ESCOLA POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA

HARDWARE AND SOFTWARE VALIDATION SYSTEMS

Practical case 2: Verification with CBMC

Joel Aumedes Serrano - 48051307Y
Marc Cervera Rosell - 47980320C

Academic course 2021 - 2022

Bachelor's degree in computer engineering

Contents

List of Figures

1 First problem

1.1 ”division.c” file

Source code:

```
#include <stdio.h>

int main(void){
    int D, d, r, q;
    __CPROVER_assume( (D < 0) && (D >= -300) && (d >= 0) );
    r = D;
    q = 0;
    while(r < 0){
        r = r + d;
        q--;
    }
    printf("quotient: %d, reminder: %d\n", q, r);
}
```

1.1.1 Exercise 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 macro. If there is any problem in the assumption that does not allow to have a bounded number of iterations, modify the assumption.

Indeed, there's a problem in the assumption. We're assuming that the divisor 'd' can be equal to 0. This is a problem because we cannot divide by 0 because of the arithmetic rules, and in case we could, the value of 'r' would never change, in such a way that the loop would never end. Therefore, we've to modify the assumption removing the possibility of 'd' is 0 just modifying the last condition from (d $\geq$ 0) to (d $>$ 0).

The file with the modified assumption looks like:

```c
#include <stdio.h>

int main(void){
    int D, d, r, q;
    __CPROVER_assume( (D < 0) && (D >= -300) && (d > 0) );
    r = D;
    q = 0;
    while(r < 0){
        r = r + d;
        q--;
    }
    printf("quotient: %d, reminder: %d\n", q, r);
}
```

1.1.2 Exercise 2:

Insert assertions, with arithmetic expressions that involve *D*, *d*, *q* and *r*, to check that *q* and *r* are the quotient and the reminder, respectively, of the integer division of *D* and *d*. Check the assertions.

To check if q and r are the quotient and the reminder, one *assertion* has been added after the *printf* command. In such a way that, after adding both assertions, the file code looks like:

```c
#include <stdio.h>

int main(void){
    int D, d, r, q;
    __CPROVER_assume( (D < 0) && (D >= -300) && (d > 0) );
    r = D;
    q = 0;
    while(r < 0){
        r = r + d;
        q--;
    }
    printf("quotient: %d, reminder: %d\n", q, r);
    __CPROVER_assert((D == (q * d) + r), "Division check");
}
```

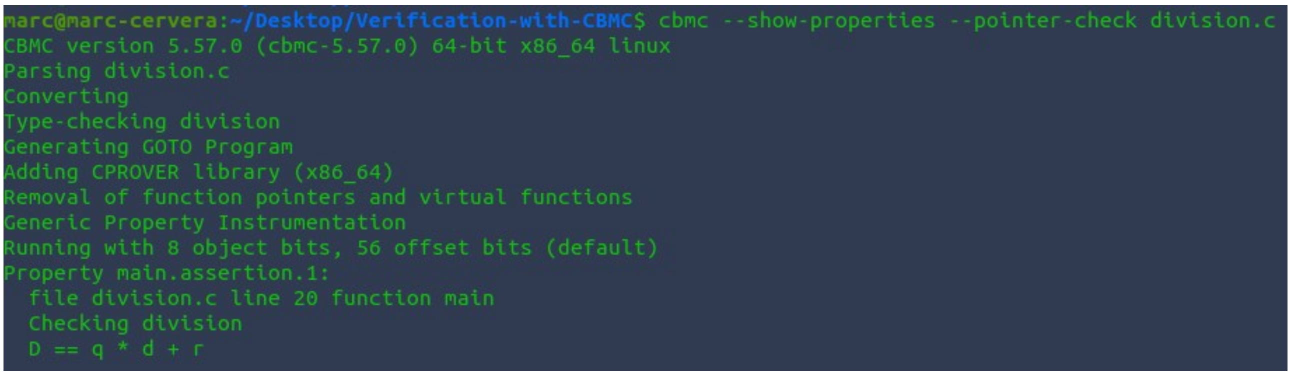
Instead of using one single macro, it could've been two.

```
__CPROVER_assert((q == D / d), "q is the quotient");
__CPROVER_assert((r == D % d), "r is the reminder");
```

But there's a problem. If a small negative number is divided by a big positive number, the real division is equal to -0.000... but the operator '/' rounds this division to -1. Therefore, the *assertion* fails. The reasoning of the second assertion is the same as the first one. The division. The bad calculated division implies a wrong reminder.

```
cbmc --show-properties --pointer-check division.c
```

If the above is typed in the CMD, a list of the properties to check will be shown as it can be seen in the following picture.



