

# Verification of C Programs with CBMC - Homework Assignment

Ramón Béjar Torres

May 26, 2022

## Goal

You have to use the bounded model checking tool for C programs, CBMC, for checking the properties we ask for each C program.

Remember that when there are loops, CBMC may not be able to infer the right amount of loop unwinding for all the loops. So, you may need to compute the right amount of loop unwinding using `--unwind together with --unwinding-assertions`.

You must deliver a PDF report giving the answers and solutions for all the problems, including all the **cbmc program executions**, with all the used options, used to find each class of errors. Include also all your modified source files with the added assertions. When you use a counterexample to explain a bug in a program, your report must show **the relevant parts of the counterexample for the explanation**, so it is not necessary to show the full execution trace (only the parts that allow to explain the bug).

## First Problem (3 Points)

Consider the following program for integer division of a negative number by a positive one.

```
1 #include <stdio.h>
2
3 int main( void ) {
4     int D,d, r, q;
5
6     __CPROVER_assume( ( D < 0 ) && ( D >= -300 ) &&
7                       ( d >= 0 ) );
8     r = D;
9     q = 0;
10    while ( r < 0 ) {
11        r = r + d;
12        q--;
13    }
14    printf( " quotient: %d,  remainder: %d\n", q, r );
15 }
```

division.c

You must:

1. Determine if there are any problems when trying to find the minimum number of unwinding needed by the loop for the values we assume in the `__CPROVER_assume` macro. If there is any problem in the assumption that does not allow to have a bounded number of iterations, modify the assumption.
2. Insert assertions, with arithmetic expressions that involve  $D, d, q$  and  $r$ , to check that  $q$  and  $r$  are the quotient and the remainder, respectively, of the integer division of  $D$  by  $d$ . Check the assertions.
3. Check for integer overflow bugs.

*Second Problem (3.5 Points)*

Consider the following C program that finds the maximum even number in an array with N integers:

```

1 #define N 10
2
3 int main()
4 {
5     int numbers[N];
6     int maxeven;
7     int x = 0, i = 0;
8
9     maxeven = 1;
10    for (i = 0; i < N ; i++) {
11        if (numbers[i] % 2 == 0 ) {
12            if (maxeven == 1 || maxeven < numbers[i]) {
13                maxeven = numbers[i];
14            }
15        }
16    }
17    if (maxeven != 1) {
18        // there are even numbers in the array,
19        // So check that maxeven is the greatest one of them
20    }
21    else {
22        // check that there are NO even numbers in the array
23    }
24 }

```

simpleWhile.c

You must:

1. Insert assertions at the end of the program for checking whether
  - (a) If there are even numbers in the array, then maxeven is equal to the greatest even number.
  - (b) When maxeven is equal to 1, then all the numbers in the array are odd.
2. Check that there are not "out of bounds errors" in the lines that operate with the array.

*Third Problem (3.5 Points)*

Consider the following C program that contains a sort function, that it is not implemented, and it should sort an array of  $N$  integers. It also contains a function, `checkSort`, that it is intended to be used for testing, with assertions, that the sort function works as expected.

```

1 #include <stdint.h>
2
3 #define N 8
4
5 void sort (int8_t a[], int size) {
6     // write your favorite sorting algorithm
7
8 }
9
10 void checkSort() {
11
12     int8_t array[N];
13     int i;
14     for (i = 0; i < N; ++i){
15         // Assume numbers in array are integers in range
16         // [0,16]
17         __CPROVER_assume(array[i] >= 0 && array[i] <= 16);
18     }
19
20     sort (array, N);
21
22     // write the assertions to check that the array is sorted
23 }

```

sort.c

You must:

1. Implement the sort function with your favorite sorting algorithm.
2. Write assertions at the end of the `checkSort` function to verify if your sorting algorithm works as expected. Check with `cbmc` the function `checkSort`, but with four different values of  $N$ : {6,8,10,12}. Write down the computation time needed by `cbmc` to verify the program with these four different array sizes. Do you think that the computation time increases exponentially ?