC is for embedded systems software, where the memory usage of the program will not typically consider dynamic memory, CBMC supports a full coverage of ANSI-C, so it allows to check also properties related to the correct access to dynamically managed objects within a program.

The assertions related to dynamic memory managment are generated by different flags. These assertions use information macros internally generated by CBMC based on information generated during the initial transformation of the program.

The main property checks are the ones related to the option `--pointer-check`. With this option, CBMC generates assertions in every point of the program with a pointer dereference (an expression of the

¹² For b negative, ANSI-C does not require the implementation of $q = b/10$ and $r = b\%10$ to follow the definition of integer division, but any result to the equation $q * 10 + r = b$ is allowed. CBMC uses the standard implementation: $q = \lfloor b/10 \rfloor$ and $r = b - q$ that for b negative is not the same result as the one given by the theorem of integer division.

kind `*p` or `p[i]` with `p` being a pointer expression) that asserts that `p` points to a valid dynamically created object.

Consider the following example:

```

1 int main( void )
2 {
3     int *p, a;
4
5     p = malloc(sizeof(int)*3);
6     free( p );
7     p[3] = 0;
8 }

```

examples/dynmemory-ex1.c

If we run CBMC with the option `--pointer-check`, it will include assertions for checking that the expression `p` is a valid pointer expression at that point of the program. In this case, CBMC will find two counterexamples, because `p[3]` is not a valid pointer expression when it is used in the last line. We can get a verbose explanation of the different assertions that CBMC is including with this option if we also include the option `--show-properties`. Then, CBMC shows the following information for our example program with respect to the properties to check for line 7 of the program (but it does not perform the bug finding process):

```

Property main.pointer_dereference.1:
  file examples/dynmemory-ex1.c line 7 function main
  dereference failure: pointer NULL in p[(signed long int)3]
  !(__CPROVER_POINTER_OBJECT((signed int *)NULL)) == __CPROVER_POINTER_OBJECT(p))

Property main.pointer_dereference.2:
  file examples/dynmemory-ex1.c line 7 function main
  dereference failure: pointer invalid in p[(signed long int)3]
  !IS_INVALID_POINTER(p)

Property main.pointer_dereference.3:
  file examples/dynmemory-ex1.c line 7 function main
  dereference failure: deallocated dynamic object in p[(signed long int)3]
  !(__CPROVER_POINTER_OBJECT(p) == __CPROVER_POINTER_OBJECT(__CPROVER_deallocated))

Property main.pointer_dereference.4:
  file examples/dynmemory-ex1.c line 7 function main
  dereference failure: dead object in p[(signed long int)3]
  !(__CPROVER_POINTER_OBJECT(p) == __CPROVER_POINTER_OBJECT(__CPROVER_dead_object))

Property main.pointer_dereference.5:
  file examples/dynmemory-ex1.c line 7 function main
  dereference failure: pointer outside object bounds in p[(signed long int)3]
  (unsigned __CPROVER_bitvector[65])__CPROVER_OBJECT_SIZE(p) >= (unsigned __CPROVER_bitvector[65])(
    __CPROVER_POINTER_OFFSET(p) + 12ul) + 4

```

```
Property main.pointer_dereference.6:
  file examples/dynmemory-ex1.c line 7 function main
  dereference failure: invalid integer address in p[(signed long int)3]
  !(__CPROVER_POINTER_OBJECT(((signed int *)NULL)) == __CPROVER_POINTER_OBJECT(p)) || p + 3l == ((signed
    int *)NULL)
```

These six assertions, generated for line 7 of our example program, say that:

1. `p` is not accessing to a `NULL` pointer.
2. `p` is not an invalid pointer (is a valid pointer expression).
3. `p[3]` is not a pointer to a deallocated dynamic object.
4. `p[3]` is not an access to a dead object.
5. `p[3]` is no accessing outside of the limits of an object.
6. `p[3]` is no accessing an invalid (`NULL`) memory address.

CBMC is able to check these assertions because during the program transformation we have presented in the first section it performs a **recursive analysis of pointer expressions** where all the dereferences are finally removed and substituted by the elements that they point to.

This program contains two bugs. If we want to find counterexamples for both of them (with a single search), we have to execute it in the following way:

```
cbmc --trace --pointer-check examples/dynmemory-ex1.c
```

It finds two counterexamples, that actually represent the same unique execution trace of this program for properties 3 (reference to a deallocated object) and 5 (pointer outside dynamic object bounds, given that `p[3]` is an access outside of the limits of the object to which `p` points to).

