VU Programm- und Systemverifikation
Homework: Propositional Logic, Hoare Logic, and Test Case Generation

Due date: May 13, 2021, 1pm

Task 1 (6 points): Prove the Hoare Triple below (assume that the domain of the variables `t`, `m`, `n` are the integers ($\mathbb{Z}$). 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.
No points for assertions not derived by using one of the rules from the lecture!

{true}
{true} (conditional rule)
`if (m > n) {`
  {$(n < m$} (strengthen pre-condition)
  {$(n \leq m$} (pre-condition of assignment rule)
  `int t = n;`
  {$(t \leq m$} (assignment rule)
  `n = m;`
  {$(t \leq n$} (assignment rule)
  `m = t;`
  {$(m \leq n$} (post-condition conditional rule)
`} else {`
  {$(m \leq n$} (skip)
  `skip;`
  {$(m \leq n$} (post-condition conditional rule)
}
{$(m \leq n$} (while rule/invariant)
`while (m < n) {`
  {$(m < n$} (strengthen pre-condition)
  {$(m + 1 \leq n$} (assignment rule)
  `m = m + 1;`
  {$(m \leq n$} (invariant)
}
{$(m \geq n) \wedge (m \leq n$} (post-condition loop rule)
{$(\mathtt{m} = \mathtt{n})$} (consequence rule; weaken post-condition)

Task 2 (5 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., $\texttt{i}, \texttt{n} \in \mathbb{N} \cup \{0\}$, 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.

$\{\texttt{n} \geq 1\}$

$\{((0 < 1) \wedge (1 \leq n)\}$ (assignment)

```
i = 1;
```

$\{((0 < i) \wedge (1 \leq n)\}$ (loop rule)

```
while (i % n != 0) {
```

  $\{(i \bmod n \neq 0) \wedge (0 < i) \wedge (i \leq n)\}$ (consequence rule; strengthen pre-condition*)

  $\{(0 \leq i) \wedge (i \leq n - 1)\}$ (consequence rule; rewrite assignment pre-condition)

  $\{(0 < i + 1) \wedge (i + 1 \leq n)\}$ (assignment)

```
  i = i + 1;
```

  $\{(0 < i) \wedge (i \leq n)\}$ (invariant)

}

$\{(i \bmod n = 0) \wedge (0 < i) \wedge (i <= n)\}$ (loop rule)

$\{i = n\}$

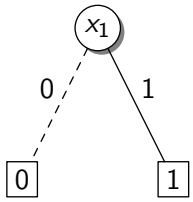
* Reasoning for strengthening of pre-condition:
  $(i \bmod n) \neq 0$ implies $i \neq 0$ and $i \neq n$
  $(i < i)$ implies $(0 \leq i)$
  $(i \neq n) \wedge (i \leq n)$ implies $(i \leq n - 1)$

Task 3 (2 points): Construct an Ordered Binary Decision Diagram that encodes the following Boolean formula:

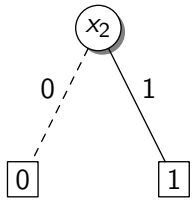
$$(x_1 \Leftrightarrow x_2) \wedge (x_1) \tag{1}$$

Ilustrate the construction steps and not just the final result!

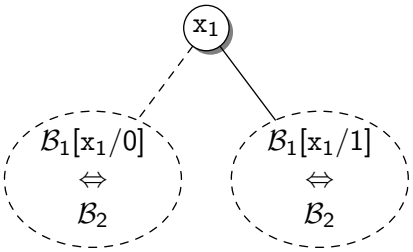
BDD $\mathcal{B}_1$ for x_1:



BDD $\mathcal{B}_2$ for x_2:

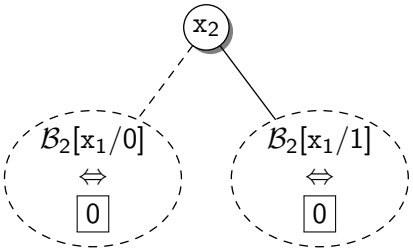


Construction of $\mathcal{B}_1 \Leftrightarrow \mathcal{B}_2$:



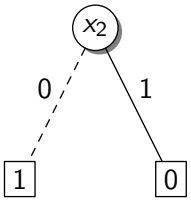
$\mathcal{B}_1[x_1/0]$: $\boxed{0}$ $\mathcal{B}_1[x_1/1]$: $\boxed{1}$

Construction of $\mathcal{B}_1[x_1/0] \Leftrightarrow \mathcal{B}_2$:

