Modelling and Simulation of a Network Management System using Hierarchical Coloured Petri Nets. Extended version

Søren Christensen
Computer Science Department
Aarhus University
Ny Munkegade, Bldg. 540
DK-8000 Århus C
Denmark

Leif Obel Jepsen
RC International
Klamsagervej 19
DK-8230 Åbyhøj
Denmark

April 1991

Abstract

Development of distributed software systems is a complex task. This paper argues that design and specification can be supported by modelling and simulation using Hierarchical Coloured Petri Nets (CP-nets). This conclusion is based on a case study of a project in which CP-nets were used in the detailed design of a software module. The software module is part of the Network Management System of the RcPAX X.25 wide area network.

The module was designed using the Design/CPN tool which allows editing and simulation of CP-nets. Furthermore invariant techniques were used to prove properties of the module.

Introduction

High Level Petri Nets have been used to model many different kinds of concurrent systems, a number of examples can be found in: [Shapiro 1990] hardware; [Hartung 1988] software; [Chehaibar 1990] protocols and [Shapiro, Pinci and Mameli 1990] formal behaviour of people — often called Command and Control systems. The main purpose of these models has been to analyze the behaviour of existing systems.

The main conclusion of this paper is that modelling and simulation by means of CP-nets can be used as an integrated part of the design phases of the development of distributed software systems. Using CP-nets as a specification technique allows the developer to simulate the design proposals very early in the project, hereby discovering the possible advantages and disadvantages of the proposals. Increasing the understanding of the dynamics of the system early in a project will enable the designer to discover logical design flaws before the actual coding starts. These statements are based on experiences using the Design/CPN tool [Albert, Jensen and Shapiro 1989] to model and simulate a software module during the design phase.

Since prior knowledge of neither CP-nets nor Network Management Systems is assumed, we will introduce the concepts necessary to understand the problem area and the presented model. Readers who have a prior knowledge of CP-nets or Network Management Systems can skip the introductory sections on these subjects.

The project was part of a large software development project carried out by RC International. The aim of the total project was to develop the RcPAX X.25 wide area network to provide the International X.25 Infrastructure Service (IXI Service) [Popper 1990]. IXI is the first major activity in the implementation of the Eureka COSINE Project. The IXI network spans 19 countries in Europe and connects 20 private and 11 public X.25 networks.

The RcPAX network consists of a number of network nodes handling access traffic tolfrom users of the network and transit traffic internally in the network. The RcPAX Network Management System (NMS) enables operators at the Network Management Centre (NMC) to monitor and control all modules of the total network. The NMS is a distributed application which is an integrated part of all network nodes. The local network management

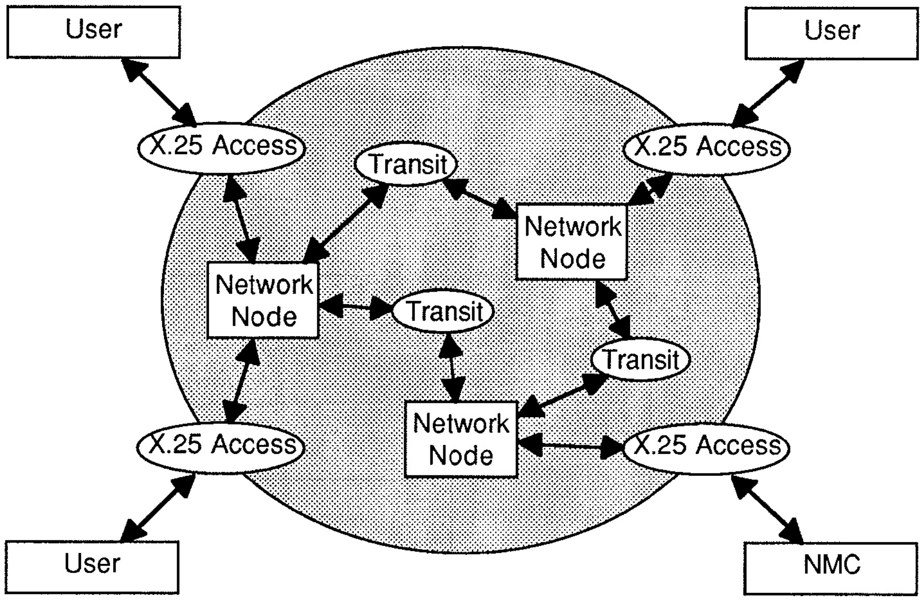

Figure 1: The Network Management Centre monitor and control all modules of the network nodes.

system will monitor and control the individual modules in the network node. Figure 1 shows an overview of the network.

The rest of the paper contains an introduction to the architecture of the network nodes and the concepts used in CP-nets. After the introductory sections we discuss how modelling, simulation and formal verification using CP-nets were used in the project. Finally we conclude by discussing benefits and problems in our approach.

A Network Node

Each network node needs to be able to communicate with the NMC. To facilitate this communication the network nodes have a software module which handles the communication between the NMC and the different software modules local to the network node. At the basic level, the NMS works by means of three types of information: the NMC can issue a *request* to a specific software module, the request will trigger an *answer* send back to the NMC and finally information on *events* at the network nodes will be sent to the NMC.

The structure of the network management part of the individual network nodes is shown in figure 2.

In our case the network node is a single machine which consists of a number of boards: one management board and a number of transputer boards. The transputer boards are running all software responsible for access and transit traffic. The local management system represents each software module as a Local Control Probe (LCP). The management board is running the Network Control Probe (NCP) which is responsible for the communication to the NMC. The function of the Sub NCP (SNCP) and the LCP Adaptor is to connect the LCPs with the NCP across the local bus called the Linkbus.

In the development project CP-nets were used in the design of the SNCP module. A detailed description of the behaviour of the SNCP was made. More rudimentary descriptions of the behaviour of the LCPs and the LCP Adaptor were added as an environment for the simulation of the behaviour of the SNCP.

Hierarchical Coloured Petri Nets

This section will introduce the concepts used in Hierarchical Coloured Petri Nets. We focus on the concepts needed to understand the model of the SNCP. For a more detailed introduction see [Jensen 1990].

Structure of CP-nets

The structure of all kinds of *Petri Nets* consists of three basic elements called *places*, *transitions* and *arcs*. Places represent states and a place is usually drawn as an ellipse. Transitions represent actions and are usually drawn as boxes. An arc represents a relation between a place and a transition, arcs are usually drawn as directed connectors.

