Distributed Data Base

Abstract
This is a small toy example which is well-suited as an introduction to occurrence graphs. The analysis of the occurrence graph is described in great detail.

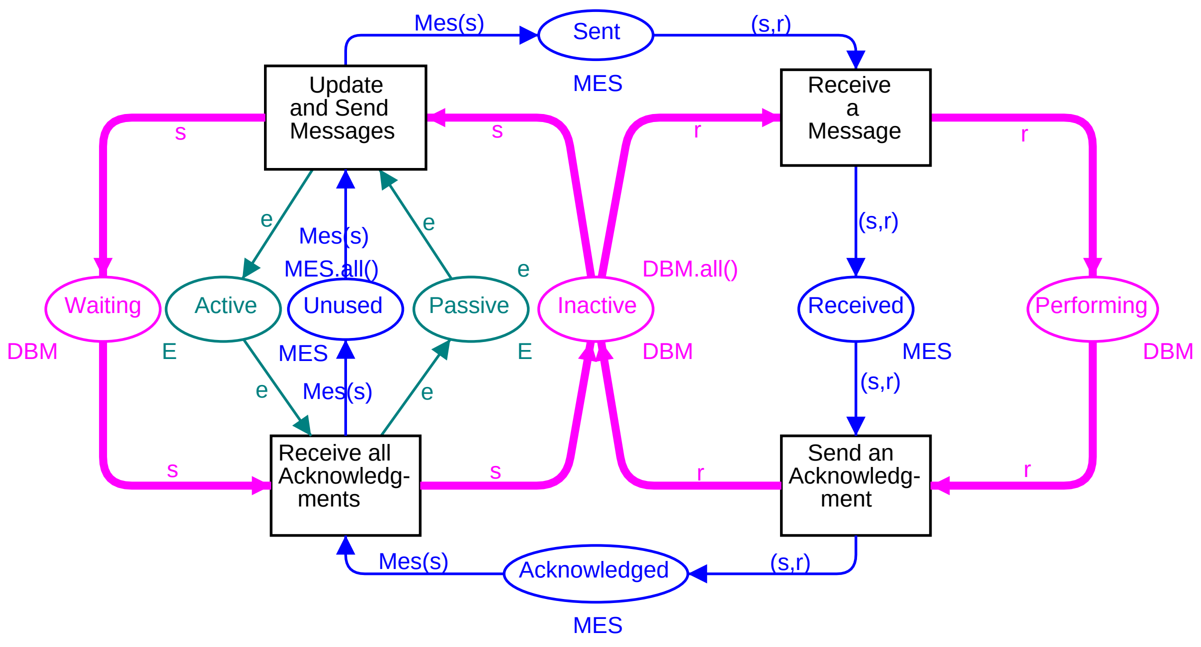
The CPN model describes the communication between a set of data base managers in a distributed system. The model is identical to the “Distributed Data Base” presented in “Introductory Examples”(which we recommend to study before this example).

The example is taken from Sect. 1.5 of Vol. 2 of the CPN book.

Developed and Maintained by:
Kurt Jensen, Aarhus University, Denmark (kjensen@daimi.au.dk).

