Babysteps into mCRL2 (draft)

J.J.A. Keiren, B. Ploeger, and E.P. de Vink*

Dept. of Mathematics & Computer Science, Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, the Netherlands

Abstract. This tutorial serves as a starting point for getting experience in using mCRL2. We present the basic use of the major tools in the toolset, guided by a set of examples. Exercises that can be used to check your understanding are also provided.
This is a draft version. Comments are welcome.

1 Introduction

In this tutorial we give a basic introduction into the basic use of the mCRL2 toolset. In each of the sections we present a number of new concepts, guided by an example, and some exercises to gain hands on experience with the tools. Note that in this tutorial we mainly focus at the use of the tools, and not on the theory that is underlying the tools. For the latter, we refer to [2] as a brief introduction of the main concepts, and to [3] for an in-depth discussion.

1.1 Getting started

Before starting this tutorial you should first get a copy of mCRL2 for your platform from the mCRL2 website at `http://mcrl2.org/mcrl2/wiki/index.php/Download`. Installation instructions can be found at `http://mcrl2.org/mcrl2/wiki/index.php/Installation_instructions`. Note that, if you are using mCRL2 on Windows, then the compiling rewriter is unavailable, meaning that the flag `-r jittyc` to any of the tools will fail.

In this tutorial, we assume that you will be using the tools from the command line. On Windows this is the command prompt, on other platforms this is a terminal. Commands that should be enterend at the prompt are displayed as

```
$ command
```

2 A Vending Machine

Contribution of this section:

1. specifying processes,
2. linearisation,
3. state space exploration,
4. visualisation of state spaces,
5. comparison/reduction using behavioural equivalences, and
6. verification of modal mu-calculus properties.

* Corresponding author: evink@win.tue.nl

New tools : `mcrl22lps`, `lps2lts`, `ltsgraph`, `ltscompare`, `ltsconvert`, `lps2pbes`, `pbes2bool`.

Our first little step consists of number of variations on the good old vending machine, a user `User` interacting with a machine `Mach`. By way of this example we will encounter the basic ingredients of `mCRL2`. In the first variation of the vending machine, a very primitive machine, and user, are specified. Some properties are verified. In the second variation non-determinism is considered and, additionally, some visualization and comparison tools from the toolset are illustrated. The third variation comes closer to a rudimentary prototype specification.

2.1 First variation

After inserting a coin of 10 cents, the user can push the button for an apple. An apple will then be put in the drawer of the machine. See Figure 1.

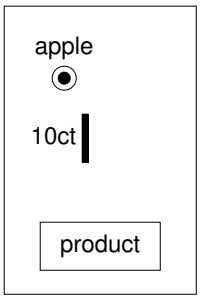


Fig. 1. vending machine 1

Vending machine 1 can be specified by the following `mCRL2`, also included in the file `vm01.mcrl2`.

```
act
  ins10, optA, acc10, putA, coin, ready ;

proc
  User = ins10 . optA . User ;
  Mach = acc10 . putA . Mach ;

init
  allow(
    { coin, ready },
    comm(
      { ins10 | acc10 -> coin, optA | putA -> ready },
      User || Mach
  ) ) ;
```

The specification is splitted in three sections: (i) `act`, a declaration of actions of 6 actions, (ii) `proc`, the definition of 2 processes, and (iii) `init`, the initialization of the system.

The process `User` is recursively defined as doing an `ins10` action, followed by an `optA` action, followed by the process `User` again. The process `Mach` is similar, looping on the action `acc10`

followed by the action `putA`. Note, only four actions are used in the definition of the processes. In particular, the action `coin` and `ready` are not referred to.

The initialization of the system has a typical form. A number of parallel processes, in the context of a communication function, with a limited set of actions allowed. So, `||` is the parallel operator, in this case putting the processes `User` and `Mach` in parallel. The communication function is the first argument of the `comm` operator. Here, we have that synchronization of an `ins`10 action and an `acc`10 action yields the action `coin`, whereas synchronization of *optA* and `putA` yields `ready`. The actions of the system that are allowed, are mentioned in the first argument of the allow operator `allow`. Thus, for our first system only `coin` and `ready` are allowed actions.

We compile the specification in the file `vm01.mcrl2` to a so-called linear process, saved in the file `vm01.lps`. This can be achieved by running

```
$ mcrl22lps vm01.mcrl2 vm01.lps
```

at the command line. The linear process in the internal representation format of `mCRL2`, not meant for human inspection. However, from `vm01.lps` a labeled transition system, LTS for short, can be obtained by running

```
$ lps2lts vm01.lps vm01.lts
```

which can be viewed by the `ltsgraph` facility, by typing

```
$ ltsgraph vm01.lts
```

at the prompt. Some manual beautifying yields the picture in Figure 2.

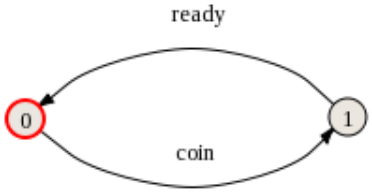


Fig. 2. LTS of vending machine 1

Apparently, starting from state 0 the system shuttles between state 0 and 1 alternating the actions `coin` and `ready`. Enforced by the allow operator, unmatched `ins`10, `acc`10, `optA` and `putA` actions are excluded. The actions synchronize pairwise, `ins`10 with `acc`10, `optA` with `putA`, to produce `coin` and `ready`, respectively.

As a first illustration of model checking in `mCRL2`, we consider some simple properties to be checked against the specification `vm01.mcrl2`. Given the LTS of the system, the properties obviously hold.

(a) *always, eventually a ready is possible* (true)

```
[true*] <true*.ready> true
```

(b) *a ready is always possible* (false)

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

(c) *after every ready only a coin follows* (true)

```
[true*.ready.!coin] false
```

(d) *any ready is followed by a coin and another ready* (true)

```
[true*.ready.!coin] false &&
    [true*.ready.true.!ready] false
```

Dissecting property (a), `[true*]` represents all finite sequences of actions starting from the initial state. `<true*.ready>` expresses the existence of a sequence of actions ending with the action `ready`. The last occurence of `true` in property (a) is a logical formula to be evaluated in the current state. Thus, if property (a) is satisfied by the system, then after any finite sequence of actions, `[true*]`, the system can continue with some finite sequence of actions ending with `ready`, `<true*.ready>`, and reaches a state in which the formula `true` holds. Since `true` always holds, property (a) states that a next `ready` is always possible.

Property (b) is less liberal than property (a). Here, `<ready> true` requires a `ready` action to be possible for the system, after any finite sequence, `[true*]`. This property does not hold. A `ready` action is not immediately followed by a `ready` action again. Also, `ready` is not possible in the initial state.

Property (c) uses the complement construct. `!coin` are all actions different from `coin`. So, any sequence of actions with `ready` as its one but final action and ending with an action different from `coin`, leads to a state where `false` holds. Since no such state exists, there are no path of the form `true*.ready.!coin`. Thus, after any `ready` action, any action that follows, if any, will be `coin`. Property (d) is a further variation involving conjunction `&&`.