The current state of a Petri Nets is represented as a distribution of

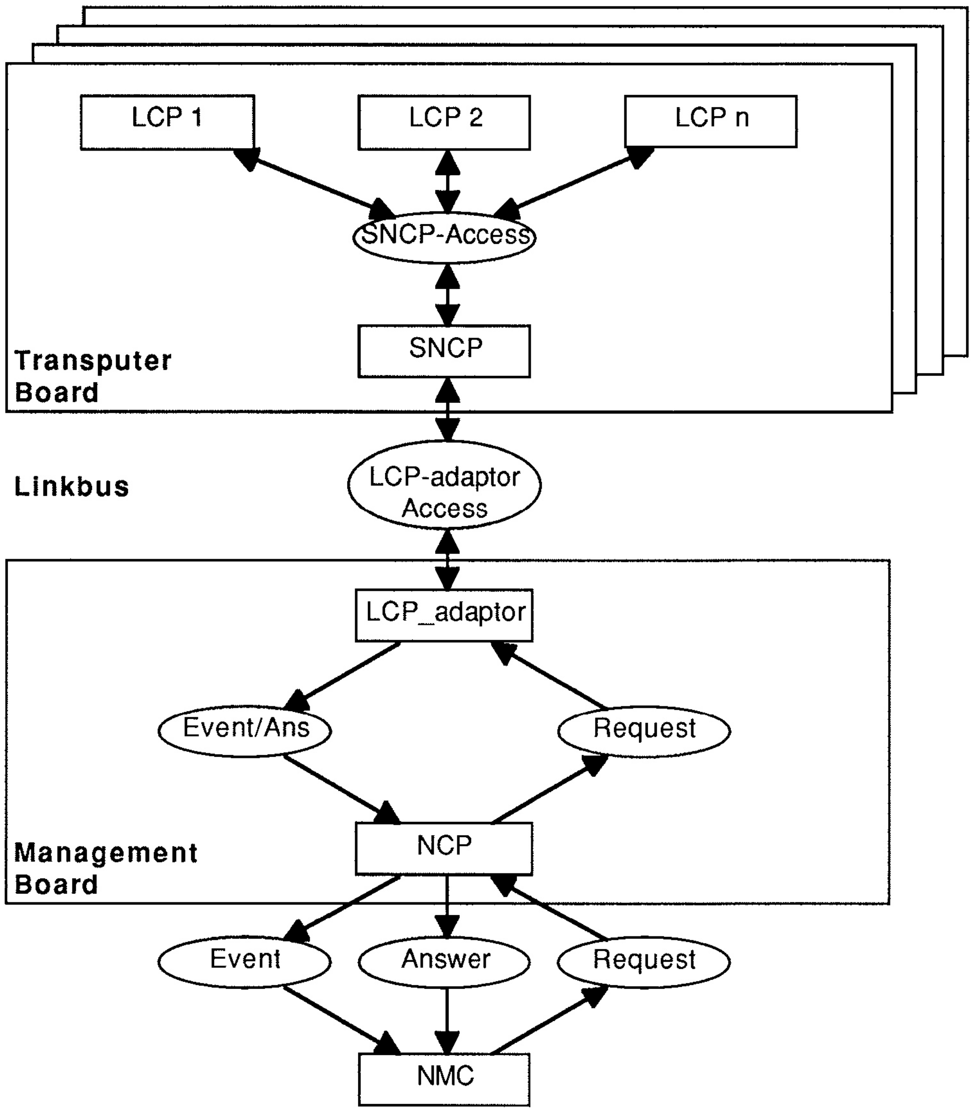

Figure 2: The structure of the Network Management System of a single network node.

tokens to the places. The dynamics of the model is represented as transitions removing or adding tokens to places. An arc directed from a place to a transition is called an input-arc. Input arcs represent restrictions on the transition — the transition will only be *enabled* if all places of input-arcs for the transition contain enough tokens. An arc directed from a transition to a place is called an output-arc. When a transition *occurs* tokens are removed from the places of input-arcs and tokens are added to the places of output-arcs.

The initial state of the model is specified by attaching an *initial-marking* to places. Usually we omit the empty initial markings.

The graphical layout of nets does not have a formal semantics. This means that the modeller can create a layout which will increase the readability of the net, without changing the semantics of the model.

Inscriptions of CP-nets

In *Coloured* Petri Nets the ability of attaching information — called *a colour* — to the tokens allows a more compact description. Instead of having a place for each value of the tokens we now have a single place which can carry all tokens of a specific colour-set. The *initial-marking* of a place now specifies a number of coloured tokens.

Having tokens which carry values — or information — makes it possible to perform more elaborated actions. This is done by attaching expressions to the arcs, called *arc-expressions*. The arc-expressions can contain *variables*. A transition is enabled if it is possible to *bind* the variables of the surrounding arc-expressions to values in such a way that the arc-expressions of all input arcs evaluate to tokens which are present at the corresponding input places. Furthermore we demand that a Boolean expression called the *guard-expression* of the transition must evaluate to true. If we omit the guard-expression, this is a short cut for the guard-expression which always evaluates to true.

When a transition occurs with a given binding, the tokens specified by the evaluation of the arc-expressions of the input arcs are removed from the corresponding input places, and the tokens specified by the evaluation of the arc-expressions of the output arcs are added to the corresponding output places.

Hierarchies of CP-nets

Models of systems using Coloured Petri Nets can still be very large and it can be convenient to have a number of formally related and smaller nets instead of a single large net. In [Huber, Jensen and Sharpiro 1990] a number of different hierarchical relations are defined, here we will introduce the two we have used in our modelling.

The main kinds of hierarchical relations that we used in the project is called *transition-substitution.* The action corresponding to an ordinary transition is specified by the arcs and by the guard, but for a *substitution transition* the corresponding action is specified by a separate *sub page.* The places surrounding the substitution transition are called socket places. Each socket place will be assigned to a port place of the sub page.

We also use the concept of *place fusion*: if a set of places belong to the same *fusion set* they will share the same tokens. This means that if a transition removes a token from one of the places in the fusion set, it will be removed from all other places in the same fusion set.

In figure 3 we show how the next operation to be performed is selected by the SNCP module. The graphical layout of the net has no formal semantics, we use it as a way of increasing the readability, in this example we use bold arcs to indicate the control structure, and plain arcs to indicate data access.

The page contains six places which each has a colour set attached. The colour sets correspond to the concept of type in programming languages. Some of the colour sets are simple values such as **E** and **BOOL**, others are more complex like **BUF** and **LBUF**. In the figure we have included the declarations of the colour sets to illustrate how these could look. Normally the declarations would not be distributed in the net, but collected in a set of declarations for the total net. Only Id16_mbx_waiting has a non-empty initial marking, the initial value of this place is a token representing an empty list denoted [ ]. Three of the places have been defined as port places, marked with a B-tag. Each of the port places has a specific relationship to the environment. Idl6Finished is an input port place which means that it can be assigned to an input place of the substitution transition related to this sub-page. Ready is an output port place and main-mbx is an inputloutput port place. Furthermore Idl6-mbx-waiting is a fusion place, which means that the

