



---

# 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

<b>1</b>	<b>First problem</b>	<b>1</b>
1.1	"division.c" file . . . . .	1
1.1.1	Exercise 1: . . . . .	1
1.1.2	Exercise 2: . . . . .	2
1.1.3	Exercise 3: . . . . .	4
<b>2</b>	<b>Second problem</b>	<b>5</b>
2.1	"simpleWhile.c" file . . . . .	5
<b>3</b>	<b>Third problem</b>	<b>6</b>
3.1	"sort.c" file . . . . .	6

## List of Figures

1	List of properties . . . . .	3
2	Successful verification of the exercise 1 code. . . . .	4
3	Integer overflow bugs check performing 300 iterations. . . . .	4
4	Integer overflow bugs check performing 1000 iterations. . . . .	5

# 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, remainder: %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:

---

```
#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, remainder: %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 remainder, respectively, of the integer division of  $D$  and  $d$ . Check the assertions.

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

---

```
#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, remainder: %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");
```

---

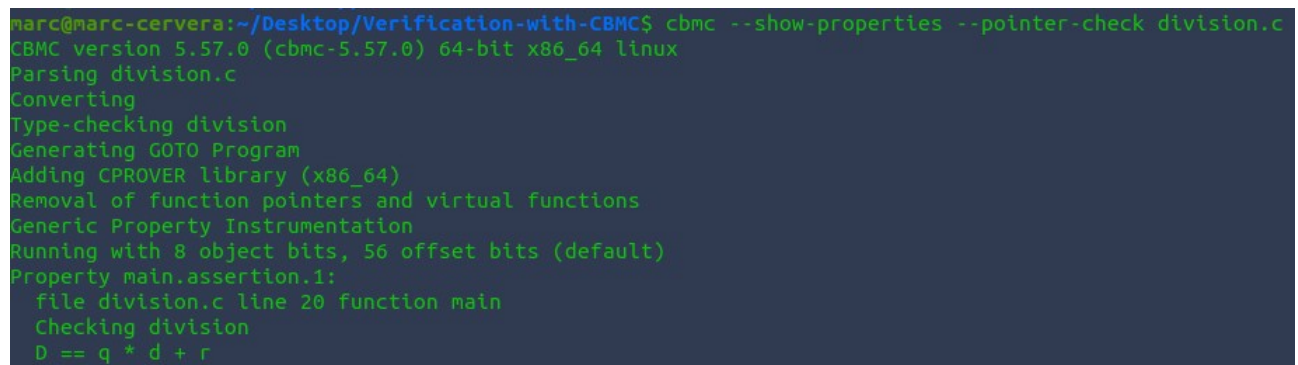
But there's a problem. If a small negative number is divided by a big positive number, the real division is equal to -0.000... but the operator '/' rounds this division to -1. Therefore, the *assertion* fails. The reasoning of the second assertion is the same as the first one. The division. The bad calculated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

---

```
cbmc --unwind k division.c
```

---

After executing the above command with  $k = 301$ , 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:

---

```
#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
    }
}
```

---

## 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
}
```

---