Software Verification Homework 5

Yosef Goren, Andrew Elashkin

January 15, 2023

Part I
Dry

1

1.1

$$R_{healthy}(u,v) = R(u,v) \wedge \neg R_{ill}(u,v)$$

This means a healthy transition is a transition and it is also not an ill transition.

1.2

In this question we assume $n = 2^N$ for some $N \in \mathbb{N}$.
First we define a formula for checking if a path $v_1, ..., v_k$ respects R_{ill}:

$$HP(v_1, ..., v_k) = \bigwedge_{i\in[k]} \bigwedge_{j\in[k]\setminus\{i\}} ((v_i = v_j \wedge v_{i+1} = v_{j+1}) \to \neg R_{ill}(v_i, v_{i+1}))$$

Now checking if a path $v_1, ..., v_k$ is a possible, respects R_{ill} and satisfies Gp:

$$F(v_1, ..., v_k) = I(v_0) \wedge HP(v_1, ..., v_k) \wedge \bigwedge_{i\in[k]} p(v_i)$$

Given this formula the algorithm is:

```
for $k \in [n+1]$ : do
    $\phi := F(v_1, v_2, ..., v_k) \wedge \left(\bigvee_{i\in[n-1]} \left(v_k = v_i \wedge \bigwedge_{j=i}^{k-1} R_{healthy}(v_j, v_{j+1})\right)\right)$
    if $SAT(\phi)$ then
        return $true$
    end if
end for
return $false$
```

If a path is possible, exists a subpath of it that is no longer than n vertecies - hence the algorithm will find said path. The condition appended to F verifies that a path is not only possible, repects R_{ill} and satisfies Gp - but it also ends with a loop of healthy transitions meaning it can be extended into an infinite path with the desired properties.

2

The requirements from a path to satisfy these requirements are:

1. The path starts at an initial state.
2. The transitions within the path are legal.
3. The path ends with a loop (thus can be extended to an infinite path with the same local properties).
4. For each vertex there is a son vertex that satisfies p.
5. For each vertex there is a son vertex that satisfies q.

Denote the set of vertecies in the graph with $\{u_i\}_{i\in[n]}$.
Now we provide a formula for each requirement:

1. $I(v_0)$
2. $\bigwedge_{i\in[k-1]} R(v_i, v_{i+1})$
3. $\bigvee_{i\in[n-1]} v_k = v_i$
4. $\bigvee_{i\in[k-1]}(\bigvee_{j\in[n]} R(v_i, v_j) \wedge p(v_j))$
5. $\bigvee_{i\in[k-1]}(\bigvee_{j\in[n]} R(v_i, v_j) \wedge q(v_j))$

So the formula for the path is:

$$F(v_1, ..., v_k) = I(v_0) \wedge \left(\bigwedge_{i\in[k-1]} R(v_i, v_{i+1})\right)$$

$$\wedge \left(\bigvee_{i\in[n-1]} (v_k = v_i)\right) \wedge \left(\bigvee_{i\in[k-1]} (\bigvee_{j\in[n]} R(v_i, v_j) \wedge p(v_j))\right) \wedge \left(\bigvee_{i\in[k-1]} (\bigvee_{j\in[n]} R(v_i, v_j) \wedge q(v_j))\right)$$

Now the algorithm is:

```
for $k \in [n+1]$ : do
    $\phi := F(v_1, v_2, ..., v_k)$
    if $SAT(\phi)$ then
        return $true$
    end if
end for
return $false$
```

Part II
Wet

1

The system is meant to verify a username and passowrd passed as command line arguments. The first parameter is a username and the second one is a password; both the username and password must be integers.

2

The changes to the code have been:

- Changed username to hardcoded ID.
- Changed pasword to `nondet_int()`.
- Asserted that the verification failed (`assert(!verification_success)`).

Also, when running the program we have passed `--trace` which requires it to provide us with a counterexample in the cases where the assertion fails. The assertion failing means that the verification succeded - hence a counterexample would include an assignment into the password variable which results with a successful verification meaning we have a correct password.

3

To produce this password we have added an `assume(password != 16)` (which is what we got before for the same username), which caused the program to produce an additional counterexample: 42.
To prove that this works for all usernames, we have set:

- `username=nondet_int();`
- `password = 42;`

- `assert(verification_success);`,

So we are essentially asing cbmc to prove that for whatever username, with password 42 - verification is successful (and it did).

4

4.1

The run finishes with `predictable.c` but goes into an infinite loop with `binsearch.c`.

4.2

If we add `--unwind 1`, the verification is successful for `binsearch.c`. `--undwind` sets a limit for how deep the verification should go into loops by converting any loops into a series of nested if statements.
`--no-unwinding-assertions` means that if the depth threashold is passed - it is not viewed as a failiure (as opposed to assuming the program should not iterate for more than the given number of times).
The verification success means that no illegal memory accesses are possible within parts of the program that aren't deeper in a loop than 1 iteration. (and we know nothing about any part of the program that is deeper than one iteration into a loop).

4.3

With `--unwind 2` the verification fails for `binsearch.c`.
This means that the programm can access memory illegally. Upon further analysis, this is due to char overflow in the index variable which results in neagtive indexation to the array:

```
Violated property:
    file binsearch.c line 12 function binsearch
    array ‘a’ lower bound in a[(signed long int)middle]
    (signed long int)middle >= 0l
```