Lecture Notes (2012.01.19)

Aims: Practice on "Process-oriented architectural design"

A. Modelling examples in mCRL2

Note: files with process definitions have extension .mcrl2
files with properties have extension .mcf

A.1 Buffers

```
---------------------------------------------------------------------------------------------------------------------------
act   in1,out1,in2,out2,i1,i2,o1,o2,ao1,ao2,bi1,bi2,c1,c2:Bool;
proc  BT = sum x:Bool. (in1(x).out1(x).BT + in2(x).out2(x).BT);
        BTa = rename({in1 -> i1, in2 -> i2, out1 -> ao1, out2 -> ao2}, BT);
        BTb = rename({in1 -> bi1, in2 -> bi2, out1 -> o1, out2 -> o2}, BT);

        B = allow({i1,i2,c1,c2,o1,o2}, comm({ao1|bi1 -> c1, ao2|bi2 -> c2}, BTa || BTb));

init hide({c1,c2}, B);

---------------------------------------------------------------------------------------------------------------------------
```

```
---------------------------------------------------------------------------------------------------------------------------
A.2 Dining Philosophers

% Naive solution; 3 dining philosophers

    sort Phil = struct p1 | p2 | p3;
         Fork = struct f1 | f2 | f3;

    map lf, rf: Phil -> Fork;
    eqn  lf(p1) = f1;
         lf(p2) = f2;
         lf(p3) = f3;
         rf(p1) = f3;
         rf(p2) = f1;
         rf(p3) = f2;

    act get, put, up, down, lock, free: Phil # Fork;
        eat: Phil;

    proc P_Phil(p: Phil) = get(p,lf(p)) . get(p,rf(p)) . eat(p) .
                           put(p,lf(p)) . put(p,rf(p)) . P_Phil(p);

         P_Fork(f: Fork) = sum p:Phil. up(p,f) . down(p,f) . P_Fork(f);

    init block( { get, put, up, down },
         comm( { get|up->lock, put|down->free },
          P_Fork(f1) || P_Fork(f2) || P_Fork(f3) ||
          P_Phil(p1) || P_Phil(p2) || P_Phil(p3)
       ));

% corrected version

    proc P_Phil(p: Phil) =
      (p == p1) -> get(p,rf(p)) . get(p,lf(p)) . eat(p) .
                   put(p,rf(p)) . put(p,lf(p)) . P_Phil(p)
                <> get(p,lf(p)) . get(p,rf(p)) . eat(p) .
                   put(p,lf(p)) . put(p,rf(p)) . P_Phil(p);

% for K philosophers

    eqn K = 100;

    map K: Pos;

    act get,_get,__get,put,_put,__put: Pos#Pos;
        eat: Pos;

    proc
      Phil(n:Pos) = _get(n,n)._get(n,if(n==K,1,n+1)).eat(n)._put(n,n)._put(n,if(n==K,1,n+1)).Phil(n);
      Fork(n:Pos) = sum m:Pos.get(m,n).put(m,n).Fork(n);

      ForkPhil(n:Pos) = Fork(n) || Phil(n);
      KForkPhil(p:Pos) =
          (p>1) -> (ForkPhil(p)||KForkPhil(max(p-1,1)))<>ForkPhil(1);    % ie ForkPhil(1) || ... || ForkPhil(k);

    init allow( { __get, __put, eat },
         comm( { get|_get->__get, put|_put->__put },
          KForkPhil(K)
       ));
---------------------------------------------------------------------------------------------------------------------------
```

Note: Linearisation

LPE (linear process equation):
one single process name is used in the LHS
and there is precisely one action in front of the recursive invocation of the process variable at the RHS

Examples (states encoded in data variables)

```
L1.
proc  X = a.b.c.X

proc  Y(s:N+)  =  (s=1) --> a . Y(2)   +   (s=2) --> b . Y(3)   +  (s=3) --> c . Y(1)
        X = Y(1)

L2.
proc   X = sum(n:N)  i(n) . o(n) . X

proc    Y(n:N, b:B)  = sum(m:N)  b --> i(m) . Y(m, -b)  +  -b --> o(n) . Y(n, -b)
          Y(n,true) = X       for any n

---------------------------------------------------------------------------------------------------------------------------
```

B. Properties

<u>Exercises</u>

a. Give modal formulas for the following properties:

1. As long as no error happens, a deadlock will not occur.

2. Whenever an a can happen in any reachable state, a b action can subsequently be done unless
   a c happens cancelling the need to do the b.