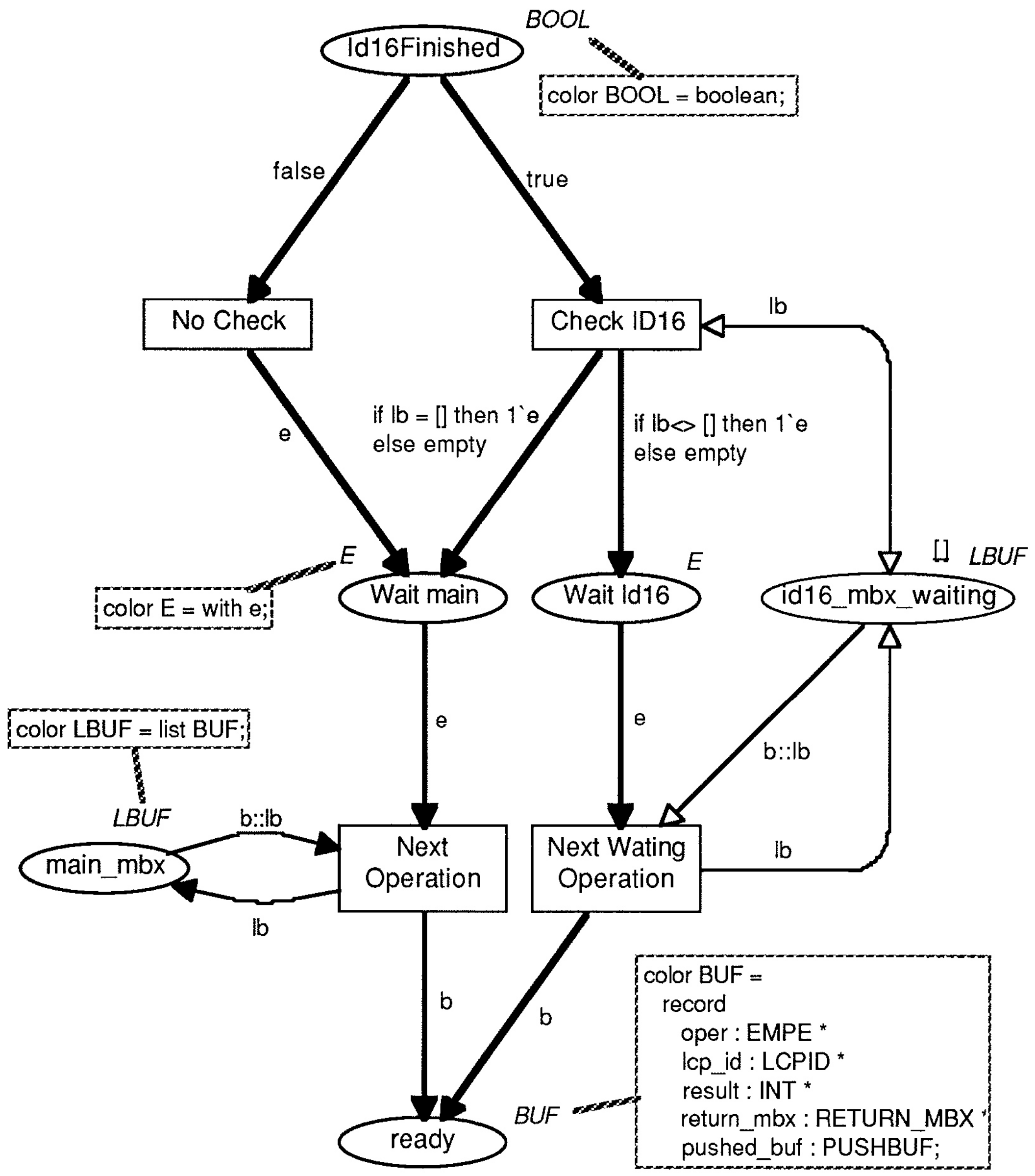

Figure 3: Detailed subnet specifying how the next operation is selected in the SNCP module.

tokens in this place will be shared with other places — on other pages — in the model.

The page also contains four transitions which each has one or more input arcs and output arcs. The bi-directed arc is an abbreviation for an input arc and an output arc having the same arc-expression. The arc expressions contain constants, variables, operations and functions. The constants are: true, false, e, empty (the empty set) and [ ] (the empty list). The variables are b and lb. The operations are :: (constructs a list from an element and a list). 1`e the set of one e token.

When a token arrives at the Idl6Finished place, the colour of it specifies if the internal mailbox called Id16_mbx_waiting should be checked or not. The check is done by the transition CheckId16 which inspects the list, and only if it contains at least one element we try to read it. In contrast to this we do not check the main_mbx before starting a read. The reason for this difference is that we want to model the fact that the system will wait until a new buffer arrives. The result will be a buffer which contains the new operation at the place ready.

The Project

The aim of the project in which we used CP-nets was to implement the SNCP module. This task was accomplished in four phases using the following amount of time (person-weeks).

Analysis:	4
Design:	2 + 2
Coding:	2
Testing:	2

The analysis phase produced a conventional textual specification of the SNCP Access Protocol in which the LCPs are acting as users of the SNCP services. The SNCP services were specified as a number of functions with layout of the buffers needed for communication in the protocol. We call these buffers for protocol elements. The services included *connect, disconnect, send_event, receive_request* and *send_answer.* A textual specification of the LCP-Adaptor Access Protocol was made as a part of another sub-project.

For each service in the protocols the typical use was illustrated in an arrow diagram as the one shown in figure 4.

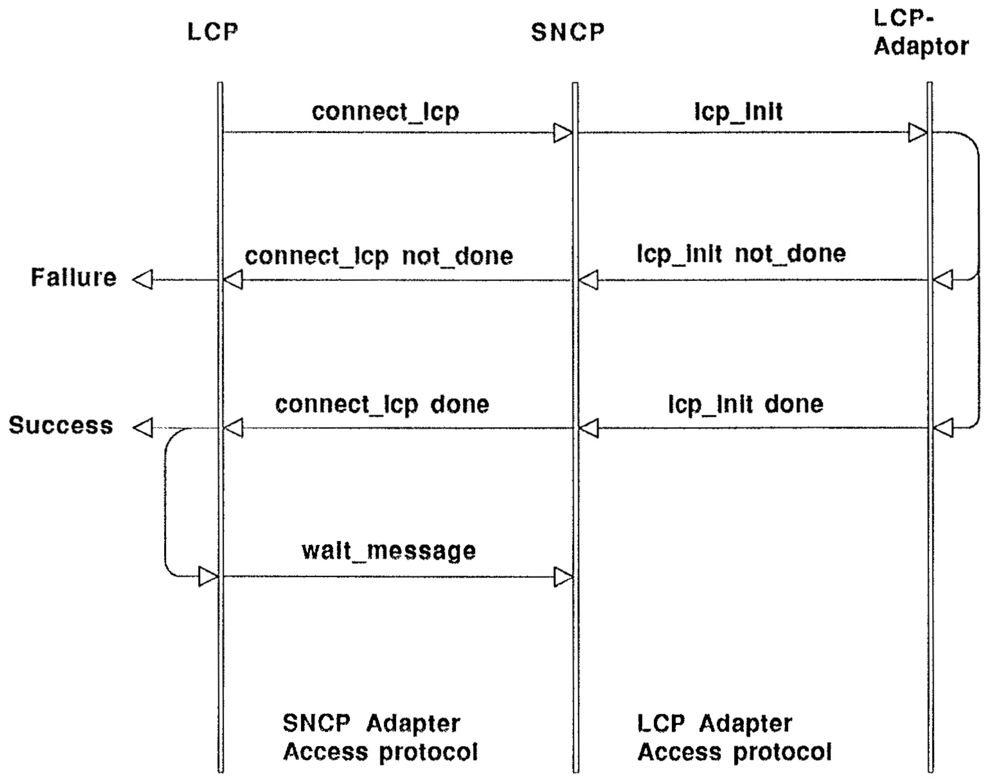

Figure 4: Specification of a typical sequence of messages.