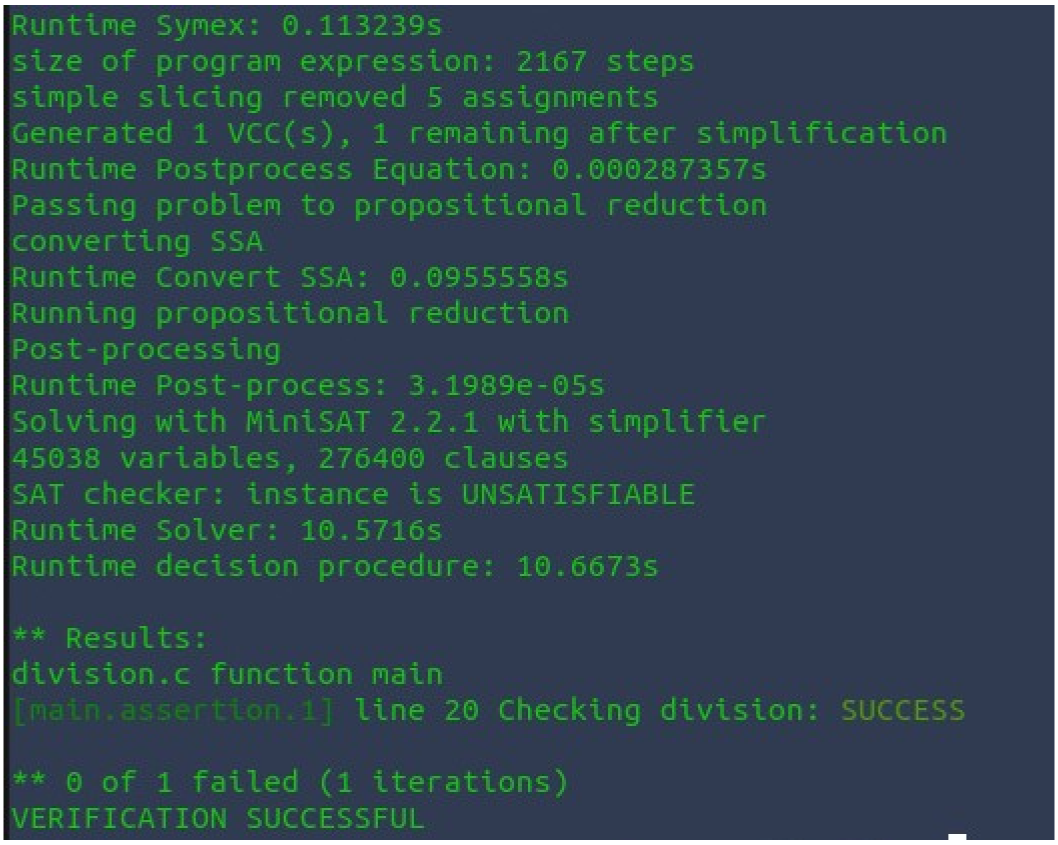
```
marc@marc-cervera:~/Desktop/Verification-with-CBMC$ cbmc --show-properties --pointer-check division.c
CBMC version 5.57.0 (cbmc-5.57.0) 64-bit x86_64 linux
Parsing division.c
Converting
Type-checking division
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Property main.assertion.1:
  file division.c line 20 function main
  Checking division
  D == q * d + r
```

Figure 1: List of properties

The last step is to check the different *asserts*. This will be performed by typing the following command in the CMD:

```
cmbc --unwind k division.c
```

After executing the above command with $k = \mathit{301}$, to perform 300 iterations, the *assertion* works ok. So, the verification is successful, as it can be seen in the following picture.



```
Runtime Symex: 0.113239s
size of program expression: 2167 steps
simple slicing removed 5 assignments
Generated 1 VCC(s), 1 remaining after simplification
Runtime Postprocess Equation: 0.000287357s
Passing problem to propositional reduction
converting SSA
Runtime Convert SSA: 0.0955558s
Running propositional reduction
Post-processing
Runtime Post-process: 3.1989e-05s
Solving with MiniSAT 2.2.1 with simplifier
45038 variables, 276400 clauses
SAT checker: instance is UNSATISFIABLE
Runtime Solver: 10.5716s
Runtime decision procedure: 10.6673s

** Results:
division.c function main
[main.assertion.1] line 20 Checking division: SUCCESS

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

Figure 2: Successful verification of the exercise 1 code.

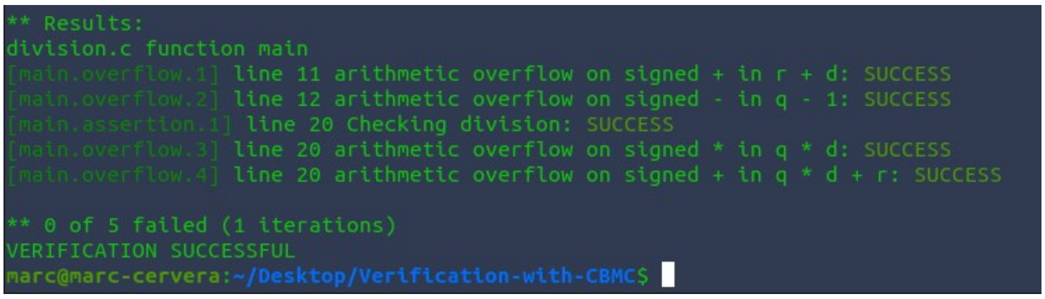
1.1.3 Exercise 3:

Check for integer overflow bugs.

To check for integer overflow bugs, the following command has to be executed in the CMD:

```
cbmc --trace --unwind k --signed-overflow-check division.c
```

After executing the above fixing the value of k to 301, the algorithm cannot find any overflow bug, as can be seen in the following picture.

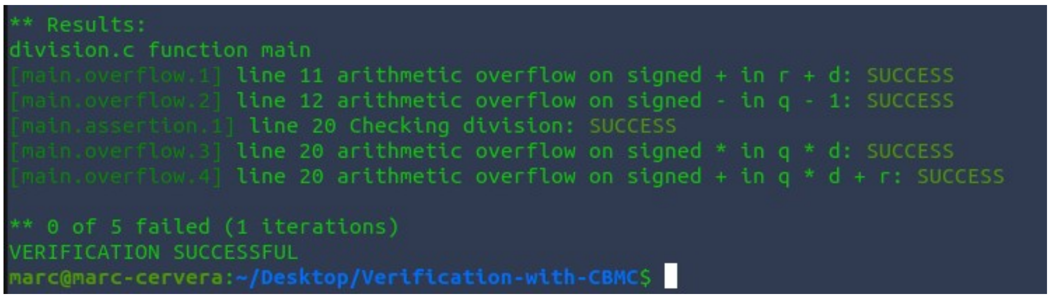


