ESCOLA
POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA

SOFTWARE AND HARDWARE VALIDATION SYSTEMS

Practical case 1: Verification of programs with Hoare logic and symbolic execution

Joel Aumedes Serrano - 48051307Y
Marc Cervera Rosell - 47980320C

Academic course 2021 - 22

Bachelor's degree in computer engineering

Contents

List of Figures

1 Exercise 1

```
{y >=  0 & y0 >= 0  & y = y0 & x = x0}
  m = 0;
  while (y > 0) {
    if ( y > 3 ) {  m = m + (4*x); y = y - 4;  }
     else { m = m + x; y = y - 1; }
  }
{m = x0 * y0}
```

Figure 1: Exercise 1 code

Statement:
Considering the breathtaking algorithm (see Figure 1) for the multiplication of two integer numbers you must:
1. Obtain an appropiate invariant for the loop and explain it
2. Perform the verification of the partial correctness of the algorithm.

1.1 Part 1

$$Invariant = \{ \text{x} = \text{x}_0 \ \& \ y \geq 0 \ \ \& \ \ \text{m} = \text{x}_0 * (y_0 - y) \}$$

- ♣ The value of 'x' never changes.
- ♣ 'y' starts ≥ 0, when it's '$= 0$', the loop ends.
- ♣ At each step we add "x" and we substract 1 "y". When $\text{y} = 0 \rightarrow \text{m} = = \text{x}_0 * y_0$.

1.2 Part 2

Assignment rule:
P = The same
[m := 0]
while(y > 0){ ... }
Q = The same

Loop rules:

Invariant initially valid (P $\rightarrow \mathcal{U}$ (Inv)):

{ y $\geq$ 0 & $y_0 \geq$ 0 & y = y_0 & $x = x_0$}
$\longrightarrow$
{ x = x_0 & $y \geq$ 0 & m = $x_0 * (y_0 - y)$}
$[m := 0]$
$\Downarrow$
{ y $\geq$ 0 & $y_0 \geq$ 0 & y = y_0 & $x = x_0$}
$\longrightarrow$
{ x = x_0 & $y \geq$ 0 & $0 = x_0 * (y_0 - y)$}

- ✓ x = x_0 is in P.
- ✓ y = y_0 is in P.
- ✓ (y_0 - y) = 0 because y = y_0 in P $\Rightarrow$ m = 0 = 0 * 0.

Invariant preserved ({ Inv & b } → { Inv }):

[m := 0]
{ $x = x_0$ & y ≥ 0 & m = x_0*(y_0 - y) y > 0 }

```
if (y > 3){
    m = m + (4 * x)
    y = y - 4
} else {
    m = m + x
    y = y - 1
}
}
```

Figure 2: Loop code

Quick remember:
Inv = { $x = x_0$ & y ≥ 0 & $m = x_0 * (y_0 - y)$ }

1.2.1 Conditional rule → True side

{ $x = x_0$ & y ≥ 0 & m = x_0*(y_0 - y) y > 0 & y > 3 }
[m := 0 || m := m + (4 * x) || y := y - 4] (More than one assignment rule)
{ $x = x_0$ & y ≥ 0 & $m = x_0 * (y_0 - y)$ }

Exit rule ($P \rightarrow \mathcal{U}$ (Q)):
{ $x = x_0$ & y ≥ 0 & m = x_0*(y_0 - y) y > 0 & y > 3 }
⟶
{ $x = x_0$ & y - 4 ≥ 0 & 0 + 4 * x = $x_0 * (y_0 - (y - 4))$ }

- ✓ x = x_0 is in P.
- ✓ y = y > 3 + y ≥ 4 → y- 4 ≥ 0.
- ✓ 4 * x = x_0 * y_0 - x_0 * y + 4 * x → 4 * x = x_0 * (y_0 - y) + 4 * x ⇒ m = 0.

1.2.2 Conditional rule → False side

{ $x = x_0$ & y ≥ 0 & m = x_0*(y$_0$ - y) y > 0 & !(y > 3) }
[m := 0 || m := m + x || y := y - 1] (More than one assignment rule)

Exit rule ($P \rightarrow \mathcal{U}$ (Q)):
{ $x = x_0$ & y ≥ 0 & m = x_0*(y$_0$ - y) y > 0 & !(y > 3) }
$\longrightarrow$
{ $x = x_0$ & $y - 1 \geq 0$ & 0 + x = $x = x_0$ * (y$_0$ - y + 1) }

- ✓ x = x_0 is in P.
- ✓ y ≥ 0 is in P and y ≥ 1.
- ✓ 0 + x = $x = x_0$ * (y$_0$ - y + 1) $\rightarrow$ 0 + x_0 = x_0 * (y$_0$ - y) + x_0 $\Rightarrow$ m = 0.

