ISSN 0105-8517

Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN

Aarhus, Denmark, 13-15 October 1999

Kurt Jensen
(Ed.)

DAIMI PB – 541
October 1999

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF AARHUS
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark

Preface

This booklet contains the proceedings of the Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN, October 13-15, 1999. The workshop is organised by the CPN group at Department of Computer Science, University of Aarhus, Denmark. The papers are also available in electronic form via the web pages: http://www.daimi.aau.dk/CPnets/workshop99/

Coloured Petri Nets and the Design/CPN tools are now used by more than 500 organisations in 40 countries all over the world (including more than 100 commercial enterprises). The aim of the workshop is to bring together some of the users and in this way provide a forum for those who are interested in the practical use of Coloured Petri Nets and their tools.

The submitted papers were evaluated by a programming committee with the following members:

H. Ammar	USA	ammar@ece.wvu.edu
J. Billington	Australia	j.billington@unisa.edu.au
C. Capellmann	Germany	capellmann@tzd.telekom.de
S. Christensen	Denmark	schristensen@daimi.au.dk
H. Genrich	Germany	hartmann.genrich@gmd.de
N. Husberg	Finland	Nisse.Husberg@hut.fi
K. Jensen	Denmark (chair)	kjensen@daimi.au.dk
C. Lakos	Australia	Charles.Lakos@adelaide.edu.au
A. Levis	USA	alevis@gmu.edu
D. Moldt	Germany	moldt@informatik.uni-hamburg.de
L. Petrucci	France	petrucci@lsv.ens-cachan.fr
D. Simpson	UK	Dan.Simpson@brighton.ac.uk
E. Stear	USA	estear@aol.com
R. Valette	France	robert@laas.fr
R. Valk	Germany	valk@informatik.uni-hamburg.de
K. Voss	Germany	klaus.voss@gmd.de
W. Zuberek	Canada	wlodek@cs.mun.ca

The programme committee has received a total of 19 papers for evaluation and of these 11 have been accepted for presentation. Two thirds of the accepted papers deal with different projects in which Coloured Petri Nets and their tools have been put to practical use - most of these in an industrial setting. The remaining papers deal with different extensions of theory and tools.

The papers from the first CPN Workshop can be found via the web pages: http://www.daimi.aau.dk/CPnets/workshop98/. After an additional round of reviewing and revision, some of the papers have also been published as a special section in the International Journal on Software Tools for Technology Transfer (STTT). For more information see: http://sttt.cs.uni-dortmund.de/

Kurt Jensen

Table of Contents

CONVERTING INFLUENCE NETS WITH TIMING INFORMATION TO A DISCRETE EVENT SYSTEM MODEL, A COLORED PETRI NET[*]

LEE W. WAGENHALS AND ALEXANDER H. LEVIS
System Architectures Laboratory
School of Information technology and Engineering
George Mason University
Fairfax, VA 22030-4444, USA
{lwagenha, alevis}@gmu.edu

ABSTRACT

Recent research has shown how to incorporate time in probabilistic modeling techniques called influence nets that are used to model complex political/military situations. By adding timing information to these models, which are static equilibrium models, they are converted to discrete event system models that can be represented as colored Petri nets. Executing these CP nets reveals the dynamic changes in the probability values of key events that are modeled as propositions in the influence net. This paper illustrates how the state space analysis capability of Design/CPN was used to verify the behavior of a generic CP net model generated from the influence net. First, the implications of incorporating time in an influence net are presented along with the type of behavior that the discrete event model should have. A procedure for interconnecting CP net modules to create the overall model is presented. Finally, the state space analysis capabilities of Design/CPN are used to verify the behavior of the model and reveal interesting properties of the dynamical model that are not intuitively obvious from the structure of the model.

1 INTRODUCTION

Influence nets [11] are inspired by and are similar in form to Bayesian nets [2, 3, 10]. Because they are simpler to construct and computationally less burdensome than the Bayesian net, they can support the development of models of situations by a group of subject matter experts who need not be experienced in Bayesian nets. The fundamental assumption in influence nets, that the parents of any node are independent, causes them to behave in a similar, but not identical, manner to Bayesian nets of the same form. This assumption also facilitates the introduction of timing information which is a fundamental enabler of this research.

The influence net models are created to identify a set of controllable events that collectively have the maximum positive influence on the objectives modeled in the network. These events are called actionable events. Analysts create the influence net model of a situation using three types of nodes (hypotheses). Terminal nodes represent the effects or objectives that are desired as the result of actions to be taken. They have no outgoing arcs. Input nodes model the occurrence of the actionable events that may directly or indirectly influence or cause the objectives to occur. The third type of node is the intermediate node. These nodes model propositions that provide influencing links between the actionable events and the objectives . Once the model is constructed, a sensitivity analysis is performed to identify the actions that have the most impact on the objectives. These actions represent the un-sequenced and un-timed elements of a *Course of Action* (COA) which is defined as a timed sequence of actionable events.

In current practice, once the un-sequenced COA has been determined, the set of selected actions are provided to operational planners. They evaluate the availability of the resources needed to cause

[*] This work was supported in part by the Air Force Office of Scientific Research under Grant No. F49620-98-1-0179

the occurrence of the actionable events and determine how to schedule those resources to achieve those actions. After the plan has been formulated and approved, it is executed under the direction of operational controllers, who make adjustment in the tasking of resources as the plan unfolds.

Several observations about the current process are in order:

1. Bayesian net or influence net are static equilibrium models. They are not dynamic; there is no concept of time. They yield a joint probability distribution that is consistent with any knowledge about the hypotheses in the net. In evaluating situations, they provide the marginal probability of the nodes that represent the objective hypotheses given occurrence of a set of input actions.
2. The probabilistic model does not provide information about the impact of sequencing or timing of the actionable events on the objectives.
3. There is limited collaboration between the situation analyst, the operational planners, and the operational controllers who design and implement the COA. Each receives an input from the other, but little dialog,, discussion, or debate takes place between the actors.

1.1 CONTRIBUTION

This research provides a formal procedure for converting the information contained in an influence net along with timing information into an executable model of the situation that will generate the timed sequence of probability values for all nodes in an influence net for a given timed sequence of actionable events that defines the COA. This procedure clearly shows the time phased impact of the scheduling of resource activities on the objectives that are desired from the set of actions. A software application, written in CPN/ML, that automatically generates the CP net equivalent of the influence and provides the charting capability to display the impact of any COA on the objectives has been documented [12]. Furthermore. the researches have used the state space analysis tools of Design/CPN [4, 5, 6, 7] to verify the behavior of the CP net generated by the code.

The remainder of this paper is organized as follows. The next section shows how adding timing information changes the influence net model to a discrete event system that can be modeled using a CP net. Section 3 reviews the CP nets that are generated by the CPN/ML code and describes how they operate to generate probability profiles. Section 4 describes the state space analysis that was done to verify the behavior of the CP nets. The last section provides future directions for research.

2 INFLUENCE NET AS A DISCRETE EVENT SYSTEM

The influence net is based on causal relationships between propositions. In constructing influence nets, a temporal precedence is assumed between a causing proposition and the propositions that it influences. This assumption is consistent with most treatments of models of causality [10]. This in turn implies a sequencing of the probability of the propositions in the model as the propositions of the initial nodes become true and their effects propagate through the network. We assume that the cause and effect relationship between nodes is based on some real world phenomenon. This can be expressed as "Proposition A can trigger Proposition B (with some probability) after a time delay, d." This time delay is caused by a real world phenomenon, such as a communication process, that transfers the knowledge of proposition A to an underlying entity that can cause Proposition B to occur. We further assume that it is possible to calculate or estimate the delay.

This description suggests that the influence net is an abstraction of an underlying dynamical system that consists of concurrent and asynchronous processes. Associating time with the arcs of the influence net introduces more complex behavior to the model of the system. This behavior can be modeled by converting the influence net to a discrete dynamical model. Any model so created must have at least three characteristics,

It must be capable of computing the marginal probability of any node, given a change in the state of one or more of its parents.

It must be capable of modeling time.

It must preserve the locality principle of the causal theory.

While several modeling frameworks could be considered, Petri nets seem most appropriate. Because of the need to compute the marginal probabilities, a Colored Petri net is required.

Because, in its intended use, the influence net is a model of causal influences where inputs will be provided only to nodes with no parents. We assume a sub-set of these input nodes will represent the actionable events, propositions that we have the ability to cause with certainty. Thus, it is assumed that the probability values of these nodes will either be zero or one. The initial probability value of all nodes representing actionable events is assumed to be zero. We assume each actionable event will be made to occur at some time, thus each such node will change from a probability value of zero to a value of one. Whenever such a node changes from zero to one or F to T, its effect will propagate through the network to one or more terminal nodes.

From this perspective, we can re-cast the model of the influence net into a discrete event system model as follows:

Let M be the set of nodes in the influence net, $m_i \in M$

Partition the set of nodes of the influence net as follows:

Let I be the set of input nodes representing the actionable events.
These are the nodes without parents

Let N be the set of non-input nodes; I+N=M

Define the state of a node m_i of the influence net be its probability, $P(m_i)$

Let **u** be the input to the model represented by the set of the nodes in I; a vector of probability values from the elements of I

Let **x** be the state vector of the set of nodes in N

Let **y** be a vector of probability values of a subset of the nodes in N, these are the nodes of interest, usually the objective nodes, that are the output of the model

Let k be variable whose domain is the set of integers that is used as an index to indicate a particular value of **u, x,** or **y** in a sequence of values

An input episode will be defined as a sequence of changes in the values of the elements of $\mathbf{u}_k$. In the initial state, none of the actionable events has occurred so $\mathbf{u}_k = 0$. As each actionable event occurs, the corresponding value of the element of $\mathbf{u}_k$ changes from zero to one. Each element changes value once. The final state of the input vector is the unit vector, $\mathbf{u}_k = 1$.

With these definitions, it is possible to view an influence net as a discrete event system under the following conditions. (1) The state of the system is the vector of marginal probability values of each of the nodes in the N. (2) There is a set of admissible inputs for the nodes with no parents. As a discrete event system, there is a single initial state defined as the equilibrium probability values, of all the nodes in N, that would be calculated by the influence net with the probability value of all of the input nodes set to zero. Furthermore, there is a single final state, regardless of the sequence of the actionable events, that is the set of probability values computed by the static influence net model with all input nodes set to one. We can consider the input space, U, and the output, space, Y, as hypercubes. Each allowable input sequence can be represented as discrete changes of position in the input hypercube. Each input sequence induces a finite sequence of changes in the output space, starting with the single initial point and ending at the single final output point. Each sequence of changes represents a probability profile over the set of nodes in the output space. Figure 1 illustrates these concepts.

Figure 1 is based on a model with three input and three output nodes or variables. In the input space, all input sequences start at the origin of the input space. For this initial point in the input space, there is a corresponding point, marked P_{00} in the output space. Each allowable input sequence can be viewed as a set of discrete jumps from the origin of the input space to various "corners" of the hypercube, ending at the point [1, 1, …, 1]. Each sequence induces a piece-wise set of jumps or steps through the output space, culminating at a single final point, labeled P_f in the figure. Two input sequences are highlighted, input sequence 1 is shaded, and input sequence 2 is solid. The corresponding notional paths through the output space are also shown. The projection of the paths in the output space on the various axes provides the probability profile for each output variable.

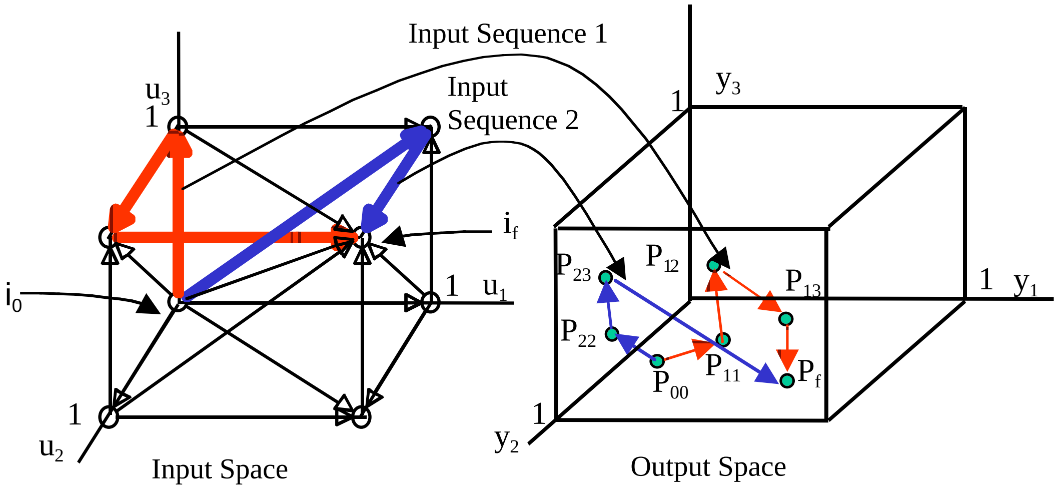


Figure 1 Admissible Input Sequences Induce Output Sequences

It is important to recognize, that there is no time or clock associated with the sequences in the above description. As described in detail by Cassandras [1], two classes of models are available that capture the behavior of discrete event system, finite state automata and Petri nets, with Petri nets being the more general model.

So far the case has been made that an influence net can be modeled as an untimed discrete event system. We need to extend the approach to incorporate time, creating a timed discrete event system model. Figure 2 shows an example of a influence net with time delays associated with the influences. A time line annotated with events is shown below the net. A time stamp can be associated with each state change of the set of input nodes. It is assumed that updates of the marginal probability of any node occur immediately after the associated time delay.

In the time line of Figure 2, the actionable events, E and F both occur at time t = 0, that is, they have a time stamp of zero. As shown on the time line, at t = 1, node A is updated due to the change in state of node E. This update uses the P[E] = 1.0 and the probability of P[F] = 0.0 even though both are one. This is denoted as $U_{A|E}$ on the time line. At t = 2, node A updates its state again due to the change in node F that occurred at t = 0. At t = 3, two updates occur simultaneously. Node B is updated due to the change of node F, and node C updates due to the first update of node A. Finally, at t = 4, node C becomes aware of the new states of both nodes A and B and thus updates its state accordingly.

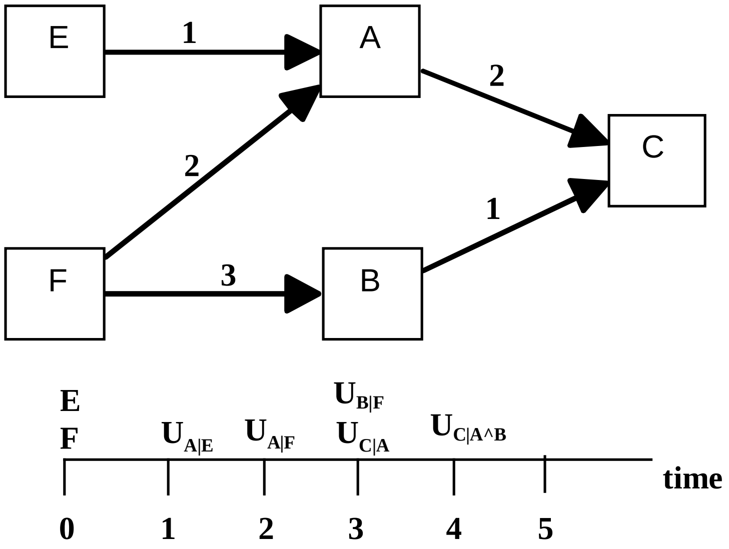


Figure 2 Timed Influence Net

This example illustrates how both the time delays associated with the influences and the time stamps of the inputs effect the order and the number of updates that occur in all of the nodes of the net. The order of the updates also affects the probability values for each update. By converting the influence net to a timed CP net, this dynamic behavior can be captured.

3 A CPN MODEL OF A TIMED INFLUENCE NET

To create a CPN model, first consider a general node in an influence net. Recall that each node in an influence net can be categorized as to one of three types, an initial node, and terminal node, or an intermediate node. The actionable events are always initial nodes, and the outcomes are the terminal nodes. Figure 3 presents a representation of an intermediate node in an influence net. Node A has *n* parents and *m* children. When the state of any parent changes, Node A will be ready to update its state after the time delay indicated on the appropriate incoming arc. Similarly, when Node A changes state, the evidence of that state change will be available at each of its children after the delay indicated on the appropriate departing arc. Note that an initial node is just an intermediate node with no parents, and a terminal node is an intermediate node with no children.

Figure 4 shows a generalized CP net design for intermediate Node A of Figure 3 with *n* inputs and *m* output arcs. The global declaration node (GDN) is shown in Figure 5. This first half of the GDN specifies the color sets and the variables used in the model. The second half of the GDN contains a set of functions that are used to compute the marginal probability of a node from a list of the marginal probabilities of the node's parents and a list containing the values of conditional probability distribution of the node. The function used in the CP net arc inscription is compmarg(<list of parents>, <list of conditional probability values>).

The net is composed of *n* + *m* + 1 places and 1 + m transitions. The places named IN1 and INn represent inputs to Node A from each of its parents. The tokens are of color set Marg that is a pair composed of a real and an integer. The real portion contains the marginal probability of the parent node, and the integer acts as a control for the transition td. This transition is enabled whenever there are tokens in all of the places representing the input nodes and the guard function evaluates to true. The subnet has an initial marking for all of the input places and each place is both in the preset and the post set of any transition it is connected to, so that any firing of such a transition removes and replaces the token. Thus there will always be a token in each place of color set Marg. Furthermore, the guard function of the td transition evaluates to true whenever one or more of the integer components of the input places is greater than zero. This means that whenever a parent is updated, it must replace the current token with a new token containing the new marginal probability value and the control integer set to one.

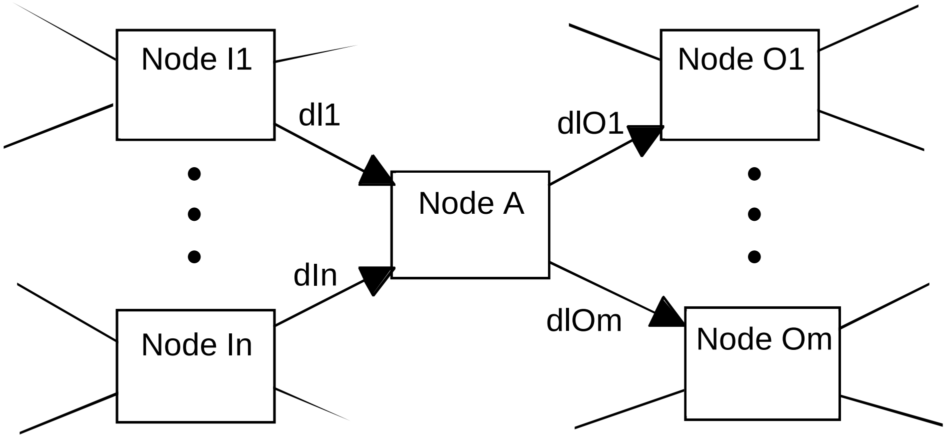


Figure 3 Generic Intermediate Node in an Influence Net

The td transition has a place of color set *Rule* in both the preset and the postset (a self loop). The *Rule* place contains tokens that are triples. The first element of the triple is an integer used as a counter. The second element of the triple contains a list of the values of the conditional probability distribution of the node. The third element is a list of the time delay values for all of the paths to the children of the node. When td fires, it removes the token in the *Rule* place and replaces it with a similar token with the counter incremented by one.

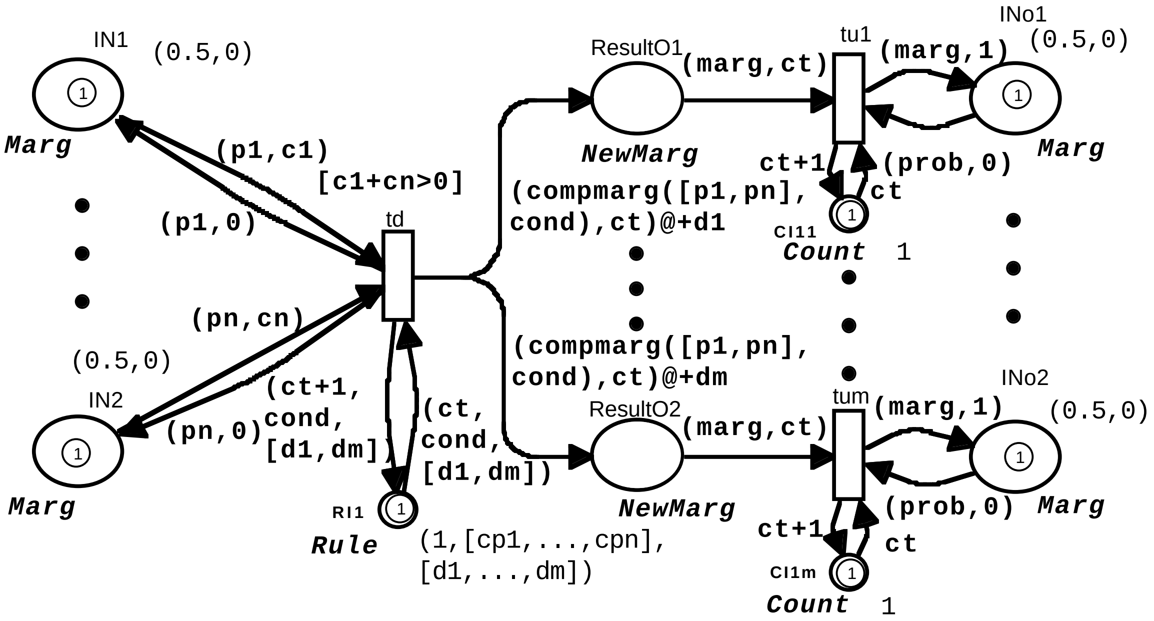


Figure 4 CP Net of an Intermediate Node of an Influence Net

```
color Control = int;
var c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,cn: Control;
color Delay = int;
var dl, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, dm: Delay;
color Prob = real;
var prob, marg ,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10,p11,p12,pn:  Prob;
color Marg = product Prob * Control;
color Count = int;
var ct: Count;
color CondList = list Prob;
var cond: CondList;
color DelayList = list Delay;
color Rule = product Count * CondList * DelayList ;
color NewMarg= product Prob * Count timed;

fun constlist 0 = []
   | constlist n = (n-1)::constlist (n-1);
fun twopower 0 = 1
   | twopower p = 2*(twopower (p-1));
fun detbase 1 n = [n]
   | detbase p n  =
      let val q = (n div (twopower (p-1)))
     in q::(detbase (p-1) (n-(q*(twopower (p-1))))) end;
fun probdet [] [] = 1.0
   | probdet (p::ps) (y::ys) = (if y=0 then p else (1.0-p))*(probdet ps ys);
fun multlist [] [] = 0.0
   | multlist ((x:real)::xs) ((y:real)::ys) = (x*y) + (multlist xs ys);
fun compmarg(pli,problist)=multlist (map (probdet pli) (map (detbase
(length(pli))) (constlist (length(problist))))) problist;
```

Figure 5 Global Declaration Node

When transition td fires, it calculates a new marginal probability for Node A by reading all of the current values of the inputs. These are the arguments for the compmarg function that is on each of the td transition's output arcs that are going to the child nodes of the influence net. The first argument is a list of the marginal probabilities of all of the parents [p1, …, pn] and the second is a list of the condition probability values for Node A that is stored in the place of color set type *Rule*. Transition td sends a token with the new marginal probability value, to *m* different output places of type *NewMarg*. *NewMarg* is a timed color set that is a double composed of a real, for the probability value, and an integer. Each of these tokens is given a time stamp consistent with the delay associated with the propagation of the influence between Node A and each of its children (d1, d2,… or dm). After the appropriate delay, each transition tu1 through tum fires and exchanges the token in the input place of the child with the new marginal probability of Node A and the control value set to one. This "signals" the child, which has a CP net that is similar to that for Node A, that a new update has arrived.

The purpose of the place of color set *Count* is to implement a first in, first out (FIFO) protocol on token in the *NewMarg* places. The FIFO protocol is required because it is possible, depending on the firing sequence, to have transition td fire more than once without advancing the simulation time clock. This means that more than one token can exist in the *NewMarg* places with the same time stamp. It is necessary to preserve the order of the generation of tokens with the same time stamp to ensure that the correct final state is reached.

The FIFO protocol is implemented by the second component of token in the *Rule* place which is a counter that is initialized to one and whose value is incremented each time transition td fires. The place with color set *Count* contains a similar counter that increments whenever its transition, tui, fires. If there are multiple tokens in the NewMarg place, they will be consumed in the order in which they were generated by transition td because the ct variable of the token must match the value of the ct in the *Count* place.

CP nets representing input and terminal nodes are similar to an intermediate node as shown in Figures 6 and 7. Note that the input node has a single input place of timed color set *NewMarg* to allow for the time stamps to be incorporated on the input tokens that represent the occurrence of an actionable event. The terminal node has a single output to which there is no delay associated. The *NewMarg* color set is used as the output place named ResultO1. It contains an initial marking that is the marking of the node that results if all the inputs are set to zero. The same *Rule* place is used to provide a counter for the tokens generated in the ResultO1 place indicating the sequence in which they were generated.

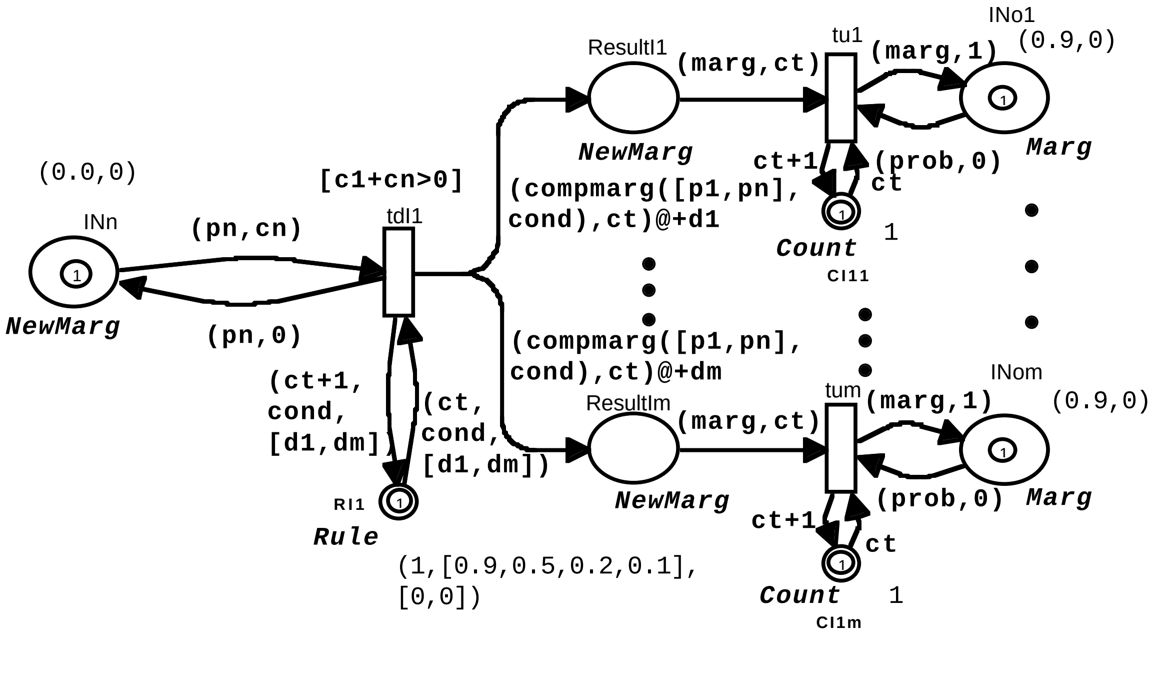


Figure 6 Generic CP Net of an Input Node of an Influence Net

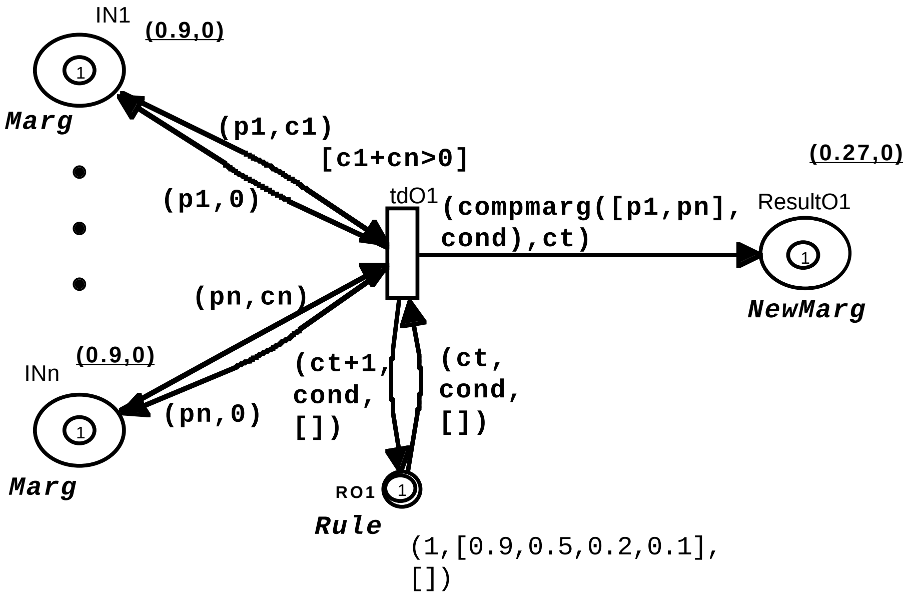


Figure 7 Generic Output Node

Converting an influence net to a time colored Petri net is straight forward. Indeed, an automatic routine for doing this has been written in CPN/ML [12]. To illustrate how the sub nets are interconnected, consider the three node timed influence net of Figure 8 with time delays d1, d2, and d3. Node A is a input or source node, node B is an intermediate node, and node C is a terminal or sink node. The corresponding colored Petri net for this influence net is shown in Figure 9. Most of the arc inscriptions have been suppressed for clarity.

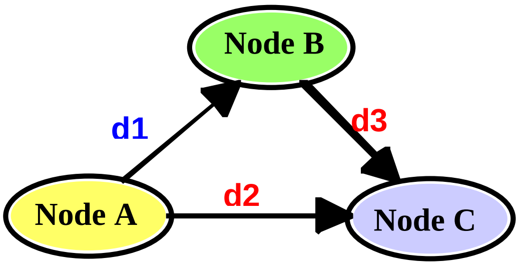


Figure 8 Three Node Timed Influence Net

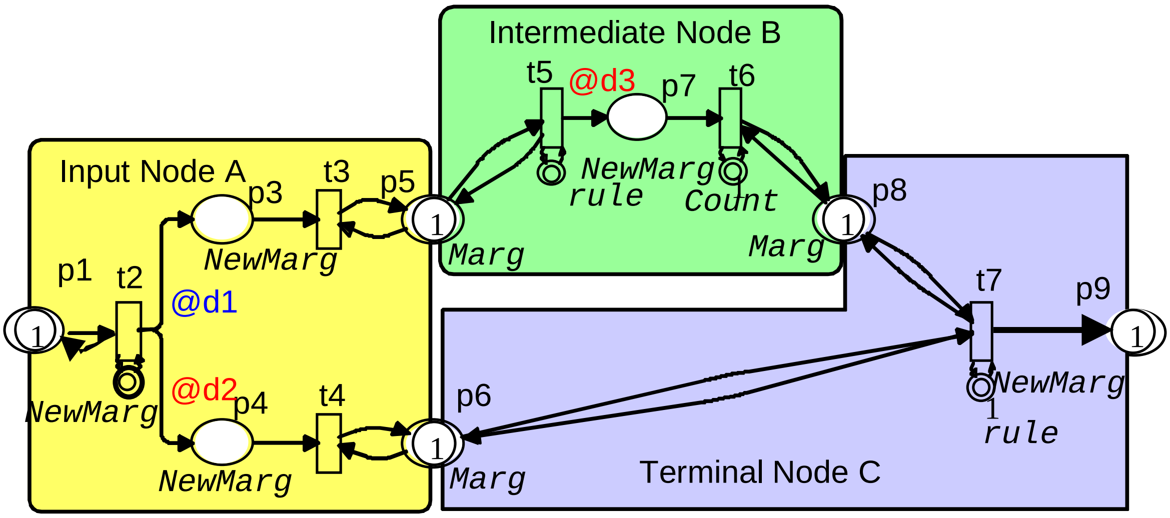


Figure 9 CP net of Influence Net

4 STATE SPACE ANALYSIS FOR VERIFICATION OF BEHAVIOR

State space analysis was used to verify the behavior of the CP net construction presented in Section 3. This section highlights key parts of the analysis and illustrates how the Occurrence Graph Analyzer (OGA) of Design/CPN can be used for behavior verification. The analysis was done in two steps. First a simple influence net model that has all three types of nodes was used and the corresponding CP net was created using Design/CPN. The OGA was used to generate occurrence graphs to generate all the possible occurrence sequences in the CP net. In addition, the state space report that provides statistical and standard properties of the CP net was used to verify its behavior. This analysis revealed that joins are a major factor in the behavior of the model. Thus, in the second step a detailed analysis of a join was conducted.

Figure 10 shows the timed influence net that was used for the verification process. Included are the conditional probability distributions for the intermediate and terminal nodes. In addition, the initial values of the conditional probabilities for each node is shown with the o subscript. The topology was selected because it has the minimal set of features needed to verify the behavior of larger nets. These features include multiple inputs and terminal nodes, an intermediate node, and at least one branch and one join.

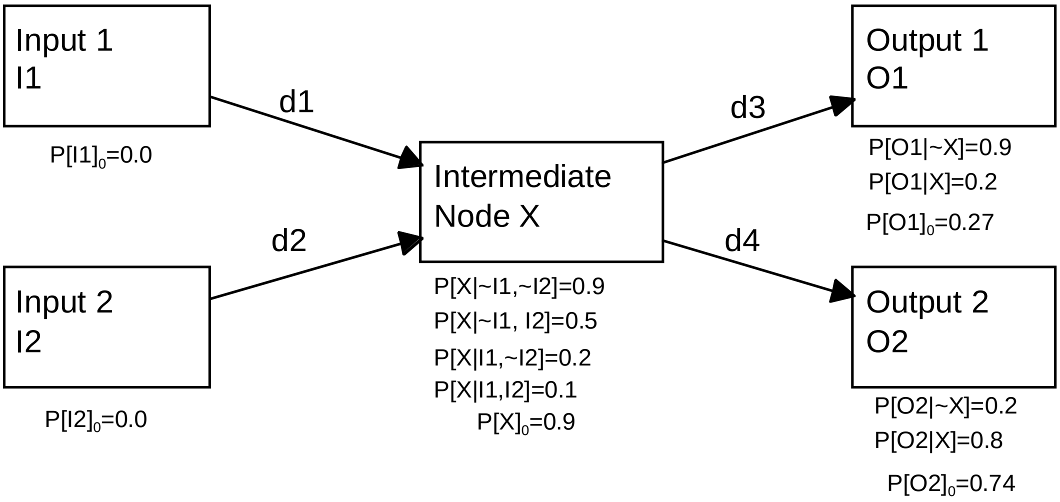


Figure 10 Verification Test Timed Influence Net

Occurrence graphs and state space reports were generated for four different cases. Each case illustrates, important features and characteristics of the discrete event model of the timed influence net. The cases are:

1.) Untimed (all delays set to zero and both input occur at time zero) single input,
2.) Timed unequal time delays and input time stamps of zero.
3.) Timed with all delays equal zero and unequal time stamps on the inputs.
4.) Timed with a combination of different time delays and time stamps on the inputs

To aid in understanding the occurrence graphs and the state space reports, the CP net of the timed influence net is shown in Figure 11. Most of the annotations have been suppressed; the names of the places and transitions are shown in Helvetica bold and the color sets are in italic.

The occurrence graph of the untimed single input (I1) case is shown in Figure 12. Details of the states and occurrences have been provided for only a few of the key nodes and arcs of the graph, and a level index has been added to help in the description of the behavior. From the topology of the CP net, we can see that with only input I1, there should be seven steps for the input to propagate its effect through the network, ending in a dead marking. The first three transitions to fire should be tdInput01, tInput01, and tdI1 because these transitions will be the only enabled transitions in this sequence. After tdI1 fires, both tu1 and tum are enabled. This results in a branch in the occurrence graph at node 4. The remaining steps of firing occur as tu1, tum, tdO1, and tdO2 fire. The exact order of firing is not specified in the CP net, hence there are several different paths from node 4 to the final node 12 in the occurrence graph. Observing the state of the CP net at node 12 reveals that the markings of the ResultO1 and ResultO2 places are correct given the input. ResultO1 shows the initial probability of 0.27 and a final probability of 0.76 which are the correct values. Similarly ResultO2 has probabilities of 0.74 and 0.32, which are also the correct values.

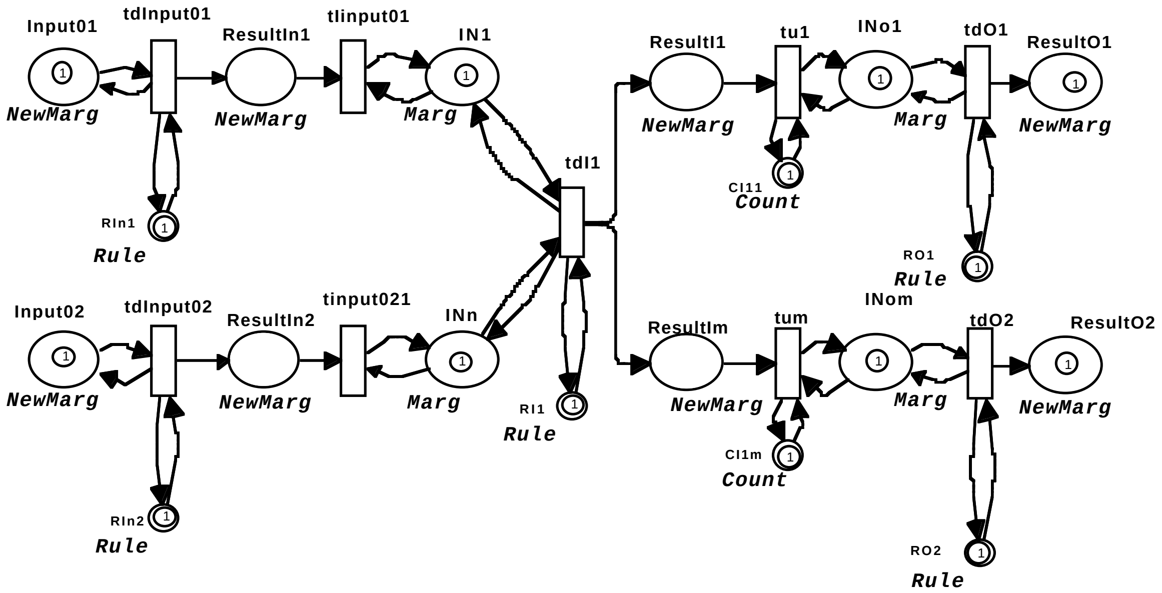


Figure 11 CP Net of Timed Influence Net

Design/CPN is capable of generating a state space report without executing the net. This report can be generated very quickly, even for large state spaces, and provides a great deal of useful information about the properties of the CP net that characterize the behavior of the net for a given marking. The report is divided into four sections: Statistical Information, Boundedness Properties, Home and Liveness Properties, and Fairness Properties. The Statistical section of the State Space Report for the Single Input case is shown in Table 1.

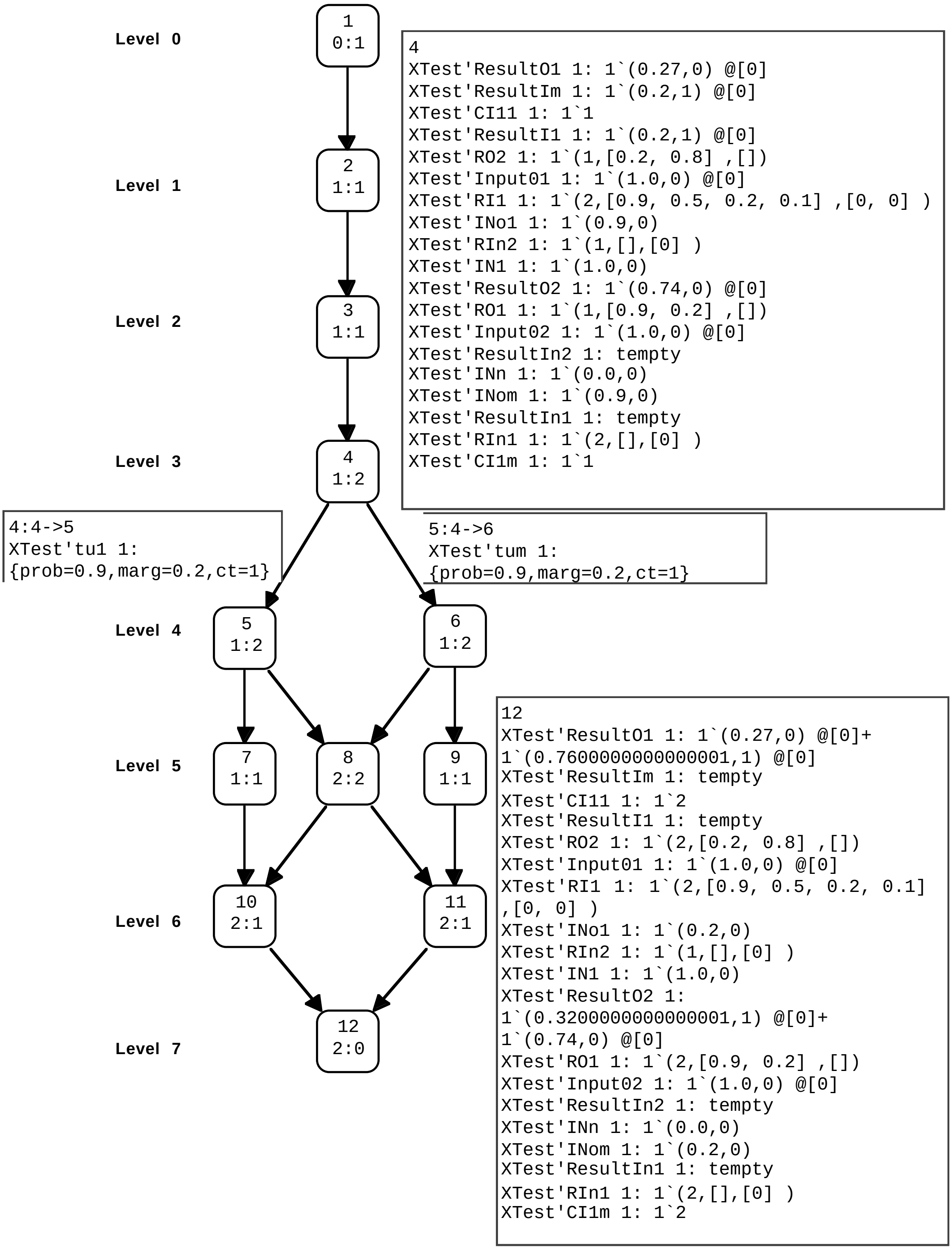
Table 1 Statistical Information For Single Input

```
Statistics
------------------------------------------------------
 Occurrence Graph          Scc Graph
   Nodes:  12              Nodes:  12
   Arcs:   15              Arcs:   15
   Secs:   0               Secs:   0
   Status: Full
```

The statistics show that the state space has 12 states with 15 arcs. The Secs indicates the number of seconds it took for the processor to generate the State Space Report. In this case it was less than one second. The status means that the full state space has been analyzed. The SCC Graph stands for Strongly Connected Component Graph. The SCC graph has the same number of nodes as the state space. Since none of the SCCs has more than one node, there are no cycles in the state space, and, therefore, no infinite occurrence sequences. This is the correct characteristic for the CP net.

The Boundedness Properties of the report are shown in Table 2. This report lists the upper and lower integer bounds for each place in the net. These bounds indicate the maximum and minimum number of tokens that can exist in each place in the CP net in all the reachable markings. This report indicates that the CP net is functioning properly. Each of the *Counter* and *Rule* places always has one token as do the places that hold the current marginal probability values of the nodes of the influence net. The *Result* places hold either zero, one or two tokens depending on their location in the net. The

ResultO1 and ResultO2 places always have at least one token, the initial marking, and have no more than two tokens after all of the updates arrive.



```
4
XTest'ResultO1 1: 1`(0.27,0) @[0]
XTest'ResultIm 1: 1`(0.2,1) @[0]
XTest'CI11 1: 1`1
XTest'ResultI1 1: 1`(0.2,1) @[0]
XTest'RO2 1: 1`(1,[0.2, 0.8] ,[])
XTest'Input01 1: 1`(1.0,0) @[0]
XTest'RI1 1: 1`(2,[0.9, 0.5, 0.2, 0.1] ,[0, 0] )
XTest'INo1 1: 1`(0.9,0)
XTest'RIn2 1: 1`(1,[],[0] )
XTest'IN1 1: 1`(1.0,0)
XTest'ResultO2 1: 1`(0.74,0) @[0]
XTest'RO1 1: 1`(1,[0.9, 0.2] ,[])
XTest'Input02 1: 1`(1.0,0) @[0]
XTest'ResultIn2 1: tempty
XTest'INn 1: 1`(0.0,0)
XTest'INom 1: 1`(0.9,0)
XTest'ResultIn1 1: tempty
XTest'RIn1 1: 1`(2,[],[0] )
XTest'CI1m 1: 1`1
```

```
4:4->5
XTest'tu1 1:
{prob=0.9,marg=0.2,ct=1}
```

```
5:4->6
XTest'tum 1:
{prob=0.9,marg=0.2,ct=1}
```

```
12
XTest'ResultO1 1: 1`(0.27,0) @[0]+
1`(0.7600000000000001,1) @[0]
XTest'ResultIm 1: tempty
XTest'CI11 1: 1`2
XTest'ResultI1 1: tempty
XTest'RO2 1: 1`(2,[0.2, 0.8] ,[])
XTest'Input01 1: 1`(1.0,0) @[0]
XTest'RI1 1: 1`(2,[0.9, 0.5, 0.2, 0.1]
,[0, 0] )
XTest'INo1 1: 1`(0.2,0)
XTest'RIn2 1: 1`(1,[],[0] )
XTest'IN1 1: 1`(1.0,0)
XTest'ResultO2 1:
1`(0.3200000000000001,1) @[0]+
1`(0.74,0) @[0]
XTest'RO1 1: 1`(2,[0.9, 0.2] ,[])
XTest'Input02 1: 1`(1.0,0) @[0]
XTest'ResultIn2 1: tempty
XTest'INn 1: 1`(0.0,0)
XTest'INom 1: 1`(0.2,0)
XTest'ResultIn1 1: tempty
XTest'RIn1 1: 1`(2,[],[0] )
XTest'CI1m 1: 1`2
```

Figure 12 Occurrence graph of Single Input Initial Marking for Untimed Net

Table 3 shows the upper multi-set bounds for the CP net. While the integer bound provides information about the maximum and minimum number of tokens in a place, the multi-set bound provides information about the value that the tokens can carry. The upper multi-set bound of the two output places, ResultO1 and ResultO2 are of particular interest. The upper multi-set bound provides a list of all the possible probability values of these place, and thus the maximum and minimum value of any possible probability profiles for the given initial marking.

Table 2 Upper Integer Bounds for Single Input Case

Boundedness Properties

Best Integers Bounds	Upper	Lower	Best Integers Bounds	Upper	Lower
CI11 1	1	1	RIn2 1	1	1
CI1m 1	1	1	RO1 1	1	1
IN1 1	1	1	RO2 1	1	1
INn 1	1	1	ResultI1 1	1	0
INo1 1	1	1	ResultIm 1	1	0
INom 1	1	1	ResultIn1 1	1	0
Input01 1	1	1	ResultIn2 1	0	0
Input02 1	1	1	ResultO1 1	2	1
RI1 1	1	1	ResultO2 1	2	1

Table 3 Upper Multi-set Bounds for Single Input Case

Best Upper Multi-set Bounds	
CI11 1	1`1+ 1`2
CI1m 1	1`1+ 1`2
IN1 1	1`(0.0,0) + 1`(1.0,0) + 1`(1.0,1)
INn 1	1`(0.0,0)
INo1 1	1`(0.2,0) + 1`(0.2,1) + 1`(0.9,0)
INom 1	1`(0.2,0) + 1`(0.2,1) + 1`(0.9,0)
Input01 1	1`(1.0,0) + 1`(1.0,1)
Input02 1	1`(1.0,0)
RI1 1	1`(1,[0.9, 0.5, 0.2, 0.1] ,[0, 0] ) + 1`(2,[0.9, 0.5, 0.2, 0.1] ,[0, 0])
RIn1 1	1`(1,[],[0] ) + 1`(2,[],[0])
RIn2 1	1`(1,[],[0])
RO1 1	1`(1,[0.9, 0.2] ,[]) + 1`(2,[0.9, 0.2] ,[])
RO2 1	1`(1,[0.2, 0.8] ,[]) + 1`(2,[0.2, 0.8] ,[])
ResultI1 1	1`(0.2,1)
ResultIm 1	1`(0.2,1)
ResultIn1 1	1`(1.0,1)
ResultIn2 1	empty
ResultO1 1	1`(0.27,0) + 1`(0.76,1)
ResultO2 1	1`(0.32,1) + 1`(0.74,0)

Table 4 shows the lower multi-set bounds for the net. A lower multi-set bound is the largest multi–set which is smaller than all of the reachable markings of a place. In other words, it is a multi–set of tokens in a place whose values never change in all of the reachable markings. The are five such places in the Single Input Case. The first three in the table are associated with the Input I2 that was never activated so the markings of the place named INn1 and the *Rule* place named Rin2 never changed. The other *Rule* and counter places change because the counter values change. The initial marking of the ResultO1 and ResultO2 places never change because there are no output arcs from either of these places.

The third part of the state space report provides information about the Liveness Properties of the CP net as shown in Table 5. It lists dead markings and dead transitions. A dead marking is a marking with no enabled transition, a final state in the occurrence graph. The report shows that there is a single dead marking of node 12 in the occurrence graph. A dead transition is a transition that is never enabled from any reachable marking. In this CP net there are two dead transitions, tdInput02 and tinput021. These transitions are on the path from the second input I2 and can never fire given the initial marking.

Table 4 Best Lower Multi-set Bounds

INn 1	1`(0.0,0)
Input02 1	1`(1.0,0)
RIn2 1	1`(1,[],[0])
ResultO1 1	1`(0.27,0)
ResultO2 1	1`(0.74,0)
All others	empty

Table 5 Liveness Properties

Dead Markings: [12]

Dead Transitions Instances:

tdInput02 1

tinput021 1

The state space analysis shows that the CP net is operating properly with a single input. The next step is to generalize the analysis for multiple inputs with the net both unitimed and timed.

The most general case is that with all inputs present and the network untimed. This case can be created by marking all inputs with tokens that enable the input transitions and setting all of the delay values to zero. This initial marking will provide the maximum flexibility in firing sequences for the transitions in the CP net. It will also provide the largest state space for the CP net. Any initial marking that incorporates at least one delay that differs from all the other delays, or incorporates a different time stamp on any input token will provide a restriction on the firing sequences and generate a subset of the state space of the "untimed" CP net.

The state space report revealed that the untimed CP net with both inputs activated has the same behavioral properties as the single input case. The statistical report showed that the SCC graph and the occurrence graph have the same number of nodes (122) and arcs (235), and, therefore, no cycles. The integer boundedness properties are also the same as for the single input case, with the exception that ResultI1 and Result1m can have two tokens as the upper integer bound and ResultO1 and ResultO2 can have three tokens as the upper bound.

The topology of the 122 node occurrence graph revealed several important characteristics of the untimed CP net. First, there were three final states in the graph and 15 levels in the graph. Two of the final states (nodes 121 and 122) are reached if all 15 levels are traversed while the third final state (node 87) requires only 10 levels. At the first level, it is possible to reach all three final states. However choices made at level 1 or level 2 could eliminate one final state as a reachable marking. Level 3 is the last level were it is possible to reach more than one final state. It is the branches in the influence net that cause these choices.

Most of the boundedness properties were the same as for the single input case. The key property is associated with the ResultO1 and ResultO2 places. The portion of multi-set boundedness properties associated with the two terminal nodes is shown in Table 6. The probability and the counter values of the tokens shows that each place starts with its initial value, (0.27, 0) and (0.74, 0), respectively. These are the initial values because the counter values are zero. The remaining elements of the multi-set show that ResultO1 can have values of (0.55, 1), (0.76, 1), (0.83, 1) and (0.83, 2). The token with the highest counter value is the final state of the node, 0.83, which is the correct value, given both inputs. The other values are intermediate values that are computed when either the first input propagates through the net before input 2 or when the second input propagates through the net before input 1. More precisely, these values correspond the sequence of arrival of the input updates at the transition tdI1 (see Figure 11) which computes the new marginal probability of node X. The value (0.83, 1) indicates that the input updates were both available to the transition tdI1 when it fired for the first time. Notice that if the updates do not arrive simultaneously then transition tdI1 will fire twice, once for each update.

Table 6 Boundedness Properties of the Untimed CP Net

Best Upper Multi-set Bounds

ResultO1 1	1`(0.27,0) + 1`(0.55,1) + 1`(0.76, 1) + 1`(0.83, 1) + 1`(0.83, 2)
ResultO2 1	1`(0.26,1) + 1`(0.26,2) + 1`(0.32, 1) + 1`(0.5,1) + 1`(0.74, 0)

Best Lower Multi-set Bounds

ResultO1 1	1`(0.27,0)
ResultO2 1	1`(0.74,0)
All Others	empty

The Liveness properties indicate that there are three dead markings, nodes 87, 121, and 122 of the occurrence graph. Unlike the single input case, there are no dead transitions because each transition can at fire least once on some path through the occurrence graph. In our case, each transition fires at least once on all paths through the occurrence graph.

The value of the ResultO1 and O2 places for dead markings is shown in Figure 13. The other places for the dead markings of nodes 121 and 122 are the same as for node 87 with the exception that the counter values for nodes 121 and 122 are three in all places with counters.

The three dead markings are consistent with the State Space Report and desired behavior of the CP net. Node 87 is the result of firing sequences in which the two updates for Input 1 and Input 2 occur before the transition tdI1 fires, hence it only fires once with the correct marginal probability of node X. Thus, the markings of the places ResultO1 and ResultO2 make one single change from the initial probability to the final probability. Occurrence Graph node 121 is reached if transition tdI1 fires when enabled by the update from input 1 without the update from input 2 and then fires a second time when the update from input 2 arrives, thus generating two tokens. The first token generates an intermediate marginal probability values of 0.76 and 0.32 in ResultsO1 and O2, respectively, before the second token generates the final marginal probability values. Occurrence graph node 122 has the same property except the result of input 2 occurs first before the final result that includes both inputs.

```
87
ResultO1  1: 1`(0.27,0) @[0]+ 1`(0.83,1) @[0]
ResultO2  1: 1`(0.26,1) @[0]+ 1`(0.74,0) @[0]
121
ResultO1 1:             1`(0.27,0) @[0]+  1`(0.76,1) @[0]+  1`(0.83,2) @[0]
ResultO2 1:             1`(0.26,2) @[0]+  1`(0.32,1) @[0]+  1`(0.74,0) @[0]
122
ResultO1 1:             1`(0.27,0) @[0]+  1`(0.55,1) @[0]+  1`(0.83,2) @[0]
ResultO2 1:             1`(0.26,2) @[0]+  1`(0.5,1) @[0]+  1`(0.74,0) @[0]
```

Figure 13 ResultO2 and O2 for the Three Dead Markings

This explanation demonstrates the need for the counters that enforce the FIFO protocol. Without this protocol it would not be possible to distinguish the final value of the marginal probability from the intermediate values.

Once the behavior of the untimed CP net was verified, the behavior of the timed net was examined. There are two types of timing constraints that can be incorporated in the CP net. The first is by setting the timing of the inputs, and the second is by incorporating time delays on the arcs between parents and children. Each was investigated individually and then in combination.

The occurrence graphs clearly showed the effects of timing. The occurrence graph with the Input 1 and 2 occurring at t = 0 and t = 1, respectively, showed that separating the inputs reduces the size of the state space from 122 nodes to 23 and reduces the dead markings from three to one. As expected, the occurrence graph is simply two single input occurrence graphs that are concatenated.

The occurrence graphs with obtained different time delays and both inputs with time stamp of zero, showed that these delays have the impact of separating the inputs but only after both input transitions have firesd This is because either input can be concurrently enabled throughout every step of the propagation the other while the simulation time remains at zero. Thus, the first half of the occurrence graph has many more nodes than that of the case where the inputs where timed. Nevertheless the final results with respect to the dead marking are the same.

Finally, a mix of time delays and time difference of the inputs was examined. These can cause the occurrence graph become a single chain as all choice is eliminated.

As was noted earlier when the occurrence graph of the untimed net was introduced, the determination of which dead marking will be reached occurs early in any firing sequence. A closer look at the occurrences in the occurrence graph reveals that it is the joins in the influence net that determines the number of dead markings in an untimed CP net model the influence net. A net with no joins will have a single dead marking. A net with at least one join will have multiple dead markings. Branches have no impact on the number of dead markings, but do impact the size of the state space.

To further examine the impact of joins on the number and composition of dead markings, a simple CP net model of a join was created as shown in Figure 14. This CP net represents a three legged join. To keep the model simple, only the places and transitions that update each leg of the join were modeled. The join has a single output because branches do not effect dead markings.

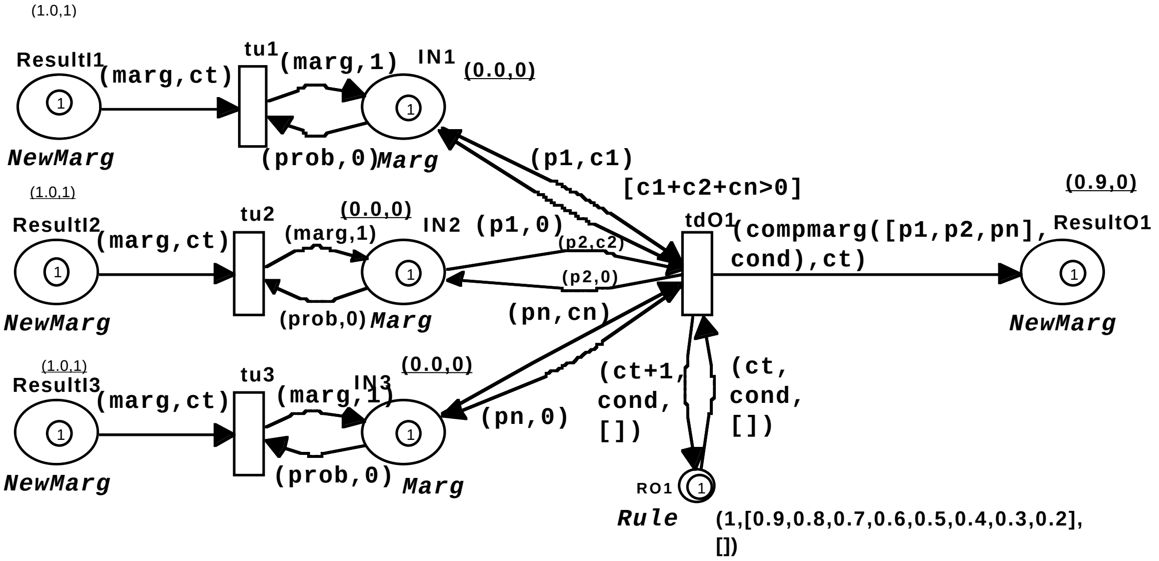


Figure 14 CP Net of Three Legged Join

The behavior of this CP net model can be anticipated. Initially transitions tu1, tu2, and tu3 will be concurrently enabled. Because there is no control on the firing sequences of concurrently enabled transitions, any one of those three can fire first. As soon as one fires, transition tdO1 becomes concurrently enabled with the remaining two tui transitions. It can fire next or wait until one or more of the two remaining tui fire. Whenever tdO1 fires, it "reads" the current probability values of each preset place (IN1, IN2, and IN3), and generates a token that is a double with the "new" marginal probability based on those values plus the current counter number. The values of the preset places depend on whether the corresponding tui has fired. Depending on the firing sequence, transition tdO1 can fire only once if it "waits" until all three tui have fired, it can fire twice, if it fires once before all three tui have fired or it can fire three times if it fires sequentially after each tui fires.

Obviously there are a number of potential firing sequences for this CP net model. We can view the combination of potential firing sequences based on all of the possible sequences of arrivals of the three inputs. For three inputs there are 13 such sequences. Each sequence will produce a different marking in the Result place, hence we can expect the occurrence graph of the CP net to have 13 dead markings.

The values probability values generated in the firing sequences can be predicted.

Let P[X] be the marginal probability of the join node that is calculated when tdO1 fires.

Let u_i be the current (old) marginal probability value of the i^{th} parent.

Let u_i' be the new, updated marginal probability value of the i^{th} parent

Then, for the 13 potential sequences of the 3 inputs there are $2^3 = 8$ marginal probability values:

$P[X|u_1, u_2, u_3]$, $P[X|u_1, u_2, u_3']$, $P[X|u_1, u_2', u_3]$, $P[X|u_1', u_2, u_3]$, $P[X|u_1, u_2', u_3']$, $P[X|u_1', u_2, u_3']$, $P[X|u_1', u_2,' u_3]$, $P[X|u_1', u_2', u_3']$

These sets of values can be arranged in a partial order based on the number of new updates in the condition.

Without loss of generality, the initial values of the old and new marginal probabilities and the conditional probability distribution have been chosen in a way that will simplify the analysis of the state space. The initial marginal probability values are set to zero and the new updates are all one.

The conditional probability distribution is a set of eight unique values starting with 0.9 and descending in equal increments to 0.2. Because the calculation of marginal probability involves products of the input probabilities and their compliments, each input sequence will uniquely result in one of the values of conditional probability distribution.

These characteristics are summarized in Table 7. The first column is an index indicating the sequence number. The second three columns indicate the order of processing of the three updates, u_1, u_2, and u_3. For example 1, 2, 3 means the updates are processed in sequence, while 2, 1, 1 means that u_2 and u_3 are processed first, simultaneously followed by u_1. This ordering corresponds to different paths through the input space that was illustrated in Figure 1. The fifth column lists, in sequence, the marginal probability values that are computed for the specific sequence. The sixth column provides the actual probability values for the set of initial markings as discussed in the previous paragraph.

The behavior described in Table 7 can also be presented graphically as shown in Figure 15. The nodes in the graph are triples that indicate the value of the three inputs that are used in the marginal probability calculation. For example, (0, 1, 0) means u_1=0, u_2=1 and u_3=0. The arcs indicate an allowable change from one value to another. The initial conditions start with the input (0, 0, 0), and the input sequence always terminates with all values of the inputs equal to one (1, 1, 1). There are 13 paths from the initial node to the terminal node. Each path presents a feasible sequence of changes in the marginal probability. Note that there are arcs from the initial node to all other nodes and arcs from all nodes to the terminal node.