```
** Results:
division.c function main
[main.overflow.1] line 11 arithmetic overflow on signed + in r + d: SUCCESS
[main.overflow.2] line 12 arithmetic overflow on signed - in q - 1: SUCCESS
[main.assertion.1] line 20 Checking division: SUCCESS
[main.overflow.3] line 20 arithmetic overflow on signed * in q * d: SUCCESS
[main.overflow.4] line 20 arithmetic overflow on signed + in q * d + r: SUCCESS

** 0 of 5 failed (1 iterations)
VERIFICATION SUCCESSFUL
marc@marc-cervera:~/Desktop/Verification-with-CBMC$
```

Figure 3: Integer overflow bugs check performing 300 iterations.

To finish this exercise, it will proceed to check if fixing the value of k to 1001, any bug can be found.



```
** Results:
division.c function main
[main.overflow.1] line 11 arithmetic overflow on signed + in r + d: SUCCESS
[main.overflow.2] line 12 arithmetic overflow on signed - in q - 1: SUCCESS
[main.assertion.1] line 20 Checking division: SUCCESS
[main.overflow.3] line 20 arithmetic overflow on signed * in q * d: SUCCESS
[main.overflow.4] line 20 arithmetic overflow on signed + in q * d + r: SUCCESS

** 0 of 5 failed (1 iterations)
VERIFICATION SUCCESSFUL
marc@marc-cervera:~/Desktop/Verification-with-CBMC$
```

Figure 4: Integer overflow bugs check performing 1000 iterations.

As it can be seen, neither with 1000 iterations, the algorithm can find any overflow bug.

2 Second problem

2.1 "simpleWhile.c" file

Source code:

```c
#define N 10
int main(){
    int numbers[N];
    int maxeven;
    int x = 0, i = 0;

    maxeven = 1;
    for(i = 0; i < N; i++){
        if(numbers[i] % 2 == 2){
            if(maxeven == 1 || maxeven < numbers[i]){
                maxeven = numbers[i];
            }
        }
    }
    if(maxeven != 1){
        // there are even numbers in the array,
        //so check that maxeven is the greatest one of them
    } else{
        // check that are NO even numbers in the array
    }
}
```

2.1.1 Exercise 1:

Insert assertions at the end of the program for checking if there are even numbers in the array, then the maxeven is equal to the greatest even number.
The first step, is to add the following code fragment to the source code file to be able to check that, in case that maxeven $\neq 1$, the maxeven is the greatest number of all the even numbers of the array:

```
//...
if(maxeven != 1){
    for(i = 0; i < N; i++){
        __CPROVER_assert(numbers[i] <= maxeven, "The maxeven is >= to any
            numbers[i]");
    }
}
//...
```

The second part of this exerrcise is to check that if maxeven $= 1$, then there are no even numbers in the array. To check this property, the following code fragment has to be added:

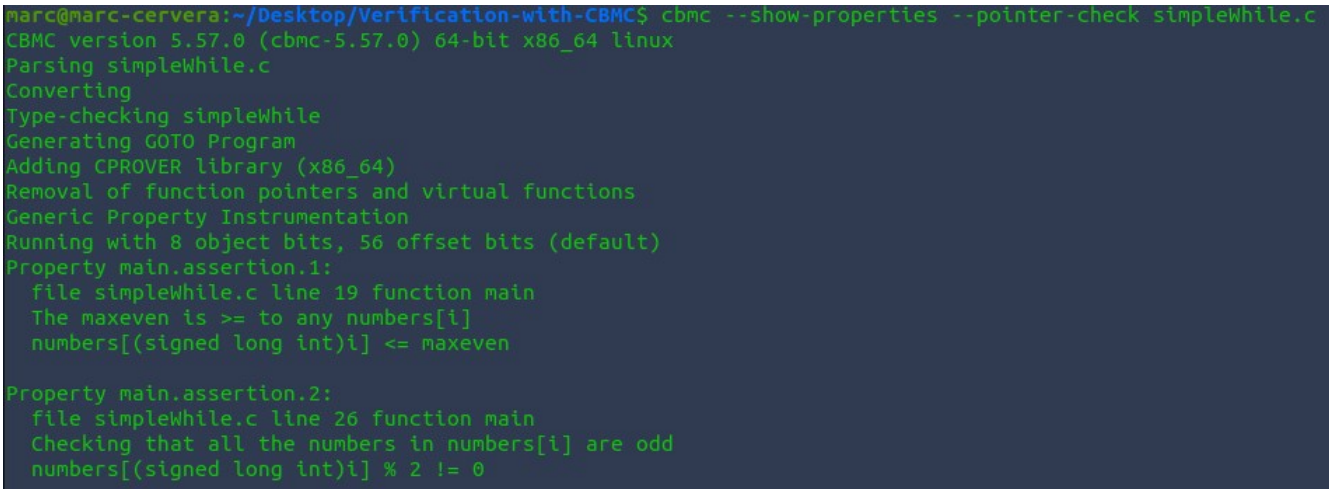
```
/...
else{
    for(i = 0; i < N; i++){
        __CPROVER_assert(numbers[i] % 2 != 0, "Checking that all the numbers
            in numbers[i] are odd.");
    }
}
//...
```

Hence, the full source code file, after the modifications, looks like:

```c
#define N 10
int main(){
    int numbers[N];
    int maxeven;
    int x = 0, i = 0;

    maxeven = 1;
    for(i = 0; i < N; i++){
        if(numbers[i] % 2 == 2){
            if(maxeven == 1 || maxeven < numbers[i]){
                maxeven = numbers[i];
            }
        }
    }
    if(maxeven != 1){
        for(i = 0; i < N; i++){
            __CPROVER_assert(numbers[i] <= maxeven, "The maxeven is >= to any
                numbers[i]");
        }
        // there are even numbers in the array,
        //so check that maxeven is the greatest one of them
    } else{
        for(i = 0; i < N; i++){
            __CPROVER_assert(numbers[i] % 2 != 0, "Checking that all the
                numbers in numbers[i] are odd.");
        }
        // check that are NO even numbers in the array
    }
}
```

If the CBMC show properties command is executed, the following list of properties will be shown:

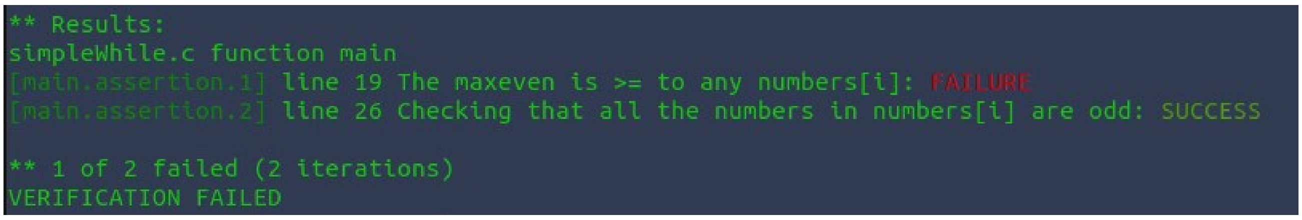


```
marc@marc-cervera:~/Desktop/Verification-with-CBMC$ cbmc --show-properties --pointer-check simpleWhile.c
CBMC version 5.57.0 (cbmc-5.57.0) 64-bit x86_64 linux
Parsing simpleWhile.c
Converting
Type-checking simpleWhile
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Property main.assertion.1:
  file simpleWhile.c line 19 function main
  The maxeven is >= to any numbers[i]
  numbers[(signed long int)i] <= maxeven

Property main.assertion.2:
  file simpleWhile.c line 26 function main
  Checking that all the numbers in numbers[i] are odd
  numbers[(signed long int)i] % 2 != 0
```

Figure 5: Properties list of the second problem

After executing the CBMC verification, one of the *assertions* that has been coded, fail. This is so obvious, because the content of the array is a binary decision. Or all the numbers are even or all the are odd, so one of the *assertions* has to fail. Indeed, the evidences can be seen in the following picture:



```
** Results:
simpleWhile.c function main
[main.assertion.1] line 19 The maxeven is >= to any numbers[i]: FAILURE
[main.assertion.2] line 26 Checking that all the numbers in numbers[i] are odd: SUCCESS

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

Figure 6: Results of the verification

To perform the verification the command used is:

```
cbmc simpleWhile.c
```

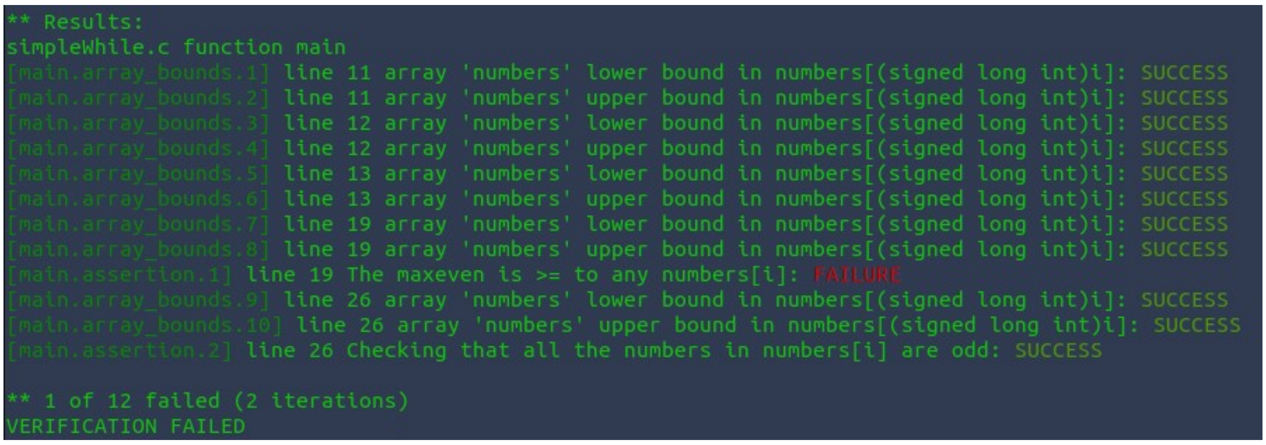
2.1.2 Exercise 2:

Check that there are not ”out of bounds errors” in the lines that operate with the array.