The recursive analysis of pointer expressions allows to catch bugs in more complex programs. For example, consider the next program:

```
1 int main( void )
2 {
3   int *p, *q, *r;
4   int a;
5
6   r = 0;
7   a = 2;
```

```

8  p = malloc(sizeof(int)*3);
9  q = p+1;
10 q[a] = 3;
11 r[0] = 9;
12 }

```

examples/dynmemory-ex2.c

In this case, CBMC finds also a bug, because the recursive analysis performed finds that $q[a]$ is equivalent to $*(p+1+a)$. So, given the value of a , it is accessing out of the limits of the dynamic object to which p points to.

The recursive analysis of pointer expressions also incorporates tracking whether an object is still *alive* (not deallocated) in each point where it is accessed. So, for example, in the following example:

```

1 int main( void )
2 {
3     int *p, *q;
4     int a;
5
6     a = 2;
7     p = malloc(sizeof(int)*3);
8     q = p+1;
9     free( p );
10    q[a] = 3;
11 }

```

examples/dynmemory-ex3.c

CBMC finds a counterexample that shows that an access to a deallocated object is performed, and that the access is also outside of the bounds of the dynamic object to which $*(p+1+a)$ points to. This is the final part of the counterexample found:

```

Violated property:
  file examples/dynmemory-ex3.c line 10 function main
  dereference failure: deallocated dynamic object in q[(signed long
    int)a]
  !(POINTER_OBJECT(q) == POINTER_OBJECT(__CPROVER_deallocated))

Violated property:
  file examples/dynmemory-ex3.c line 10 function main
  dereference failure: pointer outside dynamic object bounds in q[(
    signed long int)a]
  4l * (signed long int)a + POINTER_OFFSET(q) >= 0l &&
    __CPROVER_malloc_size >= 4ul * (unsigned long int)(signed
    long int)a + 4ul + (unsigned long int)POINTER_OFFSET(q) || !(
    POINTER_OBJECT(q) == POINTER_OBJECT(__CPROVER_malloc_object))

** 2 of 11 failed (3 iterations)
VERIFICATION FAILED

```

Checking Functions Output and Test Case Generation

A key property of programs that it is desirable to be able to check is whether every function call performed returns what it is expected for the function, given the particular input introduced (checking that the function is respecting the specification of its normal behaviour).

With CBMC, this can be checked inserting an assertion after the function call, that expresses the relation that should be satisfied by the function output and the function input.

Consider the following version of the first example of this section, where now the computation of the distance between *a* and *b* is performed by the function *getabs*:

```

1 int getabs( int a, int b ) {
2     int abs;
3
4     if ( a >= b )
5         abs = a-b;
6     else
7         abs = b-a;
8
9     return abs;
10 }
11
12 int main( void ) {
13     int a, b;
14     int abs;
15
16     abs = getabs( a, b );
17     __CPROVER_assert((abs >= 0) &&
18                     ((abs == (a-b)) || (abs == (b-a))),
19     " abs stores the absolute value of a-b" );
20
21     printf( " %i\n", abs );
22 }

```

examples/userassertions-ex5.c

But observe that in this way, we have to *remember* to include such assertion after each call to that function. Another option is to include the assertion inside of the function, just before returning the computed value:

```

1 int getabs( int a, int b ) {
2     int abs;
3

```

```

4  if ( a >= b )
5      abs = a-b;
6  else
7      abs = b-a;
8
9  __CPROVER_assert((abs >= 0) &&
10                  ((abs == (a-b)) || (abs == (b-a))),
11                  " abs stores the absolute value of a-b" );
12  return abs;
13 }
14
15 int main( void ) {
16     int a, b;
17     int abs;
18
19     abs = getabs( a, b );
20
21     printf( " %i\n", abs );
22 }

```

examples/userassertions-ex6.c

So in case of a more complex program, where the function is called many times, we know for sure that every function call will be checked.