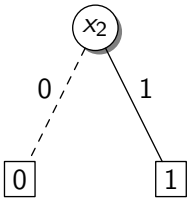


$\mathcal{B}_2[x_1/0]$: $\boxed{0}$ $\mathcal{B}_2[x_1/1]$: $\boxed{1}$

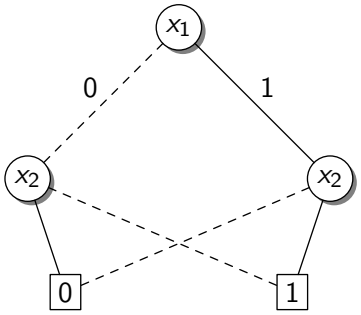
Hence $\mathcal{B}_1[x_1/0] \Leftrightarrow \mathcal{B}_2$ is:



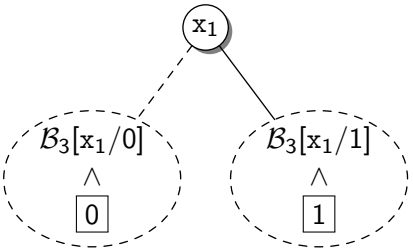
Similarly, $\mathcal{B}_1[x_1/1] \Leftrightarrow \mathcal{B}_2$:



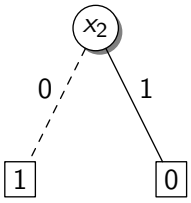
So we obtain BDD $\mathcal{B}_3$ for $\mathcal{B}_1 \Leftrightarrow B_2$:



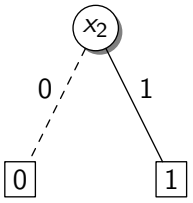
Construction of $\mathcal{B}_3 \wedge \mathcal{B}_1$:



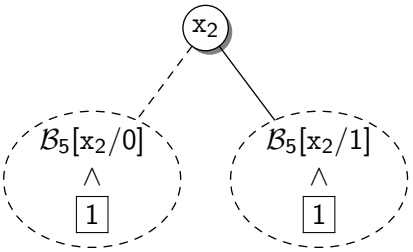
$\mathcal{B}_4$ is $\mathcal{B}_3[x_1/0]$:



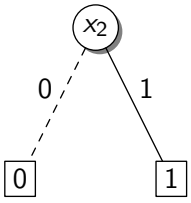
$\mathcal{B}_5$ is $\mathcal{B}_3[x_1/1]$:



Construction of $\mathcal{B}_5 \wedge \boxed{1}$:

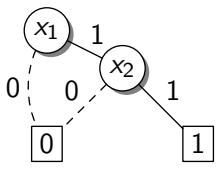


$\mathcal{B}_5 \wedge \boxed{1}$:



Similarly, $\mathcal{B}_4 \wedge \boxed{0}$:

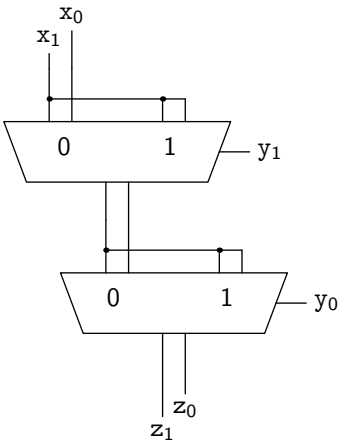
Thus, $\mathcal{B}_4 \wedge \boxed{0}$ simplifies to $\boxed{0}$. We obtain the final BDD:



Task 4 (2 points): Assume that `x` and `y` are bit-vectors of width 2. Provide an encoding of the right arithmetic shift operation `x >> y` (i.e., the most significant bit indicating the sign is replicated to fill in all vacant positions).

Use a similar circuit-based illustration as on the slides for the lecture on propositional logic.

Let's construct this systematically (like in the lecture) first:



Note that because `x` is just 2 bits wide, the example is trivial and the topmost multiplexer is redundant. So once we shift `x` even only by a single position ($\mathtt{y}_1 \vee \mathtt{y}_0$ is true), both outputs are going to be equal to $\mathtt{x}_1$. In all cases, the output $\mathtt{z}_1$ is going to be $\mathtt{x}_1$. Hence, we can simplify the encoding to:

$$(\mathtt{z}_1 \Leftrightarrow \mathtt{x}_1) \wedge \left(\begin{array}{lcl} ((\mathtt{y}_1 \vee \mathtt{y}_0) & \Rightarrow & (\mathtt{z}_0 \Leftrightarrow \mathtt{x}_1)) \\ & \wedge & \\ ((\neg \mathtt{y}_1 \wedge \neg \mathtt{y}_0) & \Rightarrow & (\mathtt{z}_0 \Leftrightarrow \mathtt{x}_0)) \end{array} \right)$$

Task 5 (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?
  - in general?

  For both cases, explain why!

1. 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.
2. No, already doesn't hold for the specific example above.

And yes, this is the same example and solution as last year ;-) I hope you still played around with the KLEE tool.

Hint: To replay the test cases, you need to make sure that the environment variable `LD_LIBRARY_PATH` points to the directory `/home/klee/klee_build/klee/lib` containing the library `libkleeRuntest.so.1.0`!

Upload a pdf file with your solutions to TUWEL by May 13, 2021.