To check if there are ”out of bounds errors”, the following command has to be executed in the CMD:

```
cbmc simpleWhile.c --bounds-check --pointer-check
```

In the following picture, can be seen that executing the above command, all the `array_bounds` pass the verification successfully.



```
** Results:
simpleWhile.c function main
[main.array_bounds.1] line 11 array 'numbers' lower bound in numbers[(signed long int)i]: SUCCESS
[main.array_bounds.2] line 11 array 'numbers' upper bound in numbers[(signed long int)i]: SUCCESS
[main.array_bounds.3] line 12 array 'numbers' lower bound in numbers[(signed long int)i]: SUCCESS
[main.array_bounds.4] line 12 array 'numbers' upper bound in numbers[(signed long int)i]: SUCCESS
[main.array_bounds.5] line 13 array 'numbers' lower bound in numbers[(signed long int)i]: SUCCESS
[main.array_bounds.6] line 13 array 'numbers' upper bound in numbers[(signed long int)i]: SUCCESS
[main.array_bounds.7] line 19 array 'numbers' lower bound in numbers[(signed long int)i]: SUCCESS
[main.array_bounds.8] line 19 array 'numbers' upper bound in numbers[(signed long int)i]: SUCCESS
[main.assertion.1] line 19 The maxeven is >= to any numbers[i]: FAILURE
[main.array_bounds.9] line 26 array 'numbers' lower bound in numbers[(signed long int)i]: SUCCESS
[main.array_bounds.10] line 26 array 'numbers' upper bound in numbers[(signed long int)i]: SUCCESS
[main.assertion.2] line 26 Checking that all the numbers in numbers[i] are odd: SUCCESS

** 1 of 12 failed (2 iterations)
VERIFICATION FAILED
```

Figure 7: Array bounds verification

The global verification fails because of the already commented at the first exercise of this problem.

3 Third problem

3.1 ”sort.c” file

Source code:

```
#include <stdint.h>
#define N 8

void sort(int8_t a[], int size){
    // write your favorite sorting algorithm
}

void checkSort(){
    int8_t array[N];
    int i;
    for (i = 0; i < N; ++i){
        // Assume numbers in array are integers in range [0, 16]
        __CPROVER__assume(array[i] >= 0 & array[i] <= 16);
    }

    sort(array, N);

    //write the assertions to check that the array is sorted
}
```

3.1.1 Exercise 1:

Implement the sort function with your favorite sorting algorithm.

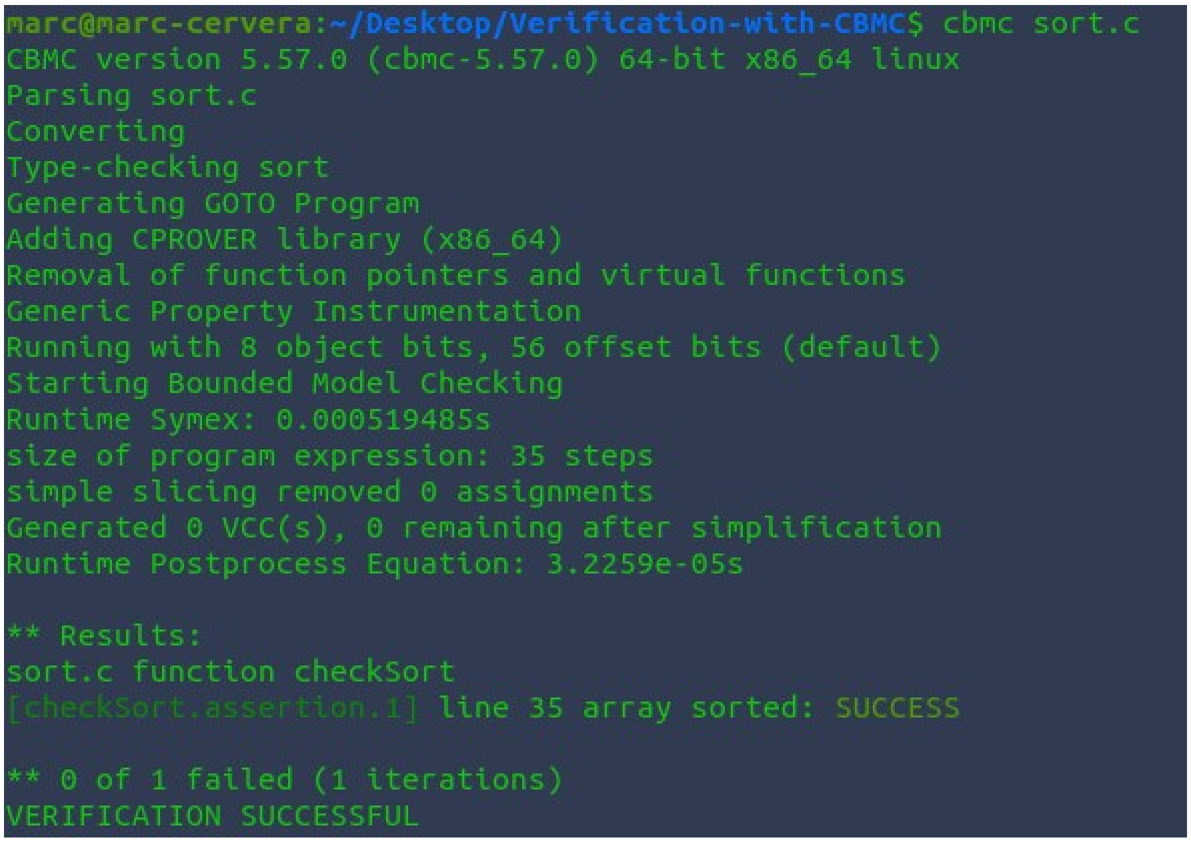
The chosen sorting algorithm is the bubble sort algorithm. To know how works the algorithm this link has to be followed: Bubble sort. Once the algorithm is implemented, the sort function code looks like:

```
//...
void sort (int8_t a[], int size) {
   /*Bubble sort algorithm*/
   int i, j;
   for(i = 0; i < size - 1; i++){
      for(j = 0; j < size - i - 1; j++){
         if(a[j] > a[j + 1]){
            swap(&a[j], &a[j + 1]);
            }
        }
    }
}
//...
```