So far, we have checked programs where the entry point is always the `main()` function. But imagine that our goal is to check the correctness of a set of functions inside of a module (that for example can be part of a library we are developing). CBMC also allows to check the correctness of individual functions (checking that they behave as expected for any possible inputs) by specifying the desired function to check in the command line:

```
cbmc --function getabs userassertions-ex6.c
```

In this way, we are asking CBMC to consider all the possible ways of executing the function `getabs(a,b)`, so if the assertions hold, we can be sure that the function will satisfy these assertions when called inside of any program (assuming real functions, i.e. they do not use global variables for doing their work). Observe that this way of checking functions is far more powerful than what is traditionally done in unit testing for software developing, where the unit tests developed for individual (or sets of) functions usually only cover some important cases of the functions, but developing a set of unit tests that cover all the possible *behaviours* of a function may be a complex task for non trivial functions.

Regarding to coverage analysis, CBMC can also be very helpful to

automatically generate a set of individual tests that cover all the possible lines of code of a complex software. Satisfying a 100% of code coverage by a set of tests can be sometimes requested for critical systems. For example, in the work ¹³ the authors develop a method, based on the use of CBMC, to create a set of tests for the full coverage of a subset of the modules of the [European Train Control System \(ETCS\)](#). Their method is basically based on the following idea. First, we insert an assertion in each part of the code we want to cover with a test (an input for the program/function that generates an execution trace that reaches that part of the code). The assertion must be a property that will be violated for any execution path that reaches that part of the code. So, this is simply the expression FALSE (o):

```

sentence i;
__CPROVER_assert( 0, "The execution reached sentence i");

```

So, if CBMC finds an execution trace that reaches that sentence of the code, it will provide that execution trace as a counterexample (even if it is not really violating any *real* property). The contents of that execution trace provide the contents for a test that when executed covers, at least, that line of code. Remember that CBMC stops as soon as it finds a counterexample for one assertion. But if we want to search counterexamples for a whole set of assertions, we can ask CBMC to search counterexamples for only one of the assertions of the program. First, we have to get the list of identifiers for all the assertions of the program with this option (assertions are called properties in the output of CBMC):

```

cbmc --show-properties file1.c file2.c ...

```

Then, for each such assertion we call CBMC in this way to search for counterexamples only for that assertion (the others are ignored and not used in the formula *P*):

```

cbmc --property propertyID file1.c file2.c ...

```

Observe that a counterexample may cover more than one desired sentence of the program, so it may not be necessary to find one test for each desired sentence of the program. Remember also that for complex programs CBMC will not be able to determine the needed amount of unwinding for all the loops, so performing code coverage will be typically performed by incremental increase of the allowed amount of unwinding, until a complete set of tests is found.

¹³ Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *J. Automatic Reasoning*, 45(4):397–414, 2010

Consider the following function, where we have included coverage assertions (with condition to satisfy equal to FALSE) in addition to a final assertion for checking that its final returned value is correct:

```

1 int minxyz( int x, int y, int z ) {
2     int min;
3
4     if (x <= y) {
5         if ( z <= x )
6             { min = z;
7               __CPROVER_assert( 0, " z <= x <= y" ); }
8         else
9             { min = x;
10              __CPROVER_assert( 0, " x <= (z,y)" ); }
11     }
12     else {
13         if ( z <= y )
14             { min = z;
15               __CPROVER_assert( 0, " z <= y <= x" ); }
16         else
17             { min = y;
18               __CPROVER_assert( 0, " y <= (x,z)" ); }
19     }
20
21     __CPROVER_assert(
22         ( min <= x && min <= y && min <= z &&
23           ( min == x | min == y | min == z )),
24         " function minxyz returns the min of x,y,z " );
25     return min;
26 }

```

examples/min-function.c

For performing individual checking of assertions in this function to get test cases, we first obtain its list of assertion identifiers:

```
cbmc --show-properties --function minxyz min-function.c
```

getting this list of assertions together with their identifiers:

```

Property minxyz.1:
  file min-function.c line 7 function minxyz
  z <= x <= y
  (_Bool)0

Property minxyz.2:
  file min-function.c line 10 function minxyz
  x <= (z,y)
  (_Bool)0

Property minxyz.3:

```

```

file min-function.c line 15 function minxyz
  z <= y <= x
  (_Bool)0

Property minxyz.4:
  file min-function.c line 18 function minxyz
  y <= (x,z)
  (_Bool)0

Property minxyz.5:
  file min-function.c line 21 function minxyz
  function minxyz returns the min of x,y,z
  min <= x && min <= y && min <= z && (_Bool)((signed int)(min == x
    ) | (signed int)(min == y) | (signed int)(min == z))