Table 7 Sequences of Updates

No.	u_1	u_2	u_3	Marginal Probability Calculations	Values
1	1	2	3	$P[X \mid u_1', u_2, u_3]$, $P[X \mid u_1', u_2', u_3]$, $P[X \mid u_1', u_2', u_3']$	0.5, 0.3, 0.2
2	1	3	2	$P[X \mid u_1', u_2, u_3]$, $P[X \mid u_1', u_2, u_3']$, $P[X \mid u_1', u_2', u_3']$	0.5, 0.4, 0.2
3	2	1	3	$P[X \mid u_1, u_2', u_3]$, $P[X \mid u_1', u_2', u_3]$, $P[X \mid u_1', u_2', u_3']$	0.7, 0.3, 0.2
4	2	3	1	$P[X \mid u_1, u_2, u_3']$, $P[X \mid u_1', u_2, u_3']$, $P[X \mid u_1', u_2', u_3']$	0.8, 0.4, 0.2
5	3	1	2	$P[X \mid u_1, u_2', u_3]$, $P[X \mid u_1, u_2', u_3']$, $P[X \mid u_1', u_2', u_3']$	0.7, 0.6, 0.2
6	3	2	1	$P[X \mid u_1, u_2, u_3']$, $P[X \mid u_1, u_2', u_3']$, $P[X \mid u_1', u_2', u_3']$	0.8, 0.6, 0.2
7	1	2	2	$P[X \mid u_1', u_2, u_3]$, $P[X \mid u_1', u_2', u_3']$	0.5, 0.2
8	2	1	1	$P[X \mid u_1, u_2', u_3']$, $P[X \mid u_1', u_2', u_3']$	0.6, 0.2
9	1	1	2	$P[X \mid u_1', u_2', u_3]$, $P[X \mid u_1', u_2', u_3']$	0.3, 0.2
10	2	2	1	$P[X \mid u_1, u_2, u_3']$, $P[X \mid u_1', u_2', u_3']$	0.8, 0.2
11	1	2	1	$P[X \mid u_1', u_2, u_3']$, $P[X \mid u_1', u_2', u_3']$	0.4, 0.2
12	2	1	2	$P[X \mid u_1, u_2', u_3]$, $P[X \mid u_1', u_2', u_3']$	0.7, 0.2
13	1	1	1	$P[X \mid u_1', u_2', u_3']$	0.2

The occurrence graph of the three legged join CP Net had 51 nodes, 58 arcs, and 13 dead markings. The other properties are consistent with the state space analysis performed on the untimed CP net of Figure 11. The most interesting property is the upper multi-set bound for the ResultO1 node. This multi-set is 1`(0.2,1) + 1`(0.2,2) + 1`(0.2,3) + 1`(0.3,1) + 1`(0.3,2) + 1`(0.4,1) + 1`(0.4,2) + 1`(0.5,1) + 1`(0.6,1) + 1`(0.6,2) + 1`(0.7,1) + 1`(0.8,1) + 1`(0.9,0).

This upper multi-set bound shows eight potential marginal probability values for the three legged join. These are the same values shown in Table 7. The 13 markings for ResultO1 extracted from the dead markings are shown in Table 8. Each row of the table represents one of the 13 paths from the initial to the final value and each path contains the sequence of probability values that can occur in a probability profile.

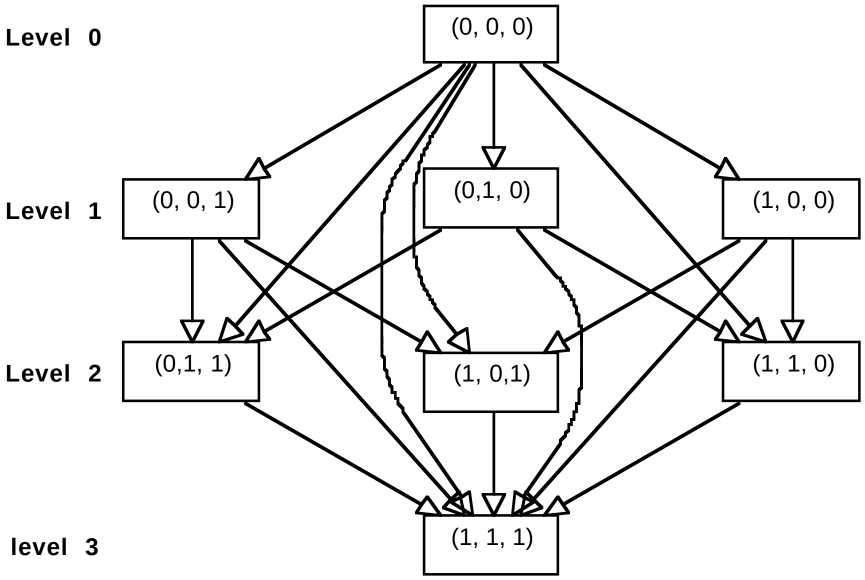


Figure 15 Graph of 13 Sequences Given 3 Inputs

Table 8 Dead Markings for the ResultO1 Place

OG Node	Counter = 0	Counter = 1	Counter = 2	Counter = 3
25	0.9	0.2		
37	0.9	0.3	0.2	
38	0.9	0.4	0.2	
35	0.9	0.5	0.2	
42	0.9	0.6	0.2	
40	0.9	0.7	0.2	
44	0.9	0.8	0.2	
46	0.9	0.5	0.3	0.2
47	0.9	0.5	0.4	0.2
48	0.9	0.7	0.3	0.2
49	0.9	0.7	0.6	0.2
50	0.9	0.8	0.4	0.2
51	0.9	0.8	0.6	0.2

If one or more of the inputs to the three legged join are given a time stamp that is different from the other inputs, the impact should be to reduce both the size of the state space and the number of dead markings. This is because the time difference in the inputs separates the availability of the tokens and reduces the number of transitions that can be concurrently enabled.

This behavior was clearly depicted in the occurrence graph of the three legged join with inputs 1 and 2 with time stamps of zero and input 3 with a time stamp of 2. In each dead marking , the initial marginal probability value of the node was 0.9 with a counter value of zero. All three dead markings showed the correct final probability value of 0.2 that occurs at time equal 2. The three dead markings differed in the set of markings at time equal zero. Each has the same final probability value at t = 0 but each shows a different intermediate value caused by a different path through a portion of the output space. The intermediate probability values occur because of different firing sequences due to concurrent enablements of transitions. These intermediate values with the same time stamp are artifacts of the updating process and have no real meaning. The FIFO protocol ensures that tokens produced at joins will be processed by children of the join in the order they were created, with the last token value being the correct and final value of the update process at a given point in time. The sequenced set of correct and final probability values of a node constitute a probability profile for the node for a given input. In the example, each of the dead markings generates an identical probability profile starting with the initial value of 0.9 that changes to 0.3 at time t = 0 and remains at that value until t = 2 when it changes to 0.2.

Figure 16 shows the output space for the join. The arcs are labeled with the updated probability that arrived at the join that triggered the transition in state. The bolded nodes and arcs indicate the portion of the state space that is reachable for this timed example.

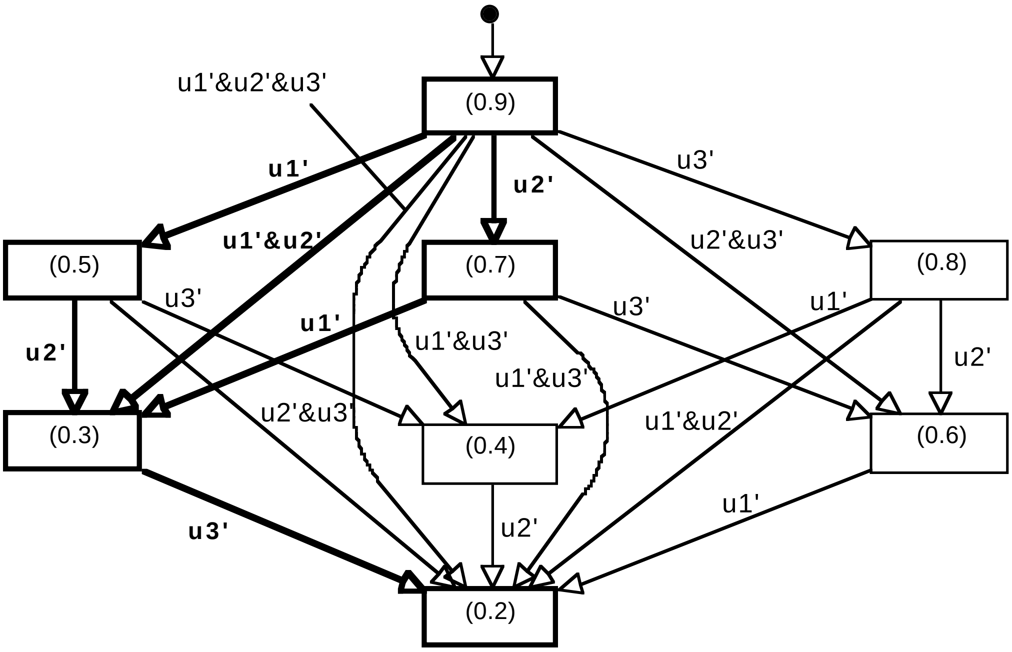


Figure 16 Output Space of ResultO1

This example can be used to illustrate the concept of the probability profile. If the arrival times of the inputs to the join can be selected, it is possible to control the probability profile. Figure 17 shows two profiles that can be generated from the join. The profile with the square points is for the case where input 2arrives at time = 1, input 3 arrives at time = 2 and input 1 arrives at time = 4. If the arrival times of the inputs are changed to inputs 1, 2 and 3 at times 1, 1, and 2, respectively, the probability profile changes to the one denoted by the circles. If the objective was to cause the output probability to be as small as possible, the second profile is preferred over the first because decreases more rapidly. While it is easy to determine the probability profile of a simple net, the state space analysis has shown that the CP net can be used to generate the probability profile for any influence net with time delays and a specific set of timed inputs.

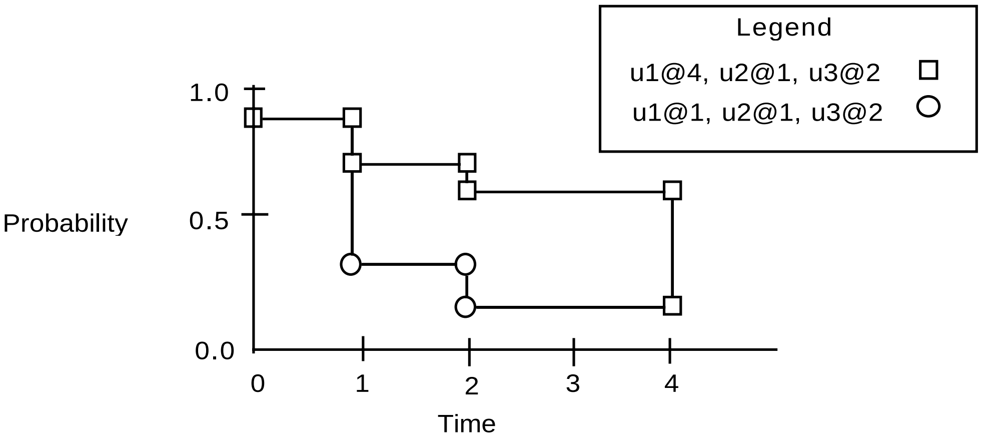


Figure 17 Example of a Probability Profile for the Join

5 SUMMARY AND FUTURE DIRECTIONS

We have introduced the formal concepts that permit the introduction of time to influence nets. Influence nets are basic static equilibrium models. They have no concept of time. It was shown that in order to correctly capture the behavior of an influence net with time delays, they must be converted to a discrete event system model. The most general model of discrete event systems, the colored Petri net, is a suitable modeling formalism. Three similar CP nets that can model the behavior of input, output, and intermediate nodes of the net have been proposed. A method of interconnecting the three types of CP nets to form a complete model of the timed influence net was demonstrated. State space analysis of the CP net components was conducted to analyze their behavior. This analysis demonstrated that the CP net construct behaves correctly and can be used with confidence to model any timed influence net.

Future research areas include developing ways to use the CP net models to support effects based planning and dynamic re-planning of actions that support Courses of Action. We plan to investigate methods for using these models to track and measure progress of actions as situations unfold. The goal is to determine how the information produced by these models on the changing potential for achieving desired effects and results can be used in dynamic re-planning.

ACKNOWLEDGEMENTS

The authors wish to thank the anonymous referees for their review and comments which significantly improved the quality and accuracy of this paper.

REFERENCES

[1] Cassandras, C. G. (1993), *Discrete Event Systems, Modeling and Performance Analysis* , Aksen Associates Incorporated, Boston MA.

[2] Friedman, N and Goldszmidt, M. (1996), Learning Bayesian Networks with Local Structure, *Uncertainty in Artificial Intelligence: Proceedings of the 12th Conference*, Morgan Kaufmann.

[3] Jensen, F. V. (1996) An Introduction of Bayesian Networks, UCL Press limited, London

[4] Jensen K. (1992). *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* Springer-Verlag, Berlin, Germany.

[5] Jensen K. (1997). *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volumn 1. Basic Concepts*. Monographs in Theoretical Computer Science, Springer-Verlag, Berlin, Germany.

[6] Jensen K. (1997). *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volumn 2. Analysis Methods* Monographs in Theoretical Computer Science, Springer-Verlag, Berlin, Germany.

[7] Jensen K. (1997). *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use. Volumn 3. Practical Use*. Monographs in Theoretical Computer Science, Springer-Verlag, Berlin, Germany.

[8] Kristensen, L. M., Christensen, S, and Jensen K. (1998) The Practitioner's Guide to Coloured Petri Nets, Software Tools for Technology Transfer.

[9] Murata, T. (1989). Petri Nets: Properties, Analysis and Applications, *Proceedings of the IEEE,* Vol. 77, No. 4, April 1989.

[10] Pearl, J. and Verma, T. S. (1991), A Theory of Inferred Causation, in Allen, J., Fikes, R. and Sandewall, E. (Eds.) *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference*, San Mateo, CA Morgan Kaufmann.

[11] Rosen, J. A. and Smith, W. L. (1996) *Influence Net Modeling with Causal Strengths: An Evolutionary Approach*, Proceedings of the Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey CA, June 25-28, 1996.

[12] Wagenhals, L. W., Shin, I, and Levis, A. H. (1998) Creating Executable Models of Influence Nets with Colored Petri Nets, International Journal for Software Tools for Technology Transfer, vol. 2 no. 2, pp 168-181, Springer-Verlag.

Using Coloured Petri Nets to Investigate Behavioural and Performance Issues of TCP Protocols*

Jorge C. A. de Figueiredo[†‡] and Lars M. Kristensen[‡]

[†] Departamento de Sistemas e Computação, UFPB, Brazil
E-mail: abrantes@daimi.au.dk

[‡] CPN Group, Department of Computer Science, University of Aarhus, Denmark
E-mail: kris@daimi.au.dk

Abstract

This paper deals with modelling and analysis of the Transmission Control Protocol (TCP). The TCP protocol is the transport protocol used on the Internet. We present a generic Coloured Petri Net model (CPN model) suited for performance and behavioural analysis of different versions of the TCP protocol. The CPN model covers connection establishment and termination, data transfer phase, and concepts such as slow start, congestion avoidance, fast retransmit, and fast recovery.

As a representative example we show how the CPN model can be instantiated to analyse two different versions (TCP Reno and TCP Tahoe) of the TCP protocol. By means of simulations of the CPN model, we investigate the behaviour of both TCP versions when they face different packet loss situations, and we investigate the impact of packet loss on the throughput obtained with the TCP Reno and TCP Tahoe protocols.

1 Introduction

The Internet has over the past decade experienced a crescent and rapid growth. Despite the continuous and significant advances in Internet technology, severe congestion problems have appeared with the growth of Internet applications. One of the main causes for congestion problems is the transport protocol implementations [11]. The *Transmission Control Protocol (TCP)* is the transport protocol used on the Internet. Different versions of the TCP protocol have been defined [8, 10, 22] in order to cope with congestion problems.

The window flow control in TCP is one of the mechanisms for handling congestion on the Internet. Basically, the window flow control manages the demand on the receiver's capacity. Four intertwined algorithms have been defined and introduced in modern TCP versions. They can act as a preventive way to avoid congestion as well as a congestion recovery scheme to restore an operating state, when faced with unexpected changes such as an increase in traffic and loss of packets. These algorithms, named *slow start*, *congestion avoidance*, *fast retransmit* and *fast recovery*, have been successfully applied and widely adopted. Some of them are classified as mandatory for TCP implementations.

*This work was partially supported by CNPq/Brazil under grant 201462/91-5 (NV).

Many case studies have been conducted to investigate the performance of the different versions of TCP over congested networks. Kumar [17] uses a stochastic model to predict the throughput of different versions of TCP, considering the presence of random losses on a wireless link in a local area network. Lakshman and Madhow [18] examine the TCP performance over wide area networks when data traffic coexists with real-time traffic. Fall and Floyd [8] present the benefits of adding selective acknowledgements and a selective strategy to TCP. Fall and Floyd used simulation to make comparisons between different TCP versions. Goyal et al [9] study the design issues for improving TCP performance over an *ATM Unspecified Bit Rate (UBR)* service. Simulation is used to obtain different performance measures and TCP is analysed over different switch drop policies. Ost and Haverkort [21] use a Stochastic Petri Net model to evaluate performance of windowing mechanisms in world-wide web applications.

In this paper we apply hierarchical Coloured Petri Nets (CP-nets or CPNs) [12–14, 16] for modelling and simulation-based performance analysis of TCP protocols. For construction and simulation of the CPN models we use the Design/CPN tool [2, 20]. The Design/CPN tool has previously been used in a number of projects on performance analysis, e.g., in the areas of high-speed interconnects [3] and ATM networks [4, 5]. The reader is assumed to be familiar with the basic concepts of high-level Petri Nets [15].

The primary objective of this paper is to present a *generic* CPN TCP model to analyse the performance and the behaviour of TCP protocols. The model incorporates aspects such as *slow start*, *congestion avoidance*, *fast retransmit* and *fast recovery.* The model has been constructed in such a way that different TCP versions can be analysed. Performance analysis can be carried out by conducting lengthly simulations of the CPN model. Different kind of performance measures can be defined and observed. Such model and simulation-based performance analysis can be applied to investigate *what-if* questions, and used as a test-bed for evaluation of different aspects and variations of the TCP protocol. As a representative example, we compare the behaviour of the two most common TCP versions (*TCP Reno* and *TCP Tahoe*) when they face different packet loss situations. We also consider performance analysis for both TCP versions.

This paper is organised as follows. Section 2 introduces the basic concepts of TCP protocols with emphasis on the data transfer part and the concepts of *slow start*, *congestion avoidance*, *fast retransmit*, and *fast recovery.* Section 3 presents the developed CPN TCP model. Section 4 describes the simulation scenario. Section 5 presents the analysis performed for the *TCP Reno* and *TCP Tahoe* protocols for different packet loss situations. Finally, in Sect. 6 we sum up the conclusions.

2 The Transmission Control Protocol

The Transmission Control Protocol (TCP) provides a connection-oriented, reliable, byte stream service [6, 22]. A connection is initialised when two processes want to communicate. When a connection is established, the TCP protocol is able to transfer a continuous stream of octets. Octets are packed into segments[1] for transmission over the Internet. When the communication is completed, the connection is terminated. TCP connections are full duplex, i.e., messages can flow in both directions. The focus of this paper is on the data transfer part. Therefore, we do not consider the connection part in any detail. The reader interested in the connection part can refer to [7, 22].

[1] Throughout this paper we will use the terms *segment* and *packet* interchangeably.

During the data transfer part, each segment transfered in a TCP connection is given a unique sequence number. The sliding window strategy is employed to provide efficient transmission and flow control. The TCP sender keeps for each connection, the necessary information in order to guarantee the correct behaviour of the sliding window strategy. The TCP sender also maintains a variable that contains the maximum receiver window size indicating the buffering capacity of its counterpart.

The TCP receiver can accept out-of-order packets. The TCP receiver has a finite buffer where packets can be stored. The packets must be delivered in correct sequence to its TCP user. The TCP receiver returns an acknowledgement for every packet successfully received. The acknowledgement contains the sequence number for the next packet it expects and the current maximum window size the TCP receiver can cope. It is important to point out that the acknowledgements are cumulative, i.e., an acknowledgement with sequence number n indicates that all packets up to and including $n-1$ were successfully received.

TCP must recover from data that is damaged, lost, duplicated, or delivered out of order by the Internet communication system. Basically, when TCP sends a segment it maintains a timer waiting for an acknowledgement. The timer is set to a *Retransmission Time-Out (RTO)* value [6], which is defined based on a running estimate of the packet *Round-Trip Time (RTT)*. The segment is retransmitted if no acknowledgement is received within this time.

The basic ideas discussed above are common to all TCP versions. However, to improve TCP throughput, many modern TCP versions incorporate modified flow control procedures to limit the number of packets in the network. We discuss such procedures in the following subsections.

2.1 Slow Start and Congestion Avoidance

Slow start and *congestion avoidance* are two independent algorithms used to treat computer networks congestion problems [11]. Although, the two algorithms are independent and have different objectives, they are in practice implemented together [22, 23].

The basic idea behind *slow start* is that the packets rate injection into the network at the sender side is based on the rate at which acknowledgements are returned by the receiver side. Thus, instead of injecting multiple packets into the network as performed by old TCPs implementations, the sender starts by transmitting few segments[2]. As soon as it receives an acknowledgement, the number of segments to be sent is gradually increased. This prevents the sender from overwhelming the network with a large amount of traffic, which is likely to cause segment losses. Actually, two segments are allowed to be sent for each acknowledgement received.

A packet can be lost either by damage in transit or by congestion in the network. Losses due to damage are extremely rare [11]. So, it is assumed that a packet loss indicates congestion in the network. *Congestion avoidance* was defined as a way to cope with the loss of packets. Thus, if congestion is detected the sender must have some strategy to decrease its utilisation of network. Congestion is detected at the sender side by a timeout or by the reception of duplicate acknowledgements (*dupACKs*)[3].

[2] As originally defined, the sender starts transmission in the slow start phase by sending one segment [11]. However a larger initial window was already suggested and evaluated in [1].

[3] Notice that the concept of duplicate acknowledgment does not refer to the communication channel duplicating the acknowledgement.

To implement these algorithms, a congestion window size variable (*CWND*) is kept at the sender as a measure of the capacity of the network. Moreover, a slow start threshold variable (*SSTHRESH*) is maintained to distinguish between *slow start* and *congestion avoidance* phases. The maximum number of data that the sender can send is defined by the minimum of *CWND* and the receiver window size (*RemoteWindow*). If $CWND < SSTHRESH$, then the connection is in *slow start* phase. Otherwise, *congestion avoidance* is performed. The sender starts transmission in the *slow start* phase by sending one segment. When the sender receives an acknowledgement for a new segment, *CWND* is incremented by 1. This means that *CWND* is doubled every round trip time. Therefore, *slow start* phase corresponds to an exponential increase in the number of data which can be sent.

When congestion is detected, one-half of the current congestion window size is saved in $SSTHRESH$. The $SSTHRESH$ value should be at least two segments. Additionally, if congestion was triggered by a timeout, then *CWND* is set to 1 and the *slow start* phase starts.

During *congestion avoidance* phase, the sender increases its $CWND$ variable by $1/CWND$ every time an acknowledgement is received. Differently from *slow start*, a linear increase in the number of data allowed to be sent is observed in this phase.

2.2 Fast Retransmit and Fast Recovery

As stated previously, whenever the TCP receiver receives new data, it sends an acknowledgement to the TCP sender specifying the sequence number for the next expected packet. However, if an out-of-order packet is received (indicating a potential packet loss), a *dupACK* is immediately sent to the sender [23].

The idea behind *fast retransmit* is to realize, as soon as possible, if a segment has been lost. It is assumed that if the cause for the duplicate acknowledgements is just a reordering of segments, the number of duplicate acknowledgements is very small. Effectively, the sender waits for a fixed small number K of *dupACKs*. Typically, K is set to 3. If more than K *dupACKs* are received, the TCP sender concludes that the segment indicated in the duplicate acknowledgements has been lost. Immediately, the sender retransmits the segment what appears to be the segment lost. At this point, the sender reduces its $CWND$ variable by half plus K segments. The addition of K is based on the assumption that K more packets have successfully left the network. Also half of the original *CWND* is saved to $SSTHRESH$. For each additional *dupACK*, the sender increases $CWND$ by one and sends a new segment whenever possible.

When a new acknowledgement is received for the retransmitted segment, *congestion avoidance* is performed instead of *slow start*. This enhancement is known as *fast recovery* and contributes to a higher throughput under moderate congestion. Thus, instead of setting $CWND$ to one segment as in a regular time-out situation, TCP sets $CWND$ to $SSTHRESH$ and performs *congestion avoidance*. This occurs approximately one *RTT* after the lost segment is retransmitted.

2.3 TCP Versions

Different TCP versions have been proposed. Basically, the various TCP versions differ in the way they recover from the loss of packets. Below we summarise a number of TCP versions. Details about the different versions can be found in [8–10, 17, 22].

TCP Vanilla: Incorporates *slow start* and *congestion avoidance* aspects. The recovery of a packet loss is performed in the original way, i.e., the TCP sender waits for a coarse timeout to retransmit the packet. After retransmission, *slow start* is performed.

TCP Tahoe: *Fast retransmit*, *slow start* and *congestion avoidance* are considered. This version tries to conclude as soon as possible that a segment has been lost. When more than K *dupACKs* are received, it behaves as if a timeout has occurred and begins retransmission. However, after retransmission, *slow start* is performed.

TCP Reno: Similar to *TCP Tahoe* but takes *fast recovery* into consideration. This means that instead of slow-starting, the TCP Reno sender makes some estimates of the amount of outstanding data upon reception of additional incoming duplicate acknowledgements. The *TCP Reno* version is said to be conservative [17] because it retransmits only one segment, even in case of multiple packet losses in one window.

TCP New Reno: A slight variation of *TCP Reno* that eliminates the approach in case of multiple losses. In such a situation, when the fast retransmission is first triggered, the sender saves the highest sequence number sent. When a new acknowledgement is received, it verifies if the acknowledgement includes all the segments sent. If so, the sender acts as in the *TCP Reno* version by setting $CWND$ to $SSTHRESH$ and performing *congestion avoidance*. On the other hand, if not all segments were acknowledged, the sender immediately retransmits the next segment that appears to be lost. This continues until all the segments are acknowledged.

TCP SACK: Incorporates the *selective acknowledgement* approach to efficiently recover from multiple segment losses [8]. In this version, the information carried in the acknowledgements is more elaborate and contains additional details about the segments which have been received by the TCP receiver. From this information, the TCP sender can infer about the segments that were not received by its counterpart.

3 The CPN TCP Model

This section contains a detailed description of the CPN TCP model. Considering that the primary purpose of TCP is to provide a reliable data transfer and connection oriented service between pairs of processes, four aspects were considered in the model as suggested in [7]: connection, basic data transfer, flow control, and reliability. The CPN TCP model is a timed hierarchical CPN model and was developed in such a way that different TCP versions can be analysed with only minor modifications to the CPN model.

Figure 1 provides an overview of the CPN TCP model. The CPN TCP model is divided in two main parts: the connection part (right part of the hierarchy page) and the data transfer

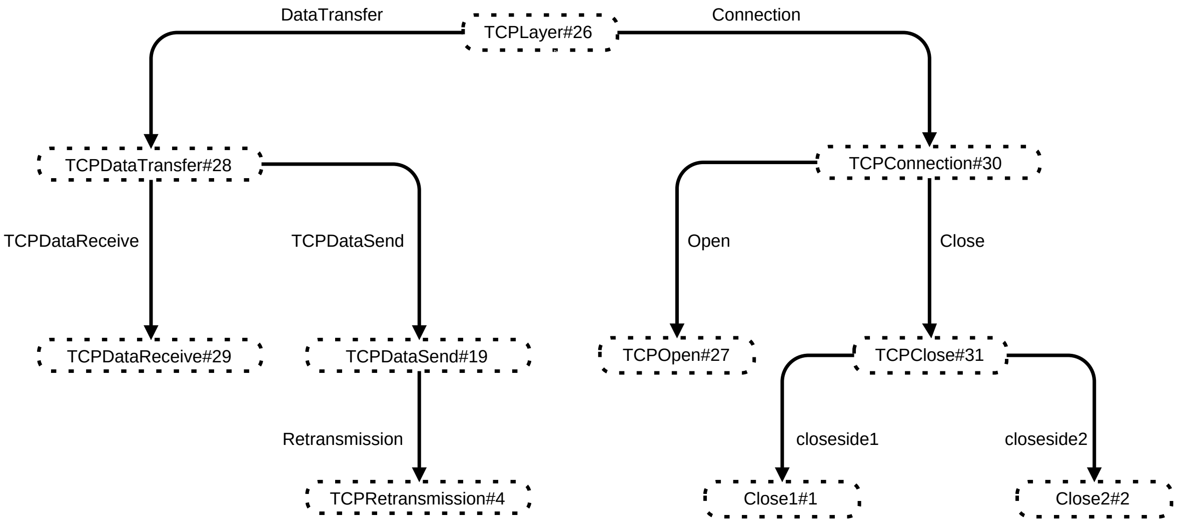


Figure 1: The Hierarchy page for CPN TCP model

part (left part of the hierarchy page). The TCP connection part is divided in two subparts. Page *TCPOpen* models the three-way handshake procedure used for TCP connection establishment. Page *TCPClose* and its subpages *Close1* and *Close2* model the termination of a TCP connection.

The data transfer phase is modelled by the subpages of *TCPDataTransfer*. During this phase, the two processes involved in the connection exchange messages. Recall that each message is packed in segment(s) which are transfered via the communication system. Segments are reassembled in the receiver side to reconstitute the original message. An acknowledgement is required for each segment successfully received. A TCP process is not only exclusively a sender or a receiver, i.e., each process must be able to act as a sender and also as a receiver. The TCP data transfer part is therefore divided in a data send part (pages *TCPData Send* and *TCPRetransmission*) and a data receive part (page *TCPDataReceive*).

Although a complete TCP model has been developed, we present a simplified version in this paper. Since the main objective of this paper is to analyse TCP performance and behaviour during the data transfer phase, the entire TCP connection phase was omitted. It is assumed that the connection is successfully initialised and all connection information accordingly set. The data transfer part of the model is described in the following subsections.

3.1 The Transmission Control Block

Much information is generated when a connection is opened. For example, the processes should agree on the initial sequence numbers. This information is updated when data transmission occurs. The *Transmission Control Block (TCB)* is the data structure where the connection information is stored [7]. In the CPN model, TCB is modelled by an accordingly named place. There is one token on this place for each of the currently open TCP connections. Figure 2 lists the definition of the colour set (type) *TCB*.

Id is a pair of endpoints that identifies a connection. *LocalWindow* determines the maximum number of bytes the process can receive. This information should be included in all data

```
color TCB = record Id: Connection *
                   LocalWindow: Int *
                   RemoteWindow: Int *
                   CWND: Int *
                   SSTHRESH: Int *
                   SN: Int *
                   SNm: Int *
                   RN: Int *
                   NACK: CongAvoid *
                   NSeg: Int *
                   RNStatus: Status *
                   ConState: States;
```

Figure 2: The TCB colour set

segments that are transmitted. *RemoteWindow* indicates the maximum number of bytes the remote process can receive. This information is used to determine the data segment size.

CWND and *SSTHRESH* keep current information about congestion window size and threshold, respectively. These values are used to perform *fast retransmit* and *fast recovery* as explained previously. *SN*, *SNm*, and *RN* are three attributes used to control the flow of segments and to define the sliding window. *SN* indicates the next segment to be sent. *SNm* indicates the sequence number of the segment its counterpart is waiting for. This information is important to determine if a new segment can be sent as well as in the retransmission mechanism. *RN* indicates the segment the local process is expecting to receive. The *NACK* attribute indicates if the connection is performing *fast retransmit*. Moreover, when receiving *dupACKs*, it keeps track of the number of *dupACKs* which have been received. *NSeg* is an auxiliary attribute used to update *CWND*, in case of *congestion avoidance*. *RNStatus* can assume two values: *acked* or *notacked*. As soon as a new segment is received, *RNStatus* is set to *notacked*. When the acknowledgement is sent, it changes to *acked*. An acknowledgement is sent only when *RNStatus* is set to *notacked*. *ConState* indicates either a connection phase or a data transfer phase.

3.2 Sending Segments

In the TCP model two different types of segments are defined: *ack segments* and *data segments*. *Ack segments* carry only the sequence number of the next expected segment. *Data segments* are more complex. Besides its sequence number, each *data segment* carries the remote window size information. Remember that since data can be sent in both directions, a data segment could be used to acknowledge receipt of a segment.

Figure 3 depicts page *TCPDataSend* modelling the transmission of segments. In this paper, the part of the model responsible for the fragmentation of data from the TCP user layer was simplified. It was assumed that the TCP sender is an infinite TCP source, i.e., TCP always sends a segment whenever allowed by the window. All segments are also assumed to have the same size. The part of page *TCPDataSend* modelling access to *TCB* has been highlighted using dashed lines and arcs.

Occurence of transition *Accept* (top) indicates the acceptance of a new segment to be sent,

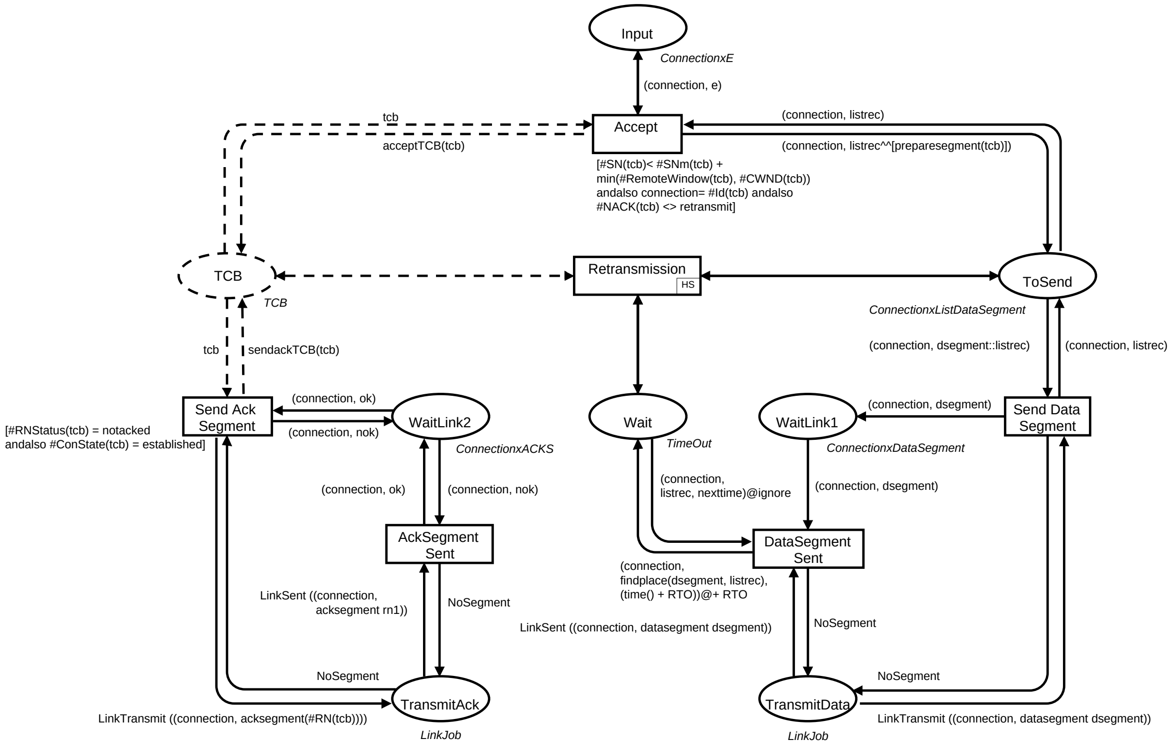


Figure 3: The *TCPDataSend* page

i.e., the window is not full. According to the sliding window strategy, a new segment will be accepted if its sequence number is less than the last sequence number acknowledged plus the minimum value of the remote window and congestion window sizes. The guard of transition *Accept* guarantees that this condition is satisfied. If transition *Accept* occurs, a new (token) segment is created by the function *preparesegment* and deposited on place *ToSend*. Also, TCB is updated by function *acceptTCB*. Place *ToSend* keeps a list of data segments to be sent. A FIFO policy is adopted.

Occurence of transition *Send Data Segment* (right) represents that the segment is being enqueued in the transmission link buffer. A copy of the data segment is kept on place *WaitLink1*. When the *data segment* is effectively enqueued, transition *DataSegment Sent* occurs and a token is deposited on place *Wait*. Place *Wait* maintains a list of *data segments* sent that will be used for retransmission when necessary (subpage *Retransmission*). This is a timed place and the Retransmission Time Out (*RTO*) value is used to determine the timestamp for its tokens.

Occurence of transition *Send Ack Segment* (left) represents the sending of an *ack segment*. This is possible when the status for the reception of segments is *notacked*, as indicated in the guard of the transition. After sending an *ack segment*, the status is set to *acked* by the function *sendackTCB*. Place *WaitLink2* and transition *AckSegment Sent* has a similar functionality as *WaitLink1* and *DataSegment Sent*.

Figure 4 depicts page *Retransmission*. This it the subpage of the substitution transition *Retransmission* in Fig. 3. Two different forms of retransmission are modelled: a timeout retransmission or a fast retransmission. Transition *Timeout* models the regular retransmission,

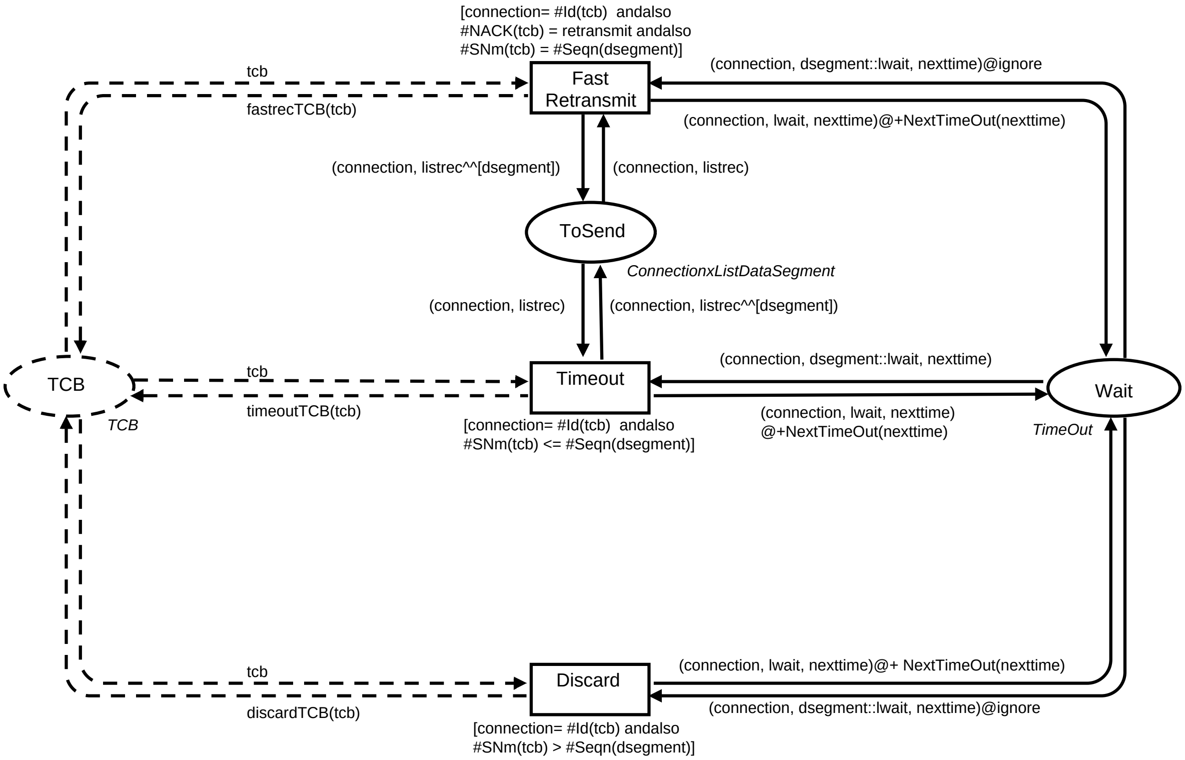

[connection= #Id(tcb) andalso
#NACK(tcb) = retransmit andalso
#SNm(tcb) = #Seqn(dsegment)]
tcb
fastrecTCB(tcb)
Fast Retransmit
(connection, dsegment::lwait, nexttime)@ignore
(connection, lwait, nexttime)@+NextTimeOut(nexttime)
(connection, listrec^^[dsegment])
(connection, listrec)
ToSend
ConnectionxListDataSegment
(connection, listrec)
(connection, listrec^^[dsegment])
TCB
TCB
tcb
timeoutTCB(tcb)
Timeout
[connection= #Id(tcb) andalso
#SNm(tcb) <= #Seqn(dsegment)]
(connection, dsegment::lwait, nexttime)
(connection, lwait, nexttime)
@+NextTimeOut(nexttime)
Wait
TimeOut
tcb
discardTCB(tcb)
Discard
[connection= #Id(tcb) andalso
#SNm(tcb) > #Seqn(dsegment)]
(connection, lwait, nexttime)@+ NextTimeOut(nexttime)
(connection, dsegment::lwait, nexttime)@ignore

Figure 4: The *Retransmission* page

i.e., when no acknowledgement is received and the timeout timer expires. After occurence of transition *Timeout*, the TCB attributes are updated accordingly by the function *timeoutTCB*. For example, *CWND* and *SSTHRESH* are set to 1 and half of the current *CWND*, respectively.

The TCP *fast retransmit* aspect is modelled by the transition *Fast Retransmit*. This transition occurs only when K *dupACKs* have been received. When the *NACK* attribute in *TCB* has the value *retransmit*, indicating that K duplicate acknowledgements have been received, transition *Fast Retransmit* can occur. Function *fastrecTCB* updates *CWND* and *SSTHRESH* as described in Sect. 2.2. Transition *Discard* removes segments from the list on place *Wait* whenever segments are acknowledged.

3.3 Receiving Segments

Page *TCPDataReceive* modelling the reception of segments is shown in Fig. 5. Transitions *Receive Ack Segment* (right) and *Receive Data Segment* (left) model the reception of *ack segments* and *data segments*, respectively. When an *ack segment* is received, function *ackrecTCB* updates *TCB*. Minor adjustments in function *ackrecTCB* are required in order to consider different versions of TCP.

If a *data segment* is received the TCB information is updated by the function *datarecTCB*. Moreover the segment received is stored on the place *Segments Received*. A special function *updatelr* is used in order to verify if the segment was already received or if it was a new one. This function also reassembles the messages before they are delivered to upper protocol

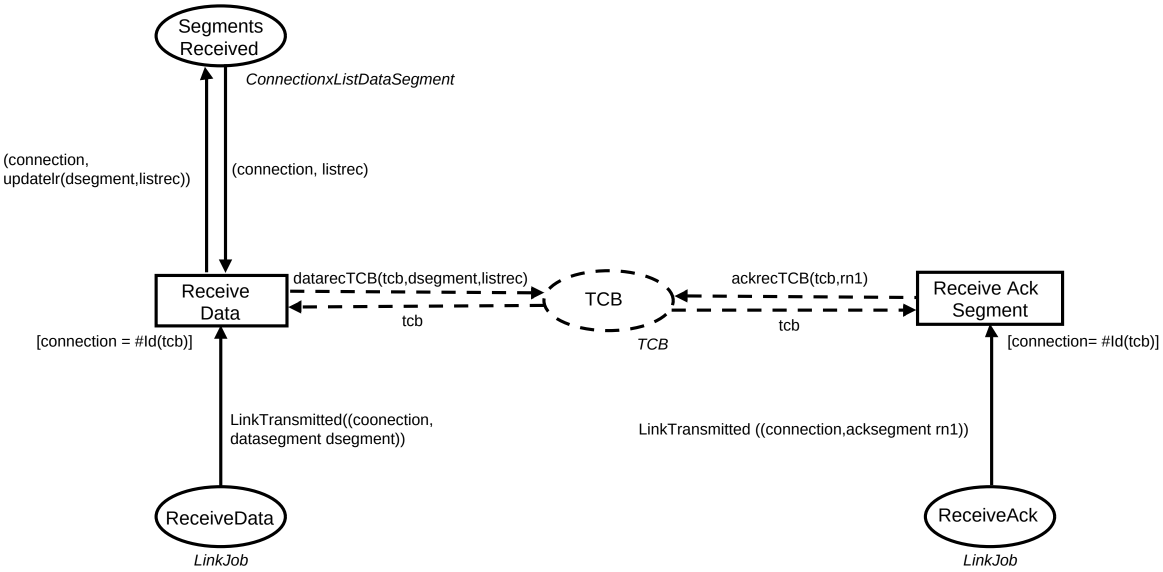

Segments Received
ConnectionxListDataSegment
(connection, updatelr(dsegment,listrec))
(connection, listrec)
Receive Data
datarecTCB(tcb,dsegment,listrec)
tcb
TCB
TCB
ackrecTCB(tcb,rn1)
tcb
Receive Ack Segment
[connection = #Id(tcb)]
[connection= #Id(tcb)]
LinkTransmitted((coonection, datasegment dsegment))
LinkTransmitted ((connection,acksegment rn1))
ReceiveData
LinkJob
ReceiveAck
LinkJob

Figure 5: The *TCPDataReceive* page

layers.

4 Simulation Scenario and Model

The simulation scenario considered was based on the scenario used in [8] by Fall and Floyd to compare different TCP versions.

Figure 6 shows the network topology for the considered simulation scenario, i.e., the environment in which we study the TCP Reno and the TCP Tahoe protocols. The bandwidth between the *Sender* and the *Switch* is $8Mbps$ (Mbit per second) and the link delay is $0.1ms$. The bandwidth and delay for the link between the *Switch* and the *Receiver* are $0.8Mbps$ and $100ms$, respectively.

Data are sent exclusively in one direction, from the *Sender* to the *Receiver*. A finite-buffer drop tail switch was considered. Also, three TCP connections from the *Sender* to the *Receiver* was considered, however only the first connection was analysed. The other two connections were used only to achieve the desired pattern of drops to be analysed (1, 2 or 3 packet losses in one window of packets). The pattern of packet drops is changed by the number of packets sent by the second and third connections. Thus, simulations and analysis were performed considering three situations: one packet loss (packet number 14), two packet losses (packets 14 and 28) and three packet losses (packets 14, 26 and 28).

Figure 7 provides an overview of the model considered for simulation. It represents an in-

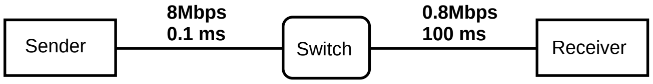


Figure 6: Simulation scenario

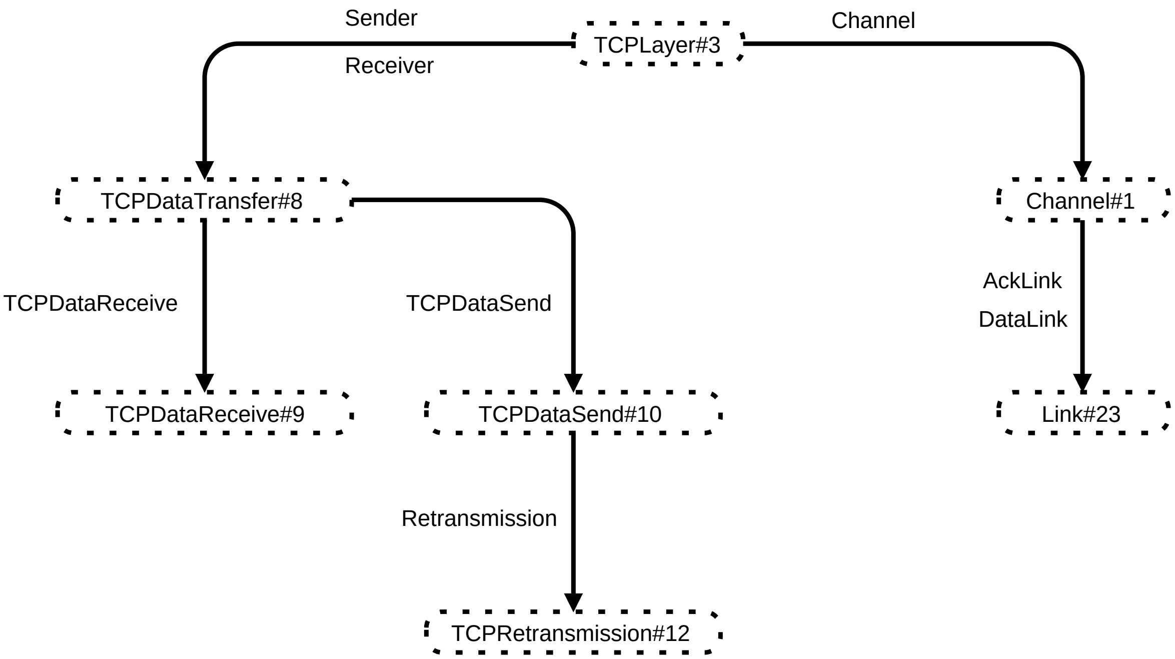


Figure 7: The hierarchy page for the simulation model

stantiation of the CPN TCP model presented in the previous section in such a way that the two TCP versions can be studied. The model consists of two instances of page *TCPDataTransfer*: one representing the *Sender* and one representing the *Receiver*. In page *TCPDataReceive* the function *ackrecTCB* was defined differently for the two TCP versions. In case of TCP Reno, *ackrecTCB* was set to define *fast retransmit* and *fast recovery* concepts. For the TCP Tahoe, *ackrecTCB* does not consider *fast recovery*. In both cases, the congestion window threshold K was set to 3.

The switch and link bandwidth capacities and delays are modelled by the pages *Channel* and *Link*. Figure 8 depicts page *Link*. Packets to be transmitted are represented by *LinkTransmit* tokens in place *Incoming Packets*. Occurence of transition *Packet Arrival* indicates that the packet was transmitted. Function *LinkResourceJobArrival* checks the availability of space in the buffer. If there is no space left, the packet is dropped. A *LinkSent* token is deposited back in place *Incoming Packets* with timestamp set to the transmission time value. This indicates that it is necessary to wait at least the transmission time to be able to transmit a new packet. Transition *Schedule Packet* occurs whenever the timestamp of the token in place *LinkResource* is satisfied. This timestamp is set based on the link delay value.

The Design/CPN tool [20] was used to simulate the model. The performance facilities [19] of the Design/CPN simulator were used to define the analysis functions and to perform data collection. Basically two functions were defined: a function that keeps track of all data sent by the TCP sender and a function that determines the number of packets successfully delivered at the receiver side.

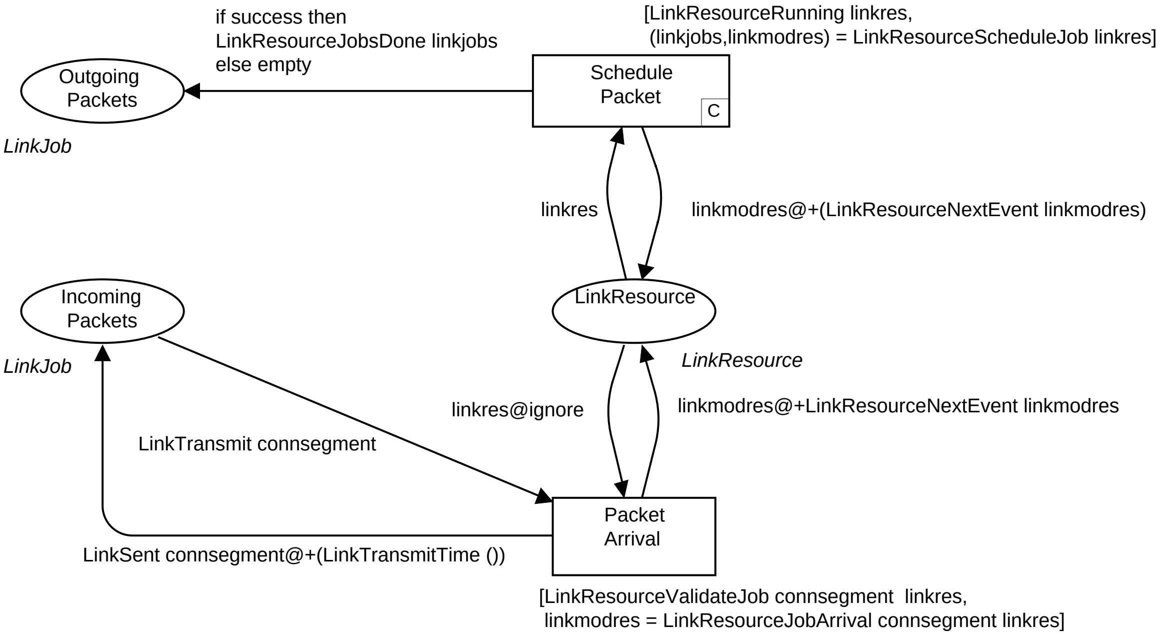

if success then
LinkResourceJobsDone linkjobs
else empty
[LinkResourceRunning linkres,
(linkjobs,linkmodres) = LinkResourceScheduleJob linkres]
Outgoing Packets
LinkJob
Schedule Packet
C
linkres
linkmodres@+(LinkResourceNextEvent linkmodres)
Incoming Packets
LinkJob
LinkResource
LinkResource
linkres@ignore
linkmodres@+LinkResourceNextEvent linkmodres
LinkTransmit connsegment
Packet Arrival
LinkSent connsegment@+(LinkTransmitTime ())
[LinkResourceValidateJob connsegment linkres,
linkmodres = LinkResourceJobArrival connsegment linkres]

Figure 8: CPN model for the link

5 Analysis of the Model

Simulation was performed considering one, two and three packet loss situations. For each situation, the sequence of packets sent by the TCP sender was observed. In addition, the number of packets delivered was obtained and a performance measure was computed.

5.1 Behavioural Analysis

To observe the behaviour of the two TCP versions, we defined a graph that shows the sequence of packets sent from the *Sender* to the *Receiver*. We have time on the x-axis, and the packet number (mod 60) on the y-axis. A square represents each packet as it arrives in the switch. Packet losses are represented in the figure by a cross mark.

One Packet Loss

We consider the situation where the packet with sequence number 14 is lost. Figures 9 and 10 show the sequence of packets sent by the *Sender*, considering the *TCP Reno* and *TCP Tahoe* versions respectively.

In case of *TCP Reno*, packets 0-13 are sent without problems (14 square marks in the four initial windows in Figure 9). During this phase, TCP performs slow start and the congestion window increases exponentially from 1 to 15. Packet number 14 is dropped, represented by a cross mark in Fig. 9. This means that all the packets for the fourth window were successfully delivered, except packet 14. Thus, the TCP sender receives 7 *ack segments* and it increases *CWND* from 8 to 15. The TCP sender is allowed to send packets 15-28 (fifth window in Figure 9).

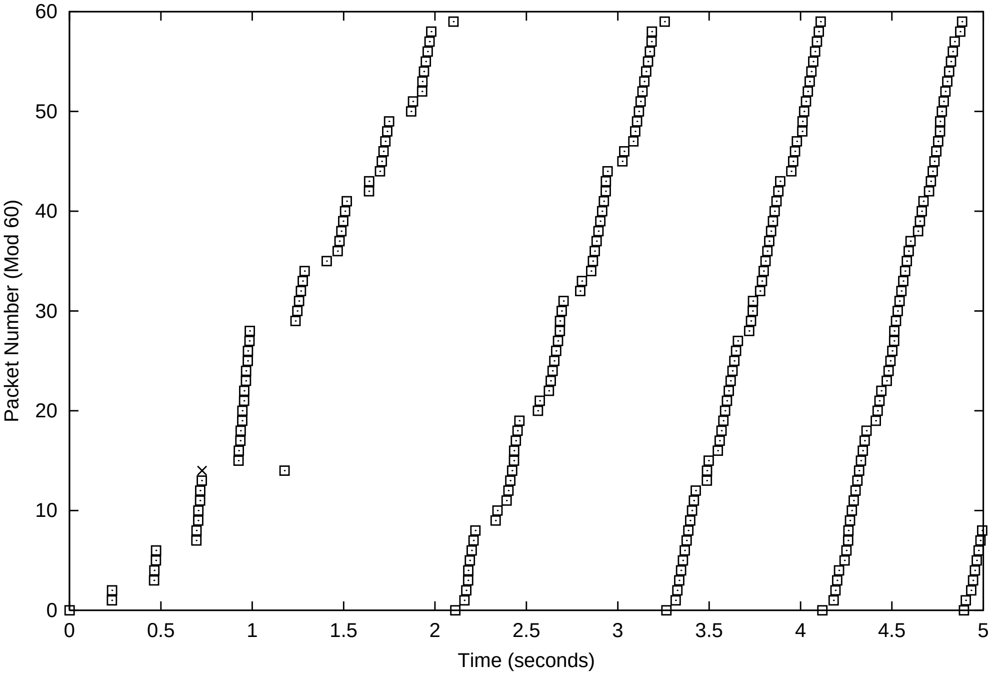


Figure 9: TCP Reno - one packet loss

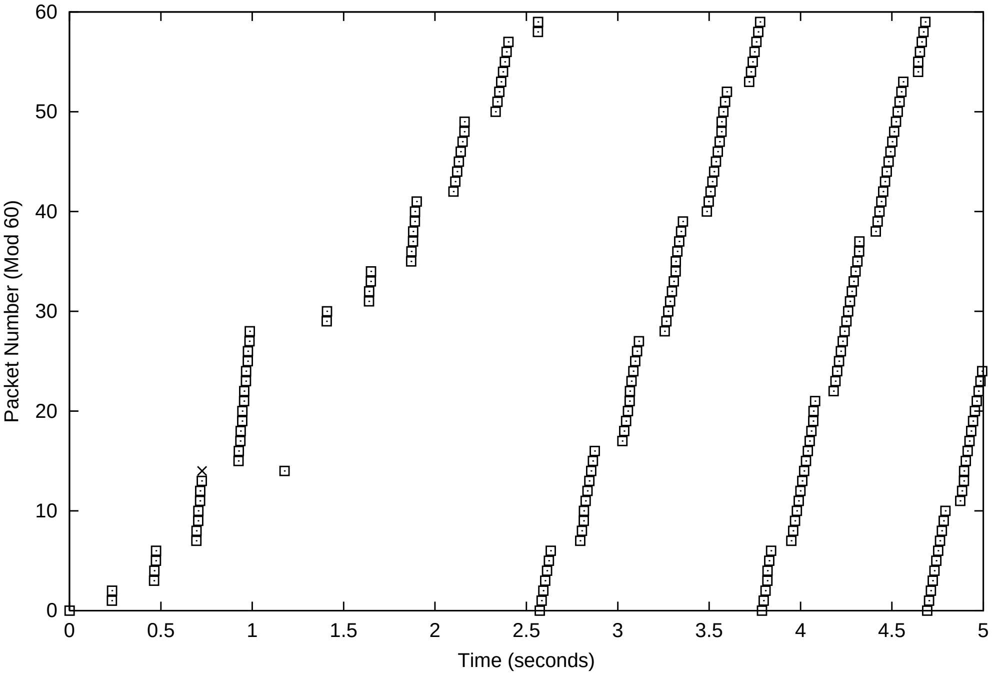


Figure 10: TCP Tahoe - one packet loss

When the receiver successfully receives packet number 13, it acknowledges reception by asking for packet number 14. Packets 15-28 were successfully delivered. Therefore, besides the first acknowledgement asking for packet 14, the sender receives 14 additional acknowledgements for this same packet (*dupACKs*).

After reception of the third *dupACK* asking for packet 14, fast retransmission is performed. At this point, *SSTHRESH* is set to 7 (half of *CWND*) and *CWND* is set to 10 (half plus K). Packet 14 is then retransmitted. Since 11 *dupACKs* were received after retransmission of packet 14, *CWND* is increased up to 21.

When *CWND* is increased for the last six *dupACKs*, the sender is allowed to send packets 29-34. When the retransmitted packet is received, the sender exits fast recovery and starts congestion avoidance with *CWND* set to 7.

In case of *TCP Tahoe*, the behaviour is identical to *TCP Reno's* behaviour until the reception of the third *dupACK* for packet 14. Then, the sending *TCP Tahoe* reduces it congestion window to one and retransmits packet 14. When an acknowledgement is received for the retransmitted packet, *CWND* is increased by 1 and packets 29 and 30 are sent. The sender continues in *slow start* and when packet 34 is sent, the slow-start threshold is reached and *congestion avoidance* starts.

Two Packet Losses

Figures 11 and 12 show the sequence of packets sent for the two packet loss situation considering *TCP Reno* and *TCP Tahoe*, respectively. The two lost packets are 14 and 28, as shown by the cross marks in Figures 11 and 12.

The protocol behaves exactly the same as in the one drop situation until packet 28 is sent. Since packet 28 is also lost, the number of duplicate acknowledgements asking for packet 14 is 13 instead of 14 for the one-drop situation.

Considering the *TCP Reno* protocol, the sender is able to send packets 29-33. After reception of the third duplicate acknowledgement, *CWND* drops to 10 and increases up to 20 due to the reception of the last 7 duplicate acknowledgements asking for packet 14. When the retransmitted packet 14 is acknowledged, packet 28 is expected. Since this is the first acknowledgement that asks for packet 28, the sender is allowed to send a new packet (packet 34). At this point, the sender exits fast recovery with *CWND* of 7.

Since packet 28 is also lost, the sender will receive 6 duplicate acknowledgements asking for packet 28. Reception of the third duplicate acknowledgement triggers a second fast retransmission situation. *CWND* and *SSTHRESH* are set to 6 and 3 respectively. *CWND* increases to 9 when the sender receives the sixth duplicate acknowledgement. Then, the sender is able to send packets 35 and 36. When the acknowledgement for the second retransmitted packet is received, the sender exits again fast recovery with a congestion window of three.

In the case of *TCP Tahoe* protocol, after the reception of the third *dupACK* for packet 14, *CWND* is reduced to 1 and packet 14 is retransmitted. The sender then receives an acknowledgement asking for packet 28 and *CWND* is increased by one. Packets 28 and 29 are sent and the sender continues in the *slow start* phase. The *congestion avoidance* phase starts when the sender sends packet 40.

