Modelling and verifying behaviour in mCRL2

José Proença

System Verification – 2024/2025

To do

Produce a report as a PDF document including the answers to the exercises below. Read, modify, and produce mCRL2 specifications, following the instructions in the exercises.
Auxiliary files in https://fm-dcc.github.io/sv2425/exercises/farmer.zip

What to submit

The PDF report and the files requested in the exercises: `farmer1.mcrl2`, `farmer2.mcrl2`, `farmer3.mcrl`, and `vending.mcrl2`.

How to submit using git

1. Create a private git repository in your favouring host (e.g., github or bitbucket).
2. **Name it `SV2425-g<group number>`.**
3. Add `jose.proenca@fc.up.pt` as a member to the group (read-permissions are enough).
4. Include all the files to be submitted in the repository.

Note that **all students should push commits.**

Deadline

4 Nov 2024 @ 23:59 (Monday)

The farmer-fox-goose-beans problem

A **farmer** wants to transport a **fox**, a **goose**, and some **beans** across a river (from the **left** margin to the **right** margin). Unfortunately, he can only carry one at a time. Furthermore, if the farmer is not present, the fox will eat the goose and the goose will eat the beans. The problem is solved if the farmer can carry all animals across the river.

```
%% 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 ;

  Sys = 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
        ));

init
  Sys;
```

Exercise 1. We will encode the same problem using mCRL2's process algebra. Start by downloading the auxiliary files for this assignment at https://fm-dcc.github.io/sv2425/exercises/farmer.zip, where you will find the `farmer1.mcrl2` file above. This is a simplified (but not fully accurate) specification of our farmer-fox-goose-beans problem.

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. Create a new project `farmer1` using `mcrl2ide`, and add the resulting project folder to your git repository. Produce the labelled transition system (LTS) of this mCRL2 specification and **show a screenshot of the LTS (make sure it is understandable).**

1.2. This specification is not fully accurate yet, i.e., it does model all aspects of the puzzle. **Explain informally why our model is not accurate**, 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 restrictions `allow` and `comm`) **would you obtain more or less states** than with the original specification? **Why?**

Exercise 2. We present below a new specification for the same problem consisting of a single process `State` that keeps the state information, found in the provided auxiliary file `farmer2.mcrl2`. This new specification includes more advanced features of mCRL2, including: a *data structure*, actions with *data parameters*, processes with *data 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));

   Sys = State(left,left,left,left);

init
  Sys;
```

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 in a new project named `farmer2`. **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 in a new project `farmer3` 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.)

Exercise 3. You will now verify properties of these systems. In `mcrl2ide`, a property can be written using `Tools>Add Property`. Answer the questions below on the use of mu-calculus for specifying properties in mCRL2.

3.1. What does the property "`[true*]<win>true`" mean? Does it hold for `farmer1` and for `farmer2`?

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

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

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

Add this property to your project and **verify** it using mCRL2 toolset. **Reformulate** this property for `farmer1` and add it to that project, and **verify** if it holds.

3.4. Consider now the extended system `farmer3` from Exercise 2.2. In this example there is a an extra process called `Counter(n:Nat)`. **Define the following two properties** over actions of this counter:

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

A vending machine

Exercise 4. Specify two interacting processes in mCRL2:

- **a vending machine** with 2 products, apples and bananas, costing 1eur and 2eur respectively; and
- **a user** who can insert 1eur or 2eur coins and request for products.

Provide a specification of this system and include them in a `vending.mcrl2` file, according to the requirements below. Try to keep the specifications simple. **Submit this file in your git repository.**

Requirements:

- The user must be able to get apples and bananas;
- The machine accepts up to 3eur, and not more than that;
- The machine must give change back when applicable;
- The machine can be powered off and powered on;

Exercise 5. You will now specify two interacting processes in mCRL2:

- **a vending machine** with 2 products, apples and bananas, costing 1€ and 2€ respectively; and
- **a user** who can insert 1€ or 2€ coins and request for products.

5.1. Specify this system in a new file `vending.mcrl2` such that the properties below hold. Try to keep the specifications simple.

```
[true*.pay2eur.pay2eur] false
[true*.pay2eur.pay1eur] false
[true*.pay2eur]<(!pay1eur && !pay2eur)*.getApple> true
<true*.pay2eur.true*.getBanana> true
```

Submit this file in your git repository. Show your specification and **show a screenshot of its LTS**.

5.2. Now it is your turn to formalise the requirements. Write a set of requirements using mCRL2's formulas that capture the informal requirements stated below. Do not implement the mCRL2 model. If necessary, include an explanation of the actions and assumptions that you used.

1. *I would like the vending machine to sell 3 items: apples, bananas, and chocolates.*
2. *It should be possible to buy chocolates for 2€ and fruit for 1€.*
3. *Only 1€ and 2€ coins are accepted.*
4. *The machine has a maximum capacity for 1€ coins and for 2€ coins.*
5. *When the machine has no 1€ coins, it cannot not sell fruit with a 2€ coin.*