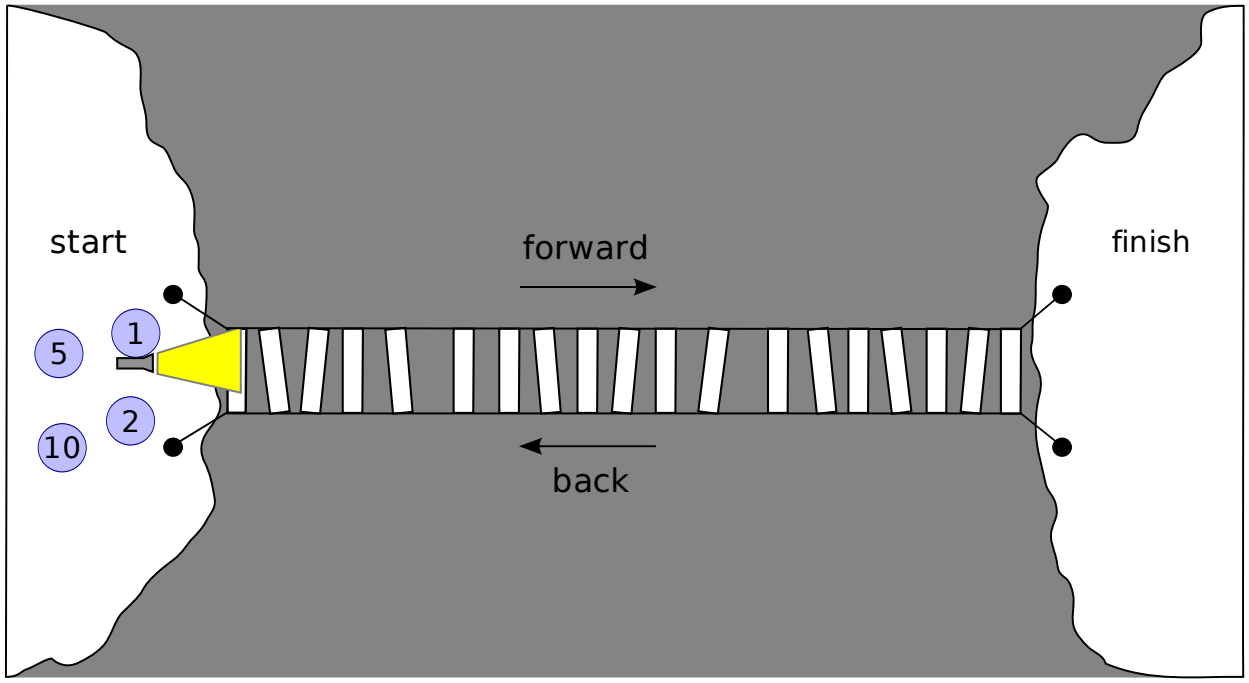
mCRL2 tutorial: The rope bridge

José Proença
Arquitectura e Cálculo – 2015/2016

February 18, 2016

(This tutorial was originally developed by Bas Ploeger in 2008 and given presented by Erik de Vink and others in 2012.)



In the middle of the night, four adventurers encounter a shabby rope bridge spanning a deep ravine. For safety reasons, they decide that no more than 2 persons should cross the bridge at the same time and that a fhashlight needs to be carried by one of them on every crossing. They have only one flashlight. The 4 adventurers are not equally skilled: crossing the bridge takes them 1, 2, 5, and 10 minutes, respectively. A pair of adventurers cross the bridge in an amount of time equal to that of the slowest of the two adventurers.

One of the adventurers quickly provlaims that they cannot get all four of them across in less than 19 minutes. However, one of her companions disagrees and claims that it can be done in 17 minutes. We shall verify this claim and show that there is no faster strategy using mCRL2.

The file `bridge-holes.mcrl2` contains an outline for a model of this situation. We will gradually fill the holes in a number of exercises.

The specification specifies the `Flashlight` process, and everything needed by it, and it gives some clues about the datatypes that are to be used.