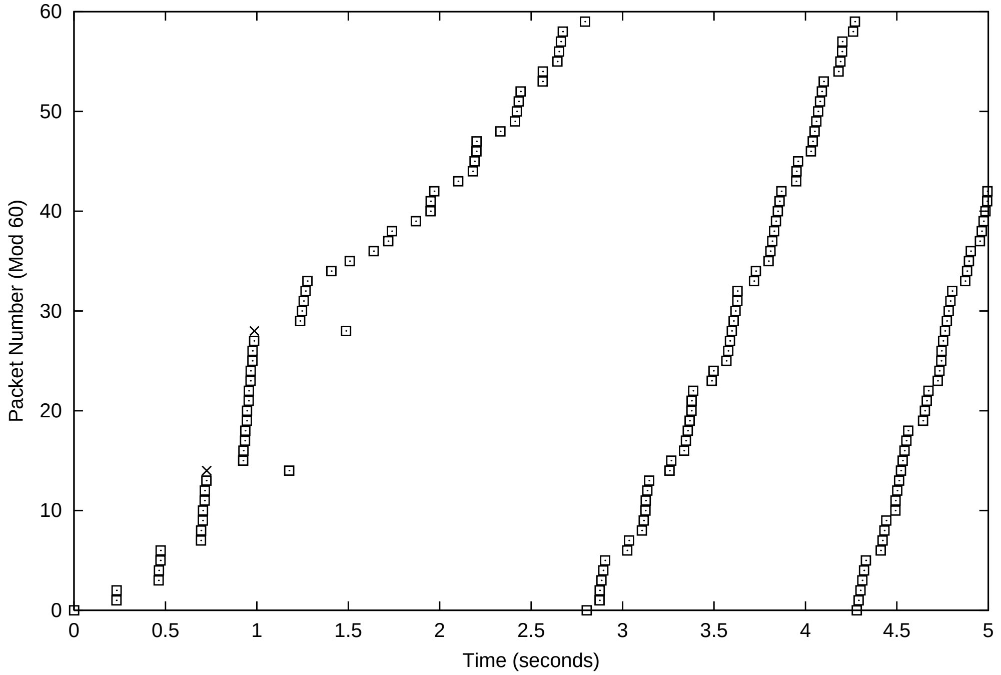


Figure 11: TCP Reno - two packet losses

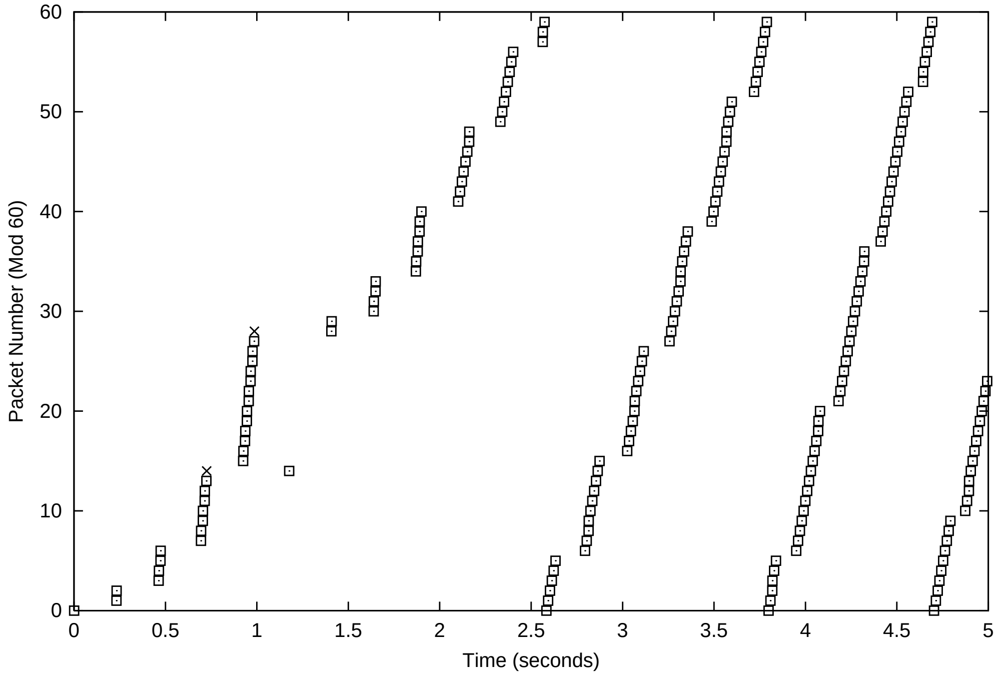


Figure 12: TCP Tahoe - two packet losses

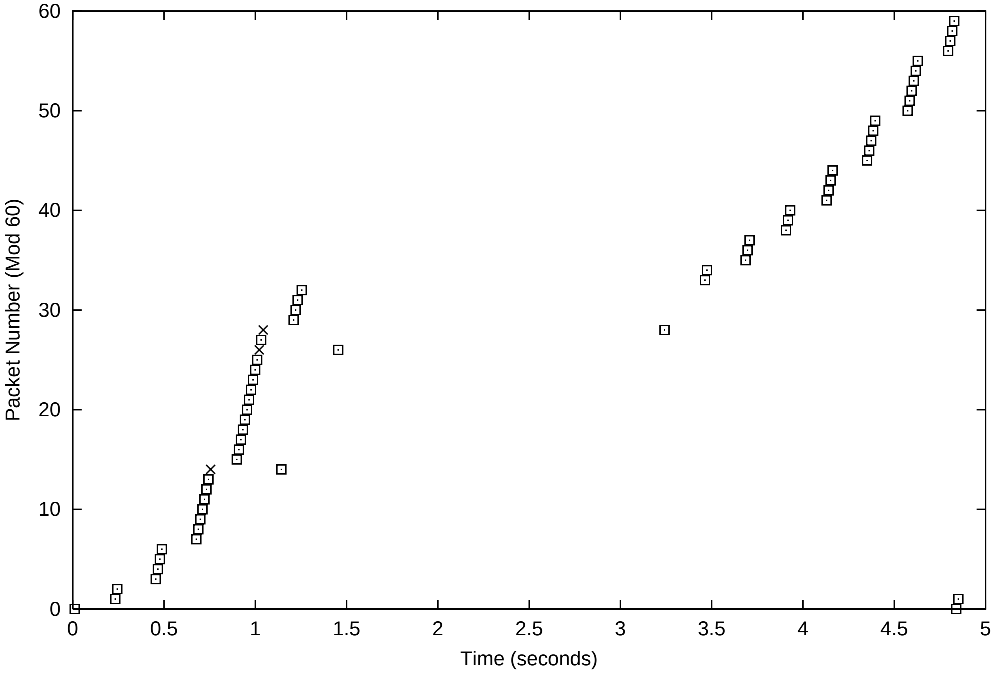


Figure 13: TCP Reno - three packet losses

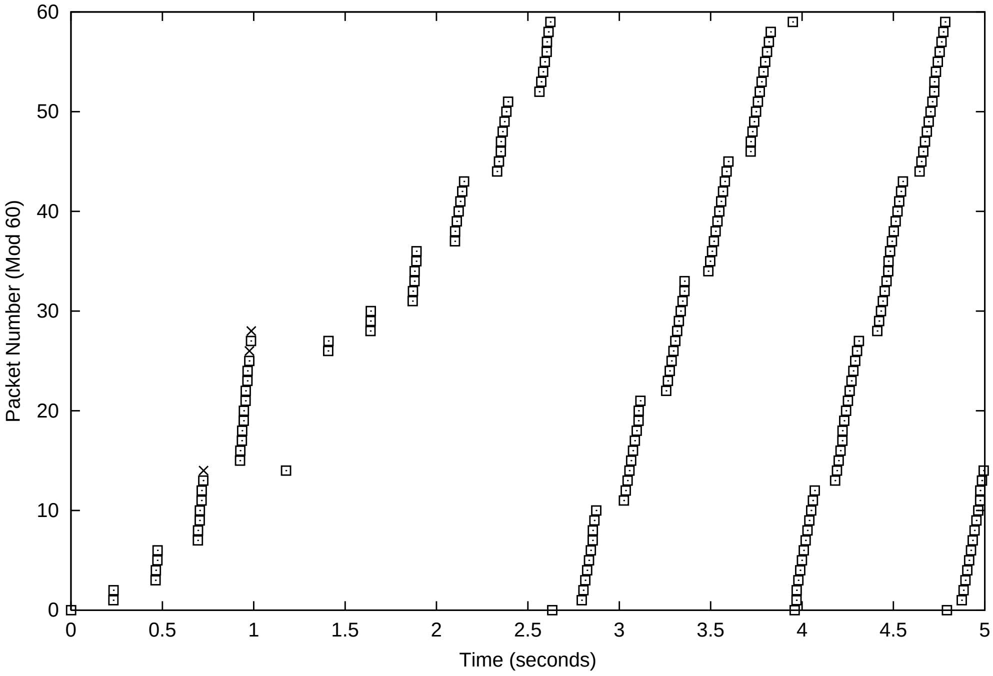


Figure 14: TCP Tahoe - three packet losses

Three Packet Losses

In addition to loss of packets 14 and 28, now also packet 26 is lost. Figure 13 shows the sequence of packets sent by the *TCP Reno* sender for the three packet loss situation. Figure 14 shows the sequence of packets for the *TCP Tahoe* protocol.

In case of *TCP Reno*, packets 0-13 are sent without problems. When packets 15-28 are sent, the TCP sender receives 12 *dupACKs* asking for packet 14. After retransmission of packet 14, *CWND* decreases to 10 and reaches 19 due to *dupACKs*. When the TCP sender receives an acknowledgement asking for packet 26, it exits *fast recovery*. At this point, *CWND* is set to 7. However, since packet 26 is lost, after receiving three *dupACKs* asking for packet 26, the TCP sender performs a second fast retransmission. *SSTHRESH* is reduced to 3 and *CWND* is set to 6. The three next *dupACKs* increase *CWND* to 9. After receiving an acknowledgement asking for packet 28, the sender exits *fast recovery* and *CWND* is reduced to 3.

Since *CWND* is 3 and three packets still unacknowledged, the sender is stalled and is unable to perform fast retransmission. This is reflected in Fig. 13 as the absence of squares in the period from 1.5 seconds to about 3 seconds. Thus, the sender waits for a retransmission timeout. When timeout occurs, packet 28 is retransmitted and the TCP sender sets *CWND* to 1, and *slow start* is then performed. The congestion window *CWND* now increases exponentially to a value of 3, after which congestion avoidance is entered since *SSTHRESH* is 3.

TCP Tahoe acts as in the two previous cases to cope with the loss of packet 14. When the sender receives an acknowledgement asking for packet 26, the *CWND* is increased to 2. In some situations, *TCP Tahoe* sender may forget the fact that some packets were previously sent. This problem is likely to appear when multiple packet losses occur in one window of packets. So, packets 26 and 27 are sent for the second time. Two acknowledgements for packet 28 are received. However the congestion window is increased to 3 because only one of the two acknowledgements represents the acknowledgement for a new data. The sender remains in the *slow start* phase and switches to *congestion avoidance* phase when packet 37 is sent.

5.2 Performance Analysis

Different kind of performance measures can be defined and analysed by simulation of the CPN TCP model. In this section we consider efficiency, which is defined as the number of packets delivered to the receiver divided by the maximum number of packets that could be delivered (when there is no loss of packets). It means that efficiency is 1 when there is no packet drop.

In the simulations we have assumed a segment size of 1000 bytes and a TCP maximum receiver window size of 32K bytes. All simulations runs correspond to 5 seconds of real time.

We observed the efficiency in both TCP versions for the situations of packet losses presented in the previous section. Table 1 summarises the results.

The results indicate that the *TCP Tahoe* keeps almost the same efficiency for the three situations. If we analyse the *TCP Tahoe* behaviour presented in the last section, we can observe that *TCP Tahoe* recovers from the packet losses in the three situations performing one *slow start*.

TCP Reno presents greater efficiency when it faces the one packet loss situation. This

	Efficiency	
Situation	TCP Reno	TCP Tahoe
1 packet loss	0.59	0.47
2 packet losses	0.38	0.47
3 packet losses	0.14	0.45

Table 1: Efficiency Results

occurs because Reno uses *fast recovery* after retransmission of the loss packet, allowing a smoothly recover from the packet loss. For the two packet losses situation, *TCP Reno* was forced to perform two successive *fast retransmit* and *fast recovery* procedures. Thus, the congestion window was reduced in half twice. Since the reductions were performed in two successive round-trip times, the performance of TCP connection was degraded. *TCP Reno* presents vary poor performance (efficiency of 14%) for the three packet losses situation. In this case, *TCP Reno* waited for a retransmit timeout to recover from a packet loss and a considerably performance reduction was observed.

The experiments showed that *TCP Reno* performs nicely in the case of a simple packet drop. For two or more packet losses the performance of *TCP Reno* is significantly degraded.

6 Conclusion

We have presented a timed hierarchical CPN model for behavioural and performance analysis of TCP protocols. The model considers the four most prevalent TCP algorithms for congestion control. We have used the term *generic* to emphasise that with minor changes it is possible to adapt the model for different TCP versions.

Simulation based performance analysis for the *TCP Reno* and *TCP Tahoe* protocols was also presented focusing on efficiency under different packet loss scenarios. The obtained results indicate that the model can be successfully applied to investigate TCP performance and can be used as a test-bed to evaluate different scenarios and variations of TCP.

The CPN TCP model has also been used as a building block in two other performance analysis projects. In one project it was used as a component in a model focusing on capacity planning and performance analysis of web servers. The TCP protocol is used in this context for transmission of documents between web clients and web servers. In another project, it was used as component in a model analysing the performance of a web based application for distributed teaching. The TCP protocol is used in this context to implement the remote procedure call mechanism by means of which clients and servers communicate.

Although the preliminary results indicate that the CPN TCP model can be applied to analyse performance of TCP protocols, further investigation on this issue still needed. The idea is to use the newly developed batch simulation facilities in the Design/CPN tool in order to make a larger number of simulations, considering various network scenarios and different switch drop policies.

Further refinements in the CPN TCP model to incorporate additional aspects are also important. For example, a more detailed model for *RTO* estimation can be developed. The timer granularity is an important factor in determining TCP performance.

References

[1] M. Allman, C. Hayes, and S. Ostermann. An Evaluation of TCP Slow Start Modifications. *Computer Communication Review*, 28(3):41–52, 1998.

[2] S. Christensen, J.B. Jørgensen, and L.M. Kristensen. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of TACAS'97*, volume 1217 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 1997.

[3] G. Ciardo, L. Cherkasova, V. Kotov, and T. Rokicki. Modeling a Scaleable High-Speed Interconnect with Stochastic Petri Nets. In *Proceeding of PNPM'95*, pages 83–93. IEEE Computer Society Press, 1995.

[4] H. Clausen and P. R. Jensen. Validation and Performance Analysis of Network Algorithms by Coloured Petri Nets. In *Proceeding of PNPM'93*, pages 280–289. IEEE Computer Society Press, 1993.

[5] H. Clausen and P. R. Jensen. Analysis of Usage Parameter Control Algorithms for ATM Networks. In S. Tohmé and A. Casada, editors, *Broadband Communications II (C-24)*, pages 297–310. Elsevier Science Publishers, 1994.

[6] D. E. Comer. *Internetworking with TCP/IP, Vol. 1, Principles, Protocols and Architecture.* Prentice Hall, 1994.

[7] DARPA. Transmission Control Protocol — Protocol Specification. RFC — 793, September 1981.

[8] K. Fall and S. Floyd. Simulation-Based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3):5–21, 1996.

[9] R. Goyal, R. Jain, S. Kalyanaraman, S. Fahmy, and B. Vandalore. Improving the Performance of TCP over the ATM-UBR Service. *Computer Communications*, 21(10):898–911, July 1998.

[10] J. C. Hoe. Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In *Proceedings of SIGCOMM'96*. ACM, August 1996.

[11] V. Jacobson. Congestion Avoidance and Control. In *Proceedings of SIGCOMM'88*, pages 314–329. ACM, August 1988.

[12] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[13] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1995.

[14] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Volume 3, Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[15] K. Jensen and G. Rozenberg, editors. *High-level Petri Nets.* Springer-Verlag, 1991.

[16] L. M. Kristensen, S. Christensen, and K. Jensen. The Practitioner's Guide to Coloured Petri Nets. *Software Tools for Technology Transfer*, 2(2):98–132, 1999.

[17] A. Kumar. Comparative Performance Analysis of Versions of TCP in a Local Network with a Lossy Link. *IEEE/ACM Transactions on Networking*, 6(4):485–498, August 1998.

[18] T. V. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwith-Delay Products and Random Loss. *IEEE/ACM Transactions on Networking*, 5(3), June 1997.

[19] B. Lindstrøm and L. M. Wells. Simulation Based Performance Analysis in Design/CPN. In K. Jensen, editor, *Proceedings of Workshop on Practical Use of Coloured Petri nets and Design/CPN*, pages 117–130, 1998.

[20] Design/CPN Online. http://www.daimi.au.dk/designCPN/.

[21] A. Ost and B. R. Haverkort. Analysis of Windowing Mechanisms with Infinite-State Stochastic Petri Nets. *Performance Evaluation Review*, 26(2):38–46, August 1998.

[22] W. R. Stevens. *TCP/IP Illustrated, Vol. 1, The Protocols.* Professional Computing Series. Addison-Wesley, 1994.

[23] W. R. Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms. RFC — 2001, January 1997.

Automatic Code Generation from Coloured Petri Nets for an Access Control System

Kjeld H. Mortensen
University of Aarhus, Department of Computer Science,
Ny Munkegade, Bldg. 540, DK-8000 Aarhus C, Denmark
k.h.mortensen@daimi.au.dk

Abstract

We describe in this paper a general method for automatic code generation from Coloured Petri Nets (CP-nets or CPN). The method is supported by the Design/CPN tool which has been extended during the past few years, such that it also can be used to generate code automatically from a CPN model. We do not describe the algorithms for code generation but rather the context such a tool is used in.

The rough outline of the method is as follows. One models the system of interest with CP-nets and Design/CPN. The modelled system behaviour is debugged and analysed, and when one has significant confidence in the model then the automatic code generation tool is applied, giving the final executable implementation as a result. Thus the behaviour of the model and executable are identical, and the traditional implementation phase has been eliminated.

In this paper we demonstrate that the method is usable in practice for an industrial example, namely an access control system developed by the Danish security company Dalcotech A/S.

This CPN model is a first version of the next generation of access control systems to be developed by Dalcotech. We describe the model and how they apply the automatic code generation method in order to obtain a system implementation quickly and safely. In this way Dalcotech now has the capability to reduce the time spent in the implementation phase dramatically. Another benefit is that they also dramatically reduce the amount of time spent on debugging the implementation.

1 Introduction

Coloured Petri Nets (CP-nets or CPN) [6, 9] is a language with an unambiguous formal definition and can therefore be subject to compilation and execution in a computing environment. The Design/CPN tool [30] has so far supported this in the form of a simulation engine, which is the traditional approach to execution of CPN models.

In an earlier project with Dalcotech [1] we made a proof-of-concept experiment with automatic code generation for a complex alarm system [20]. Standard ML code was extracted from a CPN model and burned into two PROMs that were mounted in a prototype of the final hardware. (See [15, 19] for more information on the functional computer language ML.) The experiment showed that automatic code generation from CPN models can be used in practice, but the experiment also showed a need to produce more efficient code and to develop additional tool support, e.g., for debugging, testing, and interfacing libraries.

However, automatic code generation was not the focus of that project. In fact the implementation of the system was done by hand in C++ based on the CPN model that was built. This was very time-consuming. Therefore it was decided to follow up on the challenges in a new project dedicated to automatic code generation. The project is called AC/DC (Automatic Code Generation from Design/CPN) [24] which is the main topic of this paper. The AC/DC project was supported by the Danish National Centre for IT-Research (CIT) [26].

The AC/DC project was initiated in April 1997 and ended successfully in April 1999. The project was supported by a consultancy group of 4 people (2 from the University of Aarhus [27], 1 from Dalcotech [28], and 1 from DELTA [29]) and a work group of 2 people from Dalcotech and 2 people from University of

Aarhus. About 2 man-years was spent on the project. The total budget was 1 million Danish kroner of which CIT contributed with half of the funding.

The purpose of the AC/DC project was to develop techniques and tools for automatic generation of code from CPN models. In this way it is possible to obtain an automatic implementation of systems that are designed by means of CP-nets and the Design/CPN tool. This eliminates a time-consuming and error-prone manual implementation phase. It also implies that the final system is behaviourally equivalent to the system design that was validated by means of simulation. Testing of the system can be accomplished already in the modelling phase. We elaborate on the technique and tools for automatic code generation in Sect. 2.

In the AC/DC project we applied the techniques and tools in practice. This was accomplished by means of a CPN model of a new embedded system for controlling buildings (access control, alarm system, energy control, energy measurement, etc.). The access control system was designed and specified by a CPN model, and it was implemented by means of automatic code generation directly from the model alone. We elaborate on the access control model in Sect. 3.

In Sect. 4 we compare our work with others, and finally we conclude the paper with a summary of main results of the AC/DC project and a report on ongoing and future research.

2 Automatic Code Generation Techniques and Tools

The method for automatic code generation constitutes a number of techniques and tools which we present in this section. Firstly the general approach is presented and then we explain how specialised libraries can be added in order to support the environment used by Dalcotech. The section after this (Sect. 3) describes the application of the method on an industrial case.

2.1 General Approach

The method for automatic code generation is summarised in Fig. 1. The figure indicates a number of stages and intermediate products. The central idea is that the system in mind is modelled by means of CP-nets in the Design/CPN tool. The Design/CPN tool supports several analysis techniques, such as state spaces and simulation, in order to debug the model. Once sufficiently confident that the model is accurate and complete we are ready for the code generation stage. The code, in the form of ML source, is extracted directly from the simulator because the simulator already implements a super-set of the model. (Details on the simulator can be found in a thesis [4].) Graphics simulation feedback is removed because it has no influence on the semantics of the implementation and is not needed in the final system. We only extract the kernel of the simulation engine and the implementation of the CPN elements of the specific model (cf. Fig. 2).

The ML source can now be compiled by whichever ML compiler is suitable for the target hardware platform in mind. In the figure there are two more stages which are specific for the AC/DC project, and we therefore postpone these technical stages to Sect. 2.2 below.

There are two major advantages of taking the implementation directly from the simulator. Firstly we get an implementation which corresponds exactly to what is included in the model. This immediately gives a number of benefits:

- Always consistent documentation of the system. As the model is considered as documentation it is only the model that is ever modified. Never the implementation.
- The implementation is of standard quality since the ML compiler always generates the same style of code.
- Analysis results found in the model can also be applied on the running system, assuming that the environment has been emulated accurately in the model.
- Errors are discovered early in the development process. The turn-around time is short should we find an error in the running system.

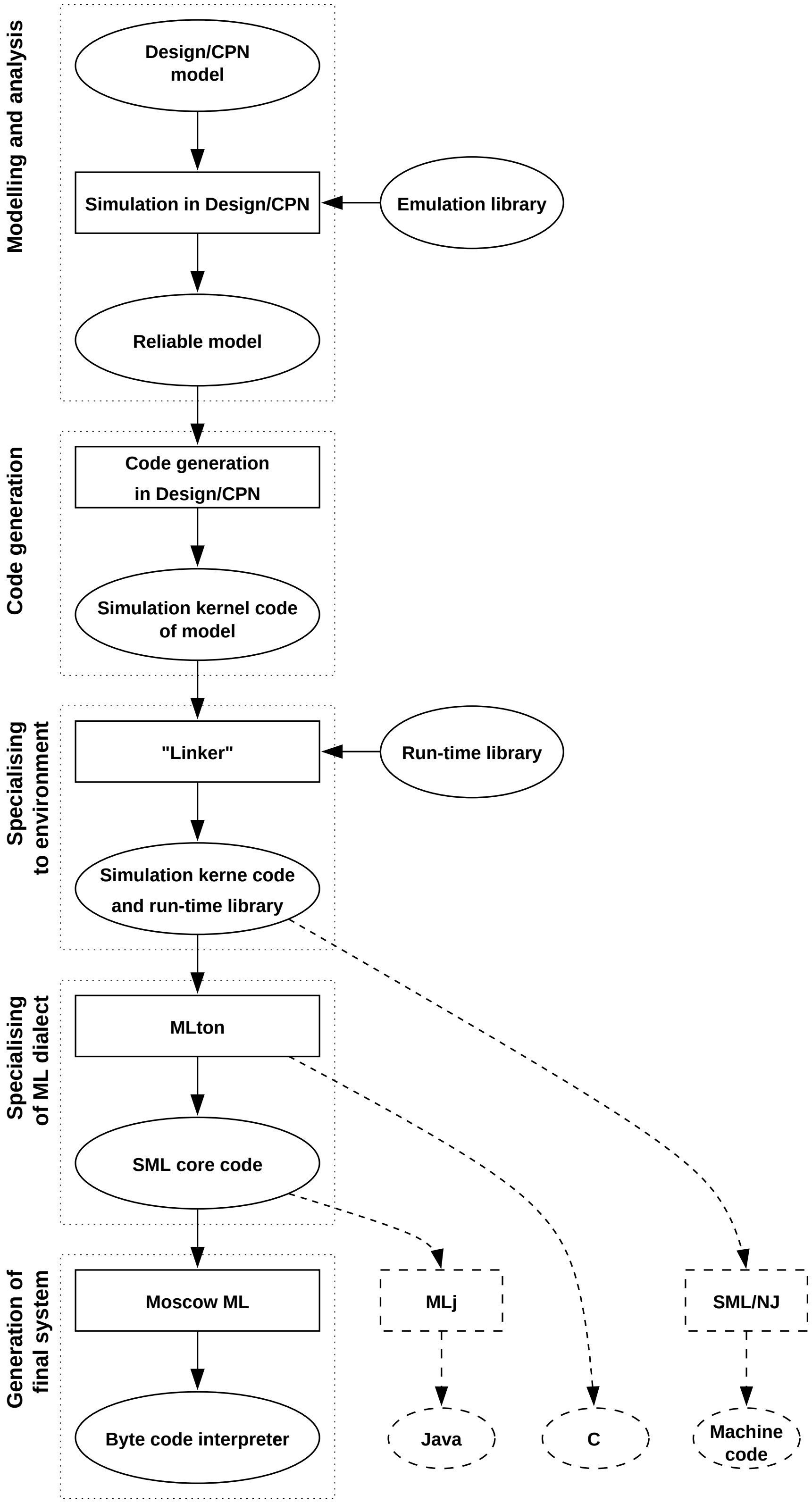


Figure 1: Overview of method as developed in the AC/DC project.

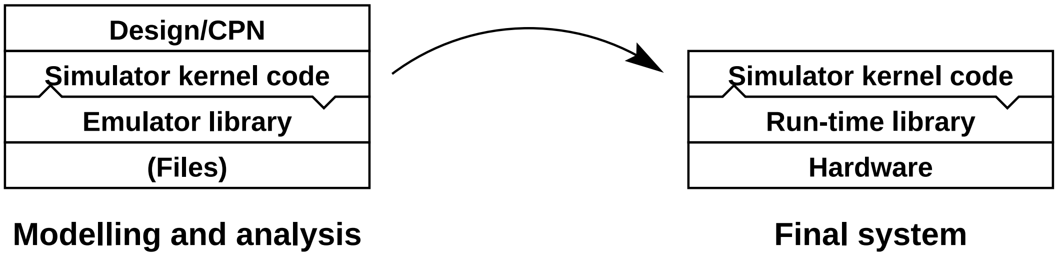


Figure 2: Same code used in simulator and final system implementation.

Secondly since the code generation method is automatised, we eliminate a time consuming and error-prone implementation phase.

- Errors do not originate from implementation since it is made automatically from the ML compiler.
- We save time because the implementation phase has been reduced to “pushing a button”.

Figure 1 also suggests the possibility for using a number of different ML compilers to generate the final system. Each compiler has each own advantages and limitations which we elaborate on in the following. We have chosen to focus on four compilers which generate very different kinds of code, namely SML/NJ, Moscow ML, MLton, and MLj. There are other ML compilers but their features are covered by these four. SML/NJ [14] generates native machine code and produces very fast executions. However the size of the code generated is very large. Moscow ML [21] generates byte-code which needs an interpreter. The execution is typically 50 times slower than SML/NJ but the size of the generated code is typically 75 times smaller than SML/NJ. MLton [23] generates C source code for which there exists numerous optimised compilers. Execution time is typically 3 times slower than SML/NJ but the size is about half of what SML/NJ generates. MLj [5] generates Java which has the advantage of being portable and supported by many platforms. We do not have experience with this compiler, but we expect that it has slower execution time than SML/NJ in the case of a Java interpreter. There are therefore a comfortable range of compilers for ML, and it is up to the user of the code generation method to judge the trade-offs in a given application.

Since we have no control over the implementation we may encounter some problems which depend on the target platform chosen. We lose control over performance of the automatically generated code. The simulation engine may not handle all kinds of models efficiently and could therefore be useless for some real-time systems. We also lose control over memory management which may be inappropriate for some kinds of embedded systems. We see later that these problems are not significant in the AC/DC project.

2.2 Code Generation for Embedded Systems at Dalcotech

Above we have described the general techniques and tools for automatic code generation. In the case of CPN models made by Dalcotech we need to take account of other factors. There are a number of requirements of embedded systems and limitations of specific environments described below. To solve these issues a special add-on library was made to handle the specifics of the environment used by Dalcotech. The post-processing supports add-on libraries (see Fig. 1).

Requirements and Limitations

Most of the security systems designed by Dalcotech, such as Seculon [20], contains an embedded controller which is the core of our concern. The controller has, as other controllers in embedded systems, limited memory and is required, to some extent, to exhibit real-time behaviour. However the limitations are not as severe as other embedded systems such as those found in pumps, wrist watches, and weights.

With the systems produced by Dalcotech we also need to take into consideration the fact that they are required to run in special standard hardware and software configurations. Otherwise the systems will not be approved by national and international quality organisations. The controlling unit must run under MS DOS and with a special network called LonWorks. The network uses the protocol called LonTalk. [1]

The reader may wonder why we consider an access control system to be a kind of embedded system, since the system is distributed over a network and there is user interaction. We generate only code for the controlling unit which is part of a larger LonWorks network configuration. The controlling unit senses hardware generated messages sent via the network and may send messages to the network. We therefore consider the controller unit to be a self-contained embedded system.

LonTalk Library

In order to support the LonTalk protocol in CPN models we have developed a library written in ML such that it is directly usable as inscriptions in Design/CPN. The LonTalk library is documented in a manual [18].

LonWorks is a networking architecture typically used for process control in building automation and manufacturing environments. The network speed is 40 kbit/s, and the protocol used, LonTalk, is OSI based. The protocol is reliable in the sense that if a send-message call terminates successfully it is guaranteed that the message was also received successfully and intact. (Authentication is also supported but not used in this project.) Devices, such as actuators and sensors, are attached to the network via a *Neuron chip*. A Neuron supports the LonTalk protocol, and input/output to the device in question. For instance, a device could be a lamp (actuator) or an infrared detector (sensor). A Neuron is responsible for maintaining and communicating values of network variables to other Neurons on the LonWorks network, and are therefore in some sense analogous to a neuron in a brain.

On the application level one does not know about the details of devices directly, but rather on an indirect level by means of *network variables*. A network variable is an abstraction of the state of devices on the network. For instance, the application programmer updates the value of a network variable which represents the state of a lamp or reads another network variable which represents the state of an infrared detector. A network variable may even be related with another network variable such that if one is updated then the other is also updated automatically. On system startup the network variables are configured to fit the setup of devices in question.

Apart from initialisation and configuration functions the interface of the library has two important functions, namely those to be used for input/output of messages to/from the LonWorks network. Below we have listed the signature of the functions:

```
val LONin  : LON_Millisec    -> LON_TIMED_EVENT
val LONout : NV_IDX * NV_VAL -> LON_RESULT
```

The function `LONin` attempts to read a message from the network within the specified time-out (`LON_Millisec`). If a message arrives before time-out then it is returned immediately to the caller, otherwise at time-out the caller is informed that no messages arrived (`NO_EVENT`). The `LONout` function sends a message to the network and returns a diagnostic value. The `LONin` function should be called regularly in order to prevent input buffer overflow.

While modelling with CP-nets one often wishes to make a simulation in order to analyse the system. In a model where LonTalk is being used it is usually not possible to make a simulation because the network is not present. In order to remedy this problem a special version of the LonTalk library can be used which is able to emulate messages sent and received on the network. All one has to provide is a file which specifies which messages are to be received by the model and at which points in time. In this way one can debug the model with a simple list of messages and then run the model on the input and investigate how the model responds either at the end of a simulation run or in a stepwise fashion. This emulation version of the library is also documented in [18].

Generic Post-processing Tool

When we build models which are based on LonWorks we need to use the LonTalk library somehow in the model. In the model we wish to specify that we send and receive messages on the LonWorks network.

[1] LonWorks and LonTalk are trademarks of Echelon Corporation [31].

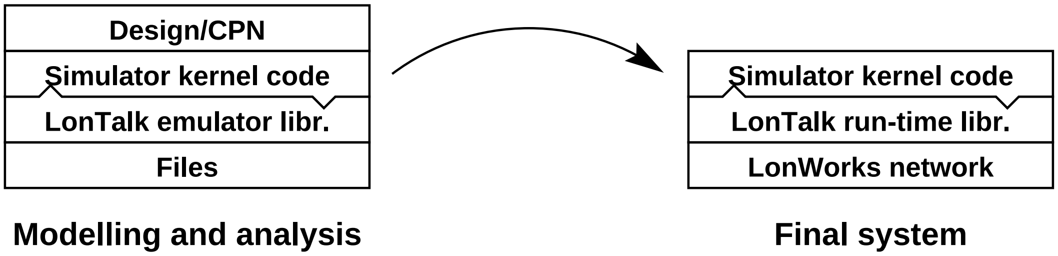


Figure 3: Dalcotech's use of libraries and automatically generated code.

While simulating with Design/CPN we cannot rely on the network being present. On the other hand the code generated from the model must run in a LonWorks environment. Figure 3 depicts the situation for Dalcotech and is therefore a specialisation of Fig 2.

To support this we have built a post-processing tool which takes the generated simulation code as input. The tool recognises platform dependent code, such as the LonTalk library, via special include statements specified by the user in the global declaration node: `use "_<library-name>.sml"`. It is the leading underscore which is recognised by the post-processing tool. The tool produces one self-contained source file, with the platform dependent code linked in, which can be compiled by whatever compiler is needed in relation with the target hardware in question. In the AC/DC project we run on the DOS platform and the compiler we use is Moscow ML [21]. This compiler generates compact byte-code for a DOS-based interpreter, and is therefore useful in the AC/DC project. The post-processing tool is documented in a manual [17]. In Fig. 4 the code generation method as used by Dalcotech is summarised. It is a specialisation of the general method as depicted in Fig. 1.

The disadvantage of Moscow ML is that it does not support the full module language of SML/NJ [14] which is what the code generator of the simulator produces. We therefore apply a, so called, de-functorriser [23] which unfolds the source code to the Standard ML core language [15]. The unfolded source can be compiled by Moscow ML.

The post-processing tool does not depend on the LonTalk library as used in the AC/DC project. The tool is in fact generic in the sense that it can be parameterised with whatever platform dependent libraries are needed. A library which is used with the tool must conform to the procedure as described in the manual [17]. In short one needs to make two versions of the library: A simple version which should consist of at least an interface, and a full version with complete functionality. The simple version is used in the model and the full version is linked with the simulator generated code by means of the post-processing tool. In the case of the LonTalk library the simple version is a simple emulation of the LonTalk protocol as described earlier in this section. The advantage is that we can use a platform dependent library both in the simulator and the final system, but the disadvantage is that it is the user's responsibility to ensure that the two versions of the library behave similarly.

3 Access Control Model

In this section we present the access control model which was made by Dalcotech in the AC/DC project. We demonstrate how the code generation method, described in the previous section, was successfully applied, and we show that the model architecture is flexible in case a customer requests new features.

The model was built mainly by two people from Dalcotech which took less than two man-months including specification, design/modelling, analysis, and code generation. People from the consultancy group was only involved in two model reviews. The reviews were very useful and effective means for improving the modelling strategy and architecture. The first review resulted in guidelines for continuing the modelling work while the second review was used for minor adjustments in order to finish the model.

The access control system works roughly as follows (see Fig. 5): A person outside the room may open the door by punching his personal code on the code entry unit (1). The system recognises the person,

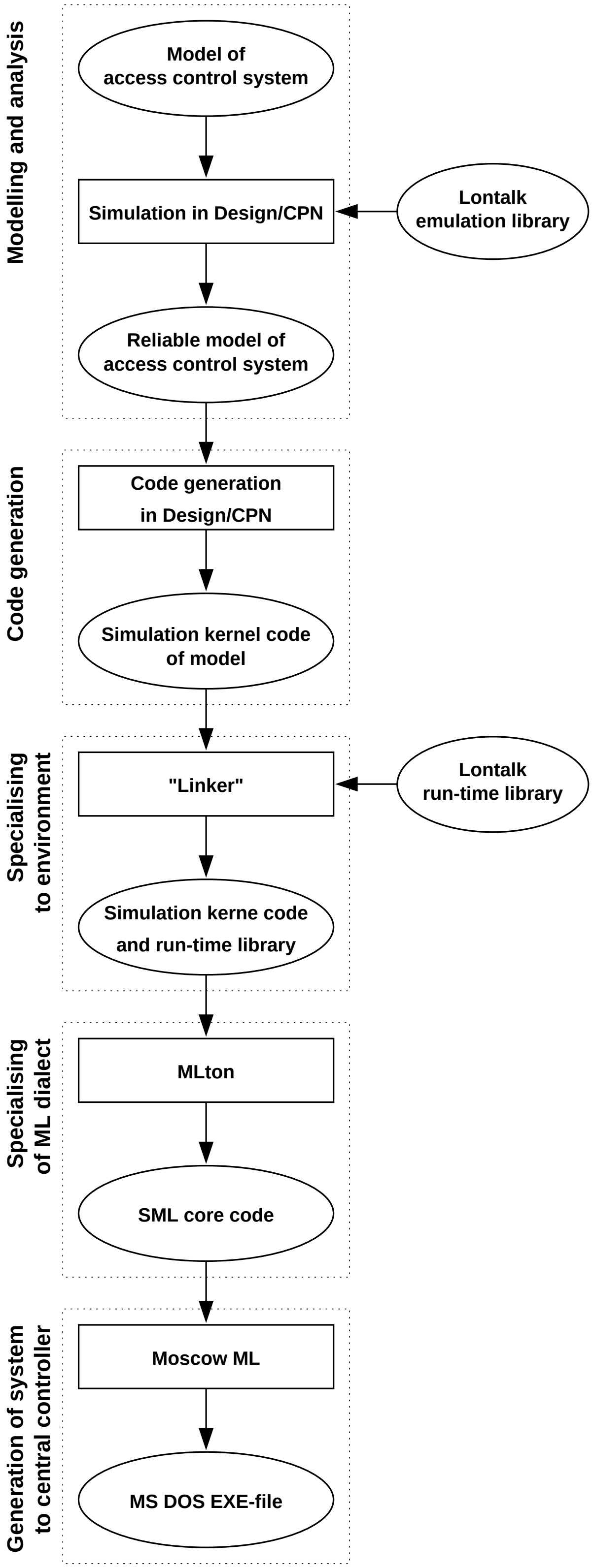

Modelling and analysis
Model of access control system
Simulation in Design/CPN
Lontalk emulation library
Reliable model of access control system
Code generation
Code generation in Design/CPN
Simulation kernel code of model
Specialising to environment
"Linker"
Lontalk run-time library
Simulation kerne code and run-time library
Specialising of ML dialect
MLton
SML core code
Generation of system to central controller
Moscow ML
MS DOS EXE-file

Figure 4: Overview of method as applied by Dalcotech.

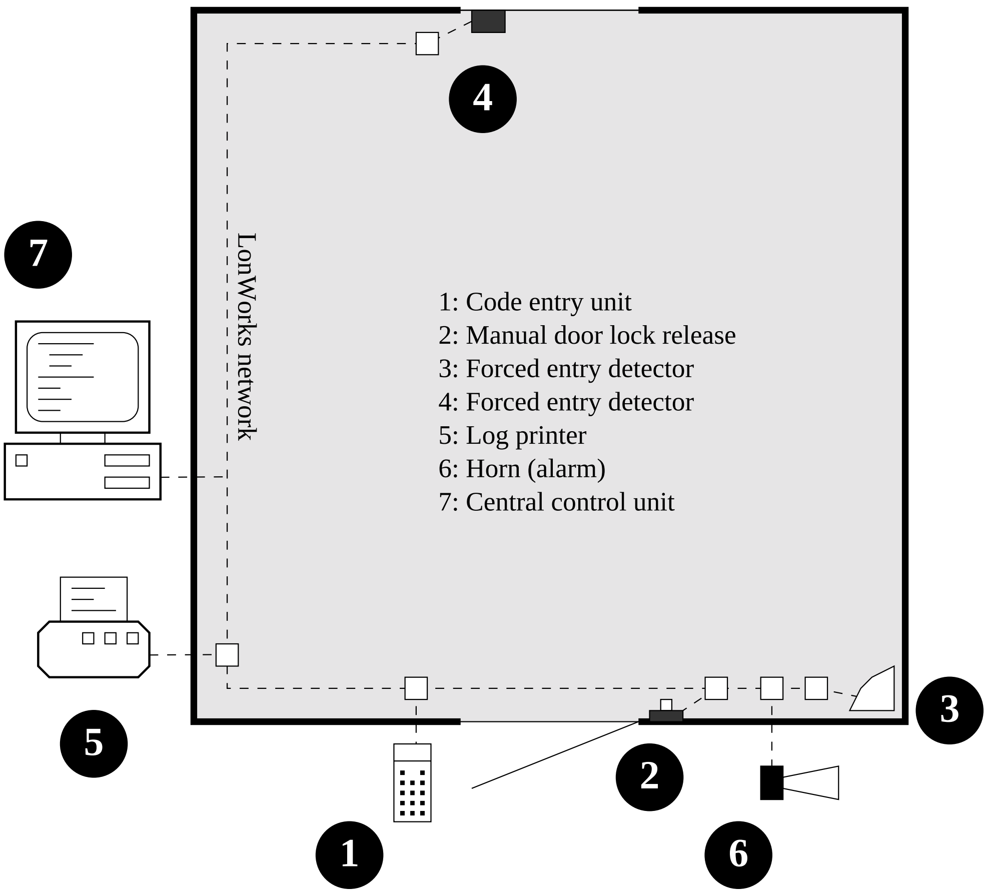


Figure 5: Access control scenario. White squares on the network represents the standard interface (Neurons) to various devices.

checks that the person is allowed in the area at this time, disables the alarm, and unlocks the door for a limited period of time. The system log is updated by printing status information on the log printer (5). More people may enter, each punching a personal code, and the system updates the log. In order to leave the room a person must push the open door button (2). The last person leaving the room must punch in the personal code in order to enable the alarm again. In alarm mode the horn (6) can be triggered in case one of the entry detectors are activated (3 and 4). There is other interaction possibilities and combinations of actions, but they are left out of this paper.

3.1 Model Structure

The structure of the CPN model of the access control system is depicted in Fig. 6. The model is of moderate size, and consist of 21 pages and 70 transitions. There are three important parts in the model, namely initialisation (`Initialize`), Neuron (`Neuron`), and event handler (`Main`). They are described in the following sections.

Initialize

The model is initialised in several steps in a sequential fashion, see also Fig. 6.

1. All network variables are initialised in order to related each network variable with a devices in the network. (See also Fig. 7.) It is worth noting here that this configuration covers three code entry units

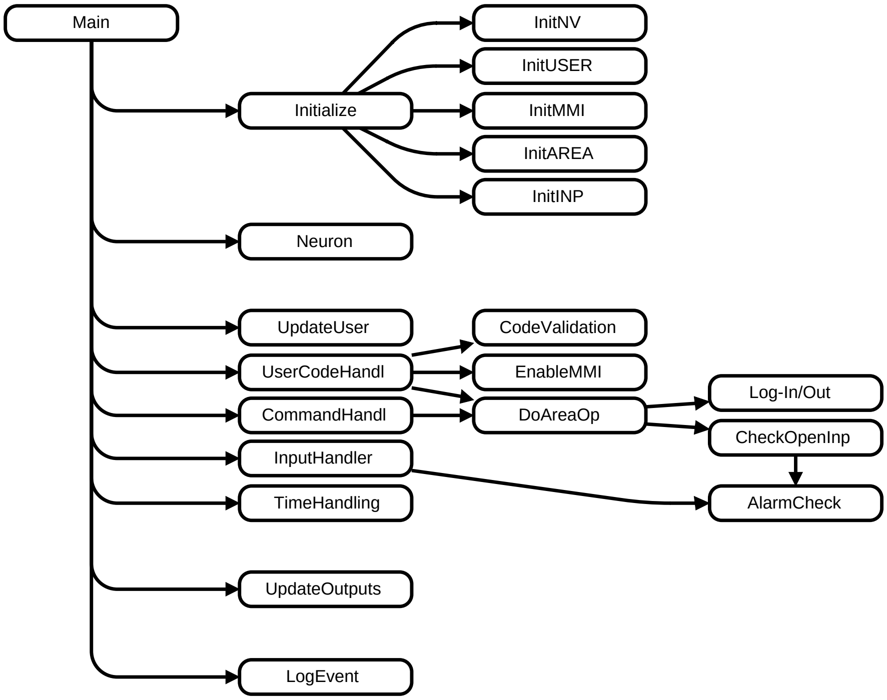


Figure 6: Hierarchy page of CPN model.

which corresponds to covering three rooms (or doors). Furthermore there are 10 user configurable network variables.

2. The user database is initialised with names related with entry codes and where people are allowed to be and when.

3. The database containing information about man-machine interfaces (MMI), such as the code entry unit, are initialised with logic for managing enabling and disabling of alarms in case the correct code is punched.

4. The expected initial states of inputs are initialised. For instance we assume that all input devices are off.

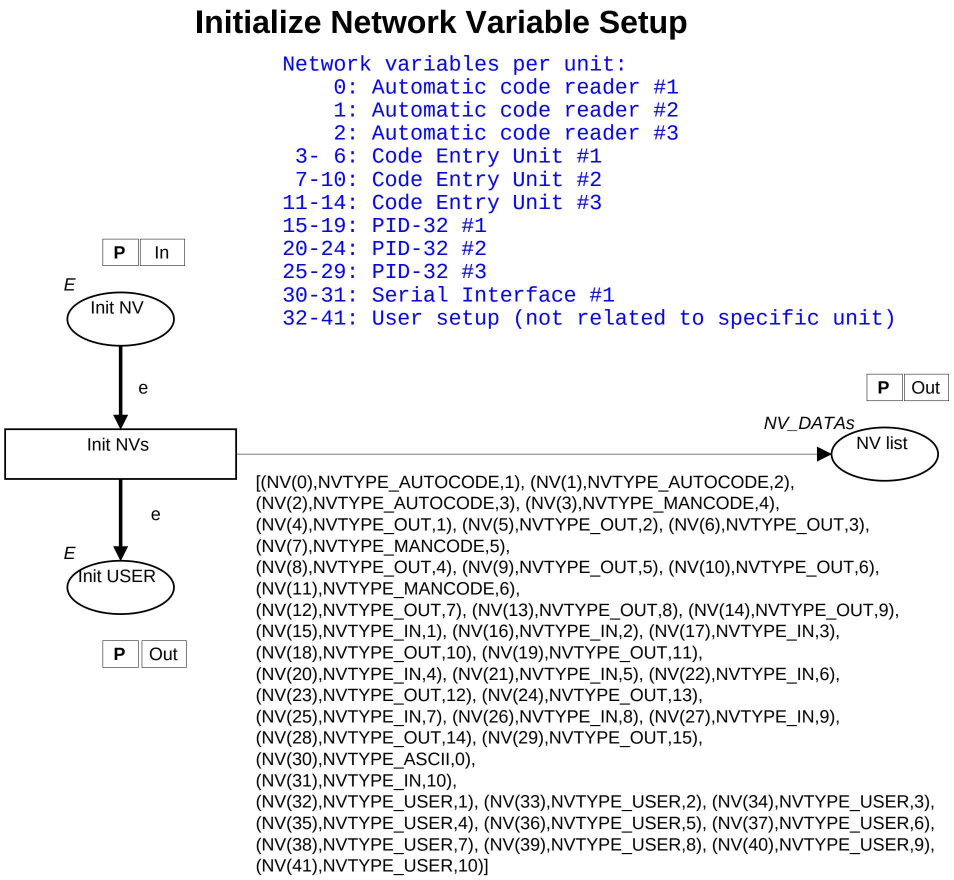
5. Finally we initialised the status of the areas. All areas a initially armed.

After this sequence the configuration is extracted from the databases in the model and sent to the network. If the configuration fails then error information is passed on to the main loop of the model.

Neuron

The Neuron is a standard component in LonWorks based systems. Care should be taken to model such a component as it is expected to be reused in other models of LonWorks systems. The model made by Dalcotech is depicted in Fig. 8. Relevant colour sets are shown below:

```
colorset NV_UPDATE = product
  NV_IDX *      (* LonWorks Network variable index *)
  NV_VAL;       (* LonWorks Network variable value *)
colorset LON_EVENT = union
  ...
  NO_EVENT +   (* No events from LonWorks *)
```



```
Network variables per unit:
    0: Automatic code reader #1
    1: Automatic code reader #2
    2: Automatic code reader #3
 3- 6: Code Entry Unit #1
 7-10: Code Entry Unit #2
11-14: Code Entry Unit #3
15-19: PID-32 #1
20-24: PID-32 #2
25-29: PID-32 #3
30-31: Serial Interface #1
32-41: User setup (not related to specific unit)
```

```
[(NV(0),NVTYPE_AUTOCODE,1), (NV(1),NVTYPE_AUTOCODE,2),
(NV(2),NVTYPE_AUTOCODE,3), (NV(3),NVTYPE_MANCODE,4),
(NV(4),NVTYPE_OUT,1), (NV(5),NVTYPE_OUT,2), (NV(6),NVTYPE_OUT,3),
(NV(7),NVTYPE_MANCODE,5),
(NV(8),NVTYPE_OUT,4), (NV(9),NVTYPE_OUT,5), (NV(10),NVTYPE_OUT,6),
(NV(11),NVTYPE_MANCODE,6),
(NV(12),NVTYPE_OUT,7), (NV(13),NVTYPE_OUT,8), (NV(14),NVTYPE_OUT,9),
(NV(15),NVTYPE_IN,1), (NV(16),NVTYPE_IN,2), (NV(17),NVTYPE_IN,3),
(NV(18),NVTYPE_OUT,10), (NV(19),NVTYPE_OUT,11),
(NV(20),NVTYPE_IN,4), (NV(21),NVTYPE_IN,5), (NV(22),NVTYPE_IN,6),
(NV(23),NVTYPE_OUT,12), (NV(24),NVTYPE_OUT,13),
(NV(25),NVTYPE_IN,7), (NV(26),NVTYPE_IN,8), (NV(27),NVTYPE_IN,9),
(NV(28),NVTYPE_OUT,14), (NV(29),NVTYPE_OUT,15),
(NV(30),NVTYPE_ASCII,0),
(NV(31),NVTYPE_IN,10),
(NV(32),NVTYPE_USER,1), (NV(33),NVTYPE_USER,2), (NV(34),NVTYPE_USER,3),
(NV(35),NVTYPE_USER,4), (NV(36),NVTYPE_USER,5), (NV(37),NVTYPE_USER,6),
(NV(38),NVTYPE_USER,7), (NV(39),NVTYPE_USER,8), (NV(40),NVTYPE_USER,9),
(NV(41),NVTYPE_USER,10)]
```

Figure 7: CPN page modelling the initialisation of network variables.

```
  UPDATE: NV_UPDATE; (* Update of network variable *)
colorset LON_TIMED_EVENT = product
  LON_EVENT * (* LonWorks event *)
  INT;        (* Ticks passed since last time reporting ticks *)
```

As with the physical Neuron, the model is divided into two parts: A device driver part (left) and a part providing an interface to the application (right). The device driver part just makes basic input/output calls to the LonTalk library which is the basic interface to the network. The LonTalk protocol is rather complicated and is therefore not modelled as a CP-net. The application part converts network messages to higher level messages (events) appropriate for the application (upper half) and also converts higher level messages to network messages (lower half). In case no message (NO_EVENT) is received from the network then timer information is propagated. This information can be used elsewhere in the access control model for managing user access times and alarm timeouts. Notice that the model does not use timed CP-nets but instead stores information on the (real) time passed since last event was read (LON_TIMED_EVENT).

The Neuron is modelled such that either input or output is treated exclusively, never both at the same time. This is to avoid multi-threaded behaviour in the access control model. Why we have made this choice will become more evident below where we describe the model part for event handling.

We could also imagine that the model would benefit from using timed CP-nets. Timeouts and delays could then be expressed directly as modelling primitives and then potentially make the model simpler. The timing logic of the transition scheduler of the simulation engine would then need to be related with the system clock. Otherwise the timeouts and delays would not be real-time. Unfortunately this simulation architecture is not very well studied, and there are problems to be solved. For instance, what if the model

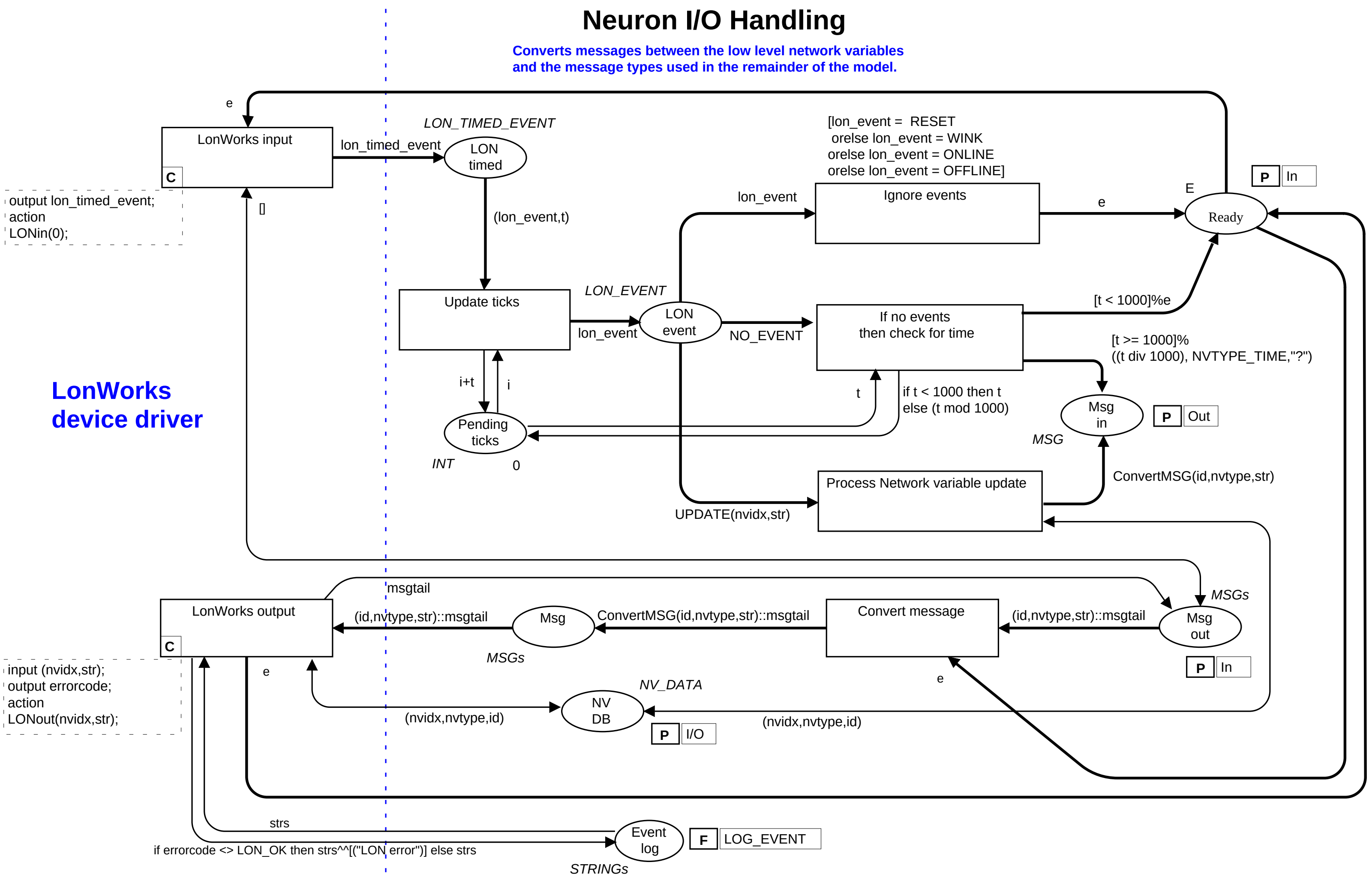


Figure 8: CPN page modelling the Neuron.

gets significantly behind the real-time clock? This could happen if a calculation in a transition takes up too much CPU time. We leave this as a research topic for the future.

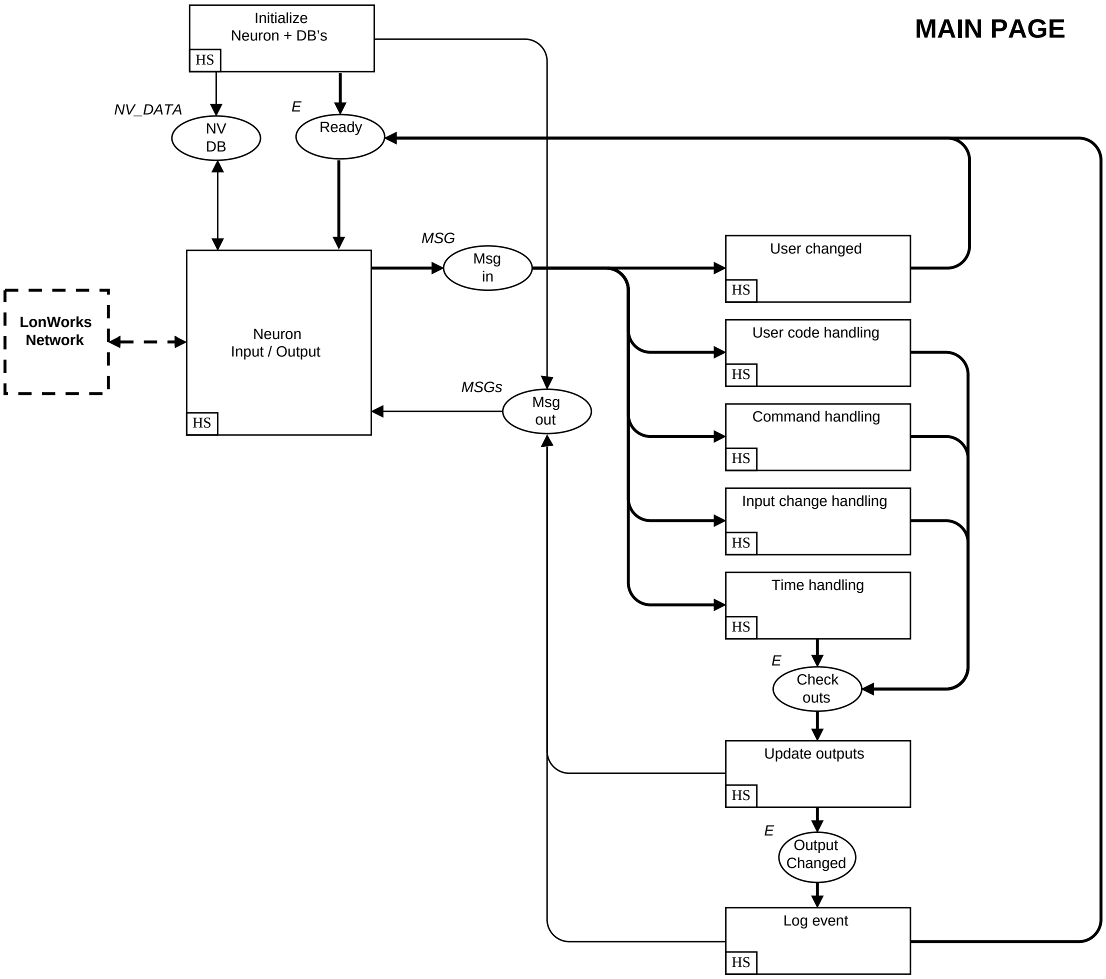


Figure 9: CPN page modelling the main event handling cycle.

Event Handling

Event handling is a central cycle in many systems, in particular reactive systems where user input is involved. The access control model also has an event cycle which is depicted in Fig. 9. The relevant colour sets are listed below:

```
colorset MSG = product
  INT *     (* Id *)
  NV_TYPE * (* Network variable type *)
  NV_VAL;   (* Data *)
colorset MSGs = list MSG;
```

Once the system is initialised (top left substitution transition) the Neuron is activated such that new messages from the network can be read. The Neuron, `Neuron Input/Output`, converts a network message to an event which is put on the place `Msg in`. This is subsequently dispatched to the appropriate event handler which can be one of `User changed`, `User code handling`, `Command handling`, `Input change handling`, or `Time handling`. Once an event is fully treated then the event cycle checks if we need to generate any messages to the network (`Update outputs`), and then generates messages for the log printer on the network. For instance, if a user has punched in an access code, then the system registers this and then sends an unlock door message, a turn on green light indicator message, and a log entry message to the printer. Finally the Neuron is activated again for processing all assembled messages on `Msg out`, and then the Neuron is ready to process the next input message from the network.

Note that the description just provided is purely sequential. As noted before where we described the Neuron page, concurrency is something we have chosen to avoid at this level. Handling of multiple events concurrently would make the model much more complicated because there are several data bases accessed throughout the model, and ensuring mutual data base consistency would be difficult. Also, we know that the target hardware platform only has a single CPU, thus there is not an obvious performance benefit of having concurrent activities in the model. However, the idea was to start with a simple solution and then make it more advanced with respect to concurrency as more experience was obtained.

3.2 Code Generation Method Applied

In the following we describe in detail how the code generation tools are applied for the access control model. We assume that we have the access control model and that we are sufficiently confident that it works. In the global declaration node we have made a reference to the platform dependent LonTalk library by including the statement `use "_Lontalk.sml"`. The steps described below corresponds to the high-level description in Sect. 2.

We load the access control model into the Design/CPN tool and invoke the ML simulation engine (called the switch to simulator). As part of the syntax check and simulator setup, the ML source code is extracted and then saved to a file, say `/tmp/ml.out`. Then we call the post-processor on a Linux machine to make platform dependent source:

```
linux% cpnsim2mosml.pl /tmp/ml.out /tmp/accsctrl.sml
```

The post-processor recognises the special use-statement and includes the full version of the lontalk library. Finally we move `accsctrl.sml` to the DOS platform and create the final executable:

```
dos% mosmlc accsctrl.sml
```

This command creates a DOS executable called `mosmlout.exe` which now can be started, assuming the LonWorks network is attached to the machine. The whole procedure takes 10–20 minutes on a 200MHz Pentium PC with 64Mb RAM for the access control model.

Once the automatically generated implementation is executing on the DOS platform we can investigate the behaviour and responsiveness of the system. We have a simplified demonstration hardware scenario similar to Fig. 5. Instead of having an alarm horn there is a lamp, and instead of specific sensors such as forced entry detectors there is a switch which can be controlled manually. This is a cheap but still effective scenario for testing the running system.

We are now able to investigate the running system. Although we are able to run the same system in the simulator of Design/CPN, we may discover anomalies which were not found while simulating. The state space tool in Design/CPN could have helped us to find more problems as a means to make an exhaustive simulation, but this tool was never applied in the AC/DC project, mainly because the model has an infinite state space (messages on the network is unbounded). The model can however be modified such that we obtain a finite state space, but this was not pursued further. Another interesting property of the running system is how it performs. How quick can it react to user inputs and is it sufficiently fast to generated alarms? On the 33MHz DOS machine with 32Mb of RAM the system was indeed running fast enough. The system reaction time to user input is less that half a second, which is within the margins of this kind of access control systems. We could in principle have carried out more systematic performance analysis in Design/CPN as it is supported in the tool, but this was out of scope of the project.

3.3 Benefits of Model Architecture

The model we have describe above in Sect. 3.1 has a number of useful engineering properties. It has reusable components, is extensible, and adaptable. It is of great interest for Dalcotech to learn from modelling experiences and use this model in future projects. Usually a customer will need an access control system with special features and it is therefore important to avoid building the system from scratch every time. Reusing existing modules is faster and safer. In the following we show that the model has such properties.

Extensible

Consider the CPN module in Fig. 9 which depicts the event handler of the access control model.

The structure of the event handler is divided into cases for each kind of possible event. A new customer would typically request support for new devices in the system which consequently would induce new kinds of events from the network. Extending the case structure is relatively easy as one just needs to add a new substitution transition to handle the new kind of event. An event may require updating some of the access control data bases in the system, but these are easily accessible as they are located on fusion places.

It is also relatively safe to add support for a new event with respect to data base integrity, because only one event at a time can be processed by the system.

Reusable

Consider the CPN module in Fig. 8 which depicts the Neuron of the access control model. A Neuron is a standard component to interface between devices and the LonWorks network. We therefore only need to model a Neuron once.

If we model the Neuron as one page, as in our case, we can easily use the save/load sub-page feature in Design/CPN in order to imitate a simple module facility. Should we need the Neuron module in another model, all we have to do is to make a substitution transition and load the Neuron module via load sub-page and then assign the ports to the sockets around the substitution transition.

As the Neuron is a standard component it will be used frequently and therefore also tested in many different contexts. We therefore expect such a module to become increasingly robust over time. As the module matures it will become safer to use in new contexts.

Another example of re-usability in the access control model is the underlying data bases which are used to manage the access control logic. This manager is not described further, but we would like to note that the functions operating on the data bases are written in ML and therefore easy to include as libraries in the global declaration node. ML libraries are in general frequently used as reusable components in Design/CPN models.

Adaptable

Consider the initialisation phase which is described in Sect 3.1. One step in the initialisation consists of sending configuration information to the LonWorks network (Fig. 7) such that network variables are configured for the attached devices. The configuration is in fact adaptable by the end-user within a limited range. For instance, a specific code generated system targeted for small buildings will typically be pre-configured to be able to handle about three rooms with 20 sensors and actuators as is the case in Fig. 7. In this way Dalcotech does not need to code generate a new system for every new customer, but can instead have a limited number of products each covering a range of building sizes.

