CSC2108: Automated Verification
Assignment 3.
Due: November 14, classtime.

1. Recall the notion of *alternation depth* in μ-calculus formulas. An alternation depth of a formula is one if results of a least fixpoint are not used by the greatest fixpoint, or vice versa.

   (a) Consider the CTL formula

   $$AG(p \Rightarrow AF(q \Rightarrow AGr))$$

   Translate it into μ-calculus.

   (b) What is the alternation depth of the above formula?

   (c) Prove that this is an upper bound for the alternation depth for an arbitrary CTL formula expressed in μ-calculus. Hint: prove this by induction on the structure of the formula.

   (d) Now consider the following formula expressed in fair CTL:

   $$EGf = \nu Z.f \ \wedge \ \bigwedge_{k=1}^{n} EXE[f \ U \ (Z \ \wedge \ P_k)]$$

   What is the alternation depth of this formula, when expressed in μ-calculus?

   (e) What is the alternation depth of every fair CTL property, when translated into μ-calculus?

2. Using CBMC. For this problem we will use the CBMC program analysis tool. You can run it using the following command:

   ```
   cmbc --unwind 5 --function foo program.c
   ```

   For this problem, you will use the following C functions:

```
/** GLOBALS */
int buf[4];
int hi = 0;
int lo = 0;
int size = 4;

/** FUNCTION enqueue */
void enqueue (int x)
{
  buf [hi] = x;
```

```
  hi = (hi + 1);
}

/** FUNCTION dequeue */
int dequeue ()
{
  int res = buf [lo];
  lo = (lo + 1);
  return res;
}

/** FUNCTION queue_test */
void queue_test (void)
{
  while (nondet_int ())
    {
      if (nondet_int ())
        {
          int x = nondet_int ();
          enqueue (x);
        }
      else
        dequeue ();
    }
}

/** FUNCTION BINSEARCH */
int binsearch (int x)
{
  int a[16];
  signed low = 0, high = 16;

  while (low < high)
  {
    signed middle = low + ((high - low) >> 1);

    if (a[middle]<x)
      high = middle;
    else if (a [middle] > x)
      low = middle + 1;
    else /* a [middle] == x ! */
      return middle;
  }
 return -1;
```

```
}
```

(a) Use CBMC to show that the function `queue_test` can cause a buffer overflow. Show the execution that causes an overflow. What was the smallest unwinding bound required for finding this counterexample? (Hint: you will want to disable unwinding assertions (`--no-unwinding-assertions` for this example).

(b) Fix the buffer overflows in the previous example. Use `CBMC` to verify that there is no overflow after the loop is unrolled 40 times. Can this result be used to conclude that the program has no buffer overflows?

(c) Use CBMC to check whether the function `binsearch` accesses the array `a` outside of its bounds in 5 steps or less (i.e., use unwind bound of 5). What can we learn from the result? Is the function safe, unsafe, or nothing is known?

(d) Repeat part (c) with unwinding bound of 10. What can you conclude from the result?

3. From Programs to Constraints. For this problem, you will need to convert a program into a set of constraints. The constraints should be such that every satisfying assignment corresponds to an execution of the program that violates an assertion. The program is given below:

```
void main (void)
{
  int    i;
  char * buf;
  char   last;

  if (buf [i] == '\0')
    {
      int start = 0;
      while (start < i)
        {
          buf [start] = 'f';
          start++;
          last = buf [start - 1];
        }
    }
}
```

(a) Show the result of unrolling the loop twice (2) and adding an unwinding assertion.

(b) Convert the loop-free program (with assertion) from part (a) into Single Static Assignment (SSA) form.

(c) Generate a constraint system C from the SSA program in part (a) such that C is satisfiable if and only if the unwinding assertion can be violated.

(d) Is the constraint system C from part (c) satisfiable? If yes, show a satisfying assignment.

4. Checking Custom Properties. Locks are commonly used to grant exclusive access to resources. Assume that we have a locking API consisting of two functions: `lock`, and `unlock`. The function `lock` acquires a lock, and the function `unlock` releases it. Furthermore, a proper use of the API must conform to a locking discipline: a lock cannot be acquired (released) twice in a row without a corresponding `unlock` (`lock`) in between. For example, a sequence of calls

```
lock (); unlock (); lock (); unlock ();
```

is legal, and the sequence

```
lock (); lock (); unlock (); unlock ();
```

is not.

In this problem you will need to use CBMC to check whether the following two functions conform to the locking discipline or not.

```
/** prototypes */
int nondet_int ();
void lock (void);
void unlock (void);
int read_input (void) { return nondet_int (); }

/** FUNCTION LOCK1 */
void lock1 (void)
{
  int in_irq;
  int buf[10];
  int i;

  if (in_irq) lock ();

  for (i = 0; i < 5; i++)
    buf [i] = read_input ();

  if (in_irq) lock ();
}
```

```
/** FUNCTION LOCK2 */
void lock2 (void)
{
  int request, old, total;

  request = 0;
  do {
    lock ();
    old = total;
    request = read_input ();
    if (request)
      {
        unlock ();
        total = total + 1;
      }
  } while (total != old);

  unlock ();
}
```

(a) Instrument the two functions above with asserts to check that a lock is never acquired or released twice in a row.

(b) Use CBMC to check whether the function `lock1` satisfies the locking discipline. Is the function correct? If not, show an erroneous execution.

(c) Use CBMC to check whether the function `lock2` incorrectly acquires or releases a lock twice in a row within 20 executions of the loop. What can you conclude from the result of the analysis? Can the program be considered safe? Will increasing the unwinding bound be useful?

(d) Use CBMC to check that if the lock is not held before execution of the body of the loop in `lock2`, then the lock is held after execution of the body of the loop if and only if `total == old`. Does the result indicate that the program is correct or not? Explain your answer.