Use case ({ Inv & !b } $\rightarrow$ { Q }):

{ $x = x_0$ & y ≥ 0 & m = x_0*(y$_0$ - y) !(y > 0) }
[m := 0] (Assignment rule)
{ m = x_0 * y_0 }

Exit rule:
{ $x = x_0$ & y ≥ 0 & m = x_0*(y$_0$ - y) !(y > 0) }
$\longrightarrow$
[m := 0]
{ m = x_0 * y_0 }
$\Downarrow$
{ $x = x_0$ & y ≥ 0 & m = x_0*(y$_0$ - y) !(y > 0) }
$\longrightarrow$
{ m = x_0 * y_0 }

- ✓ (!(y > 0) & y ≥ 0) $\rightarrow$ y = 0 $\Rightarrow$ m = x_0 * (y$_0$ - 0) = x_0 * y_0

2 Exercise 2

$$r = ((\sum_{i=1}^{N} a[i])\ (mod\ 2))$$

Figure 3: Exercise 2 formula

```
{ N >= 1 }

  [i := 1  ||  r := 0]
    while (i <= n) {
       r = (r + a[i]) (mod 2);
       i = i + 1;
    }

{ ((∑_{i=1}^{N} a[i])  (mod  2)) = r }
```

Figure 4: Exercise 2 code

Statement:
Verify the algorithm of the Figure 4 for computing the parity of the sum of the elements of an array a[N] with N ≥ 0 and storing it in the result variable r. You must:
1. Obtain an appropriate invariant for the loop and explain it.
2. Perform the verification of the partial correctness of the algorithm.

2.1 Part 1

To perform the validation, we're going to use the following invariant formula:

$$Invariant = \{\ [(((\text{a[i]} + \text{a[i} + 1]) \ \%\ 2 = 0) \longrightarrow (r = 0 \wedge (a[i] + a[i+1] = 2n))) \vee (((a[i] + a[i+1])\%2 > 0) \longrightarrow (r > 0 \vee (a[i] + a[i+1]) = (2n+1)))] \wedge ((summation(i, n, a))\%2 = r)\}$$

- ♣ The first logical disjunction, is a tautology, therefore, it always will be true. It's a tautology , because a number, always, is odd or even. That is, the parity of a number is a binary question. Either it's odd or it's even.
- ♣ If we go one step further into the tautology, we will see that the first part (left part of the disjunction), represents the even numbers because, we can see that when we apply the %2, the result is 0 and r i 0 too.
- ♣ The right part of the disjunction represents the odd numbers.

- ♣ The predicate *summation(i, n, a)*, represents the summation of the array elements. Finally, once the summation is done, we apply the modulus and we match this with r.
  - The parameter i of the predicate, is the lower limit.
  - The parameter n of the predicate is the higher limit.
  - The parameter a of the predicate, is the array.

2.2 Part 2

To simplify the document writing, we're going to apply directly the multiple assignment and exit rules. Anyway we will specify where are applied.

As we can see in Figure 4 the assignment rule is already applied at the beginning (before the loop), hence we are able to start with the loop rules and the partial verification.

Loop rules:

Invariant initially valid (P $\rightarrow \mathcal{U}$ (Inv)):

{ n ≥ 1}
$\longrightarrow$
{ $[(((a[1] + a[2])\%2 = 0) \longrightarrow (0 = 0 \wedge (a[1] + a[2] = 2n))) \vee (((a[1] + a[2])\%2 > 0) \longrightarrow (0 > 0 \wedge (a[1] + a[2]) = (2n + 1)))] \wedge ((summation(1, n, a))\%2 = 0)$}

Initially, $r = 0$, so that means that we're talking about an even number. Hence, "(a[1] + a[2] = 2n" is true. Beside, if we take a look after applying the exit rule to the invariant, it's the only logical thing since in the disjunction we can observe "0 > 0", and that's completely false. Therefore as 0 is not greater than 0, the conjunction is false. So that something implies something false ($X \rightarrow F$) becomes true, the only way is to evaluate to false the left side. Thus confirming that we've said at the beginning (we're talking about an even number). At this point we've evaluated to true the disjunction with a logical reasoning. now we've to make true the global conjunction. As we've said previously, we're talking about an even number, so the summation of the array elements among two limits module 2 has to be 0, and taking a look at the predicate clause we can confirm that. Finally if we want to express what we've in logical variables: $(True \vee False) \wedge True$.