4 Related Work

There are more than 50 tools for Petri nets which are actively being developed [32]. Many of these tools have some sort of simulation support for Petri nets. Typically a given Petri net model is translated into a different language (C/C++, Java, SML, etc.) which subsequently is interpreted or compiled by already established and well-known tools. In this section we compare our work with other tools and approaches, however limited to tools supporting high-level Petri nets or other high-level languages such as object-oriented languages.

LOOPN++

LOOPN++ supports a kind of object-oriented high-level Petri nets [10]. It contains a translator which can generate C++ source code which is then subsequently compiled to native machine code and executed. The tool has also been extended to generate Java source code [11]. LOOPN++ is only a compiler and does not contain a simulator, and is therefore very different from Design/CPN in the respect that LOOPN++ does not have a direct behaviour relation between model and generated code (cf. Fig. 2).

CPN-AMI

CPN-AMI is a CASE environment based on some kind of high-level Petri nets [13]. Among other components CPN-AMI contains a simulator (CPN/DESIR) and a code generator (H-Tagada) which generates ADA code. The code executed by the simulator and the generated ADA code are different. Thus there is not as strong relationship between simulation and generated code as with the code generation method of Design/CPN (cf. discussion of Fig. 2). However, the H-Tagada code generator is able to separate a Petri net model into processes (G-objects), thus making potential for true concurrent executing in multi-processor environments. This is not supported in the Design/CPN code generator.

Another tool called Artifex [25] is similar to CPN-AMI in that it has both a simulator and code generator, however Artifex generates C/C++ code. There are a number of other tools which has the same code generation architecture as these.

PEP

PEP [3] has many different kinds of translators which can translate back and forth many different kinds of languages such as P/T nets (Petri box), high-level nets (M-net), and textual language ($B(PN)^2$). A modification in one language representation can automatically be updated in another, in a consistent fashion. This is analogous to what sometimes can be found in UML notation based environments. Updating a textual representation of a class will consistently update a diagrammatic representation of a class and vice versa. An example of such system is the Mjølner System [8].

The code generation method of Design/CPN does not support what can be called reverse engineering, i.e., translation from implementation to a CPN model. However, this is not the intention either. Once the code has been generated it should neither be investigated nor modified manually. One would then ask if we could support reverse engineering, but this would be difficult in the case of SML and Petri nets because they are radically different languages. This is not the case for the UML based environments where the diagrams and textual languages are very similar in structure. Also in the case of PEP the various languages supported are mutually similar, and could thus be worth investigating closer in case we wish to support translation in both directions in Design/CPN.

ML compared with C/C++

We are often asked whether it would be easier to generate C/C++ source code instead of ML. There are two main reasons that C/C++ would be harder to generate in the current architecture. Firstly the inscription language of CPN models is ML. Most of the inscriptions are directly implementable in ML. Generation to C/C++ would require a translation, in particular for the cases of pattern matching. Pattern matching is a powerful construct and is frequently used in CPN models. ML directly supports pattern matching but C/C++ does not.

Secondly the ML compiler is interactive which is in contrast to most compilers for C/C++ which are batch compilers. It is easy the change a CPN model given an interactive compiler because we just send new source code to directly overwrite existing representations of the model. With C/C++ batch compiling the turn-around time is longer because a new application needs to be generated.

In general the CPN language is more in the nature of functional languages where side effects are not allowed. C/C++ is imperative and side effects are more naturally used.

5 Conclusion

Dalcotech is a small company and it is therefore limited what the company can invest of resources on a project like AC/DC. However, given that Dalcotech was already familiar with CP-nets and previous positive experience with CPN development projects they were quite certain that the AC/DC project would be feasible [7].

Although Dalcotech has used CP-nets in practice for a number of years, it is in AC/DC the first time they use CP-nets directly for automatically generating the implementation. In this way Dalcotech now has the capability to dramatically reduce the time spent in the implementation phase. Additionally they also dramatically reduce the amount of time spent on debugging because errors in the implementation in principle originate from the model only.

In the AC/DC project Dalcotech has successfully conducted a code generation case study where a CPN model has been built for the next generation of access control systems the company has planned to produce. By means of the code generation method and their model, Dalcotech has obtained an automatic implementation of the embedded controlling unit of the access control system. This non-trivial case study proves that the method can be applied in practice by engineers who are not completely familiar with the advanced technology behind automatic code generation. Additionally, Dalcotech has during the modelling process gained better insight into access control systems in general.

The AC/DC project has for our part provided an excellent opportunity to develop and apply the techniques and tools in the context of an interesting industrial case study. Furthermore, we have been able to extend the use of CP-nets into the new area of automatic code generation for embedded systems and developed a method to support this [16]. It has been a useful challenge to match the needs of Dalcotech, and we have thus become more experienced with the techniques and tools and more confident that our method works in practice [7].

A significant part of Dalcotech's revenue is from development contracts and it is therefore expected that automatic code generation from CPN models will have a significant role in the future [7].

Another ongoing activity is the effort to conduct technology transfer of the method to other companies. A full day seminar was conducted for the Danish industry. The seminar included introduction to CP-nets, the method for automatic code generation, and its application. Such seminars are held under the auspices of DELTA. In this way we help to make advanced computer science technology more easily available for a wider audience. At the seminar the attendants were motivated and asked a lot of relevant questions. One of the main concerns, however, was how they could convince their companies to use the CPN methodology. We proposed a number of arguments that could be used: The CPN language is simple to learn and yet powerful to use. It is a visual approach where diagrams can be explained for non-experts. The CPN methodology can be introduced in a company for enriching existing methods, such as UML — not as a replacement. Typically CP-nets can be used in situations where the existing methods are limited, e.g., in analysis and verification phases. We have also seen in a previous CPN project with Dalcotech that the CPN methodology was used as part of a valid argument for approving a security system product based on CPN.

We have also made contact with research groups in Copenhagen who are working with optimising ML compilers such that garbage collection can be avoided [22]. Most ML systems have a garbage collector of some kind. They are interested in AC/DC because they are looking for an industrial case study. We are interested because for embedded systems an automatic garbage collector may cause unpredictable and unwanted long waiting times, and getting rid of garbage collection is expected to be a benefit in practice. In the access control system the garbage collector has never been an issue, but this may be different for other kinds of embedded systems, e.g., where real-time behaviour is more critical.

Finally we need to generalise and optimise code generation in several directions. There is ongoing work of defining parametrised CP-nets [2, 12]. Thus we need to consider code generation for parametrised CPN modules which is expected to be realistic because SML also supports parametrised modules. It is also interesting to minimise the amount of generated code. With a minimal CPN model and Moscow ML we need at least about 160kb memory which is the best we can hope for at the moment because Moscow ML is the only ML compiler we know of which generates the least amount of code. Can we find alternative strategies for code generation? What if we generate code, not for a ML interpreter or Java virtual machine, but instead a CPN virtual machine? Can we make a compact CPN interpreter for which we can generate small sized applications? According to the description of the Petri net tool called PACE [32], it can translate models into code for a virtual Petri net machine. This should be investigated further.

Acknowledgements

The AC/DC project was supported by the Danish National Centre for IT-Research (CIT) [26]. The first version of the CPN code generator was developed by T.B. Haagh and T.R. Hansen and is documented in their thesis [4].

Seculon is a trademark of Dalcotech A/S [28]. LonWorks and LonTalk are trademarks of Echelon Corporation [31].

Finally we acknowledge the four referees of this paper who have provided many useful suggestions for improvement.

References

[1] T. Andersen. SECU-DES, Improved Methodology for the Design of Communication Protocols in Security Systems. Final Report, ESSI Project 10937, Dalcotech A/S, May 1996.
Online version: www.esi.es/ESSI/Reports/All/10937/.

[2] S. Christensen and K.H. Mortensen. Parametrisation of Coloured Petri Nets. Technical Report DAIMI PB – 521, University of Aarhus, Computer Science Department, April 1997. ISSN 0105-8517.

[3] B. Grahlmann. The PEP Tool. In *Tool Presentations of ATPN'97 (Application and Theory of Petri Nets)*, June 1997.

[4] T.B. Haagh and T.R. Hansen. Optimising a coloured petri net simulator. Master's thesis, University of Aarhus, Department of Computer Science, Denmark, 1994.

[5] Persimmon IT. MLj a SML to Java Bytecode Compiler. Web site. http://www.dcs.ed.ac.uk/home/mlj/.

[6] K. Jensen. *Coloured Petri Nets — Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts*. Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 1992.

[7] B. Jørgensen. Målte dybden før springet. *CIT NYT*, 2:8–9, June 1999. Article in newsletter, in Danish.

[8] J.L. Knudsen and B. Magnusson M.Löfgren, O.L. Madsen, editors. *Object-Oriented Environments: The Mjølner Approach*. Prentice-Hall, 1993.

[9] M. Kristensen, S. Christensen, and K. Jensen. The practitioner's guide to coloured petri nets. *International Journal on Software Tools for Technology Transfer*, 2(2):98–132, December 1998.

[10] C. Lakos and C. Keen. LOOPN++: A new language for object-oriented petri nets. In *Proceedings of Modelling and Simulation (European Simulation Multiconference)*, pages 369–374, Barcelona, 1994. Society for Computer Simulation.

[11] G.A. Lewis. Producing network applications using object-oriented petri nets. Master's thesis, University of Tasmania, November 1996.

[12] T. Mailund. Parameterised Coloured Petri Nets. In K. Jensen, editor, *Proceedings of the Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, Department of Computer Science, University of Aarhus, 1999. To appear.

[13] MARS-Team. The CPN-AMI environment. Technical report, MASI lab, Institut Blaise Pascal, Universit Pierre & Marie Curie, Paris, France, October 1993.

[14] D. McQueen. Standard ML of New Jersey. Web site. cm.bell-labs.com/cm/cs/what/smlnj/.

[15] R. Milner, R. Harper, and M. Tofte. *The Definition of Standard ML*. MIT Press, 1990.

[16] University of Aarhus, Dalcotech A/S, DELTA, and CIT. Automatisk kodegenerering fra farvede petri net. Final report for CIT-project number 106. In Danish, May 1999.

[17] J.-H. Paulsen. *Design/CPN Automatic Code Generation User's Guide*. University of Aarhus, Department of Computer Science, Denmark, version 0.6.6 edition, June 1999.

[18] J.-H. Paulsen. *Design/CPN LonTalk Library User's Guide*. University of Aarhus, Department of Computer Science, Denmark, version 0.4.5 edition, June 1999.

[19] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 2 edition edition, 1996.

[20] J.L. Rasmussen and M. Singh. Designing a security system by means of coloured petri nets. In J. Billington and W. Reisig, editors, *17th International Conference on Application and Theory of Petri Nets 1996*, volume LNCS 1091 of *Lecture Notes in Computer Science*, pages 400–419, Osaka, Japan, June 1996. Springer-Verlag.

[21] S. Romanenko and P. Sestoft. *Moscow ML owner's manual*, version 1.43 edition, April 1998. Web site: www.dina.kvl.dk/∼sestoft/mosml.html.

[22] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T.H. Olesen, P. Sestoft, and P. Bertelsen. Programming with regions in the ml kit. Technical Report 25, University of Copenhagen, Department of Computer Science, 1998.

[23] S. Weeks. MLton User's Guide. Available online: www.neci.nj.nec.com/PLS/MLton/.

[24] AC/DC Project. CIT project number 106. Web Site. www.daimi.au.dk/CPnets/ACDC.

[25] Artifex by ARTIS. Web Site. www.artis-software.com.

[26] The Danish National Centre for IT Research. Web Site. www.cit.dk.

[27] Coloured Petri Nets at the University of Aarhus. Web Site. www.daimi.au.dk/CPnets.

[28] Dalcotech A/S. Web Site. www.dalcotech.dk.

[29] DELTA Software Engineering. Web Site. www.delta.dk.

[30] Design/CPN Online. Web Site. www.daimi.au.dk/designCPN.

[31] Echelon Corporation. Web Site. www.echelon.com.

[32] World of Petri Nets. Web Site. www.daimi.au.dk/PetriNets.

Generation of Executable Object-based Petri Net Skeletons Using Design/CPN

Daniel Moldt and Heiko Rölke

University of Hamburg, Dept. for Computer Science, Vogt-Kölln-Str. 30,
D-22527 Hamburg, {moldt, 3roelke}@informatik.uni-hamburg.de

Abstract. Object-oriented Petri nets have gained a lot of interest in the last years. We used the tool Design/CPN in several projects, especially for Object-Oriented Coloured Petri Nets (OOCPN). However, Design/CPN does not support OOCPN directly. To allow easier modelling we developed a net generator (GPS = Generator for Petri net Systems). GPS works on class diagrams like those presented in the Unified Modelling Language (UML). The output of GPS are OOCPN, where each class and its objects are represented in one separate specific Coloured Petri Net (CPN). The generated nets are directly executable in Design/CPN as far as creation and deletion of objects are concerned. Methods and variables are prepared as well, but the user has to give some meaning to the method skeletons. Associations are represented as classes.

Keywords: Coloured Petri Nets, Design/CPN, Net Skeletons, Net Generation, Object-Orientation, Prototyping, Computer Tools

1 Introduction

In our group the integration of object-orientation and Petri nets is an ongoing topic (see [BM93], [Mai96], [Mol96], [Val91], [Val98]). One major goal is to allow for the use of Petri nets as the means to execute object-oriented specifications. To achieve this a transformation of the models to Petri nets is necessary. In [Mol96] for several techniques transformation schemes were proposed, but no tool was presented at that time. In this paper we will concentrate on the central technique, the class diagrams. For the class diagrams again only the basic kinds of associations and some tool support by Design/CPN are considered. Another major goal of the transformation is to get a deeper understanding of static models and to provide better semantics to techniques like class diagrams.

Class diagrams exist in several variants. The most important one is that of the Unified Modeling Language (UML) (see [BRJ99,JBR99,RJB99]). This version of class diagrams combines most of the features that have been developed for static relationship modelling so far.

Our planned specification environment aims at the support of several techniques. This includes the integration of the different models. Since object orientation is the central paradigm, the starting point for the integration are the classes (or objects). In [Mol96] three main views on a system are discussed: the static view, the dynamic view, and the functional view. All these views are modelled for an object and can then directly be integrated after they are transformed into Coloured Petri Nets. On this basis only one formalism is given and hence the integration is easier. Modellers can present an executable specification to users. However, this requires some additional input for the improvement of the user interface. Otherwise the user has to interpret the net simulation directly.

At the current state only fragments have been supported by tools. Due to the fact that tools are a precondition for the successful application of OOCPN, we are working on these tools. In this paper we will show for the central concept of classes (and objects) how to transform class models to OOCPN class models. The basic assumption is that even for the static models (class diagrams) an operational semantics has to be provided. Each relation can e.g. be classified as being mandatory or optional. Mandatory relations cause some actions in the related classes when in the other classes objects are created or deleted. This cascading of operations is usually not modelled elsewhere in UML. To interrupt this cascading would violate the integrity constraints that are expressed by the relationship. When providing an operational semantics for this feature of the class diagrams, a specific protocol can be added to other ”standard” protocols for methods, attributes etc. All other kinds of attributes of relations require also some kind of protocol. These protocols may differ considerably.[1]

It is important to notice that we do not present a generator of Petri nets for class diagrams in general including all features and their implicit protocols[2]. So far we have concentrated on to provide the basic infrastructure, which means to generate one Hierarchical Coloured Petri Net (HCPN) for one class. Each usual relationship (association) is also considered to be a class. This allows to assign methods and attributes to associations. The related protocols of the behaviour e.g. to a `(1:N)` relationship are not automatically generated. As many other features these could be integrated on demand. However, this would require some specific programming for which we had no time resources up to now. It is interesting to see that Entity-Relationship-Diagrams (ERD) (see e.g. [You89]) can be handled in the same way as class diagrams.

1.1 Modelling process

Figure 1 shows the steps of how to derive from object-oriented models the Object-Oriented Coloured Petri nets within the Design/CPN tool. In the following description we concentrate on the class diagrams and omit the other techniques. The Petri nets generated for the other techniques would be integrated into the here described models.

The user models the class diagrams and receives by this the executable Petri nets in Design/CPN. The large box represents the overall action. First of all

[1] With protocol we do not mean e.g. TCP/IP, but the rules which determine simple sequences or parallel actions that serve to support the interaction of two or more objects. An example is the instantiation of a second object which needs to be present due to an association between the related classes and the instantiation of an object of one of the classes. This will be explained in the following sections in more detail.

[2] This is the reason why we use object-based instead of object-oriented in the title of this contribution (see also [Weg87])

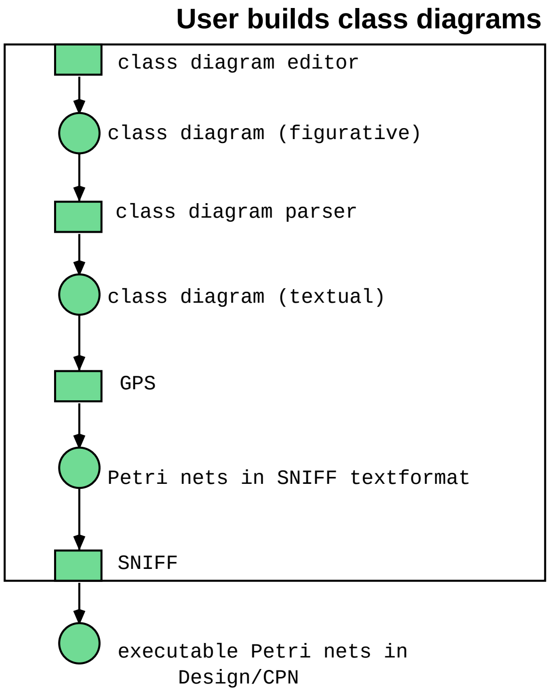


Fig. 1. Steps to derive executable Object-based Petri net skeletons

the user has to draw the class diagrams. There is an ongoing work (diploma thesis) at our department to support the drawing of class diagrams within the Design/CPN environment[3]. Any other tool than Design/CPN can be used as long as its output format can be translated to the class diagram text format that is currently used by the GPS toolset[4]. Its output format are Design/CPN-oriented Coloured Petri nets in the SNIFF text format [MMR98][5]. This text format for Coloured Petri nets can be imported to Design/CPN and can be handled in a way as if the models were drawn by the user.

Except for the drawing of the class diagrams all parts of this procedure are executable within Design/CPN because they are implemented using the Standard ML language [Des93,Pau92]. All functions are written in a way that enables their concatenation: The output of the first function is taken as the input for the next and so on. These function calls can be hidden from the user, who only has to call one function without arguments. The user designs the class diagrams

[3] Remark: One goal is to do all tasks within one environment, here the Design/CPN environment.

[4] This text format is an ad-hoc approach to class diagram text formats. At this point also a standard like IDL (Interface Definition Language) could be used. With the times changing we will surely switch over to a standard notation.

[5] SNIFF is an input/output library for Design/CPN presented at the Design/CPN workshop in Aarhus in 1998. The newest version of Design/CPN has now a tool being build in that allows the im- and export, however, the format is different (see [LM98]).

within one page of Design/CPN and gets the resulting Object-Oriented Coloured Petri nets on just one mouse click. Changes to the underlying diagrams do not require a change of the environment. Therefore, the ML functions acting behind the scene are invisible for the user, they behave like a refined transition of a Petri net. Therefore, the large box in Fig. 1 can also be interpreted as one transition, not showing the required input from the outside in the same way as this is not done for the single transitions (or functions).

Nevertheless all single functions are accessible for the experienced user. This assures maintainability, adaptability, and expandability.

1.2 Why object-based nets?

Petri nets have proven to be a useful formalism to express concurrency. Problems related to concurrent processes and distributed algorithms are getting more and more important to be managed, i.e. solved, even in mainstream computer science. One idea to tackle this situation is to combine the results from different fields of computer science. We have chosen Coloured Petri Nets (CPN) and object orientation (OO) as the main concepts. This allows to combine the structuring facilities of OO with the advantages of a technique with a sound theoretical background. Up to now object-oriented specifications, like those of UML, are hardly executable. With our approach this problem is (partially) overcome. However, the resulting nets are too large to be used directly by a modeller. Therefore, we propose the use of specialised tools and here we concentrate on the GPS. Due to the simple nets, we support up to now, we call the generated nets object-based skeletons. These skeletons have to be completed to contain the functionality. In combination with other tools which cover the modelling of the functionality this could also be automated as well (as far as a generator can help here).

The rest of the paper contains two main parts: Section 2 shortly presents the main relevant concepts of Object-oriented Coloured Petri Nets and section 3 discusses the generator GPS and its implementation. The paper ends with a short conclusion and an outlook for future work.

2 Object-Oriented Coloured Petri Nets

This paper is intended to show the computer aided implementation of Object-Oriented Coloured Petri nets as proposed by Moldt [Mol96]. With respect to this aim other approaches to combine both the advantages of Object-Orientation and Petri nets are not discussed here. For some work of object-orientation and Petri nets see for example [SB94,Lak95,Val98,Mai96,Kum00,BG91,EMNW99,MM99].

Due to limited space the introduction of Petri nets in general and Coloured Petri nets in special is skipped here as well as the presentation of well-known concepts of Object-Orientation and techniquess like UML class diagrams. To become familiar with these concepts see for the study of Petri nets [Pet62,Rei92,Jen92] and for Object-Oriented concepts [Lou93,Fow97,RJB99].

Objects, classes, and methods are the basic components of Object-Oriented programming languages [Lou93]. An object invokes a method (service) of another object by sending it a message. In the following subsections, Moldt's transformation of objects, classes, and a messaging mechanism to Petri net representation will be introduced (see [Mol96]).

2.1 Objects

An object encapsulates its state and behaviour. The only way to change an object's state is through the methods provided at the interface of the object. In an Object-Oriented programming language classes are defined and objects, so-called *instances* of these classes, are created during runtime. Local variables of an object are also called instance variables.

```
class Class_x {
public:
   int method_1 ();
   void method_2 ();
   ...
   int method_n ();
protected
   anytype inst_var;
};
```

Fig. 2. Class definition from Moldt [Mol96]

Transformation of Objects The class definition in figure 2 defines `n` methods[6]. In addition, a local variable `inst_var` is declared which can only be used by the class itself, or by a subclass. An object (instance) of this class, called `OBJECT_X`, is shown as a net in figure 3 and is called Object-Net[7]. Variables are transformed to places, methods to transitions. An object is a marked net page according to Jensen [Jen92][8].

The object receives and sends messages using the two places `in_pool` and `out_pool`. These two places are the interface between the object and the message

[6] The examples are given using a notation similar to C++ without claiming to be syntactically correct in that way.

[7] The net inscriptions are given in the functional language ML as it is used in the Petri net tool Design/CPN. For the sake of simplicity parts of the net inscriptions are omitted to focus on the core concepts. The given nets are therefore not syntactically correct according to Jensen [Jen92] in the sense that arc inscriptions etc. have to be added. Concurrency is not restricted by the model introduced here. Objects can process several messages at the same time.

[8] The marking is normally omitted because the main purpose here is to show the static aspects.

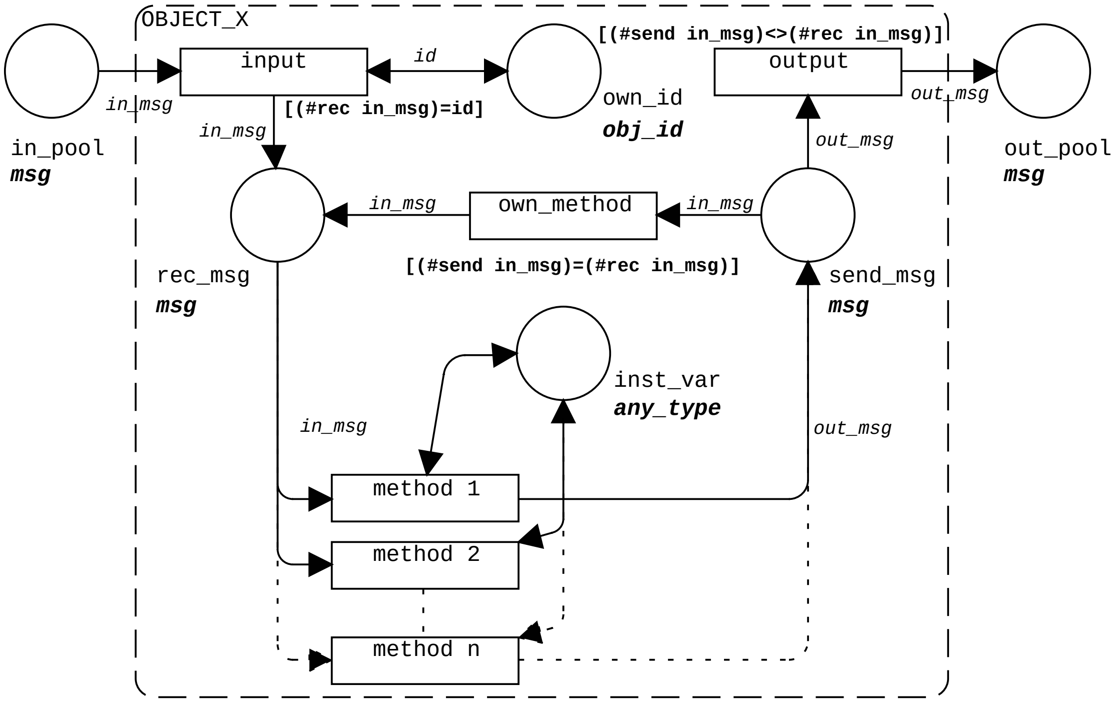


Fig. 3. Object-Net from Moldt [Mol96]

handler. Depending on whether one considers these places as part of the object or not, an object is a place- or transition-bounded net. The message handler is responsible for transporting the messages from `out_pool` to `in_pool` places of objects.

The transition `input` selects the appropriate messages from the `in_pool` using the side condition `own_id`, thus ensuring that only a message for the object reaches the place `rec_msg`. Because a method can only be invoked by a message in the place `rec_msg`, the presented Object-Net realises an encapsulation of the object's state and behaviour.

A method selects its appropriate message using a guard that refers to the method name in the message. The functionality of the method can be given as a net refinement or a code segment as being used in the Petri net tool Design/CPN. After computation a reply message is put into the place `send_msg` that is sent to `out_pool` by the transition `output` or, if the receiver of the message is the object itself, is put to the place `rec_msg` by the transition `own_method`.

One can look at an Object-Net as consisting of two parts. The transitions `input`, `output`, `own_method` and the places `rec_msg`, `send_msg` are common to all objects and can be used as a template for objects. The specific state and behaviour of an object is being realised through the places for instance variables and the transitions for methods.

Object Identity Beside its state and behaviour, an object has a clear identity, i.e. a unique key that is used as a reference. In this proposal a unique key called Object-ID is used. An Object-ID denotes exactly one Object-Net and is generated during the creation of an object. To get a unique key the following convention is used. The Object-ID consists of the class name and an atomic expression locally unique for this class, usually a number of type *integer*. The class name is assumed to be unique inside the system.

2.2 Classes

A class defines the internal state and the behaviour of its objects and therefore their type. One can look at classes as having a construction plan to manufacture objects. Sometimes they are therefore called "Factory Objects". In addition to defining its objects' type, a class provides operations to create, destroy and manage them. A class can receive and send messages and can therefore be looked at as an object, too.

The creation and deletion of new objects, the refinement of methods, and the concept of relationships will be discussed in more detail in Section 3 but not in this subsection.

Folding Objects to Classes Corresponding to the type definition of variables, objects are declared belonging to a class, assumed the language is typed. Therefore we need a mechanism for nets to define classes and later create and manage objects of that class. This is done by folding Object-Nets with the same state and behaviour[9] to so-called Class-Nets (Figure 4). The type of each place is extended with the Object-ID. A place in the Object-Net with type `any_type` now gets the new type `obj_id*any_type` in the Class-Net. The places `own_id` in the Object-Nets are folded to the place `all_Inst` (all instances) in the Class-Net. The place `class_id` in the Class-Net holds the Object-ID (the name) of the class. The type of the variables used in the arc expressions is extended in the same way. Thus it is ensured that the Class-Net has the same behaviour as the folded Object-Nets before.

In addition, the Class-Net is enriched with class operations like `new`. It is therefore capable of defining the type of its objects and to create and manage them. In a Class-Net, the tuple `obj_id*any_type` is called `inst_type` for the sake of simplicity, `obj_id*msg` is called `msg`. A Class-Net gets the reserved Object-ID `(<classname>, 0)`, called Class-ID. Using this Class-ID one can send a message to a Class-Net, e.g. to invoke the creation of a new object. The way to create or discard objects or to perform other class operations is based on the idea that classes are objects. A class contains method transitions that describe the desired action to be invoked by calling the method.

[9] Again we refer to the static aspects of state and behaviour: All folded Object-Nets must have the same places and transitions but may differ in their markings at runtime.

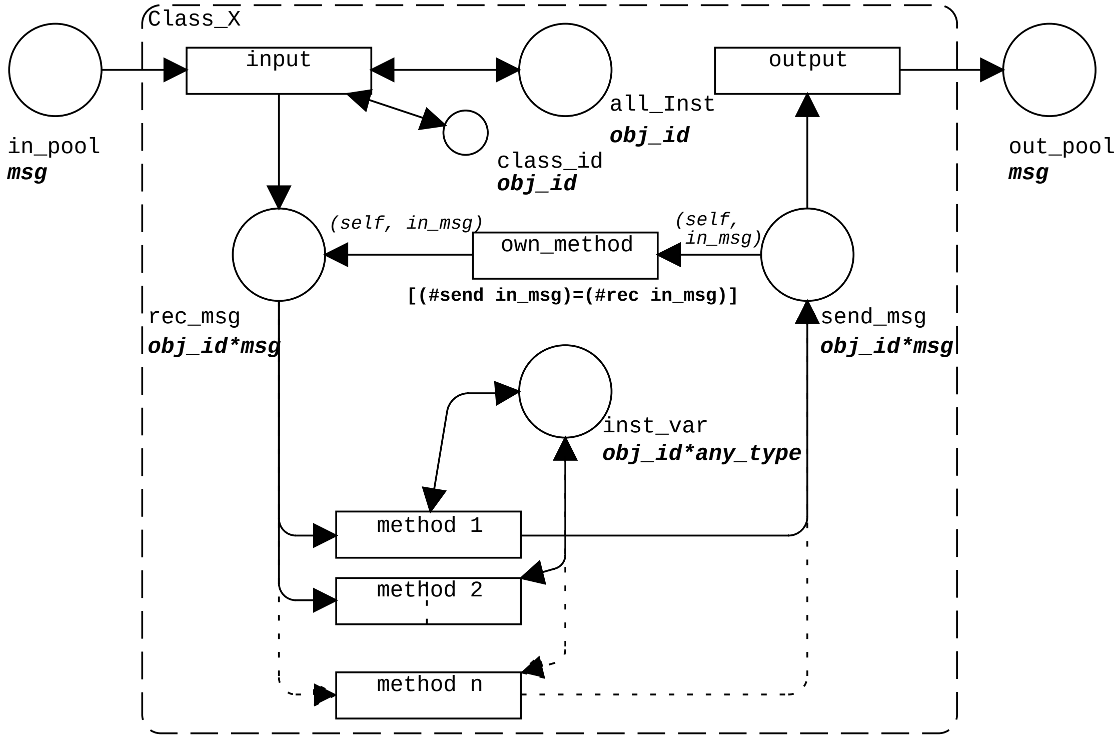


Fig. 4. Folding Object-Nets to Class-Nets

2.3 Messages

Messages are used to invoke a method in the destination object. In this approach messages are realised using tokens that are exchanged between objects. An error-free messaging mechanism is assumed. The message format has to cover information about sender and receiver in term of class, object, and method as well as parameters which have to be passed around. Depending on the kind of message system different message formats are required. In this paper we use a straightforward one which explicitly represents all information directly. Consequently the message parsing has to be done within the net. One could argue that this could be done within the message handler and could therefore be hidden. This is true, however, in [Mol96] the principle structure should be visible and for convenience we have also chosen this format for our generator. In further work we will also support other message formats.

The Message-Handler A message-handler is used to route the messages between objects. One way to realise this message-handler is by simply fusing all the in_- and out_pool places using place fusion. In a distributed environment a more sophisticated version should be used to model different routing strategies or incorporate possible losses of messages. The concept of a message-handler has therefore been included to provide the possibilities to model these relevant aspects of distributed systems.

3 Design

The developer of an environment for computer aided design of Object-Oriented Coloured Petri nets has to take care of certain procedures like the creation of new objects. The traps like the finding of a unique identifier for the new object should be handled by the system in order to aid the user. Object-Oriented Coloured Petri nets offer some features from the Object-Oriented Analysis like associations. These features lead to even more complex protocols that sometimes include the interplay of several different classes. To release the modeller from the burden of explicitly modelling we propose to generate these protocols from more abstract diagrams. The case of an automatic generation of association skeletons without the protocols for the specific behaviour will be handled during this chapter. All Petri nets illustrating the following pages are shown as they are (automatically) generated by the GPS tool set. They are not adjusted in any way.

3.1 Association protocols

A protocol to handle relations between objects serves on the one hand to disburden the modeller of an Object-Oriented system. The protocol is needed to automatically handle a relation and to move it to the desired code. On the other hand a protocol gives a clear meaning to the constructs of a diagram. In larger projects the class diagrams are cut into pieces and given to different programmers. Every programmer has its own view on the system and on how to handle e.g. the relations. This can lead to misinterpretations followed by mistakes. The code of one programmer is not understandable by another.

In order to reach this aim it is necessary to have a clear understanding of relations between classes. Usually relations are categorised being “simple” or “complex”. A simple relation should be handled by the related classes itself while a complex relation needs a so-called “relation class” to be implemented. Again it is not clear how to formalise such a distinction. Due to this reason the presented system generates an new class for each relation. This is obviously not necessary for very simple cases like a one-to-one relation without attributes or methods but the computer does not complain about the extra work. The disadvantages concerning performance, however, are obvious. Therefore, we plan to provide protocols for ”simple” relations such that no separate relation has to be build. It might even be possible to integrate two classes that have a one-to-one relation. However, this has not been done up to now.

The set of possible relations between objects or classes divides into two subsets with different properties: Mandatory and optional relations. This distinction is motivated by giving example procedures. The principle situation is as follows: Two classes have a certain association between them. This association has a certain multiplicity for each of the objects. It has for each of them the information if the relation between objects of this class is mandatory or optional. If for one of the classes an abject is created the association now implies a certain behaviour (protocol) at the other side of the association. This is discussed in the following.

Mandatory relations An object of a class in a mandatory relation shall be created. We first consider the case of an one-to-one relation, that has to be distinguished from other multiplicities like one-to-many or many-to-many. In the first case exactly one corresponding counterpart of the newly created object and an object of the relation class have to be created. In the latter case the new object may be related to one or more existing objects of the relation class. The relation class itself has to handle the co-ordination between the objects, which can be toilsome especially when the multiplicities of the relations have restricted number areas. The choice of the right relation classes for a newly created object can not be done automatically. The user has to be prompted for that choice or the diagrams have to be extended to cover some information about this issue. This is true in general. However, depending on certain conventions, project specific rules, application specific requirements etc. there are some cases where this could be automated. In the normal case, however, the developer has to decide what has to be done. This process again can be done at different points of time and depends heavily on the available tools. One idea this paper wants to present is that there is a problem and that there are certain ways to solve them. Here only the main stream is discussed. For an efficient and practical application tools are a prerequisite.

The erasure of an object in an one-to-one relation enforces the erasure of the related object and the object of the relation class. The erasure of related objects can lead to troublesome situations[10]. The situation of having some states when the whole system is inconsistent is one of the main reasons to use protocols. Only if a protocol finishes successfully the operation is really performed. Otherwise the system has to reset the already performed actions. Exactly this point makes the protocols so important and so difficult to describe in general. A general simple solution is not available. However, with the appropriate tools the situation becomes more relaxed. It should not be forgotten that in the area of databases this problem has largely been solved. E.g. the automatic generation of database schemes shows how to cope with this kind of problem. For the area of analysis and in specific for Petri nets there are still some open questions.

Optional relations If a relation between objects is not mandatory things are a little bit easier to handle. If it is not necessary for an object to have a related counterpart our environment does not have to take care about the relation class at all. But such a procedure might not be an adequate aid for the user of our system. So the user can be prompted if he is willing to put a new object in some relation or not.

3.2 Class net frontpages

This subsection contains an example of a generated Object-Oriented Petri net. Before we introduce the example with a one-to-many multiplicity, we introduce the example with the artificial one-to-one relationship. The company with only

[10] See for example [Mol96].

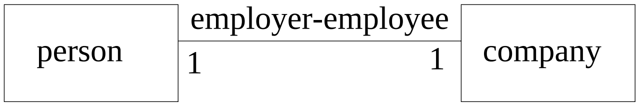


Fig. 5. Simple class diagram with relationship

one employee is very small here and gets larger in the one-to-many case. The example consists of two classes, *company* and *person*, standing in relation employer-employee (see Figure 5). The first trivial example will also be extended by payment as an attribute and by a method to tell the payment for a special instance of the relationship.

A textual description[11] of this relationship would be as follows:

```
interface Person {
   associates
      company 1-1 employer-employee
}

interface company {
   associates
      person 1-1 employer-employee
}
```

To make the example more interesting we enhance the class *company* with an attribute, the company's name and two methods to get and set this name. Figure 6 shows the Petri net frontpage (or rootpage) of this class. The extended textual description is:

```
interface company {
   associates
      person 1-1 employer-employee
   attributes
      string companyName
   methods
      string getName(void);
      bool setName(string);
}
```

[11] See appendix A for the syntax of this description language.

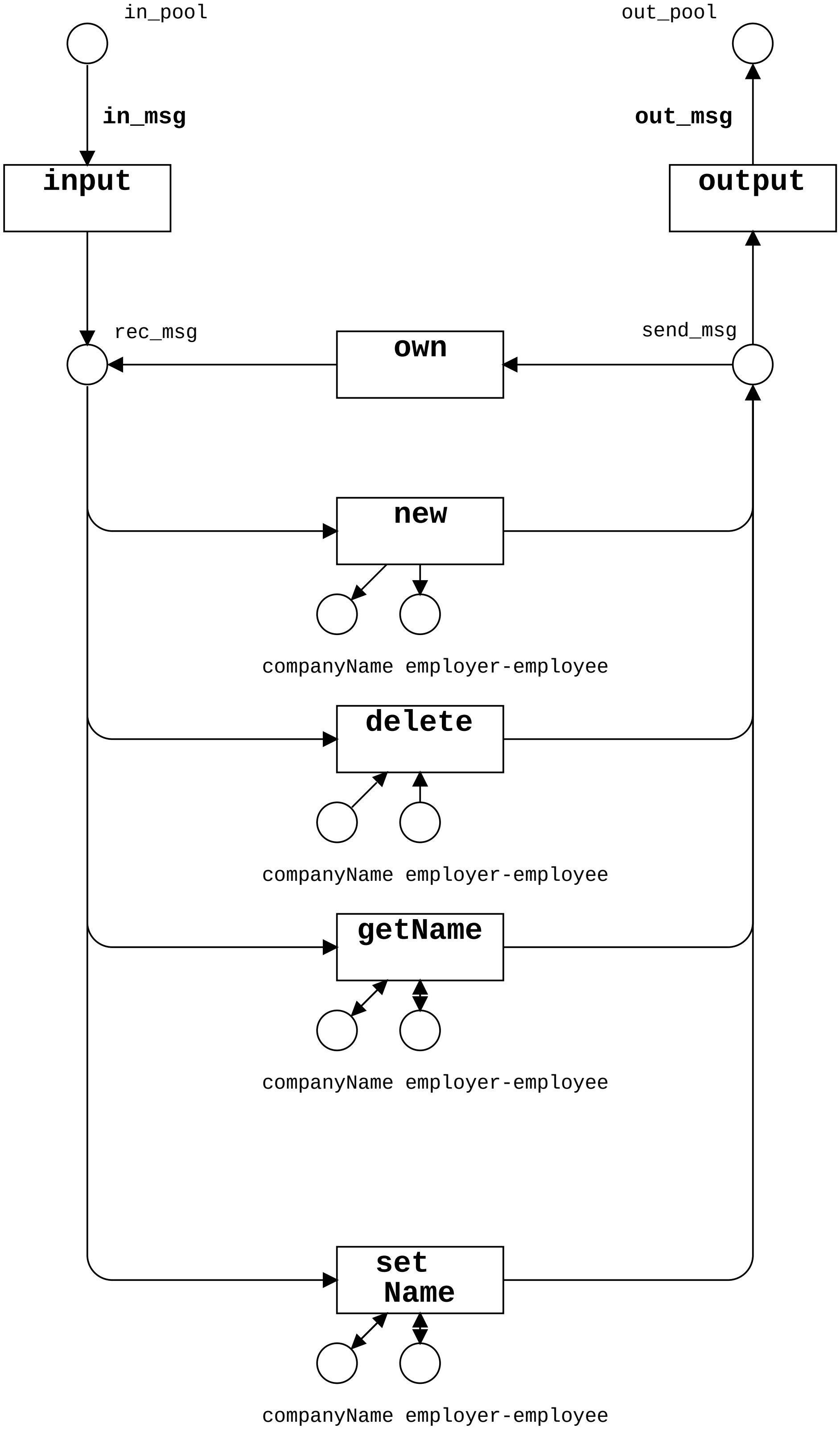

in_pool
out_pool
in_msg
out_msg
input
output
rec_msg
own
send_msg
new
companyName employer-employee
delete
companyName employer-employee
getName
companyName employer-employee
set
Name
companyName employer-employee

Fig. 6. Frontpage of Class-Net *company*

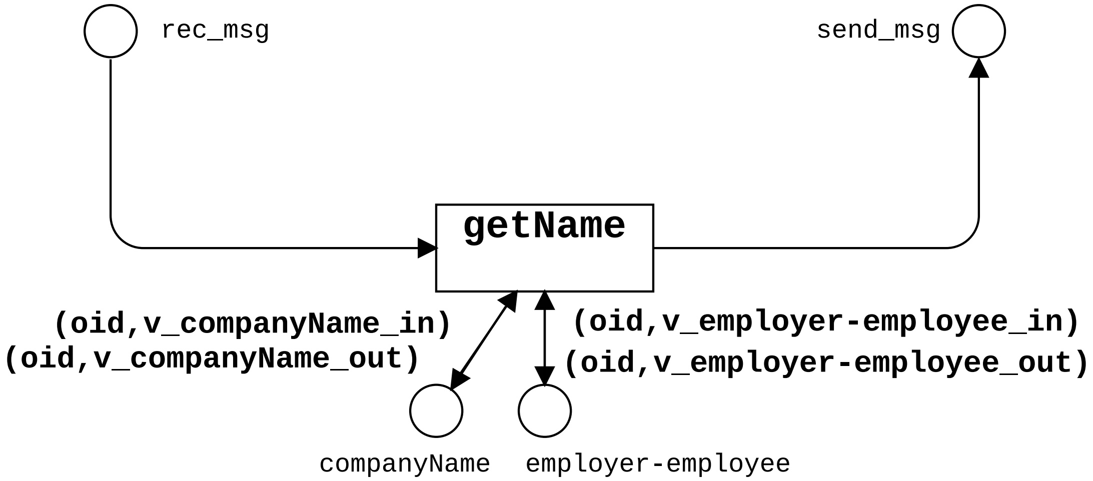


Fig. 7. Subpage of method *getName*

All methods can access the two attribute places. Place *companyName* results from the attribute of the class description above. Note that all places with the same name are connected via the concept of place fusion available in Design/CPN[12]. These places behave like one physical place and always contain the same marking. The attribute place *employer-employee* corresponds to the relationship between the classes company and person. Its name is that of the relation and its marking contains the identifier of the relation class instance that belongs to the actual object. Despite of the few visible inscriptions this net is fully executable in the sense that one can call the class method *new* to create new instances of that class and operate with them at will.

The implementation of the method functionality is generally handled using subpages, the hierarchy mechanism of Design/CPN. This assures a clear frontpage even in complex classes with many methods. Notice that the arc orientation reflects the method's functionality: The method *new* puts new markings to the attribute places, *delete* takes them away. As there is no information on the internals of the other methods they generally get two-way arcs as a default.

3.3 Method subpages

Figure 7 shows the subpage to the method *getName* as it is generated by GPS. On subpages the arcs that access the attribute places are labelled. They get their own names depending on reading (_out) or writing (_in) access. The transition that holds the name of the refined method offers an easy way to implement the methods functionality. It is attached with an predefined code region that is not mentioned in figure 7. This code region imports all attributes that are connected

[12] The tool GPS automatically hides the detailed inscriptions that are necessary in Design/CPN to get things like place fusion going. If the user wants full access to the generated nets he can easily adjust GPS not to hide anything.

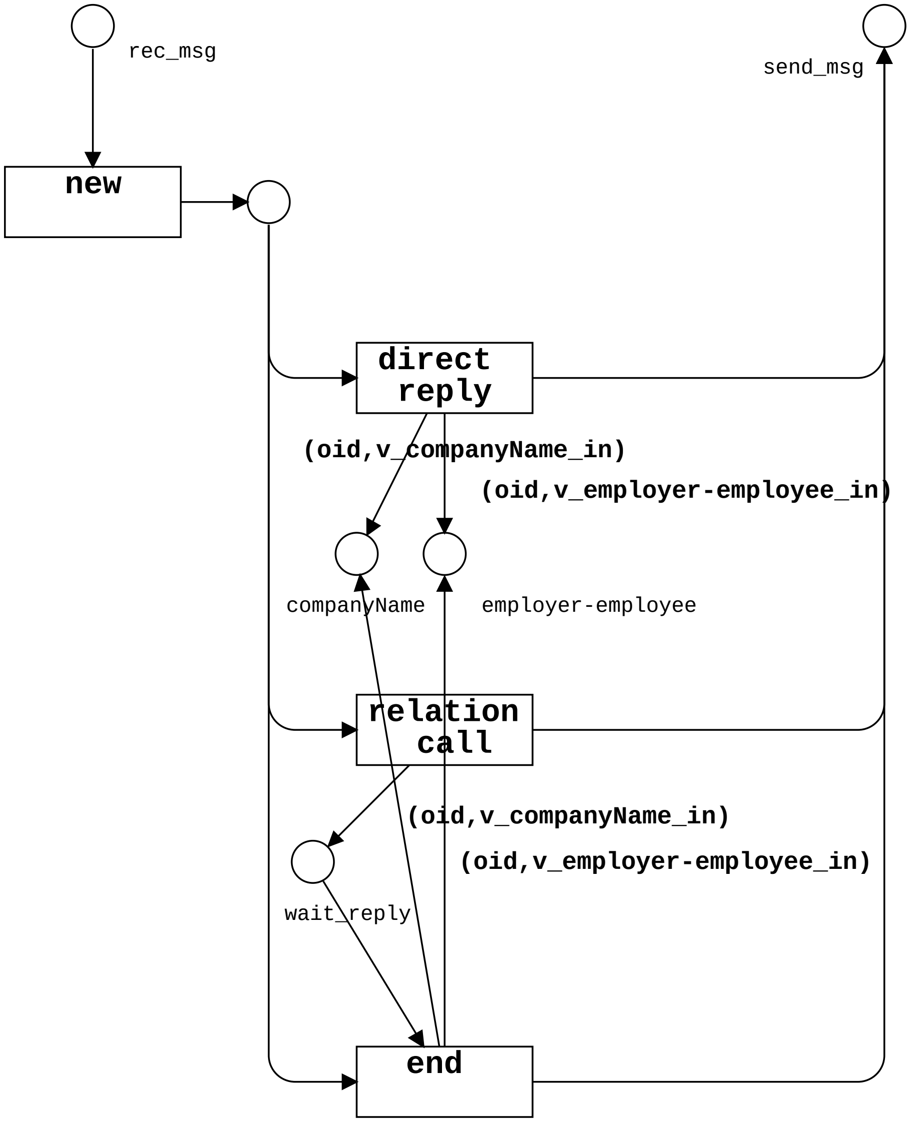


Fig. 8. Subpage of method *new*

with the transition and contains a predefined reply message. Simple methods can be finally implemented by the user in a very short amount of time using this predefined code region. A more complex method can certainly be implemented by the user utilising all the functionality offered by Design/CPN.

The class method *new* can normally be left unchanged by the user. Figure 8 shows the method's subpage in the case of a class connected to another via one relationship. The method has to distinguish two variants of being called: It can be called by any external object that knows the class or by the relation class. The call of *new* by the relation class is easier to handle because all possible dependencies that arise from the relationship are already satisfied by the calling relation class. The method just has to put some default values and the identifier of the calling relation class instance into the attribute places and reply to the caller. This is done by the transition *direct_reply* in figure 8. If the method is called by an external object there is a little bit more work to be done: A new instance of the relation class has to be created, this instance has to fulfil all dependencies arising from the relationship. After doing this the relation class object reports back to the waiting new method where the transition *end* is activated and finishes the creation of the object. The call of the relation class is done by transition *relation_call*. This transition also puts an appropriate token into place *wait_replay*. The reply message sent back by the newly instantiated relation class object matches to this token, so that transition *end* can fire. Again a reply message is send to the first calling external object.

3.4 Relation classes

The frontpage of the relation class is so much alike to that of a normal class that the picture is skipped here. It shows the class methods *new* and *delete* and the methods *person* and *company* that help detecting the related counterparts of a given object.

Figure 9 shows the new method's subpage of the relation class hiding less net elements then the figures before. This is done to allow a closer look on the complexity of the generated nets. Normally most of these details are not shown to the user but the degree of complexity shown is easily adjustable within GPS. Due to limited space a description of the course of events happening in the relation class is omitted here. The interested reader may refer to [Hei99] where some detailed example procedures are presented as well as the quite complex code regions of the transitions. Furthermore, the declaration nodes and how construct them according to the input by the designer can be found there. Especially the creation of correct inscriptions is a tidious task. This can be supported by tools.

4 Conclusions and Future Work

To design Coloured Petri Nets in an object-oriented way implicit protocols for the object behaviour have to be created. The GPS tool allows the automatic generation of nets from class diagram descriptions. Relationships between classes are also considered to be classes. They are special classes that incorporate a specific behaviour to ensure the operational behaviour that one expects for the associated classes (e.g. when creating or deleting objects or when navigating from one object to another). However, the protocols which cover these different kinds of behaviour are not shown in this paper. They still have to be added. It should also be mentioned that these protocols must cover many different cases. Even for simple standard cases the starting point of an creation of some objects (related by a mandatory relationship) can affect the kind of protocol. Furthermore, the information that has to be added (in programming languages these would be the parameters), has to be provided. When also covering inheritance there is an

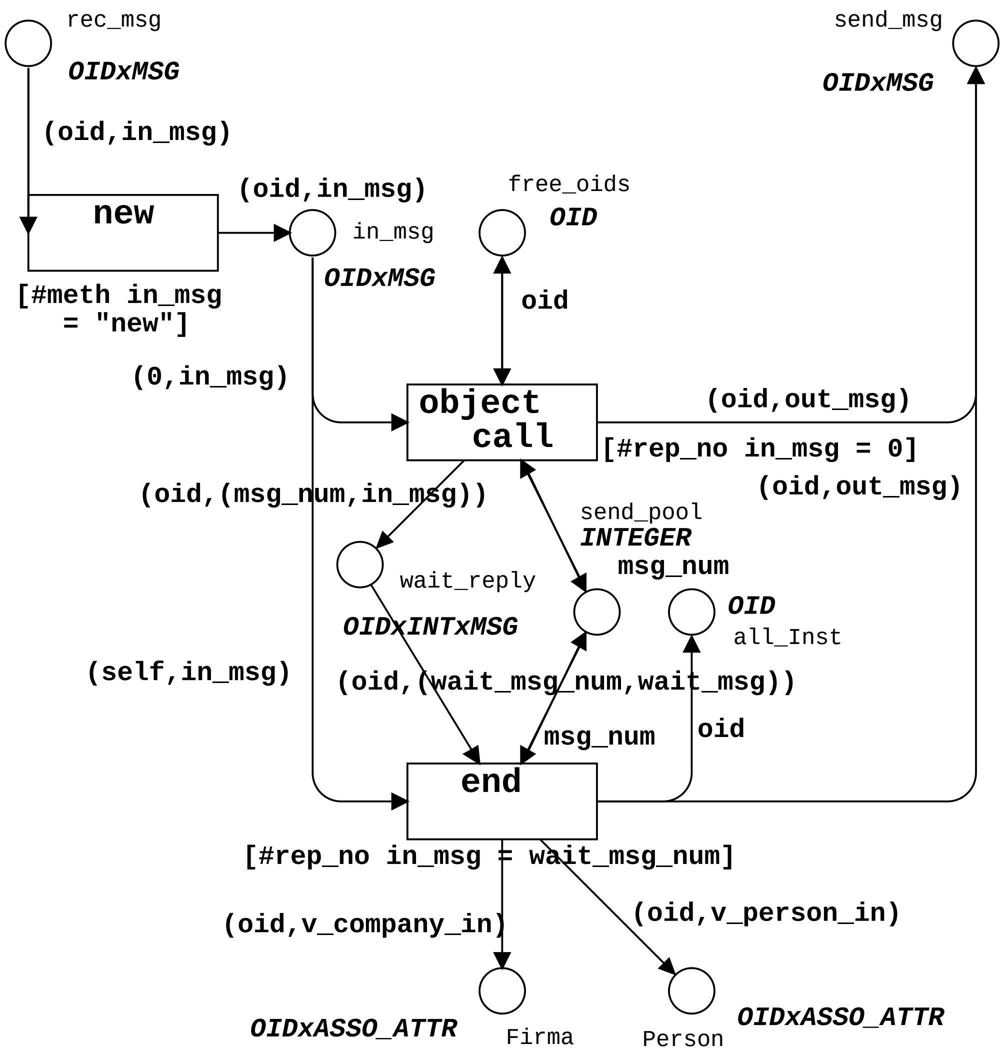


Fig. 9. Subpage of method *new* of the relation class

explosion of the possible entry points[13]. We have drawn some protocols by hand, but did not write the generator for it. Furthermore, more complex associations like inheritance have to be handled. For our general attempt to use OOCPN for specification or at least for execution the missing features have to be provided to provide a reasonable specification environment. The generator is build in a way that it can be easily extended by plug-ins. These plug-ins could then cover the different protocols.

The use of ML for the generator allows the easy integration into the Design/CPN environment, however, hinders the transformation to other environments which are e.g. based on Java.

[13] In [Mol96] delegation is used to implement the inheritance. This concept is quite powerful, however, it requires the explicit representation of aspects which should normally be hidden to a user.

The generated nets and some attempts to apply the OOCPN by hand (without tool support (see [Net99]) show that there have to be some additional, more abstract concepts for users to build OOCPN models. The principal ideas for providing an operational semantics for class diagrams in form of Petri nets have shown to be good, however, the operational side has some disadvantages. Problematic is that diagrams got quite large. Overhead to explicitly model all the operational behaviour induced by the object-oriented concepts could not be ignored. It made up a large percentage of the whole model. The large models to express relatively simple behaviour have a considerable impact on the performance (and the applicability of analysis tools, especially due to the growing state space). Many transitions have to fire before the, from the view point of object-orientation, simple action has been performed. This has lead us to look for shorthand notations. UML can be considered to be one possible solution. However, we can not directly provide an abstraction recommendation which can be applied within the Coloured Petri nets themselves.

One of the goals is to get a better understanding of the semantics of e.g. class diagrams. This can be achieved when providing an operational semantics for them. Tool support helps on this way and so does the GPS tool set.

A Class diagram description language

```
diagram                 ::= <classdescription>*

classdescription         ::= 'class' <classname> '{'
                              <relations>
                              <attributes>
                              <methods>  '}'


relations               ::= 'relationship' <relationset>
                            | ()

attributes              ::= 'attributes' <attributeset>
                            | ()

methods                 ::= 'methods' <methodset>
                            | ()


relationset             ::= (
                             <relatedClass>
                             <relationtype>
                             <relationname> ';'
                            )*

attributeset            ::= ( <type> <attributename> ';' )*

methodset               ::= ( <type> <methodname> <arguments> ';' )*


relatedClass,
relationname,
attributename,
methodname              ::= identifier

relationtype            ::= '1-1' | '1-n' | 'n-1' | 'n-m'

arguments               ::= '(' <typeset> ')'

typeset                 ::= <type> ',' <typeset> | <type>

type                    ::= 'void' | 'int' | 'string' | 'real'
```

References

[BG91] Didier Buchs and Nicolas Guelfi. CO-OPN: A Concurrent Object Oriented Petri Net Approach. In *Application and Theory of Petri Nets, 12th International Conference, Gjern, Denmark*, pages 432–454. University of Aarhus, IBM Deutschland, June 1991.

[BM93] Ulrich Becker and Daniel Moldt. Objektorientierte Konzepte für gefärbte Petrinetze. In Gert Scheschonk and Wolfgang Reisig, editors, *Petri-Netze im Einsatz für Entwurf und Entwicklung von Informationssystemen*, Informatik Aktuell, pages 140–151, Berlin Heidelberg New York, 1993. Gesellschaft für Informatik, Springer-Verlag.

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The unified modeling language user guide: The ultimate tutorial to the UML from the original designers.* Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[Des93] Meta Software Corporation, Cambridge, MA, USA. *Design/CPN Handbook Version 2.0*, 1993.

[EMNW99] Adriana Engelhardt, Daniel Moldt, Marc Netzebandt, and Frank Wienberg. Erweiterung objektorientierter gefärbter Petrinetze um Typisierung und Schnittstellen. Fachbereichsmitteilung planned, FBI-HH-M-xxx/99, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, October 1999.

[Fow97] Martin Fowler. *UML Distilled.* Addison-Wesley Longman, Inc., 1st edition, 1997.

[Hei99] Heiko Rölke. Transformation von Klassendiagrammen in Objekt-Orientierte Petrinetze unter besonderer Berücksichtigung von Assoziationen. Studienarbeit, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, April 1999.

[JBR99] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process: UML; The complete guide to the Unified Process from the original designers.* Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[Jen92] Kurt Jensen. *Coloured Petri Nets: Volume 1; Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin Heidelberg New York, 1992.

[Kum00] Olaf Kummer. *Referenznetze.* Dissertation, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, 2000.

[Lak95] C.A. Lakos. From Coloured Petri Nets to Object Petri Nets. In *16th International Conference on the Application and Theory of Petri Nets*, number 935 in Lecture Notes in Computer Science, pages 278–297, Torino, Italy, 1995. Springer.

[LM98] R.B. Lyngsø and T.M Mailund. Textual Interchange Format for High-Level Petri Nets. In *Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, pages 47–64, Ny Munkegade, Building 540, DK-8000 Aarhus C, Dänemark, 1998. Computer Science Department, Aarhus University.

[Lou93] Kenneth C. Louden. *Programming languages: principles and practice.* PWS-Kent, Boston, 1993.

[Mai96] Christoph Maier. Darstellung von Konzepten der objektorientierten Modellierung und Programmierung mit Petrinetzen. Studienarbeit, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, May 1996.

[MM99] Christoph Maier and Daniel Moldt. Object Coloured Petri Nets – a Formal Technique for Object Oriented Modelling. In G. Agha, F. De Cindio, and G. Rozenberg, editors, *Concurrent Object-Oriented Programming and Petri Nets*, number yet unknown in Lecture Notes in Computer Science, Berlin, 1999. Springer-Verlag.

[MMR98] Christoph Maier, Daniel Moldt, and Heiko Rölke. SNIFF – An Input/Output Library for Design/CPN. In *Workshop on Practical Use of Coloured Petri Nets and Design/CPN*, pages 65–82, Ny Munkegade, Building 540, DK-8000 Aarhus C, Dänemark, 1998. Computer Science Department, Aarhus University.

[Mol96] Daniel Moldt. *Höhere Petrinetze als Grundlage für Systemspezifikationen.* Dissertation, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, August 1996.

[Net99] Marc Netzebandt. Untersuchung der Einsatzmöglichkeiten von Petrinetz-Konzepten in der objektorientierten Analyse am Fallbeispiel eines Reiseunternehmens. diploma thesis, University of Hamburg, Department for Computer Science, Vogt-Kölln Str. 30, 22527 Hamburg, Germany, Januar 1999.

[Pau92] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1992.

[Pet62] Carl Adam Petri. Kommunikation mit Automaten. Dissertation, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik an der Universität Bonn, Bonn, 1962.

[Rei92] Wolfgang Reisig. *A Primer in Petri Net Design.* Springer Compass International. Springer-Verlag, Berlin Heidelberg New York, 1992.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The unified modeling language reference manual: The definitive reference to the UML from the original designers.* Addison-Wesley object technology series. Addison-Wesley, Reading, Mass., 1999.

[SB94] C. Sibertin-Blanc. Cooperative nets. In Robert Valette, editor, *15th International Conference on the Application and Theory of Petri Nets*, number 815 in Lecture Notes in Computer Science 815, pages 471–490, Berlin Heidelberg New York, 1994. Springer-Verlag.

[Val91] Rüdiger Valk. Modelling Concurrency by Task/Flow EN Systems. In *Proceedings 3rd Workshop on Concurrency and Compositionality*, number 191 in GMD-Studien, St. Augustin, Bonn, Germany, 1991. Gesellschaft für Mathematik und Datenverarbeitung.

[Val98] Rüdiger Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In Jörg Desel, editor, *19th International Conference on Application and Theory of Petri nets*, number 1420 in LNCS, Berlin, 1998. Springer-Verlag.

[Weg87] Peter Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, Cambridge, 1987. Cambridge University Press.

[You89] Edward Yourdon. *Modern Structured Analysis.* Prentice-Hall, Englewood Cliffs New Jersey, 1989.

Batch Scripting Facilities for Design/CPN

Bo Lindstrøm & Lisa Wells

Department of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C
Denmark
E-mail: {blind,wells}@daimi.au.dk

Abstract

This paper contains a design proposal for facilities for doing batch simulations in Design/CPN. These facilities can be used for running a group of simulations without user interaction. We discuss what kind of functionality is needed to run batch simulations. We also give proposals on how to implement this functionality in Design/CPN. To illustrate the use of the batch scripting facilities we present examples of how batch simulations can be defined.

Keywords. Coloured Petri Nets, Design/CPN, batch simulations.

1 Introduction

It is often necessary to run many simulations in order to obtain satisfactory or necessary results. There are many situations in which a series of simulations of Coloured Petri Nets (CP-nets or CPNs)[2–4] need to be run in Design/CPN[1], and it is not always necessary for a user to be present while the simulations are executed. Several such situations are:

- Calibration - tuning parameters in the CP-net so that simulation results resemble results from the modelled system.
- Performance analysis and/or data collection - analysing results generated by different start parameters.
- Sensitivity analysis - discovering which parameters have the most significant impact on simulation results.

Design/CPN lacks support for running a group or *batch* of simulations without user interaction between simulations. This paper describes ideas for the design of batch facilities which could solve this problem. The batch facilities will largely consist of a number of *primitives* that can be combined in *scripts* for defining a batch of simulations. Batch scripts will simply be ML code which executes a series of ML functions, also referred to as primitives. The batch facilities are currently being implemented. With the batch facilities it will be easy to specify and run a batch of simulations. It will be possible to change model parameters between simulations and specify where output should be saved. Furthermore, using the batch facilities will save time when running several simulations because the simulations will be automatically run one after the other without having to wait for a user to explicitly start each simulation.

The structure of this paper is as follows. Section 2 describes the basic design proposal for the batch facilities in terms of how primitives and scripts can be used. Section 3 describes some of the changes that need to be made in existing primitives so that they can be used to define a batch run. Section 4 discusses user scripts, which are scripts that can be totally defined, and therefore customised, by a user. Finally, Sect. 6 presents a design proposal for a standard script which allows a user to add some extra functionality to a partially-predetermined batch script.