Model checking with `mCRL2` is done by constructing a so-called parametrized boolean equation system or PBES from a linear process specification and a modal μ-calculus formula. For example, to verify property (a) above, we call the `lps2pbes` tool. Assuming property (a) to be in file `vm01a.mcf`, running

```
$ lps2pbes vm01.lps -f vm01a.mcf vm01a.pbes
```

creates from the system in linear format and the formula in the file `vm01a.mcf` right after the `-f` switch, a PBES in the file
filevm01a.pbes. On calling the PBES solver on `vm01a.pbes`,

```
$ pbes2bool vm01a.pbes
```

the `mCRL2` tool answers

```
The pbes is valid
```

So, for vending machine 1 it holds that action `ready` is always possible in the future. Instead of making separate steps explicity, the verification can also be captured by a single, pipe-line command:

```
$ mcrl22lps vm01.mcrl2 | lps2pbes -f vm01a.mcf | pbes2bool
```

Running the other properties yields the expected results. Properties (c) and (d) do hold, property (b) does not hold.

2.2 Second variation

Next, we add a chocolate bar to the assortment of the vending machine. A chocolate bar costs 20 cents, an apple 10 cents. The machine will now accept coins of 10 and 20 cents. The scenarios allowed are (i) insertion of 10 cent and purchasing an apple, (ii) insertion of 10 cent twice or 20 cent once and purchasing a chocolate bar. Additionally, after insertion of money, the user can push the 'change' button, after which the inserted money is returned. See Figure 3.

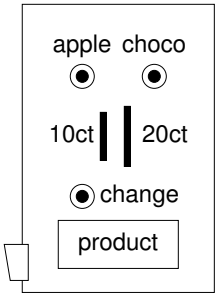


Fig. 3. vending machine 2

Exercise 2.1. Write an mCRL2 specification in file `vm02.mcrl2` for the vending machine sketched above, involving the actions

```
ins10, ins20, acc10, acc20, coin10, coin20,
chg10, chg20, ret10, ret20, out10, out20,
optA, optC, putA, putC, readyA, readyC, prod
```

A possible specification of the `Mach` process may read

```
Mach =
  acc10.( putA.prod + acc10.( putC.prod + ret20 ) + ret10 ).Mach +
  acc20.( putA.prod.ret10 + putC.prod + ret20 ).Mach ;
```

The machine is required to perform a `prod` action, for administration purposes. □

A visualization of the specified system can be obtained by first converting the linear process into a labeled transition system (in so-called SVC-format) by

```
$  lps2lts vm02.lps vm02.svc
```

and next loading the SVC file `vm02.svc` into the ltsgraph tool by

```
$  ltsgraph vm02.svc
```

The LTS can be beautified (a bit) using the 'start' button in the optimization panel of the user interface. Manual manipulation by dragging states is also possible. For small examples, increasing the natural transition length may provide better results.

Exercise 2.2. Prove that your specification satisfies

(a) "no three 10ct coins can be inserted in a row"
(b) "no chocolate after 10ct only"
(c) "an apple only after 10ct, a chocolate after 20ct" □

The file `vm02-taus.mcrl2` contains the specification of a system performing `coin10` and `coin20` actions as well as so-called τ-steps. Use the `ltscompare` tool to compare your model under branching bisimilarity with the LTS of the system `vm02-taus`, after hiding the actions `readyA`, `readyC`, `out10`, `out20`, `prod`. This can be done as follows.

```
$ ltscompare -ebranching-bisim --tau=out10,out20,readyA,readyC,prod \
    vm02.svc vm02-taus.svc
```

Minimize the LTS for `vm02.mcrl2` using `ltsconvert` with respect to branching bisimulation after hiding the readies and returns:

```
$ ltsconvert -ebranching-bisim --tau=out10,out20,readyA,readyC,prod \
    vm02.svc vm02min.svc
```

Also, compare the LTSs `vm02min.svc` and `vm02-taus.svc` visually using `ltsgraph`.

2.3 Third variation

A skeleton for a vending machine with parametrized actions is available in the file `vm03.mcrl2`.

Exercise 2.3. Modify this specification such that all coins of denomination 50ct, 20ct, 10ct and 5ct can be inserted. The machine accumulates upto a total of 60 cents. If sufficient credit, an apple or chocolate bar is supplied after selection. Money is returned after pressing the change button. Prove that your specification satisfies the properties in file `vm03.mcf`. □

3 Water cans

Contribution of this section:

1. use of standard data types, and
2. use of simulator.

New tools: `lpsxsim`.

Two water cans of known capacity, say of x and of y liters, a tap and a sink are at our disposal. The cans do not have a graduation to measure their content. The challenge is to pace an exact volume of water, say of z liter, in one of the cans.

Given the description, there are basically four things one can do with a can:

- to empty the can in the sink, which makes only sense if the can is non-empty;
- to fill the can completely from the tap, a proper thing to do for a can that is not full already
- to pour from the can into the other, provided there is water in the can;
- to fill the can by pouring from the other can, assuming the first is not brimful yet.

So, somehow we need to keep track of the actual content of the water cans, to see if an empty-to-sink action or an pour-to-other action can be done with the can, or that a fill-from-tap action or a fill-from-other action applies.

Processes in mCRL2 may carry one or more parameters. We can write, for example, `BigCan(3)` to express that the bigger of the two watercans contains 3 liter. In the process definition we need to declare the parameter, *e.g.* we write `BigCan(m:Nat) = ...` to have for `BigCan` the variable `m` ranging over the naturals, that include 0.

Similary, actions can have parameters. Here, we have occasion to express that ℓ liters of waters have been poured out. If the action `lose` denotes the pouring out, we do this by writing `lose(`ℓ`)`. In the definition of the action `lose` we need to indicate the parameter, *e.g.* by writing `lose: Nat` in the `act` part of the mCRL2 specification. Thus, ℓ ranges over non-negative integer values.

For the concrete case of cans of 5 and 8 liter to yield 4 liters, a first approximation may be as follows:

```
act

  empty, fill, done;
  lose, gain, pour: Nat;

proc

  BigCan(m:Nat) =
    ( m != 4 ) -> (
      %% some code for the big can
    ) <> done ;

  SmallCan(m:Nat) =
    ( m != 4 ) -> (
      %% some code for the small code
    ) <> done ;

init

  allow(
    { empty, fill, pour },
  comm(
    { lose|gain -> pour },
  BigCan(0) || SmallCan(0)
    ));
```

In this set-up, a `lose` action by the one can will synchronize with a `gain` action by the other can, together synchronizing as a `pour` action. The three actions all carry a parameter of type `Nat`, that needs to be equal for synchronization to succeed.

Another choice made above is the test whether the current content of the required volume to pace. If no, we do some activity left unspecified here, if yes we do the action `done`. The general form of the if-then-else construction in mCRL2 is `c -> p <> q` for condition `c` and processes `p` and `q`.

