Distributed Data Base

Abstract
This is a small toy example which describes the communication between a set of data base managers in a distributed system. The managers are supposed to keep their data bases identical. Hence, each update must be followed by a broadcast to all the other managers, asking them to perform a similar update.

The CPN ML declarations are described in great deal. Moreover, the example is used to illustrate three of the very basic concepts of net theory: concurrency, conflict and causal dependency.

The example is taken from Sect. 1.3 of Vol. 1 of the CPN book.

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

Graphical Quality
The figures in this document are inserted via PICT format. This is why some of the arcs and place borders look a bit ragged. A postscript printout from Design/CPN (and the screen image in Design/CPN) has much higher graphical quality.

CPN Model

This example describes a very simple distributed data base with n different sites (n is a positive integer, which is assumed to be greater than or equal to 3). Each site contains a copy of all data and this copy is handled by a local **data base manager**. Thus we have a set of data base managers:

$$DBM = \{d_1, d_2, \ldots, d_n\}.$$

Each manager is allowed to make an update to its own copy of the data base – but then it must send a **message** to all the other managers (so that they can perform the same update on their copy of the data base). In this example we are not interested in the content of the message – but only in the header information, which describes the **sender** and the **receiver**. Thus we have the following set of messages:

$$MES = \{(s,r) \mid s,r \in DBM \wedge s \neq r\}$$

where the sender s and the receiver r are two different data base managers. When a data base manager s makes an update, it must send a message to all other managers, i.e., the following messages:

$$Mes(s) = \sum_{r \in DBM - \{s\}} 1`(s,r)$$

where the summation indicates that we form a multi-set, with n–1 elements, by adding the multi-sets $\{1`(s,r) \mid r \in DBM - \{s\}\}$, each of which contains a single element. The resulting multi-set contains one appearance of each message which has s as sender.

Together these definitions give us the following declarations:

constants: `n : integer (* n≥3 *);`

colour sets:
```
DBM = {d1, d2, ..., dn};
MES = {(s,r) | s,r ∈ DBM ∧ s≠r};
E = {e};
```