3.1.2 Exercise 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?

Test for an array of 6 positions:



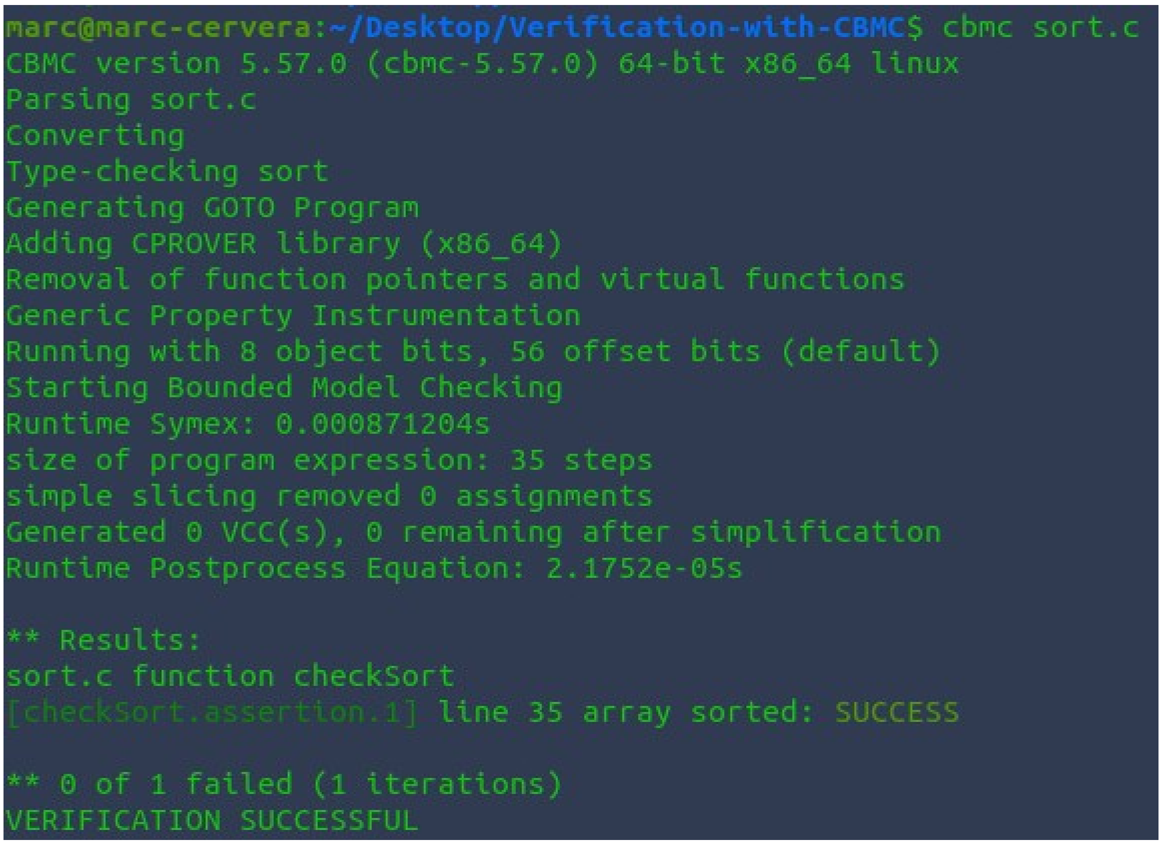
```
marc@marc-cervera:~/Desktop/Verification-with-CBMC$ cbmc sort.c
CBMC version 5.57.0 (cbmc-5.57.0) 64-bit x86_64 linux
Parsing sort.c
Converting
Type-checking sort
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
Runtime Symex: 0.000519485s
size of program expression: 35 steps
simple slicing removed 0 assignments
Generated 0 VCC(s), 0 remaining after simplification
Runtime Postprocess Equation: 3.2259e-05s

** Results:
sort.c function checkSort
[checkSort.assertion.1] line 35 array sorted: SUCCESS

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

Figure 8: Testing the sorting algorithm with N = 6

Time $\longrightarrow$ 3.2259x10^{-5}*seconds.*



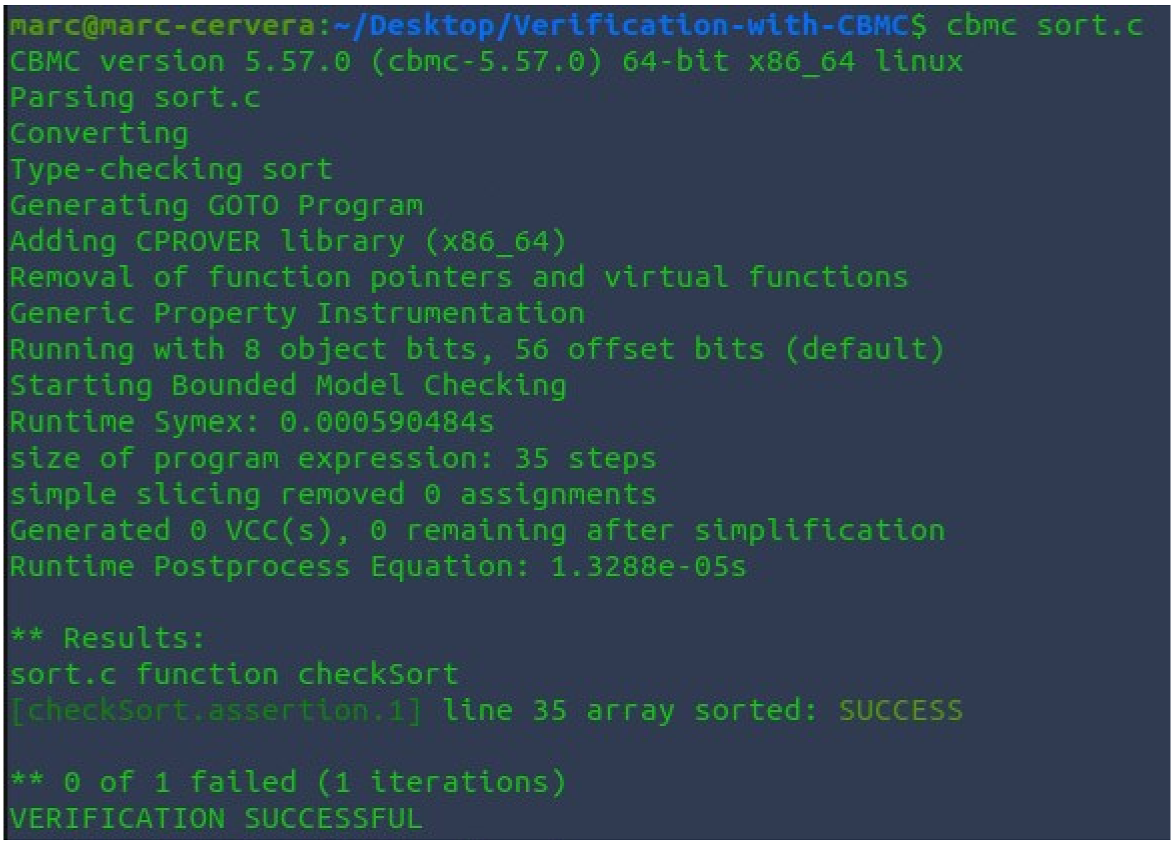
```
marc@marc-cervera:~/Desktop/Verification-with-CBMC$ cbmc sort.c
CBMC version 5.57.0 (cbmc-5.57.0) 64-bit x86_64 linux
Parsing sort.c
Converting
Type-checking sort
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
Runtime Symex: 0.000871204s
size of program expression: 35 steps
simple slicing removed 0 assignments
Generated 0 VCC(s), 0 remaining after simplification
Runtime Postprocess Equation: 2.1752e-05s