```

Observe that assertion identifiers are always prefixed by the name of the function where they are located. Then, for each coverage assertion, we try to find a test case that reaches it (an execution path that reaches the line where the assertion is located) with this execution:

```
cbmc --property propertyID --function minxyz min-function.c
```

So, that for our four coverage assertions we get the following test cases:

- Assertion 1 ($z \leq x \leq y$). Test case generated: $z=0$, $x=1$, $y=3$.
- Assertion 2 ($x \leq (z, y)$). Test case generated: $x=-2147483644$, $y=5$, $z=0$.
- Assertion 3 ($z \leq y \leq x$). Test case generated: $z=0$, $y=1$, $x=2$.
- Assertion 4 ($y \leq (x, z)$). Test case generated: $y=-2147483643$, $x=-1073741823$, $z=-1073741820$.

So, we have obtained test cases for all the parts of the function.

If we are finally interested in checking that the function always return correctly the minimum number from its three input numbers, we can check the final assertion:

```
cbmc --property minxyz.5 --function minxyz min-function.c
```

and this time we get the message telling us that the verification is successful, as the last assertion is satisfied for any possible input for this function.

In order to automate the full process of obtaining a complete set of test cases that covers all the decision points of a function, the complete process presented in the work ¹⁴ to create the set of tests for the full coverage of a subset of the modules of the [European Train Control System \(ETCS\)](#) can be summarized with the pseudo-code shown here in Function `GetTestSet(f,n,U)`.

Function `GetTestSet(f,n,U)`

```

BestMValError := 1.1
TestSet := {}
k := 1
// Initialize the test set to the empty set and the number of loop unwindings to 1
for i := 1 to n do
  Insert assert(0) macro after decision point i in f()
while (  $\frac{|covered(f,TestSet)|}{n} < 1$  )  $\wedge$  (k ≤ U) do
  for i := 1 to n do
    if decisioni not covered by TestSet then
      test := CBMC(−function f,−claim f.i,−unwind
        k,−no-unwinding-assertions)
      if test ≠ null then
        // If a counterexample (test) has been found for
        // decision point i, update the test set
        TestSet := TestSet ∪ {test}
  k := k + 1
return TestSet

```

In this function, we first insert an `assert(0)` macro after each decision point i of the function $f()$ that we want to cover with a set of test cases. Then, in the main loop of the function we try to find a test case for each decision point i not yet covered by some of the test cases already in the current set `TestSet` ¹⁵. Each one of these test cases is searched using CBMC with the settings shown in the code. Given that we run CBMC with an upper bound k on the number of loop unwindings, it may happen that it does not find a test case for decision point i because k unwindings for all the loops is not enough to reach such decision point. Then, after we have searched test cases for trying to cover all the decision points, we perform a new iteration if we have failed to cover all the decision points, but increasing the value of k , but only if we have not overpassed the upper limit U for the value of k . So, if all the decision points in function f can be reached by some execution with a finite number of steps, using a high enough value for U we will find a complete test set. In order to try to minimize the size of the test set generated, it is worth noticing that **the order** in which the decision points are traversed in the inner loop is important and it can also be a good strategy to start with a

¹⁴ Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *J. Automatic Reasoning*, 45(4):397–414, 2010

¹⁵ Observe that a test set can cover more than one decision point, so it is not always necessary to find n different test cases for covering n decision points

value for k much bigger than 1, such that it is high enough to reach the deepest decision points of f . Can you guess why it is a good strategy to try to reach first the deepest decision points ?

Appendix: CBMC (5.56) Full Command Line Options

```
* *          CBMC 5.56.0 (cbmc-5.56.0) 64-bit          * *
* *          Copyright (C) 2001-2018                  * *
* *          Daniel Kroening, Edmund Clarke            * *
* * Carnegie Mellon University, Computer Science Department * *
* *          kroening@kroening.com                     * *
* *          Protected in part by U.S. patent 7,225,417 * *
```

Usage:

Purpose:

```
cbmc [-?] [-h] [--help]    show help
cbmc file.c ...            source file names
```

Analysis options:

```
--show-properties          show the properties, but don't run analysis
--symex-coverage-report f  generate a Cobertura XML coverage report in f
--property id              only check one specific property
--trace                    give a counterexample trace for failed properties
--stop-on-fail             stop analysis once a failed property is detected
                           (implies --trace)
--localize-faults          localize faults (experimental)
```

