Abstraction of REVERSE by TLV

In this assignment we consider proof of various properties of program REVERSE, using abstraction via TLV.

The unrestricted version of the program is given by:

\[ y := \text{null}; \quad \text{while } x \neq \text{null} \text{ do } (x, y, x.\text{Next}) := (x.\text{Next}, x, y) \]

In Fig. 1, we present a TLV version of this program in which we restrict the heap to be of size 4. This figure actually contains the definition of REVERSE as a concrete system and a place holder, called AS, which defines the abstract variables xnull, ynull, rxt, and ryt.

In Figures 2 and 3, we present the pf file for performing the abstraction and model checking the abstract system for the property that if (the node pointed to by) \( t \) is included in list \( x \) before execution of the program, then \( t \) will be included in list \( y \) when (and if) the program terminates. Both of these files are available on the course’s home page. Running these two files on TLV confirms that program REVERSE transfers all elements which are originally placed in \( x \) into list \( y \).

1 Verifying Disjointness of \( x \) and \( y \)

The next property we would like to verify is that throughout the computation, lists \( x \) and \( y \) never share a common element. The formal specification of this property is the assertion \( \neg (\text{reach}(x, t) \land \text{reach}(y, t)) \) where, \( t \), as before, represents a generic element. Model checking this property on reverse4 yields the following counter example:

\[
\begin{align*}
\text{--- State no. 1} & = \\
& x = 4, \quad y = 0, \quad t = 4, \\
& \text{Next[1]} = 0, \quad \text{Next[2]} = 0, \quad \text{Next[3]} = 0, \quad \text{Next[4]} = 4, \\
\text{--- State no. 2} & = \\
& x = 4, \quad y = 4, \quad t = 4, \\
& \text{Next[1]} = 0, \quad \text{Next[2]} = 0, \quad \text{Next[3]} = 0, \quad \text{Next[4]} = 0,
\end{align*}
\]
As we can see, this counter example is generated by a list \( x \) which contains a node (node 4) which is connected to itself, i.e. an ill-structured list. To avoid such irrelevant cases, we strengthen the initial condition to

\[
y = \text{null} \land t \neq \text{null} \land \text{reach}(x, t) \land \text{reach}(x, \text{null})
\]

The conjunct \( \text{reach}(x, \text{null}) \) guarantees that \( x \) is a linear acyclic list.

Your task is to verify by abstraction the invariance of the “disjointness” assertion which, in abstract terms is expressible by

\[
!(rxt \land ryt)
\]

This verification should be done under the extended initial condition. To perform the abstraction you will need to extend the predicate base beyond the 4 predicates which are used in file \texttt{abs-reverse4.smv} and \texttt{abs-reverse4.pf}. In particular, it is necessary to add the predicate \( \text{reach}(x, \text{null}) \). Possibly, you will need some additional predicates.

## 2 Reversal of Order

Another property that we expect program \textsc{reverse} to satisfy is the reversal of element order. We define the predicate

\[
\text{precedes}(t_1, t_2) = \text{reach}(t_1, t_2) \land \neg \text{reach}(t_2, t_1)
\]

Thus \( t_1 \) precedes \( t_2 \) if \( t_2 \) is reachable from \( t_1 \) but \( t_1 \) is not reachable from \( t_2 \).

Your task is to verify, using abstraction, the invariance of the assertion

\[
x = \text{null} \rightarrow \text{precedes}(t_2, t_1)
\]