** Results:
sort.c function checkSort
[checkSort.assertion.1] line 35 array sorted: SUCCESS

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

Figure 9: Testing the sorting algorithm with N = 8

Time $\longrightarrow$ 2.1752x$10^{-5} seconds$.



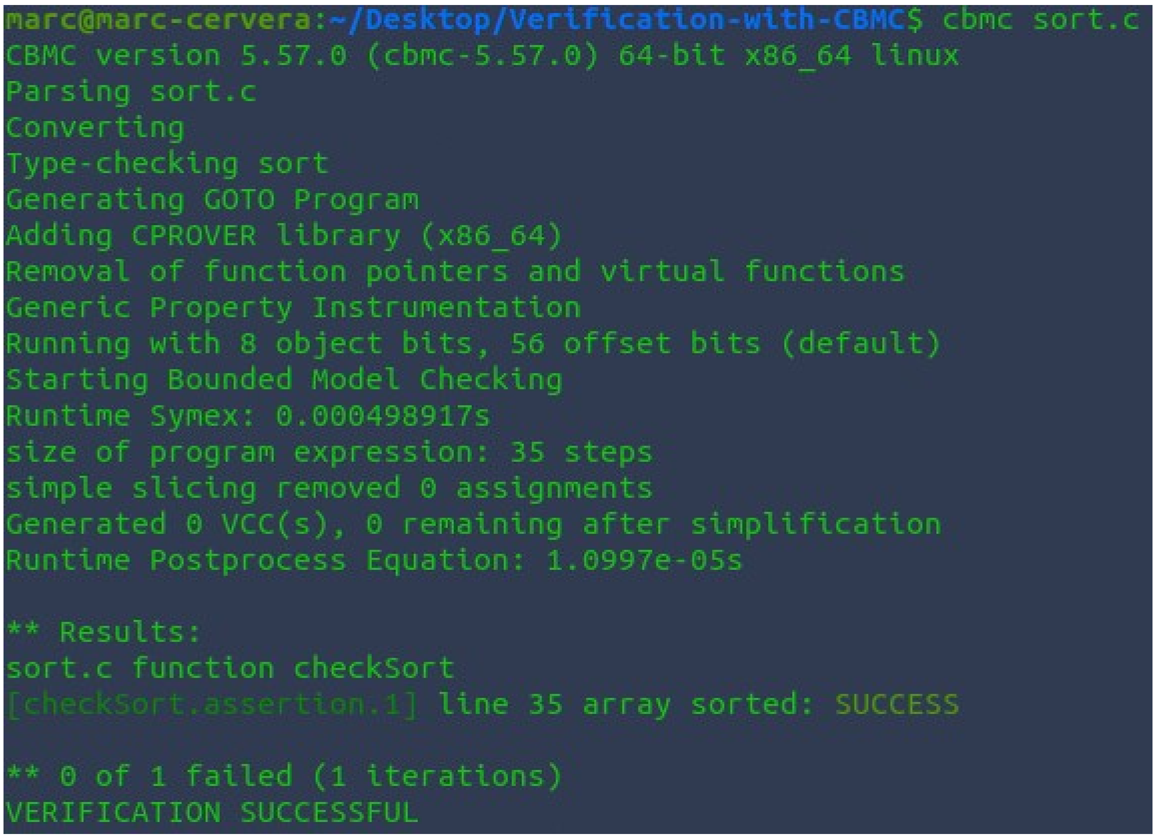
```
marc@marc-cervera:~/Desktop/Verification-with-CBMC$ cbmc sort.c
CBMC version 5.57.0 (cbmc-5.57.0) 64-bit x86_64 linux
Parsing sort.c
Converting
Type-checking sort
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Generic Property Instrumentation
Running with 8 object bits, 56 offset bits (default)
Starting Bounded Model Checking
Runtime Symex: 0.000590484s
size of program expression: 35 steps
simple slicing removed 0 assignments
Generated 0 VCC(s), 0 remaining after simplification
Runtime Postprocess Equation: 1.3288e-05s

** Results:
sort.c function checkSort
[checkSort.assertion.1] line 35 array sorted: SUCCESS

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

Figure 10: Testing the sorting algorithm with N = 10

Time $\longrightarrow$ 1.3288x10^{-5}*seconds*.