3. Whenever an a action happens, it must always be possible to do a b after that,
   although doing the b can infinitely be postponed.

b. Show that the identities [R1·(R2 + R3)]ϕ = [R1·R2+R1·R3]ϕ and ⟨R1·(R2 + R3)⟩ϕ = ⟨R1·R2+R1·R3⟩ϕ hold.
This shows that regular formulas satisfy the left distribution of sequential composition over choice,
which justifies to say that regular formulas represent sequences.

Answers

a.
1. [-error*] <true>true

2. [true* . a]⟨true* . (b ∪ c)⟩true

3. [true* . a . -b*]⟨true* . b⟩true

b. [R1·(R2 + R3)]ϕ = [R1]([R2]ϕ ∧ [R3]ϕ) = [R1][R2]ϕ ∧ [R1][R3]ϕ = [R1·R2+R1·R2]ϕ.

Further examples

Compare:
mu X. X (false) and nu X. X (true)
mu X. <a> X and nu X. <a> X
wrt a a-reflexive state: only the second is valid in {s}; the first is valid in all states in {}
both {} and {s} are solutions to the equation

always p = [true*] p and eventually p = <true*>

stronger: p will eventually become valid along every path
mu X. ([true] X ∨ p) (true also for paths ending in deadlock)
vs mu X. (([true] X ∧ <true> true) ∨ p) (deadlock explicitly avoided)

mu X. [-a]X (action must unavoidably be done, provided there is no deadlock before)
mu X. ([-a]X ∧ <true>true) (action a must be done anyhow, possibility of deadlock excluded)

both are false for a system with a state with a loop labelled by "b" and a unique outgoing labelled by "a"
because "b" can infinitely be done (and "a" avoided)
A valid formula however is mu X. ([-a]X ∨ <a>true)

Trick:
An effective intuition to understand whether or not a fixed point formula holds is by thinking of it as a graph to be traversed,
where the fixed point variables are states and the modalities ⟨a⟩ and [a] are seen as transitions.
A formula is true when it can be made true by passing a finite number of times through the minimal fixed point variables,
whereas it is allowed to traverse an infinite number of times through the maximal fixed point variables.

Nesting
* conditional
* fairness:
  express that some action must happen, provided it is unboundedly often enabled,
  or because some other action happens only a bounded number of times.

mu X. nu Y.((⟨a⟩true ∧ [b]X) ∨ (¬⟨a⟩true ∧ [b]Y))

A state without a-transitions must infinitely often be enabled.
Because the X is preceded by a minimal fixed point, the X can only be finitely often 'traversed'.
Within that the variable Y, can be traversed infinitely often, as it is preceded by a maximal fixed point.

nu X. mu Y.((⟨a⟩true ∧ [b]X) ∨ (¬⟨a⟩true ∧ [b]Y))
on each sequence states reachable via "b" actions, only finite substretches of states can not have an
outgoing "a" transition. So, typically, if "a" is enabled in every state.

mu X . nu Y . [−a]Y ∧ [a]X
(action a occurs an finite number of times)

nuX . muY . [−a]Y ∧ [−]X ∧ ⟨−⟩true
(action a occurs an infinite number of times)

Encodings:
<R*> p = mu X. <R> X ∨ p
[R*] p = nu X. [R] X ∧ p
<R+> p = <R> <R*> p
<R+> p = [R] [R*] p

Exercises

a. Consider the formulas

ϕ1 = mu X . [a]X
ϕ2 = nu X . [a]X

If possible, give transition systems where ϕ1 is valid in the initial state and ϕ2 is not valid and vice versa.

b. What do the following formulas express?
Is there a process that shows that these formulas are not equivalent?

mu X . nu Y. ([a]Y ∨ [b]X)
nu X . mu Y. ([a]Y ∨ [b]X)

Answers

a. mu X. [a]X in invalid in a-loop, the other is valid. As nu X. [a]X equivales true here is no transition system for which
the minimal fixed point formula holds, and the maximal one does not.

b. Each sequence consisting of only a and b actions ends in an infinite sequence of a's.
Each sequence of a and b actions contains only finite subsequences of a's.
The first formula is valid and the second is not valid for the process P defined by P = a . P