When an LCP is created it must inform the NMC. In figure 4 it is shown how this is done. First the LCP sends a *connect_lcp* to the SNCP which will send an *lcp_init* to the LCP-Adaptor. The reply from the LCP-Adaptor with either an *lcp_init not_done* or an *lcp_init* done. Finally the SNCP returns the *connect_lcp* including the corresponding result to the LCP. It is important to notice that arrow diagrams only specify a typical sequence of protocol events. If error handling should be documented, it would be necessary to create additional arrow diagrams. Since arrow diagrams only show examples of sequences of protocol events it is also hard to describe how concurrent events are handled, e.g. if a new request for the LCP is received before the *connect_lcp* answer is returned.

The task of the next phase was to design the SNCP module according to the specification of the SNCP and LCP Access protocols. Although the functions of the module were well understood, this was a complex task.

The main difficulties were:

- The SNCP should handle the communication with the LCP Adaptor on another CPU board including re-transmission of lost messages, acknowledges, etc.
- The SNCP should handle all LCPs and the LCP Adaptor in parallel.
- Error situations and corresponding actions should be identified.

The complexity of the problem implied a need to work on the control structure and the internal state of the module at the design level, without going into too much detail of the coding. This was the original motivation for the use of CP-nets.

The Design/CPN tool was used to develop the detailed design of the control structure and the internal state of the SNCP module. Furthermore a rudimentary description of the surrounding components was added to provide an environment making it possible to evaluate both the internal and external behaviour of the SNCP.

The project was carried out by one of the authors. Prior to the project he had no experience using CP-nets for modelling. The other author was an experienced user of Design/CPN. His primary task was to assist the modeller using the tool and to discuss how CP-nets could be used in the modelling of the system. Learning to use the tool was part of the design phase. We estimate the learning part of the design phase to be 2 of the 4 person weeks used.

The resulting model of the SNCP was used as a basis for the coding of the module. The implementation language was a variant of Pascal called Real Time Pascal (RTP). RTP includes facilities for concurrent processes, mailbox handling and buffers for communication between processes.

The coding and test phases followed the usual development procedure and the module is now a running component of the network nodes.

Modelling

The aim of the modelling phase was to design the control structure and internal status information of the SNCP module. The starting point of the design was the textual protocol specifications made in the analysis phase. The resulting model was the basis for the implementation. This meant that it was very important that the constructs used in the model could be realised in the language used for the implementation and that the structure of the model could easily be mapped to the structure of the program. The model was also used to evaluate the design proposal. An experienced developer without knowledge of CP-nets had less than an hour of informal introduction. After this he was able to understand the CP-net model and to give qualified feedback in the form of proposals for changes to the model.

Top Down Modelling — Hierarchical Decomposition

When you develop a model of a complex system it is necessary to be able to focus your attention on different aspects of the system as the modelling progresses.

In our case we modelled the system in a Top-Down way. The top level illustrates the different hardware components and how they are interconnected. Figure 2 showed the structure of a single network node and figure 5 shows how this was actually done in the CP-net model.

If we compare figure 2 and figure 5 we notice some differences: The LCP-adaptor Access state has been expanded to include an active component, called the Linkbus. This was necessary since we would like to be able to model errors which could occur in the transmission between the management board and the transputer boards. We did not need to distinguish between the LCP_adaptor and the NMC in the modelling of the environment for the SNCP, thus we have described the management board as a single component. We have modelled all the LCPs as having the same internal structure. Therefore we had to add a place holding the identity of the individual LCPs.

The rest of the structure is due to the fact that mailboxes had to be handled explicitly in the model.

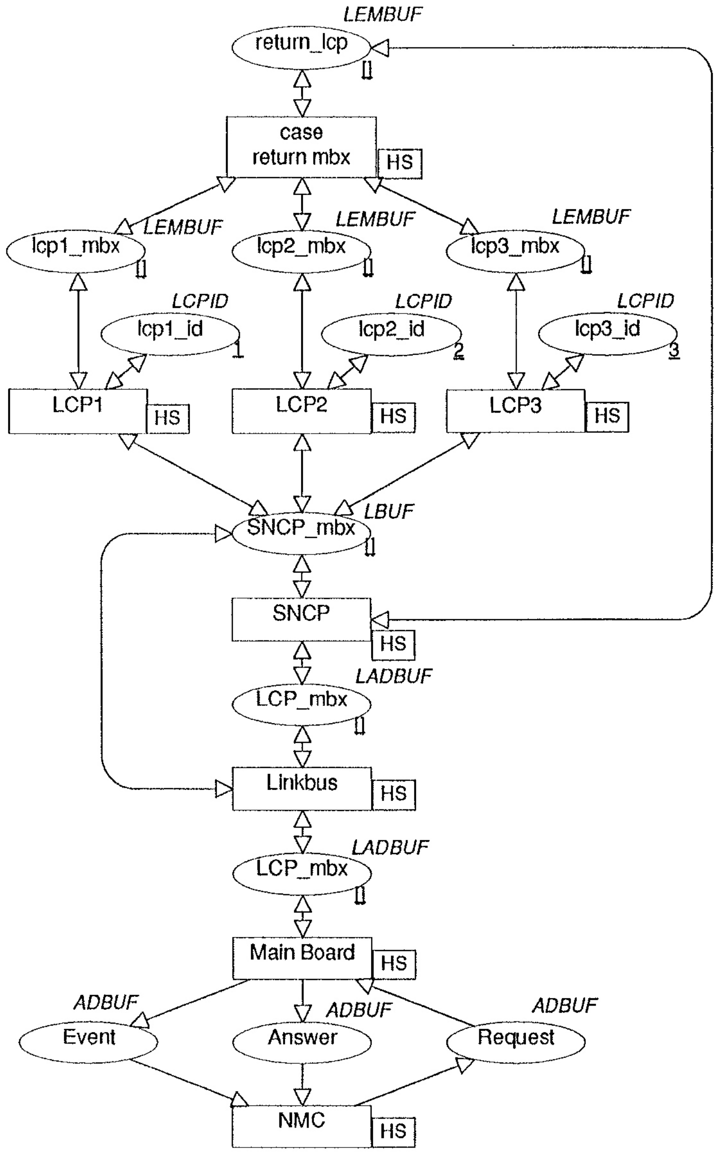

Figure 5: The top level in the CP-net model (main#1).