Invariant preserved ({ Inv & b } $\rightarrow$ { Inv }):

{ [(((a[i] + a[i + 1]) % 2 = 0) $\longrightarrow (r = 0 \wedge (a[i] + a[i+1] = 2n))) \vee (((a[i] + a[i+1])\%2 > 0) \longrightarrow (r > 0 \vee (a[i] + a[i+1]) = (2n+1)))] \wedge ((summation(i, n, a))\%2 = r) \wedge (\mathrm{i} \leq n)$ }
$\longrightarrow$
{ $[(((a[i+1] + a[i+2])\%2 = 0) \longrightarrow (((r + a[i+1])\%2) = 0 \wedge (a[i+1] + a[i+2] = 2n))) \vee (((a[i+1] + a[i+2])\%2 > 0) \longrightarrow (((r + a[i+1])\%2) > 0 \vee (a[i+1] + a[i+2]) = (2n+1)))] \wedge ((summation(i+1, n, a))\%2 = (r + a[i+1])\%2)$}

We've applied the assignment rule and the exit rule at the right part of the implication. The update is the code of the inside of the loop. The update is:

$$[\ \mathrm{r} = (\mathrm{r} + \mathrm{a[i]}\ \%\ 2\ )\ \ ||\ \ \mathrm{i} = \mathrm{i} + 1\ ]$$

As we've said at first part of the exercise, the disjunction is a tautology, then it always be true. The predicate part is true too because to know the parity of a sum, we need the r of the previous sum. In other words, to know the parity of a sum "p", where "p" ¿ 1 we need the sum parity of the previous elements of the array. To watch it more graphically, see Figure 5

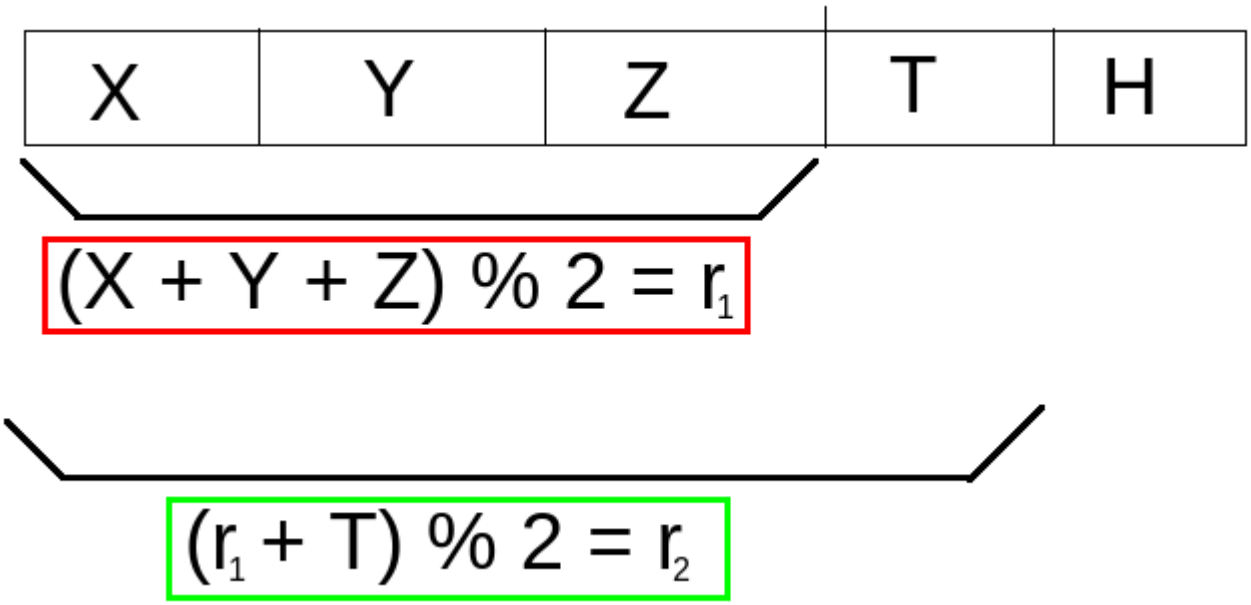

Figure 5: Graphical representation of the correctness of the predicate after a random iteration.

Use case ({ Inv & !b } $\rightarrow$ { Q }):

{ [(((a[i] + a[i + 1]) % 2 = 0) $\longrightarrow (r = 0 \wedge (a[i] + a[i+1] = 2n))) \vee (((a[i] + a[i+1])\%2 >$ 0) $\longrightarrow (r > 0 \vee (a[i] + a[i + 1]) = (2n + 1)))] \wedge ((summation(i, n, a))\%2 = r) \wedge (i > n)\}$
$\longrightarrow$
$\{((summation(i, n, a))\%2 = r)\}$

As there's no more code to execute, the invariant formula with the negated iteration condition implies the post-condition (without applying any more rule) and as we can find the post-condition above, the implication is true.