functions: $Mes(s) = \sum_{r \in DBM - \{s\}} 1`(s,r);$

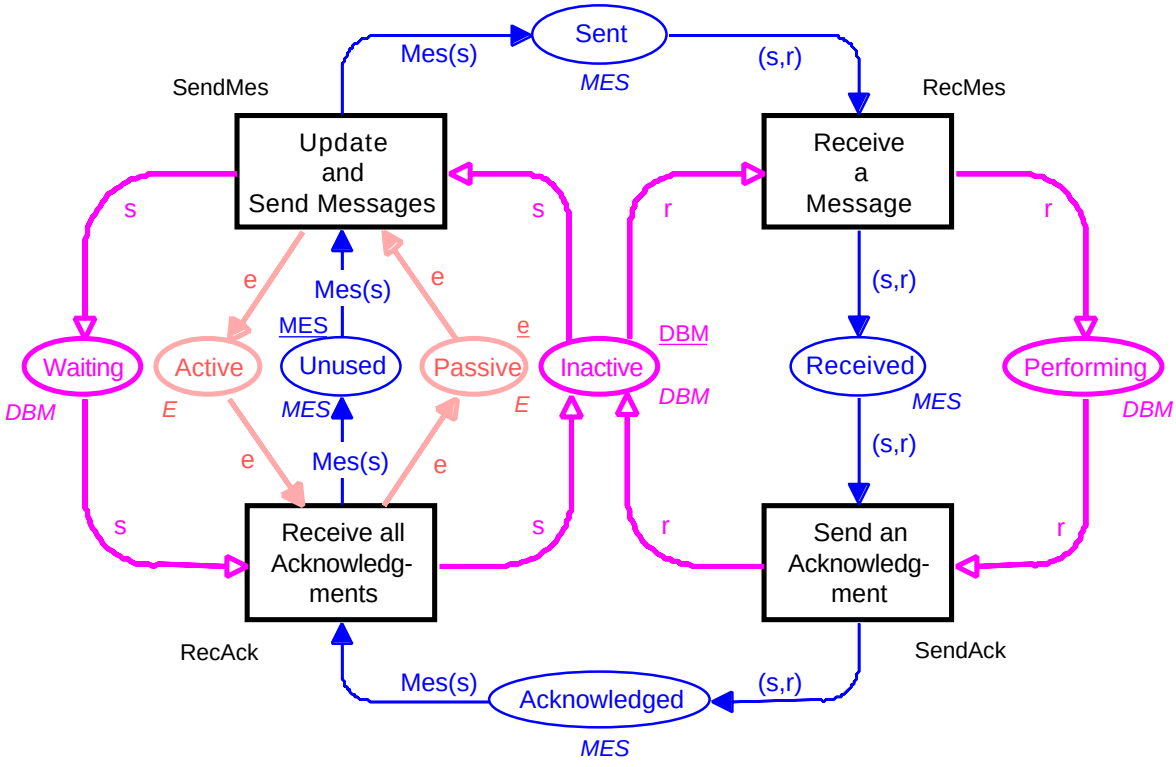
variables: `s,r : DBM;`

Now let us look at the CP-net (shown at the next page). Each data base manager has three different states: *Inactive, Waiting* (for acknowledgments) and *Performing* (an update requested by another manager). Each message can be in four different states: *Unused, Sent, Received* and *Acknowledged*. Finally, the system can be either *Active* or *Passive*.

Initially all managers are *Inactive* and all messages are *Unused.* This is indicated by the initialization expressions. DBM denotes the multi-set which contains exactly one appearance of each colour in the colour set DBM. Analogously, MES denotes the multi-set which contains exactly one appearance of each colour in the colour set MES.

When a manager, s, decides to make an *Update and Send Messages*, its state changes from *Inactive* to *Waiting*, while the state of its messages Mes(s) changes from *Unused* to *Sent*. Now the manager has to wait until all other managers have acknowledged the update. When one of these other managers, r, *Receives a Message*, its state changes from *Inactive* to *Performing* (the update), while the state of the corresponding message (s,r) changes from *Sent* to *Received*. Next the data base manager r may *Send an Acknowledgment* (saying that it has finished the update) and its state changes from *Performing* back to *Inactive*, while the state of the message (s,r) changes from *Received* to *Acknowledged*. When all the messages Mes(s), which were sent by the manager s, have been *Acknowledged*, the manager s may *Receive all Acknowledgments* and its state changes from *Waiting* back to *Inactive*, while the state of its messages Mes(s) changes from *Acknowledged* back to *Unused*.

To ensure consistency between the different copies of the data base this simple synchronisation scheme only allows one update at a time. In other words, when a manager has initiated an update, this has to be performed by all the managers before another update can be initiated. This mutual exclusion is guaranteed by the place *Passive* (which is marked when the system is passive). Initially *Passive* contains a




```
val n = 4;
color DBM = index d with 1..n  declare ms;
color PR = product DBM * DBM  declare mult;
fun diff(x,y) = (x<>y);
color MES = subset PR by diff  declare ms;
color E = with e;
fun Mes(s) = mult'PR(1`s,DBM-1`s);
var s, r : DBM;
```

single e-token (as for the resource allocation system, we use e to denote “uncoloured tokens”, i.e., tokens with no information attached).

It should be remarked that our description of the data base system is very high-level (and unrealistic) – in the sense that the mutual exclusion is described by means of a global mechanism (the place *Passive*). To implement the data base system on distributed hardware, the mutual exclusion must be handled by means of a “distributed mechanism”. Such an implementation could be described by a more detailed CP-net – which also could model how to handle “loss of messages” and “disabled sites”.

It should also be noted that the CP-net description of the data base system has several redundant places – which could be omitted without changing the behaviour (i.e., the possible sequences of steps). As an example, we can omit *Unused.* Then there will only be an explicit representation of those messages which currently are in use (i.e., in one of the states *Sent, Received* or *Acknowledged).* We can also omit *Active* and *Performing.* It is very common to have redundant places in a CP-net, and this often makes the description easier to understand – because it gives a more detailed and more comprehensive description of the different states of the system.