From the start of the modelling activities we knew that mail-boxes and buffers were important for the implementation. The implementation language contains a number of high-level constructs to handle communication

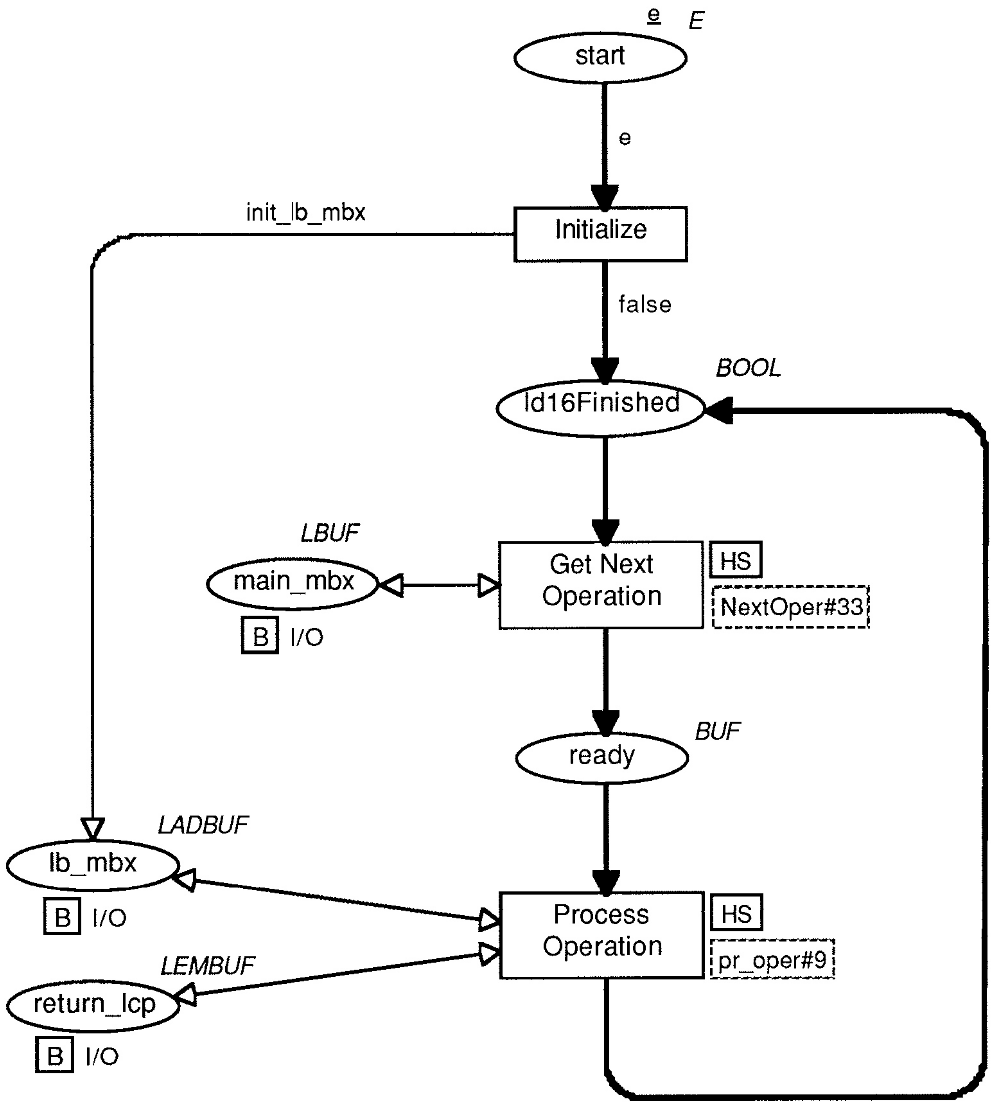

Figure 6: First level of decomposition of the SNCP (sncp#7).

between processes using mail-boxes and buffers.

Since our main interest was to design the SNCP module, we started out by defining the first level of decomposition of this module. In figure 6 this level is shown, here we see that it contains one ordinary transition called

Initialize and two substitution transitions called Get Next Operation and Process Operation. The arcs surrounding the substitution transitions are only used to define which places are sockets for the sub-page. The sub-page of the Get Next Operation was shown in figure 3. To increase the readability of the net we have given corresponding port places and socket places the same names. If you develop your model top-down, the system will automatically support this naming scheme. Figure 6 provides an overview of the overall control structure while all information on state variable is hidden inside the lower levels of decomposition.

The sub-page of the transition called Process Operation contains one substitution transition for each operation in the SNCP module. This hierarchical decomposition allowed us to model and simulate each of the operations in full detail before the rest of the design was done. The first operation we designed was the connect_lcp. The experience from modelling and simulation of the connect_lcp operation was utilized in the design of the rest of the operations.

When you create a CP-net consisting of many pages, it can be hard to keep track of the relation between pages. The Design/CPN editor provides an overview of the pages and the relations between pages. The overview of a model is represented as a page-hierarchy, see figure 7. In the page hierarchy each node represents a page in the diagram, the name and number of the corresponding page are shown in the nodes. The relations between pages which are created by use of transition substitution are represented by a page connector. The page that is the source of the connector contains a substitution transition which has the destination of the connector as a sub-page.

We have designed the model in such a way that each page has been implemented as a separate process or procedure.

Representation of Protocol Elements

From the initial specification of the SNCP access protocol and the LCP-Adaptor access protocol it was known what information the protocol elements should contain. This information could be used in the design of the module: Each of the basic types in the protocol elements was modelled by a corre-

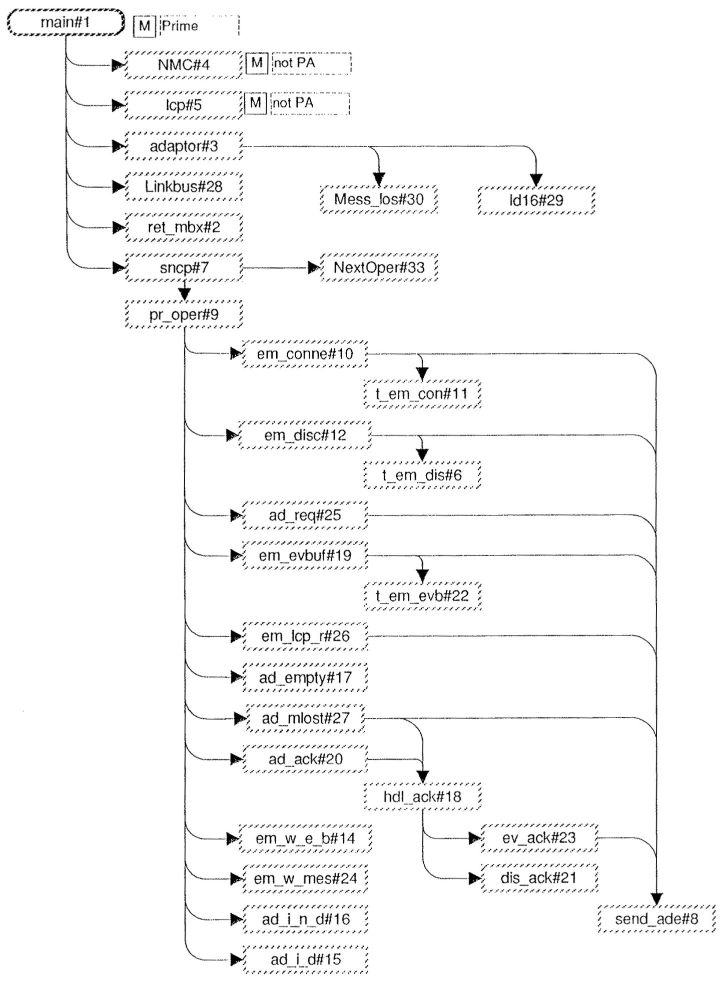

Figure 7: The Page Hierarchy of the CP-net model.

sponding colour declaration and then all protocol elements were constructed from these.

