Assignment 1: Crossing the river – in mCRL2

Luís Barbosa & José Proença

Arquitectura e Cálculo – 2017/2018

To do: Write a report using LaTeX including the answers to the exercises below.

To submit: The report in PDF by email to lsb@di.uminho.pt.

Deadline: 23 March 2018 @ 23h59 (Friday)

Modelling the farmer-fox-goose-beans problem

Exercise 1. Recall last semester's specification of the farmer-fox-goose-beans problem using SMV.

```
%%%%  SMV specification.  %%%%
VAR
        farmer : boolean;
        fox    : boolean;
        goose  : boolean;
        beans  : boolean;
IVAR
        thing : {nothing,f,g,b};
ASSIGN
        init(farmer) := FALSE;
        init(fox)    := FALSE;
        init(goose)  := FALSE;
        init(beans)  := FALSE;
        next(farmer) := !farmer;
        next(fox)    := thing = f  ?  !fox   : fox;
        next(goose)  := thing = g  ?  !goose : goose;
        next(beans)  := thing = b  ?  !beans : beans;
TRANS
        thing = f  →  farmer = fox   &
        thing = g  →  farmer = goose &
        thing = b  →  farmer = beans
DEFINE
        safe := (fox = goose → fox = farmer) & (goose = beans → beans = farmer);
        goal := farmer & fox & goose & beans;
INVAR
        safe
LTLSPEC
        G !goal
CTLSPEC
        AG !goal
```

We will encode the same problem using mCRL2's process algebra. This algebra focus on *actions* rather than *state*, making it less optimal for this particular problem. However, it will help clarifying the key differences between a state-based approach and an action-based approach to model. We start with a simplified (but incomplete) version `farmer1.mcrl2` below.

```
%% file: famer1.mcrl2
act
  fr,fl,gr,gl,br,bl,                          % actions by the passengers
  ffr,fgr,fbr,farmerr,ffl,fgl,fbl,farmerl,    % actions by the farmer
  foxr,foxl,gooser,goosel,beansr,beansl,      % actions by the system
  winf,wing,winb,win;                         % actions to detect winning conditions

proc
  Fox = fr.(fl+winf).Fox ;
  Goose = gr.(gl+wing).Goose ;
  Beans = br.(bl+winb).Beans ;
  Farmer = (ffr+fgr+fbr+farmerr).(ffl+fgl+fbl+farmerl).Farmer ;

init
  allow(
    { foxr,foxl,gooser,goosel,beansr,beansl,farmerl,farmerr,win },
  comm(
    { fr|ffr -> foxr,   fl|ffl -> foxl,
      gr|fgr -> gooser, gl|fgl -> goosel,
      br|fbr -> beansr, bl|fbl -> beansl,
      winf|wing|winb|farmerl -> win
    },
    Fox || Goose || Beans || Farmer
  ));
```

The specification is split into three sections: ***act***, a declaration of 24 actions, ***proc***, the definition of 4 processes, and ***init***, the initialisation of the system.

1.1. Produce the labelled transition system (LTS) of this mCRL2 specification using (1) `mcrl22lps` to linearise the system and (2) `lps2lts` to produce the final LTS. Finally, visualise the resulting LTS with `ltsgraph` and **show a screenshot of the LTS (make sure it is understandable).**

1.2. This specification is not complete yet, i.e., it does not fully model the original SMV specification. **Explain informally why this specification is not complete**, by explaining what is being modelled and what is still missing.

1.3. If you replace the ***init*** block by only `Fox || Goose || Beans || Farmer` (i.e., without the restrictio0ns *allow* and *comm*) **would you obtain more or less states** than with the original specification? **Why?**