The data base system can be used to illustrate three of the very basic concepts of Petri nets: concurrency, conflict and causal dependency. In the initial marking of the data base system, the transition *Update and Send Messages* is enabled for all managers, but it can only occur for one manager at a time (due to the single e-token on *Passive*). This situation is called a **conflict** (because the binding elements are individually enabled, but not concurrently enabled) – and we say that the transition *Update and Send Messages* is in **conflict with itself**.

When the transition *Update and Send Messages* has occurred for some manager s, the transition *Receive a Message* is concurrently enabled for all managers different from s. This situation is called **concurrency** – and we say that the transition *Receive a Message* is **concurrent to itself**.

The transition *Receive all Acknowledgments* is only enabled when the transition *Update and Send Messages* has occurred for some manager s, and the transitions *Receive a Message* and *Send an Acknowledgment* have occurred for all managers different from s. This situation is called **causal dependency** (because we have a binding element which can only be enabled after the occurrence of certain other binding elements).

We have drawn the places and arcs in three different ways. This has no formal meaning, but it makes the net more readable to humans – because it makes it easier to distinguish between: state changes of the data base managers (thick lines), state changes of the messages (thin lines), and the mechanism to ensure the mutual exclusion (shaded lines). The distinction may also help to make the model more consistent (because it helps the modeller to see whether he has covered all parts of the system).

Now let us explain why the CPN ML declarations (in the dashed box at the bottom of the CP-net) are equivalent to the declarations outlined, at the beginning of this section.

Line 1 declares n to be a constant, and gives it the value 4. Constants in CP-nets are used in a way which is similar to that of programming languages. This means

that we can change the number of data base managers by changing the declaration of n (instead of having to change a large number of declarations and net inscriptions).

Line 2 declares the colour set DBM. This is done by means of a built-in colour set constructor which makes it is easy to declare **indexed colour sets**, i.e., sets on the form $\{x_i, x_{i+1}, \ldots, x_{k-1}, x_k\}$ where x is an identifier (i.e., a text string), while i and k are two integers (specified by means of two integer expressions). CPN ML does not recognise different font styles and thus x_r is written as x(r) or as x r (where the space after x is significant). By adding "declare ms" (where ms stands for multi-set) we instruct the CPN ML compiler to declare a constant multi-set, having a single appearance of each element in DBM. This constant is predefined, but to save space on the ML heap it is only declared when the user asks for it. The constant is denoted by ms'DBM or simply by DBM, and it is used in the initialization expression of *Inactive*. We could also have declared DBM by means of an enumeration colour set, but then it would have been impossible to make the declaration independent of the actual value of n.

Lines 3–5 declare the colour set MES. First we declare a colour set PR which is the cartesian product of DBM with itself. By adding "declare mult" we instruct the CPN ML compiler to declare a function mult'PR by which we can multiply two DBM multi-sets with each other in order to get a PR multi-set. This function is predefined, but to save space on the ML heap it is only declared when the user asks for it. As an example, we have mult'PR($2`d_3+1`d_4$, $1`d_2+3`d_3$) = $2`(d_3,d_2)+6`(d_3,d_3)+1`(d_4,d_2)+3`(d_4,d_3)$. The function mult'PR is used in line 7 of the declarations (see below). Next we declare a function diff. The function takes two arguments and tests whether these are different from each other (<> means ≠). Finally we declare MES to be the subset of PR which contains exactly those elements for which diff(s,r) is true. By adding "declare ms" we get a constant multi-set, denoted by ms'MES or simply by MES – and this is used in the initialization expression of *Unused.*

Line 6 declares the colour set E, which is used in a similar way as in the resource allocation system.

Line 7 declares the function Mes mapping data base managers into multi-sets of messages. This is done by means of the predefined multiplication function mult'PR declared in line 3. The minus sign denotes multi-set subtraction.

Finally, line 8 declares the two variables s and r of type DBM.