Modal formulas with data

1. Quantifiers for action formulas

Whenever an error with some number n is observed, a shutdown is inevitable:

```
[true*. exist n:N . error(n)]  mu X. ([-shutdown] X ∧ <true>true)
[true*. exist n:N . (error(n) ∩ fatal(n))]  mu X. ([-shutdown] X ∧ <true>true)
```

As long as no fatal error occurs, there will be no deadlock:

```
[(for all n:N . -(error(n) ∩ fatal(n))*]  <true>true
```

2. <u>Quantifiers over data (in the usual sense)</u>

The same value is never delivered twice can be done as follows:

```
forall n:N. [true* . deliver(n) . true* . deliver(n)] false
```

After sending a message that message can eventually be delivered with some error code n.
(the error code is irrelevant for this requirement, just a parameter of the action)

```
forall m:Mess . [true* . send(m)] <true* . exists n:N. deliver(m,n)]> true
```

3. <u>Data in the fixed point variables</u>

The property that the merger must satisfy is that as long as the input streams at r1 and r2 are ascending, the output must be ascending too.
Variables in1 , in2 and out contain the last numbers read and delivered.

It does not appear to be possible to phrase this property without using data in the fixed point variables.

```
nu X  (in1:N:=0, in2:N:=0, out:N:=0)
      forall s:N . (  [r1(s)] (s >= in1 -->  X(s, in2, out))  ∧
                      [r2(s)] (s >= in2 -->  X(in1, s, out))  ∧
                      [o(s)] (s >= out  -->  X(in1, in2, s))  )
```

<u>Exercises</u>

a. Specify a unique number generator that works properly if it does not generate the same number twice.
b. Express the property that a sorting machine only delivers sorted arrays. Arrays are represented by a function f :N → N.
c. Specify that a store with products of sort Prod is guaranteed to refresh each product. The only way to see this, is that the difference in the number of enter(p) and leave(p) is always guaranteed to become zero within a finite number of steps.

<u>Answers</u>

```
a. forall n:N . [true* . gen(n) . true* . gen(n)] false

b. forall f: N-->N . [true* . deliver(f)] (forall n:N . f(n+1) > f(n))

c. forall x:Product . nu X(n:N:=0) . [-(enter(x) union leave(x))] X(n)  ∧  [enter(x)] X(n+1)  ∧ [leave(x)] X(n-1)
                                     ∧
                                       mu Y(x:Product, n:N) . (n=0) ∨  [-(enter(x) union leave(x))] Y(x,n)  ∧  [enter(x)] Y(x,n+1)  ∧ [leave(x)] Y(x,n-1)
```

C. **More examples (mCRL2)**

C1 Road&Rail

```
act     car,train,ccross,tcross,
        wait_up,set_up,up,wait_dw,set_dw,dw,
        wait_green,set_green,green,wait_red,set_red,red;
proc    Road = car . wait_up . ccross . set_dw . Road;
        Rail = train . wait_green . tcross . set_red . Rail;
        Signal = set_green.wait_red.Signal + set_up.wait_dw.Signal;

  RR = hide({up,dw,green,red},
            comm({wait_up|set_up -> up, wait_dw|set_dw -> dw,
                  wait_green|set_green -> green, wait_red|set_red -> red},
                 Road || Rail || Signal));

init RR;
```

----

need to "unhide":

```
liveness:       [true*] <true*.up> true
starvation:     [true*] nu X. <!up> X
safety:         [true*.green.(!red)*.up] false
```

C2 Philosophers

* No deadlock (every philosopher holds a left fork and waits for a right fork (or vice versa):

```
[true*]<true>true
```

* No starvation (a philosopher cannot acquire 2 forks):

```
forall p:Phil. [true*.!eat(p)*] <!eat(p)*.eat(p)>true
```

* A philosopher can only eat for a finite consecutive amount of time:

```
forall p:Phil. nu X. mu Y. [eat(p)]Y && [!eat(p)]X
```

* there is no starvation: for all reachable states it should be possible to eventually perform an eat(p) for each possible value of p:Phil.

```
[true*](forall p:Phil.  mu Y. ([!eat(p)]Y && <true>true))
```

D. **Case study: movable patient support unit (from [Groote&Reniers, 2010] book)**

(in annex)