SAT and SMT Solving SS 2018 LVA 703048

Exercises 12

June 28, 2018

[2] $\boxed{1}$ The following code snippets swap the values of two 8-bit variables `a` and `b`:

(a)
```
char tmp = a;
a = b;
b = tmp;
```

(b)
```
a = a ^ b;
b = b ^ a;
a = a ^ b;
```

(c)
```
a = a - b;
b = b + a;
a = b - a;
```

While the obvious version (a) uses a temporary, the other versions try to avoid such an additional variable. Are all of these code snippets equivalent for all values of `a` and `b`? Use an SMT solver with bitvectors to check.

To this end, it might be convenient to perform the assignments on two variables `a'` and `b'` rather than to `a` and `b` themselves, and check afterwards whether `b == a'` and `a == b'` hold.

$\boxed{2}$ The following program computes the square root of an integer:

```
int squareRoot (int n) {
  int r = 1, q = n;
  while (r + 1 < q ) {
    int p = (r + q) / 2;
    if (n < p*p)
      q = p;
    else
      r = p ;
  }
  assert r * r <= n && (r+1)*(r+1) > n;
  return r ;
}
```

[2] (a) Draw the program graph.

[2] (b) Encode a transition relation T on states (pc, n, p, q, r).

[2] (c) Apply bounded model checking to check the assertion. Are there values for which it does not hold? (You can for instance check this by modifying `verification.py` accordingly.)

$\boxed{3}$ Bit twiddling hacks are popular in low-level programming and compiler optimizations to gain efficiency. Use SMT encodings with bit vectors to check whether the following ones are correct (using some arbitrary bitwidth, if not specified otherwise).

[2] (a) The combination of two bitvectors `a` and `b` according to a bit mask is `(a & ~mask) | (b & mask)`, computed in the obvious way. A faster way is `a ^ ((a ^ b) & mask)`, as it has one operation less. Is this equivalent? (Like in C, `~` is bitwise negation, `&` is bitwise and, `|` is bitwise or, and `^` is bitwise xor.)

[2] ⋆ (b) Division and modulus is comparatively expensive on many machines. It is proposed to substitute the modulus operation `n % d` by `n & (d - 1)` for a divisor `d` that is a power of 2 (so `d = 1 << s` for some `s` smaller than the bitwidth). Is this equivalent?

[3] ⋆ (c) The bits in a byte can be reversed in the following way with a loop:

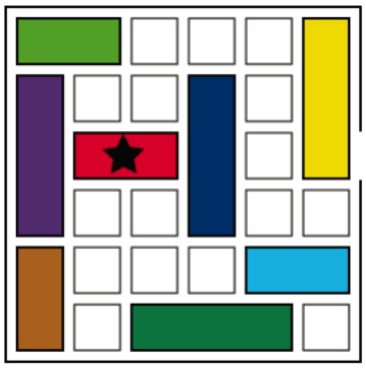
```
unsigned char b; // reverse this (8-bit) byte
unsigned char r = b; // r will be reversed bits of b; first get LSB of b
int s = 7;
for (b >>= 1; b; b >>= 1) {
  r = r << 1;
  r = r | (b & 1);
  s--;
}
r = r << s; // shift when b's highest bits were zero
```

But it can also be done in only three operations! Using larger bit vectors, though:

```
unsigned char b; // reverse this (8-bit) byte
unsigned char r; // r will be reversed bits of b
r = (b * 0x0202020202ULL & 0x010884422010ULL) % 1023;
```

Here `ULL` indicates a 64-bit value.
(Knowing that `b` has only 8 bits and is shifted initially, we can bound the number of loop iterations by 7.)

[4] ⋆ $\boxed{4}$ In the *Rush Hour* puzzle the goal is to drive the red car out of a jammed parking lot, like in the following picture:



The playing board (i.e., the parking lot) is always a 6 × 6 grid. A *move* in the game consists of moving one car back or forth onto empty fields (but the cars cannot turn). You can also play online.

Solve the above rush hour puzzle using an SMT encoding and bounded model checking, i.e., by limiting the number of steps.

The following approach might be helpful (though there are many possibilities for an encoding):

- The state, i.e., the current configuration can be described by the x-positions of all horizontal and the y-positions of all vertical cars. So in the example, a state is of the form

$$s = \langle x_{\text{lgreen}},\ x_{\text{red}},\ x_{\text{lblue}},\ x_{\text{dgreen}}, y_{\text{brown}},\ y_{\text{purple}},\ y_{\text{dblue}},\ y_{\text{yellow}}\rangle$$

- It is easy to define a predicate $I(s)$ describing the initial state. In the example, it has to include $y_{\text{brown}} = 1$, $x_{\text{red}} = 2$, etc. (assuming we count from bottom left).

- Define when a cell (X, Y) is *free* in a state s. A cell is free if it is not occupied by any car. For example, (X, Y) is not occupied by the red car if $\neg(x_{\text{red}} \leqslant X < x_{\text{red}} + 2 \wedge Y = 4)$ holds, because $len_{\text{red}} = 2$ and 4 is the (constant) y-position of the red car.
- Using the encoding of a free cell, one can define a predicate $T(s, s')$ to describe a valid transition from state s to state s': In a valid transition every car was moved in a valid way. E.g. for the red car, this can be described as fgollows: if x_{red} in s is different from x'_{red} in s' then the difference cell(s) must have been free in s. (This approach also allows parallel moves, but this is ok.)
- Finally, a success predicate $S(s)$ demanding $x_{\text{red}} = 5$ states that the red car can exit.
- Suppose there are $k + 1$ states $s_0, \ldots, s_k$. There is a solution in at most k steps iff

  $$I(s_0) \wedge \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} S(s_i)$$

  is satisfiable.
- One has to make sure that all states are *valid* in the sense that all its variables are between 1 and $6 - len_{\text{color}}$, where len_{color} is the length of the respective car.

Exercises marked with a $\star$ are optional. Solving them gives bonus points if you submit them before the course via OLAT or email.