One may wonder in what way the specification for the `BigCan` process will differ from that for the `SmallCan`. It seems more appealling to make the capacity of the can a parameter too. An incomplete specification of a solution of the watercan problem is displayed in Figure 4. The term `Can(n,m)` indicates that a can of capacity `n` is currently holding a volume of `m`.

```
act

  empty, fill, done;
  lose, gain, pour: Nat ;

proc

  Can(n:Nat,m:Nat) =
    ( m != 4 ) -> (
      %% empty, if non-empty
      (m > 0) -> ( empty . Can(n,0) ) +
      %% fill, if not full
      (m < n) -> ( fill . Can(n,n) ) +
      %% pour to other if not empty
      (m > 0) -> (
        sum l:Nat . (
          ( (0 < l) && (l <= m) ) -> (
            lose(l) . Can(n,Int2Nat(m-l)) ) ) )  +
      %% pour from other if not full
      (m < n) -> (
        sum l:Nat . (
          ( (0 < l) && (l <= n-m) ) -> (
            gain(l) . Can(n,m+l) ) ) )
    ) <> done . delta ;

init

  allow(
    { empty, fill, done, pour },
  comm(
    { lose|gain -> pour },
  Can(5,0) || Can(8,0)
    ) ) ;
```

Fig. 4. Watercan specification 1

An LTS generated by `mcrl22lps`, `lps2lts` and `ltsconvert` via

```
$ mcrl22lps watercan01.mcrl2 watercan01.lps -D
$ lps2lts watercan01.lps watercan01.lts
$ ltsconvert -e branching-bisim watercan01.lts watercan01min.lts
```

has 46 states and 265 transitions. The option `-D` when calling `mcrl22lps` is necessary to suppress timing aspects that are not supported by `lps2lts`. Also note the explict termination as expressed by the constant `delta`. In this particular case, if left out the tool `mcrl22lps` would get start

generating an infinite process, rather than a finite one. The property `<true*.done> true`, asking whether it is possible to do the `done` action, is confirmed by model checking.

We can get some more feedback on what is going on by using the simulator `lpsxsim` with which we can step through the LTS and follow the the values of parameters. Calling at the command line the `lpsxsim` tool with the linear process `watercan01.lps`, produced by the `mcrl22lps` tool from the `mCRL2`-specification `watercan01.mcrl`, by typing

```
$ lpsxsim watercan01.lps
```

opens an application with two smaller windows, the top one listing possible transitions, the bottom one listing the values of the parameters in the current state. Parameters have symbolic names as a result of the linearization process. It looks similar to

```
Transitions
     Action                 State Change
       fill s31_Can1 := 2, m_Can11 := 8;
       fill    s3_Can1 :=2, m_Can1 := 5;

Current State
    Parameter                       Value
      s3_Can1                           1
       n_Can1                           5
       m_Can1                           0
     s31_Can1                           1
      n_Can11                           8
      m_Can11                           0
```

By double clicking on a transition, the transition can be taken. For example, clicking on the top `fill` transition yields a new list of transitions and an update current state. Now, besides a `fill` action also the actions `empty`, `pour(1)` to `pour(5)` are possible. The state now holds, *e.g.*, the value `8` for the contents `m_Can11` for the bigger can, as a result of the `fill` action.

The simulator reveals that we have made a mistake. Given a full can of 8 liter and an empty can of 5, the only volume we can pour from the bigger can into the smaller can is the volume of 5 liters, as no measure is available on the cans. Our specification, however, allows for all volumes from 1 upto 5 liters. Measuring 4 liters would then be easy, just pour 4 liters into the smaller can.

We can restrict the possible volumes that are poured over, by noting that either (i) the complete content of a can is poured into the other provided the latter can can hold, (ii) an amount of water is poured from a can into the other such that the other can is brimful. Hence, the minimum of the content of the from-can and the remaining capacity of the to-can determines the amount of water that is going from the one can to the other by pouring.

The basic idea then is to distinguish between an action `lose_all` and an action `lose_some` for pouring into the other can, and between an action `gain_all` and `gain_some` for getting from the other can. These actions will have a parameter for the amount of water involved. An action `lose_all(m)` synchronizes with the action `gain_some(m)`, with `m` liters in the first can; the action `lose_some(n-m)` matches the action `gain_all(n-m)`, now with `n` liters and `m` liters as capacity and current content of the second can, respectively. As synchronization function we will then have `lose_all | gain_some -> pour` as well as `lose_some | gain_all -> pour`. So, `pour` actions can be the result of two pairings of actions, `lose_all` with `gain_some` and `lose_some` with `gain_all`. See Figure 5.

```
sort

  Name = struct A | B ;

act

  empty, fill, done: Name ;
  lose_all, gain_some, lose_some, gain_all, pour: Name # Name # Nat ;

map
  sizeA, sizeB, target: Nat;

eqn
  sizeA = 5; sizeB = 8; target = 4;

proc

  Can(N:Name,n:Nat,m:Nat) =
    ( m != target ) -> (
      %% empty, if non-empty
      (m > 0) -> ( empty(N) . Can(N,n,0) ) +
      %% fill, if not full
      (m < n) -> ( fill(N) . Can(N,n,n) ) +
      %% pour all to other if not empty
      (m > 0) -> (
        sum M:Name . (
          lose_all(N,M,m) . Can(N,n,0) ) ) +
      %% pour some to other if not empty
      (m > 0) -> (
        sum l:Nat,M:Name . (
          ( (0 < l) && (l <= m) ) -> (
            lose_some(N,M,l) . Can(N,n,Int2Nat(m-l)) ) ) )  +
      %% pour all from other if not full
      (m < n) -> (
        sum M:Name . (
          gain_all(M,N,Int2Nat(n-m)) . Can(N,n,n) ) ) +
      %% pour some from other if not full
      (m < n) -> (
        sum l:Nat,M:Name . (
          ( (0 < l) && (l <= Int2Nat(n-m)) ) -> (
            gain_some(M,N,l) . Can(N,n,m+l) ) ) )
    ) <> done(N) . delta ;

init

  allow(
    { empty, fill, done, pour },
  comm(
    { lose_all|gain_some -> pour, lose_some|gain_all -> pour },
  Can(A,sizeA,0) || Can(B,sizeB,0)
    ));
```

Fig. 5. Watercan specification 2

In Figure 5 a further modification has been done, the introduction of names `A` and `B` for the cans. At the start of the specification a new sort is introduced, *viz.* the sort name. It is an enumeration sort, following the format `<sort name> = struct entity 1 | entity 2 | ... | entity k ;`. Here we have two entities, the name `A` and the name `B`, hence `k` equals 2. The new actions have been added, but also they can carry the name or names of cans involved. E.g., the action `empty(A)` indicates that can `A` has been emptied, `done(B)` indicates that the target value has been left in can `B`, whereas `lose_all(B,A,3)` represent that all of the current content of can `B`, apparently 3 liters, will be poured into can `A`. Therefore, the sort of the actions `empty` and `done` is `Name`, as they take a name as parameter, the sort of the action `lose_all` is `Name # Name # Nat` as the actions takes two names and a natural number as parameter. The summations in the `mCRL2` specification, now quantify over the name of the other can, called `M`, and exclude to pour from the can in itself by demanding `M != N`.