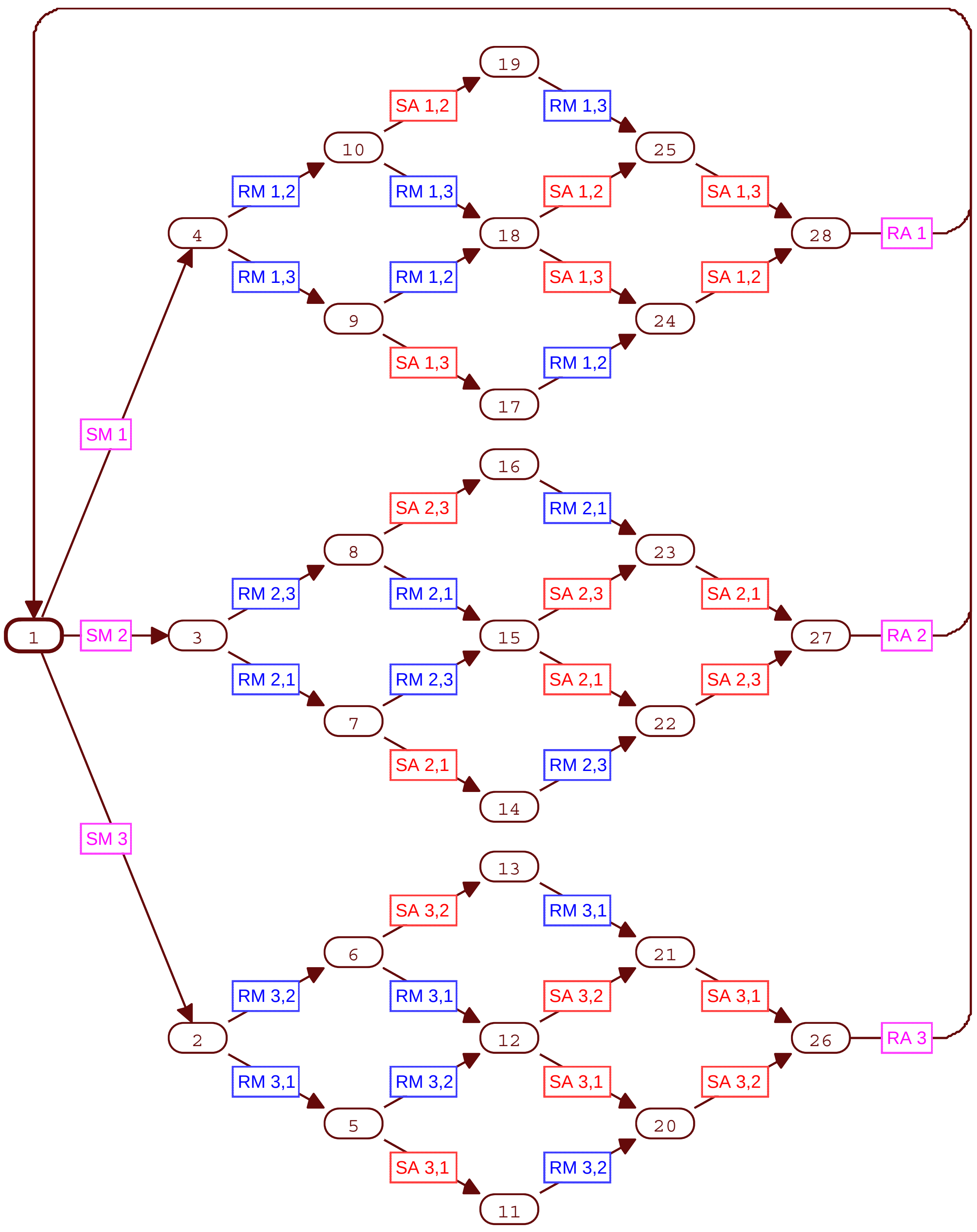
CPN Model

In this example we study the O-graph for the data base system, i.e., the O-graph for the following CP-net:



```
val n = 3;
color DBM = index d with 1..n;
volor PR = product DBM * DBM;
fun diff(x,y) = (x<>y);
color MES = subset PR by diff ;
color E = with e;
fun Mes(s) = PR.mult(1`s, DBM.all()--1`s);
var s,r: DBM;
```

For three data base managers the O-graph looks as shown below. The current version of CPN Tools does not include facilities for drawing O-graphs. Node number one is the initial marking. To save space the transition names are abbreviated to SM, RM, SA, and RA. Moreover, we write SM i and RM i, k instead of (SM,<s = d_i>) and (RM,<s = d_i, r = d_k>), and analogously for SA and RA.



The standard report looks as shown below.

From the statistics we see that there only is one strongly connected component. This means that all reachable states are reachable from each other.

```
 Statistics
-----------------------------
  Occurrence Graph
    Nodes:  28
    Arcs:   42
    Secs:   0
    Status: Full

  Scc Graph
    Nodes:  1
    Arcs:   0
    Secs:   0
```

