Verification of C Programs with CBMC - Homework Assignment

Ramón Béjar Torres

May 29, 2021

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.

```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 );
}
```

division.c

You must:

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.
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.
3. Check for integer overflow bugs.

Second Problem (3.5 Points)

Consider the following C program that contains two functions for computing if the sum of the numbers in a array is an odd number:

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

int mod(int a, int b) {
    int r = a % b;
    return r < 0 ? r + b : r;
}

int checkOddArray1(int8_t a[]) {
    int i, odd;

    odd = 0;
    for (i = 0; i < N ; i++) {  odd = odd + a[i]; }
    odd = mod( odd , 2 );
    return odd;
}

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

    odd = 0;
    for (i = 0; i < N ; i++) { odd = (odd + a[i])  %  2; }
    return odd;
}

int checkOdd( ) {
  int8_t  a[N], t;
  int i;

  for (i = 0; i < N; ++i){ __CPROVER_assume(a[i] >= -18 && a[i] <= 18);  }

  // Write here assertions to check if checkOddArray1
  // and checkOddArray2 return the same result for the
  // same input
}
```

simpleOdd.c

You must:

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.
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.

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.

```c
#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

}
```

sort.c

You must:

- Implement the sort function with your favorite sorting algorithm.
- 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 ?
- Introduce an small error in the implementation of your sort function. Then, check it again to see if cbmc finds the error.