In the textual protocol specification we get a fixed layout of the data independent of the implementation language. In the CP-net model we have abstracted away from the actual representation on the machine level. All protocol elements are specified as a common record type having a field which specifies the operation contained in the buffer.

Representation of Internal State Information

It was important that the CP-net model could be used to analyse the data needed to implement the SNCP module and to what extent this should be local or global. From the start of the modelling we expected to store relative complex information on the internal state of the individual LCPs but during the modelling it turned out that much less information was needed. Thus the explicit modelling of internal state information led to a simpler representation in the final program.

Level of Detail

Most of the concepts from the implementation language can be modelled directly using CP-nets, but for the mailbox handling we had to include this explicitly in our model, to have a model of the behaviour of the resulting system.

```
IF idl6.finished THEN
   IF open (idl6.waiting_em)
   THEN
        wait (main_msg,idl6.waiting_em)
   ELSE
        wait (main_msq,main_mbx^)
ELSE
   wait (main_m sg,main_mbx^);
```

Figure 8: Actual implementation of Get Next Operation.

It was important that the model could be used as a specification of how the actual implementation should be done. To illustrate that it could be done, figure 8 shows the code from the implementation which corresponds to the sub-net in figure 3.

It may be noticed that each statement in the implementation corresponds to a set of transitions, but it is also important to notice that the code is not just a simple translation from the CP-net model — it still requires refinement of the specification.

When you model a system you have to abstract a number of details away. It is important to be aware of what you do not model and it is often hard to find an appropriate level of detail in the model. In our case one of the details we abstracted away was the concrete representation of buffers and data in the implementation. This meant that we did not model the different aspects of data conversion — between the data representation of the transputer boards and the management board.

We could abstract away from these details without losing information necessary for designing the program structure and the internal state of the SNCP module.

Even though the implementation language placed strong restrictions on the concepts used in the CP-net model it was without major problems to create the model. The main problem was due to the explicit modelling of the mailboxes.

It is important to be able to evaluate the dynamic behaviour of a model early in the modelling process. In the next section we describe how we used simulation to gain insight in the dynamics of the CP-model.

Simulation

Creating a model gives a lot of insight into the structure of the system, in particular with models like CP-nets where you have the possibility of checking the consistency of the model. But it is hard to gain information about the dynamic behaviour of the system without simulating.

Simulation gives the same possibilities as early prototypes in the sense that it provides the possibility of testing the behaviour of the system, before making an expensive and time consuming implementation. In the project it would be very difficult to use prototyping as the hardware used was special purpose machines with a very limited possibility of interacting with a test environment.

Modelling of the environment

It is important that a model is prepared for simulation and it is often necessary to include parts of the environment in the model to give the possibility of interacting with the model. This also gives an explicit representation of the assumptions of the behaviour of the environment.

In our case we included a rudimentary description of the LCPs and the LCP-Adaptor/NMC. It is essential that the description of the environment can be much more rudimentary and fragmented than the rest of the model — otherwise it would not be possible to simulate models as part of a complex environment.

Simulation as Debugging

We think of simulation as a way of debugging a model. The main similarity between simulation and debugging is the focus on the dynamic aspects of the system. Simulation and debugging are not formal verification methods in the sense that you can prove properties of a system to hold for all possible states of the system, but they are very powerful ways of gaining insight in the dynamics of the system and getting more correct models.

A good simulator should provide the same range of possibilities as a good debugger. This means that you should be able to:

- Inspect the model while simulating, both information on the current state and possible future actions.
- Single step and decide in all situations where choice is possible and resolve all conflicts in this manner.
- Automatic run where conflicts are resolved by random choice.
- Execute until a specified situation is reached.

It is possible to use the graphical layout of a CP-net directly in a simulation. The state of the system can be shown as current marking of places. The possible events can be shown as enabled transitions, and the dynamics as tokens being removed or added to the current markings. This is in contrast to simulation/debugging of a textual programming language in which an execution cannot be understood in a local execution of the instructions, but often needs to include knowledge about the state of the total store.

The Basic step of a Simulation

Figures 9–11 show a part of a net being simulated. The current marking of a place is shown as a small circle showing the number of tokens which reside on the place and the text next to it shows the actual colour of the token. If the current marking is empty nothing is shown. The graphical layout of these tokens is controlled by options which allow the user to set up a number of defaults, e.g. position, size and shape. These defaults can always be overwritten for the individual objects. In figure 9 the Wait main place has the current marking of one e token and the main_mbx has one empty list as the marking. This means that the transition Next Operation cannot occur.

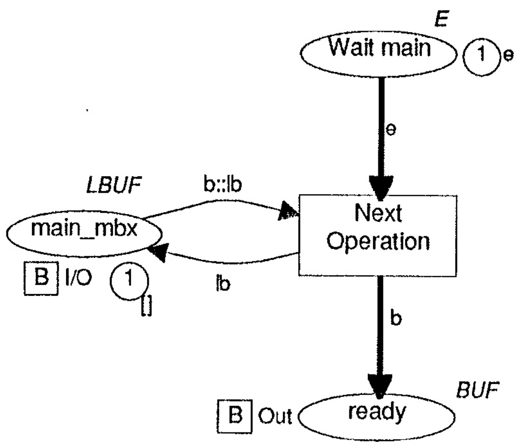

Figure 9: The process is waiting at the Wait main, but the main_mbx is empty.

Next Operation will not be enabled until the main-mbx is updated. When the transition occurs all variables surrounding it must be bound to values in such a way that all input arc-expressions evaluate to tokens which are present in the current marking of the corresponding places, and if the transition have a guard this must also evaluate to true.

In figure 10 it is shown how Next Operation is occurring with the variable **b** bound to the value of the buffer in the main_mbx and **lb** bound to the empty list.

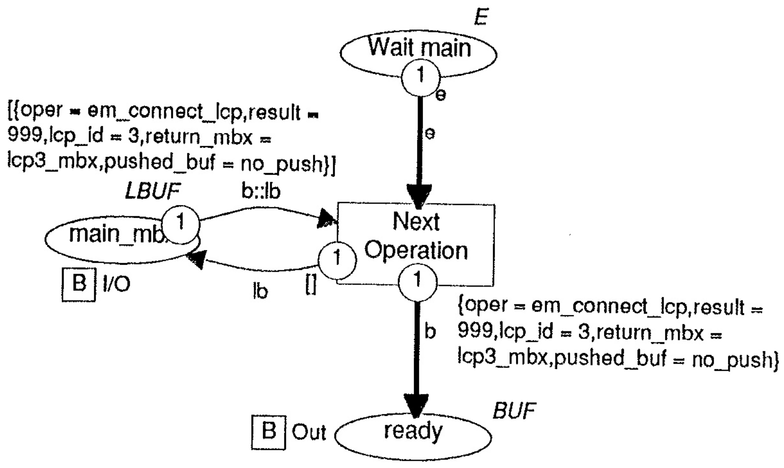
E
Wait main
1
e
e
[{oper = em_connect_lcp,result = 999,lcp_id = 3,return_mbx = lcp3_mbx,pushed_buf = no_push}]
LBUF
b::lb
main_mbx
1
Next Operation
1
B I/O
lb
[]
1
{oper = em_connect_lcp,result = 999,lcp_id = 3,return_mbx = lcp3_mbx,pushed_buf = no_push}
b
BUF
B Out
ready

Figure 10: Next Operation is occurring and we can inspect the tokens being moved along the different arcs.

