Homework Assignment

Verification of C Programs with CBMC

Authors:

Francisco Javier Roig Gregorio - 47433543E

Jose Miguel Avellana López - 18068091G

Course 2020 – 2021

First Problem - division

1. Determine if there are any problems when trying to find the minimun 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.

The loop can never end because d can be assigned to 0 and so, r never be incremented.

We can modify it with the assumption d>0, and then the verification is successful.
Using the command:

cbmc --unwind 301 --stop-on-fail --unwinding-assertions --trace division.c

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 by d. Check the assertions.

For assert that we use:

__CPROVER_assert((D == (q*d + r)) && (r >= 0) && (q < 0)

because:

1. D == (q*d + r)) verifies the correct division relation,
2. r can't be lower than 0,
3. q can't be upper or equal than 0

This assertion is checked doing the command:

cbmc --unwind 301 --stop-on-fail --unwinding-assertions --trace division.c

And seeing that the verification is correct.

3. Check for integer overflow bugs.

For checking integer overflows we use the cbmc's tool --signed-overflow-check for that enables signed arithmetic over- and underflow checks.

So we use the command:

cbmc --unwind 301 --stop-on-fail --unwinding-assertions --trace --signed-overflow-check division.c

And it says that the verification is successful.

Second Problem - simpleOdd

1. Insert assertions at the end of the program for checking whether both functions (checkOddArray1 and checkOddArray2) return the same result for the same input. Check the assertions.

For assert that we use:

__CPROVER_assert(checkOddArray1(a) == checkOddArray2(a),);

And, when we check it doing:

cbmc --trace --stop-on-fail --function checkOdd simpleOdd.c

It says that the verification fails.

2. If you find a counterexample, use it to explain where is the error in function checkOddArray2 (identify the first iteration where the current value of the variable odd is wrong). And then fix what this function does in every iteration such that the final value of the variable odd is right.

A counterexample can be { 17, 18, -17, -1, 0, -9, 17, -18 }, on the fourth iteration (i=3) of the loop the odd value is -1.

It is because the % operator in C is not the modulo operator but the remainder operator. Modulo and remainder operators differ with respect to negative values. With a remainder operator, the sign of the result is the same as the sign of the dividend while with a modulo operator the sign of the result is the same as the divisor.

An option to solve it would be:

```
int checkOddArray2(int8_t a[])  {
    int i, odd, tmp;

    odd = 0;
    for (i = 0; i < N ; i++) {
        tmp = (odd + a[i])  %  2;
        if(tmp<0){
            tmp = tmp * -1;
        }
        odd = tmp;
    }
    return odd;
}
```

It could also have been solved using the mod function, as in the first checkOddArray1 algorithm.

Third Problem - sort

1. Implement the sort function with your favorite sorting algorithm.

We use the bubble sort algorithm.

2.Write assertions at the end of the checkSort function to verify if your sorting algorithm works as expected.

We assert the sort function with:

```
    // check if the array is sorted.
    unsigned is_sorted = 1;
    for (int i=1; i <  N; ++i){

        // write the assertions to check that the array is sorted
        //__CPROVER_assert( array[i-1] <= array[i] ,"check that the previous is
upper or equal.");
        if (array[i-1] > array[i]){
            is_sorted = 0;
            break;
        }
    }

    // write the assertions to check that the array is sorted
    __CPROVER_assert( is_sorted==1 ,"check that the array is sorted.");
```

Using the command:

cbmc --trace --stop-on-fail --function checkSort sort.c

We could use the in-loop assert too.

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.

N=6 -> 0.334s
N=8 -> 0.385s
N=10 -> 0.582s
N=12 -> 1.254s

Do you think that the computation time increases exponentially ?

Yes, the computation time increases exponentially, this exponential increase is due to the validation that is carried out using the sat solver.

3. Introduce a small error in the implementation of your sort function. Then, check it again to see if cbmc finds the error.

The error introduced in the bubble sort algorithm is to assume that the first elements of the array are already in order, and not the last elements as it really is.

```
void sortError (int8_t a[], int size) {
    // write your favorite sorting algorithm
   int i, j;
   for (i = 0; i < size-1; i++)

       // First i elements are already in place?
       for (j = 1; j < size-i; j++)
           if (a[j] > a[j+1])
              swap(&a[j], &a[j+1]);
}
```