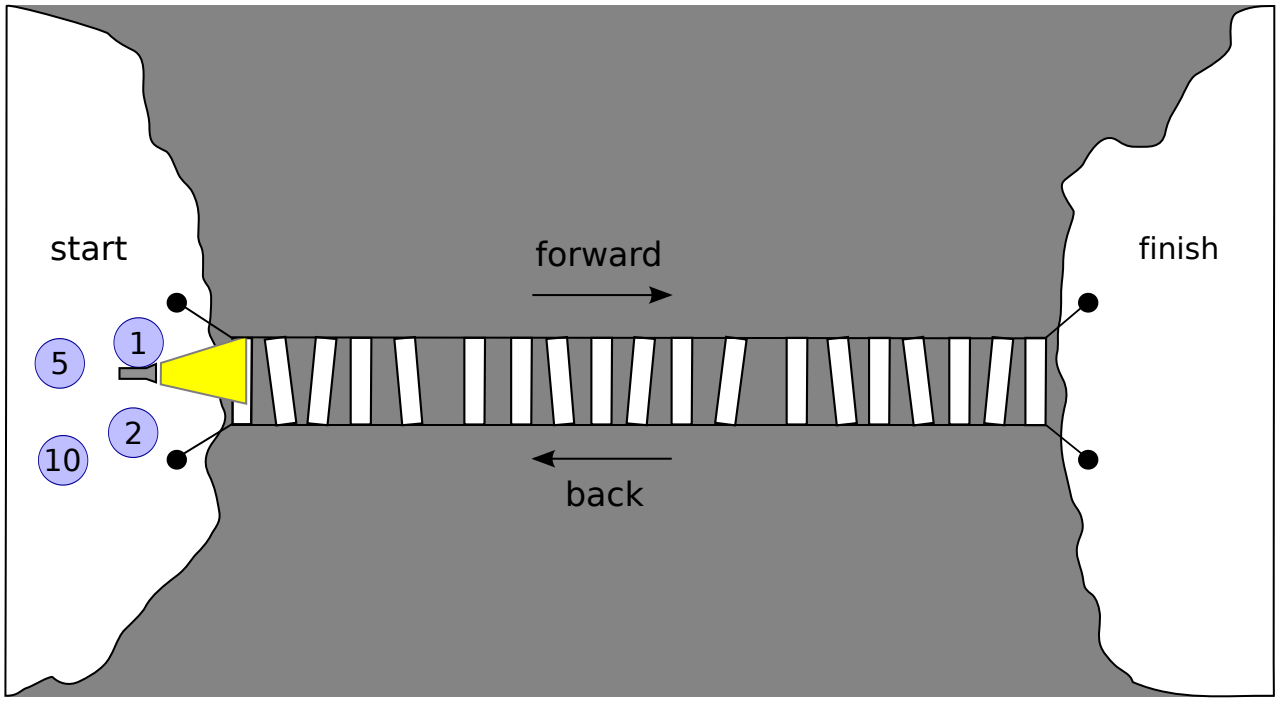
The Rope Bridge

Contribution of this section

1. exercise with processes,
2. use of state space exploration for checking properties, and
3. use of advanced visualisation techniques.

New tools: ltsview, tracepp



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 two persons should cross the bridge at the same time and that a flashlight needs to be carried by one of them on every crossing. They have only one flashlight. The four adventurers are not all 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 proclaims 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 dicates 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: Study the stub specification in `bridge-holes.mcrl2`. Then add the process definition for an adventurer. For this, answer the following questions:

- 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 at the places marked `%%% TODO (Exercise 1):` in the mCRL2 specification.

▲ Solution

The clue in the file gives away that an adventurer is identified by her speed. Furthermore, we need to keep track of the position of the adventurer; is she at the `start` or at the `finish` side of the bridge. This gives rise to defining an `Adventurer(speed: Int, pos: Position)`.

The adventurer needs to be able to cross the bridge in two directions; forward with another adventurer, and back on his own. Inspired by the actions defined for the flashlight, we define the following actions:

```
act forward_adventurer: Int # Int;
    back_adventurer: Int;
```

We combine this into a process in a way similar to the `Flashlight` process. This way we obtain the following definition:

```
% Models an adventurer who can move to the other side of the bridge with
% its designated speed
proc Adventurer(speed:Int, pos:Position) =
   (pos == start) ->
     sum s:Int . forward_adventurer(speed,s) . Adventurer(speed,finish)
   <> % position == finish
     back_adventurer(speed) . Adventurer(speed,start);
```

When we look closely at this process, and think about this, the resulting state space is more complex than it needs to be. We get two transitions for every forward movement of adventurers with speeds `X` and `Y`, `forward(X,Y)` and `forward(Y,X)`, both leading to the same state and modelling the same event. Because of this, we keep the speeds on the forward and backward actions sorted, as is done in the following definition of the process.

```
% Models an adventurer who can move to the other side of the bridge with
% its designated speed
proc Adventurer(speed:Int, pos:Position) =
       (pos == start) ->
         ( sum s:Int .
             % keep the parameters of forward actions sorted; otherwise
             % we get two transitions for every forward movement of
             % adventurers with speeds X and Y -- forward(X,Y) and
             % forward(Y,X) -- both leading to the same state and
             % modelling the same event.
             (s > speed) -> forward_adventurer(speed,s)
                          . Adventurer(speed,finish)
                         <> forward_adventurer(s,speed)
                         . Adventurer(speed,finish)
         )
       <> % position == finish
         back_adventurer(speed) . Adventurer(speed,start);
```

Exercise: Add the four 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:
  - Two adventurers and a flashlight move forward over the bridge.
  - One 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*, `allow`.

