Bounded Model Checking - Exercises

Ramón Béjar Torres

May 19, 2021

Introduction

In the following exercices, use the syntax of assertions and assumptions of the CBMC tool to be able to verify the properties asked in the exercises. You may then use CBMC to check whether your solution achieves its goal. However, if your assertions and assumptions are not correct, then it may happen that they are still checked right by CBMC even if they are not expressing what it was asked at the exercise.

Take into account also that when using CBMC, for checking whether your assertions and assumptions work as expected, if there are loops in the program CBMC **may fail** to find automatically the right level of loop unwinding for the loops of the program. In those cases, you will have to provide your own computed bounds for the loop unwindings with the flags that CBMC allows to use.

BMC for Ansi-C Programs

1. Consider the following function for deciding whether an integer number a differ is a prime number. [1]

```
int prime( unsigned int a ) {
  unsigned int d, bprime, l;

  if (a <= 3)  return 1;
  if (a % 2 == 0)  return 0;

  bprime = 1;
  d = 3;
  l = a/2 + 1;
  while ( d <= l && bprime) {
    if (a % d == 0) bprime = 0;
    d += 2;
  }

  return bprime;
}
```

[1] This is not, by far, the most efficient algorithm for primality testing !, but its verification is easier.

Use assumptions and assertions to check these properties of the function prime: [2]

(a) If The function returns 1, then a is a odd number, for any

[2] Remember that if you use CBMC to check these properties, you may need to manually provide bounds for loop unwinding.

integer number in the range $[1, 500]$.

(b) After the loop finishes, it satisfies the property:

$$\{\forall i \ (3 \leq i < d \wedge a > 4 \wedge bprime = 1) \rightarrow (a \ mod \ i) \neq 0\}$$

when `a` is in the range $[1, 50]$.

2. Write an `Ansi-C` program for determining whether an array `a[N]` with N integer elements is an ascending array (i.e. any element `a[i]` is greater than or equal to `a[i-1]`). Check this property for $N = 20$, but bounding each element $a[i]$ in the range $[-100, 100]$ using `CBMC` assumptions.

3. Consider the following function that computes an integer number function over an array `a[N]` with N integer elements:

```
#define N 20

int sumall( int a[], int n ) {
  int  sum = 0;
  int i;

  for ( i = 0 ; i < n ; i++) {
    sum += a[i];
  }
  return sum;
}

int main( void ) {
  int a[N];

  return sumall( a, N );
}
```

Use assumptions and assertions to check these properties of the function:

(a) Use `CBMC` with automatic assertion insertion to verify whether the program is free from over/underflow bugs when N is 1.

(b) Use `CBMC` with automatic assertion insertion to verify whether the program is free from integer over/underflow bugs when N is 10.

4. Consider the following function that swaps the value of two integer variables `a` and `b`:

```
void swap( int *a, int *b ) {
  *a = *a + *b;
  *b = *a - *b;
```

```
  *a = *a - *b;
}
```

Use assumptions and assertions to check these properties of the `swap` function:

(a) When the function ends, the value of `a` and `b` has been swapped [3] when $a, b \in [1, 20000]$.

(b) The program is free from integer over/underflow bugs (use automatic assertion insertion options for this) when $a, b \in [1, 30000]$.

[3] Observe that for expressing this, you will have to include additional code in the program for being able to refer to the initial value of `a` and `b`

5. Consider the following recursive function that computes a real number based on a recursive definition:

```
float f( float a, unsigned int n ) {
  float t;

  if ( n == 0 ) return 1.0;

  t = f( a, n-1 );

  return 1.0/a-t;
}
```

Use assumptions and assertions to check these properties of the function:

(a) Use `CBMC` with automatic assertion insertion to verify whether the program is free from division by zero errors when $a \in [1.0, 3.0]$ and $n \in [1, 10]$.

(b) Use `CBMC` with automatic assertion insertion to verify whether the program is free from division by zero and *not a number value* bugs when $a \in [2.0, 3.0]$ and $n \in [1, 10]$.

(c) If any counterexamples (bugs) are found in the two previous steps, explain the source of the bugs using the counterexamples.

6. Consider the following program for performing the shuffle of an array with N integers [4]:

[4] Beware: this is not the best algorithm for shuffling an array, given that it does not produce any possible permutation with equal probability.

```
int nondet_int( void );

void swap( int c[], int i, int j ) {
   int t;

   t = c[i];
   c[i] = c[j];
   c[j] = t;
}
```

```
void shuffle( int c[], int N ) {
  int i, uj, uk;

  for ( i = 0; i < (2*N) ; i++ ) {
      uj = nondet_int();
      uk = nondet_int();
      __CPROVER_assume( uj >= 0 && uj < N &&
                        uk >= 0 && uk < N );
      swap( c, uj, uk );
  }
}

int main( void ) {
   int c[10];

   shuffle( c, 10 );
}
```

Use assumptions and assertions to check these properties of the program [5]:

(a) After the call to function `shuffle` inside the main function, it always holds that `c[3] != c[7]`.

(b) Repeat the previous checking, but this time initialize the array with the sequence $[1,2,3,4,5,6,7,8,9]$.

7. Consider the following program for computing the largest (`max`) and the smallest (`min`) values from an array with `N` integers and the value:

$$med = \lfloor \frac{max - min}{2} \rfloor$$

This is the program, for the particular value $N = 10$:

```
#define N 10

int main( void ) {
  int c[N];
  int min, max, med;

  min = c[0];
  max = c[0];
  for ( i = 1 ; ( i < N ) ; i++ ) {
    if ( min > c[i] ) min = c[i];
    if ( max < c[i] ) max = c[i];
  }
  med = max-min/2;
```

[5] Observe that in this program we use a function with prefix `nondet` but we only give its prototype, not its code. This represents for `CBMC` a function that can return any possible value for its return type (integer in our case), so when checking assertions `CBMC` will consider execution paths for all possible return values for each point where the function is called. Also, observe that we have included an assumption macro call to filter only the good return values we are interested in: values in the range $[0, N-1]$.

```
}
```

Use assumptions and assertions to check these properties of the program:

(a) The program computes in the `min`, `max` and `med` variables the expected values when the array values are in the range $[0, 500]$. [6]

(b) As before, but now assume the array values are in the range $[0, 10000]$.

[6] You may need to insert new variables and code into the program for asserting expressions that check the right values of these two variables.