We also have occasion to introduce constants. The is a specific use of the facilities of `mCRL2` in supporting abstract data types. Here, after the keyword `map` we introduce three constants over the natural numbers, called `sizeA`, `sizeB` and `target`. Next, following the keyword `eqn`, we define them to hold the values `5`, `8` and `4`, respectively. The corresonding LTS has 35 states and 94 transitions. The LTS of a smaller example, can sizes 4 and 3 and target volume 2 with 18-states and 46 transitions after reduction modulo strong bisimulation is depicted in Figure 6.

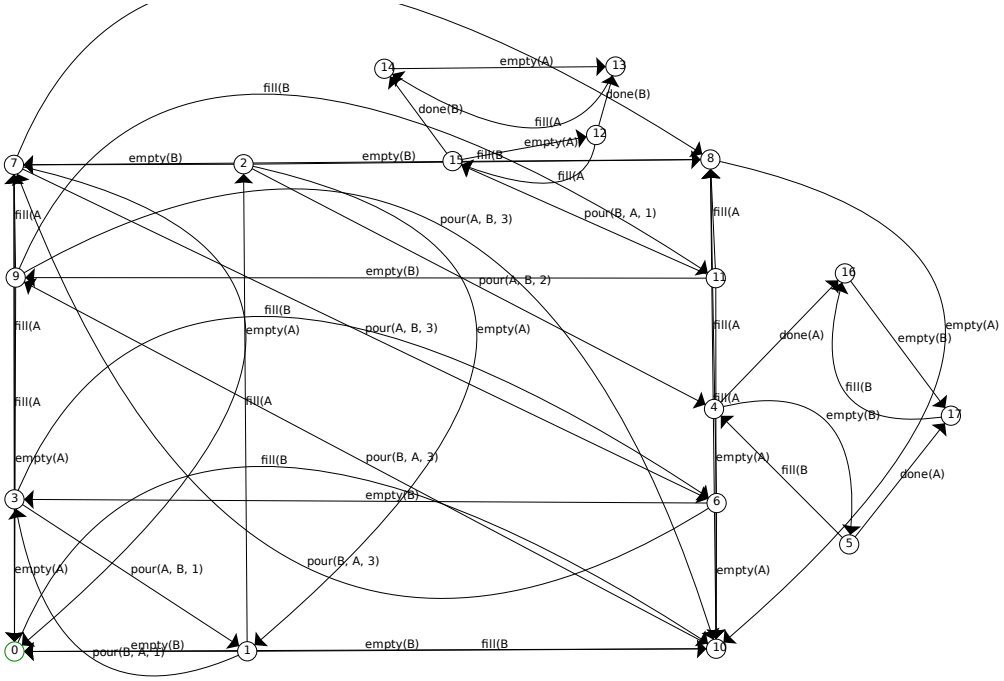


Fig. 6. LTS of watercan example with cans of 4 and 3 liters and target volume of 2 liters

4 Towers of Hanoi

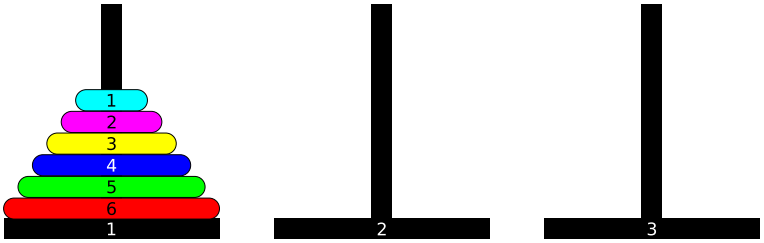
Contribution of this section:

1. use of lists,
2. use of functions,
3. use of data μ-calculus formulae.

New tools: none.

The Towers of Hanoi is a classic mathematical puzzle that involves three pegs (numbered 1, 2 and 3) and $N \geq 1$ discs. Every disc has a unique diameter and has a hole in the center so that it can slide onto any of the pegs. The discs are numbered 1 to N increasingly with their sizes. Initially, all discs are stacked onto peg 1 in increasing size from top to bottom (see the figure below where $N = 6$). The puzzle is solved when all discs are on peg 3 in the same order. Discs can be moved from one peg to any other, as long as the following rules are obeyed:

1. Only one disc can be moved at a time.
2. Only the topmost disc on a peg can be moved.
3. A disc cannot be placed on top of a smaller disc.



Over the next exercises we gradually construct an mCRL2 specification of the Towers of Hanoi puzzle. For this, we shall specify a `Peg` process to represent a peg and we shall use a list to represent the contents of a peg.

The list data structure is predefined in mCRL2. All elements in a list must be of the same type. This type is determined by the list's type declaration, which consists of the word `List` followed by the type of its elements between parentheses. For example, a list `l` of natural numbers is declared by `l:List(Nat)`. Lists can be explicitly enumerated, so that `[]`, `[1]`, and `[2,3,5]` are all valid list expressions, representing the empty list, the list with single element 1 and the list with elements 2, 3 and 5, respectively. Furthermore, the following operations on lists are provided:

- cons `|>`: insert an element at the front of a list, *e.g.* `1 |> [2]` gives `[1,2]`;
- snoc `<|`: insert an element at the back of a list, *e.g.* `[2] <| 1` gives `[2,1]`;
- head: return the first element of a list, *e.g.* `head([1,2])` gives `1`;
- tail: return a list without its first element, *e.g.*`tail([1,2,3])` gives `[2,3]`.

Note that the head of an empty list is undefined, so that `mCRL2` will not further reduce the term `head([])`. The tail of an empty list is simply the empty list, as is the tail of a list with one element only. The mCRL2 language supports more operations on lists, but they are not used in this section.

We shall use lists to represent stacks of discs, such that the head of the list corresponds with the top of the stack. A disc is represented by a positive natural number, which is an element of the predefined data sort `Pos`. Consider the following, incomplete data specification:

```
sort Stack = List(Pos);
map  empty: Stack -> Bool;
     push:  Pos # Stack -> Stack;
     pop:   Stack -> Stack;
     top:   Stack -> Pos;
var  s: Stack;
     x: Pos;
eqn  empty(s)  = ...;             % return whether s is empty
     push(x,s) = ...;             % put x on top of s
     pop(s)    = ...;             % remove top element from s
     (!empty(s)) -> top(s) = ...; % return top element of s
```

This defines the `Stack` data sort as lists of positive numbers and declares the functions (or *maps*) `empty`, `push`, `pop` and `top` as operations on stacks. These functions have to be defined using equations, in which variables can be used to represent any term of a certain type. For example, the second equation defines `push(x,s)` for any positive number `x` and any stack `s`, where variables `x` and `s` have been declared above the `eqn`-block. Equations can also have *guards* which limit the set of terms to which that equation applies. For example, the last equation defines `top(s)` only for stacks `s` for which the guard `!empty(s)` holds, *i.e.* non-empty stacks `s`.

Exercise 4.1. Complete the specification for the `Stack` data sort using the list operations introduced above.

Exercise 4.2. Your specification for the Towers of Hanoi puzzle has to be "parameterized" by the number of discs N, such that changing the value of N requires a change in one place of your specification only. For this, introduce the following maps:

- `N: Pos`, which holds the value of N;
- `build_stack: Pos # Pos -> Stack`, which creates a stack of discs.