2 Basic Design Proposal

One of the simplest forms of batch simulations is the execution of a constant number of simulations. Figure 1 contains a script that can be used to execute a given number of simple simulations. The primitive simulate will run a simulation, and the primitive init_state initialises the state of the CP-net. Evaluating the expression runBatch(10) will execute 10 simulations, and the state of the CP-net will be initialised before each simulation is started.

```
fun runBatch (noOfSimulations) =
    if noOfSimulations > 0 then
        (init_state();
         simulate();
         runBatch (noOfSimulations−1))
    else ();

runBatch 10;
```

Figure 1: A simple batch script.

2.1 Primitives

Running a batch of simulations usually requires more than just executing a group of simulations. Users will use the batch facilities for different purposes which means it should be possible to customise a batch run. It has to be possible for a user to specify actions, details, or changes to be made for each individual simulation. The central idea of this design proposal is to ensure that a set of essential primitives are available for defining a batch run. The following list is a set of actions that are likely to be useful when specifying a batch run. For each action, it is noted whether or not a primitive for executing the action currently exists in Design/CPN.

1. **Update reference variables.** A reference variable can be used for many different purposes in a CP-net: as a parameter for a function, as an initial marking, to identify an input file, to model a characteristic of a part of the modelled system, e.g. the speed of a CPU, etc. The value of reference variables can be changed anywhere which means that reference variables will be particularly useful when running batch simulations. *Primitive:* the ML assignment operator := can be used to change the value of a reference variable.

2. **Manipulate input files.** Files are often used to provide input for a simulation. *Primitive:* there are already primitives for opening and closing files by means of standard ML functions.

3. **Initialise the state of the CP-net.** *Primitive:* the function init_state changes the state of a CP-net to the initial marking. For timed CP-nets, the function init_time resets the model clock to the initial time.

4. **Initialise data collectors.** This is relevant only when running batch simulations in the Design/CPN Performance Tool [6, 7]. *Primitive:* the function initAllDataCollectors initialises all data collectors.

5. **Set and read simulation options.** For example, simulation stop criteria. *Primitive:* there are currently no primitives for setting or reading options, but it will be easy to add such primitives. Simulation options are currently set using dialog boxes.

6. **Run the simulation.** *Primitive:* the function simulate[1] will execute a simulation.

7. **Gather results.** A user may want to collect the following types of information after a simulation finishes: markings, values of reference variables, performance reports, simulation options. *Primitive:* files can be opened, updated, and closed by users. Reference variables can be dereferenced, and the values can be converted to strings which can be saved in files. The function savePerformanceReport will save a performance report in a file when simulating in the performance tool.

8. **Stop batch run.** It is necessary to be able to stop a batch run. Stopping a batch could depend on different types of criteria, e.g. number of simulations run, the value of a reference variable, the number of steps taken, the output of the previous simulation, etc. *Primitive:* users can define functions which examine different criteria, and which determine whether the batch should stop based on the status of the criteria. The functions step and time can be used to examine how many steps have been taken and the current model time.

[1] The primitive simulate is just an alias for the function sac_step which is used to execute a simulation in Design/CPN.

If the batch facilities contain primitives for executing these 8 actions, then a wide variety of batch simulations can be defined. Most of the actions mentioned here are already supported in the version of Design/CPN that includes the Design/CPN Performance Tool. Some of the primitives that exist will be modified slightly in the future, and other primitives need to be created. Section 3 discusses the modifications that will be made.

2.2 Scripts

The aforementioned actions can be combined in different ways to create different batch runs. For example, using varying combinations of these actions in runBatch in Fig. 1 would result in different forms of batch runs. We propose using ML functions or expressions to specify exactly which actions should be executed for a batch of simulations. Such a function or expression will be called a *script*. Again, Fig. 1 is an example of a simple script. The reason for selecting ML as the scripting language is that ML is used throughout Design/CPN.

Exactly how these scripts should be defined and executed is an important issue. In order to make the batch facilities as general as possible, it must be possible for a user to specify precisely which actions should be executed and when they should be executed during a batch run. For this reason, the facilities will support *user scripts* with which a user can totally define a batch run using the available primitives. A user script will consist of arbitrary ML code. Section 4 presents examples of user scripts.

The building blocks of scripts are primitives. As the name indicates these are low-level functions. Certain actions, e.g. updating variables, will often be repeated in different batch runs. In order to make the batch facilities easier to use, high-level functions for performing often used functionality will be introduced. These high-level functions are discussed in Sect. 5.

While user scripts may be useful for defining special or non-standard batch runs, they may be difficult to define for people with limited ML experience. It is also reasonable to expect that many users will define similar batches. For example, many batch runs will probably consist of changing simple variable values, running simulations and gathering relevant results. Therefore, the batch facilities will eventually include support for running standard batch runs or *standard scripts*. A standard script will often be parameterised, offering some predetermined functionality, but it will still be possible for a user to specify certain actions to be executed during the batch run. Examples of a standard script will be discussed in Sect. 6.

The idea for supporting both user scripts and standard scripts was inspired by the Design/CPN Occurrence Graph Tool (OG-Tool, [5]). In the OG-Tool queries can be made concerning the details of occurrence graphs. There are a number of standard queries which are either totally predefined or parameterised. For the parameterised queries a user needs only to specify relatively simple parameter values. These two types of queries are very easy to use. Finally, it is possible for a user to totally define his own queries. User-defined queries can be used to investigate characteristics that are relevant only for a given net. For example, a user query could be used to discover if a token with a certain colour was ever found on a given place. By supporting these three different types of queries, the query facility in the OG-tool is both user friendly and general. The batch facilities will support user scripts and standard scripts in an attempt to achieve the same balance between user friendliness and generality.

3 Basic Modifications

Most of the primitives that were mentioned in Sect. 2.1 can be used as they are. However, there are some primitives that will be modified in order to make them more useful within the batch facilities. One of the actions that will be used frequently is initialising the state of the CP-net between simulations. Section 3.1 discusses the implications of and problems concerning initialising the state of a CP-net. Section 3.2 contains suggestions for improving the functionality of the simulate primitive which executes a simulation. Section 3.3 discusses what kind of primitives need to be created for setting and reading options.

Before discussing the modifications that will be made to primitives, it should be noted that slight modifications will also have to be made to the user interface of the simulator for running batch simulations. The user interface of Design/CPN provides feedback about the status of a simulation to a user. This is done by means of dialog boxes, updating the status bar, beeps, etc. In particular, a dialog box is opened when a simulation stops, and this dialog box must be closed by user interaction before a new simulation can start. Since the purpose of batch simulations is to execute several simulations without user interaction, the batch facilities must include a primitive for disabling the relevant parts of the user interface when running batch simulations. The feedback that is provided by the user interface could instead be redirected to a log file.

3.1 Initialise State Function

A user will have to decide whether or not the state of the CP-net should be initialised between simulations in a batch of simulations. Section 3.1.1 discusses the difference between initialising the state and not initialising the state of the CP-net between simulations. Item 1 in the list of essential actions (Sect. 2.1) mentions that variables may be changed between simulations. It should be possible to influence the initial marking of a CP-net by updating reference variables. Section 3.1.2 discusses why the function that calculates the initial marking will need to be modified in order to make this feasible.

3.1.1 To Initialise, or Not

Whether or not the state of a CP-net is initialised between simulations will have a significant impact on the meaning of the batch simulations. In particular, it will influence the independence of the individual simulations from each other.

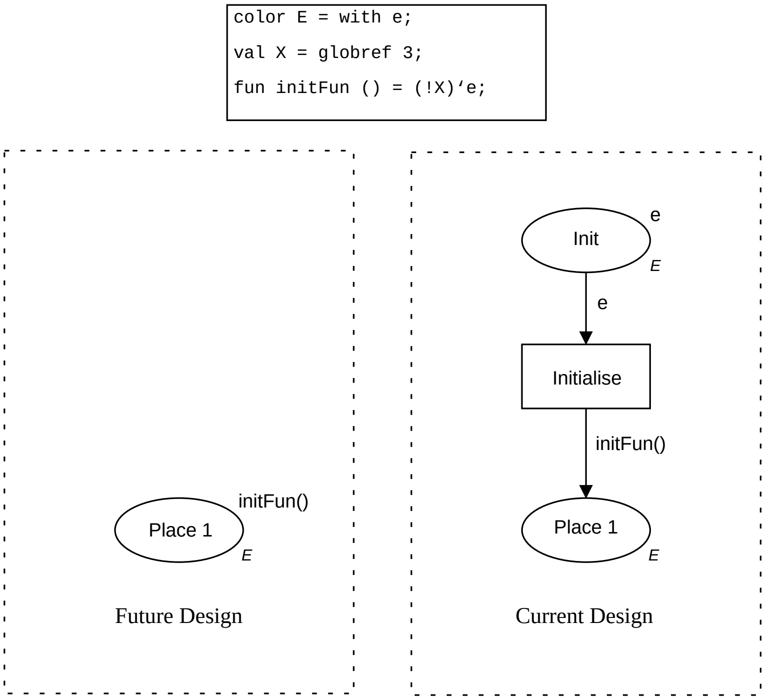
If the state of the net is initialised between simulations, then each simulation in a batch run is a new simulation. In this case all simulations will be independent from the others. Every simulation will start from an initial marking where $step = 0$. This type of batch run can be used for a variety of different purposes. By initialising the state – and possibly setting the random seed of the simulator (see Sect. 3.3) – it is possible to use different start parameters or different input sources and compare the performance of the system for different start conditions. It is also possible to run several independent simulations for one given set of parameters, and then study variations in the performance of the system for the different simulations. This last option is useful when the behaviour of the system is non-deterministic.

The other alternative is to choose not to initialise the state between simulations. If no state initialisation is undertaken, the batch run itself is a single simulation that is divided into sub-simulations. The outcome of each simulation is dependent on the simulations that preceeded it. This type of batch run can be used to collect data and differentiate between data from different intervals of one simulation.

3.1.2 Calculating the Initial State

We have previously mentioned that it should be possible for users to change the values of reference variables between simulations. These variables can be used, for example, as parameters for functions, in arc inscriptions, and in initial markings for places. The idea is that after changing the value of the variable before a simulation, the new value of the variable should always be used throughout the simulation. Unfortunately, the current implementation of init_state does not use the changed variable values. This section will describe this area in the current design of the state initialisation primitive in Design/CPN, and it will propose an alternative design for the primitive.

The following situation will be used to illustrate the difference between the current design of the state initialisation facility and our proposal. The left-hand side of Fig. 2 illustrates how we would like to be able to indicate the initial markings for a place. Assume that place Place 1 should have an initial marking that is dependent on the reference variable *X*. The initial marking of Place 1 could be calculated using the function initFun which dereferences the reference variable *X*.



```
color E = with e;
val X = globref 3;
fun initFun () = (!X)‘e;
```

Figure 2: Calculating the initial markings.

In the current implementation of Design/CPN, the initial marking of the CP-net is calculated once and for all during the switch from the editor to the simulator. The marking for each place is calculated during the switch, and the marking is saved for possible later use. If a user chooses to return to the initial state while in the simulator, then the markings that were previously calculated are simply reinstalled for each place instead of being recalculated. Assume that the initial marking of Place 1 was given directly by the inscription initFun() (as shown on the left-hand side of Fig. 2). Assigning a new value to *X* before returning to the initial state will *not* change the initial marking of the place. Installing pre-calculated markings can save execution time, but it means that it is difficult to use different initial markings.

If one is interested in using different initial markings, then there are ways to work around the problem in the current implementation. The right-hand side of Fig. 2 illustrates one workaround: using transitions to compute pseudo-initial markings that can differ from one simulation to another. Instead of declaring an initial marking for the place Place 1, it is possible to add extra nodes to the net for initialisation purposes. In this case Place 1 will have an empty initial marking, and when the transition Initialise occurs, the desired tokens will be put on the places (see the arc from Initialise to Place 1). Thus, the pseudo-initial marking is created. Now, if the value of *X* is changed before initialising the state, then the pseudo-initial marking of Place 1 will reflect the new value of *X*. With this workaround it is possible to define different "initial" markings without user interaction.

At the moment it is unclear how much extra time would be needed to recalculate the initial marking of a net when using the new design proposal. It may be the case that recalculating the initial marking during state initialisation requires significantly more time than simple reinstallation of pre-calculated values. It may also be possible to combine the two strategies and recalculate only certain parts of the initial marking when returning to the initial state. With this alternative the "constant" initial markings of places could be calculated once and for all during the switch, while only the markings that could change from one simulation to another are recalculated when initialising the state of the net.

An alternative to ensuring that the desired initial marking is recalculated when the state of a CP-net is initialised, is to make adjustments to the marking after the state has been initialised. In the simulator, there is a dialog box that can be used to change the marking of a place. The functionality of this dialog box could be captured in a primitive, which could then be used to modify markings. The primitive changeMarking would take a place name, and a marking as parameters, and it would replace the old marking of the given place with the new marking. An advantage of having such a primitive would be that this primitive could be used at any point during a simulation and not just when initialising the state of the net.

3.2 Simulate

We now turn to the primitive for executing an individual simulation in a batch run. An individual simulation of a batch run will be a slightly modified version of a simulation that is currently started by invoking *Automatic Run* in the simulator. Section 3.2.1 contains design ideas for new stop criteria. Section 3.2.2 discusses ideas for better exception reporting during simulation.

3.2.1 Stop Criteria

Many simulations can run for many steps before there are no more enabled transitions. Some CP-nets give rise to infinite firing sequences which means that other simulations could continue running forever. Therefore, the batch facilities need some mechanism for specifying when each individual simulation is to stop. It is possible to allow a simulation to run until there are no more enabled transitions. In Design/CPN it is also possible to set some so-called stop criteria. The stop criteria that are currently supported are based either on the number of steps that have occurred or on the model time. These stop criteria are quite useful, but they are somewhat limited. In this section we propose designs for new stop criteria. These stop criteria would be incorporated in the standard simulator and would, therefore, also be available when using the batch facilities. In other words, these new stop criteria would not just be available when running batch simulations.

User Defined The current stop criteria are net-independent, i.e. the stop criteria can be used for every CP-net. It is currently impossible to define stop criteria that are dependent on how a net is defined (net-dependent), e.g. stop criteria based on markings or binding elements. In order to make the stop-criteria facilities net-dependent, we propose adding a function for setting a stop flag. When this function is invoked the simulator will stop after the current step. The function for setting the stop flag can be used anywhere in a model. This means that a simulation can be stopped at any time. Some examples are:

1. When a certain transition occurs. In this case the stop function could be written in the corresponding code segment.

2. When a specific marking has been reached. This will currently only be possible when in the performance tool. A data collector can be used to inspect the marking of the net, and could call the stop function if the appropriate marking is present.

Regarding item 2 there are two possible drawbacks to using the facilities of the performance tool only for stopping a simulation using the stop flag. The first drawback is that more time will be needed to make a switch to the simulator because extra structures need to be created during the switch for accessing markings and binding elements. The other drawback is that more time will be needed to run a simulation if the entire net marking is extracted after each step of the simulation. Preliminary tests show that simulation time increases by approximately 25% when the net marking is calculated after each simulation step (for more details see Sect. 8 in [7]).

Real Time and CPU Time The stop criteria that are currently supported in the simulator are dependent only on the state of a simulation of a CP-net, i.e. either the model time or the number of steps that have occurred. In some cases it may be useful to be able to stop a simulation after it has run for a certain amount of real time. For example, a user may want to stop a simulation 30 minutes after it had started. Similarly it might be useful to be able to stop a simulation after it has had a certain amount of CPU time. Using CPU time instead of real time would ensure that the simulation had actually run for the desired amount of time. It is likely that a simulation would be only one of many processes running on a CPU, therefore real time would only indicate how long ago the simulation started, and not actually how long it had run. Supporting stop criteria depending on real and CPU time could be advantageous.

3.2.2 Exception Reporting

Another aspect of the simulator that needs to be improved is exception reporting. Currently, a user is only notified that an exception has been raised causing a simulation to stop. No indication is given about what type of exception has been raised. There is also no feedback about where an exception has been raised, e.g. when evaluating a code segment or a guard. The performance tool provides some support for exception reporting (see Sect. 9 in [6]). However, this exception reporting is used only when an exception has been raised while executing performance tool specific functions, e.g. evaluating observation functions, opening observation log files, updating statistical variables within data collectors, etc.

Exception reporting facilities would be useful when running single simulations, and they would be especially useful when running batch simulations. If a batch simulation is started, and some or all of the simulations stop because of exceptions, then a user will want to know why the exceptions were raised. If detailed feedback is not provided when exceptions are raised during batch simulations, then it may be difficult to find the source of the problem for each individual simulation.

3.3 Set and Read Options

There are several different types of simulation options. Stop criteria are simulation options. It is also possible to indicate whether step information or bindings should be saved in case a simulation report will be saved at the end of a simulation. Selecting a fast simulation or a fair simulation is also done by setting a simulation option. New primitives will need to be created for setting and reading simulation options.

There may be situations in which a user will want to set simulation options differently for each simulation in a batch run. For example, given the results of one simulation, it may be clear that the next simulation should run for more steps, in which case it would be useful to be able to set new stop criteria during a batch simulation. Another example of a simulation option that could be changed is the seed for the random number generator of the simulator. When several simulations are performed using different input, e.g. from a file, it could be useful to be able to minimise random behaviour by ensuring that each simulation uses the same seed when running a batch.

Currently, all simulation options are set using dialog boxes. There are no primitives for either reading or setting simulation options. Simulation options are saved using reference variables. Reading and setting a simulation option is simply a matter of dereferencing or reassigning the appropriate reference variable. A user should not have direct access to the reference variables in question, therefore data encapsulation methods should be used. To this end simple primitives can be made for reading and writing the relevant simulation options. It is possible to define meaningful batch simulations without primitives for reading and setting options, but the batch facilities would be improved with such primitives.

4 User Scripts

In this section we illustrate how the batch scripting facilities can be used to create user scripts. User scripts are defined using the primitives discussed in Sect. 2.1. In addition to these primitives, it is also possible to define a script using any other user-defined ML function, ML primitive, or any ML function that is available in the simulator of Design/CPN.

4.1 Updating Several Variables

In this section we will give a simple example of a batch that runs simulations with different values of two reference variables. Batch simulations may be used to explore the impact of different combinations of parameter values on simulation results. Therefore, it will often be the case that the values of one or more reference variables in the CP-net will need to be changed before starting a new simulation. An example of such a situation is described in Example 1:

> **Example 1** *Assume that* x *and* y *are parameters in a CP-net, and that they are declared in the global declaration box as global reference variables. A simulation of a model should be performed for every integer value of the variable* x *within the interval* $[3; 4]$ *and for every even value of the variable* y *within the interval* $[2; 6]$*. This means that a simulation should be run for each of the following combinations of parameters* (x, y)*:* $(3, 2), (3, 4), (3, 6), (4, 2), (4, 4),$ *and* $(4, 6)$.

Figure 3 illustrates how Example 1 *could* be implemented using the primitives that are available in the batch facilities. In this user script the two reference variables x and y are updated between the simulations. A simulation is run for each combination of the parameter values that have been given. The model will possibly behave differently for each combination of the variables.

4.2 Data Collection

It is likely that the batch facilities will often be used to perform batches of simulations in the Design/CPN Performance Tool. The performance tool is used to collect and output data during simulations. Observation log files are generated automatically during a simulation, but a user needs to explicitly ask for a performance report to be generated. Figure 4 illustrates a user script for running simulations in the performance tool.

The parameter *i* is used to count how many simulations have been run. Before each simulation is started, the state of the CP-net is initialised. When initialising the state of a CP-net in the performance tool, any existing data collectors are deleted, and the performance functions are reevaluated in order to install new data collectors. When a simulation is run, data is collected

```
fun varScript(i) =
    if i <= 4 then
        let
            fun runLocalScript(j) =
                if j <= 6 then
                    (x:= i;      (* x is a parameter in the model *)
                     y:= j;      (* y is a parameter in the model *)
                     init_state();
                     simulate(); (* Run a simulation using the newly assigned values of x and y *)
                     runLocalScript(j+2))
                else ()
        in
            runLocalScript 2;
            varScript (i + 1)
        end
    else ();

varScript 3;
```

Figure 3: A batch script changing two reference variables.

```
fun dataCollectionScript(i:int, n:int) =
    if i <= n then
        (init_state();
         simulate();
         savePerformanceReport("PerfReportBatch"^makestring(i)^".txt");
         (dataCollectionScript(i+1, n))
    else ();

dataCollectionScript(1, 10);
```

Figure 4: A batch script for collecting data in the performance tool.

as usual, and statistical variables and observation log files are updated. After a simulation has stopped the primitive savePerformanceReport is used to save a performance report. Note that the value of *i* is incorporated into the name of the file containing the performance report. This is done to create unique file names that indicate from which simulation the performance report came. Before starting the next simulation the counter is incremented, and if *n* simulations have been run the batch stops. Evaluating dataCollectionScript(1,10) will execute ten simulations in the performance tool, and ten performance reports with unique names will be saved during the batch.

5 High-level Functions

When users write user scripts for running batches some functionality is implemented over and over again, e.g. changing values of variables between simulations. It would be useful if the batch facilities provided functions for standard actions. This section discusses some functions that will be provided in the batch scripting facilities to provide auxiliary functionality, and it will give a few examples of how the functions can be used in user scripts. Section 5.1 describes high-level facilities for updating variables. Section 5.2 discusses batch status files for giving an overview of the batch simulation by summing up some of the main results of the batch. Section 5.3 describes some of the results that can be gathered during a batch run, and it describes how to manage the many files that may be generated. Finally, Sect. 5.4 discusses support for standardised batch stop criteria.

5.1 High-level Support for Updating Variables

Figure 3 in Sect. 4.1 illustrates how to write a batch script for updating two variables with some specific values. When the number of different variables increases it may become a laborious task to write the code for considering all combinations of variables. We want to provide high-level support for indicating which variables should be updated before a simulation is started.

Supporting this kind of value[2] specification would be useful, and it would increase the user-friendliness of the batch facilities. When considering how general the specification of values of variables should be, we discussed several different possibilities. The first idea was that for each variable the user should be able to specify a minimum value, a maximum value, and a constant step value as described in Example 1 in Sect. 4.

We realised that one may not be interested in increasing the value by a constant for each simulation. Instead, one might want to specify a function that would return the step increase. By using such a step-increase function one could, for example, specify an exponential increase in the value of a variable between simulations. This step-increase function would make the value-specification facility more flexible.

The value-specification facility will include the following functionality. The user can specify which variables to change and how to change them, as discussed in the last proposal above, by means of the initialisation function initVarUpdate shown in Fig. 5. The parameter for this function is a list of records. Each record specifies how to update one reference variable: it contains the name of the reference variable (*var_name*), the minimum value (*min*), the maximum value (*max*), and the step-increase function (*inc*).

After defining which variables should be updated, it is time to start simulating. After executing a single simulation the variables need to be updated. This will be done using the function varUpdate. By invoking the function varUpdate all of the variables are automatically updated properly. After each simulation a user ought to be able to check whether all combinations of values in the variables have been used. This is done by means of the function stopVarUpdate

[2] In this section the term *values* refers to values of type integer or real, though, some of the discussion may be relevant for other types, too.

```
fun varScript() =
    let
        fun runLocalScript() =
            if not (stopVarUpdate()) then
                (init_state();
                 simulate();
                 varUpdate();
                 runLocalScript())
            else ()
    in
        initVarUpdate([{var_name=x, min=3, max=4, inc=fn () => 1},
                       {var_name=y, min=2, max=6, inc=fn () => 2}]);
        runLocalScript()
    end;

varScript ();
```

Figure 5: A batch script changing two reference variables using high-level primitives.

which returns `true` when no more variable updates need to be performed.

As an alternative to specifying the maximum value of a variable, we also considered supporting the use of a predicate function that could be used to indicate when a value no longer should be increased. However, to keep the design of this value specification facility simple, this idea was discarded. This facility is designed to be *easy to use* and to support *simple* variable updates. Requiring a user to define a predicate function for determining when to stop increasing a value would contradict these two goals. Furthermore, this facility of specifying intervals of values is particularly useful for users with limited experience in programming CPN ML. Therefore, it is important to keep the interface simple.

5.2 Batch Status File

After having performed a batch of simulations it may be difficult for a user to maintain a general overview of all the different simulations: which simulations have stopped with no more enabled transitions, which exceptions have been raised, and information like this. One way to preserve the overview is to save the status of each simulation in a so-called *batch status file*. The purpose of the batch status file is to present the most important information from each individual simulation.

Some of the questions that arise when designing support for a batch status file are the following. How detailed should the information in this batch status file be? Should the contents of the file be totally predetermined by the batch facilities, or should it be possible for the user to influence the contents of the file?

If the contents of the batch status file are totally predetermined by the batch facilities, then one of the advantages is that it will be easy to compare different batch status files. However, the information that will be contained in such a file would be restricted to information like:

- Simulation number.
- Number of simulated steps.
- Model time.
- Why the simulation stopped:
    - No more enabled transitions.
    - Stop criteria fulfilled.
    - Exception raised during the simulation.
- How long the simulation took:
    - In real time.
    - In CPU time.
- Use of memory.

In some cases this kind of model-independent information may not be sufficient for the user. The user may want some user-defined information to be written in the batch status file, e.g. the value of a particular variable. The reason might be that he would like a single file containing the main results of the batch of simulations. We think that it will be necessary to include the possibility to make the batch status file user definable. If we disallowed this possibility then the user would have to make the files himself – and therefore introduce yet another status file. If we allow the user to specify some text to be included in the batch status file, then we could omit the need for having more than one status file.

```
fun statusFileScript n =
    let
        fun runLocalScript i =
            if i > 0 then
                (simulate();
                 updateBatchStatusFile("x has the value: "^(makestring (!x)));
                 runLocalScript(i−1))
            else ()
    in
        initBatchStatusFile("filename.txt");
        runLocalScript(n)
    end;

statusFileScript 10;
```

Figure 6: A batch script updating the batch status file.

Figure 6 illustrates how a batch status file could be used. First of all the batch status file needs to be created. This can be done using the function initBatchStatusFile as shown in Fig. 6.

When the batch status file should be updated, the function updateBatchStatusFile must be invoked. The parameter to the function is the user-defined text that the user wants to be added to the standard contents of the file which are described above.

A user may not want to have all of the standard information in the batch status file. Other primitives could be designed so that a user could control exactly which information should be included. Separate primitives could be made for each of the items of information that were mentioned earlier. Then a user could choose exactly which information was relevant and use the appropriate primitives for including it in the batch status file. For example, a function like updateBatchStatusFileModelTime would add only the model time to the batch status file. The advantages of using such primitives is that a user can easily save relevant information, but he does not have to manage the details of opening, closing, and updating the status file himself.

5.3 Gathering Results

It is very likely that users will want to collect data and results from batch simulations. This means that the data and results need to be stored – usually in files. The batch status file can be used to collect some standard information concerning the status of simulations, and it can also include some user defined information. However, a user may want to save simulation results or data separately from simulation status information. For example, if a batch consists of twenty different simulations and five variables are changed in each simulation, then it would be desirable to remember the exact values for each variable within each simulation. In such a situation, it may be valuable for a user to save relevant information about each simulation after the simulation ends. Batch simulations will also often be used to compare and evaluate results from several different simulations. Again, files are needed for storing the results. When simulating in the performance tool, it is possible that many observation files will be created in addition to performance reports.

Other Results Here we will briefly mention other types of results or information that may be useful to save after a simulation is finished. The first item is a *simulation report*. Options can be set such that step information and bindings are saved. These can then be saved in yet another file. A primitive will need to be created for doing this, since the only way to save a report in the current implementation is via a dialog box. Another primitive will be needed for clearing the simulation report.

The results and information that we have discussed previously all have one thing in common: it is data that is extracted from either the CP-net or from the the simulator. In other words, the data represents a small part of the information that is available in the simulator. When running single simulations, the entire state of the simulator is often saved as an ML image. This state image can then be used later to continue a simulation or to avoid a lengthy switch from the editor to the simulator. It is conceivable that there will be occasional need for saving the state of the simulator during batch runs. A new primitive will be created for this purpose.

File Management It should now be obvious that it is quite possible that many files will be generated during a batch run. Remembering which files were generated by which simulation may be a real problem. In order to alleviate this problem the batch facilities will include some high-level support for file management.

One way to differentiate between files is to ensure that the files have names that indicate from which simulation they were generated. This could be done by using a string value, or label, which contains information that is unique for each separate simulation. For example, in Example 1 in Sect. 4.1 the values of x and y could be used to make unique labels. It should be possible to provide a function for creating labels given variable names. For example if $x = 3$ and $y = 2$, then the function could return the label “x3_y2”. This type of label could certainly be used to define unique file names for different simulations. If one file is used to gather results from all of the simulations, then the label could also be used to separate results in that file.

Another alternative for managing files, is to save results in a new directory for each new simulation. Again, a label could be used to give each directory a unique, mnemonic name. The batch facilities could also include a function which automatically creates directories with simple names, together with a function that returns the name of the current batch directory. Figure 7 illustrates the use of primitives for creating new directories.

```
fun dirScript i =
    if i > 0 then
         (createNewDirectory();
          init_state();
          simulate();
          savePerformanceReport(newDirectoryName()^"PerfReport.txt");
          dirScript (i−1))
    else ();

dirScript 10;
```

Figure 7: A batch script creating new directories.

The function createNewDirectory would create a new directory name with a name that indicates when the directory was made. For example the directory name “Batch1_Sim1” would indicate that the directory was created for the first simulation in the first batch of simulations. After the simulation is finished, the performance report is saved in this directory by creating a file name using the function newDirectoryName to access the name of the most recently created directory. Additionally, if the names of observation log files are also prefixed with the name of the new directory, then all of the performance related output from one simulation can be found in the same directory.

5.4 Batch Stop Criteria

In this section we consider high-level functions for specifying when to stop a batch of simulations. Of course the user could define stop criteria himself, but it may be useful to provide functions for the criteria that are frequently used. In the following we will describe some of the most useful criteria for determining when to stop a batch of simulations.

All of the criteria that are currently used to stop simulations could also be used as batch stop criteria. In other words, batch stop criteria could also depend on the number of steps taken or the model time. The simplest form of batch stop criteria would be to stop the batch after a constant number of simulations have been run. We have introduced simulation stop criteria that depend on real or CPU time. An often recurring question when running simulations that run for a long time is: will the batch ever stop, or is it cycling forever? It would also be useful to have batch stop criteria that ensure that the entire batch will stop within a certain time limit. New functions will need to be created for setting and checking batch stop criteria.

In contrast, by giving the user the ability to write his own criteria for stopping a batch, we allow the number of simulations to depend on the results of previously performed simulations (within the same batch run). This gives the possibility of setting parameters in a CPN model, then simulating the model, and deciding whether another simulation should be run based on the results obtained.

6 Standard Scripts

As described in Sect. 5 some batch runs have a certain common structure. An example could be that many batches have a well-defined loop structure, possibly with a well-defined initialisation before starting a simulation and a well-defined way of reporting results after a simulation. When such standard skeletons for implementing batches are identified, it is possible to integrate these in the batch facilities by providing standard scripts implementing these skeletons. Depending on how standardised the batch run is, it may be necessary for the user to be able to slightly extend standard scripts, i.e. to add some functionality to the standard script. Standard scripts are not replacements for user scripts, instead they are alternatives to user scripts. Some standard scripts may be for very specific, but often used, situations, e.g. just updating variables before running a simulation. This idea of using standard scripts is similar to using standard queries in the OG-tool [5] of Design/CPN. In this section we will propose a standard script that will make it simple to run standard batches. In the future when applying the batch facilities in practice, we will surely identify other useful standard scripts.

The idea behind a general standard script is that many batches will contain the scenario illustrated in Fig. 8. First of all, the batch job is initialised. To prepare the simulation the user specifies some initialisation, followed by possibly initialising the state of the model before starting a simulation. Then simulations are executed until a stop criterion is satisfied. When the simulation stops the user may want to gather results, and finally possibly start a new simulation.

This standard batch script can be used in many different situations. It is so general that many different batches can be expressed by means of this standard batch. It should not be hard to see that the batch script in Fig. 5 using high-level primitives could easily be implemented by means of this standard script. To get the intended behaviour the user supplies functions defining the user-specified behaviour as parameters to the standard script. It should be easier for the user to use such a standard script than having to implement the entire script himself.

```
fun simpleStandardScript(initBatch, endBatch, stopCriteria, beforeSimulation, doInitState, afterSimulation) =
   let
      fun runLocalScript() =
         if not (stopCriteria()) then
            (beforeSimulation();
             if doInitState() then
                init_state()
             else ();
             simulate();
             afterSimulation();
             runLocalScript())
         else ()
   in
      initBatch();
      runLocalScript();
      endBatch();
   end;
```

Figure 8: Implementation of a general standard script.

7 Conclusion

In this document we have described the design of batch scripting facilities for running batch simulations in Design/CPN. We have described what changes need to be made in the Design/CPN tool, and what functions would be useful to have when writing batch scripts. Many of the primitives that we have discussed are already implemented. As soon as users know what primitives are available, they will be able to begin writing user scripts. The facilities will be improved when new primitives are created. We also discussed different levels of writing batch scripts, i.e. user-defined scripts and standard scripts.

An issue that we have not discussed is how the scripts should be defined and invoked. There are at least two different possibilities. First of all, the user could write the batch script in an auxiliary node and then use ML-evaluate to start the batch. The batch could also be run by starting Design/CPN from the command prompt with a batch option and a name of a file containing a batch script. Then Design/CPN would automatically run the batch script without the graphical interface. This would make it possible to use Design/CPN as an engine, and it would be easier to use for a user who is not familiar with the graphical interface of Design/CPN.

In Sect. 6 we discussed using standard scripts for easing the work when implementing batch scripts. Another option would be to use templates for batch scripts. A user could select a certain template, and then some code, possibly something like Fig. 8, could be added to an auxiliary box. Then the user could modify this code to get the intended behaviour of the batch. Template scripts would be both easy to use and fully customisable.

The design of the batch facilities was developed immediately after designing, implementing and using the Design/CPN Performance Tool. This may have influenced the design of the batch facilities in both positive and negative ways. On a positive note, we were very aware of how the batch facilities could be used to easily collect data from a variety of different simulations. Therefore, we were determined to include primitives that would ease the collection and output of data during batch simulations. On the other hand, we may have overlooked some important issues pertaining to simulating without using our data collection facilities in the performance tool. We are indeed aware that the present design is only based on needs that have been expressed by Design/CPN users. In the future we will look into other simulation tools that use batch facilities and consider relevant literature in the area of batch simulation. Additionally, we will also take into account comments and feedback from future users of the batch facilities.

To test the design of the batch facilities we made a few scripts using the primitives discussed in this paper. Preliminary tests have shown that the batch scripting facilities are well-suited for defining batch runs. From early experiments we think that this design provides very general and easy-to-use facilities for writing batch scripts. Additional experiments will indicate whether or not the design presented here will need to be modified.

References

[1] Design/CPN Online
Online: http://www.daimi.au.dk/designCPN/.

[2] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[3] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997. 2nd corrected printing.

[4] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use. Vol. 3, Practical Use*. Monographs in Theoretical Computer Science. Springer-Verlag, 1997.

[5] Kurt Jensen, Søren Christensen, and Lars M. Kristensen. *Design/CPN Occurrence Graph Manual*. Department of Computer Science, University of Aarhus, Denmark, 1996.
Online: http://www.daimi.au.dk/designCPN/man/.

[6] Bo Lindstrøm and Lisa Wells. *Design/CPN Performance Tool Manual*. Department of Computer Science, University of Aarhus, Denmark, 1999.

[7] Bo Lindstrøm and Lisa Wells. *Tool Support for Simulation Based Performance Analysis using Coloured Petri Nets*, 1999. Department of Computer Science, University of Aarhus, Denmark.

Design and Animation of Coloured Petri Nets Models for Traffic Signals

Angelo Perkusich[†], Luciana M. de Araújo[‡],
Roberta de S. Coelho[‡], Kyller C. Gorgônio[‡],
Erica de L. G. Ribeiro[‡] and Adriano J. P. Lemos[‡]
[†]Departamento de Engenharia Elétrica
[‡]Laboratório de Redes de Petri
Departamento de Sistemas e Computação
Universidade Federal da Paraíba
Caixa Postal 10105, 58109-970 - Campina Grande - PB - Brazil
perkusic@dee.ufpb.br

Abstract

This paper addresses the design and visualization of animated models for coordinated and decentralized discrete event control systems, e.g. manufacturing systems and traffic control systems. We introduce an approach to the design and animation of such controllers based on Coloured Petri nets models and the Design/CPN tool. The approach is illustrated by a simple traffic control network modeling.

Keywords: discrete event system control, Coloured Petri nets, model animation.

1 Introduction

The design of complex control systems such as Intelligent Transportation Systems [10] requires the coordination of decentralized controllers in different intersections of a traffic network. Also, the need for adaptive control for such systems is increasing in importance so that parameters of the decentralized controllers can be tuned according to the environmental needs, e.g. traffic flow. As a complex system the need for visual

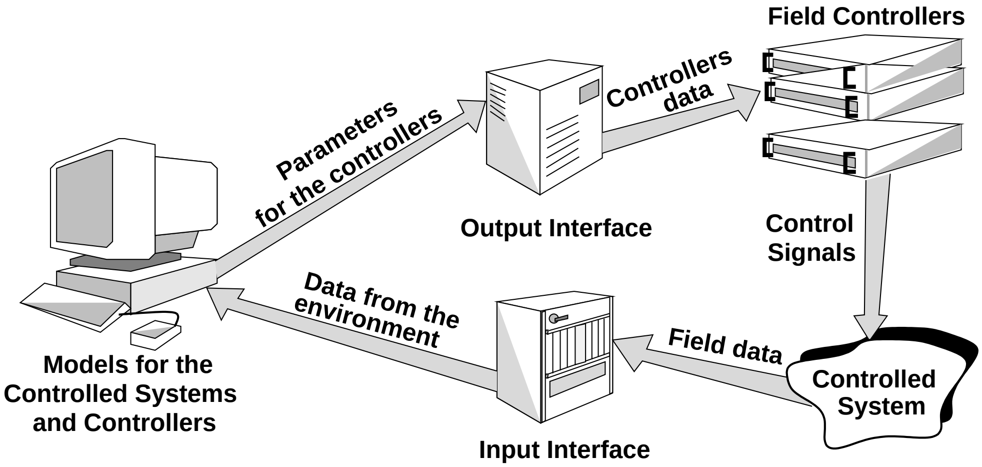


Figure 1: Adaptive control architecture

tools to help during the design phase for a better understanding of the domain problem are very important.

In this paper we introduce an approach to the design and animation of coordinated decentralized controllers for discrete event systems based on Coloured Petri (CPN) nets [3, 4, 2] models created using the Design/CPN tool [5].

The example used in this paper is a traffic signal control system as discussed in [8, 9, 1, 10, 13].

The paper is organized as follows: the architecture and the functionalities of an adaptive control for distributed discrete event systems is presented in Section 2. In Sections 3 and 4 we introduce basic concepts related to the coordinated control of traffic light control and modeling respectively. In Section 5 we present an approach to the animation of a CPN model constructed using the Design/CPN. In Section 6 we detail, based on a simple example the introduced approach. Finally, in Section 7 we present the conclusions of the paper.

2 Decentralized Adaptive Controllers

Discrete event control system are usually organized in a hierarchical structure [12, 11]. In the lowest level of the hierarchy are the local controllers performing local tasks, such as machines controllers in manufacturing systems and traffic light signal controllers. These controllers are usually decentralized and coordinated based on the parameters defined on the upper levels of a control hierarchy. Considering the case of traffic lights, sequences of green, red and yellow are repeated for each traffic lights. In the case of fixed length intersections, the coordination level defines the timing of the green signals for consecutive intersections along a desired path. Usually this timing considers only the arterial traffic, so that the timing for a green wave of lights is defined. In this case the major problem the optimization of the timing for the traffic lights in all intersections in the given path.

The block diagram for an adaptive decentralized control architecture is shown in Figure 1. The computer running the models for the controlled system (e.g. a traffic network) and the controllers (e.g. each traffic light controller) is interfaced with the controlled system by means of input and output interfaces. The input interface collects data from the field and sends data to the machine running the model. The parameters of the model that depends on this data are then updated so that new output parameters are defined for each controller in the field. Then, these parameters are sent to the field controllers through an output interface.

As shown in Figure 2 in the context of this paper we use a CPN model built using the Design/CPN tool [5]. Then, the occurrence graph is generated and the behavior specification for the system maybe defined, either using functions written in CPN-ML or defined using ASKCTL [7], a temporal logic used for model checking in the Design/CPN tool.

A file with the state information is created, so that the behavior of all field controllers parameters can be extracted from it. The data from the environment is stored in a file and transitions in the CPN model are properly parameterized with the information from this file in a similar way as discussed in [6].

3 Traffic Signal Systems

Nowadays a traffic signal may exist as a single isolated controller or may be part of a multi-signal traffic control system. These traffic control systems are comprised of interacting components such as signals, detectors, and communication infrastructure that are arranged in a manner which effectively and efficiently coordinate traffic flow along a corridor or throughout a network. Traffic engineers are continually refining the control strategies in an effort to allow for the safe and efficient movement of people and goods. Due to the prohibitive costs, the increase in the use of motor vehicle without a proportional increase in the infrastructure lead to many different problems, such that increase in fuel consumption and pollution. The sophistication level of traffic control has advanced beyond the exclusive use of time-of-day programs and local flow detection. Today, many research is devoted to real-time traffic adaptive control systems. These systems not only operate based on local detector demand but also make control decisions based on predicted future demand [13, 10].

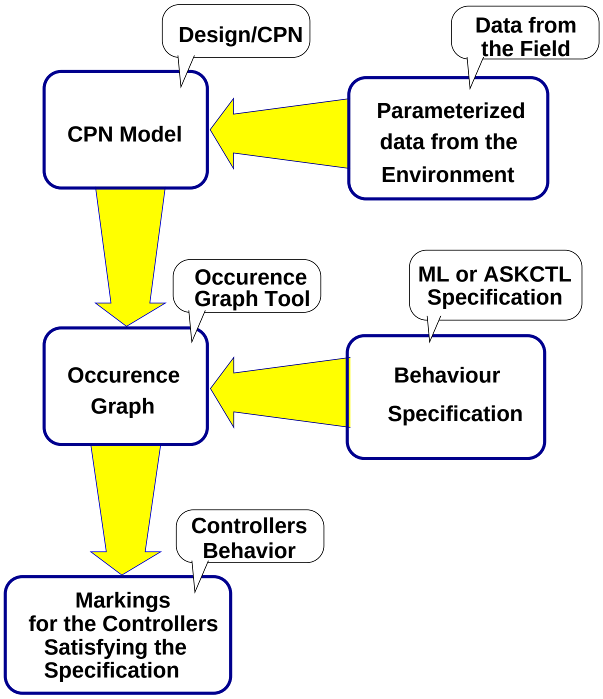

Design/CPN
Data from the Field
CPN Model
Parameterized data from the Environment
Occurence Graph Tool
ML or ASKCTL Specification
Occurence Graph
Behaviour Specification
Controllers Behavior
Markings for the Controllers Satisfying the Specification

Figure 2: Block diagram illustrating the life cycle for the field controllers behavior specification

In arterial and network systems, traffic is thought of as moving groups, namely platoons. Typically, signals located at the boundary of the system group vehicles during the red phase and then discharge this platoon under green. The traffic control system coordinates the cycles for traffic signals at the intersections in order to maintain the traffic flow. Such a control system consists of coordinated-actuated controllers that are distributed in nature. Also, the control systems must provide actuated control capabilities that are intended to improve the functioning of a traffic control system. This is accomplished by allowing individual intersections to respond to varying traffic demands while also maintaining coordination from one intersection to another.

To improve the current practice of using coordinated-actuated controllers for traffic signal control systems, advanced traffic control systems are being developed. The reader can refer to Figure 1 in what follows. The basic methodology for these adaptive systems is to have a model for the traffic network, an input interface to gather data from the intersection traffic flow, and an output interface to the various intersection controllers. The model is then used together with up to date information from the traffic flow in the network and then finds out a better control policy for a given behavior specification, as for example optimize green wave on throughput.

There are different levels of sophistication in traffic control systems. The most

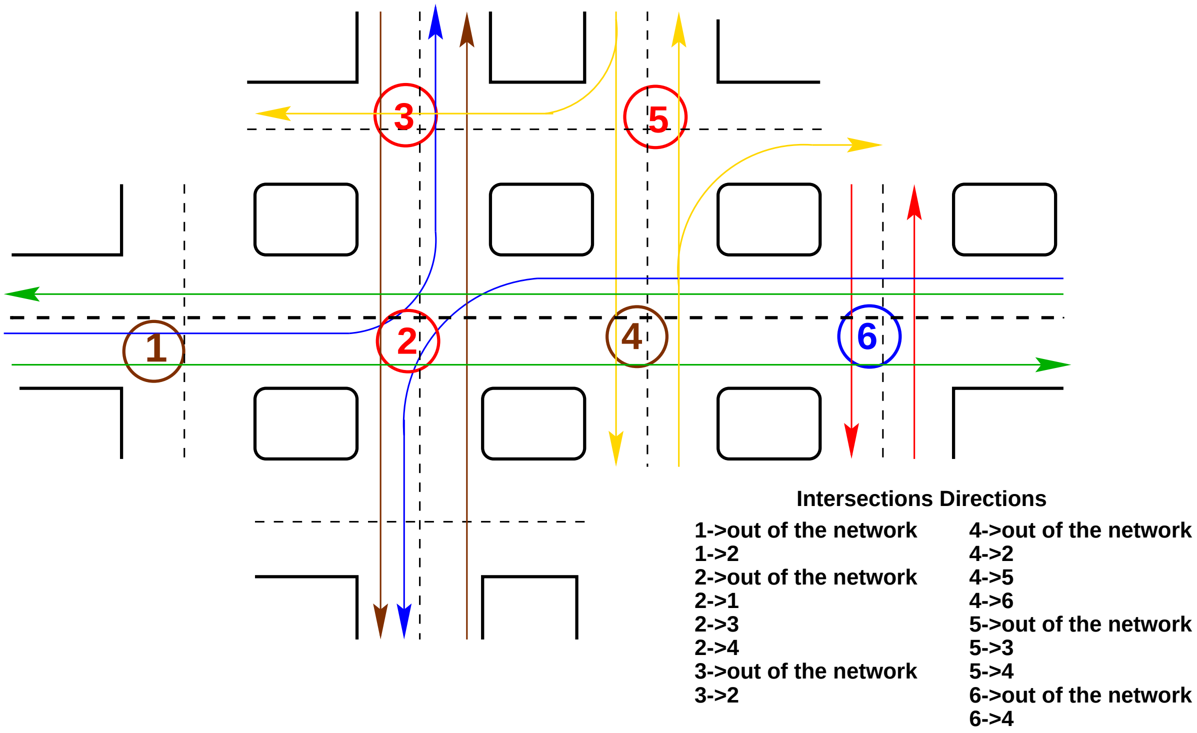


Figure 3: Example of a traffic network

simple is known as Webster's Method. It indicates how to divide the amount of green time in a periodic cycle between opposing flows of traffic at an intersection. Optimal sequences of green time are determined using predicted arrivals of vehicles or blocks. For more details the interested reader may refer to [8, 9, 1, 10, 13].

4 Design of a Traffic Control System

The problem of the design of a traffic control system consists of elaborating a model of a traffic network that allows the control of the traffic lights, guaranteeing that conflicting routes (in one particular intersection) do not have green signs at the same time, as well as modeling the dynamics of the stream of vehicles (or blocks) through the intersection. Also, a block of cars always stop as minimum as possible when in arterial traffic. The vehicles passing through the network may come from each one of the existing intersections or they may come from another traffic network. Thus, it is necessary to represent the interaction of this network with the environment, modeling the input of vehicles by the edges of the network and the output of vehicles after crossing it.

An example of a network composed of six intersections is shown in Figure 3. The

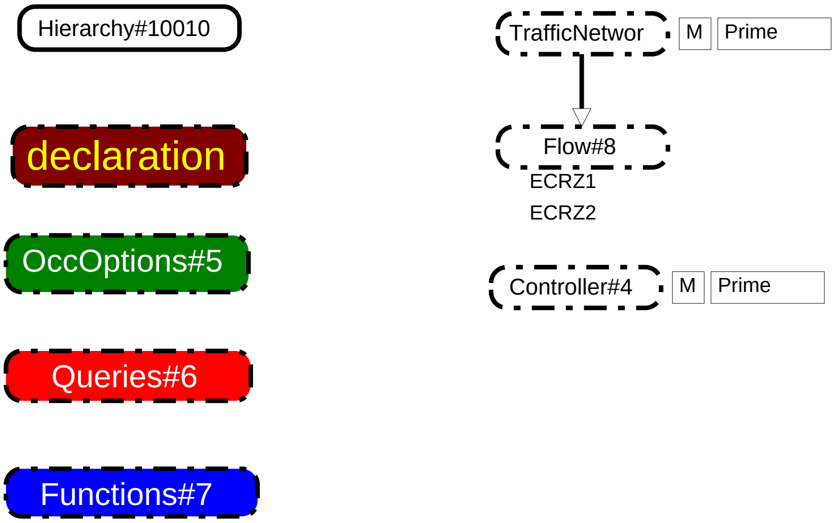


Figure 4: The hierarchy page for the model

routes are represented by directed arcs. We assume that the controllers for each intersection have the same structure.

A Hierarchical Coloured Petri net model was built and the hierarchy page is shown in Figure 4. For this model three pages were built: the traffic network page, the control page, and the flow page. Also, it is shown a page were the CPN-ML functions, queries, and occurrence graph options for the model are defined.

The traffic network page models the network connecting all the intersections. The control page models the controllers implementing the sequence of allowed phases for each intersection and the timing for the phases. The flow page represents the movement of blocks through the intersection including timing parameters for the phases, as well as how long a block takes to move through an intersection.

In what follows we present the declaration node including the color sets and functions and some details related to each one of the pages for the model.

4.1 Color Sets

The Color Sets for the entire model is shown below.

```
color MAX_T = int with 8..13;
color YELLOW_T = int with 3..7;
color INTERSECTION = with c1|c2|c3|c4|c5|c6 timed;
color PHASE_N = with ph1 | ph2 | ph3 | ph4;
color CR_MINT_YELT = product INTERSECTION*MIN_T * YELLOW_T timed;
color CR_YELLOWT = product INTERSECTION*YELLOW_T timed;
color CR_YELT = product INTERSECTION*YELLOW_T;
color CR_MAXT= product INTERSECTION * MAX_T timed;
color PHASES_SET = product PHASE_N*MAX_T*MIN_T*YELLOW_T;
color CR_PHS = product INTERSECTION*PHASES_SET;
color PHSET_LIST = list PHASES_SET;
color CR_PH = product INTERSECTION*PHSET_LIST;
color CR_PHASE = product INTERSECTION*PHASE_N;
color CROS_PH_LIST = list CR_PHASE;
color B = with b;
var x : PHASES_SET;
var max_t : MAX_T;
var min_t : MIN_T;
var yellow_t : YELLOW_T;
var phase_n : PHASE_N;
var ph : PHASE_N;
var vph1 : PHASE_N;
var vph2 : PHASE_N;
var cr : INTERSECTION;
var cr1 : INTERSECTION;
var cr2 : INTERSECTION;
var cr3 : INTERSECTION;
var cr4 : INTERSECTION;
var cr5 : INTERSECTION;
var cr6 : INTERSECTION;
var last_cr : INTERSECTION;
var phset_list : PHSET_LIST;
var cros_ph_list : CROS_PH_LIST;
var cros1_ph1_list : CROS_PH_LIST;
var cros_ph : CR_PHASE;
var cros1_ph1 : CR_PHASE;
```

Some of the color used in the model are explained as follows:

PHASE_N is a 4-tuple, where each element indicate a possible phase that will be in use in the intersection, at sometime. Phases for example are as shown in Figure 5.

PHASES_SET is a 4-tuple. The first element (PHASE_N) indicates the phase that will be, or is, currently, active for the intersection; the second (MAX_T) and the

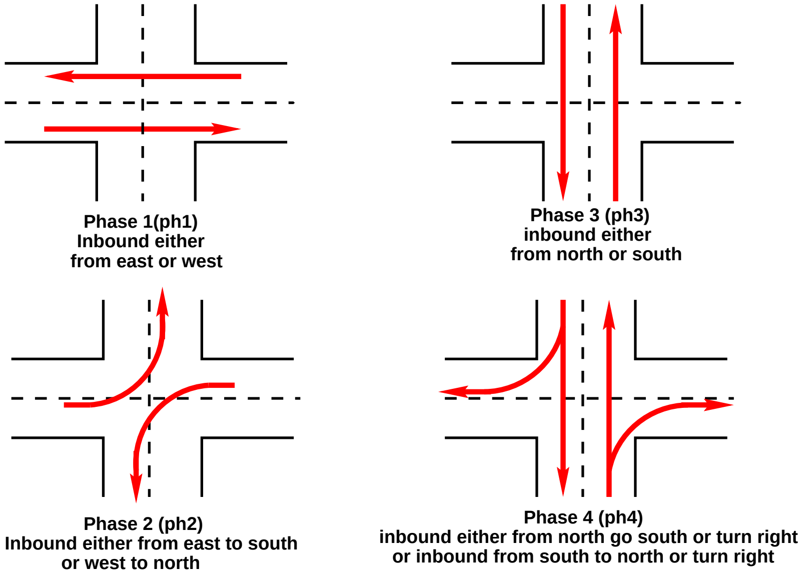


Figure 5: Phases for the example

third (MIN_T) elements indicate, the maximum time and the minimum time that a phase remains active, respectively. And the fourth element (YELLOW_T) indicates the time that the yellow light will remain active.

MAX_T represents the maximum green time for the traffic light.

MIN_T represents the minimum green time for the traffic light.

YELLOW_T represents the yellow time for the traffic light, it varies from phase to phase.

INTERSECTION represents the intersections of the traffic network.

CR_PHASE is a pair defining the active phases for the intersections.

CROS_PH_LIST represents is a list defining pairs specifying the intersection and the phases (CR_PHASE) of a path for a block.

CR_PH represents a pair defining the intersection and a list of possible phases for the intersection with the maximum and minimum green time and the yellow time for the phase.

4.2 Controller Page

The controller page is shown in Figure 6. This CPN models all the controllers of the system. Each controller is represented by a token of the color set CRPH (see Section 4.1) in place IntersectionsAndCorrespondingPhases. The red time of the traffic light was not stored since it is not considered in this model. The firing of transition MakePhases, models the beginning of a phase for each controller. The green time can vary from a minimum value to a maximum one, depending on the existence or not, of a block of cars crossing the intersection. If the minimum time was consumed, WaitMinTime fires, putting a token in place MinimumTimeExpired. If there is no block of cars crossing the respective intersection, in other words, there is a token in place Intersections1, corresponding to this intersection, ChangePhase2 fires and the controller will change the traffic light to yellow, on the contrary the light remains green until max_t, after that transition WaitMaxTime fires. A token in place YellowTime models the amount of time that the traffic light must remain on yellow. A token in Intersections2, represents the end of a phase.

4.3 Flow Page

The Flow page is shown in Figure 7. A token in place BlockOfCarsItinerary represents a block of cars arriving at the intersection or a block of cars waiting for a specified phase. To cross the intersection, blocks of cars wait at the intersection, represented by tokens in place BlockOfCarsItinerary, until the specified phase is the active one for the corresponding intersection and there is no block of cars crossing it. The active phases are defined by tokens in fusion place CrossingPhase (see Controller page description). When these conditions occur, transition EnteringCrossing fires putting a token in place BlockOfCars1 and removing one token, corresponding to the current intersection, from place Intersections1. This allows only one block of cars to cross the intersection at a time. The timed transition BlockCrossingTime models the time spent by a car to move through the intersection. Observe that, this amount of time may vary from block to block, but for simplicity in this model we are not taking this into account. When transition BlockCrossingTime fires, a token corresponding to the actual intersection for a block is put in place Intersections1, and another is put place IntinerartOfBlockOfCars2. The token representing the path is a list of pairs for the intersection and the phase for the block, when a token is removed from place IntinerartOfBlockOfCars1 the head of the list is removed, thus the token put in place IntinerartOfBlockOfCars2 defines the rest of the path.

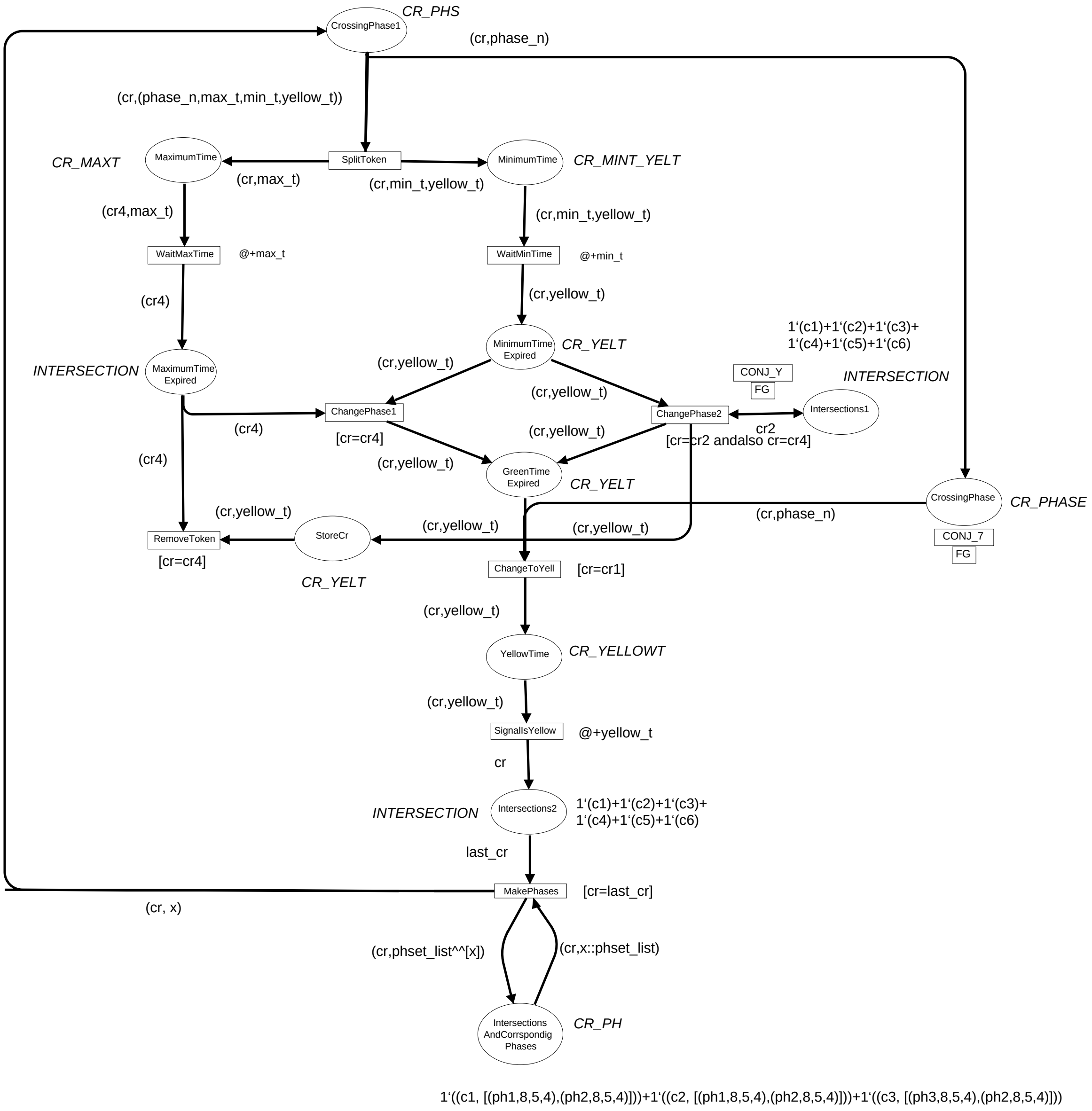


Figure 6: CPN page for the intersection controllers

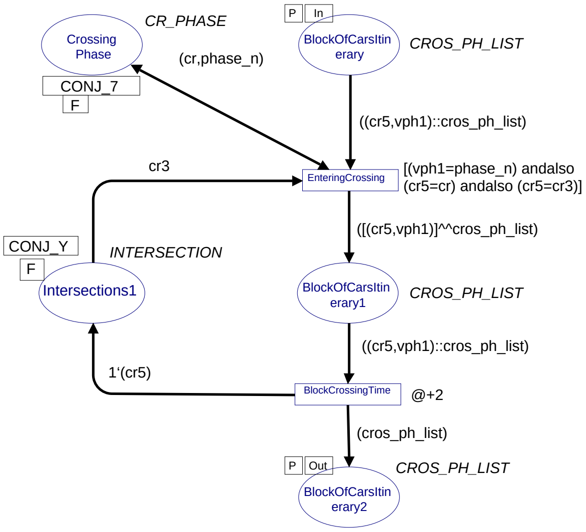


Figure 7: CPN page for the flow of blocks of cars in an intersection

4.4 Traffic Network Page

The Traffic Network page is shown in Figure 7. This is the model the Traffic Network itself, as shown in Figure 3. A token in any of the places Intersectioni (i = 1,2,3,4,5,6) represents a block of cars entering in the intersection. Each transition EnteringInteri (i = 1,2,3,4,5,6) is a substitution transition, modeling the flow page for each intersection. The input places of these transitions correspond to place BlockOfCarsItinerary, modeling the arrival of a block of cars in an intersection as well as a block of cars waiting for a specified phase. When a block of cars is at the intersection a token is put at place ChooseDirectioni (i = 1,2,3,4,5,6). The information represented by this token indicates if a block of cars continues in the system, as for example a token such that 1'([(c2,ph2),(c3,ph3)]) defines that the block of cars goes to the intersection 2, in this case the transition LeavingInter2 fires. Otherwise the block of cars leaves the system, the list specifying the path is empty or the next specified intersection is not immediately reachable. For these cases any of the transitions ExitingSystemi (i = 1,2,3,4,5,6) fire.

In the following section we introduce the main concepts related to the animation of CPN models.