All the integer bounds are as expected (see below). In particular, we see that at most one process can be *Waiting*. This tells us that a new update cannot start until all data base managers have finished the processing of the previous one. Also the multi-set bounds are as expected. To improve the readability, we have substituted *DBM* for the multi-set 1\`d(1) + 1\`d(2) + 1\`d(3) and *MES* for 1\`(d(1),d(2)) + 1\`(d(1),d(3)) + 1\`(d(2),d(1)) + 1\`(d(2),d(3)) + 1\`(d(3),d(1)) + 1\`(d(3),d(2)).

```
 Boundedness Properties
------------------------------
  Best Integers Bounds
              Upper   Lower
  Acknowledged   2       0
  Active         1       0
  Inactive       3       0
  Passive        1       0
  Performing     2       0
  Received       2       0
  Sent           2       0
  Unused         6       4
  Waiting        1       0

  Best Upper Multi-set Bounds
  Acknowledged   MES
  Active         1`e
  Inactive       DBM
  Passive        1`e
  Performing     DBM
  Received       MES
  Sent           MES
  Unused         MES
  Waiting        DBM

  Best Lower Multi-set Bounds
  Acknowledged   empty
  Active         empty
  Inactive       empty
  Passive        empty
  Performing     empty
  Received       empty
  Sent           empty
  Unused         empty
  Waiting        empty
```

The home properties tell us that all reachable markings are home markings. From the drawing of the occurrence graph, we can actually deduce that the system has a much stronger property. It is not only *possible* to return to the initial marking. This will *always* happen – whenever $2*n$ transitions have occurred.

```
 Home Properties
------------------------------------
  Home Markings:  All
```

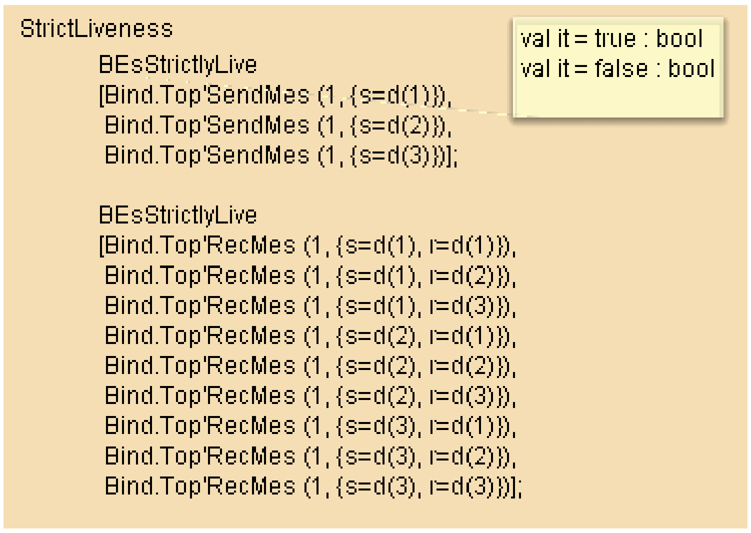
Also the liveness properties are as expected. There are no dead markings and all transitions are live.

```
 Liveness Properties
------------------------------------
  Dead Markings:  None
  Dead Transitions Instances: None
  Live Transitions Instances: All
```

Finally, the fairness properties tells us that all transitions are impartial. This is also easy to see from the drawing of the occurrence graph. Whenever $2*n$ transitions have occurred *SendMes* and *RecAck* have occurred exactly one time each, while *RecMes* and *SendAck* have occurred exactly $n-1$ times each.

```
 Fairness Properties
------------------------------------
  SendMes                     Impartial
  RecMes                      Impartial
  SendAck                     Impartial
  RecAck                      Impartial
```

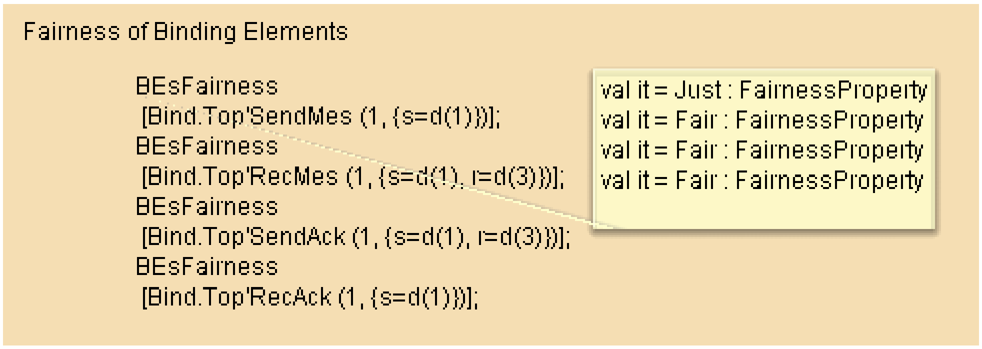
Now let us look at some model dependent properties. First we investigate whether the transitions are strictly live. For *SendMes* and *RecMes* the queries look as shown below. They show us that *SendMes* is strictly live, while *RecAck* is not – because binding elements such as (RecAck,<s=d(2),r=d(2)>) are dead. If we add a guard, [s<>r], to *RecAck*, the transition becomes strictly live.



```
StrictLiveness
        BEsStrictlyLive
        [Bind.Top'SendMes (1,{s=d(1)}),
         Bind.Top'SendMes (1,{s=d(2)}),
         Bind.Top'SendMes (1,{s=d(3)})];

        BEsStrictlyLive
        [Bind.Top'RecMes (1,{s=d(1), r=d(1)}),
         Bind.Top'RecMes (1,{s=d(1), r=d(2)}),
         Bind.Top'RecMes (1,{s=d(1), r=d(3)}),
         Bind.Top'RecMes (1,{s=d(2), r=d(1)}),
         Bind.Top'RecMes (1,{s=d(2), r=d(2)}),
         Bind.Top'RecMes (1,{s=d(2), r=d(3)}),
         Bind.Top'RecMes (1,{s=d(3), r=d(1)}),
         Bind.Top'RecMes (1,{s=d(3), r=d(2)}),
         Bind.Top'RecMes (1,{s=d(3), r=d(3)})];
```

```
val it = true : bool
val it = false : bool
```