Define equations for the function `build_stack` such that `build_stack(x,y)` returns the stack `[x,x+1,...,y]` for any positive numbers `x` and `y`. For example, `build_stack(1,4)` should return `[1,2,3,4]`. For now, define `N` to be equal to 3.

Exercise 4.3. Specify the `Peg` process in `mCRL2`. It should have two parameters:

- `id: Pos`, the peg's number;
- `stack: Stack`, the peg's stack of discs.

What actions can a single peg perform? What data parameters must these actions have? Declare the actions first and then define the `Peg` process in `mCRL2`.

Exercise 4.4. Specify the initial process. Use the `allow` and `comm` operators to enforce communication between the `Peg` processes.

The complete specification is given below, where the following actions are used:

- `move(d,p,q)`: disc d is moved from peg p to peg q;
- `receive(d,p,q)`: peg q receives disc d from peg p;
- `send(d,p,q)`: peg p sends disc d to peg q.

A `move` action is the result of synchronizing a `send` and a `receive` action.

```
map  N: Pos;
eqn  N = 3;

sort Stack = List(Pos);
map  empty: Stack -> Bool;
     push:  Pos # Stack -> Stack;
     pop:   Stack -> Stack;
     top:   Stack -> Pos;
var  s: Stack;
     x: Pos;
eqn  empty(s)  = s == [];
     push(x,s) = x |> s;
     pop(s)    = tail(s);
     (!empty(s)) -> top(s) = head(s);

map  build_stack: Pos # Pos -> Stack;
var  x,y: Pos;
eqn  (x >  y) -> build_stack(x,y) = [];
     (x <= y) -> build_stack(x,y) = push(x,build_stack(x+1,y));

act  send, receive, move: Pos # Pos # Pos;

proc Peg(id:Pos, stack:Stack) =
     sum d,p:Pos . (empty(stack) || top(stack) > d) ->
       receive(d,p,id) . Peg(id,push(d,stack))
     +
     sum p:Pos . (!empty(stack)) ->
       send(top(stack),id,p) . Peg(id,pop(stack));

init allow({move},
     comm({send|receive -> move},
       Peg(1,build_stack(1,N)) || Peg(2,[]) || Peg(3,[])
     ));
```

When generating the state spaces for $N = 1, \ldots, 6$, we find that the number of states is precisely 3^N. The state space for $N = 3$ is depicted in figure 7.

We use the tool `lps2lts` to see if there are any deadlocks by passing the `-D` option. No deadlocks are reported. This implies that this specification allows to continue moving discs when the solution has already been obtained. We disallow this by strengthening the guard for the `send` action in the `Peg` process to:

```
!empty(stack) && !(#stack == N && id == 3)
```

This ensures that the system deadlocks when all discs are on peg 3. When checking for deadlocks of the new specification we find precisely one, as expected. We save a trace to this deadlock in a file by adding the `-t` option. The contents of the file can be printed using `tracepp`:

```
move(1, 1, 3)
move(2, 1, 2)
move(1, 3, 2)
move(3, 1, 3)
```

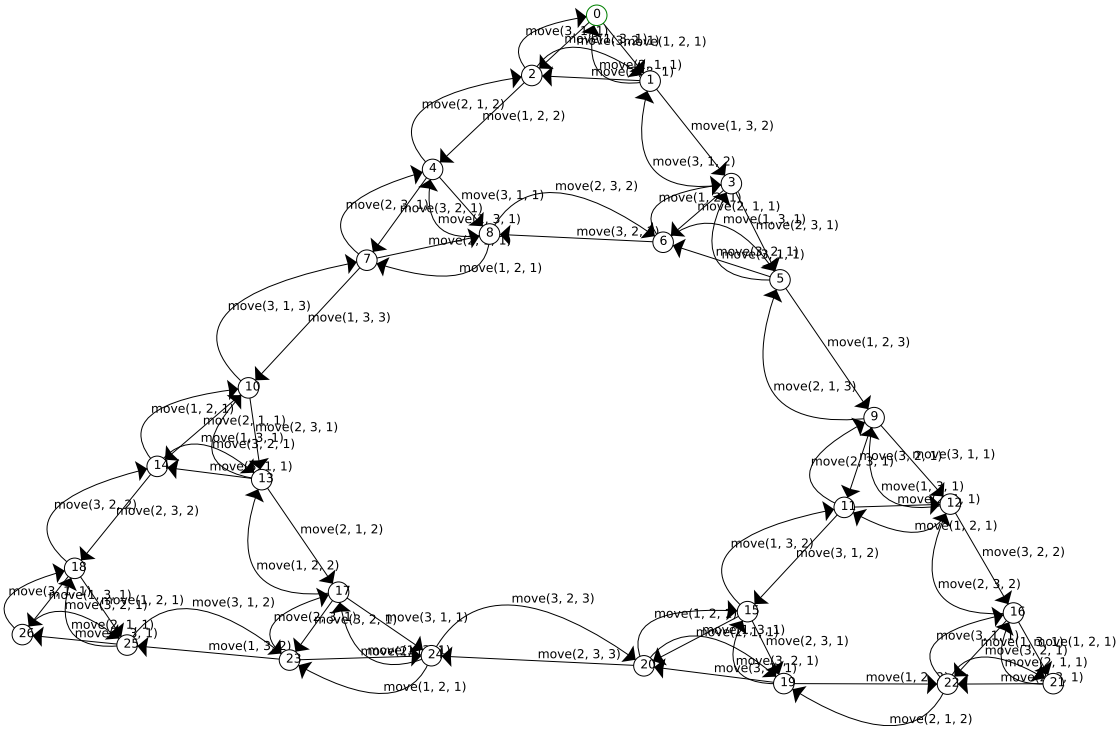


Fig. 7. State space of the Hanoi puzzle for 3 discs

```
move(1, 2, 1)
move(2, 2, 3)
move(1, 1, 3)
```

This is a sequence of moves leading towards the solution. It consists of 7 moves, and we now prove that there is no shorter path to the solution. In fact, we shall prove two properties:

1. There is a sequence of $2^N - 1$ moves to a deadlock.
2. There is no shorter sequence of moves to a deadlock.

These properties are captured by the following μ-calculus formulae:

1. $(\mu X(n{:}\mathbb{N}) \,.\, (n = 2^N - 1 \wedge [\top]\bot) \vee (n < 2^N - 1 \wedge \langle\top\rangle X(n+1)))(0)$
2. $(\nu X(n{:}\mathbb{N}) \,.\, n \geq 2^N - 1 \wedge ([\top]X(n+1) \vee \langle\top\rangle\top))(0)$

which can be expressed in the ASCII syntax as follows:

1. `mu X(n:Nat = 0) . (val(n == exp(2,N)-1) && [true]false) ||`
   `(val(n < exp(2,N)-1) && <true>X(n+1))`
2. `nu X(n:Nat = 0) . val(n >= exp(2,N)-1) &&`
   `([true]X(n+1) || <true>true)`

Supposing that a formula is contained in file `hanoi.mcf` and the LPS in `hanoi.lps`, we check the formula on the specification by generating and solving a PBES as follows:

```
$ lps2pbes -f hanoi.mcf hanoi.lps | pbes2bool
```

This yields *true* for both formulae, so both properties hold. Check these properties for various values of N.