Exercise 2. We now present a new specification for the same problem consisting of a single process `State` that keeps the state information. This new specification includes more advanced features of mCRL2, including: a data structure, actions with data parameters, processes have parameters, and user defined functions **inv** and **ok**.

```
%% file: farmer2.mcrl2
sort
  Position = struct left | right;
map
  inv : Position → Position ;
  ok  : Position # Position # Position # Position → Bool ;
var
  fm,f,g,b: Position;
eqn
  inv(left)   = right ;
  inv(right)  = left ;
  ok(fm,f,g,b) = %% (1) %%;

act
  fox,goose,beans,farmer : Position; % system actions, parameterised on the position
  win;                               % actions to detect the winning condition
proc
  State(fm:Position,f:Position,g:Position,b:Position) = % (farmer,fox,goose,beans)
     ((fm==f && ok(inv(fm),inv(f),g,b)) → fox(inv(f)).State(inv(fm),inv(f),g,b))
   + ((fm==g && ok(inv(fm),f,inv(g),b)) → goose(inv(g)).State(inv(fm),f,inv(g),b))
   + ((fm==b && ok(inv(fm),f,g,inv(b))) → beans(inv(b)).State(inv(fm),f,g,inv(b)))
   + (         ok(inv(fm),f,g,b)       → farmer(inv(fm)).State(inv(fm),f,g,b))
   + ((fm==right && f==right && g==right && b==right)   → win.State(left,left,left,left));
init
  State(left,left,left,left);
```

2.1. This new specification has a hole in the definition of ok, marked with %% (1) %%. Extend the given mCRL2 definition by replacing this hole with the code that describes the desired state invariant and save the resulting specification as `farmer2.mcrl2`. **Show your new definition of the function ok.**

2.2. Without modifying the process `State`, adapt the specification by adding a new process `Counter(n:Nat)` that runs in parallel with `State(left,left,left,left)` and counts the number of traversals made by the boat. Save the resulting specification as `farmer3.mcrl2` and **show your new specification**. (hint: it could be useful to use a bound for the `Counter`), i.e., do not allow n to be bigger than a certain number.)

LTS Equivalence

Exercise 3. Recall the action-based `farmer1.mcrl2` specification from Exercise 1 and the state-based `farmer2.mcrl2` specification from Exercise 2.

3.1. Modify the initial process (*init*) of both `farmer1.mcrl2` and `farmer2.mcrl2` to hide all allowed actions except `win` (using *hide*), and save them as `farmer1-tau.mcrl2` and `farmer2-tau.mcrl2`, respectively. In `farmer2-taus.mcrl2` redefine the function ok by setting it to true, i.e., define `ok(fm,f,g,b)=true;`. **Show the resulting *init* block from each file.**

3.2. Generate the `.lts` files corresponding to `farmer1-tau.mcrl2` and `farmer2-tau.mcrl2`, and compare them using strong bisimulation using the following command. **What can you conclude?**

```
$ ltscompare --equivalence=bisim farmer1-taus.lts farmer2-taus.lts
```

3.3. Using `ltsconvert`, minimise the LTS for `farmer2-taus.mcrl2` with respect to branching bisimulation, using the command below. **Include a screenshot of the minimised LTS and explain what information can we infer from this LTS.**

```
$ ltsconvert --equivalence=branching-bisim farmer2-taus.lts
```

Verification of the farmer-fox-goose-beans problem

Exercise 4. Recall the LTSs `farmer1.lts` (Exercise 1) and `farmer2.lts` (Exercise 2 – after completing Exercise 2.1). You will now verify properties of these systems. In mCRL2, a property can be written in a text file `prop.mcf`, and verified against a system `system.mcrl2` using the following two commands.

```
$ mcrl22lps system.mcrl2 system.lps
$ lps2pbes  system.lps -f prop.mcf system.pbes
$ pbes2bool system.pbes
```

4.1. What does the property "`[true*]<ready>true`" mean? Does it hold in any of these 2 LTSs?

4.2. Does the property "`[true*.foxr.win]false`" holds for `farmer1.lts`? Does the equivalent property "`[true*.fox(right).win]false`" holds for `farmer2.lts`? What can you conclude?

4.3. Recall that `farmer1.lts` is less complete than `farmer2.lts`, because it fails to include some important invariants. Write a **single property** for `farmer2.lts` to capture that:

- no bad state is reached, and
- the goal is reached (everyone can cross).

Verify it using mCRL2 toolset. **Verify** if its equivalent formulation holds for `farmer1.lts`.

4.4. Consider now the extended system `farmer3.mcrl2` produced in Exercise 2.2. In this example there is a an extra process called `Counter(n:Nat)`. Using this extra process, **define the following two properties**:

1. It is possible to win after exactly 7 moves.
2. It is not possible to win in less than 7 moves.