```
marc@marc-cervera:~/Desktop/Verification-with-CBMC$ cbmc sort.c
CBMC version 5.57.0 (cbmc-5.57.0) 64-bit x86_64 linux
Parsing sort.c
Converting
Type-checking sort
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
Runtime Symex: 0.000498917s
size of program expression: 35 steps
simple slicing removed 0 assignments
Generated 0 VCC(s), 0 remaining after simplification
Runtime Postprocess Equation: 1.0997e-05s

** Results:
sort.c function checkSort
[checkSort.assertion.1] line 35 array sorted: SUCCESS

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

Figure 11: Testing the sorting algorithm with N = 12

Time $\longrightarrow$ 1.0997x10^{-5}*seconds*.

To check if the time has en exponential order, the last step is to create a graphic with all the times.

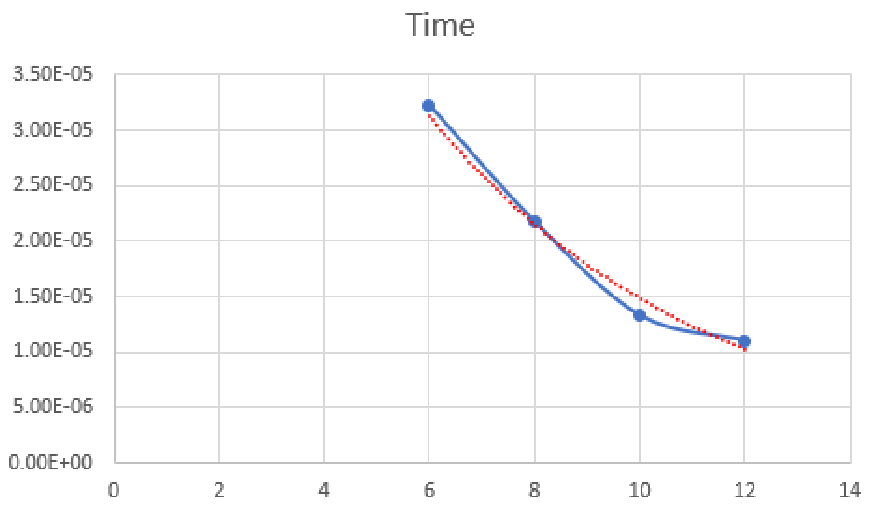


Figure 12: Execution times with an exponential trend line

As it can be seen, the exponential trend line does not fit very well. As it can be seen in the following picture the trend line that fits better is a polynomial trend line of third order:

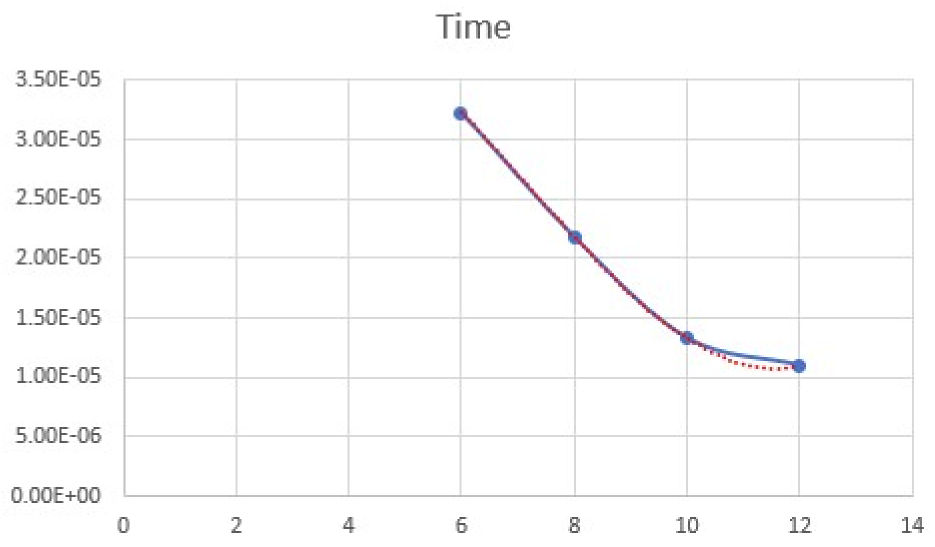


Figure 13: Execution times with a polynomial trend line