Next we investigate the fairness properties of some typical binding elements. We see that the binding element of *SendMes* is just, while those of the other three transitions are fair.



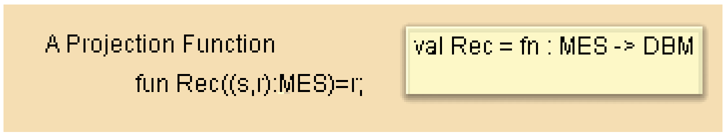
```
Fairness of Binding Elements

        BEsFairness
         [Bind.Top'SendMes (1,{s=d(1)})];
        BEsFairness
         [Bind.Top'RecMes (1,{s=d(1), r=d(3)})];
        BEsFairness
         [Bind.Top'SendAck (1,{s=d(1), r=d(3)})];
        BEsFairness
         [Bind.Top'RecAck (1,{s=d(1)})];
```

```
val it = Just : FairnessProperty
val it = Fair : FairnessProperty
val it = Fair : FairnessProperty
val it = Fair : FairnessProperty
```

Finally, let us demonstrate that occurrence graphs also can be used to check whether place invariants are fulfilled. It should, however, be stressed that the best way to check place invariants (for complex systems) is by checking the place flow property, which is a static and local property that can be checked *without* generating all possible system states. We want to check the following two place invariants:

$$\text{M(Performing)} = \text{Rec(M(Received))}$$
$$\text{Mes(Waiting)} = \text{M(Sent)} + \text{M(Received)} + \text{M(Acknowledged)}.$$

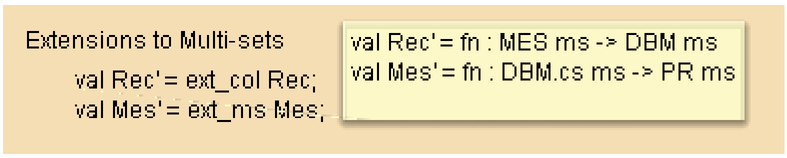
We first define a function *Rec* that maps a message into its receiver.



```
A Projection Function
        fun Rec((s,r):MES)=r;
```

```
val Rec = fn : MES -> DBM
```

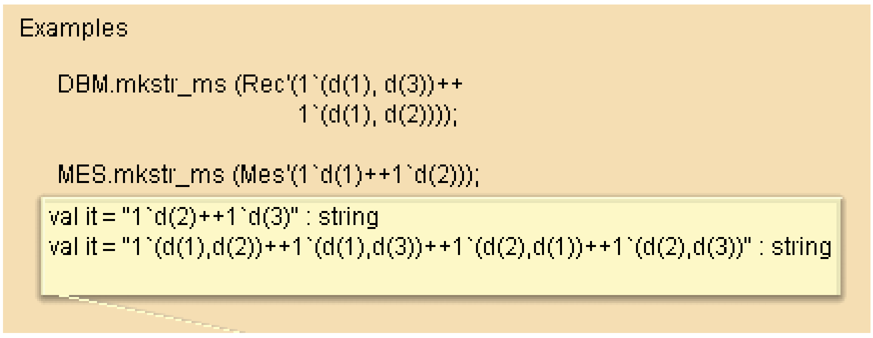
Then we extend *Rec* and *Mes* to two new functions *Rec'* and *Mes'* which can be applied to *multi-sets* of data base managers. This is done by means of two predeclared functions *ext_col* and *ext_ms*. The first of these extends a function $[A \varnothing B]$ to a function $[A_{MS} \varnothing B_{MS}]$, while the second extends a function $[A \varnothing B_{MS}]$ to a function $[A_{MS} \varnothing B_{MS}]$.



```
Extensions to Multi-sets

     val Rec'= ext_col Rec;
     val Mes'= ext_ms Mes;
```

```
val Rec'= fn : MES ms -> DBM ms
val Mes'= fn : DBM.cs ms -> PR ms
```