The tokens on the arcs indicate the tokens being moved along this arc, the number in the circle shows the number of tokens and the text gives the actual colour of the tokens, these colours are the evaluation of the arc-expressions with the specified values of **b** and **lb**. The occurrence of the transition will then update the surrounding markings and the new current markings can be calculated from the previous ones by subtracting the token of the input arcs and adding the token of the output arcs. The means that the situation will look like figure 11 after the occurrence of the transition.

During a simulation a number of choices have to be made: In a given situation more transitions can be enabled. If these are in conflict – i.e. compete for the same input tokens – it must be decided which one should occur. If they can occur concurrently it must be decided whether this should happen or whether some of the enabled transitions should be postponed. In the next section we will discuss different ways of making these decisions.

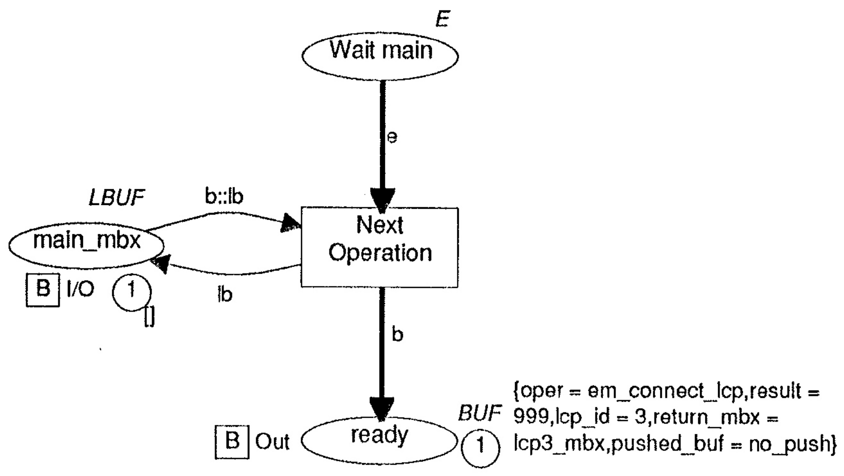
E
Wait main
e
LBUF
b::lb
main_mbx
Next Operation
B I/O
1
[]
lb
b
{oper = em_connect_lcp,result = 999,lcp_id = 3,return_mbx = lcp3_mbx,pushed_buf = no_push}
BUF
B Out
ready
1

Figure 11: After Next Operation has been performed the main_mbx is empty and the content of the received buffer in now at the place ready.

Modes of simulating

As the purpose of simulating a model changes as the modelling process progresses this implies changes in the way the simulation is performed. Early in the process the modeller often uses the simulations to test the behaviour of specific transitions and small sub-systems. Later the focus can change to include the inputloutput relation of the total model.

The Design-CPN simulator allows a range of modes of simulating the model. The basis of all simulations is the possibility of inspecting the current marking of all places and of directly observing which transitions are enabled. If the user wants explicitly to control all parts of the simulation he has to select an enabled transition and specify a binding of variables for this transition. This will include the transition in the set of transitions to occur in the next step. In the other extreme the user just starts an automatic sequence of simulation steps and then observes the simulation without interfering in the simulation. When you move to a more automated simulation you need a way of controlling this new level of automation. In the Design-CPN simulator this is done by setting a number of options. A simulation can be in a range from a total sequential simulation to a simulation where a maximal set of transitions will occur in the following step.

In the project, we used the simulator very early in the modelling process. In the beginning only a small sub-model was available and all simulation was done manually. The first sub-model included the initial situation where an LCP made a connect request to the NMC.

After more of the model had been made we started to simulate more automatically, just triggering an automatic run of our model by specifying an input from an LCP and/or from the NMC. Here we used the specification of typical sequences from the initial protocol specification.

During the simulations we discovered a number of different kinds of bugs in our model, and especially in the beginning we had to remodel parts of the system. The bugs that we discovered during the simulations can be divided into a number of categories including the following: Erroneous or insufficient tests before services were performed, buffers disappearing, the model not being specified in sufficiently detail or simply referring to missing parts of the model.

We did not use fully automatic simulations since this would require a much more detailed modelling of the environment and only be of little interest for the design of the interior of the SNCP module. In situations where the modeller focuses more on inputloutput relations of the model and less on the internal state and actions of the model, it could be useful to set up the simulation in a more automatic way, hiding the details of the simulation of the model. An example of this can be found in [Shapiro 1990] in which a model of a VLSI design was investigated by means of both manual and fully automatic simulation.

In the project we used simulation as a way of validating our design. This gave us very detailed insight in the dynamics of the module before the actual implementation was done. By simulation the modeller can gain insight in the dynamics of the model and hereby be able to discover and remove bugs in the model. Another reason why it is important to use simulation early in the project is that it allows the modeller to use the knowledge of the behaviour of the model in the further modelling process. Furthermore, in projects where both hardware and software are developed it is often impossible to evaluate prototypes until very late in the project. In these cases simulation can be of even more interest.

It is not possible to formally validate a model by simulation. To formally verify properties of the model it is necessary to apply other techniques. In the next section we describe how place-invariants have been used to verify properties of the model.

Formal Validation

Using a specification method with a formal semantics like CP-nets makes it possible to make formal validation of properties of the model. In this section we give some examples of the kind of formal analysis we have performed.

There exists a number of different ways to perform formal analysis of CP-nets, especially by means of occurrence-graphs and place-invariants. Without tool support it is only practically possible to perform place invariant analysis, so this is what we have done. For a discussion on the different analysis methods see [Jensen 1987]; [Jensen 1990].

When you define a place invariant you specify a weight-function for each place in the model and verify that the weight of the tokens removed by an occurrence of a transition is equal to the weight of the tokens produced. This means that the sum of all weighted tokens will be constant for all possible states of the model.

Properties of buffers

An important property of the management of the buffers in the system is that they do not disappear. This can be expressed directly as a place invariant. We define a function which counts the number of tokens residing on a place and attach this to all places containing buffers. We call this function Count. In our model we have allowed buffers to be "pushed" on top of each other. To calculate the right number of buffers we need to count buffers which have an additional buffer pushed as two buffers. For all mailboxes which were represented as a list we need to traverse the list to calculate the number of buffers in the mailbox. We call this function Length. For all other places we will attach a Zero function just specifying that these should not be taken into consideration.

