VU Programm- und Systemverifikation
Homework: Hoare Logic and Bounded Model Checking

Due date: May 26, 2020, 1pm

Task 1 (2 points): Use Hoare's loop rule to prove the Hoare Triple below. The variable `done` is of type Boolean, and `i` is an integer. The function `f` is an *arbitrary* function that takes `i`, performs a complex computation and returns a new integer value. (For proving the Hoare Triple below, it is completely irrelevant what `f` computes!)

$\{\mathsf{true}\}$ (strengthen pre-condition with consequence rule)

$\{\mathsf{true} \vee i = 42\}$ (assignment rule)

```
done = false;
```

$\{\neg done \vee i = 42\}$ (loop invariant)

```
while (!done) {
```

$\{(\neg done) \wedge (\neg done \vee i = 42)\}$ (consequence rule)

$\{(\neg done \vee f(i) = 42)\}$

```
  i = f(i);
```

$\{(\neg done \vee i = 42)\}$ (consequence rule)

$\{(\neg done \vee i = 42) \wedge (i \neq 42 \vee i = 42)\}$ (consequence rule)

$\{(\neg done \wedge i \neq 42) \vee i = 42\}$ (consequence rule)

$\{(\neg(done \vee (i = 42)) \vee i = 42)\}$ (assignment rule)

```
  done = done || (i == 42)
```

$\{(\neg done \vee i = 42)\}$ (invariant)

}

$\{done \wedge (\neg done \vee i = 42)\}$ (loop rule)

$\{(\mathsf{i} == 42)\}$ (consequence rule)

Task 2 (8 points): Prove the Hoare Triple below (assume that the domain of all variables except `done` in the program are the unsigned integers including zero, i.e., $\mathtt{i}, \mathtt{m}, \mathtt{n} \in \mathbb{N} \cup \{0\}$, and that `done` is a Boolean variable). You need to find a sufficiently strong loop invariant.

Annotate the following code directly with the required assertions. Justify each assertion by stating which Hoare rule you used to derive it, and the premise(s) of that rule. If you strengthen or weaken conditions, explain your reasoning.

$\{\mathsf{true}\}$

```
if (m > n)
  i = n;
else
  i = m;
```

$\{\mathsf{true}\}$ (strengthen pre-condition)

$\{\mathsf{false} \Rightarrow (m\%i = 0)\}$ (assignment rule)

```
done = false;
```

$\{(done \Rightarrow m\%i = 0)\}$ (loop rule, pre-condition)

```
while ((i > 1) && !done) {
```

$\{((i > 1) \wedge \neg done) \wedge (done \Rightarrow (m\%i = 0))\}$ (strengthen pre-condition again)

Note that the consequence of the implication doesn't matter, since *done* is false anyway

$\{((i > 1) \wedge \neg done) \wedge (done \Rightarrow (m\%(i-1) = 0))\}$ (strengthen pre-condition)

$\{(done \Rightarrow (m\%(i-1) = 0))\}$ (conditional rule)

```
  if ((m % i == 0) && (n % i == 0))
```

$\{(m\%i = 0) \wedge (done \Rightarrow m\%(i-1) = 0)\}$ (strengthen pre-condition)

$\{(true \Rightarrow (m\%i = 0)\}$ (assignment rule)

```
    done = true;
```

$\{(done \Rightarrow (m\%i = 0)\}$ (loop invariant)

```
  else
```

$\{\neg(m\%i = 0) \wedge (done \Rightarrow m\%(i-1) = 0)\}$ (strengthen pre-condition)

$\{(done \Rightarrow ((m\%(i-1) = 0)\}$ (assignment rule)

```
    i = i - 1;
```

$\{(done \Rightarrow (m\%i == 0)\}$ (loop invariant)

```
}
```

$\{(i \leq 1 \vee done) \wedge (done \Rightarrow ((m\%i == 0))\}$ (loop rule)

$\{(\mathsf{i} = 0) \vee (\mathsf{m} \ \% \ \mathsf{i} = 0)\}$ (consequence rule; strengthen post-condition)

To strengthen the final post-condition, we do a case-split over $i \leq 1 \vee done$:

- $i = 0$: ok, because it implies $(i = 0)$
- $i = 1$, then $m\%i = 0$, which implies the post-condition
- *done*, then $(m\%i == 0)$ is implied, which also implies the post-condition

Task 3 (5 points): Download the the C Bounded Model Checker (CBMC) from `http://www.cprover.org/cbmc/`[1] and familiarize yourself with the tool using the manual you can find on the same web-page. Use CBMC to detect the heartbleed bug (which we discussed in the lecture) in the simplified code below, and explain how you used the tool to detect the bug:

- Which unwinding depth was required?
- Which command-line parameters did you have to specify?
- Which property was violated (as reported by CBMC)?

Please indicate *which version* of `Cbmc` you have used to obtain your results, since different versions of the tool will require a different unwinding depth!

- 46 (for version 5.6 of CBMC; newer versions only require an unwinding depth of 1)
- `--unwind 46 --pointer-check --bounds-check heartbleed.c`
- `dereference failure: object bounds in src[i]: FAILURE`
  (for version 5.6 of CBMC, newer versions report a violation of the pre-condition of `memcpy`)

[1] Ubuntu users can simply install the `cbmc` package using `apt`.

Task 4 (5 points): Use the KLEE symbolic simulator (using the Docker image from `klee.github.io` as explained in the lecture) to test the following implementation of Euclid's algorithm:

```
1  unsigned gcd (unsigned x, unsigned y)
2  {
3    unsigned m, k;
4    if (x > y) {
5      k = x;
6      m = y;
7    }
8    else {
9      k = y;
10     m = x;
11   }
12
13   while (m != 0) {
14     unsigned r = k % m;
15     k = m; m = r;
16   }
17   return k;
18 }
```

Use KLEE to generate test inputs from the following *specification*:

```
1  #define MIN(x, y) ((x)<(y))?(x):(y)
2  #define MAX(x, y) ((x)<(y))?(y):(x)
3  #define IS_CD(r, x, y) (((x)%(r)==0)&&((y)%(r)==0))
4
5  unsigned gcd (unsigned x, unsigned y)
6  {
7    for (unsigned t = MIN (x,y); t>0; t--) {
8      if (IS_CD(t, x, y))
9        return t;
10   }
11   return MAX(x, y);
12 }
```

(The source code of both implementations can be downloaded from TISS.)

- How many test cases are required *at least* to achieve branch coverage for the implementation? **2**
- Provide a *minimal* number of test cases generated with KLEE such that branch coverage for the implementation is achieved!

x	y	gcd(x,y)
2	1	1
2	3	1

- If a given test suite achieves branch coverage for the specification, does the same test suite also achieve branch coverage for the implementation ...
  - for this specific example?
    No, the test inputs `x`=0, `y`=0 and `x`=2, `y`=3 cover all branches of the specification, but not all branches of the implementation.
  - in general?
    No, already doesn't hold for the specific example above.