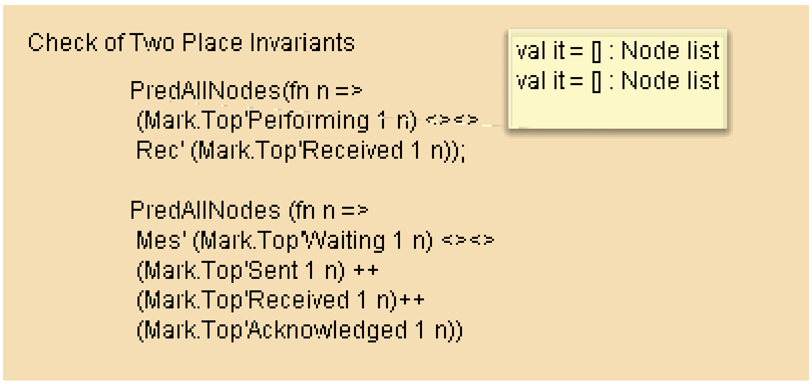
By definition the result of using an extended function to a multi-set is obtained by using the original function to each element in the multi-set – adding the results. This is illustrated by the following examples. The two *mkstr_ms* functions map the ML representation of a DBM/MES multi-set into a much more readable string representation.



```
Examples

    DBM.mkstr_ms (Rec'(1`(d(1), d(3))++
                       1`(d(1), d(2))));

    MES.mkstr_ms (Mes'(1`d(1)++1`d(2)));
```

```
val it = "1`d(2)++1`d(3)" : string
val it = "1`(d(1),d(2))++1`(d(1),d(3))++1`(d(2),d(1))++1`(d(2),d(3))" : string
```

Finally, we use a predeclared search function called *PredAllNodes* to list all nodes violating the two invariants. There are no such nodes, and hence we have proved that the invariants are fulfilled in all reachable markings. Please note that the <><> operator checks whether two multi-sets differ from each other (if you replace <><> by <> you only check whether the *representations* of the two multi-sets differ from each other).



```
Check of Two Place Invariants

          PredAllNodes(fn n =>
           (Mark.Top'Performing 1 n) <><>
           Rec' (Mark.Top'Received 1 n));

          PredAllNodes (fn n =>
           Mes' (Mark.Top'Waiting 1 n) <><>
           (Mark.Top'Sent 1 n) ++
           (Mark.Top'Received 1 n)++
           (Mark.Top'Acknowledged 1 n))
```

```
val it = [] : Node list
val it = [] : Node list
```

For the data base system it is rather easy to calculate how fast the O-graph grows – when we increase the number of data base managers. The results are as shown below. They illustrate the **space complexity** of the O-graph algorithm:

&DBM &	Nodes	Arcs
$O(\mathrm{n})$	$O(\mathrm{n} * 3^{\mathrm{n}})$	$O(\mathrm{n}^2 * 3^{\mathrm{n}})$
2	7	8
3	28	42
4	109	224
5	406	1,090
6	1,459	4,872
7	5,104	20,426
8	17,497	81,664
9	59,050	314,946
10	196,831	1,181,000
15	71,744,536	669,615,690
20	23,245,229,341	294,439,571,680

As illustrated above, it is often the case that the O-graph of a CP-net grows very fast when the sizes of the involved colour sets increase. However, in practice, it is fortunately often sufficient to consider rather small colour sets in order to verify the logical correctness of a given CP-net. Having convinced ourselves that the data base system has the correct behaviour for 4 or 5 managers, we can feel pretty sure that the design also works correctly for any larger number of managers. Sadly, a similar statement is not true when we try to evaluate the performance of a given system.