▲ Solution

Adding the four adventurers to the initial process definition (which already contains the flashlight) gives us the following definition:

```
init Adventurer(1,start) || Adventurer(2,start)  ||
     Adventurer(5,start) || Adventurer(10,start) ||
     Flashlight(start)
```

So all adventurers, with their different speeds, as well as the flashlight, are at the `start` side of the bridge.

We declare the action `forward: Int # Int` for two adventurers and a flashlight moving forward over the bridge. The action `back: Int` denotes one adventurer moving back over the bridge with the flashlight.

Synchronisation of the actions is done using the communication operator. Combined with the original initial process, we obtain the following:

```
init comm({ forward_adventurer | forward_adventurer
            | forward_flashlight -> forward,
           back_adventurer | back_flashlight -> back },
        Adventurer(1,start) || Adventurer(2,start)  ||
        Adventurer(5,start) || Adventurer(10,start) ||
        Flashlight(start));
```

Now we use the `allow` operator to only allow the `forward` and `backward` actions, as follows:

```
init
  allow({forward, back, report},
   comm({forward_adventurer | forward_adventurer | forward_flashlight
            -> forward,
          back_adventurer | back_flashlight -> back},
       Adventurer(1,start) || Adventurer(2,start)  ||
       Adventurer(5,start) || Adventurer(10,start) ||
       Flashlight(start)
     ));
```

Exercise: Simulate the model using the mCRL2 toolset by executing the following commands:

- Linearise the specification to obtain an LPS:

  ```
  $ mcrl22lps bridge.mcrl2 bridge.lps
  ```

- If linearisation fails, try to fix the reported errors. Otherwise, start the GUI simulation tool:

  ```
  $ lpsxsim bridge.lps
  ```

  The bottom part of the window shows the state parameters along with their values in the current state. (The simulator starts in the initial state of the system.) The top part shows the actions that can be performed from the current state, along with their effects on the parameter values.
- Simulate the system by executing a sequence of actions. You can execute an action by double-clicking it in the list. Notice how the state parameter values get updated in the bottom part.

After playing around with the simulator for a while, did you notice any weird or incorrect behaviour? If so, try to improve your model of the rope bridge and simulate it again.

▲ Solution

A complete specification is the following (also available as `bridge.mcrl2`.

```
% Specification for the rope bridge problem
% Written by Bas Ploeger, June 2008.

% Sort for the position of adventurers and 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;

act forward_adventurer, % an adventurer moves forward
    forward_flashlight, % the flashlight moves forward
    forward_referee,    % the referee processes a forward movement
    forward: Int # Int; % two adventurers and a flashlight move forward
                        % and the referee processes this

    back_adventurer, % an adventurer moves back
    back_flashlight, % the flashlight moves back
    back_referee,    % the referee processes a back movement
    back: Int;       % one adventurer and a flashlight move back and the
                     % referee processes this

    report: Int; % the referee reports that all adventurers have
                 % crossed the bridge along with the time that it took


% Models the flashlight which can move to the other side of the bridge
proc Flashlight(pos:Position) =
       (pos == start) ->
         sum s,s':Int . forward_flashlight(s,s') . Flashlight(finish)
       <> % position == finish
         sum s:Int . back_flashlight(s) . Flashlight(start);


% Models an adventurer who can move to the other side of the bridge with
% its designated speed
proc Adventurer(speed:Int, pos:Position) =
       (pos == start) ->
         ( sum s:Int .
             % keep the parameters of forward actions sorted; otherwise
             % we get two transitions for every forward movement of
             % adventurers with speeds X and Y -- forward(X,Y) and
             % forward(Y,X) -- both leading to the same state and
             % modelling the same event.
             (s > speed) -> forward_adventurer(speed,s)
                          . Adventurer(speed,finish)
                         <> forward_adventurer(s,speed)
                         . Adventurer(speed,finish)
         )
       <> % position == finish
         back_adventurer(speed) . Adventurer(speed,start);

init
  allow({forward, back, report},
   comm({forward_adventurer | forward_adventurer | forward_flashlight
            -> forward,
         back_adventurer | back_flashlight -> back},
       Adventurer(1,start) || Adventurer(2,start)  ||
       Adventurer(5,start) || Adventurer(10,start) ||
       Flashlight(start)
     ));
```

Exercise: Generate the state space of your model by executing the following command:

```
$ lps2lts bridge.lps bridge.svc
```

The state space can be viewed using the LTSGraph tool:

```
$ ltsgraph bridge.svc
```

An alternative, 3D view of the state space can be given by LTSView:

```
$ ltsview bridge.fsm
```

Exercise: 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 adventurer(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, add a `Referee` process definition and extend the initial process definition, including the communication and allow operators.

