Prof. Dr. Andreas Podelski
Tanja Schindler

Hand in until January 9th, 2019
15:59 via the post boxes
Discussion: January 14th/15th, 2019

Tutorial for Cyber-Physical Systems - Discrete Models
Exercise Sheet 9

The purpose of this exercise sheet is to familiarize you with algorithms for invariant checking.

Exercise 1: Invariant checking I

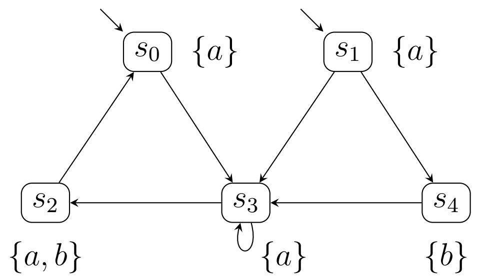
In the lecture, you have seen an algorithm for invariant checking by forward depth-first search. We display this algorithm below.

Algorithm 1: DFS-based invariant checking

```
input  : a finite transition system TS and a propositional formula Φ
output: "yes" if TS ⊨ "always Φ", otherwise "no" and a counterexample
R := ∅;                                     // set of states (''Reachable'')
U := ε;                                     // stack of states (''Unfinished'')
forall s ∈ I do
    if DFS(s, Φ) then
        return("no", reverse(U));           // path from s to error state
    end
end
return("yes");                              // TS ⊨ ''always Φ''

function DFS(s, Φ)
    push(s, U);
    if s ∉ R then
        R := R ∪ {s};                       // mark s as reachable
        if s ⊭ Φ then
            return("true");                 // s is an error state
        else
            forall s' ∈ Post(s) do
                if DFS(s', Φ) then
                    return("true");         // s' lies on a path to an error state
                end
            end
        end
    end
    pop(U);
    return("false");
end
```

Apply this algorithm to the following transition system whose set of atomic propositions is $AP = \{a, b\}$. The invariant Φ to be checked is the propositional logical formula a. Whenever you iterate over a set of states, always take state s_i before state s_j if i is smaller than j.
Present the execution of the algorithm by writing down the contents of the set R and the stack U directly before every call to the function **DFS**.



Exercise 2: Invariant checking II
The “DFS-based invariant checking” algorithm presented above (and in the lecture) always computes a minimal counterexample (minimal in the sense that you cannot remove the last state). However, the algorithm does not necessarily compute a counterexample of minimal length (there might be two minimal counterexamples of different lengths). What is an example that shows that the counterexample that is returned does not always have minimal length?

For this purpose, provide the following.

- A transition system that has three states s_0, s_1, s_2.
- An invariant.
- The (non-minimal) counterexample that is computed by the algorithm that uses the following strategy for iterating over a set of states: always take state s_i before state s_j if i is smaller than j.
- A minimal counterexample.

Exercise 3: Invariant checking III
Give an algorithm (in pseudocode, analogously to the algorithm presented above or in the lecture) for invariant checking such that, in case the invariant is refuted, a counterexample of minimal length is provided as an error indication.
The algorithm should terminate for all finite transition systems.

Hint: You may modify the algorithm presented above (or in the lecture) appropriately. You may also want to use two data structures: A *queue* and a *map*.
A queue is a list with two operations:

- `void add(Element)` adds a new element at the end.
- `Element remove()` removes the element at the front (FIFO principle).

A map behaves like a partial function. That is, it stores a value for a given key. It has the following operations:

- `void add(Key, Value)` adds a new mapping from a key to a value.
- `Value get(Key)` returns the value for the given key.
- `boolean has(Key)` returns `true` iff the map stores a value for the given key.

You can use the map to store a predecessor state for a given state. This can be helpful for constructing the counterexample in the end.