After specifying the weight-functions for all places, we need to go through all transitions and check that they preserve the weighted-token count. In figure 12 it is shown how the invariant would look around a single transition. To validate transition Next Operation we must show that it preserves the weighted token count:

Weight of input tokens:

Length (b::lb) + **Zero** (e) =
Length (b::lb) + 0 =
Count (b) + **Length** (lb)

which is the weight of the output tokens.

In this way we have proved that the number of buffers in the model is constant, i.e. no buffers appear or disappear.

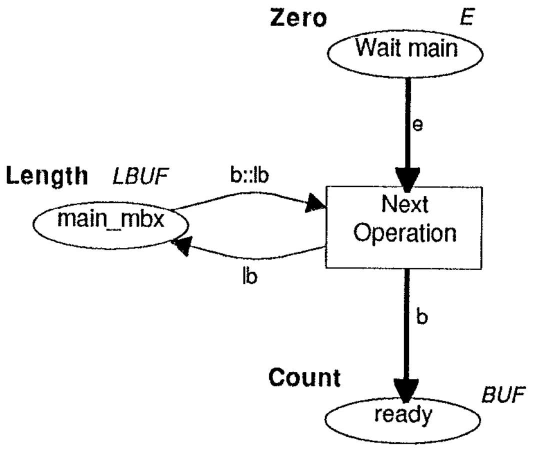

Figure 12: Weight functions attached to places. Occurrences of the transition will leave the total weight unchanged.

Deadlock free

Software modules must often fulfil the property of being free of deadlocks. This means that it is not possible to bring these into a state where no transition is enabled. In the case of the SNCP module the module should contain a deadlock: All activity in the module must cease if the environment does not produce any requests. We would like the module to stop in the state. The only possible action is to receive a request from the environment.

Place Invariants can also be used in the proof of absence of deadlock, but the arguments are a little different from the ones used in the previous section. We need to prove that all markings which lead to dead situations will violate an invariant.

Information lost on the linkbus can be re-established

The content of buffers can get lost during the transmission through the linkbus. Thus we must be able to reestablish the contents of the buffer. To prove that the necessary information was present we made small extensions to the CP-net model which allowed us to compare the information which was actually lost with the information which was re-established. Using the extended model it was possible to use invariants to prove that all buffers which could get lost could also be reestablished.

Tool Support

In the project we only made very limited use of formal validation. After the model was built we used invariants to prove a number of the most important properties of the model. One of the main reasons why we only proved a relatively small set of properties of the model is the lack of tools supporting this effort. This means that the formal analysis has been done in a total manual way.

Having had the right tool support could have made it possible to use the formal validation as an integrated part of the modelling. This could be a valuable supplement to the simulation of the models. Currently a large effort is put into the development of tools supporting formal validation.

Using CP-nets in the project

After the SNCP software module has been developed the use of CPnets is viewed as a success. The implementation of the module was fast, it was easy to extend it afterwards, and only a few bugs were found in the test phase. We conclude that the use of CP-nets in the design phase contributed to the development of a better product using fewer resources.

In general it is very hard to test programs distributed on special-purpose hardware in a wide area network. It is impossible to create debugging environments and this makes it even more important to validate the detailed design of new programs to run in this environment. The alternative of using CP-nets for modelling and simulation could be a prototyping approach. But in our case the module should fit in an environment of Hardware and Software being developed in parallel to the module.

General guide-lines for modelling and case descriptions from similar projects could be useful, especially at the start of a project. If we compare the project to experiences from similar projects, it looks as if the design phase was more time consuming and the actual coding much faster.

The model built in the project was well suited as a basis for the implementation. Much of the implementation was straightforward "translation" of the model to the programming language, even though we do not think it would be possible to generate the code automatically from the CP-net. This would require the model to be much more detailed.

References

[Albert, Jensen and Shapiro 1989] Albert, K.; K. Jensen and R.M. Shapiro. 1989. "DesignICPN. A tool package supporting the use of Coloured Petri Nets." *Petri Net Newsletter 32 (April)*, 22—36.

[Chehaibar 1990] Chehaibar, G. 1990. "Validation of Phase-executed protocols modelled with coloured Petri nets." In *Proceedings of the llth International Conference on Application and Theory of Petri Nets*, (Paris 1990), 84—103.

[Christensen and Jepsen 1991] Christensen, S and L.O. Jepsen. 1991. “Modelling and Simulation of a Network Management System using Hierarchical Coloured Petri Nets.” In *Proceedings of the 1991 European Simulation Conference*, (Copenhagen 1991).

[Hartung 1988] Hartung, G. 1988. “Programming a closely coupled multiprocessor system with high level Petri nets.” In *Advances in Petri Nets 1988*, G. Rozenberg, eds. Lecture Notes in Computer Science vol. 340, Springer-Verlag 1989, 154—174.

[Huber, Jensen and Sharpiro 1990] Huber, P.; K. Jensen and R.M. Shapiro. 1990. “Hierarchies in coloured Petri nets.” In *Advances in Petri Nets 1990*, G. Rozenberg eds. Lecture Notes in Computer Science, vol. 483, Springer-Verlag 1991, 313—341.

[Jensen 1987] Jensen, K. 1981. “Coloured Petri nets and the invariant method.” *Theoretical Computer Science 14*, 317—336.

[Jensen 1987] Jensen, K. 1987. “Coloured Petri nets.” In *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986 Part I*, W. Brauer; W. Reisig and G. Rozenberg eds. Lecture Notes of Computer Science vol. 254, Springer-Verlag 1987, 248—299.

[Jensen 1990] Jensen, K. 1990. “Coloured Petri nets: A high-level language for system design and analysis.” In *Advances in Petri Nets 1990*, G. Rozenberg eds. Lecture Notes in Computer Science, vol. 483, Springer-Verlag 1991, 342—416.

[Jepsen 1990] Jepsen, L.O. 1990. “X25/X75 Access pa RC5000, Forslag til LCP Adaptor gendringer.” PN: ENG-MEGA.LEJ.272, May 2 1990. (In Danish)

[Klausen and Jepsen 1990] Klausen, M.B. and L.O. Jepsen. 1990. “X25/X75 Access pa RC5000, Udkast til SNCP (EM Handler) Reference Manual.” PN: RCA.MEGA.LEJ.273, Feb. 2 1990. (In Danish)

[Pinci and Shapiro 1990] Pinci, V. and R.M. Shapiro. 1990. “Development and implementation of a strategy for electronic funds transfer by means of hierarchical coloured Petri Nets.” In *Proceedings of the 11th International Conference on Application and Theory of Petri Nets*, (Paris 1990), 161—179.

[Popper 1990] Popper, P. 1990. "Provision of X.25 Infrastructure (1x1) COSINE S1" *iesnews*, Issue No 28, June 90, 15.

[Shapiro 1990] Shapiro, R.M. 1990. "Validation of a VLSI chip using hierarchical coloured Petri Nets." In *Proceedings of the l l t h International Conference on Application and Theory of Petri Nets*, (Paris 1990), 224—243.

[Shapiro, Pinci and Mameli 1990] Shapiro, R.M.; V. Pinci and R. Mameli. 1990. *Modelling a NORAD command post using coloured Petri Nets.* IDEF users group, Washington DC, May 1990.