▲ Solution

A possible definition of the `Referee` process is the following.

```
% Models the referee who counts the number of minutes passed and the
% number of adventurers that have reached the far side of the bridge
proc Referee(minutes:Int, num_finished:Int) =
       sum s,s':Int . forward_referee(s,s')
                    . Referee(minutes + max(s,s'), num_finished + 2)
       +
       (num_finished < 4) ->
         sum s:Int . back_referee(s)
                   . Referee(minutes + s, num_finished - 1)
       <> % num_finished >= 4
         report(minutes) . Referee(minutes, num_finished);
```

Observe that the referee keeps track of the minutes that have passed, and that the `forward_referee`, `back_referee` and `report` actions have to be added. The `report` action is responsible for reporting the number of minutes that have passed when all adventurers have reached the far side of the bridge.

The initial process needs to be modified to the following, to incorporate the referee process.

```
init allow( { forward, back, report },
     comm(  { forward_adventurer | forward_adventurer |
              forward_flashlight | forward_referee -> forward,
              back_adventurer | back_flashlight | back_referee -> back },
       Adventurer(1,start) || Adventurer(2,start)  ||
       Adventurer(5,start) || Adventurer(10,start) ||
       Flashlight(start)   || Referee(0,0)
     ));
```

The complete specification is available as `bridge-referee.mcrl2`

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.

Exercise: Express each of these properties in the modal μ-calculus. Add the formulas to the files `formula_A.mcf` and `formula_B.mcf` using a text editor.

▲ Solution

Property A can e.g. be expressed using the following formula (available as `formula_A-final.mcf`):

```
% Express property A as a formula in the modal mu-calculus.
%
% It is possible for all adventurers to reach the 'finish' side in 17
% minutes.

< true* . report(17) > true
```

Property B can be expressed using the following formula (available as `formula_B-final.mcf`):

```
% Express property B as a formula in the modal mu-calculus.
%
% It is not possible for all adventurers to reach the `finish' side in
% less than 17 minutes.

forall x:Nat . val(x < 17) => [true* . report(x)] false
```

An alternative phrasing for property B is the following:

```
forall x:Nat . [true* . report(x)] val(x >= 17)
```

However, executing pbes2bool does not terminate if this phrasing is used.

Exercise: Verify the formulas using the toolset.

- Generate a PBES from your LPS and one of the formulas:

```
$ lps2pbes --formula=formula_X.mcf bridge.lps bridge_X.pbes
```

- Solve the PBES:

```
$ pbes2bool bridge_X.pbes
```

Alternatively, this can be done with a single command:

```
$ lps2pbes --formula=formula_X.mcf bridge.lps | pbes2bool
```

▲ Solution

This boils down to executing the following commands on `bridge.mcrl2`, assuming that this file contains the specification including the `Referee` process:

```
$ mcrl22lps bridge.mcrl2 bridge.lps
$ lps2pbes --formula=formula_A.mcf bridge.lps | pbes2bool
true
$ lps2pbes --formula=formula_B-final.mcf bridge-referee.lps | pbes2bool
true
```

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

Exercise: Verify the properties by generating traces as follows. 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. Also, a trace is written to 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 gives an optimal strategy for crossing the bridge in 17 minutes as claimed by the computer scientist adventurer.

▲ Solution

The following is the output for our solution:

```
$ lps2lts --action=report -t20 bridge.lps
[16:11:01.848 info]    detect: action 'report(50)' found and saved to 'bridge-r
[16:11:02.271 info]    detect: action 'report(40)' found and saved to 'bridge-r
[16:11:02.273 info]    detect: action 'report(34)' found and saved to 'bridge-r
[16:11:02.274 info]    detect: action 'report(33)' found and saved to 'bridge-r
[16:11:02.276 info]    detect: action 'report(37)' found and saved to 'bridge-r
[16:11:02.278 info]    detect: action 'report(30)' found and saved to 'bridge-r
[16:11:02.279 info]    detect: action 'report(24)' found and saved to 'bridge-r
[16:11:02.281 info]    detect: action 'report(23)' found and saved to 'bridge-r
[16:11:02.282 info]    detect: action 'report(36)' found and saved to 'bridge-r
[16:11:02.284 info]    detect: action 'report(27)' found and saved to 'bridge-r
[16:11:02.286 info]    detect: action 'report(21)' found and saved to 'bridge-r
[16:11:02.288 info]    detect: action 'report(20)' found and saved to 'bridge-r
[16:11:02.290 info]    detect: action 'report(26)' found and saved to 'bridge-r
[16:11:02.292 info]    detect: action 'report(19)' found and saved to 'bridge-r
[16:11:02.295 info]    detect: action 'report(17)' found and saved to 'bridge-r
```

So we indeed observe that `report(17)` is the trace with the lowest time. The optimal trace that we obtain is the following:

```
$ tracepp bridge-referee.lps_act_14_report.trc
```