4.1 Optimal strategy

It is known that the shortest sequence of moves for solving the Hanoi puzzle with N discs, is precisely the sequence that we obtain by repeatedly alternating the following two moves until all discs are on peg 3:

1. Move the smallest disc one peg to the left if N is odd, and to the right if N is even.
2. Perform the move that does not involve the smallest disc.

For move 1 we consider peg 1 to be right of peg 3 and peg 3 to be left of peg 1. Observe that move 2 exists and is uniquely defined, except for the initial and final situations of the puzzle.

We now adapt our `mCRL2` specification to model this optimal strategy only. In other words, the state space of our model will only consist of the shortest sequence of moves that leads to the solution. For this we first introduce a function `next` that yields the next peg to which the smallest disc has to move, according to move 1:

```
map  next: Pos -> Pos;
var  x:Pos;
eqn  (N mod 2 == 0) -> next(x) = x mod 3 + 1;
     (N mod 2 == 1) -> next(x) = (x-2) mod 3 + 1;
```

Our strategy for enforcing that only move 1 and move 2 occur alternatingly is to add a fourth process to the model that allows precisely those moves only. This process will take part in the synchronization of the `send` and `receive` actions with an `allowed` action to produce a `move` action. We then rely on the fact that *all* actions that participate in a synchronous communication have to be present in order for that communication to succeed. This way, a `move` can *only* occur if `send`, `receive` and `allowed` happen at the same time, with the same parameter values.

The process that performs the `allowed` actions is actually modelled by two processes: `AllowSmall` that allows move 1 and `AllowOther` that allows move 2. After performing an `allowed` action, every process then calls the other process to ensure that move 1 and move 2 alternate indeed. Below are the action declaration of `allowed` and the process definitions:

```
act  allowed: Pos # Pos # Pos;

proc AllowSmall =
       sum p:Pos . allowed(1,p,next(p)) . AllowOther;

     AllowOther =
       sum d,p,q:Pos . (d > 1) -> allowed(d,p,q) . AllowSmall;
```

Now, we enforce the aforementioned synchronization in the initial process definition. Because move 1 comes first, we call `AllowSmall` in the parallel composition.

```
init allow({move},
     comm({send|receive|allowed -> move},
       Peg(1,build_stack(1,N)) || Peg(2,[]) || Peg(3,[]) ||
       AllowSmall
     ));
```

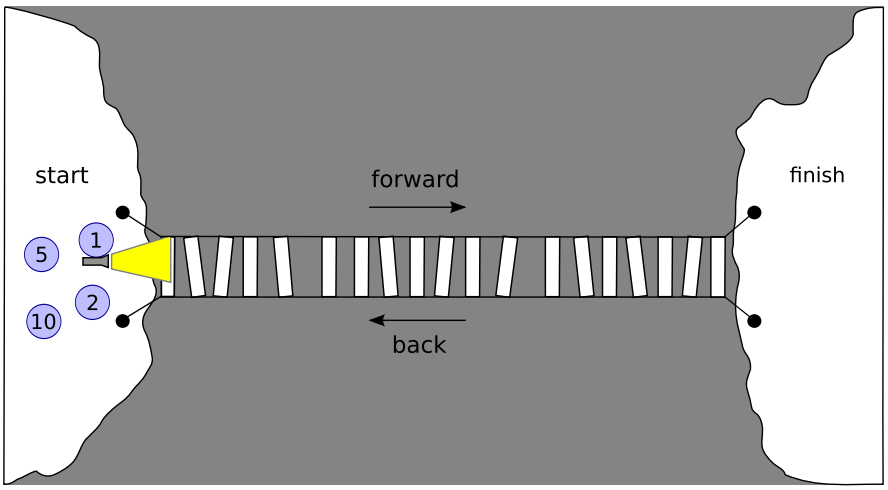
Generating the state space via `mcrl22lps -D` and `lps2lts` yields 8 states and 7 transitions for $N = 3$. In general, the state space has 2^N states and $2^N - 1$ transitions, as may be expected after our model-checking exercises on the complete model in the previous section. The action trace can be visualized by loading the state space into `ltsgraph`, or it can be simulated by loading the LPS into `lpsxsim`.

5 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 folder `RopeBridge` contains the files for these exercises:

- `bridge.mcrl2` which contains an incomplete `mCRL2` specification of the rope bridge problem;
- `formula_A.mcf` and `formula_B.mcf` to which μ-calculus formulas will be added.

Exercise 5.1. Open `bridge.mcrl2` in a text editor and study its contents. 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 designated places in the `mCRL2` specification. □

Exercise 5.2. 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`.

□

Exercise 5.3. Simulate the model using the mCRL2 toolset by executing the following commands:

– Linearise the specification to obtain an LPS:

  ```
  $ mcrl22lps -D bridge.mcrl2 bridge.lps
  ```

  Here, the `-D` option is passed because the specification does not contain time operators.
– 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. □

Exercise 5.4. 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, for which the state space first has to be converted to the FSM file format:

```
$ ltsconvert --lps=bridge.lps bridge.svc bridge.fsm
$ ltsview bridge.fsm
```

□

Exercise 5.5. 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 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. □

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 5.6. 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. □

Exercise 5.7. Verify the formulas using the toolset by executing the following commands:

- 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
```

□

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 5.8. 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. □

6 A Telephone Book

Contribution of this section:

1. use of functions with updates,
2. modelling considerations,
3. realistic verification process and its problems.

New tools: none.

In this section we describe the model of a telephone book in mCRL2.[1] We base our model on the following requirements:

[1] This example is based on the Phone Book example in [1].

– A phone book shall store phone numbers.
– It shall be possible to add and delete entries from a phone book.
– It shall be possible to retrieve a phone number given a name.

By looking at these requirements, we identify the following entities:

– phone book
– phone number
– name

We start by giving abstract types for phone numbers and names; their concrete form will be given later. For the phone book we decide that it is a mapping of names to numbers. This gives us the following specification of sorts.

```
sort Name;
     PhoneNumber;
     PhoneBook = Name -> PhoneNumber;
```

As a user, you need to be aware that the types as given here, are predetermined. This means that they cannot be extended on the fly. As a consequence, all names and phone numbers that can ever be added to the phone book must be known upfront.

If we again look at the requirements, our phone book must support the following operations:

addPhone Adds a phone number for a name.
delPhone Deletes a phone number corresponding to a name.
findPhone Finds the phone number corresponding to a name.

These operations will be the *actions* of our process. We need to decide on the parameters that the actions are going to take. We assume that our process will support a single phone book, *i.e.* the process itself model a phone book. It is then natural to model `addPhone` with parameters `Name` and `PhoneNumber`, `delPhone` with a `Name`, and `findPhone` with a `Name`. This gives rise to the following action specification.