First of all, it dictates how the position of the flashlight and the adventurers should be encoded:

```
% Data type for the position of adventurers and the flashlight.
% Initially, they are all on the 'start' side of the bridge. In the end,
% they should all have reached the 'finish' side.
sort Position = struct start | finish;
```

The following clue is given about the processes describing the adventurers:

```
% Adventurers are identified by their "speeds", i.e. the number of
% minutes they need for crossing the bridge. For this we use the basic
% integer data type of mCRL2 called 'Int'.
```

The flashlight process is given by:

```
% The Flashlight process models the flashlight:
% 1. If it is at the 'start' side, it can move forward together with any
%    pair of adventurers.
% 2. If it is at the 'finish' side, it can move back together with any
%    adventurer.
proc Flashlight(pos:Position) =
       (pos == start) →
         % Case 1.
         sum s,s':Int . forward_flashlight(s,s') . Flashlight(finish)
       <>
         % Case 2.
         sum s:Int . back_flashlight(s) . Flashlight(start);
```

Here the actions `forward_flashlight` and `back_flashlight` are described by:

```
act
    % Action declarations:
    % – 'forward'  means:  from start to finish
    % – 'back'     means:  from finish to start

    % The flashlight moves forward together with two adventurers,
    % identified by the action's parameters.
    forward_flashlight: Int # Int;

    % The flashlight moves back together with one adventurer, identified
    % by the action's parameter.
    back_flashlight: Int;
```

In the following exercises we will extend the specification towards a full description of the problem, and we will find out the minimal time needed for all adventurers to cross the bridge.

Exercise 1. Study the stub specification in `bridge-holes.mcrl2`. Then add the process definition for an adventurer. For this, answer the following quesions:

- What data parameters will the process have?
- What actions will the process be able to perform?

You will have to add action declarations and a process definition in the placed marked with *%%% TODO (Exercise 1):* in the mcrl2 specification.

Exercise 2. Add the 4 adventurers to the initial process definition. Apart from adding parallel processes to the definition, you have to take care of the synchronisation between actions of these processes:

- Declare actions for the following events:
  - 2 adventurers and a flashlight move forward over the bridge;
  - 1 adventurer and a flashlight move back over the bridge.
- For each of these actions to occur, certain actions of the separate processes have to be synchronised. Specify the synchronisation between the actions using the communication operator `comm`.
- Ensure that only the synchronised actions can occur, using the `allow` operator.

Exercise 3. Simulate the model using mCRL2 toolset by (1) executing the `mcrl22lps` to produce the linearised specification, and (2) executing `lpsxsim` to start a GUI simulation of the resulting linearised specification.

If the linearisation fails, try to fix the reported errors. To simulate the system you can execute an action by double-clicking it in the upper-left list. Notice how the state parameter values get updated in the bottom part.

If you notice any weird or incorrect behaviour in the simulator try to improve your model and simulate it again.

Exercise 4. Generate the labelled transition system (LTS) of your model by using the `lps2lts` command. Visualise the LTS using the `ltsgraph` tool. Alternatively, you can view a 3D representation of the same LTS using the `ltsview` tool.

Exercise 5. The total amount of time that the adventurers consumed so far is not yet being measured within the model. For this purpose, add a new process to the specification, called `Referee`, which:

- counts the number of minutes passed and updates this counter every time the bridge is crossed by some aventurer(s);
- reports this number when all adventurers have reached the *finish side*. (This implies that it also needs to be able to determine when this happens!)

You will have to add action declarations, and a *Referee* process definition and extend the initial process definition, including the communication and allow operators.

Hint: `max(a,b)` returns the maximum of `a` and `b`.

Verification

Exercise 6. We shall now verify the following properties using the toolset:

- A. It is possible for all adventurers to reach the finish side in 17 minutes.
- B. It is not possible for all adventurers to reach the finish side in less than 17 minutes.

Express each of these properties using mCRL2's modal calculus. Add the formulas to the files `formula_A.mcf` and `formula_B.mcf`.

Hint: the modal calculus accepts forall quantifiers over naturals.
For example, "`forall x:Nat . val(x < 17) =>` ψ" will check if ψ holds for every x smaller than 17.

Exercise 7. Verify the formulas using the toolset, with the commands `lps2pbes` and `pbes2bool`.

Exercise 8. A disadvantage of using PBES's for model checking is that insightful diagnostic information is hard to obtain. We shall now verify both properties again using the LTS tools.

Verify the properties by generating traces using the `lps2lts` command with extra options. Assuming that the action that reports the time is called `report`, execute:

```
$ lps2lts --action=report -t20 bridge.lps
```

This outputs a message every time a `report` action is encountered during state space generation. It also writes a trace to a file for the first 20 occurrences of this action. Properties A and B can now be checked by observing the output messages. Moreover, the trace for property A can be printed by passing the corresponding trace file name as an argument to the `tracepp` command, *e.g.*:

```
$ tracepp file.trc
```

This give an optimal strategy for crossing the bridge in 17 minutes as claimed by the computer scientist adventurer.