5 Animation Environment

The animation approach introduced in this paper is based on a Java applet that receives as input a textual description of the animation environment and control information for

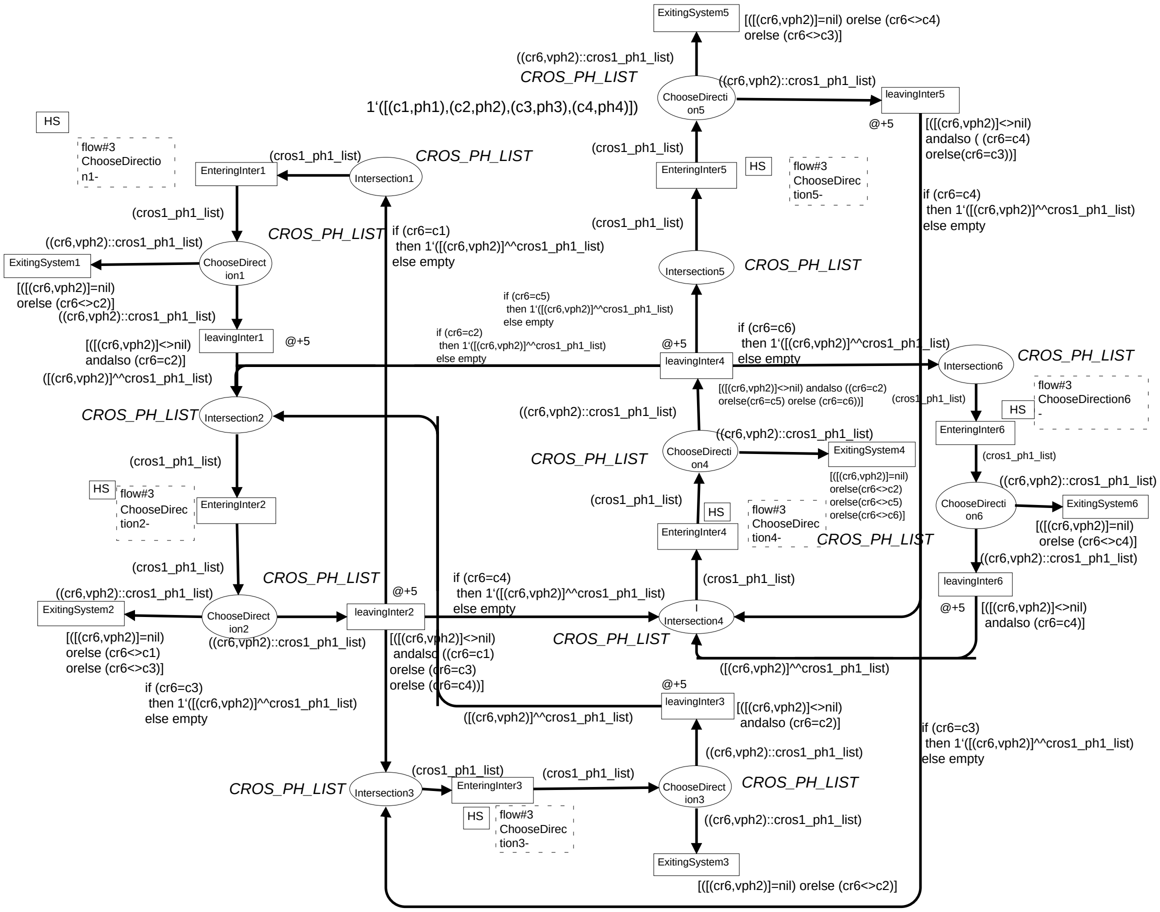


Figure 8: CPN page for the flow of blocks of cars in the network

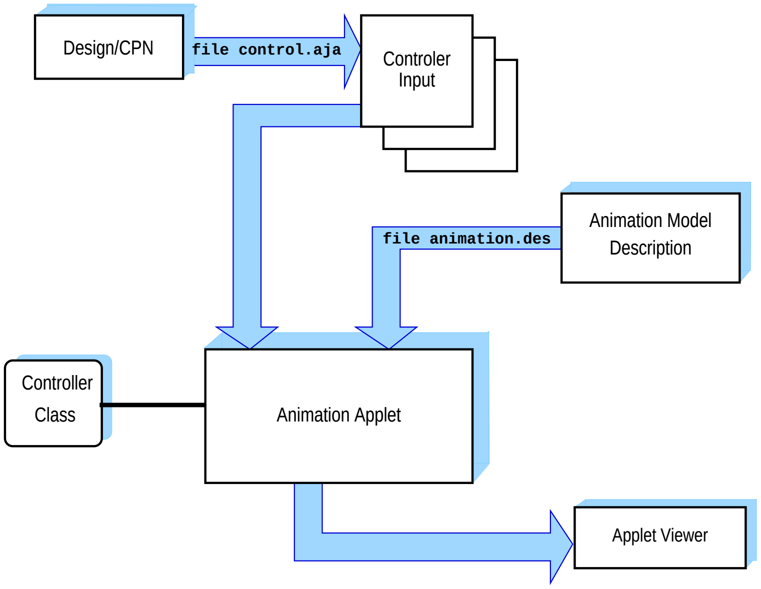


Figure 9: Block diagram for the animation environment

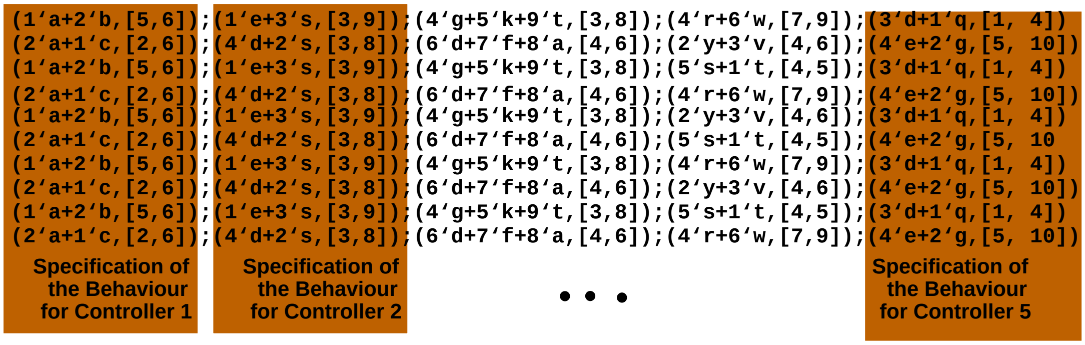
animation purposes from a file. This control file is generated based on the occurrence graph of Design/CPN as explained in Section 2. In Figure 9 a block diagram for the animation environment is shown.

A description of the animation environment follows a grammar that defines all the components taking part in a specific animation. Basically the components are: fixed and mobile objects, a *board* defining the grid where the animation takes place, the routes for the mobile objects, and the controllers for the shared regions in the grid.

Two different files are used for animation purposes: one describing the objects, and a control data file (or stream) Indeed, the control file is a set of vectors (markings) including the specific markings for the controller or controllers defined for the target system. In this paper each position of a vector in the control file defines the phases for the set of signals for each intersection for the traffic network shown in Section 4.

The *Animation Applet* is a set of classes to manipulate the interface actions so that the visual objects can behave according to some basic rules, e.g. objects cannot be in the same area at the same time or they cannot enter in shared regions that are blocked to them. This can be used for example to stop a block of cars in an intersection.

The main characteristic of the implementation is the use of an applet based on a generic controller class that can be defined depending on the system being modeled and animated. For example, in the case of the traffic signal controllers the control file defines the phases and the timing for the controllers of each intersection in the traffic network.



```
(1‘a+2‘b,[5,6]);(1‘e+3‘s,[3,9]);(4‘g+5‘k+9‘t,[3,8]);(4‘r+6‘w,[7,9]);(3‘d+1‘q,[1, 4])
(2‘a+1‘c,[2,6]);(4‘d+2‘s,[3,8]);(6‘d+7‘f+8‘a,[4,6]);(2‘y+3‘v,[4,6]);(4‘e+2‘g,[5, 10])
(1‘a+2‘b,[5,6]);(1‘e+3‘s,[3,9]);(4‘g+5‘k+9‘t,[3,8]);(5‘s+1‘t,[4,5]);(3‘d+1‘q,[1, 4])
(2‘a+1‘c,[2,6]);(4‘d+2‘s,[3,8]);(6‘d+7‘f+8‘a,[4,6]);(4‘r+6‘w,[7,9]);(4‘e+2‘g,[5, 10])
(1‘a+2‘b,[5,6]);(1‘e+3‘s,[3,9]);(4‘g+5‘k+9‘t,[3,8]);(2‘y+3‘v,[4,6]);(3‘d+1‘q,[1, 4])
(2‘a+1‘c,[2,6]);(4‘d+2‘s,[3,8]);(6‘d+7‘f+8‘a,[4,6]);(5‘s+1‘t,[4,5]);(4‘e+2‘g,[5, 10
(1‘a+2‘b,[5,6]);(1‘e+3‘s,[3,9]);(4‘g+5‘k+9‘t,[3,8]);(4‘r+6‘w,[7,9]);(3‘d+1‘q,[1, 4])
(2‘a+1‘c,[2,6]);(4‘d+2‘s,[3,8]);(6‘d+7‘f+8‘a,[4,6]);(2‘y+3‘v,[4,6]);(4‘e+2‘g,[5, 10])
(1‘a+2‘b,[5,6]);(1‘e+3‘s,[3,9]);(4‘g+5‘k+9‘t,[3,8]);(5‘s+1‘t,[4,5]);(3‘d+1‘q,[1, 4])
(2‘a+1‘c,[2,6]);(4‘d+2‘s,[3,8]);(6‘d+7‘f+8‘a,[4,6]);(4‘r+6‘w,[7,9]);(4‘e+2‘g,[5, 10])
```

Specification of the Behaviour for Controller 1
Specification of the Behaviour for Controller 2
• • •
Specification of the Behaviour for Controller 5

Figure 10: Control information for file `control.aja`

The class for the controller, the control file, and the description of the animation file, are integrated by the *Animation Applet*. Each row in the control file represents the states of all the controllers, and the state of each controller is in a specific position of this row. This information is extracted from the occurrence graph. In Figure 10 we illustrate an example for the control file specifying the behavior of five controllers. It is important to point out that at this moment we do not assume that the structure of the controllers are the same. Different classes for each controller can be *plugged* in the animation applet. For the example of the traffic light controllers the behavior is a simple finite state machine that controls sequential timed changes of phases for each intersection.

5.1 Generic Controllers

The idea of generic controllers is based on the possibility of defining abstract classes and interfaces in Java. The generic behavior of a controller is specified by an abstract class named *Controller*. Any controller belonging to the class *Controller* must implement the following basic functionalities:

- take as input a list, as shown in Figure 10, specifying the reachable states.
- have a set of methods to manage the list and take the appropriate control actions.
- implement synchronization mechanisms to support different threads of execution.
- to turn public its status.

For the example of traffic lights the list have the phase and timing information for each controller. Therefore, for each class the designer must define a method to extract this information from the control file. In the next section we shown an example for a simplified traffic network controller.

The controllers operate over shared structures or regions that are accessed by moving objects during the animation. For animation purposes this corresponds to avoid that visual objects occupy the same area on the screen. For example if a traffic light is closed in a given direction the block moving towards that direction must stop. Since the control objects as well as visual moving objects are controlled by different threads of control operating over different critical regions, it is possible to implement a parallel animation, what would not be possible otherwise [6].

5.2 Grammar for the Input Control File

As said before the information needed to describe the animation model is stored in a file. In the following the grammar for description of a model is presented.

```
<animation model> := <input><grid><routes><objects><controllers>
<input>           := input = <path>;
<routes>          := <route><routes> | <route>
<route>           := route IDENTIFIER = <points>;
<objects>         := <object> <objects> | <object>
<object>          := object IDENTIFIER = <size>,<speed>,<quantity>,
                     <id_route>[,<pos>];
<grid>            := grid = <size>,{<active>};
<controllers>     := <controller><controllers> | <controller>
<controller>      := controller IDENTIFIER = <points>,<pos>;
<points>          := <point>,<points> | <point>
<point>           := (INT,INT)
<size>            := <point>
<active>          := <lines>
<lines>           := <line>,<lines> | <line>
<line>            := <row> | <column>
<row>             := r(INT,INT,INT)
<column>          := c(INT,INT,INT)
<pos>             := INT
<path>            := IDENTIFIER
<speed>           := INT
<quantity>        := INT
<id_route>        := IDENTIFIER
```

The terminals are very intuitive and the rules are always very short. Except for the connection between the control file and the controller description, all the other components are basically visual informations. The controller must be associated to a column defining a place of the marking vector for the CPN model representing the state of a controller for the target system. The controllers update their status according to the values defined in each specific column in the marking vector. Based on this grammar one can define the animation in a straightforward way. An example based on the traffic network control is as follows:

```
input = control.file;
grid = (10,10),{r(5,1,10),r(6,1,10),c(5,1,10),c(6,1,10)};
route RT1 = (6,1),(6,10);
route RT2 = (6,1),(6,5),(10,5;
route RT3 = (5,10),(5,1);
route RT4 = (5,10),(5,5),10.5);
object BLOCK1 = (1,1),1,1,RT1;
object BLOCK2 = (1,2),1,1,RT3;
object BLOCK3 = (1,1),1,1,RT4;
controller SEMAPHORES = (5,5)(5,6),(6,5)(6,6),2;
```

For the example shown for the animation file, observe for example the routes `RT1`,`RT2`,`RT3` and `RT4` and the critical region (`SEMAPHORE`) in Figure 11. For more complex configurations, as in the example presented in Section 4 the definition of the animation model is very simple.

6 A Case Scenario

In this section we present details related to the process to obtain the behavior specification for the controllers for a simplified traffic network with two intersections each one with two phases. For this case the occurrence graph (OG) was obtained in 87 seconds, on a time-shared Pentium II, with 256 Mbytes of main memory, running Linux 2.2.12. The OG has 468 nodes and 1408 arcs.

To obtain the specification of the states for the controllers for the green wave the following steps were executed:

1. Define the initial marking and timing definition. In this case, the traveling time between the intersections was 3 time units, and the crossing time for the intersections was 2 time units. The block of cars generation rate was 12 time units.

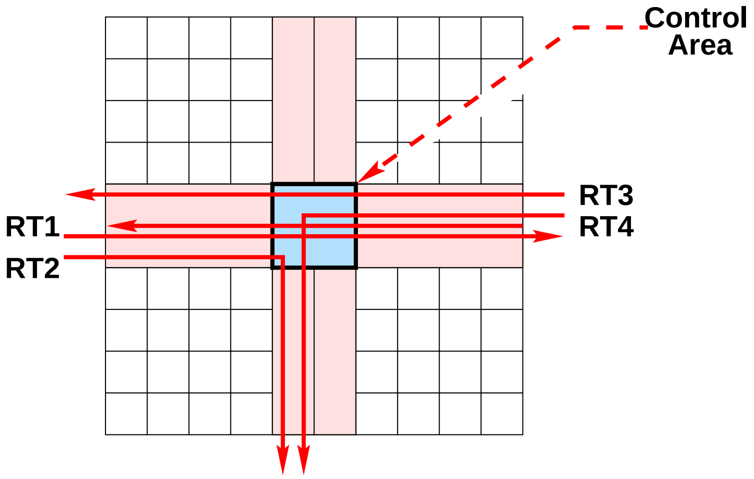


Figure 11: Animation grid in the board

2. Verify whether a green wave exists, if not, then change the timing for the initial marking. To do so, a set of CPN-ML functions were defined and are shown below.

3. Find a set of markings satisfying the green wave property specifying the behavior for the controllers.

```
(* Find the path matching the green wave *)

fun FindThePath ( [] ) = [] |
FindThePath (head::tail) =
let
    val Path = (NodesInPath(InitNode,head)) handle NoPathExists => nil;
in
    if  ((Path <> nil)  andalso ((length(RemoveDup (Path))) > 1))
    then Path
    else FindThePath (tail)
end;

(* Remove duplicated marking from the path (ignoring time stamps) *)

fun RemoveDup( [] ) = [] |
RemoveDup ( h::t ) =
let
val aux = [];
in
  if (memberNode(h,t) orelse
    (StripTime (Mark.Flow'IntersectionsAndCorrespondingPhases 1 h) == empty))
  then RemoveDup (t)
  else h::RemoveDup (t)
end;



(* Verify if a node is member of a path *)

fun memberNode (x,nil) = false
| memberNode (x,h::t) = if (st_Mark.Flow'RemoveDup 1 x) =
                        (st_Mark.Flow'RemoveDup 1 h)
                        then true
                        else memberNode (x, t) ;

(* Save the state of the controller for the green wave *)

fun SaveMarkings (file,[]) = false |
SaveMarkings (file,h::t) =
  (output(file,(st_Mark.Controller'IntersectionsAndCorrespondingPhases 1 h));
SaveMarkings (file,t); true);
```

The specification for the controllers is a file which contents is as follows:

```
C'ICP 1: 1‘(c1,[(ph2,5,3,1),(ph1,6,4,2)])@[0]+ 1‘(c2,[(ph1,5,3,1),(ph2,8,6,2)])@[0]
C'ICP 1: 1‘(c1,[(ph1,6,4,2),(ph2,5,3,1)])@[6]+ 1‘(c2,[(ph1,5,3,1),(ph2,8,6,2)])@[0]
C'ICP 1: 1‘(c1,[(ph1,6,4,2),(ph2,5,3,1)])@[6]+ 1‘(c2,[(ph2,8,6,2),(ph1,5,3,1)])@[9]
C'ICP 1: 1‘(c1,[(ph2,5,3,1),(ph1,6,4,2)])@[10]+ 1‘(c2,[(ph2,8,6,2),(ph1,5,3,1)])@[9]
C'ICP 1: 1‘(c1,[(ph2,5,3,1),(ph1,6,4,2)])@[10]+ 1‘(c2,[(ph1,5,3,1),(ph2,8,6,2)])@[13]
                            |_________|  |___|
     (CL1)_______________________|         |__________(Time Stamp)
```

Observe that ICP is the place IntersectionsAndCorrespondingPhases for the CPN for the controller shown in Figure 6. From each line of this file the applet extracts the required information to manage all the controllers at a given moment. For the example shown, to coordinate the behavior of the first controller "c1", the applet takes the tuples of the column CL1 and the Time Stamp. Based on these two information the applet activates the controller at time zero (@[0]) and phase 1 (ph1). Observe that the value of the Time Stamp defines when controller changed to the actual phase. Thus, the phase remained green for 4 time units and yellow for 2 time units. This information is calculated from the difference between the the current time stamp, the previous time stamp, and the yellow time defined for the previous phase, in this case 2 time units.

7 Conclusions

In this paper we presented an approach to the specification, analysis, design and animation of distributed controllers based on Coloured Petri nets and the Design/CPN tool. From the point of view of the animation the main advantage is that the user can observe the concurrent behavior of a given system modeled using the Design/CPN tool. Also the approach introduced for the definition of the behavior of decentralized controllers can be used for adaptive control for systems which behavior can change, as is the case of traffic light controllers for a network.

To illustrate the approach we presented the design of a coordinated traffic supervision, based on decentralized controllers for the intersections. We have used the model for defining the control for the case of a green wave for arterial traffic. It is important to point out that there are other aspects related to coordinated traffic control that can be considered, such as the “early return to green” problem, inefficient green splits, and cycle lengths being too short or too long [10].

In this paper we have used an approach based on ML functions to search for markings expressing the desired properties for the model. The results obtained indicates that the approach for both, the specification of the behavior of the controllers as well as the animation is correct. In the case of the use of model checking it is still necessary to make some changes in the ASKCTL library, so that information such as the identification of node at the defined property was proved or not.

Also, we are currently investigating the applications of the introduced approach to manufacturing and batch systems supervision and animation. Also, we are refining the solution, both the controllers behavior and the animation, so that a more automatic and interactive tool can be developed. Apart from developing the animation, we are investigating the use of the approach introduced for control. Basically, what is necessary to do is to substitute the the animation applet by a controller applet.

References

[1] D. Bullock and C. Hendrickson. Roadway traffic control software. *IEEE Control Systems Technology*, 2(3):255–264, September 1994.

[2] K. Jensen. *Colored Petri Nets, basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, 1992.

[3] K. Jensen. *Coloured petri nets-Basic Concepts, Analisys Methods and Practical Use*, volume 2. Springer-Verlag, 1997.

[4] K. Jensen. An introduction to the practical use of coloured petri nets. In *Lectures on Petri Nets II: Applications*, volume 2, pages 237–292. Springer, 1998.

[5] K. Jensen and et. al. *Design/CPN Manuals.* Meta Software Corporation and Department of Computer Science, University of Arthus, Denmark. On-line version:http://www.daimi.aau.dk/designCPN/.

[6] O. Kummer, D. Moldt, and F. Wienberg. Framework for interacting design/cpn- and java-processes. In *CPN'98 Workshop Proceedings, http://www.daimi.au.dk/CPnets/workshop98/.* 1998.

[7] Meta Software Corporation and Department of Computer Science, University of Arthus, Denmark. *Design/CPN-ASKCTL Manual.* On-line version:http://www.daimi.aau.dk/designCPN/libs/askctl/.

[8] I.R. Porche. *Dynamic Traffic Control: Decentralized and Coordinated Methods.* PhD thesis, The University of Michigan, Ann Arbor, MI, USA, 1998.

[9] I.R. Porche, M. Sampath, Y.-L. Chen, R. Sengupta, and S. Lafortune. A decentralized scheme for real-time optimization of traffic signals. In *IEEE Proceedings of the 1996 IEEE International Conference on Control Applications*, 1996.

[10] G. Shoup. Traffic signal system offset tuning procedure using travel time data. Master's thesis, West Lafayette, Indiana, USA, 1998.

[11] M. Silva and E. Teruel. Petri nets for the design and operation of manufacturing systems. In *First International Workshop on Manufacturing and Petri Nets*, pages 31–61, Osaka, Japao, Junho 1996.

[12] M. Silva, E. Teruel, R. Valette, and H. Pinguad. Petri nets and production systems. In *Lectures on Petri Nets II: Applications*, volume 2, pages 85–124. Springer, 1998.

[13] J.S. Watson. Reconcilled platoon accomodation at isolated intersections. Master's thesis, West Lafayette, Indiana, USA, 1998.

Redesigning Design/CPN:
Integrating Interaction and Petri Nets In Use

Paul Janecek, Anne V. Ratzer, Wendy E. Mackay
Department of Computer Science, Aarhus University
Aabogade 34
8200 Aarhus N., Denmark
{pjanecek,avratzer,mackay}@daimi.au.dk

Abstract

This paper describes the redesign of the interface for Design/CPN, a graphical editor for building, simulating, and analyzing Coloured Petri Nets. One of our goals in the redesign is to explore how recent advances in graphical interfaces and interaction techniques can improve the editor's support for CP-net designers. This requires an understanding of Petri Nets In Use, our term for the collection of guidelines, work styles, and interaction patterns that influence the way designers build CP-nets. This concept is central to our design framework, and has guided our design process.

In this paper we describe our design framework and give an overview of our design process, a series of empirical studies, brainstorming sessions, and design exercises. We then follow two key issues (setting graphical attributes and alignment) through our design process, and describe how the Petri Nets In Use perspective influenced the evolution of toolglasses and magnetic guidelines as solutions for these issues in the new tool.

Keywords

Participatory Design, Coloured Petri Nets, Design Process, Toolglasses, Magnetic Guidelines

1 INTRODUCTION

A Coloured Petri Net is not simply a mathematical formalism; it uses a graphical representation because it is easier for people to understand. The Design/CPN tool, designed in the 1980's, recognized this and provided a state-of-the-art graphical editor to interactively create, simulate and analyze CP-nets.

The CPN2000 project is a complete redesign of the Design/CPN tool developed at the University of Aarhus. The project draws from four areas of expertise within the computer science department: Coloured Petri Nets, object-oriented languages, human-computer interaction and advanced interaction techniques. The project, funded by Hewlett-Packard, began in February, 1999, and involves participants from each of these research areas, with a core group of 11 people involved in the design.

One goal of the project is to take advantage of new graphical interaction techniques that have been developed in the past decade. However, this is not simply a question of replacing old technology for new. Instead, our approach is to combine these new techniques in novel ways to support the real patterns of use. We call this collection of patterns **Petri Nets In Use**: how people borrow from existing Petri nets, rearrange objects for practical and aesthetic reasons and iteratively test and modify their designs.

This paper describes our attempts to identify the critical elements of Petri Nets In Use and explore how to support them with new interaction techniques in the design of CPN2000. Our research approach includes observation of CPN users at work, and various design activities that capture processes associated with designing Coloured Petri Nets.

The structure of the paper is as follows: First, we explain our design framework, including examples of advances in interaction techniques and an explanation of the concept of Petri Nets In Use. Second, we give an overview of the design process. Finally, we follow two design issues (managing graphical attributes, and aligning objects) through the design process, and show how participatory design and our understanding of Petri Nets In Use led us to develop effective solutions to these issues.

2 DESIGN FRAMEWORK

Our design framework is composed of both technology and use domains. Factors that influence the architecture of the tool, such as its functionality, the graphical interface and interaction techniques, are part of the technology domain. Factors related to the use of the tool, including design guidelines, overall work styles and interaction patterns, are part of the use domain. Our design framework is based on developing an in-depth understanding of these two domains and the interaction between them and combining this with the Petri Nets In Use concept to guide the design of the new tool.

2.1 Design principles

There are three key principles that we use in the design of the overall graphical interface:

1. **Reification:** The process of turning interaction patterns into first class objects. Thus, commands can be made accessible as instruments, combinations of properties can be turned into styles and the selection of multiple objects can be tagged and accessed as groups.
2. **Polymorphism:** Similar operations may be applied to different objects. Thus, various objects can be cut, copied or pasted, any operation can be undone, and operations that apply to a single object can be applied to groups of objects.
3. **Reuse**: Previous commands and the responses by the system can be reused by the user. Thus, input may be reused as the "redo" command and macros, output may be reused as input to other commands, and new commands may be created out of existing commands and a partial list of pre-defined arguments.

The reification principle has strongly influenced the design of the new tool. For example, in the current tool a set of objects are aligned by applying a “vertically align center” command, but additional objects cannot be added without reselecting the aligned objects. In the new tool, this alignment command is reified into a **magnetic guideline**, a visible first-class object which is continuously accessible and modifiable. New objects can be attached to the guideline, and moving the guideline also moves all the objects attached to it.

2.2 Interaction Techniques

The interaction techniques of most traditional graphical interfaces use some combination of Windows, Icons, Menus, and Pointing (WIMP). Although these have a number of strengths, e.g., when well designed they are self-revealing to a novice user, their interaction is often limited to indirect manipulation techniques. This type of interaction forces users to divert their focus from the objects they are working with to commands embedded in menus and dialog boxes. In the context of a graphical editor, this separation between object and action is inefficient and slow. Contextual menus and floating palettes are improvements, but are still indirect manipulation techniques.

In contrast, direct manipulation techniques (Shneiderman, 1998) follow three principles: continuous representation of the objects and actions of interest with meaningful visual metaphors, physical actions or presses of labeled buttons, instead of complex syntax, and rapid incremental reversible operations whose effect on the object of interest is visible immediately.

We are working with the concept of Instrumental Interaction (Beaudouin-Lafon, 1998), which encompasses the range of techniques between direct and indirect manipulation. In this model, instruments mediate the interaction between a user and the objects in the interface.

Toolglasses™ (Bier et al., 1993), for example, are floating, semi-transparent instruments for direct, two-handed manipulation. They are positioned with the non-dominant hand and applied with the dominant hand. A toolglass similar to a color palette would allow a user to apply a color to or absorb a color from an underlying object directly. The non-dominant hand moves the desired color over the object of interest, and then applies the color by clicking through the toolglass on the underlying object with the dominant hand. This allows the user to specify both the object and the action with a single mouse click, in context. An extension of this idea is a **Magic Lens**™, a transparent instrument that shows a modified version of underlying objects. For example, a magic lens version of a color palette would show the portion of the underlying objects within each cell in that cell's color. Each cell in the palette would effectively become a graphical preview.

Another advanced interface technique is **layers** (Fekete, 1996), which creates graphical sets of objects whose visibility and depth can be controlled like overlapping layers of transparencies. Layers are an effective way of separating structural objects from informational objects. This allows the user to manage the complexity of the view by adjusting the visibility of a layer based on its relevance to the current activity. For example, comments could be placed into one layer and simulation feedback in another, allowing the user to fade or hide items when they are not the focus of attention.

Standard architectures for graphical interfaces do not support these "Post-WIMP" interaction techniques and they are not trivial to add to existing interface toolkits. We have, therefore, completely redesigned the user interface architecture to support these new types of interaction.

2.3 Petri Nets In Use

To introduce the new interaction techniques and apply the design principles, we have conducted several studies of current CP-net design practice. Petri Nets In Use is our term for the collection of factors that influence the way a designer works with Coloured Petri Nets. We divide these factors into three groups: visual representation, work style, and interaction patterns.

The graphical characteristics of the **visual representation** are essential in communicating the underlying meaning of the CP-net to others. We look at this communication role both within smaller workgroups, in larger communities of designers in a certain domain area, and in general between designers familiar with the language of CP-nets.

CP-net designers have developed a number of guidelines for building readable nets. Some of these guidelines are applicable to graphs in general, some are specific to CP-nets, and some are used only within certain workgroups. Jensen's (1992) readability guidelines, for example, are general and applicable to most CP-nets. They include suggestions on the use of structure and graphical attributes to emphasize the flow of data and the semantic relationships between objects. Information visualization techniques (Noik, 1994) and guidelines from graph drawing research (Di Battista, 1999) are additional sources for techniques designers may employ to communicate and emphasize the meaning of their net. Understanding these guidelines and how designers employ them is essential in the process of designing an editor to support them.

Interaction patterns describe the designer's low level interactions with the design tool during the process of building a CP-net. For example, does the designer use keyboard shortcuts instead of menus? Does the designer create new objects or cut, paste and edit existing objects? Understanding these patterns in the current tool helps us determine if they should be supported in the future tool, improved or replaced with different interaction patterns.

The **work styles** describe the designer's overall process of creating a net. Several of the modeling guidelines described in Jensen (1992) suggest a top-down framework: the designer first models the core functionality and then progressively adds details, switching between editing (building the model) and simulation (testing the model). Schön (1983) described design less formally, as an iterative process of "reflection-in-action", and "seeing and moving". Feedback from our user group and examples from industrial use (Christensen et al, 1997) show how these approaches are employed in CP-net design. During this process, the designer may create a net in isolation, or use parts of existing nets, design guidelines, and group conventions. Jensen, (1992) discusses additional guidelines for modeling CP-nets involving the use of abstraction and structure. How individual designers employ these guidelines is their work style.

The components of Petri Nets In Use are intertwined: the guidelines that a workgroup use for the visual aspects of a net will affect the way that members of that workgroup create a net; the factors that influence the work process will also affect the result. Understanding these relationships helps us design a tool to support them.

2.4 Redesigning Petri Nets In Use

Figure 1 illustrates how the components of the design framework combine in the process of creating the new tool. Through the different design process activities, the users have worked on redesigning their Petri Nets In Use, and we have used this input to guide our design of the new tool.

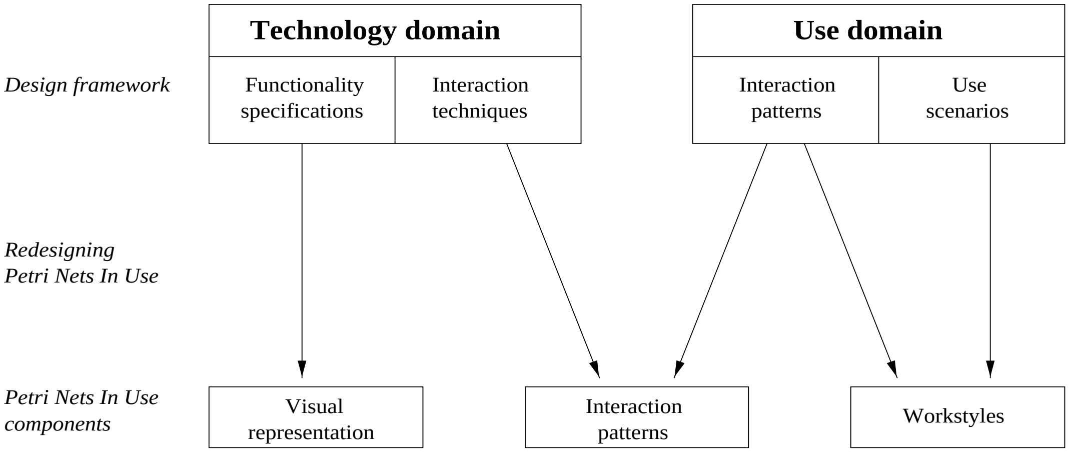


Figure 1. Petri Nets In Use and the Design Framework

The **functionality specifications** of the design tool are created through a requirements analysis and define what the new tool must support both in terms of the visual representation, and the simulation/analysis of the underlying model. Our analysis of the current tool, research in graphical interfaces, and the participation of CPN experts define the functionality of the future tool.

In the use domain, the **interaction patterns** and **use scenarios** describe how the users of the current tool create and work with Coloured Petri Nets. We rely on observation of users and interviews of experts to identify and understand these components. Through analysis of these observations and interviews, and feedback from users, we identify areas where work styles can be improved and integrated in the new tool. The **interaction techniques** we have introduced and the **interaction patterns** from the current tool together help define the interaction patterns of the new tool. This area of the design is entirely reinvented through our design process.

3 DESIGN PROCESS

The project uses a participatory design approach, in which we rely on the active participation of expert CP-net designers throughout all phases of the project. Our design process involves four key activities: studying Petri nets as they are actually used in real-world settings, generating ideas for new ways of interacting with the CPN2000 tool, designing the tool itself, and evaluating the design ideas (Figure 2).

Studying Use	Generating Ideas	Designing	Evaluating designs
observation	text brainstorming	design workshops	experiments
interviews	video brainstorming	software prototypes	prototype evaluation
analysis	scenario brainstorming	design scenarios	design walkthroughs

Figure 2. Design activities

3.1 Studying use

We began by studying how novice and expert users build and edit CP-nets. We conducted two types of user studies: video observations and interviews. For the video observations, we visited a range of users, including expert users at a small company working on their current project, an academic expert working through the solutions to student exercises as well as his own project, and a group of students trying out the same set of exercises. In each case, we visited people in their offices (or in the computer lab) and asked them to work as they would normally rather than give us a demo. We interviewed each person afterwards and asked them to explain their actions and identify key problems. We also interviewed several local CP-net experts about their ways of using the tool and what they felt were problems with using the tool.

We performed several types of data analysis. The most basic involved coding activities and problems from the videotapes and making transcripts of the interviews. From these, we created a video summary of the most common problems, contrasting the experiences of novice and expert users. We also created **use scenarios**, comprised of typical activities and problems observed in the field studies. These scenarios enabled us to create concrete representations of Petri Nets In Use, including current work styles and interaction patterns, and helped us focus on areas where they could be improved.

We were particularly struck by the time the users spent adjusting layout and how often they were forced to constantly shift their attention as they searched for commands and other information. Subsequent design activities were specifically organized to address the problems we identified during the field studies.

3.2 Generating ideas

To kick off the design activities, we engaged in several different types of brainstorming sessions, with the goal of generating, rather than evaluating, new ideas. We began with traditional brainstorming sessions, with both CPN experts and other researchers. Everyone was asked to generate a list of problems with the new tool and ideas for possible solutions. We provided inspiration the week before by showing video clips of new interaction techniques that might be relevant to the new CPN tool.

Subsequent brainstorming sessions involved video, which forced participants to demonstrate, rather than just talk about, new ideas. Participants were given paper, scissors, transparencies, pens, post-it notes and other supplies and asked to simulate ideas for new ways of interacting with the new tool. We found that having participants perform the interaction in front of the camera forced them to work out the details of the interaction and created a video record that was used later to create software prototypes. Early video brainstorming sessions were open-ended, with participants offering design solutions to a range of problems within a specified area, such as layout. Later video brainstorming sessions were organized around use scenarios, in which participants were asked to generate multiple ways of supporting the work activities shown in the video.

3.3 Designing prototypes

Our design process involves a combination of activities, including a systematic review of the technical functionality required in the tool, a comparison of the interaction techniques possible for each function, and an analysis of how the choice of particular interaction techniques affects users' patterns of interaction with the tool. Whereas the brainstorming sessions helped expand the range of possibilities being considered, the design sessions involved the difficult task of balancing tradeoffs and actually making design decisions. Using our design principles helped us generate new solutions to design problems, ensure uniformity within the interface and resolve design conflicts. Most design sessions generated new issues and some design decisions were deferred, requiring either additional user studies or new brainstorming sessions.

Our design activities included a series of design workshops, with a mix of software designers, human-computer interaction researchers and Coloured Petri Net experts. Each workshop examined a different design issue. We also developed video design scenarios to help us examine how CP-net developers would use the tool under different work conditions. For example, designing a new net is different from editing an existing net or modifying it to make it conform to local guidelines. We had to ensure that the new design could accommodate developers working in each context.

To explore the consequences of particular design decisions we created software prototypes, derived from the video brainstorming sessions and other design activities. We also developed a completed demonstration of the current state of the design, in the form of a video prototype scenario. We used this video to communicate the current look and feel of the design to each other, new members of the team and to the programmers developing the code.

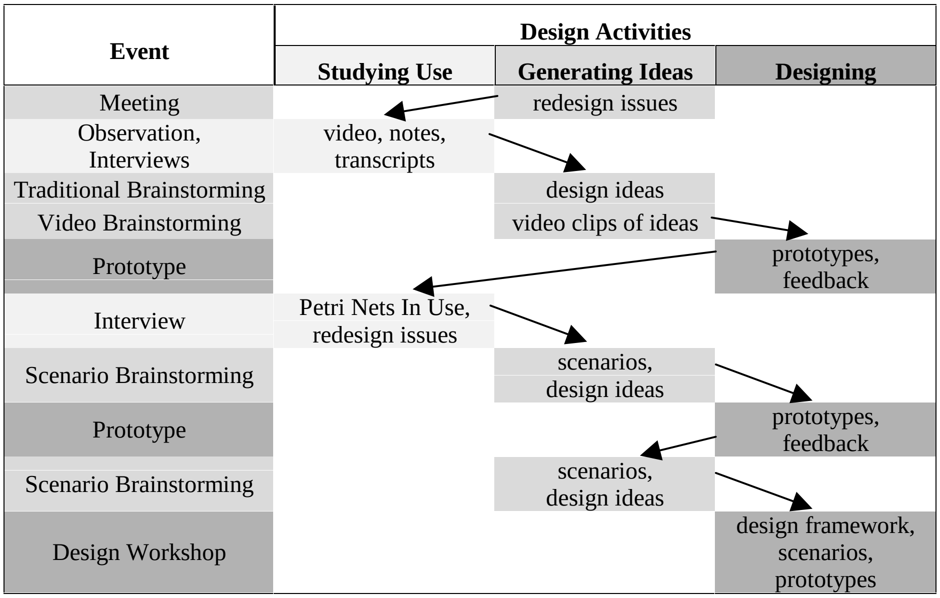
3.4 Evaluating designs

Our design process involves evaluation of ideas throughout, since we obtain feedback on a regular basis from experienced CP-net users. In addition, we have begun a series of more formal evaluation activities, including experiments to test specific design decisions and design walkthroughs to give us qualitative feedback about the look and feel of the tool. These activities are currently underway but are beyond the scope of this paper.

4 DESIGN PROBLEMS AND SOLUTIONS

Figure 3 shows the flow of our design process. The sequence of events is listed in the left column, and each event is shaded to match its corresponding design activity. The artifacts generated by each event are placed in the column corresponding to the type of design activity, and the flow of the design process is emphasized with arrows. The artifacts created in this process were critical in our identification of problems, our understanding of Petri Nets In Use, and the evolution of ideas into the design of the new tool.

Two significant issues identified during this process were improved support for managing graphical attributes and positioning objects. The new editor supports these issues with toolglasses and magnetic guidelines. The following sections will trace the evolution of these ideas through the design process, and how participatory design and our focus on Petri Nets In Use has helped us to refine and improve these ideas.

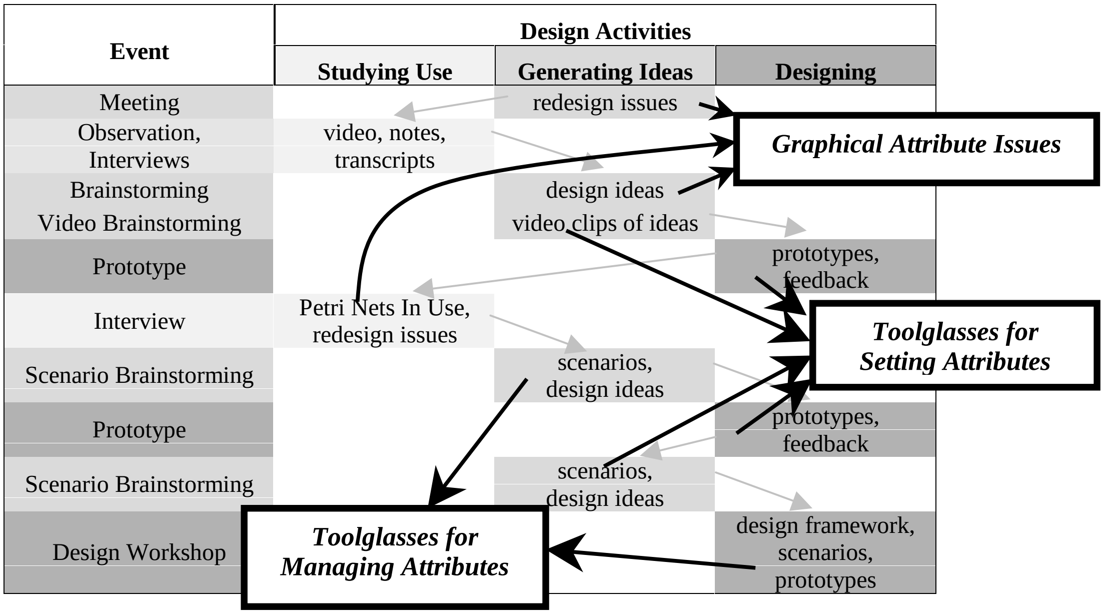


Event	Design Activities		
	Studying Use	Generating Ideas	Designing
Meeting		redesign issues	
Observation, Interviews	video, notes, transcripts		
Traditional Brainstorming		design ideas	
Video Brainstorming		video clips of ideas	
Prototype			prototypes, feedback
Interview	Petri Nets In Use, redesign issues		
Scenario Brainstorming		scenarios, design ideas	
Prototype			prototypes, feedback
Scenario Brainstorming		scenarios, design ideas	
Design Workshop			design framework, scenarios, prototypes

Figure 3. The flow of events and their relationship to design artifacts

4.1 Problem 1: Managing graphical attributes

Figure 4, which is based on figure 3, shows how the different activities in our design process contributed to our understanding of graphical attribute issues. The preliminary meeting raised two issues concerning graphical attributes: support for styles, and a better way of copying attributes from one object to another.



Event	Design Activities		
	Studying Use	Generating Ideas	Designing
Meeting		redesign issues	
Observation, Interviews	video, notes, transcripts		
Brainstorming		design ideas	
Video Brainstorming		video clips of ideas	
Prototype			prototypes, feedback
Interview	Petri Nets In Use, redesign issues		
Scenario Brainstorming		scenarios, design ideas	
Prototype			prototypes, feedback
Scenario Brainstorming		scenarios, design ideas	
Design Workshop			design framework, scenarios, prototypes

Graphical Attribute Issues
Toolglasses for Setting Attributes
Toolglasses for Managing Attributes

Figure 4. Sources for graphical attribute issues and toolglass ideas

In the first brainstorming session, people raised several issues on styles and setting attributes: style sheets, personal configuration of styles, selection by graphical property (e.g., all red transitions). At this point, the ideas were general; they described ways the users wanted to be able to work but did not contain details on interaction. Style sheets, the first idea, make it easier to formalize standard guidelines used for related groups of objects or types of objects. The second idea, personal configuration, supports switching between the guidelines developed inside different work groups, e.g. one group may use thick and thin lines, while another group might use the colors blue and green. The third idea, selection by property, overlaps with the style idea, but is more informal. For example, a user can select all red transitions and change them to thick lines directly, without changing a style.

These issues are important components of Petri Nets In Use; sharing a style among objects often emphasizes a semantic connection between the two, or adherence to specific guidelines. Both of these issues highlight a need for better support for managing and reusing object attributes.

4.2 Solution 1: Toolglasses

The lower two boxes in figure 4 show the evolution of toolglasses in our design process as a way to solve these graphical attribute issues. Toolglasses appeared to us as an obvious choice for working with attributes. As explained earlier, they are transparent tools that bring the action to the object, a feature that is not supported with traditional menus and dialog boxes. We explained the concept of toolglasses to the CPN experts participating in our **video brainstorming** workshop, and they developed several design ideas using toolglasses to set graphical attributes.

Figure 5 shows an example of a "styleglass" produced in the video brainstorming workshop. The video clip created during the workshop shows how a toolglass displays objects in the net in different styles: one style for the client side of the network, one for the server side, etc. The toolglass is moved with the non-dominant hand, using a trackball, and the dominant hand can use the mouse to click on the toolglass frame with the desired style to apply that style to the object. Participants at the workshop developed several variations on this idea: One-handed toolglasses, toolglasses that can be resized, and toolglasses for port assignment.

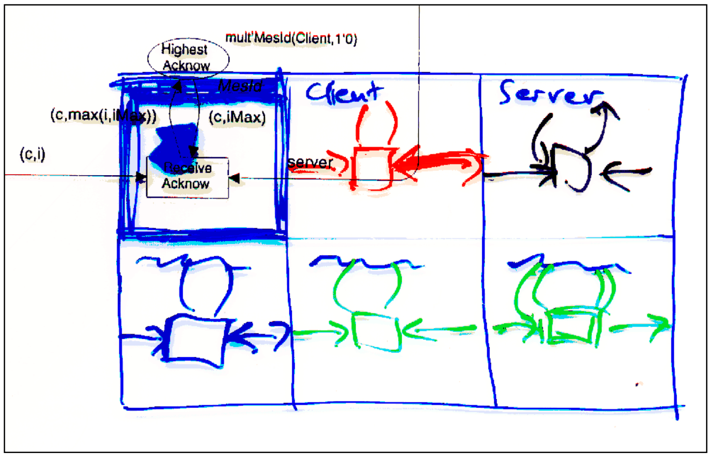


Figure 5. Style preview toolglass

The styleglass has one transparent frame where the object of interest in the underlying net can be seen unmodified. Surrounding this transparent frame are eight opaque frames for displaying the object in different styles. A variation of this is a toolglass where all frames are semi-transparent, but each displays the underlying net in a different style. Both toolglasses apply the style to the object of interest by clicking on the style. There is a trade-off between the two concerning the visual complexity of the screen image: With the opaque frames one object is shown in several styles making comparison easy, but they cover parts of the underlying net. With the transparent frames, it is more difficult to compare the different styles for one object, but more of the net structure is visible. These are some of the tradeoffs we investigated in the first prototype.

We wanted to test how the toolglass with styles would actually look and feel so that we could compare the different versions. We built several **prototypes** of the toolglasses (e.g., changing the number and color of the frames, switching between one and two-handed input) in Tcl/Tk and presented them to the project group for evaluation.

The general feedback was positive. People were surprised at how easy and natural two-handed input felt. The advantages of two-handed input (smooth movement and positioning of the toolglass, easy interaction with the underlying net) clearly made up for the small amount of desk space the extra input device occupies. Actually trying the toolglasses convinced the group of their potential, and they became central to the new design as a result of this prototype.

Given the simplicity of these first prototypes, the feedback was mostly on the idea level and not really an in-depth evaluation. Some of this feedback (how many frames should we have, where do you click to apply changes to an object, etc.) was used in the design phase and for the subsequent prototypes.

Given the input from the earlier brainstorming sessions, we wanted to develop these ideas further in the context of realistic scenarios. In the **scenario brainstorming** workshop we constructed two scenarios illustrating realistic use situations, and recorded them on videotape. These two clips showed simple editing of graphical attributes on a single page, selecting all items with some particular property and changing them to a new style. We showed the participants the video clip, and then asked them to brainstorm how the task could be performed with the future tool.

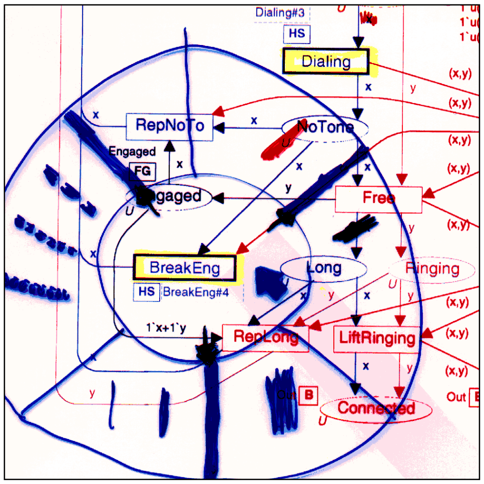


Figure 6. Graphical search and replace toolglass

The workshop resulted in a number of different versions of a search and replace toolglass: a toolglass that makes it easy to find and change items with a certain property or combination of properties. Figure 6, an artifact from the workshop, shows a circular toolglass with three sections around the outside for line style, line thickness, and line color. Clicking through the center picks up the attributes of the object underneath, changes the current settings in the different sections of the toolglass, and highlights other objects in the diagram with the same attributes. A related toolglass allowed the user to specify the values of attributes to search for and the attributes to replace them with. It is possible, for example, to set the toolglass to find all objects that are red and make their outlines dashed. After this workshop, one CPN expert commented that if graphical search and replace was very easy, then formal support for styles may not be necessary.

In the second use scenario workshop, one group developed a toolglass to pick up and apply attributes. Its appearance was very similar to the search-and-replace toolglass, showing the values picked up from the objects in the fields of the toolglass. Clicking through the middle of the toolglass on an object then applied these attributes to the object. It was also possible to set the desired values of attributes directly in the toolglass.

We based the second **prototypes** on the evaluation of the first and the design ideas from the use scenario workshop. All prototypes in this phase were programmed in Beta and connected to the underlying user interface tool kit.

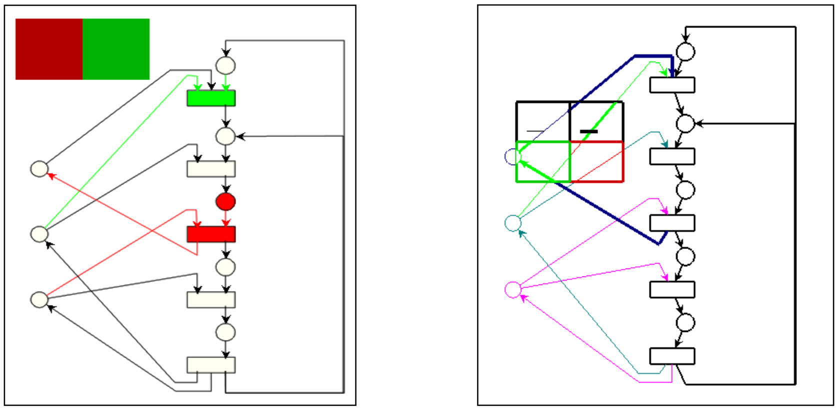


Figure 7. Toolglass prototypes. (a) Click through Toolglass. (b) Magic Lens version of Toolglass

We built two simple versions of toolglasses for changing the graphical attributes of objects in the net. Figure 7a shows a toolglass prototype that a user clicks through to apply an attribute to an underlying object. Figure 7b shows a **magic lens** version of the earlier prototype with transparent frames showing the net underneath in different styles, with click-through for applying the styles. The upper two cells of the toolglass show the arcs in different line thicknesses, and the lower two cells show the arcs in different colors.

In the **design workshops**, we continued to define and evaluate the low-level interaction with toolglasses through scenarios of use. We have also begun looking into how to integrate and manage toolglasses in a workspace.

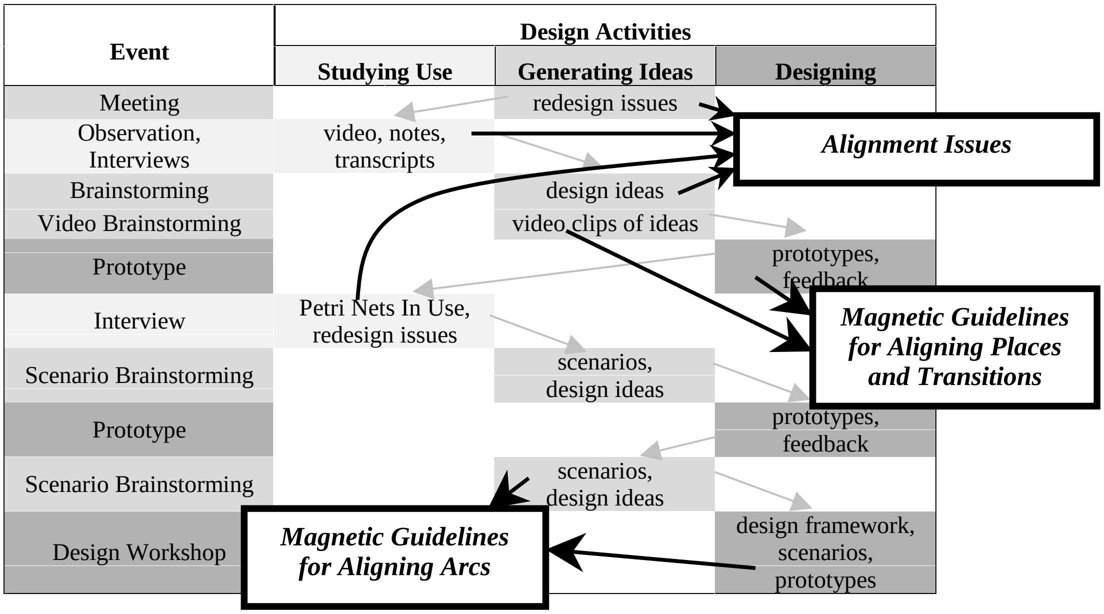
The adoption of toolglasses as a solution for setting attributes was not clear until the first working prototypes where users could actually experience them. Since then, they have become increasingly sophisticated. Originally they were a way of managing styles, and more recently they have become tools for inspecting attributes, and graphical search and replace.

4.3 Problem 2: Positioning objects

The second problem we focus on in this paper is positioning objects. The box in the upper right of figure 8 shows an overview of the events that highlighted the problems of positioning objects, and especially rerouting arcs. For example, we analyzed a video clip of an expert from industry engaged in the relatively simple task of creating a new place, redirecting arcs to that place from another, and connecting other objects in new ways. This task takes approximately three minutes, over 60% of which involve tedious and repetitive rerouting operations. Clearly much time and effort could be saved if a better way of interacting with arcs was invented. Positioning of arcs should be more highly automated, according to the positions of the objects the arcs are attached to, based on simple guidelines for how arcs should be routed through a net.

The first input on the issue of positioning of objects came from the preliminary meetings, and was about having constraints as a supplement to ordinary alignments. From the first brainstorming session there were only a few ideas on constraints - one was to have constraints between bendpoints on arcs, an idea that later became essential for ideas on manipulating arcs.

The issues of manipulating arcs and aligning objects might appear to be separate, but both have to do with how to position objects easily, and arcs are essential to this. When an object is moved, the arcs must follow. The users have attempted to integrate these two areas of work throughout the design process.



Event	Design Activities		
	Studying Use	Generating Ideas	Designing
Meeting		redesign issues	
Observation, Interviews	video, notes, transcripts		
Brainstorming		design ideas	
Video Brainstorming		video clips of ideas	
Prototype			prototypes, feedback
Interview	Petri Nets In Use, redesign issues		
Scenario Brainstorming		scenarios, design ideas	
Prototype			prototypes, feedback
Scenario Brainstorming		scenarios, design ideas	
Design Workshop			design framework, scenarios, prototypes

Alignment Issues
Magnetic Guidelines for Aligning Places and Transitions
Magnetic Guidelines for Aligning Arcs

Figure 8. Sources for alignment issues and magnetic guideline ideas

4.4 Solution 2: Magnetic Guidelines

The lower two boxes in figure 8 show an overview of the events that influenced our design of **magnetic guidelines** as a solution to the problem of positioning objects. Figure 9a shows a video clip from the **video brainstorming** workshop illustrating the idea of magnetic guidelines. In this clip, when an object is moved over another object, guidelines will appear for both of them, and the object moved will “snap-to” the guideline of the other object, so that they are aligned. The guidelines can be more or less advanced, appearing only for the center of the object or for center and sides, so that it is possible to align to different points of the objects this way.

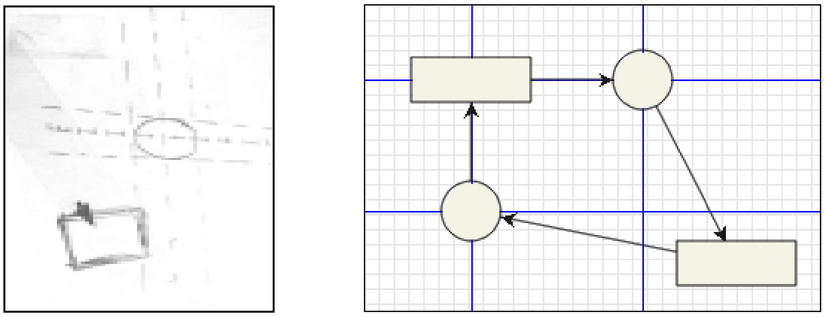


Figure 9. Magnetic Guidelines: (a) video clip of an idea from video brainstorming (dashed lines are guidelines); (b) software prototype showing three objects attached to two horizontal and two vertical guidelines. Objects dragged near a guideline snap onto it. Moving a guideline moves the objects attached to it.

Figure 9b is a screenshot from a **prototype** of magnetic guidelines. In the prototype, it is possible to insert objects (places and transitions) on a canvas, connect them with arcs and create guidelines. When moved over the guidelines, the objects will snap to the guidelines as demonstrated in the video brainstorming clip. The guidelines can be toggled to show or not show, but objects will still snap to when near a (possibly invisible) guideline. An important feature demonstrated in the prototype is that guidelines can be selected and moved, which also moves any attached objects. This is an example of a **reification** of the alignment constraint - the constraint between the aligned objects has been turned into an object that can itself be moved and manipulated.

For the **scenario brainstorming** workshops we constructed use scenarios from user study videotapes. Figure 10a is from a video clip on rerouting arcs that was turned into a use scenario with a storyboard. In the workshops we worked through the scenario to redesign the interaction in a new design context.

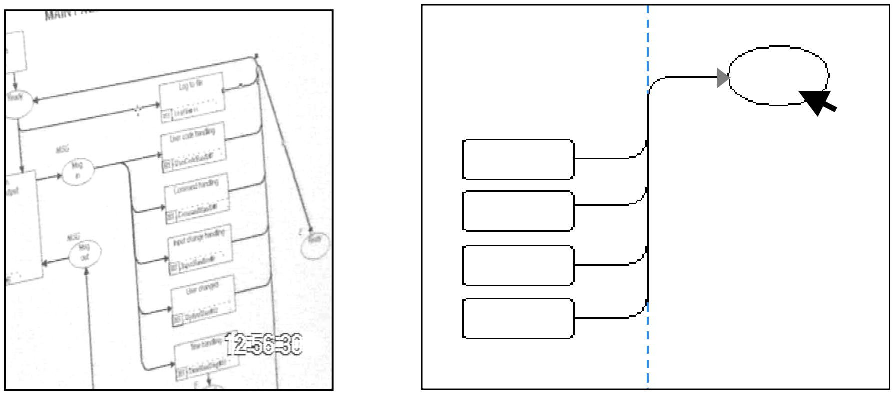


Figure 10. Magnetic guidelines for arcs. An image from the video scenario is shown in (a), and a sketch for the design idea is shown in (b)

Inspired by the prototype on magnetic guidelines and the use scenario, one of the groups in the workshop developed the idea of guidelines for arcs. These guidelines control the routing of an arc by holding the bendpoints of an arc in a fixed position relative to the objects the arc is connected to. The alignment of an arc segment is adjusted automatically when the guideline or one of the objects is moved, as shown in figure 10b. This is a continuation of the reification of alignment constraints - the constraints between the arc segments and bendpoints (originally mentioned in the first brainstorming) can now be manipulated.

In the **design workshops** we have unified the concepts of aligning and moving arcs, arc segments, nodes, etc. through the use of guidelines. We have recently begun integrating spreading constraints into guidelines as well. These guidelines evenly spread objects that are attached to them. More complex layouts can then be achieved by combining these two types of guidelines. For example, attaching several alignment guidelines to a spreading guideline creates equally spaced parallel rows of objects.

Turning all constraints into guidelines of the same overall nature makes it possible to create **polymorphic** interaction techniques for manipulating different kinds of constraint guidelines. These guidelines can then be combined or split according to a “guideline algebra,” making constraints a powerful and flexible functionality in the new tool.

The evolution of design ideas to handle graphical attributes and object alignment demonstrates the influence of Petri Nets In Use in our design process. Design issues were highlighted during observation, interviews and feedback from users. Design ideas were generated and progressively refined by CP-net experts, who were creating their future work styles. The participation of CP-net users has been essential throughout the design process.

5 CONCLUSIONS

This paper describes a series of design activities with a multidisciplinary group of participants, including CP-net experts, to redesign the Design/CPN graphical editor. We followed two key issues (managing graphical attributes, and positioning objects) through our design process and showed how the Petri Nets In Use perspective guided the evolution of our design solutions. These solutions integrate recent advances in graphical interfaces and interaction techniques (toolglasses and magnetic guidelines), and they have been tailored to fit the specific needs of CP-net designers.

Our goal in this paper is to share our current understanding of Petri Nets In Use with members of the CP-net community; both to get feedback about our observations, and input into the design of the new tool. We hope that increased understanding of Petri Nets In Use will increase the effectiveness of the new CPN tool and provide even greater support for CP-net designers.

6 ACKNOWLEDGMENTS

We would like to thank the CPN users, students and experts who participated throughout the design process, and the members of the Beta, HCI and CPN groups for their support. We would also like to thank Susanne Bødker for her comments on an earlier draft of this paper.

7 REFERENCES

Beaudouin-Lafon, M. (1997) Interaction instrumentale: de la manipulation directe à la réalité augmentée. In Actes Neuvièmes journées francophones sur l'Interaction Homme Machine (IHM'97), Futuroscope, September.

Bier, E., Stone, M., Pier, K., Buxton, W., and DeRose, T. (1993) Toolglass and magic lenses: The see-through interface. In *Proceedings of the 20th annual conference on Computer graphics*, pp. 73-80.

Christensen, S., Jørgensen, J., Madsen, K. (1997) Design as Interaction with Computer Based Materials. In *Proc. ACM Conference on Designing Interactive Systems: Processes, Practices, Methods, and Techniques, DIS'97,* Amsterdam, The Netherlands, pp. 65-71.

Di Battista, G., Eades, P., Tamassia, R., & Tollis, I. (1999) Graph Drawing -- Algorithms for the Visualization of Graphs. Prentice Hall, New Jersey.

Fekete, J. (1996) Using the Multi-Layer Model for Building Interactive Graphical Applications. In *Proc. ACM Symposium on User Interface Software and Technology, UIST'96*, Seattle, USA, pp. 109-118, November.

Jensen, K. (1992) Coloured Petri Nets -- Basic Concepts, Analysis Methods and Practical Use. Volume 1, Basic Concepts. EATCS Monographs on Theoretical Computer Science, Springer-Verlag.

Noik, E. G. (1994) A Space of Presentation Emphasis Techniques for Visualizing Graphs. In *Proc. Graphics Interface '94*, pp. 225-233.

Schön, D. (1983) The Reflective Practitioner. New York: Basic Books.

Shneiderman, B. (1998) Designing the User Interface. Addison-Wesley.

Parameterised Coloured Petri Nets

Thomas Mailund

Department of Computer Science
University of Aarhus
DK-8000 Aarhus C
Denmark
mailund@daimi.au.dk

Abstract. In this paper we examine Coloured Petri Nets extended with parameters. We characterise three kinds of parameterisation and formally define *Parameterised Coloured Petri Nets*. We then discuss how parameterised Coloured Petri Nets can be used to create libraries of Coloured Petri Nets modules in the same way as libraries for programming languages. Finally we discuss how to implement a simple simulator for such modules.

1 Introduction

In Coloured Petri Nets (CPN), as in traditional programming languages, it is infeasible to work on industrial size problems as one single unit. To tackle problems of a certain size, it is necessary to work on smaller units, which can later be composed into the full system.

A modular approach to modelling makes larger systems easier to handle. There is less to validate which reduces the debugging period. Even verification usually takes benefit of modular models, though the situation here is a bit more complex, since the environment a module is put into (i.e. the neighbourhood of the substitution transition) tends to influence the dynamic properties of the module, that one wishes to verify.

But perhaps more importantly, a modular framework opens up for reuse of commonly used constructions. From a modelling point of view, reuse saves time and effort. From a validation point of view, reuse increases faith in correctness, since a module tested in depth in one environment, is likely to work as expected in a similar environment. Needless to say, reusing a verified module is vastly better than creating a new, untested module.

It is often the case that large models contain a number of similar constructions, where only a few details differ. Modularisation alone does not allow for much reuse in such circumstances, however, the constructions can in most cases be changed in a way, such that their differences depend on a set of parameters, which can be assigned when needed. This observations leads to the concept of Parameterised Coloured Petri Nets (PCPN).

In [2] three kinds of parameterisation were identified: type, expression,[1] and net parameterisation, and a short discussion of a possible implementation was given. In this paper we formally define Parameterised CPN, and the three kinds of parameter assignments. We also repeat the discussion on implementation issues, but this time based on an actual implementation of a simulator.