C/C++ frontend options:

```
--preprocess              stop after preprocessing
-I path                   set include path (C/C++)
-D macro                  define preprocessor macro (C/C++)
--c89/99/11               set C language standard (default: c11)
--cpp98/03/11             set C++ language standard (default: cpp98)
--unsigned-char            make "char" unsigned by default
--round-to-nearest         rounding towards nearest even (default)
--round-to-plus-inf        rounding towards plus infinity
--round-to-minus-inf       rounding towards minus infinity
--round-to-zero            rounding towards zero
--no-library               disable built-in abstract C library
--max-nondet-tree-depth N  limit size of nondet (e.g. input) object tree;
                           at level N pointers are set to null
--min-null-tree-depth N    minimum level at which a pointer can first be
                           NULL in a recursively nondet initialized struct
```

--function name set main function name

Platform options:

--arch set architecture (default: x86_64)
 --os set operating system (default: linux)
 --16, --32, --64 set width of int
 --LP64, --ILP64, --LLP64,
 --ILP32, --LP32 set width of int, long and pointers
 --little-endian allow little-endian word-byte conversions
 --big-endian allow big-endian word-byte conversions
 --gcc use GCC as preprocessor

Program representations:

--show-parse-tree show parse tree
 --show-symbol-table show loaded symbol table
 --show-goto-functions show loaded goto program
 --list-goto-functions list loaded goto functions

Program instrumentation options:

--bounds-check enable array bounds checks
 --pointer-check enable pointer checks
 --memory-leak-check enable memory leak checks
 --div-by-zero-check enable division by zero checks
 --signed-overflow-check enable signed arithmetic over- and underflow checks
 --unsigned-overflow-check enable arithmetic over- and underflow checks
 --pointer-overflow-check enable pointer arithmetic over- and underflow checks
 --conversion-check check whether values can be represented after type cast
 --undefined-shift-check check shift greater than bit-width
 --float-overflow-check check floating-point for +/-Inf
 --nan-check check floating-point for NaN
 --enum-range-check checks that all enum type expressions have values in the enum range
 --pointer-primitive-check checks that all pointers in pointer primitives are valid or null
 --retain-trivial-checks include checks that are trivially true
 --error-label label check that label is unreachable
 --no-built-in-assertions ignore assertions in built-in library
 --no-assertions ignore user assertions
 --no-assumptions ignore user assumptions
 --assert-to-assume convert user assertions to assumptions
 --cover CC create test-suite with coverage criterion CC,
 where CC is one of assertion[s], assume[s],
 branch[es], condition[s], cover, decision[s],
 location[s], or mcdc
 --cover-failed-assertions do not stop coverage checking at failed assertions
 (this is the default for --cover assertions)

<code>--show-test-suite</code>	print test suite for coverage criterion (requires <code>--cover</code>)
<code>--mm MM</code>	memory consistency model for concurrent programs (default: <code>sc</code>)
<code>--malloc-may-fail</code>	allow malloc calls to return a null pointer
<code>--malloc-fail-assert</code>	set malloc failure mode to assert-then-assume
<code>--malloc-fail-null</code>	set malloc failure mode to return null
<code>--fp-reachability-slice f</code>	remove instructions that cannot appear on a trace that visits all given functions. The list of functions has to be given as a comma separated list <code>f</code> .
<code>--reachability-slice</code>	remove instructions that cannot appear on a trace from entry point to a property
<code>--reachability-slice-fb</code>	remove instructions that cannot appear on a trace from entry point through a property
<code>--full-slice</code>	run full slicer (experimental)
<code>--drop-unused-functions</code>	drop functions trivially unreachable from main function
<code>--havoc-undefined-functions</code>	for any function that has no body, assign non-deterministic values to any parameters passed as non-const pointers and the return value

Semantic transformations:

<code>--nondet-static</code>	add nondeterministic initialization of variables with static lifetime
------------------------------	---

BMC options:

<code>--paths [strategy]</code>	explore paths one at a time
<code>--show-symex-strategies</code>	list strategies for use with <code>--paths</code>
<code>--show-goto-symex-steps</code>	show which steps symex travels, includes diagnostic information
<code>--show-points-to-sets</code>	show points-to sets for pointer dereference. Requires <code>--json-ui</code> .
<code>--program-only</code>	only show program expression
<code>--show-byte-ops</code>	show all byte extracts and updates
<code>--depth nr</code>	limit search depth
<code>--max-field-sensitivity-array-size M</code>	maximum size <code>M</code> of arrays for which field sensitivity will be applied to array, the default is 64
<code>--no-array-field-sensitivity</code>	deactivate field sensitivity for arrays, this is equivalent to setting the maximum field sensitivity size for arrays to 0
<code>--show-loops</code>	show the loops in the program
<code>--unwind nr</code>	unwind <code>nr</code> times
<code>--unwindset [T:]L:B,...</code>	unwind loop <code>L</code> with a bound of <code>B</code> (optionally restricted to thread <code>T</code>)

```

--incremental-loop L      (use --show-loops to get the loop IDs)
                          check properties after each unwinding
                          of loop L
--unwind-min nr          (use --show-loops to get the loop IDs)
                          start incremental-loop after nr unwindings
                          but before solving that iteration. If for
                          example it is 1, then the loop will be
                          unwound once, and immediately checked.
                          Note: this means for min-unwind 1 or
                          0 all properties are checked.
--unwind-max nr          stop incremental-loop after nr unwindings
--ignore-properties-before-unwind-min
                          do not check properties before unwind-min
                          when using incremental-loop
--show-vcc               show the verification conditions
--slice-formula           remove assignments unrelated to property
--unwinding-assertions   generate unwinding assertions (cannot be
                          used with --cover or --partial-loops)
--partial-loops          permit paths with partial loops
--no-self-loops-to-assumptions
                          do not simplify while(1){} to assume(0)
--no-pretty-names        do not simplify identifiers
--symex-complexity-limit N
                          how complex (N) a path can become before
                          symex abandons it. Currently uses guard
                          size to calculate complexity.
--symex-complexity-failed-child-loops-limit N
                          how many child branches (N) in an
                          iteration are allowed to fail due to
                          complexity violations before the loop
                          gets blacklisted
--graphml-witness filename
                          write the witness in GraphML format to filename
--symex-cache-dereferences
                          enable caching of repeated dereferences
Backend options:
--object-bits n          number of bits used for object addresses
--external-sat-solver cmd
                          command to invoke SAT solver process
--dimacs                 generate CNF in DIMACS format
--beautify               beautify the counterexample
                          (greedy heuristic)
--smt1                   use default SMT1 solver (obsolete)
--smt2                   use default SMT2 solver (Z3)
--boolector              use Boolector
--cprover-smt2           use CPROVER SMT2 solver
--cvc3                   use CVC3
--cvc4                   use CVC4

```



```

--mathsat          use MathSAT
--yices            use Yices
--z3              use Z3
--refine           use refinement procedure (experimental)
--incremental-smt2-solver cmd
                    command to invoke external SMT solver for
                    incremental solving (experimental)
--outfile filename output formula to given file
--write-solver-stats-to json-file
                    collect the solver query complexity
--refine-strings   use string refinement (experimental)
--string-printable restrict to printable strings (experimental)
--arrays-uf-never  never turn arrays into uninterpreted functions
--arrays-uf-always always turn arrays into uninterpreted functions

```

Other options:

```

--version          show version and exit
--xml-ui           use XML-formatted output
--xml-interface    bi-directional XML interface
--json-ui          use JSON-formatted output
--json-interface   bi-directional JSON interface
--validate-goto-model
                    enable additional well-formedness checks on the
                    goto program
--validate-ssa-equation
                    enable additional well-formedness checks on the
                    SSA representation
--trace-json-extended
                    add rawLhs property to trace
--trace-show-function-calls
                    show function calls in plain trace
--trace-show-code  show original code in plain trace
--trace-hex        represent plain trace values in hex
--compact-trace    give a compact trace
--stack-trace      give a stack trace only
--flush            flush every line of output
--verbosity #      verbosity level
--timestamp <monotonic|wall>
                    print microsecond-precision timestamps.
                    monotonic: stamps increase monotonically.
                    wall: ISO-8601 wall clock timestamps.
--show-array-constraints
                    show array theory constraints added
                    during post processing.
                    Requires --json-ui.

```

References

- [1] Damiano Angeletti, Enrico Giunchiglia, Massimo Narizzano, Alessandra Puddu, and Salvatore Sabina. Using bounded model checking for coverage analysis of safety-critical software in an industrial setting. *J. Automatic Reasoning*, 45(4):397–414, 2010.
- [2] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. Technical Report CMU-CS-03-126, Carnegie Mellon University, 2003.
- [3] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2, 42:230–265, 1936.