under the assumption that the initial condition of the system is

\[
t_1 \neq \text{null} \land t_2 \neq \text{null} \land \text{reach}(x, t_1) \land \text{reach}(x, \text{null}) \land \text{precedes}(t_1, t_2).
\]
MODULE main
DEFINE
N := 4;
x := CS.x;
y := CS.y;
t := CS.t;
xnull := AS.xnull;
rxt := AS.rxt;
ryt := AS.ryt;
tnull := AS.tnull;
VAR
CS: system conc-reverse(N);
AS: system abs-reverse;
MODULE conc-reverse(N)
DEFINE null := 0;
go := (x != null);
Next[0] := 0;
Nextx := Next[x];
VAR
x : 0..N;
y : 0..N;
t : 0..N;
Next : array 1..N of 0..N;
ASSIGN
next(x) := case
    go : Nextx;
    1 : x;
esac;
next(y) := case
    go : x;
    1 : y;
esac;
next(t) := t;
for (i=1; i<=N; i=i+1){next(Next[i]) := (go & i=x) ? y : Next[i];}
MODULE abs-reverse
VAR
xnull : boolean;
rxt : boolean;
ryt : boolean;
tnull : boolean;

Figure 1: Program abs-reverse4.smv
Func reach1(x,y);
  Return (CS.Next[x]=y) & x!=nil | (x=y);
End -- Func reach1(x,y)

Func reach2(x,y);
  Local r := 0;
  For (i in 1...N)
    Let r := r | reach1(x,i) & reach1(i,y);
  End -- For (i in 1...N)
  Return r;
End -- Func reach2(x,y)

Func reach(x,y);
  Local r := 0;
  For (i in 1...N)
    Let r := r | reach2(x,i) & reach2(i,y);
  End -- For (i in 1...N)
  Return r;
End -- Func reach(x,y)

Func abs-assert(phi);
  Return (phi & abst) forsome vars1;
End -- Func abs-assert(phi);

Func abs-trans(rho);
  Return (rho & abst & abstp) forsome all_vars1;
End -- Func abs-assert(phi);

To abs-sys;
  Let _s[2].i := abs-assert(_s[1].i);
  Let _s[2].tn := _s[1].tn;
  For (i in 1..._s[1].tn)
    Let _s[2].t[i] := abs-trans(_s[1].t[i]);
  End -- For (i in 1..._s[1].tn)
  Let _s[2].jn := _s[1].jn;
  For (i in 1..._s[1].jn)
    Let _s[2].j[i] := abs-assert(_s[1].j[i]);
  End -- For (i in 1..._s[1].jn)
End -- To abs-sys;

Func concs(st); -- Concretize abstract state "st"
  Return (st & abst) forsome vars2;
End -- Func concs(st);

Figure 2: File abs-reverse4.pf, Part A
To check_counter; -- Print concrete counter example. If exists
Let vars2 := _s[2].v;
If(exist(ce[1]))
   Print "\n Concrete counter-example follows:\n";
   Local L := length(ce); Local ic := 1;
   Let cce[1] := concs(ce[1]) & _s[1].i;
   While (ic<L)
      Local nxst := succ1(cce[ic],1) & concs(ce[ic+1]);
      If(nxst) Let ic := ic+1; Let cce[ic] := nxst;
      Else Break;
   End -- If(nxst)
End -- While (ic<L)
   Let cce[ic] := fsat(cce[ic],vars1);
Let jc := ic - 1;
   While (jc>0)
      Let cce[jc] := fsat(pred1(cce[jc+1],1) & cce[jc],vars1);
      Let jc := jc - 1;
   End -- While (jc>0)
   For (i in 1...ic)
      Print "\n---- State no. " , i," =\n", cce[i];
   End -- For (i in 1...ic)
End -- If(exist(ce))
End -- To check_counter;

-- Start main Program

Let vars1 := _s[1].v;
Let all_vars1 := set_union(vars1,prime(vars1));
Let nil := 0;
Let _s[1].i := (y=nil) & (t != nil) & reach(x,t);
Let abst := (xnull <-> (x=nil)) & (tnull <-> (t=nil)) &
   (rxt <-> (reach(x,t))) & (ryt <-> (reach(y,t)));
   Let abstp := prime(abst);
abs-sys;
Print "Check presence of t in abstract system\n";
Call Invariance(xnull -> ryt,2);
check_counter;

Figure 3: File abs-reverse4.pf, Part B