The paper is organised as follows: Sect. 2 introduces the concepts through small toy examples, showing the use of type, expression, and net parameters. In Sect. 3 we formally define Parameterised CPN and the three kinds of parameter assignments. Readers only interested in the practical use of Parameterised CPN can safely skip this section. In Sect. 4 we discuss how Parameterised CPN naturally leads to a module system. In Sect. 5 we show how to implement a simulator to take benefit of this module system. Readers only interested in the use of Parameterised CPN and not tool development can safely skip this section. Finally we consider future work in Sect. 6 and conclude in Sect. 7.

Familiarity with (non-hierarchical) Coloured Petri Nets [5] is assumed throughout this paper, and familiarity with SML, especially the module system [9, 10], is assumed for Sect. 5.

[1] In [2] expression parameters were called value parameters, however, we find that the term expression parameters are more true to the definitions given in this paper.

2 A small example

In this section, we will introduce the general ideas of parameterised CPN, via a small toy example. Say we want to count the number of times a certain transition fires. We can do this by a construction like the one in Fig. 1. If we substitute the transition whose firings we wish to count with the net in the figure, we get the number of firings by the value of the token on place C.

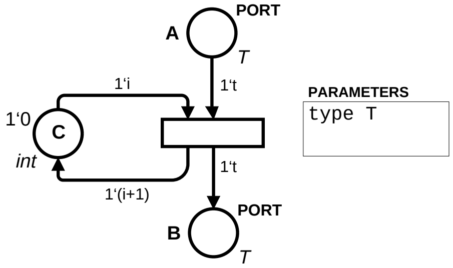


Fig. 1. $\mathcal{C}$ - Counter net.

2.1 Type Parameters

In Fig. 1 we do not know, nor do we care about the type T of the places A and B. We only care about the token on place C which counts the firings of the transition. Any transition between places A and B (with the same colour set) could be substituted with the net in Fig. 1, regardless of the type of A and B. We would like to think of the type T as a parameter to be specified at the location where we use the net.

Consider Fig. 2 where we have places A and B with colour set **int**. To count the firings of transition Y we could substitute Y with the net $\mathcal{C}$ where we have assigned type T to **int**.

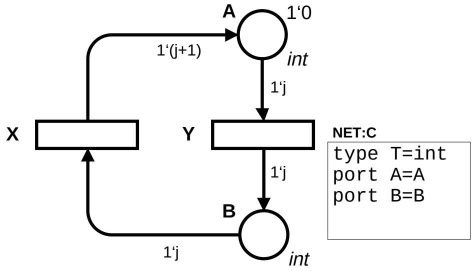


Fig. 2. Using the counter net.

The resulting net is shown in Fig. 3. Here transition Y has been removed and the counter net $\mathcal{C}$ has been inserted in its place. The two nets are then connected through the border places of the substituted transition in the first net, and a subset of the nodes of the second net. The connection places of the first net are called the *sockets* of the substituted transition. The connection places of the second net are called the *ports* of the net.

A substitution can be thought of as an operation on nets. Given two nets it creates a new net by replacing a transition in the first net with the second net as described. For a formal description of substitution we refer to Sect. 3.

In practice we do not create the new net in this way, but simply describe the substitution with a *NET* region as seen in Fig. 2. Here we name the net to substitute with, together with assignments of parameters and port/socket assignments, determining how the two nets should be combined.

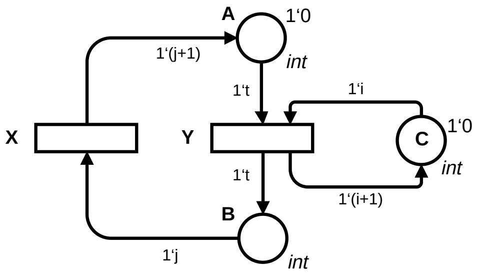


Fig. 3. Fig. 2 and the counter net after substitution.

If more than one transition is substituted with the same net, we create copies of that net, and let a copy substituted each transition. This makes it possible to use the same net in several locations, instantiated with different types. Consider Fig. 4. Here we have places A and B of type **int** separated by transition X, and places A′ and B′ of type **bool** separated by transition Z. To put a counter on both X and Z we could substitute it with the net in Fig. 1, assigning type T to be **int** for X and **bool** for Z.

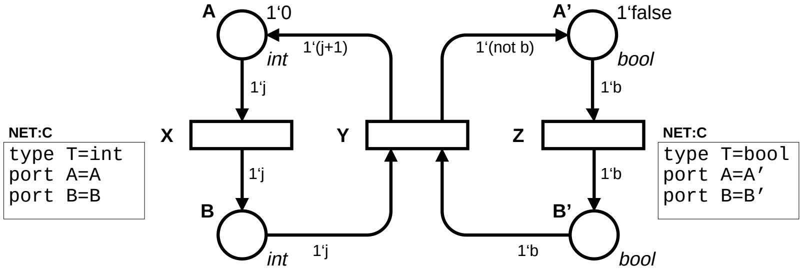


Fig. 4. Using the counter net with different types.

This would then lead to the net shown in Fig. 5.

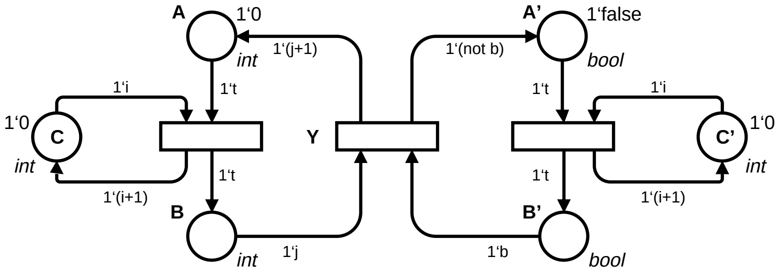


Fig. 5. Using the counter net with different types, after substitution.

2.2 Expression Parameters

We might not want to count *all* firings, but only some, depending on the binding, and we might not want to increment the counter with the same value in all cases. We cannot specify this directly in our counter net, since the choice of which firings to count will depend on the specific uses of the net. Instead we can add a predicate to the net to decide when to increment the counter, and a value specifying the step with which to increment, and let both be parameters of the net. These parameters are called *expression* parameters of the net.

In Fig. 6 we see the counter net from Fig. 1 extended with a predicate p that, given a value of type T, returns a boolean, which then determines whether to increment the counter. The predicate is an expression parameter to the net, as indicated in the box to the right of the net. Furthermore we have the expression s of type **int**. This is used to specify the step with which to increment the counter.

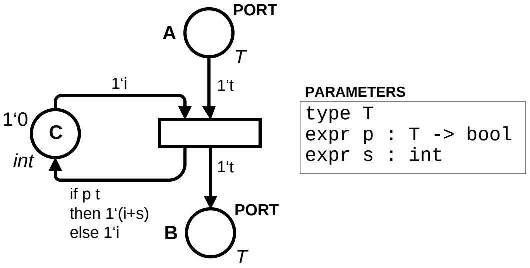


Fig. 6. $\mathcal{C}'$ - Counter with value parameters.

These expression parameters are mentioned in the *PARAMETERS* box, together with the type parameter T. We have typed the parameters in the declaration, and we only allow expressions with the right type to be assigned to the expression variables.

Fig. 7 shows the new net in use. The left part counts all firings of X after i has exceeded 5, and does this with an incrementation step of 2. The right part only counts the firings of Z when b is *true*. Since b is negated in each cycle, only every second firing of Z is counted. The incrementation step is 1.

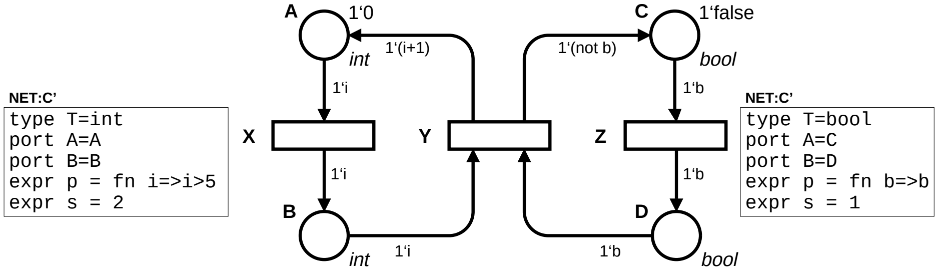


Fig. 7. Using the counter with expression parameters.

2.3 Net Parameters

Not only types and values are interesting as parameters. Sometimes, we would like to be able to leave parts of nets undefined, until the net is used. What we need is a way of parameterising parts of a net, and a way of substituting such parts with other nets. For the latter we need to decide exactly how the rim of a parameter part in one net should be glued together with the second net. In this paper we will only allow transitions to be parameters, and glue nets together by fusing the border places of substituted transitions with a subset of places from the second net. For two other ways of substitution (place and arc substitution) we refer to [8].

In a way we have already seen how net parameters should work. In Fig. 2 we said that we substituted the transition Y with the counter net $\mathcal{C}$. This was done by simply referring to $\mathcal{C}$ in the *NET* region of Y. Here we assumed that $\mathcal{C}$ was globally known. Instead we could demand that the net referred to was a parameter.

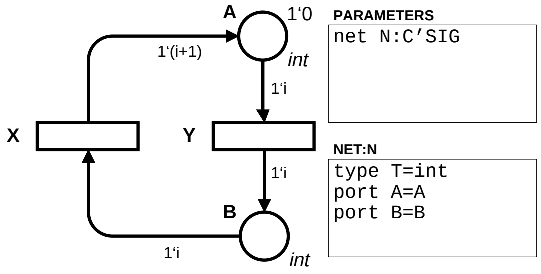


Fig. 8. Net with net parameter.

Consider Fig. 8. This is almost the same net as in Fig. 2. The difference is the *PARAMETERS* box and the name in the *NET* region. In Fig. 8 we substitute Y with a net $\mathcal{N}$ which is given as a parameter to the net.

The net parameter is restricted to a specific *signature*. The signatures do for net parameters what type declarations does for expression variables. It specifies which nets are legal parameters, by fixing how any parameter must look like to ensure a legal substitution. Signatures can be automatically generated from the nets. We will assume this has been done for the nets in this paper, and we will refer to the signature of a net $\mathcal{N}$ as $N'SIG$.

For a net, the signature is defined by the set of ports (a subset of the places) and the parameters the net takes, i.e., the type, expression, and net parameters. In the case of the counter net $\mathcal{C}$ (Fig. 1), the signature specifies one type parameter T and two ports, named A and B, both with colour set T. $C'SIG$ refers to this exact signature. The net in Fig. 8 accepts any net with type parameter T and ports A and B with colour set T.

Another net with signature $C'SIG$ is show in Fig. 9. This is a net that logs all bindings of a transition on the place L. The net in Fig. 8 can be instantiated with any net with signature $C'SIG$, and thus with both the counter net and the log net as net $\mathcal{N}$.

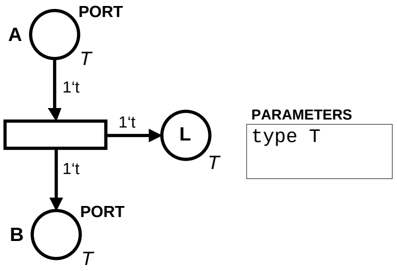


Fig. 9. $\mathcal{L}$ - Log net.

Of course we did not really need net parameters for switching between $\mathcal{C}$ and $\mathcal{L}$. We could simply change the name in the *NET* region. It gets more interesting when net parameters are nested, that is, when the assignment of net parameters depend on the net they are used in.

Consider Fig. 10. Denote this net $\mathcal{LR}$. $\mathcal{LR}$ chooses randomly whether the left or the right transition fires. The net takes a type parameter T and a net parameter $\mathcal{N}$, with signature $C'SIG$. We can assign both $\mathcal{C}$ and $\mathcal{L}$ to N. If we assign $\mathcal{C}$, $\mathcal{LR}$ will count how many times each transition fires, if we assign $\mathcal{L}$, $\mathcal{LR}$ will log the binding elements.

Now consider Fig. 11. This net takes a net parameter $\mathcal{M}$ with signature $LR'SIG$. It substitutes the transition X with $\mathcal{M}$ where the net $\mathcal{N}$ is assigned $\mathcal{L}$, and it substitutes transition Z with $\mathcal{M}$ where $\mathcal{N}$ is assigned $\mathcal{C}$. If $\mathcal{M}$ was assigned $\mathcal{LR}$ we would log left and right on X and count left and right on Z.

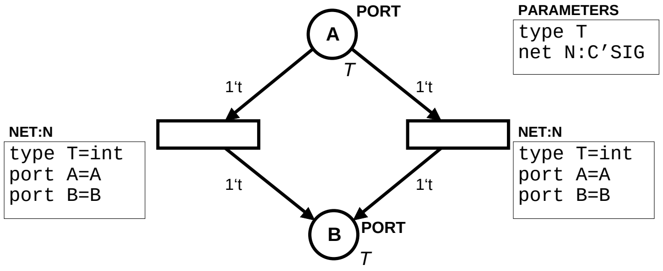


Fig. 10. $\mathcal{L}R$ - Random left or right.

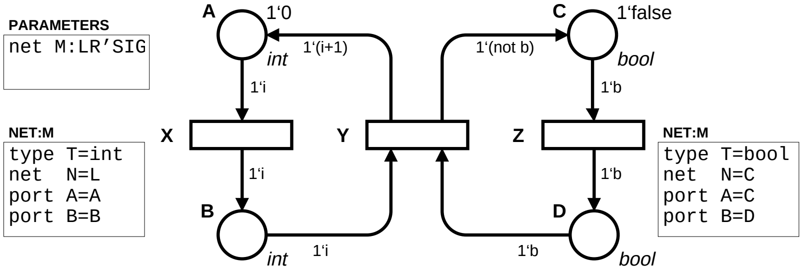


Fig. 11. Nested net parameters.

3 Formal Definitions

In this section we will define formally what we mean by Parameterised CPN. We define three different kinds of parameters: type, expression, and net parameters. We then combine these into Parameterised CPN. For net parameters we will only consider transition substitution. Place and arc substitution have been examined in [8]. The important ideas however appear in the definition of transition substitution, and examining all three kinds of substitution will only complicate the definitions. For a more detailed description of the definitions we refer to [8].

3.1 Type Parameters

As we saw in the examples in Sect. 2 we would like to leave some types undefined until a specific type is actually needed. We would like to refer to *type variables* instead of actual types, and assign types to variables when necessary. In the examples we only used type variables for colour sets, but in general we would want to use the type variables when constructing types, e.g., lists or products of a variable type.

We will assume the existence of some type language, e.g., a set of basic types and ways of constructing products, sums etc. of types. We let $\mathbb{T}$ denote this language, i.e., $\mathbb{T}$ denotes the set of expressions created over the type constructions. Furthermore we assume that $\mathbb{T}$ includes variables from a set of *type variables* **TV** which is disjoint from any other set in the definition.

We assume that no type in $\mathbb{T}$ is empty, i.e. $\forall T \in \mathbb{T} \,.\, T \neq \emptyset$. $\mathbb{T}$ must contain the type **bool** $= \{\text{true}, \text{false}\}$. This is used in the definition of *guards* (see Def. 6). Finally, we assume that $\mathbb{T}$ contains a multiset construction $(\,\cdot\,)_{MS}$. This is used in the definition of *arc functions* and *initialisation functions* (see Def. 6).

Definition 1 (Type assignment). *A* type assignment $\Gamma : \mathbf{TV} \rightharpoonup \mathbb{T}$ *is a partial mapping from type variables* **TV** *to type expressions in* $\mathbb{T}$. □

For partial mapping Γ we let $dom(\Gamma)$ denote the elements on which the mapping is defined.

Definition 2. *For type expression $T \in \mathbb{T}$ and type assignment Γ, denote by $T[\Gamma]$ the expression obtained by replacing all variables in the domain of Γ, $v \in dom(\Gamma)$ appearing in T with $\Gamma(v)$.* □

We will only consider legal type assignments, i.e., for expressions $T \in \mathbb{T}$ we will only consider $\Gamma : \mathbf{TV} \rightharpoonup \mathbb{T}$ such that $T[\Gamma] \in \mathbb{T}$.

3.2 Expression Parameters

As for type parameters we would like to use *expression variables* in inscriptions, and assign actual expressions to these variables when necessary.

We will assume the existence of an inscription language $\mathbb{E}$, with a set of type rules such that all valid expressions in $\mathbb{E}$ evaluate to values with a type in $\mathbb{T}$. We will assume that we can talk about the variables of an expression $E \in \mathbb{E}$ and that we can talk of the type of a variable. We assume two different kinds (i.e. two disjoint sets of variables) of variables, "ordinary", or *binding*, variables and *expression variables* **EV**. The former is used in bindings, the later for expression parameters. For variable v let $Type(v)$ denote the type of v. We will demand that $Type(v) \in \mathbb{T}$ for all variables. By $Type(E)$ we denote the type of expression E, and we demand that $Type(E) \in \mathbb{T}$. By $Var(E)$ we denote the binding variables in E.[2]

Definition 3 (Expression assignment). *An* expression assignment Δ *is a partial mapping from expression variables* **EV** *to expressions in* $\mathbb{E}$, $\Delta : \mathbf{EV} \rightharpoonup \mathbb{E}$. □

Definition 4. *For expression* $E \in \mathbb{E}$ *we denote by* $E[\Delta]$ *the expression obtained by replacing all variables* $v \in dom(\Delta)$ *appearing in* E *with* $\Delta(v)$. □

We will only consider legal value assignments, i.e., for expressions $E \in \mathbb{E}$ we will only consider $\Delta : \mathbf{EV} \rightharpoonup \mathbb{E}$ such that $E[\Delta] \in \mathbb{E}$.

3.3 Net Parameters

Intuitively net parameters work in much the same way as type and expression parameters. We associate a variable to parts of a net, and allow for this part to be substituted with another net when necessary. Net parameters are a little more complex however, since assigning actual nets to variables requires that we define a way to glue two (or more) nets together. In this paper we only consider *transition substitution* to glue nets together.

In the following we assume we are considering some universe of Parameterised CPN closed under the operations described in this section. Let **PCPN** denote the set of all PCPN in the universe, let **P** denote the union of all place sets in **PCPN**. We define PCPNs shortly (Def. 6), but for now, just think of **PCPN** as a set, where each element has an associated set, called the places of the net. We furthermore assume we have a set of *net variables* **NV** and a set of *port names* **PN**.

Definition 5 (Net assignment). *A* net assignment *is a mapping* $\Theta : \mathbf{NV} \rightharpoonup Pow(\mathbf{PN} \times \mathbf{P}) \times \mathbf{PCPN}$, *where* $Pow(\mathbf{PN} \times \mathbf{P})$ *denotes the powerset of* $\mathbf{PN} \times \mathbf{P}$. Θ *should satisfy for all* $net \in dom(\Theta)$ *with* $\Theta(net) = (R, \mathcal{N})$, *if we let* P *denote the set of places of* $\mathcal{N}$ *the relation* $R \subseteq \mathbf{PN} \times P$ *relates port names to the places in* $\mathcal{N}$. □

A net assignment maps net variables to nets and give a relation which names a subset of the ports of the nets. The naming of ports is used later for gluing nets together.

3.4 PCPN

With the three kinds of assignments defined, we are ready to define PCPN, and describe how assignments affects nets.

Let **TASS** denote the set of type assignments, i.e., $\mathbf{TASS} = (\mathbf{TV} \rightharpoonup \mathbb{T})$, let **EASS** denote the set of expression assignments, i.e., $\mathbf{EASS} = (\mathbf{EV} \rightharpoonup \mathbb{E})$, and let **NASS** denote the set of net assignments, i.e., $\mathbf{NASS} = (\mathbf{NV} \rightharpoonup Pow(\mathbf{PN} \times \mathbf{P}) \times \mathbf{PCPN})$.

Definition 6 (PCPN). *A* Parameterised CPN (PCPN) *is a tuple* $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$ *such that*

1. P *is a finite set of* places

[2] $BVar(E)$ would perhaps be a better notation, however $Var(E)$ is used in [5] so we stick to this convention.

2. *T is a finite set of* transitions
3. *A is a finite set of* arcs, *such that*

$$P \cap T = P \cap A = T \cap A = \emptyset$$

4. *N is a* node *function.* $N : A \rightarrow (P \times T \cup T \times P)$.
5. *C is a* colour *function* $C : P \rightarrow \mathbb{T}$.
6. *G is a* guard *function* $G : T \rightarrow \mathbb{E}$ *such that*

$$\forall t \in T \,.\, \mathit{Type}(G(t)) = \mathbf{bool}$$

7. *E is an* arc expression *function,* $E : A \rightarrow \mathbb{E}$ *such that*

$$\forall a \in A \,.\, \mathit{Type}(E(a)) = C(p(a))_{MS}$$

where $p(a)$ is the place of $N(a)$, and $C(p(a))_{MS}$ denotes the set of multi-sets over $C(p(a))$.

8. *I is an* initialisation *function,* $I : P \rightarrow \mathbb{E}$ *such that*

$$\forall p \in P \,.\, \Big(\mathit{Type}(I(p)) = C(p)_{MS} \;\wedge\; \mathit{Var}(I(p)) = \emptyset \Big)$$

9. *$\mathit{Ports} \subseteq P$ is a subset of the places.*
10. *TS is a* substitution mapping $TS : T \rightharpoonup \mathbf{NV} \times \mathit{Pow}(P \times \mathbf{PN}) \times \mathbf{TASS} \times \mathbf{EASS} \times \mathbf{NASS}$ *is a partial mapping from the transitions of $\mathcal{N}$ to pairs of net variables and relations of places and port names. For transition $t \in T$ let $S(t) = \bullet t \cup t \bullet$. $S(t)$ is called the* sockets *of t. TS should satisfy, that for all $t \in \mathit{dom}(TS)$ with $TS(t) = (\mathit{net}, R, \Gamma, \Delta, \Theta)$, $R \subseteq S(t) \times \mathbf{PN}$, that is the relation is a relation between the sockets of t and port names. $\mathit{dom}(TS)$ is called the* substitution transitions *of $\mathcal{N}$.*

□

The definition of *TS* looks a little ghastly, but will hopefully be more clear after we have discussed the three kinds of assignments related to PCPN.

Intuitively, *TS* relates to each substitution transition a net variable and a naming of sockets, and a triple of a type, expression, and net assignment. Given a net assignment, which relates the net variable to an actual net, the assignments are applied to this net and the relation is used to “glue” the two nets together.

In the examples we used the *NET* regions to define *TS* (see Fig. 12). The name (after the colon) is the net variable, and refers to the net parameter in the PARAMETERS box. The relation and the three assignments are defined in the box. We use *type* to define type assignments, *expr* to define expression assignments and *net* to define net assignments. The relation is defined by the *port* assignments. Whenever we refer to a net variable which is not a parameter, we assume that the variable refers uniquely to a specific net. We can then think of the net as already being assigned.

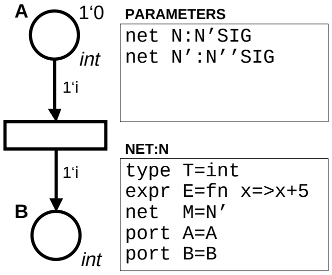


Fig. 12. Example of *NET* region.

If we ignore *Ports* and *TS* for a moment, we can see that PCPN looks a lot like non hierarchical CPN as defined in [5] (Def. 2.5). We do not have the set of colour sets Σ. Instead we have the type languages $\mathbb{T}$. Furthermore, CPN as defined in [5] does not allow for type nor expression variables.

We will only consider the behaviour of PCPN with no type and expression variables, i.e., for which all variables have been assigned (see Def. 16 and Def. 12). This behaviour is the same as for CPN.

For transition t let $\mathit{Var}(t)$ denote the (binding) variables of t i.e., the union of the binding variables in the guard of t and the arc expressions on the arcs with source or destination in t. $\mathit{Var}(t)$ does not include any expression variables.

$$\forall t \in T \,.\, \mathit{Var}(t) = \{\, v \mid v \in \mathit{Var}\,(G(t)) \vee \exists a \in A(t)\,.v \in \mathit{Var}\,(E(a))\,\}$$

where $A(t)$ denotes the set of arcs with source or destination in t.

For expression $E \in \mathbb{E}$ and mapping $b : \mathit{Var}(E) \to \mathbb{E}$ denote by $E\langle b\rangle$ the expression obtained by replacing all variables in the domain of b, appearing in E with $b(v)$. This closely matches the definition of expression variable substitution, however the different set of brackets $[-]$ vs. $\langle - \rangle$ distinguish the two.

Definition 7 (Binding, [5] Def. 2.6). *A* binding *of a transition t is a function b defined on $\mathit{Var}(t)$, such that*

1. $\forall v \in \mathit{Var}(t)\,.\, b(v) \in \mathit{Type}(v)$
2. $G(t)\langle b\rangle = \mathit{true}$

By $B(t)$ we denote the set of all bindings for t. □

Definition 8 ([5] Def. 2.7). *A* token element *is a pair (p, c) where $p \in P$ and $c \in C(p)$, while a* binding element *is a pair (t, b) where $t \in T$ and $b \in B(t)$. The set of all token elements is denoted TE while the set of all binding elements is denoted by BE.*

A marking *is a multi-set over TE while a* step *is a non-empty and finite multiset over BE.* □

Let $E(x, y)$ denote the (multi-set) sum of expressions on arcs between x and y. This will be well defined since these expressions will range over the same multi-set. Either x or y must be a place, and the expressions will range over the associated colour set.

$$\forall (x, y) \in (P \times T \cup T \times P)\,.\, E(x, y) = \sum_{a \in A(x,y)} E(a)$$

where $A(x, y)$ is the set of arcs from x to y.

Definition 9 (Enable, [5] Def. 2.8). *A step Y is* enabled *in a marking M if*

$$\forall p \in P\,.\, \sum_{(t,b)\in Y} E(p,t)\langle b\rangle \le M(p)$$

Definition 10 (Occur, [5] Def. 2.9). *When a step Y is enabled in a marking M it may* occur, *changing the marking to another marking M', defined by*

$$\forall p \in P\,.\, M'(p) = \left(M(p) - \sum_{(t,b)\in Y} E(p,t)\langle b\rangle \right) + \sum_{(t,b)\in Y} E(t,p)\langle b\rangle$$

□

3.5 Assignments

We can now define how to assign values to parameters.

Definition 11. *For PCPN $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$ and type assignment Γ, let $\mathcal{N}[\Gamma]_T = (P, T, A, N, C', G, E, I, Ports, TS')$ denote the net obtained by replacing in $\mathcal{N}$ all appearances of type variables $v \in dom(\Gamma)$ with $\Gamma(v)$. Then*

$$\forall p \in P \ . \ C'(p) = C(p)[\Gamma]$$

and TS' defined for all $t \in dom(TS)$ by $TS'(t) = (net, R, \Gamma^*, \Delta, \Theta)$ when $TS(t) = (net, R, \Gamma', \Delta, \Theta)$ where $\Gamma^*(v) = \big(\Gamma'(v)\big)[\Gamma]$ whenever $v \in dom(\Gamma')$ □

Substituting types in a net changes the colour function and propagates the substitution to subnets through the transition substitution function. Type substitution is likely to change the type of variables. We do not consider this more formally since it depends heavily on the way *Type*(v) is given.

Definition 12. *For PCPN $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$ and expression assignment Δ, we let $\mathcal{N}[\Delta]_E = (P, T, A, N, C', G, E, I, Ports, TS')$ denote the net obtained by replacing in $\mathcal{N}$ all appearances of expression variables $v \in dom(\Delta)$ with $\Delta(v)$, where*

1. *$G'(t) = G(t)[\Delta]$ for all $t \in T$*
2. *$E'(a) = E(a)[\Delta]$ for all $a \in A$*
3. *$I'(p) = I(p)[\Delta]$ for all $p \in P$*
4. *TS' defined for all $t \in dom(TS)$ by $TS'(t) = (net, R, \Gamma, \Delta^*, \Theta)$ when $TS(t) = (net, R, \Gamma, \Delta', \Theta)$ where $\Delta^*(v) = \big(\Delta'(v)\big)[\Delta]$ whenever $v \in dom(\Delta')$*

□

Expression substitution changes the different inscription functions. Again, the substitution is propagated to subnets via the *TS* function.

We might need to put some restrictions on the expressions we allow a given variable to be substituted for. In the examples in Sect. 2 we explicitly typed the expression parameters used. This is not explicitly required from the definition, but is used to ensure that all expression assignments used are *legal*. Other restrictions might be needed for other inscription languages or for other purposes. We will not consider this further however.

Before we define how to apply a net assignment (Def. 17), we need to define how we substitute transitions with nets (Def. 16), but first we need to define how to fuse places:

Definition 13 (Place fusion). *A* place fusion set *F of a PCPN $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$ is an equivalence class partition of P such that*

$$\forall p' \in [p]_F \ . \ C(p') = C(p) \ \wedge \ I(p') = I(p)$$

where $[p]_F$ denotes the class of F containing p. □

Definition 14. *For PCPN $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$, and a place fusion set F over P, we can* fuse *the places and get a net $\big[\mathcal{N}\big]_F = (P', T, A, N', C', G, E, I', Ports', TS)$ given by*

1. *$P' = \big\{ [p]_F \ \big| \ p \in P \big\}$*
2. *$N' : A \to P' \times T \cup T \times P'$ defined by*

$$a \mapsto \begin{cases} \big([p]_F, t\big) & \text{if } N(a) = (p, t) \in P \times T \\ \big(t, [p]_F\big) & \text{if } N(a) = (t, p) \in T \times P \end{cases}$$

3. *$C' : P' \to \mathbb{T}$ defined by $[p]_F \mapsto C(p)$.*
4. *$I' : P' \to \mathbb{E}$ defined by $[p]_F \mapsto I(p)$.*
5. *$Ports' = \{ [p]_F \,|\, p \in Ports \}$*

□

A place fusion creates a place for each fusion class and connects a (class-)place and a transition if there is an original place in the class, connected to the transition.

Notice that any class containing a port itself becomes a port.

Definition 15 (Port/Socket relation). *Let $\mathcal{N} = (P, T, A, N, C, G, E, I, Ports, TS)$ be a PCPN. For PCPN $\mathcal{N}$ and $\mathcal{N}' = (P', T', A', N', C', G', E', I', Ports', TS')$ and transition $t \in T$ a relation $R \subseteq Ports' \times S(t)$ for which*

$$R(p, s) \quad \Rightarrow \quad C'(p) = C(s) \ \wedge \ I'(p) = I(s)$$

is called a port/socket relation *(between $\mathcal{N}'$ and t).* □

A port/socket relation is a relation between the ports of a net and the sockets of a transition. The restrictions to the colour set and initial marking ensures that we can fuse ports and sockets.

In the following we use $\biguplus_{i=1,\ldots,n} A_i$ to mean the disjoint union of sets $\{A_i\}_{i=1,\ldots,n}$. We assume we have new symbols $\mathbf{1}, \mathbf{2} \ldots$, one for each $n \in \mathbb{N}$, and define $\biguplus_{i=1,\ldots,n} A_i$ to mean the set $\bigcup_{i=1,\ldots,n} \{ (\mathbf{i}, a) \mid a \in A_i \}$. We define for each $\mathbf{i}$ a function $\mathbf{in}_i \,:\, A_i \to \biguplus_{i=1,\ldots,n} A_i$ by $\mathbf{in}_i \, x = (\mathbf{i}, x)$ to map elements in A_i to the copy in the disjoint union.

Using disjoint union ensures, among other things, that we can have more than one instance of the same nets.

Definition 16 (Transition substitution). *Let $\mathcal{N} = (P_1, T_1, A_1, N_1, C_1, G_1, E_1, I_1, Ports_1, TS_1)$ be a PCPN and let Θ be a partial mapping $\Theta \,:\, dom(TS_1) \rightharpoonup \big(Pow(\mathbf{P} \times \mathbf{P})\big) \times \mathbf{PCPN}$ such that for all $t_i \in dom(\Theta)$ with $\Theta(t_i) = (R_i, \mathcal{N}_i)$, R_i is a port/socket relation between $\mathcal{N}_i = (P_i, T_i, A_i, N_i, C_i, G_i, E_i, I_i, Ports_i, TS_i)$ and t_i.*

We can substitute *with Θ and get the net $\mathcal{N}[\Theta]_{TS}$ constructed in the following way: Let $\mathcal{N}^* = (P^*, T^*, A^*, N^*, C^*, G^*, E^*, I^*, Ports^*, TS^*)$ be defined by*

1. $P^* = \biguplus_{i=1,\ldots,n} P_i$
2. $T^* = \big(T_1 - dom(\Theta)\big) \uplus \biguplus_{i=2,\ldots,n} T_i$
3. $A^* = \big(A_1 - A\,(dom(\Theta))\,\big) \uplus \biguplus_{i=2,\ldots,n} A_i$ *where $A\,(dom(\Theta))$ denotes the set of arcs connected to any transition in $dom(\Theta)$*
4. $N^* \,:\, A^* \to (P^* \times T^* \cup T^* \times P^*)$ *defined by* $\mathbf{in}_i \, a \mapsto \big((\mathbf{in}_i \times \mathbf{in}_i\,) \circ N_i\big)(a)$
5. $C^* \,:\, P^* \to \mathbb{T}$ *defined by* $\mathbf{in}_i \, p \mapsto \mathbf{in}_i \, C_i(p)$
6. $G^* \,:\, T^* \to \mathbb{E}$ *defined by* $\mathbf{in}_i \, t \mapsto G_i(t)$
7. $E^* \,:\, A^* \to \mathbb{E}$ *defined by* $\mathbf{in}_i \, a \mapsto E_i(a)$
8. $I^* \,:\, P^* \to \mathbb{E}$ *defined by* $\mathbf{in}_i \, p \mapsto I_i(p)$
9. $Ports^* = \mathbf{in}_1 \, Ports_1$
10. $TS^* \,:\, T^* \rightharpoonup \mathbf{NV} \times Pow(P^* \times \mathbf{PN}) \times \mathbf{TASS} \times \mathbf{EASS} \times \mathbf{NASS}$ *defined by* $\mathbf{in}_i \, t \mapsto TS_i'(t)$ *if* $t \in dom(TS_i)$, *where* $TS_i'(t) = \big(net, \{ (\mathbf{in}_i \, p, pn) \mid (p, pn) \in R\}\big)$ *when* $TS_i(t) = (net, R)$

Define on P^* the relation F as the smallest equivalence relation satisfying $F(\mathbf{in}_i \, p, \mathbf{in}_1 \, s)$ if $R_i(p, s)$.

Thus defined, F will relate all ports related to at least one common socket, and all sockets related to at least one common port, and the closure of this. F will be an equivalence satisfying the properties needed for a fusion set. Thus we can create the net $\left[\mathcal{N}^*\right]_F$. Let $\mathcal{N}[\Theta]_{TS}$ be this net. □

In short, a transition substitution creates copies for each instance of nets, and fuse ports and sockets related by some port/socket relation.

We relate the substitution mapping TS to net assignments Θ in the following way. Whenever we have PCPN $\mathcal{N}$ and net assignment Θ we can define $\Theta^* \,:\, T \rightharpoonup Pow(\mathbf{P} \times \mathbf{P}) \times \mathbf{PCPN}$ by

$$t \mapsto \begin{cases} (R' \circ R, \mathcal{N}^*) & \text{when } t \in dom(TS) \text{ and} \\ & TS(t) = (net, R, \Gamma, \Delta, \Theta') \text{ and} \\ & net \in dom(\Theta) \text{ and} \\ & \Theta(net) = (R', \mathcal{N}') \text{ and} \\ & \mathcal{N}^* = \Big((\mathcal{N}'[\Gamma]_T)[\Delta]_E\Big)[\Theta']_N \\ & \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here $\mathcal{N}'[\Theta]_N$ is net $\mathcal{N}'$ after net assignment as defined in Def. 17.

Please also notice the order of assignments in $\Big((\mathcal{N}'[\Gamma]_T)[\Delta]_E\Big)[\Theta']_N$. Expression substitution must always occur after type substitution, to be able to specify values of specific types, and still obey type-rules upon substitution. Placing net substitution last ensures that type and expression substitution only takes place through the substitution mappings in Sub.

Notice that the mapping Θ^* defined in this way matches the mapping from Def. 16.

Intuitively, TS maps substitution transitions to variables while Θ maps variables to nets. Θ^* is the composition that links substitution transitions to nets.

Intuitively TS relates each substitution transition to a *signature* consisting of the net variable and the naming of the sockets. The net assignment Θ then relates a signature to a specific net, by associating the net variable with a net and naming the ports. This is illustrated in Figure 13. On the left we see a substitution transition with two sockets p and q. TS maps this transition to the variable $\mathcal{N}$, and relates the socket p to the name A and socket q to the name B. The signature is drawn on the dashed line. On the right we see a net with two ports. In the figure Θ maps variable $\mathcal{N}$ to this net, and relates the upper port to name A and the lower port to name B. Θ^* can be thought of as the composition of the arrows in the figure.

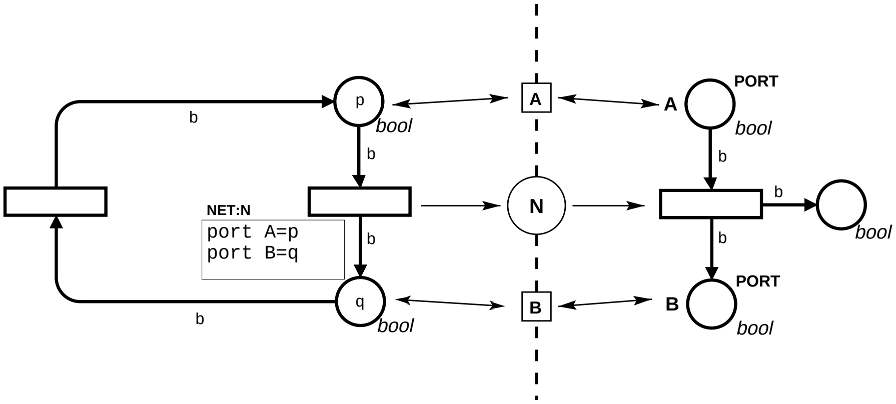


Fig. 13. Composition of TS and Θ

Missing from the figure is the assignments. In this example all assignments are empty, but in general assignments will take place before gluing the nets together.

We will only consider *legal* net assignments, that is, for PCPN $\mathcal{N}$ and net assignment Θ: For $t \in dom(\Theta^*)$ and with $\Theta^*(t) = (R, \mathcal{N}^*)$ the relation R is a port/socket relation between $\mathcal{N}^*$ and t. The signatures used in the examples in Sect. 2 are not part of the definition, but are used to ensure that all net assignments considered are *legal*.

Definition 17. *For PCPN $\mathcal{N}$ and net assignment Θ we define the the net $\mathcal{N}[\Theta]_N$ to be $\mathcal{N}[\Theta^*]_{TS}$.*

Notice that Def. 17 is a recursive definition, since it refers to the function Θ^* which refers to net assignments.

4 PCPN Modules

As defined, PCPN are a kind of *non-hierarchical* CPN with parameters. It could just as well have been defined as a *hierarchical* CPN [5] with parameters. Parameterisation is really orthogonal on the hierarchy concept. There is, however, no need to add hierarchical nets to PCPN, since net parameterisation leads naturally to a module system for PCPN.

We will think of *modules* as self-contained units with a well-defined signature, and constructions for combining modules into other modules. This is exactly what we have with net parameters in PCPN. As mentioned in Sect. 2 we can talk about the signature of a PCPN, defined from the parameters and the set of ports. Likewise we can talk about the signature of a substitution transition defined by the *NET* region, i.e., the type, expr, net, and port assignments there. Whenever the signature of a net matches that of a substitution transition we can substitute the transition with the net and get a new net.[3]

Nets with signatures work in much the same way as abstract data types. We hide the implementation (the actual net) and only access the functionality through the interface, given by the signature. This allows us to refine one module, or even replace it with a new module, without affecting any other module using the first module.

With Parameterised CPN we can make very general modules, or libraries of modules, and such libraries can then be used in several different models.

Consider a communication protocol, or a protocol suite. Each protocol layer can be modelled as a PCPN with a number of parameters, e.g., a net parameter for the lower layers, a type parameter for the data to be transmitted, and a number of expression parameters for, say, timeout interval or packet sizes. The modules can then be combined in various ways to model different protocol configurations. Furthermore, a library of protocols can be used when modelling distributed applications. Once the protocol library is modelled and validated it can be used in a number of different models.

Tools working with PCPN should of course be built to take advantage of the underlying modularity. For example, tools that generate code from PCPN for simulation, should translate nets into separate units, such that making changes to one net only leads to re-compilation (or re-code-generation) of that particular net. When we write programs in high-level languages, we expect to be able to compile each unit separately, and then link the pieces together. The same should be possible for PCPN.

5 Implementation

The modules as described in the previous section can be realized through Standard ML's module system in a straight forward manner. In this section we will indicate how this can be done. For a discussion of a "proof-of-concept" implementation of a simulator of the kind described in the following, we refer to [7].

We have chosen SML as the implementation language for its module system which closely match our requirements. SML modules consist of *structures*, *signatures*, and *functors*. Structures are used to package up related types, values, and functions. Signatures are then used to specify what components a structure must contain. A *functor* allows structures to be parameterised. In the following we will consider how to translate PCPN modules into SML structures, which can be used for simulation and analysis of the modules.

Let us return to the examples in section 2. The first module we considered was the counter shown in Fig. 14. We will refer to this module as $\mathcal{C}$. As we saw in section 2, we can think of the type T as a parameter to the module, as indicated in the *PARAMETERS* box. Furthermore we can think of the two port places A and B as parameters. A net using this module will have to specify both the type T and the places to be related to A and B. Net $\mathcal{C}$ should therefore be translated into an SML functor which takes T, A, and B as parameters and implements the behaviour of the net.

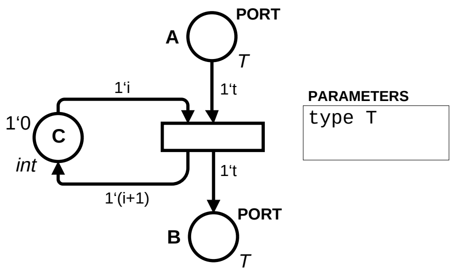


Fig. 14. $\mathcal{C}$ - Counting transition firings.

[3] In the formal definition of substitution (Def. 17 in the previous section) we only allowed *legal* net assignments. The matching signatures are one way of ensuring this.

A general PCPN module consists of the following components:

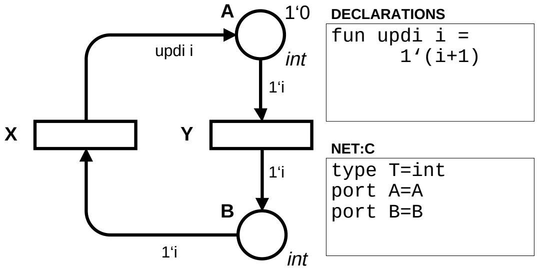
Parameters. Both the explicit parameters, e.g., the type T in Fig. 14, and the ports, e.g., the two places A and B in Fig. 14.

Declarations. Any CPN module can have a number of declarations local to that particular module. We want modules to be self-contained units, so obviously any declaration used in a module should be local.

The actual net. This is the most important part for determining the behaviour of the module, and will determine most of the code for executing the module.

Sub-modules. A module can contain a number of sub-modules. The sub-modules correspond to substitution transitions, or rather, the nets that substitutes the transitions.

Fig. 15 is a modified version of the module in Fig. 2. It is modified in the way that the arc inscription on the arc from X to A has been changed to a call of a function, defined in a *DECLARATIONS* box. This is an example of a module with no parameters, with a non-empty set of declarations, and a non-empty set of sub-modules. Denote this module by $\mathcal{N}$.



```
fun updi i =
      1'(i+1)
```

```
type T=int
port A=A
port B=B
```

Fig. 15. $\mathcal{N}$ - Counting transition firings.

This particular module contains one sub-module, which is an instance of $\mathcal{C}$, the module in Fig. 14. The module should thus be translated into a SML structure with one sub-structure, corresponding to the module $\mathcal{C}$. By implementing $\mathcal{C}$ as a functor, we get a parameterised structure in SML, and instantiating the sub-module of $\mathcal{N}$ is a matter of instantiating the SML functor with the appropriate parameters, as specified in the *NET* region.

5.1 Translating PCPN modules into SML structures

Before we can decide how to translate a net module into an SML structure, we will have to decide how the structure is going to interact with the tool using the module. In the following we will assume the modules are going to be used in a simulator. Using the modules in other tools, like say a state space tool, should work in a similar fashion.

We will assume that we have a simulator which is responsible for scheduling transition firings, and one global state. Each module will have functions responsible for updating the global state when a firing occurs, and for adding transitions to the scheduling queue when needed.

In general this will work as follows: Initially all transitions will be examined, and the enabled transitions will be added to the scheduling queue. When a transition is scheduled, an enabled binding element will be chosen randomly (if such a binding exists), and the transition will fire and update the state. Changing the state will possibly result in a number of transitions becoming enabled. Each candidate for this will be added to the scheduling queue. We will denote the combination of the selection of binding element, updating state, and scheduling new candidates, as an *event*. We will henceforth refer to the scheduling queue as the *event queue*.

All the actions associated with an event, can be encapsulated in a function or a record of functions, stored in the event queue. Once the initial events are installed in the event queue, the scheduler will only be required to select and execute the next event, if any. No further knowledge of the modules is required.

We will therefore use the signature in Fig. 16 for a PCPN module. Here the function *init* is responsible for initialising the module, i.e., adding all initial events to the event queue. The *init* function is also responsible for initialising any sub-modules, by calling their *init* function.

```
signature PCPN_MODULE =
  sig
    val init : unit -> unit
  end
```

Fig. 16. Signature for PCPN module

A general PCPN module as described above can then be translated to an SML functor of the form seen in Fig. 17.

```
functor M((* Parameters *)) : PCPN_MODULE =
  struct
    (* — Declarations — *)

    (* — Code for Handling Events — *)

    (* — Sub-modules — *)

    (* — Module Initialisation — *)
    fun init () = ((* Initialisation code *))
  end
```

Fig. 17. Template for PCPN module functor

The functor generated for the module $\mathcal{C}$ (in Fig. 14) is shown in Fig. 18. In the figure, only the code related to parameterised modules is shown, and the code for interacting with the simulator is omitted. The interesting part in the figure is the parameter part of the functor. The parameters consists of the type parameter T, and the two port places A and B. T is an SML type parameter, while A and B are structures implementing places.

We construct the parameter part of the functor from the *PARAMETERS* box of the net, and we do this in a very straightforward manner. We translate certain PCPN keywords into SML keywords, and insert the rest verbatim into the SML functor. The keywords are translated as follows: *port* is translated to *structure*, *expr* is translated to *val*, and *net* is translated to *functor*. We will return to the latter shortly. This preprocessing is done solely for allowing PCPN keywords in declarations. We could do without it by using the SML keywords. Places marked as ports are also inserted in the parameter part. These are structure parameters with signature *PLACE*.

The last line in the parameter part of the functor in Fig. 18 specifies that the colour set of the port places A and B is the same as the parameter type T. This is implicitly assumed from the colour set specified in the net inscriptions. The line is needed because we implement places as structures, each with a colour set (type) C.

Figure 19 shows the functor generated for the module $\mathcal{N}$ in Fig. 15. As can be seen, $\mathcal{N}$ takes no parameters. In the net there is no *PARAMETERS* box, and in the functor, the parameter part is empty. There is, however, a *DECLARATIONS* box in the net. The declarations there have been inserted verbatim into the functor. The declarations are put at the beginning of the functor, before any code that could possibly need it. In some cases it might be necessary to pre-process a *DECLARATIONS* box before inserting the declarations

```
functor C(type T
          structure B : PLACE
          structure A : PLACE
          sharing type T = B.C = A.C) : PCPN_MODULE =
    struct
        (* <<snipped open of needed structures>> *)

        (* <<snipped code for handling events>> *)

        fun init () =
            ((* <<snipped code>> *))
    end
```

Fig. 18. Functor for module $\mathcal{C}$.

```
functor N((* none *)) : PCPN_MODULE =
    struct
        (* <<snipped open of needed structures>> *)

        (* DECLARATIONS *)
        fun updi i = 1`(i+1)

        (* <<snipped code for handling events>> *)
        structure Y = C(type T=int structure A=A structure B=B)

        fun init () =
            ((* <<snipped code>> *))
    end
```

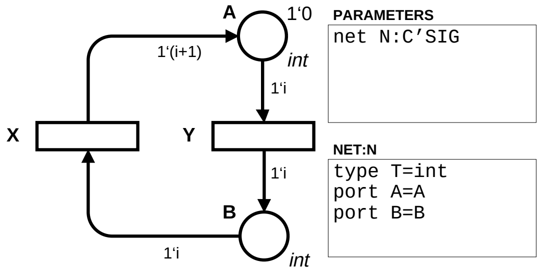
Fig. 19. Functor for module $\mathcal{N}$.

into the functor in the same way as we process $PARAMETERS$ boxes. We will assume this is not necessary for the simulator we are considering.

Notice the sub-module in Fig. 19. This is simply implemented as an SML structure, created by a call to the appropriate functor. In this case the functor C from Fig. 18. The parameters used in the functor call are created from the NET box in the same way as the parameters for the functor C were created from the $PARAMETERS$ box.

In Fig. 19 the functor used to create the sub-module must be globally known, since it is referred to simply by a name. We want to be able to handle net parameters in much the same way, but where we let the functor used to create the module be a parameter of the functor that contains the sub-module. To implement this we can use higher order functors as implemented in SML/NJ [1].

Consider the net in Fig. 20, and denote it $\mathcal{M}$. The functor created for this net is seen in Fig. 21. Here M takes as argument a functor N and creates the sub-module Y with this functor. The functor N is typed with the functor signature $C'SIG$. This signature is created automatically from the net $\mathcal{C}$ we saw in Fig. 14. The definition of $C'SIG$ is shown in Fig. 22.



```
net N:C'SIG
```

```
type T=int
port A=A
port B=B
```

Fig. 20. $\mathcal{M}$ - Net with net parameter.

6 Future work

CPN has powerful and general methods for both validation and verification. Validation is concerned with convincing ourselves that a net behaves as intended, while verification is concerned with proof that a net has a formally stated property. The obvious next step for PCPN is to extend these methods to handle parameters. This is necessary to be able to examine modules independently, and to exploit the underlying modularity.

6.1 Validation

Validation is usually done through simulation. In Sect. 5 we examined how to implement a simulator for PCPN. The discussion, however, was only concerned with how to translate PCPN into SML code, and was not concerned with how to actually simulate the nets. With our prototype simulator [7] we can only simulate instantiated nets, and we have only *defined* the behaviour of nets without free type and expression parameters.

Modules, however, are usually self-contained units, and we must expect that modules are primarily developed independently, thus we should be able to validate modules independently. At least, to as high a degree as possible. To do this we have to come up with a meaningful definition of the behaviour of a parameterised net, and tool support for this.

6.2 Verification

The primary verification techniques for CPN are state spaces and invariants [6]. Work has been done on exploiting modularity in verification with both techniques in [3] and [4].

The state space method relies heavily on the initial marking of the net in question, and it is thus hard to imagine verification independent of instantiation. Symbolic state spaces, however, could turn out to make this possible in some cases.

Invariants rely on the structure of nets rather than the initial marking, and offers perhaps more promise as a verification technique for PCPN. Type parameters will not influence invariant properties directly, but will determine the weight functions. Expression and net parameters will in general affect properties, but by putting restrictions on the possible nets and expressions assigned to parameters we might be able to prove certain properties.

7 Conclusion

In this paper we have formally defined PCPN as CPN with type, expression, and net parameters. We have seen how net parameters lead to a module system for PCPN and examined how a simulator can be implemented such that code for separate modules can be generated independently, and such that instantiated nets can be simulated.

Now, the next step is to create validation and verification techniques for handling parameterised nets individually.

Acknowledgements

Thanks to Søren Christensen, Bo Lindstrøm, Regnar B. Lyngsø, Kjeld H. Mortensen, and Lisa Wells whose discussion and comments helped improve this paper.

References

1. Special Features of SML/NJ. WWW online documentation <URL:`http://cm.bell-labs.com/cm/cs/what/smlnj/doc/features.html`>.
2. Søren Christensen and Kjeld H. Mortensen. Parametrisation of Coloured Petri Nets. Technical Report DAIMI PB - 521, Department of Computer Science, University of Aarhus, March 1997.
3. Søren Christensen and L. Petrucci. Modular state space analysis of coloured petri nets. In *Proceeding of the 16th International Conference on Application and Theory of Petri Nets, Turin*, pages 201–217, June 1995.
4. Søren Christensen and Laure Petrucci. Towards a modular analysis of coloured petri nets. In *Proceeding of the 13th International Conference on Application and Theory of Petri Nets, Sheffield, UK*, volume 616, pages 113–133. Springer-Verlag, June 1992.
5. Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. - Volume 1: Basic Concepts.* Monographs in theoretical computer science, Springer-Verlag, Berlin, 1992.
6. Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. - Volume 2: Analysis Methods.* Monographs in theoretical computer science, Springer-Verlag, Berlin, 1994.
7. Thomas Mailund. A Simulator for Parameterized CPN Modules. <URL:`http://www.daimi.au.dk/~mailund/pcpn.html`>.
8. Thomas Mailund. Formal Definition of Parameterized CPN. Technical report, Department of Computer Science, University of Aarhus, 1999. Internal Technical Report, <URL:`http://www.daimi.au.dk/~mailund/papers/definitions.ps`>.
9. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.
10. L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 2nd edition, 1996.

```
functor M(functor N:C'SIG) : PCPN_MODULE =
    struct
        (* <<snipped code>> *)

        structure Y = N(type T=int structure A=A structure B=B)

        fun init () =
            ((* <<snipped code>> *))
    end
```

Fig. 21. Functor for module $\mathcal{M}$.

```
funsig C'SIG (type T
              structure B : PLACE
              structure A : PLACE
              sharing type T = B.C = A.C) = PCPN_MODULE
```

Fig. 22. Functor signature for net $\mathcal{C}$.

A CPN model of the MAC layer

Pablo Hernández Morera, Tomás M. Pérez González
Departamento de Ingeniería Telematica
University of Las Palmas de Gran Canaria,
Las Palmas de Gran Canaria, Spain

e-mail:pablo@cma.ulpgc.es

Abstract

This paper describes the design and validation of a CPN (Coloured Petri Net) model for the MAC layer based on the IEEE 802.3 standard. The main purpose of this system is to teach the TE students at this university how Coloured Petri Nets can be used in the modeling and analysis of real systems (especially those based on concurrency), allowing them to extract net modeling and analysis techniques. Furthermore, it also aims to introduce Design/CPN as an adequate tool for the edition, simulation and validation of CPN models.

A bus LAN has been modelled. The model comprises a set of stations (each of them with the MAC layer implemented inside it) and a common channel which connects them all.

By means of this model it is aimed to analyse how the MAC layer allocates use of the shared communication medium among the competing nodes. The basic idea is to make the stations transmit packets in bursts to the common noisy channel using the CSMA/CD access method and analyse how they adquire the medium and recuperate from collisions. This access method where several stations may try to transmit their packets concurrently creates the perfect environment for the use of Coloured Petri Nets because of the natural way they include concurrency in their structure.

1 Introduction

This paper describes the designing and validation processes for the MAC layer of a bus LAN using the CSMA/CD access protocol. A previous work on modeling LANs using Petri nets can be found in [Goo87]. The Petri nets in this paper were created and analysed by Design/CPN tool [Ref93].

The basic purpose of the MAC layer is to allocate use of the shared communication medium among the competing nodes coordinating the communication among them by means of the CSMA/CD protocol [Jes94] [Tan] [Joh96]. According to this protocol, when a station generates a new packet, it first senses the broadcast channel to detect the presence of any ongoing transmission. If the channel is sensed idle, the packet transmission is started after a short delay called “interframe spacing”. If instead, the channel is sensed busy, the transmission is delayed until the channel becomes idle. During the transmission the channel is monitored to detect the presence of any interference from another transmitting station (collision). If a collision is detected, the transmission is immediately stopped, and the channel is jammed for a short time to make sure that all stations recognize the collision, and the packet transmission is rescheduled at some later time, after a random delay. By using a random delay, the stations which are involved in the collision are not likely to have another collision on the next transmission attempt. However, to ensure that this backoff technique maintains stable, a method known as *truncated binary exponential backoff* is used. In this method a station will persist in trying to transmit when there are repeated collisions. These retries will continue until either the transmission is successful or 16 attempts (the original attempt plus 15 retries) have been made unsuccessfully. At this point (if all 16 attempts fail) the packet is discarded.

The backoff strategy is quite simple. If a packet has been transmitted unsuccessfully *n* times, the next transmission attempt is delayed by an integer r times the base backoff time (which is often chosen to be twice the end-to-end propagation delay). The integer r is selected as a uniformly distributed integer in the range $0 \leq r \leq 2^k$ where k =min(n, 10), that is, k is the minimum of the number of presently attempted transmissions and the integer 10. Thus as the load becomes increasingly heavy, the stations automatically adapt to the load.

The global model has been constructed taking advantage of the hierarchical CPN capabilities [Jen92][Jen94][Jen97a][Jen97b]. Hierarchical CPN allow the construction of a large model as a set of smaller models connected to each other using well-defined interfaces (substitution transitions). In this way, a complex model like this can be reduced to the generation of smaller models which solve certain functions in the global model into which they are integrated. The model aims to be as detailed as possible, i.e., it tries to collect the whole of the functions and aspects included in the MAC layer. The size of the resulting model is proportional to the adopted level of detail.

The main drawback of such a modeling is the risk of a state space explosion. This phenomenon makes difficult the validation of the model forcing to master the size and the level of detail and trying to find an equilibrium between these two opposite factors: detail and model size vs size of the state space and easy validation.

Since the system model is rather complex -its full net specification requires some 23 pages and a total of about 200 places- this paper merely attempts to cover some basic features of its construction and its validation, but it cannot be very specific on all details. In this sense, some of the component pages of the model have not been included and only main pages have been exhaustively explained.

The paper first introduces the nets structure naming all its parts. Each part is described breaking it down into the basic net components necessary to model it. Finally, the last part of this paper is related to the validation process and the several problems found to achieve it.

2 The CPN model

The following assumptions have been adopted in order to set the basic parameters of the system:

- *Size*. The network comprises six similar stations.
- *Burst arrivals*. In each station there is a LLC (Logical Link Control) user who generates packets in bursts. In this way, it is aimed to model the real behaviour of a network information flow where it is possible to find time intervals where no packets are transmitted and other ones where one or more stations may try to transmit a great volume of information.
- *Buffering*. Each node has a buffer containing packets to be transmitted. The buffer capacity has been limited to one packet in order to simplify the model.
- *Collisions and retransmissions*. Each packet which collides with another must be retransmitted. The maximum number of retransmission attempts for a given packet has been set to 16 (the original one plus 15 retries).
- *Noise*. The channel is not ideal and contains noise. This implies that packets are submitted to two different kinds of errors: those produced by the noise and those produced by

the collision among packets from stations which try to transmit concurrently.

- *Propagation delay*. The propagation delay between two adjacent nodes, *Tr*, has the same value for all adjacent node couples and it is small in comparison with the packet transmission time, *Tp* (all packets have the same length).

Figure 1 shows the page hierarchy graph for the bus network. It can be noted that the CPN model has a very modular structure. Among its pages it is possible to distinguish the page *NET#15*, the prime page. Its content is shown in figure 2. According to the hierarchy, the rest of the net pages depend on it. This forces to a top-down analysis on its hierarchical structure. It comprises six similar stations connected to each other by a common communication channel. Each station is composed of one transition *stationx* with 1 ≤ *x* ≤ 6 where *x* is the identifying subindex for each station and three places associated to it: *nx*, *Cx* and *interfacex*. Each of these transitions (*station1*,..., *station6*) is related to the same subpage (*station#1*). This means that the hierarchical net has six page instances of *station#1*, each one with its own marking which is different from the other page instances markings.

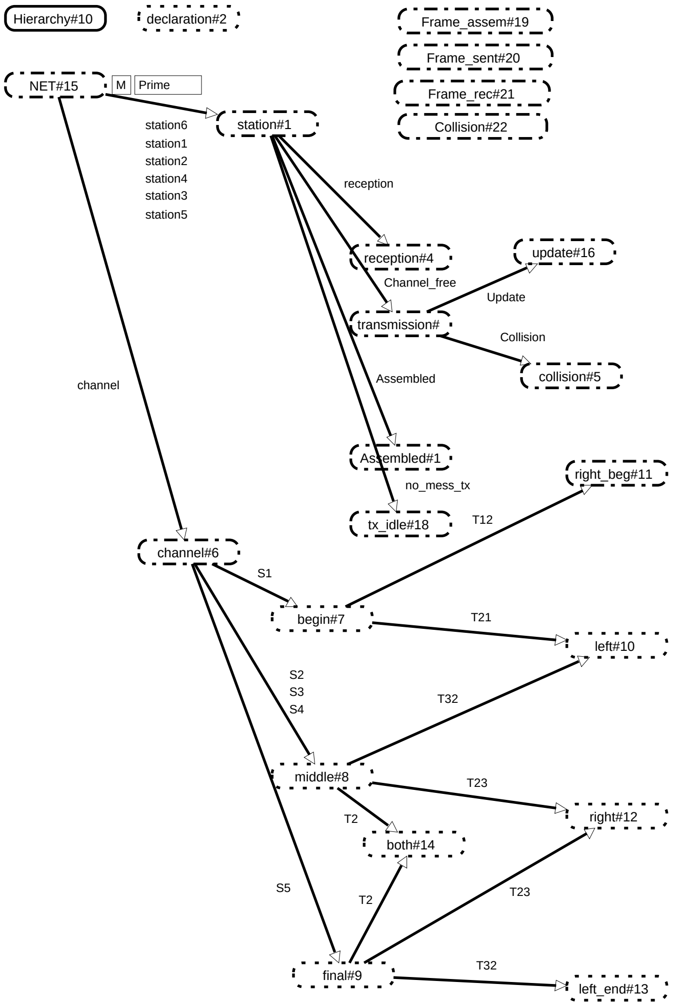


Figure 1. Page hierarchy graph.

As regard to the places:

- *Interfacex* represents the access point of the station *x* to the channel allowing the transmission and reception of packets. The color *Packet* associated to this place must be of the

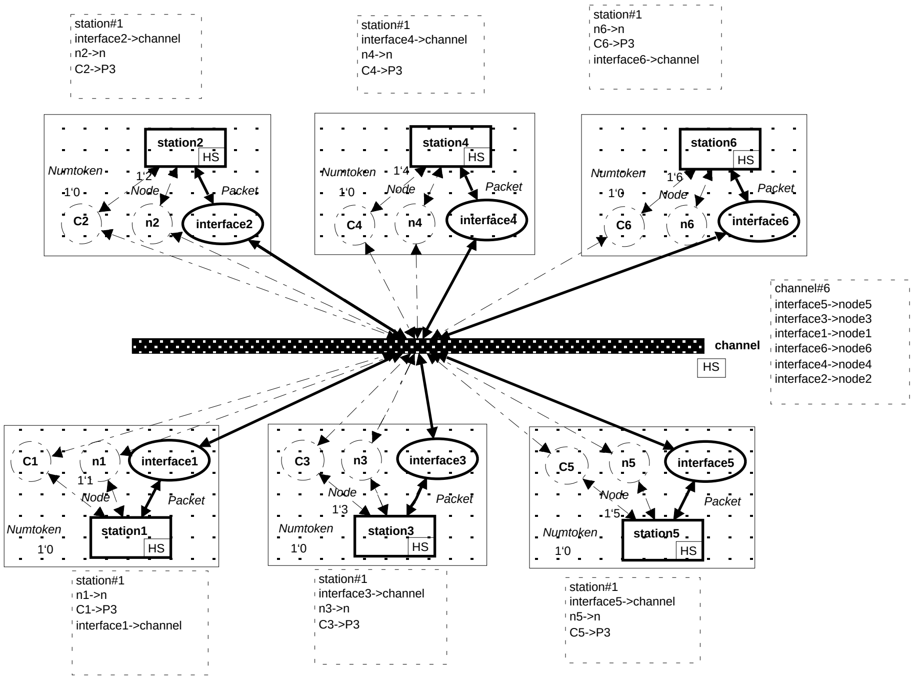


Figure 2. Page NET#15.

general form:

*color Packet=product State*Origin*Destination*CRC timed;*

with

color State=with STX | MTX | ETX | JAM;

color Node=int with 0..maxnodes[1]*;*

color Origin=Node;

color Destination=Node declare ms;

color CRC=int with 0..1;

According to its declaration, *State* can adopt four different values: *STX* (Start Transmission), *MTX* (Middle Transmission), *ETX* (End Transmission) and *JAM* (collision signal), indicating the part of the packet that is being transmitted. *STX*, *MTX* and *ETX* refer to the head, data and end of the packet respectively. *JAM* is a general value to be used when the token which transports it does not represent a packet but a collision signal.