```
act  addPhone: Name # Number;
     delPhone: Name;
     findPhone: Name;
```

We now need to take care that not every number is in every phone book. In order to describe a phone book as a total function, we introduce a special phone number, `p0`, to indicate that a name has no associated phone number.

```
map  p0: PhoneNumber;
```

Given this decision, we can specify the empty phone book as the phone book that maps every name to `n0`.

```
lambda n: Name . p0;
```

In modelling the empty phone book we use lambda abstraction. In this expression `lambda n: Name` says that we are defining a function that takes arguments of type `Name`, and for each name produces `p0` as a result. As `p0` is of type `PhoneNumber`, `lambda n: Name . p0` describes a function of type `Name ->PhoneNumber`, which is by definition equal to `PhoneBook`.

Given a function `b` of type `PhoneBook`, a name `n` and a phone number `p`, we can set the value of `n` in `b` to `p` using the expression `b[n -> p]`. This has as property that, for all names `m != n`, `b[n -> p](m) = b(m)`, and `b[n -> p](n) = p`.

Using the above ingredients, we can model a simple phone book using the following specification.

```
sort Name;
     PhoneNumber;
     PhoneBook = Name -> PhoneNumber;

%% Phone number representing the non-existant or undefined phone number,
%% must be different from any "real" phone number.
map  p0: PhoneNumber;

%% Operations supported by the phone book.
act  addPhone: Name # PhoneNumber;
     delPhone: Name;
     findPhone: Name;

%% Process representing the phone book.
proc PhoneDir(b: PhoneBook) =
     sum n: Name, p: PhoneNumber . addPhone(n, p) . PhoneDir(b[n->p])
   + sum n: Name . findPhone(n) . PhoneDir()
   + sum n: Name . delPhone(n) . PhoneDir(b[n->p0])
   ;

%% Initially the phone book is empty.
init PhoneDir(lambda n: Name . p0);
```

Exercise 6.1. There are some obvious flaws in the phone book that we have specified. Can you find and explain them? □

In the previous specification, the "special" phone number `p0` can be assigned to a name freely. Furthermore, a `findPhone` action can be performed, but the actual phone number is never reported.

Exercise 6.2. Fix these issues in the above specification. □

Preventing the assignment on `p0` to a name can easily prevented by guarding the `addPhone` action with `p != p0`. Fixing the second issue requires some more thought. There are two possible ways around fixing the problem. We can either assume that reporting of the result is immediate, and add the resulting phone number as a parameter to the `findPhone` action, or we can assume that querying for a phone number is asynchronous, and my take time, and split the query into the action initiating the query (`findPhone`) and an action reporting the result, *e.g.* `reportPhone`.

The first approach is suitable when, *e.g.*, modelling a phone book that is a library in a synchronous program in, say, C or Java. In that case indeed the program pointer of the calling program will not change before the result has been returned, making a model in which reporting the result a faithful representation of reality.

If we are, *e.g.*, modelling a phone book that is a web service, where a client performs a request, and in the meantime may do other kinds of actions like sending requests to other web services, the previous approach provides too coarse an abstraction. In this case it is more accurate to use the second approach, in which performing the query and obtaining the result are truely decoupled.

The following specification gives the first variation, in which the result is obtained instantaneously. Watch the extra parameter to `addPhone`.

```
%% Telephone directory, modified to actually report the phone number as an
%% answer to a query instantaneously.
```

```
sort Name;
     PhoneNumber;
     PhoneBook = Name -> PhoneNumber;

%% Phone number representing the non-existant or undefined phone number,
%% must be different from any "real" phone number.
map  p0: PhoneNumber;

%% Operations supported by the phone book.
act  addPhone: Name # PhoneNumber;
     delPhone: Name;
     findPhone: Name # PhoneNumber; % Added phone number as argument

%% Process representing the phone book.
proc PhoneDir(b: PhoneBook) =
     sum n: Name, p: PhoneNumber .
                 (p != p0) -> addPhone(n, p) . PhoneDir(b[n->p])
   + sum n: Name . findPhone(n,b(n)) . PhoneDir()
   + sum n: Name . delPhone(n) . PhoneDir(b[n->p0])
   ;

%% Initially the phone book is empty.
init PhoneDir(lambda n: Name . p0);
```

The second variation is given by the following specification.

```
%% Telephone directory, modified to asynchronously report the phone number
%% corresponding to the queried name.

sort Name;
     PhoneNumber;
     PhoneBook = Name -> PhoneNumber;

%% Phone number representing the non-existant or undefined phone number,
%% must be different from any "real" phone number.
map  p0: PhoneNumber;

%% Operations supported by the phone book.
act  addPhone: Name # PhoneNumber;
     delPhone: Name;
     findPhone: Name;
     reportPhone: Name # PhoneNumber; % Added action

%% Process representing the phone book.
proc PhoneDir(b: PhoneBook) =
     sum n: Name, p: PhoneNumber .
                (p != p0) -> addPhone(n, p) . PhoneDir(b[n->p])
   + sum n: Name . findPhone(n) . reportPhone(n, b(n)) . PhoneDir()
   + sum n: Name . delPhone(n) . PhoneDir(b[n->p0])
```

```
   ;

%% Initially the phone book is empty.
init PhoneDir(lambda n: Name . p0);
```

In the rest of this tutorial we will stick to the specification with asynchronous reporting.

In complex specifications, it can be convenient to introduce additional functions, with descriptive names, that take care of the modifications of parameters that is done in the process. As a bonus this usually makes it easier to change the data structures used in a specification.

Exercise 6.3. Modify the specification in `phonebook2b.mcrl2` by adding functions `emptybook`, `add_phone`, `del_phone` and `find_phone` with the following signatures.

```
map  emptybook: PhoneBook;
     add_phone: PhoneBook # Name # PhoneNumber -> PhoneBook;
     del_phone: PhoneBook # Name -> PhoneBook;
     find_phone: PhoneBook # Name -> PhoneNumber;
```

□

A solution to the above exercise is given by the following specification.

```
%% Telephone directory, modified to asynchronously report the phone number
%% corresponding to the queried name. Functions have been added to increase
%% readability and flexibility.

sort Name;
     PhoneNumber;
     PhoneBook = Name -> PhoneNumber;

%% Phone number representing the non-existant or undefined phone number,
%% must be different from any "real" phone number.
map  p0: PhoneNumber;
     emptybook: PhoneBook;
     add_phone: PhoneBook # Name # PhoneNumber -> PhoneBook;
     del_phone: PhoneBook # Name -> PhoneBook;
     find_phone: PhoneBook # Name -> PhoneNumber;

eqn  emptybook = lambda n: Name . p0;

var b: PhoneBook;
    n: Name;
    p: PhoneNumber;
eqn  add_phone(b, n, p) = b[n->p];
     del_phone(b, n) = b[n->p0];
     find_phone(b, n) = b(n);

%% Operations supported by the phone book.
act  addPhone: Name # PhoneNumber;
     delPhone: Name;
     findPhone: Name;
     reportPhone: Name # PhoneNumber; % Added action
```

```
%% Process representing the phone book.
proc PhoneDir(b: PhoneBook) =
     sum n: Name, p: PhoneNumber .
               (p != p0) -> addPhone(n, p) . PhoneDir(add_phone(b,n,p))
   + sum n: Name . findPhone(n) . reportPhone(n, find_phone(b,n)) . PhoneDir()
   + sum n: Name . delPhone(n) . PhoneDir(del_phone(b,n))
   ;

%% Initially the phone book is empty.
init PhoneDir(emptybook);
```

It should be noted that, instead of using a function of names to phone numbers, we could also have modelled the phone book using a set of pairs of names and phone numbers. A model using sets is likely to become complicated in this case.

Exercise 6.4. Modify the previous specification such that it uses a set of pairs of names and phone numbers instead of function from names to phone numbers to store the phone numbers internally. □

The following is a possible solution to the above exercise. Note that the function `find_phone` cannot be implemented using sets, because no concrete elements can be taken from the set. Therefore, the functionality of `find_phone` is modelled using the `sum` operator on a process level.

```
%% file phonebook4.mcrl2
%% Telephone directory, modified to asynchronously report the phone number
%% corresponding to the queried name. Functions have been added to increase
%% readability and flexibility, and instead of functions, sets are used.

sort Name;
     PhoneNumber;
     Pair = struct pair(name: Name, phone: PhoneNumber);
     PhoneBook = Set(Pair);

%% Phone number representing the non-existant or undefined phone number,
%% must be different from any "real" phone number.
map  p0: PhoneNumber;
     emptybook: PhoneBook;
     add_phone: PhoneBook # Name # PhoneNumber -> PhoneBook;
     del_phone: PhoneBook # Name -> PhoneBook;

eqn  emptybook = {};

var b: PhoneBook;
    n: Name;
    p: PhoneNumber;
eqn  add_phone(b, n, p) = b + {pair(n, p)};
     del_phone(b, n) = { x: Pair | x in b && name(x) != n };
% alternative definition for del_phone:
%     del_phone(b, n) = b - { x: Pair | name(x) == n };
```

```
%% Operations supported by the phone book.
act  addPhone: Name # PhoneNumber;
     delPhone: Name;
     findPhone: Name;
     reportPhone: Name # PhoneNumber; % Added action

%% Process representing the phone book.
proc PhoneDir(b: PhoneBook) =
     sum n: Name, p: PhoneNumber .
             (p != p0) -> addPhone(n, p) . PhoneDir(add_phone(b,n,p))
   + sum n: Name . findPhone(n) . sum p: PhoneNumber .
            (pair(n, p) in b) -> reportPhone(n, p) . PhoneDir()
   + sum n: Name . delPhone(n) . PhoneDir(del_phone(b,n))
   ;

%% Initially the phone book is empty.
init PhoneDir(emptybook);
```

In the rest of this chapter we stick to the model in which functions occur directly in the specification. We are going to check whether our model makes sense. A suitable property for our specification is: "if a name n with phone number p is added to the phone book, and a lookup of name n is performed, then phone number p should be reported, provided that in the meantime no other phone number has been added for name n, and the phone number for name n has not been deleted". We can write this as follows:

```
forall n: Name, p,r: PhoneNumber .
  [true* . addPhone(n,p) .
     !(delPhone(n) || exists q: PhoneNumber . addPhone(n, q))* .
     findPhone(n) .
     !(delPhone(n) || exists q: PhoneNumber . addPhone(n, q))* .
     reportPhone(n, r)] val(p == r)
```

We can try checking this property using the following command.

```
  $ mcrl22lps -D phonebook2b.mcrl2 | lps2pbes -f phonebook1.mcf | pbes2bool
```

The tools fail to verify this requirement, and give throw the following error:

```
error: Cannot find a term of sort Name
```

The tool is telling you that it wants to find some representative term of sort `Name`, but is not able to do so. This indeed makes sense as we have not given a specification of names and phone numbers yet. We can try to verify this requirement for a specification in which we have some fixed set of names and numbers, as is given in `phonebook5.mcrl2`.

```
%% Telephone directory, modified to asynchronously report the phone number
%% corresponding to the queried name. The sorts Name and PhoneNumber are
%% constrained to have a small, constant number of elements.

sort Name = struct n0 | n1 | n2;
%% Phone number p0 is assumed to represent the non-existant or undefined
```

```
%% phone number. This must be different from any "real" phone number.
%% This is already guaranteed by definition of a structured sort, in which all
%% elements are different.
     PhoneNumber = struct p0 | p1 | p2 | p3 ;
     PhoneBook = Name -> PhoneNumber;

%% Operations supported by the phone book.
act  addPhone: Name # PhoneNumber;
     delPhone: Name;
     findPhone: Name;
     reportPhone: Name # PhoneNumber; % Added action

%% Process representing the phone book.
proc PhoneDir(b: PhoneBook) =
     sum n: Name, p: PhoneNumber . (p != p0) -> addPhone(n, p) . PhoneDir(b[n->p])
   + sum n: Name . findPhone(n) . reportPhone(n, b(n)) . PhoneDir()
   + sum n: Name . delPhone(n) . PhoneDir(b[n->p0])
   ;

%% Initially the phone book is empty.
init PhoneDir(lambda n: Name . p0);
```

The specification is now easily checked using the following sequence of commands.

```
mcrl22lps -D phonebook5.mcrl2 | lps2pbes -f phonebook.mcf | pbes2bool
```

Exercise 6.5. Verify whether the following property holds for `phonebook5.mcrl2`. "if a name n with phone number p is added to the phone book, and a lookup of name n is performed, then phone number p should be reported, provided that in the meantime the phone number for name n has not been deleted". You first need to formalise this property as a μ-calculus formula, and then verify whether it holds. Explain the outcome of the verification. □

We see that this verification fails, because `addPhone` allows you to add a phone number for a person that already has a phone number. If a new phone number is added for such a person, the original phone number is overwritten.

Exercise 6.6. Modify the specification in `phonebook5.mcrl2` such that `addPhone(n,p)` can only be executed if no phone number for name `n` is known. Furthermore, extend the specification with and action `changePhone` with signature

```
changePhone: Name # PhoneNumber
```

such that `changePhone(n,p)` can only be executed if `n` already has a phone number, and that afterwards the phone number of `n` has been updated to `p`. □

Exercise 6.7. Verify whether your new specification satisfies the property you formulated before. Explain the outcome. □

Exercise 6.8. If the verification in the previous exercise failed, think about the influence of the `changePhone` action on the validity of the property you are trying to check. Describe the changed property in natural language, give the modal μ-calculus formula, and do the verification. □

References

1. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A tutorial introduction to PVS. *Workshop on Industrial-Strength Formal Specification Techniques*, 1995.
2. J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and van M.J. Weerdenburg. Analysis of distributed systems with mCRL2. In M. Alexander and W. Gardner, editors, *Process Algebra for Parallel and Distributed Processing*, pages 99–128. Chapman & Hall, 2009.
3. J.F. Groote and M.A. Reniers. *Modelling and Analysis of Communicating Systems.* 2011.