Origin identifies the station which generates the frame, *Destination* indicates the address of the destination station and *CRC* models the frame check sequence (FCS) associated to the information transported by the data field. The data field, as well as other component fields of the basic MAC frame have not been modelled. This is the case of the preamble, PAD, etc.

The word “timed” included in the declaration indicates that this is a timed color, i.e., the tokens belonging to this color will have a time stamp attached to them, describing the earliest model time at which the token can be removed by a binding element.

- *Cx* is a place whose associated token always indicates the number of tokens in the place

[1] *Maxnodes* is a global constant indicating the maximum number of stations. 0 is a broadcast address.

interfacex of the station *x*. This token belongs to the color *Numtoken*, an integer type. Taking the existing tokens in each of the channel access places (and hence, of the channel itself) into account is a very important feature since it allows to know the number of packets being transmitted along the channel. In this way, it is easy to detect:

- The collisions in the channel. For a given station, a collision will be produced when the station is transmitting and the token in *Cx* has a value of two or more.

- The idle channel. This condition must be fulfilled before the transmission starts, and this will happen when the token in this place is equal to 0.

- *nx* is a place associated to the color *Node* which always stores a token establishing the address of the station to which it is attached. This address is an integer number included in the range 0..*maxnodes*.

As in any real bus network, the stations must be connected to each other by means of a common channel in order to communicate. This function is achieved by the substitution transition *channel* which is related to the subpage *channel#6* through the interfaces declared in its hierarchical inscriptions.

3 The channel model

The channel constitutes the physical device which interconnects all the stations in a network. Figure 3 shows the obtained model for the channel. It is made up of 17 places and 5 transitions.

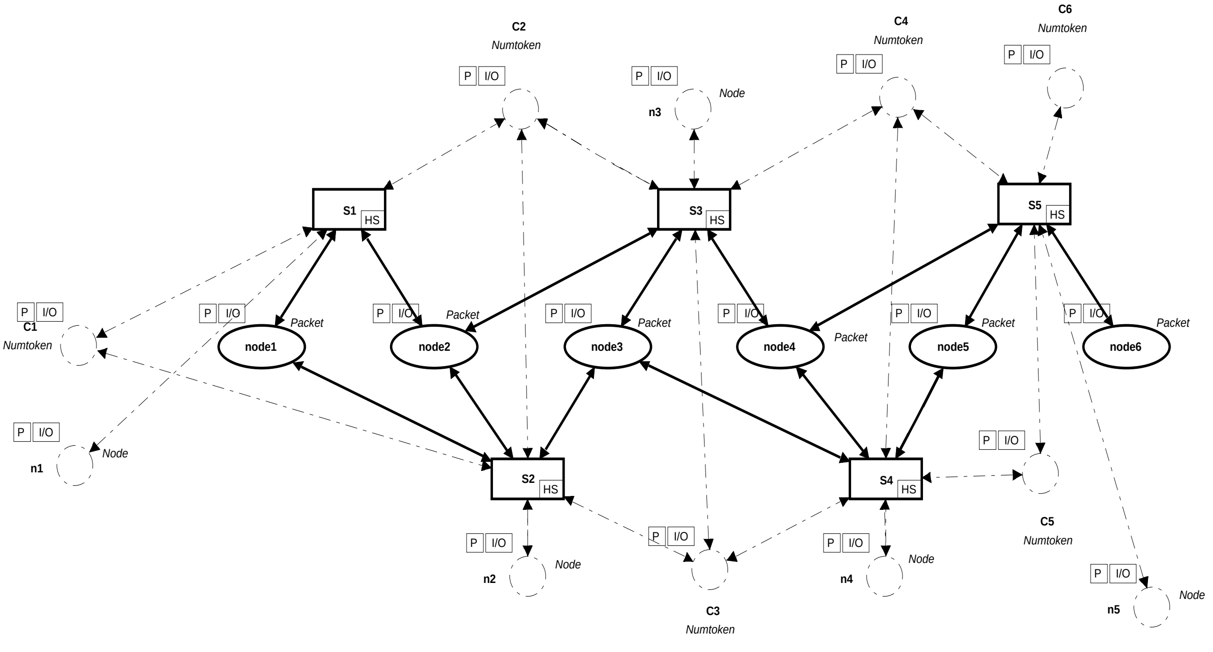


Figure 3. The channel page.

The channel is composed of six main places (*node1*,..., *node6*) modeling the physical points of the channel through which the stations transmit and receive their packets (modelled by means of tokens of color *Packet*) interconnected among them via the transitions *Si* with *i*=1..5 which also connect some control places taking account of the number of tokens in each of these main places and informing about the address of the station to which every one of these main places is attached (*C1*, ..., *C6* and *n1*, ..., *n5* respectively).

The rest of the existing channel between two consecutive stations has been modelled bearing in mind its time parameters only, i.e., the distance between two consecutive stations has been transformed into propagation delays. This implies that the movement of the tokens from a place called *nodex* to another one, *nodex+1*, must be time controlled in order to get a realistic effect in the propagation process of packets along it. The transitions *Si* is in charge of this, at the same time that they update the values of the tokens in the control places. These five transitions, *Si* with *i*=1..5, are substitution transitions in such a way that the transition *S1* is related to the subpage *begin#7*, *S5* to *final#9* and *S2*, *S3* and *S4* to the subpage *middle#8*. The subpages *begin#7* and *final#9* model the extreme segments of the channel, while the subpage *middle#8* models its intermediate segments.

This channel structure as a set of three possible types of subpages, *begin#7*, *middle#8* and *final#9*, allows us to construct a channel with the desired size by only leaving intact the substitution transitions related to the subpages *begin#7* and *final#9* and their associated places and introducing between them as many places and substitution transitions related to the subpage *middle#8* as needed.

3.1 Propagation of a packet

Once the basic channel structure has been explained, a brief introduction about the packet propagation along the channel must be given. This includes an explanation not only about the propagation of those packets correctly transmitted, but also about their propagation in those situations where collisions have been produced. The way in which the noise affects the transmitting packets and how this fact is modelled by means of the CRC field will also be explained.

One of the way of achieving the propagation of a packet along the channel consists of making the transmitting station to deposit at each time interval (with the same value as the propagation delay between two consecutive nodes) a token which moves bidirectionally, at the same time that the previous tokens propagate simultaneously along the channel in their propagation direction. This process will continue until the transmission of the packet is completed. Figure 4 describes this process for a packet.

According to the described process, all the tokens which propagate along the channel don’t need the element *State*, since they are all similar. Although this process is very realistic, it presents a serious drawback: the number of firings required to achieve the transmission of a packet is very high, slowing the simulation down and, at the same time, producing a very large state space. This last consequence leads to a disproportionate growth of the size of the occurrence graph due to the great quantity of firings required and hence, different reachable markings. Furthermore, as this process has been defined, the size of the occurrence graph depends, in a direct way, on the length of both the packet and the channel (as the number of stations per length unit remains constant, this is equivalent to affirm that it depends on the number of connected stations).

This is quite a serious problem, not only because of the simulation slowness but also because a large occurrence graph may be impossible to be analysed (by both visual inspection and standard queries), due to the limitation of RAM memory in our equipment.

The reduction of the state space has been achieved by making its size not to depend on the length of the packets but only on the number of stations connected to the channel (note that it is

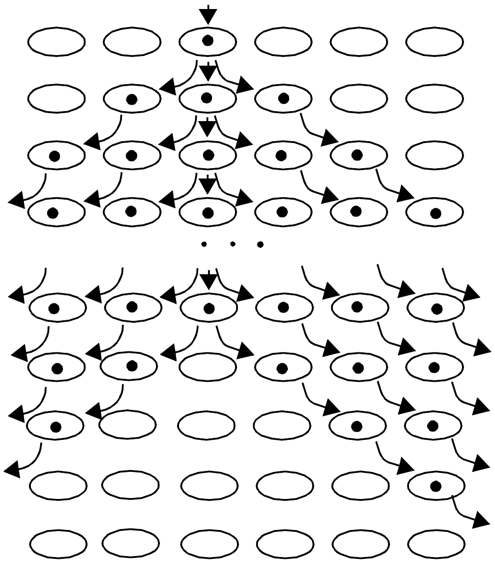


Figure 4. Packet propagation.

almost impossible to make the occurrence graph not to depend on this last factor). Now, firings must not be produced during the whole packet duration but only during the adquisition and liberation intervals, i.e., at the beginning and the end of the transmission of the packet.

Accordingly, it is necessary to distinguish between the beginning and the end of the packet. This is achieved by means of the element *State* in the color *Packet*. Therefore, it is important to emphasize that the introduction of the element *State* in the color *Packet* is not based on the MAC frame structure but on the obtained CPN structure for the channel which forces to differentiate between the beginning and the end of the packets in order to be able to achieve its propagation along the channel in an efficient way, minimizing at the same time the size of the state space.

This forces us to find a new way of performing the packet propagation along the channel. In this way, it is supposed that as soon as the station which wants to transmit senses the channel to be idle, it places in the channel a token with the form (STX,x,y,1), where STX represents the beginning of the transmission (the head is being transmitted), x is the origin address, y is the destination address and 1 indicates that the FCS is in accordance with the information carried by the packet. After a time period *Tr* (delay between two consecutive stations), this token will propagate in both directions to the adjacent places, leaving in the place it occupied a token with the form (MTX,x,y,1) which will remain in this place until the end of the packet is transmitted modeling in this way that in this part of the channel data is being transmitted.

Starting from this moment tokens with *State*=STX will move sideways in their direction of propagation leaving in the places where they move from tokens with *State*=MTX. When these tokens arrive to the end of the channel, they are lost.

The channel remains untouchable with all its places occupied by tokens with MTX until the end of the packet is being transmitted (*State*=ETX). In this moment the token with MTX placed in the place attached to the transmitting station is consumed and a token with *State*=ETX is added to it. This token will propagate in both directions leaving the places it comes from empty, modeling in this way the progressive channel freeing once the packet transmission is ended. This process will continue until the channel becomes idle. As an example, figure 5 shows this process for a packet.

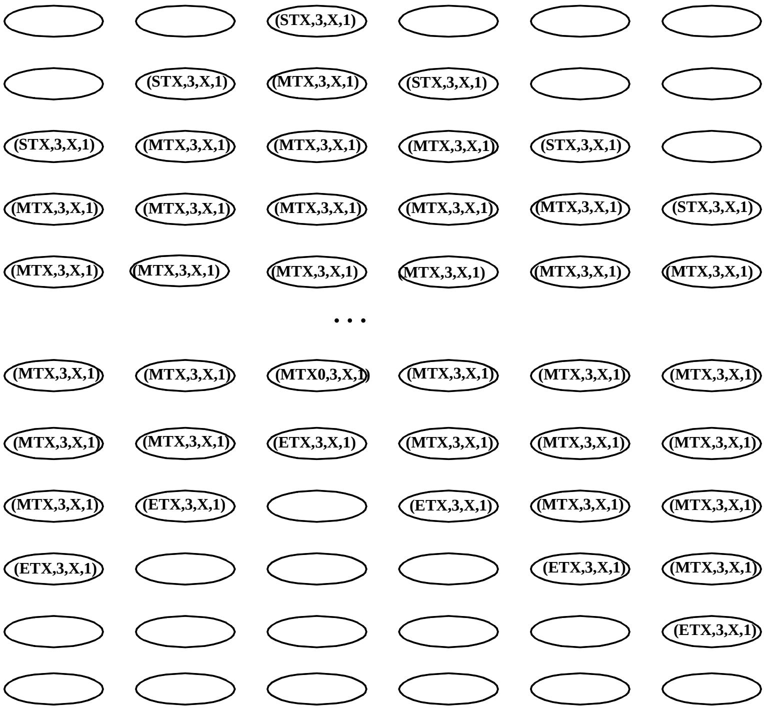


Figure 5. New packet propagation.

This form of propagation minimizes the state space size since it limits the transition firings in the channel to the time intervals needed to acquire and free it. Furthermore, there is another important reason justifying the introduction of *State* in the color *Packet*: it facilitates the modeling of the frame reception in the station reducing it to only detect the first and last tokens of a transmitting packets as it will be later explained.

3.2 Noise effect

As it has been already said, the channel is not ideal and the presence of noise may affect the packets. To model this effect it will be supposed that there is a probability of packets to be affected by this phenomenon and that all affected packets will be discarded. It is only left to be defined the way in which noise will be modelled in the net.

Although any part of the packet may be affected by noise (head, data or end), in order to simplify the model, it will be supposed that such effect will always be noticed at the end of the packet (token with *State*=ETX). It is equivalent to affirm that the probability of any packet to be affected by noise is condensed in its final part in order to be easily detected by the transitions in our obtained model.

A transmission will be considered to be noise free if the FCS received by the receptor during all the transmission is always equal to 1. If its value changes from 1 to 0 it will be supposed that it has been affected by the noise and it will be thrown away.

3.3 Propagation of several packets simultaneously: collisions

Once a packet propagation has been defined, it is only left to establish the way in which collisions are signposted and how the jamming signal (*State*=JAM) will propagate along the channel.

When two or more stations transmit their packets simultaneously to the channel, they will propagate in accordance with the method established in section 3.1 until the moment in which their signals reach the other transmitting stations. In this moment the stations detect the collision, stop their packet transmission and transmit to the channel a jamming signal with the form (JAM,x,0,1) where JAM indicates collision, x is the origin address and 0 is a broadcast address. The collision signal will propagate along the channel.

4 The station model

Figure 6 shows the obtained model for the MAC layer of the station (page *station#1*).

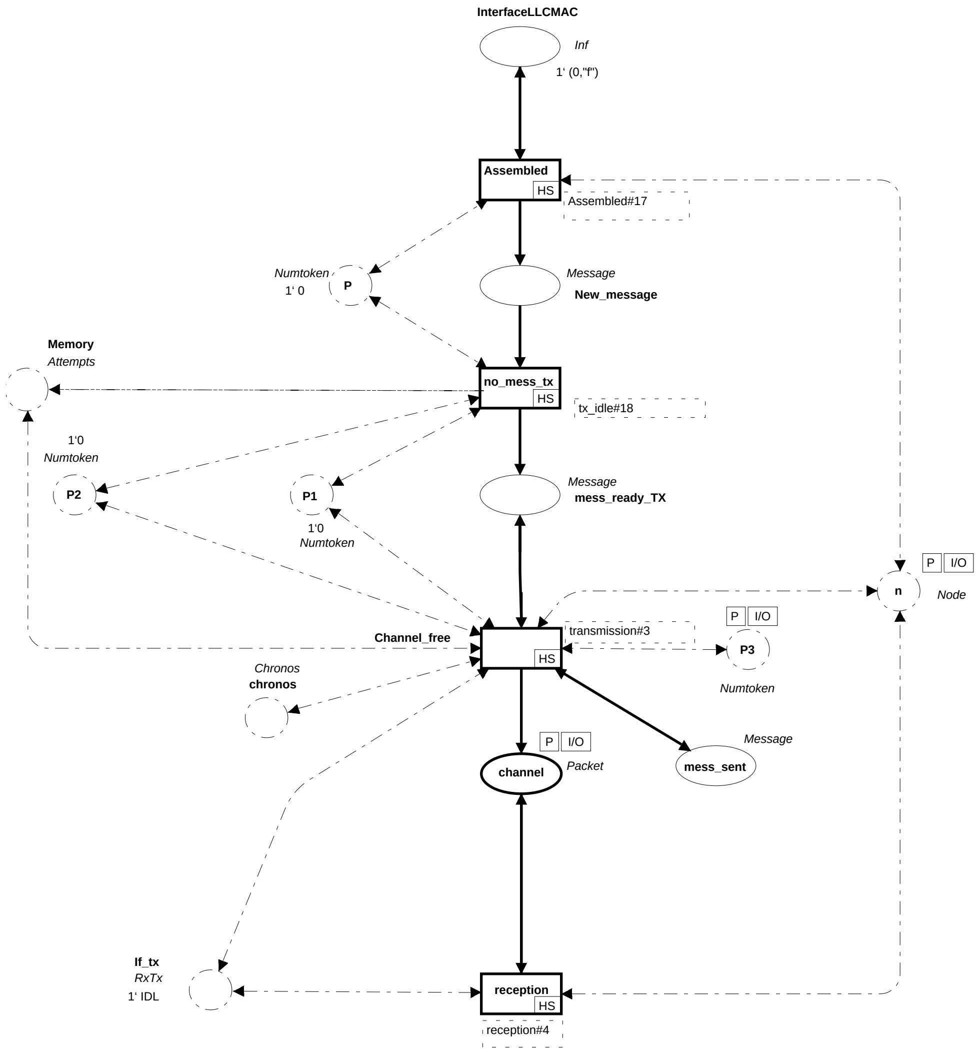


Figure 6. The station (station#1).

Information being passed from the LLC layer to the MAC layer can be found on the place *InterfaceLLCMAC* which acts as an interface between these two layers. This place stores a token of color *Inf* whose declaration is of the general form:

color Times=int with 0..5;

color Prev= string;

*color Inf= product Times*Prev;*

Their interpretation will be discussed later.

Each firing of *assembled* (related to the subpage *Assembled#17* whose content is shown in figure 7) firstly generates a new message which is stored in the place *New_message* that acts as the system buffer with capacity for one message, secondly updates the token in place *P* which acts like a counter of the number of tokens in *New_message* and thirdly sends back to *InterfaceLLCMAC* a token whose value is determined by the function *value*.

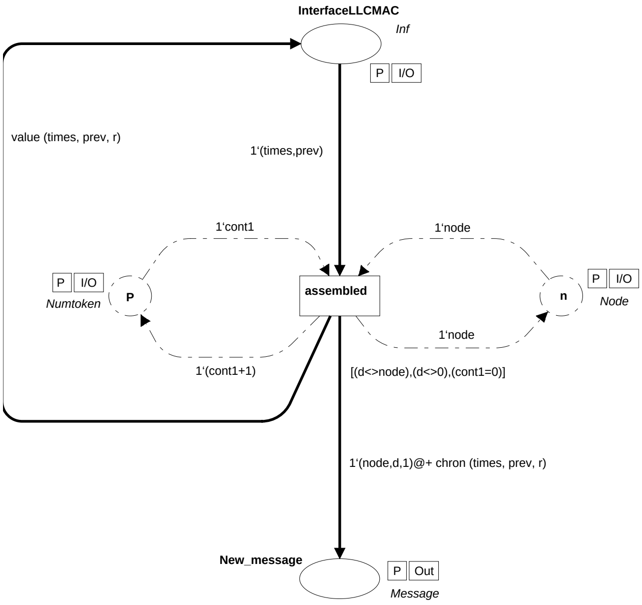


Figure 7. Subpage Assembled#17.

Since the buffer capacity is limited to one message, this transition will only fire when the buffer is empty, i.e., the token in *P* is equal to 0.

The first element of the new message is an origin node number which is extracted from the information carried by the token in place *n*. The second element identifies the destination station. Its value is fixed by the variable *d* which does not appear in the input arc inscriptions. This implies that *d* may adopt whatever value of its corresponding type, i.e., from 0 to *maxnodes*. A guard ([*d<>node*], [*d<>0*]) has been attached to the transition in order to prevent this variable to adopt the same value as the origin address (a station is not allowed transmit a packet to itself) or the broadcast address (0, reserved for collisions). During the simulation, the simulator will provide this random value automatically. The third element will be fixed to 1 to model the fact that the FCS is in accordance with the information carried by the assembled frame.

This frame generation must follow a burst function. This is done using the information carried by the token in the place *InterfaceLLCMAC* and the time stamp attached to the token added on the place *New_message*.

From a mathematical viewpoint this function has been obtained by combining three different random functions: two negative exponential distribution functions whose intensities, n_1 and n_2 (with $n_1 >> n_2$) set the time separation among frames and a third function *r* with a uniform distribution determining the pass from one exponential distribution function to another. Figure 8 shows the mathematical function structure, where *p* is the probability for generating packets with intensity n_1 and *1-p* with intensity n_2.

In order to obtain a noticeable burst effect, this last variable will be examined each five generated messages. This account will be taken by the first element of the token in *InterfaceLLC-MAC* (*times*).

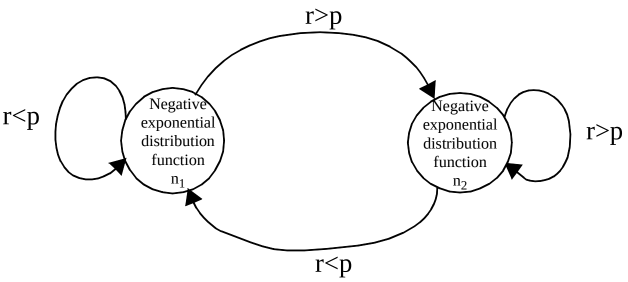

r>p
r<p
Negative exponential distribution function n_1
Negative exponential distribution function n_2
r>p
r<p

Figure 8. Mathematical structure.

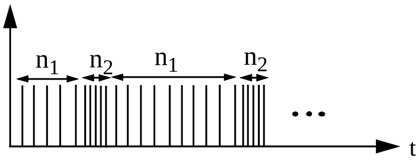


Figure 9. Frame generation.

The random variable determining the pass from an exponential distribution function to the other is the variable *r* we can find in the output arcs of the transition *assembled* arriving to the places *New_message* and *InterfaceLLCMAC*. Its value is automatically fixed by the simulator in each occurrence of *assembled* and together with the variables *times* (indicating the number of generated frames since the last inspection of *r*) and *prev* (indicating the exponential function which is being used) fix the time stamp for the token added in *New_message* by the function *chron* and the token added in *InterfaceLLCMAC* fixed by the function *value*.

Chron was implemented using random number generators. The distribution functions used have been implemented on the top of the new random function. This random function gives samples from a standard uniform distribution. The different distribution functions are then implemented by applying different procedures to this random function. In this way, the negative exponential function was extracted from [The97] and used to obtain our functions.

Once the new frame is generated, it will remain in *New_message* until no previous frame is trying to be transmitted. In this sense, the transition *no_mess_tx* in page *tx_idle#18* (figure 6) inspects the stored tokens in *P*, *P1* and *P2* only firing in the case that *P* has a token with value 1 (indicating the existence of a message in the buffer) and the places *P1* and *P2* (which take account of the number of tokens stored in the places *mess_ready_TX* and *mess_sent* respectively) have their respective token with value 0, indicating that the previous message (if existed) has already been transmitted properly or that the maximum number of transmission attempts has been reached and the packet has been thrown away.

Mess_ready_TX stores packets trying to be transmitted and *mess_sent* is a buffer where a copy of the transmitting packet is stored in order to prevent a packet loss when a collision occurs.

The firing of *no_mess_tx* passes the token from *New_message* to *mess_ready_TX* and adds a token with value 1 to *Memory* to indicate that this is the first transmission attempt. *Memory* models a counter of the number of transmission attempts. The message will remain in this new place until the channel is sensed idle. When this happens, the transmission is started after a short delay called “interframe spacing”. This process is achieved by the substitution transition *Channel_free* whose associated subpage is *transmission#3* shown in figure 10.

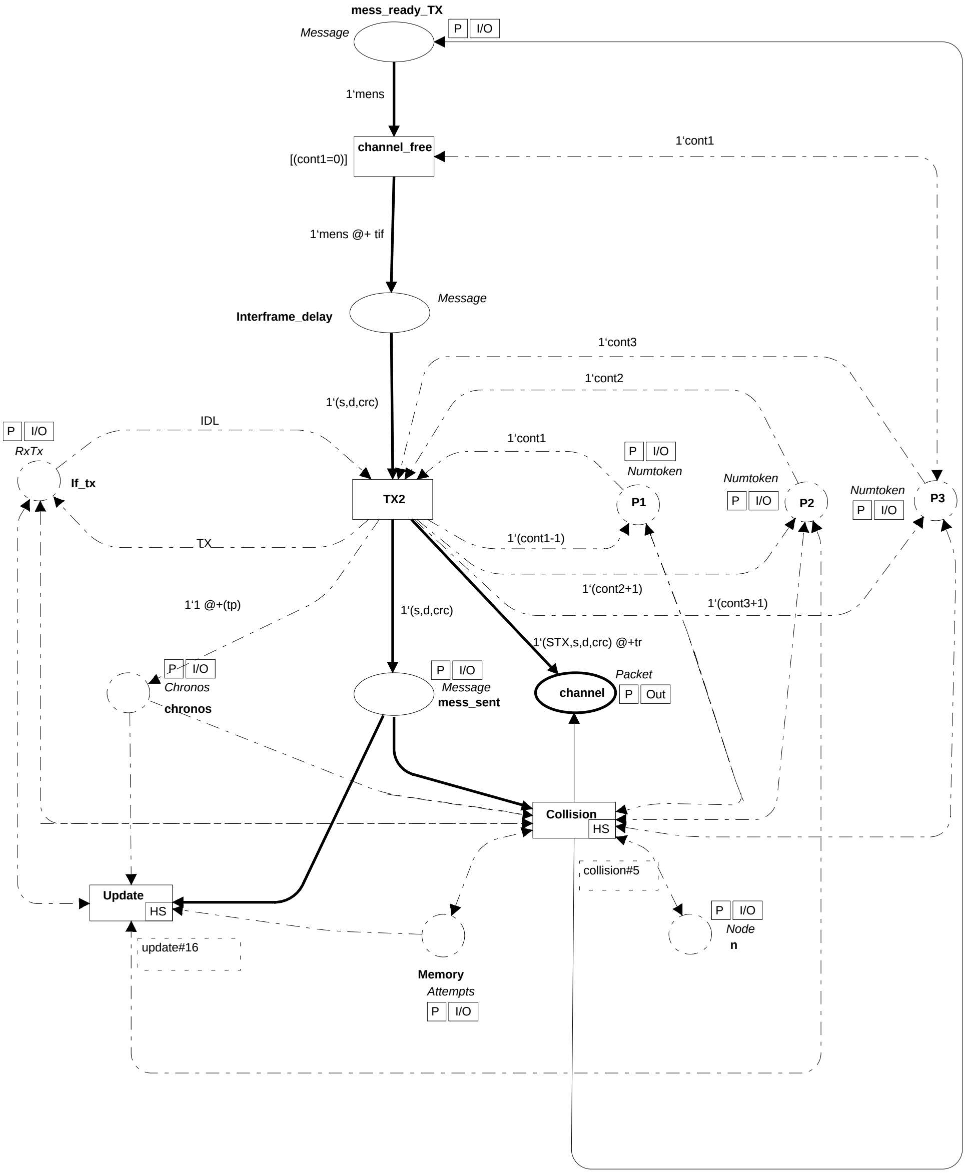


Figure 10. Subpage transmission#3.

In this page, *channel_free* inspects the place *P3* whose token indicates the number of tokens in the place *channel*, the station access point to the channel. As soon as *channel* has no tokens, this transition will fire passing the token from *mess_ready_TX* to *Interframe_delay* with a time stamp "interframe spacing" units (tif) greater than the simulated time at which the transition fired. After this time interval, *TX2* is enabled and it fires starting the packet transmission by storing the packet in the place *channel* and adding a token with value *TX* on the place *if_tx* to indicate that the station has passed from being idle (*IDL*) to be transmitting. It also adds a copy of the transmitting message to the place *mess_sent* and stores on the place *chronos* a token with a time stamp "packet length" (*tp*) greater than the current simulated time. This last place acts as a clock indicating the simulated time at which the transmission ends.

The place *if_tx* models the state of the station: whether it is idle (*IDL*), transmitting (*TX*) or receiving (*RX*). Its token ensures the mutual exclusiveness between the transmission and reception processes. In this way, any of these processes is only allowed to be achieved when the system is idle (*IDL*).

Notice that a copy of the transmitting message has been stored on the place *mess_sent*. This is necessary for the retransmission of such a message in the case that a collision is produced.

Once the transmission is started, two possibilities may occur:

- The packet is properly transmitted.
- A collision is produced and the transmission is retried if the number of attempts is smaller than 16.

In the first case, the only thing left to do to start the transmission on a another frame is to erase the copy of the transmitted message stored in *mess_sent*, update the value of the token in *P2*, remove the token stored in *Memory* and set the state of the station to idle (*IDL*).

The removal of such a copy must be time controlled, i.e., it has to be removed after ensuring that the transmission has ended without collision. That's what we use the token on the place *chronos* for. As soon as the global clock reaches the same value as the time stamp of the token stored on it, the transition *update_p2* in page *update#16* is enabled. That subpage is shown in figure 11.

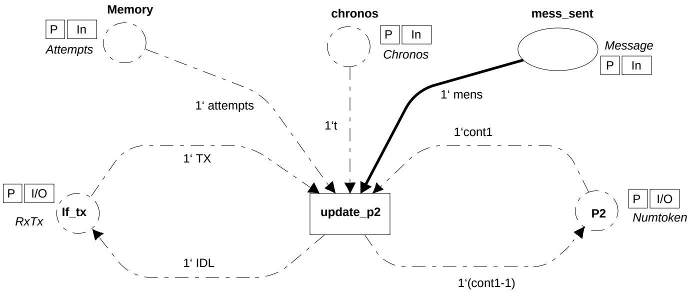


Figure 11. Subpage update#16.

In the other case (collision) the transition *collision* in subpage *collision#5* will fire. Figure 12 shows its subpage. A collision is produced when two or more stations try to transmit their packets concurrently through the channel. Since at every moment the number of tokens in each of the places of the channel is taken in account, a collision will be produced when in the access place to the channel related to a transmitting station there are two or more tokens. The detection of such a situation is performed by the transition *collision* which inspects the place *P3*.

Its firing immediately stops the transmission and lets the station idle (*IDL*). It also removes the token in the place *chronos* to avoid a later firing of *update_p2*, adds a token with *State* equal to *JAM* on the place *channel* in order to inform the rest of the stations about the collision and stores on *inter* a token with value null that does not carry information and that can be interpreted as a signal sent to the transition *top_attempts* to force it to fire.

As soon as the collision is detected and the jamming signal is transmitted, the only thing left to do is to investigate whether the message will be transmitted again or it is thrown away because of having overtaken the maximum number of transmission attempts. *Top_attempts* inspects the number of attempts stored on *Memory*. If this number is smaller than 16, the message copy is passed from *mess_sent* to *mess_ready_TX* and the number of attempts is increased by one unit.

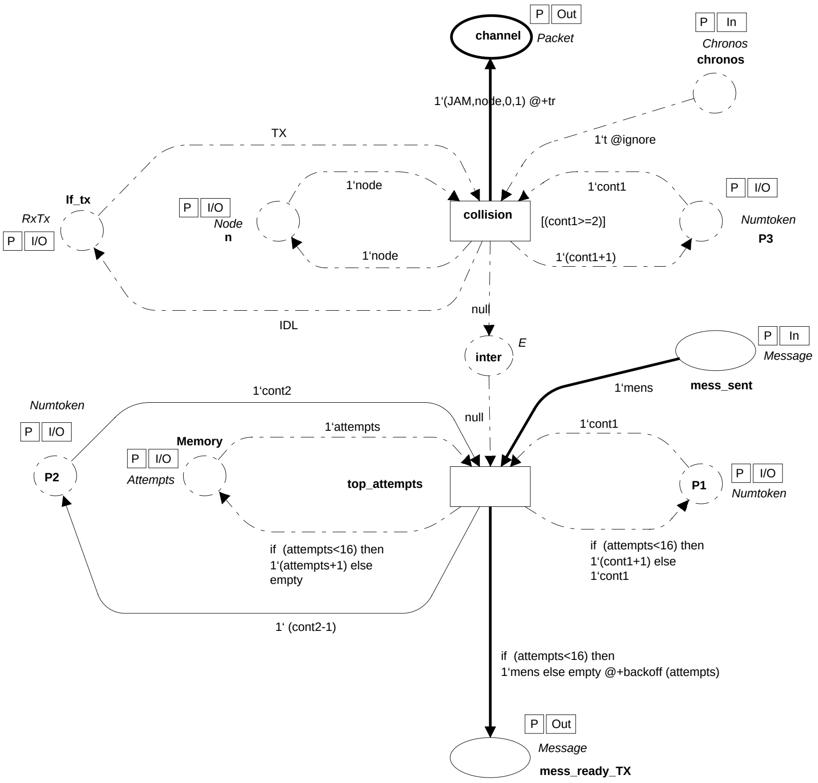


Figure 12. Subpage collision#5.

Before trying to transmit this copy again, it has to wait for a random delay fixed by a function called *backoff*. This is done by giving to the token added to *mess_ready_TX* a time stamp fixed by the function *backoff* which is of the form:

```
infix **;
fun ((x:int)**0)=1 | (x**y)=x*(x**(y-1));
fun backoff (n:attempts)= (real(CPN'randint(0, 2**(min(n,10)))))*ts;
```

where *ts* is the time slot value.

In the case that the number of attempts is equal to 16 both the copy of the message stored on *mess_sent* and the token in *Memory* will be erased.

The modelled station not only takes charge of the transmission process but also the reception one. This is performed by the substitution transition *reception* in figure 6 whose subpage is shown in figure 13. The reception of a frame starts when an idle station detects in the channel the head of a packet (a token with *State* equal to *STX*) appointed to it (*Origin* is equal to the station address). The detection of such a token is performed by *RX1* whose occurrence sends back to the channel a similar token as the consumed one and stores the origin direction on the place *Store_orig*. After this process the station has started to receive a packet, that's why the state of the station is now *RX*.

The station will continue receiving a packet until the transition *RX2* detects a token appointed to it from the same origin station as the first one indicating end of packet (*State*=*ETX*) or collision (*State*=*JAM*). If the token is a jamming signal (*State*=*JAM*), the station ceases from receiving and remains idle. On the other hand, if the received token is an end of packet indicator (*State*=*ETX*), it is supposed that the packet has been totally received. Since the channel is noisy, the receiving station analyses the FCS (*CRC*). If its value is 1, it is sup-

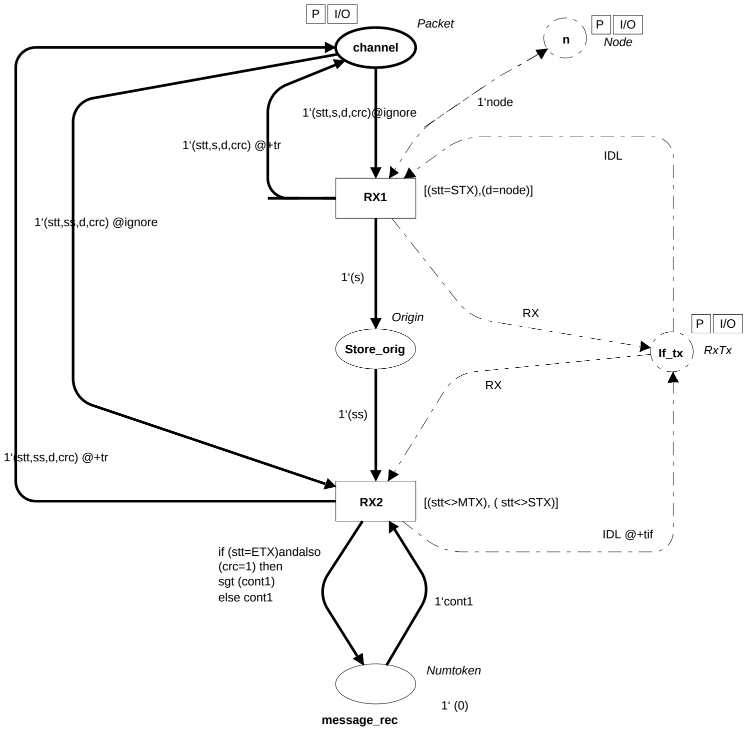


Figure 13. Subpage reception#4.

posed that the received packet is noise free and its information is passed to the LLC layer modelled by the place *message_rec* which also counts the number of well received packets.

In both cases the occurrence of *RX2* stops the reception and lets the station idle. This last fact is modelled adding to *if_tx* a token with value *IDL*. Its time stamp is tif units greater than the simulated time to avoid from receiving another packet until this period of time has passed.

5 Validation of the MAC layer

The validation is the work of proving that the obtained model reproduces the real system behavior. The obtained model may contain errors or be incomplete in the sense that it does not cover all the possible situations through which the real system may pass. It is then necessary to verify the model in order to guarantee it is error free and without any kind of anomalous properties.

To verify the model it is necessary to identify first all the properties it has to comply with. After that, it must be proven that the obtained model satisfies them.

Design/CPN supports two different analysis methods to verify the CPN model performance: simulation and occurrence graph analysis.

5.1 Simulation of the model

Simulation is an important instrument for debugging and validating CPN. It gives the developers an improved understanding of the system behaviour.

During the construction of the model, simulation was intensively used. In its early phases, manual simulations where made. This kind of simulation was applied to the individual model subpages in order to test their performance. Its application helped to detect some errors allowing us to dig down in the CPN model to locate and fix them. In this sense, an iterative approach was used, alternating between modeling and simulation.

The first prototype was gradually refined, and eventually it constituted the final model. Since manual simulations became time cost due to the model size, automatic simulations were achieved in order to check the correctness of the final model. In this last kind of simulation mode it is only possible to know the initial and final markings and the fired transitions sequence (stored in a text file), but it is nearly impossible to know the intermediate markings of the model. It was necessary to find a way of knowing what happens in the net during simulation and this was made using the report facilities.

Report facilities allow the user to apply standard graphical routines to create and manipulate graphical representations of the simulation results. With their help it is possible to obtain graphics relating the occurrence of certain events in the net with the simulated time in which they occur. In this sense, some code segments updating graphics were attached to certain transitions. Analysing the obtained graphics and bearing in mind the initial and final markings, it was easy to establish whether during the simulation the model worked properly.

As an example, figure 14 shows one of the obtained graphic for the packet generation system. It describes the bursts of packets generated by the LLC user.

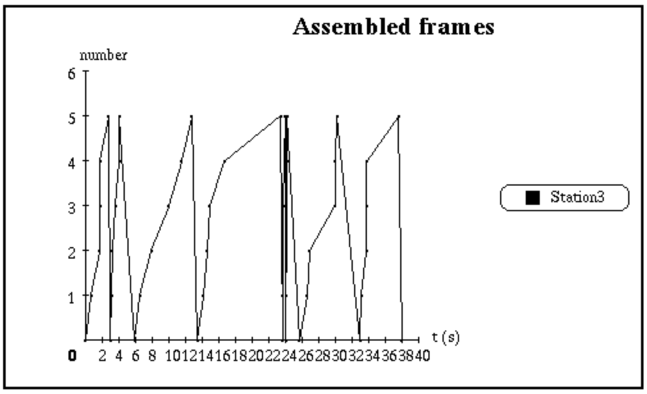


Figure 14. Bursts generation.

Although simulation is extremely useful for the understanding and debugging of a CPN, it is obvious that, by means of simulation it is impossible to obtain a complete proof of its dynamic properties. This forced us to analyse the model using occurrence graphs.

5.2 Occurrence graph analysis

At the end of the design phase, occurrence graph analysis was applied in order to detect as many errors as possible and prove that the obtained model fulfilled some dynamic properties (reachability, boundeness, liveness, etc). The basic idea behind occurrence graphs is to construct a graph containing a node for each reachable marking and an arc for each occurring binding element. Obviously such a graph may become very large and in many cases, infinite.

In our case, despite of the adopted decisions in the modeling process in order to minimize the occurrence graph size, it was quite too large to allow its analysis via visual inspection (the obtained graph for a 10 seconds analysis has a partial status and more than 10.000 nodes).

The great size is due to:

- The great quantity of different actions that may occur in the network: each station is able to send packets to the rest of the stations. At the same time, these packets may collide among them a number of times, lose because the maximum number of transmissions has been reached or because of noise, transmit properly, etc.

- The inclusion of time in a cyclic system like this (the transmission-reception process in each station is a cyclic system) makes the timed occurrence graph become infinite, because each repeated appearance of a marking corresponds to a new state and hence implies the creation of a new node.

To obtain occurrence graphs with a manageable size and a full status, it was necessary to simplify the CPN model. In this sense, the size of the network was reduced to only three stations without changing the channel and each of them was allowed to transmit only one new packet. These three packets were forced to collide and later be properly retransmitted.

A part of the occurrence graph (O-graph) for the simplified network is shown in figure 15. It has a full status and 751 nodes.

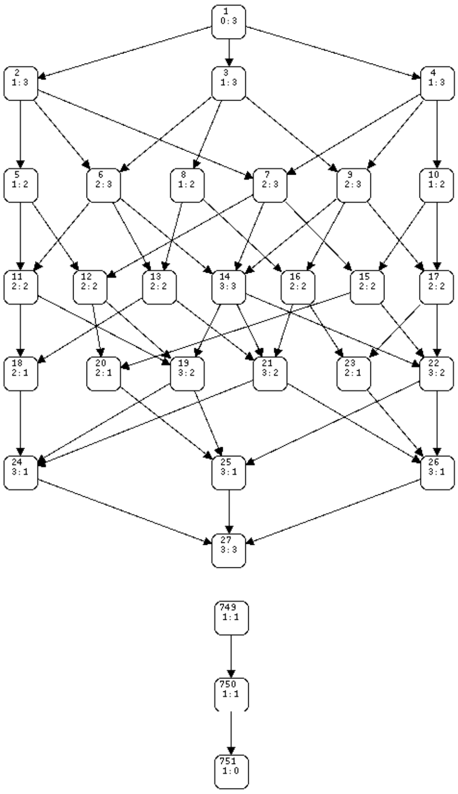


Figure 15. Occurrence graph.

The O-graphs were constructed by means of the OG tool [Occ93]. When an O-graph has been calculated, a set of standard queries makes it possible to investigate reachability, boundeness, home, liveness and fairness properties. For the reduced model a standard report was calculated and a summary is shown in table 1. The report doesn't show deadlocks and the examination of the state [751] indicates the packets have been properly transmitted and the channel is free. This standard report applies some standard queries to the model and saves the results in a text file. Its analysis revealed no errors in the model.

Table 1: OG results.

Nodes	Arcs	Time	Dead Markings
751	2257	3	1 [751]

6 Conclusions

Two different versions of the tool have been used during the construction of the model (3.0.4 and 3.1.1). The migration from the 3.0.4 to the 3.1.1 version not only depended on time reasons (note that this last version has recently appeared) but also on some problems detected in the previous version. These problems were related to an strange syntax sensibility on the tool that made the simulation and validation of the model by means of the 3.0.4 version nearly impossible. After hard enquiries in the model and Internet pages, the problem could be partially solved. The detected errors were mainly related to the introduction of carry returns in color regions of places. However some problems could not be solved neither by means of the 3.0.4 version nor by means of the 3.1.1 version. This unsolved problem, that in fact still affects our model, is the application crash when some report facilities (bar charts) are used. Despite of our efforts and other people advices it still remains.

In addition to this problem, there are some other areas that have become into serious problems along this project. Some of them have been partially solved by means of the 3.1.1 version they mainly still remain unsolved. To start with, the pass from the editor to the simulator and occurrence graph generator is very time cost. When the model has a reasonable size the time spent to enter the simulator usually makes the user drive to despair, especially during the model debugging when it is necessary to enter the simulator and return back to the editor many times in order to correct some errors. Another important problem is the great memory requirements for the application. In our case, the graduate student is disposed of a Pentium working at 120 Mhz and 48 Mbytes of Ram memory. It was rather impossible to work with this equipment. While these two problems remain unsolved it will be difficult to introduce Design/CPN as a common use tool among our students.

Despite of this, some good impressions have been extracted from the application. It must be brought out the well-suited graphical interface, the possibility of constructing modular models using the hierarchical capabilities and overall, its straightforward analysis methods: simulation and occurrence graphs. Version 3.1.1 has reinforced this last section by the introduction of occurrence graphs with equivalence classes and symmetries. It is a pity that OE/OS graphs with time have not been theoretically developed yet. On the contrary they would had been applied to our model.

Bibliography

[Goo87] G. Goods and J. Hartmanis. *Lecture Notes in Computer Science. Advances in Petri Nets.* Springer- Verlang. "An accurate performace model of CSMA/CD bus LAN". 1987

[Jen92] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volume 1. Springer-Verlag. 1992.

[Jen94] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volume 2. Springer-Verlag. 1994.

[Jen97a] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volume 3. Springer-Verlag. 1997.

[Jen97b] Kurt Jensen. *A Brief Introduction to Coloured Petri Nets.* Computer Science Department, University of Aarhus. Denmark. http://www.daimi.aau.dk/~Kjensen/paper_books/. 1997

[Jes94] Jesús García Tomás, Santiago Fernando y Mario Piattini. *Redes para procesos distribuídos*. Ra-ma. 1994.

[Joh96] John Freer. *Introducción a la tecnología y diseño de sistemas de telecomunicaciones y redes de ordenadores.* Anaya. 1996.

[Ref93] *Design/CPN Reference Manual for X-Windows.* Version 2.0. Meta Software Corporation. 1993.

[Occ93] *Design/CPN Occurrence Graph Manual.* Version 3.0. University of Aarhus. 1993

[Tan] Andrew S. Tanenbaum. *Redes de ordenadores.* Second edition. Prentice-Hall.

[The97] Theo van Drimmelen. *Implementation of Statistical Functions in Design/CPN.*

http://www.daimi.aau.dk/designCPN/libs/pdf. 1997.

Modelling and Analysis of a Flowmeter System

Louise Lorentsen
Department of Computer Science
University of Aarhus
Ny Munkegade, Bldg. 540
DK-8000 Aarhus C, Denmark
E-mail:louisel@daimi.au.dk

Abstract. In this paper the practical use of Coloured Petri Nets and Design/CPN is demonstrated through an industrial cooperation project. The project is based on studies of a concrete industrial product and on methods well-known and currently used in the design of the product in the company. The paper describes the modelling in Design/CPN of the system of the product followed by an analysis of the model in relation to a set of desired properties defined by the company. The properties are formally verified by means of occurrence graphs and the Design/CPN Occurrence Graph tool.

1 Introduction

This paper is based on a project in the Centre for Object Technology (COT), which is a joint project between industry and universities in Denmark. The project is a cooperation project between the Danish industrial production company Danfoss and the CPN group at University of Aarhus. Three persons, representing both sides, have been involved in the project as primary project members supported by the leaders from both cooperating parties. The project group has held a number of meetings and communicated electronically throughout the whole project. However, most work has been based on internal documents kindly made available for the project by the company.

The overall aim of the paper, and the underlying project, is to show the applicability of Coloured Petri Nets (CPNs) [7] and occurrence graphs (OGs) [6] in industrial settings. In order to succeed in this endeavour it is important to relate to concrete industrial products on one hand and to the solutions used so far in industry on the other hand. In the project, which this paper is based on, this means that a specific product of the cooperating industrial company, a flowmeter (Fig. 1), is chosen as a relevant study object, and that present design techniques of this system is related to and combined with the new design and analysis techniques based on CPNs and OGs.

Fig. 1. A flowmeter

Flowmeters are primarily used for measuring the flow of water through pipes. In the concrete flowmeter system of this project different processes are used to measure the flow, e.g., a flow measurement process, a temperature measurement process and a calculation process. These processes cooperates to carry out the overall task - to measure the flow of water in the pipe.

In recent years Danfoss has changed the concept of the flowmeters from a system with a centralised operating system to a distributed and more flexible system with hardware modules which can be combined in many ways. With the new distributed flowmeter system a customer can design his own system and construct it from only the needed modules. However, this makes new problems arise. In the distributed setting based on communication between the processes it is difficult to reason about the individual processes and their influence on each other.

Practical tests in Danfoss have shown that the process communication in the first design version of the flowmeter system contained at least one deadlock, and therefore a new design was needed. As part of the new design a set of desired properties of a flowmeter system have been formulated and a solution has been presented using new design methods, but the correctness of the solution still has to be formally verified. This project should be seen as part of these experiments with new methods in Danfoss. The aim of the underlying project has been to secure the quality of the design process in Danfoss introducing Coloured Petri Nets (CPNs or CP-nets) and the tool Design/CPN [3].

In the following the flowmeter system is described in more detail, a CPN model of the protocol is introduced, a set of desired properties of the flowmeter system are motivated and formulated, different design alternatives for the processes are modelled in Design/CPN and it is analysed by means of occurrence graphs using the Design/CPN OG tool [8] whether the design alternatives fulfil the desired properties of a flowmeter system.

2 The flowmeter system

The flowmeter system consists of one or more modules connected via a Controller Area Network (CAN) [9]. The processes are called CAN Applications (CANAPPs) and are placed in the modules. The placement of the CANAPPs in the modules is flexible, however, which means that the concrete placement of the CANAPPs should not affect the functionality of the total system. All communication in the system consists of message passing between the CANAPPs. To control the communication among the CANAPPs in the modules a superior communication system is needed. The communication system consists of parts in every module called drivers. This means that a flowmeter system consists of a number of modules, each containing a driver and one or more CANAPPs. A system of three modules each containing four CANAPPs is shown in Fig. 2. In the following the protocol defining the transaction procedure and message structure is presented.

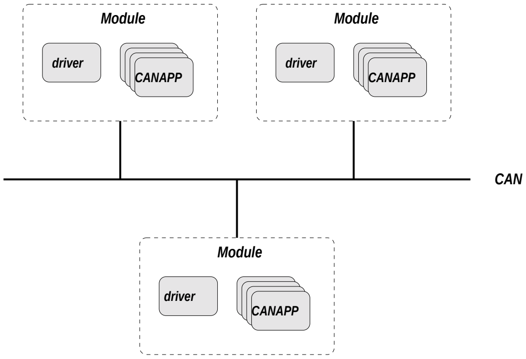


Fig. 2. A flowmeter system

2.1 The protocol

The protocol is based on a three layered architecture. The layers constitute a collapsed form of the OSI seven layer architecture, mapping onto the physical, data link and application layers of the OSI Reference Model [4].

The application layer provides six services: read, write, action, broadcast, command and event as shown in Fig. 3. The services fall into two groups, which will be dealt with separately in the following description: two-way asynchronous communication and one-way asynchronous communication.

Service	**Function**
read	Address an other CANAPP and read the value of an attribute
write	Address an other CANAPP and modify the value of an attribute
action	Address an other CANAPP and ask it to execute an action
broadcast	Cyclically distribution of actual values without any acknowledgements
command	Address all other CANAPPs and ask them to execute a system command
event	Address all other CANAPPs and report a single event

Fig. 3. The application layer services

The system considered in the rest of this section is a system of three modules each containing one CANAPP. The system is shown in Fig. 4.

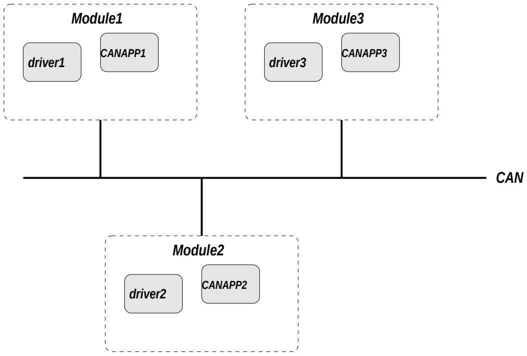


Fig. 4.

2.2 Two-way asynchronous communication

When a CANAPP invokes a read, write or action service, the CANAPP acts like a client. A destination CANAPP, which is going to act like a server in the interaction, is addressed in a request. A response is returned from the server CANAPP.

In case CANAPP 2 invokes a read service to read an attribute of CANAPP 3, the sequence of events shown in Fig. 5 occurs. Write and action services are performed in a similar way.

1. CANAPP 2 delivers the request to the driver in the same module (driver 2) to have the request sent to CANAPP 3.
2. Driver 2 transmits the request to the driver in the module in which CANAPP 3 is located (driver 3) via the CAN bus.
3. An INCAN OK acknowledgement is returned from driver 3 to driver 2 (i.e. driver 2 is ready to accept messages from any other driver).
4. Driver 3 delivers the request to CANAPP 3, which starts processing the request.
5. CANAPP 3 completes the processing, generates a response and delivers it to the driver in the same module (driver 3) to have the response sent to CANAPP 2.
6. Driver 3 sends the response to the driver in the module in which CANAPP 2 is located (driver 2) via the CAN bus.
7. An INCAN OK acknowledgement is returned from driver 2 to driver 3.
8. Driver 2 delivers the response to CANAPP 2.

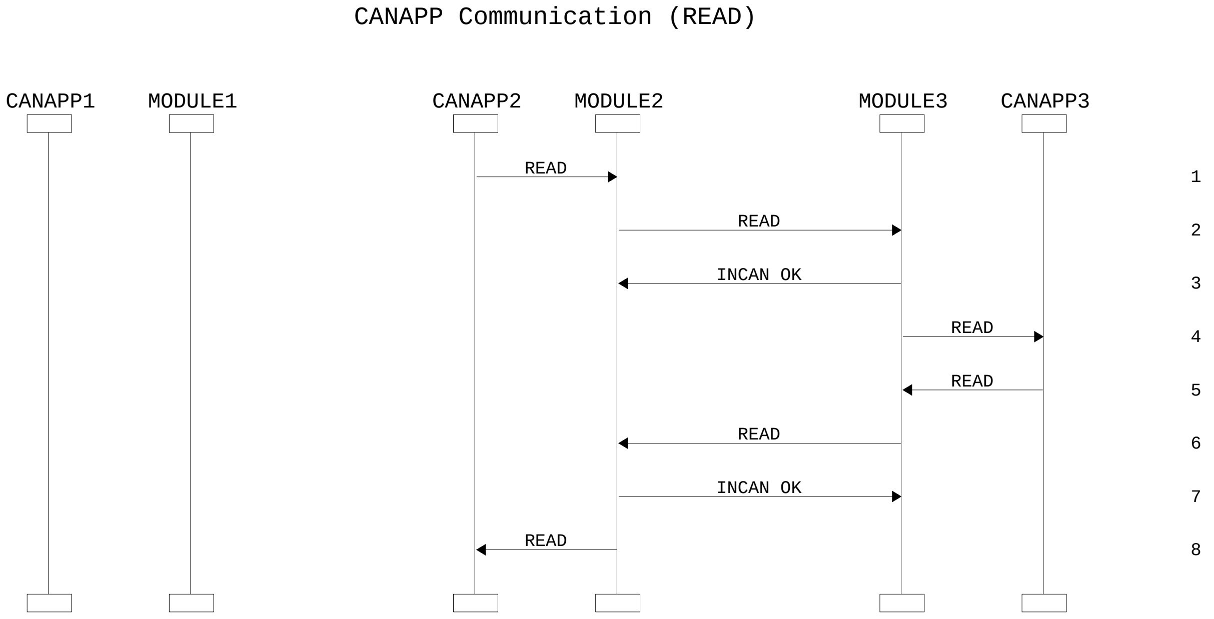


Fig. 5. Two-way asynchronous communication

2.3 One-way asynchronous communication

The one-way asynchronous communication may be used to distribute information to all the other CANAPPs in the flowmeter system - either as broadcast, as shown in Fig. 6, without any guaranty of delivery, or as commands or events, as shown in Fig. 7, with guaranteed delivery.

In case CANAPP 3 invokes a broadcast service the sequence of events shown in Fig. 6 occurs.

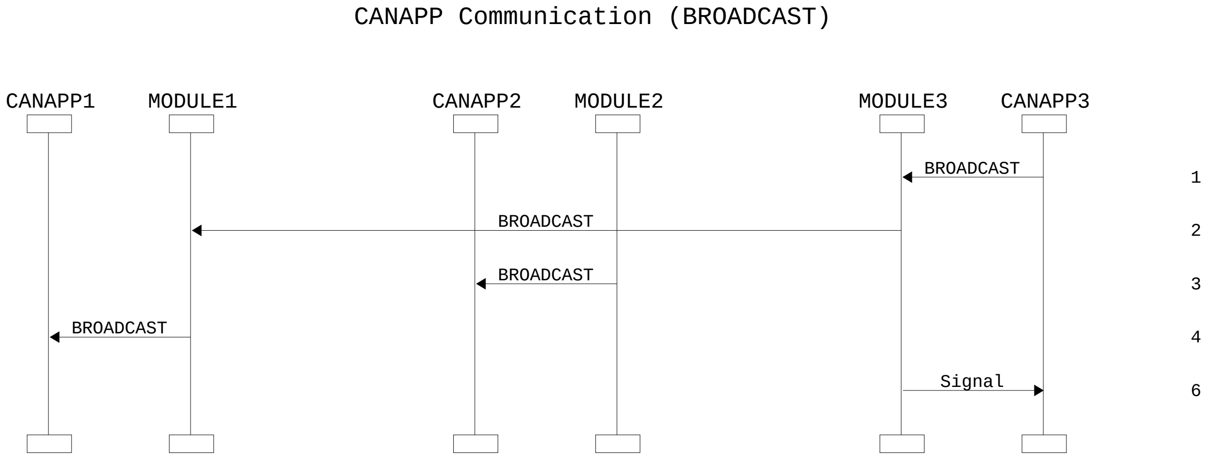


Fig. 6. One-way asynchronous communication - broadcast

1. CANAPP 3 delivers the broadcast message to the driver on the same module (driver 3) to have the message broadcasted to all other CANAPPs in the flowmeter system.
2. Driver 3 sends the message to all other drivers in the system via the CAN bus. No acknowledgements are returned.

3. Driver 2 delivers the broadcast message to its local CANAPPs (CANAPP 2).
4. Driver 1 delivers the broadcast message to its local CANAPPs (CANAPP 1).
5. CANAPP 3 is informed about completion of the transmission. This is not an acknowledgement from the individual receiver CANAPPs but a final synchronisation signal from the transmitter to the sending CANAPP. Notice that the signal can occur before the message has been received by the CANAPPs in the system.

In case CANAPP 3 invokes an event service, the sequence of events shown in Fig. 7 occurs. The command service is performed in exactly the same way as the event service. The delivery of the message is guaranteed by use of the INCAN OK acknowledgement mechanism.

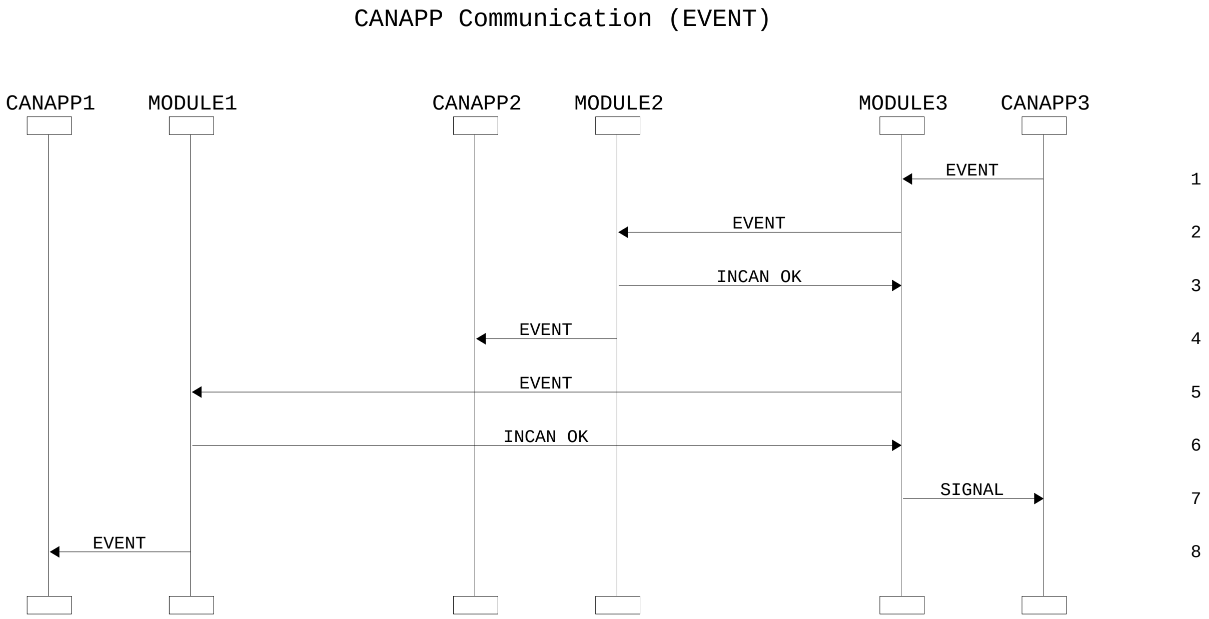


Fig. 7. One-way asynchronous communication - event

1. CANAPP 3 delivers the message to the driver in the same module (driver 3) to have the message sent to all other CANAPPs in the flowmeter system.
2. The requests are sequentially distributed to all other drivers in the system with an acknowledgement from each individual receiver.
3. When an acknowledgement is received, the request is sent to the next receiver.
4. CANAPP 3 is informed about completion of the transmission. This is not an acknowledgement from the individual receiver CANAPPs but a final synchronisation signal from the transmitter to the sending CANAPP. Notice that the signal occurs before the message has been received by CANAPP 1.

3 CPN model of the flowmeter system

The constructed model of the flowmeter system consists of 11 pages. An overview is given via the hierarchy page in Fig. 8.

The system modelled and presented in this section is a system consisting of two modules, each containing two CANAPPs. This system is shown in Fig. 9.

The topmost page Flowmeter_System of the CPN model is shown in Fig. 10. This page contains a top level description of a flowmeter system consisting of a number of CANAPPs communicating by means of the three layered protocol presented in Sect. 2. This is reflected by the four parts of the page: A part modelling

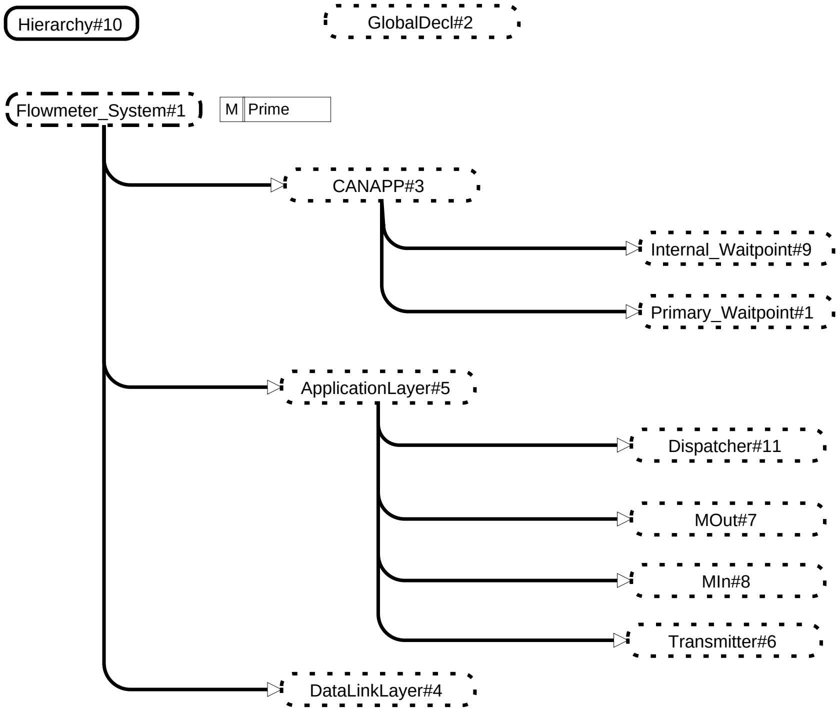


Fig. 8. The hierarchy page

the CANAPPs and three parts modelling the three layers of the protocol. CANAPP, ApplicationLayer and DataLinkLayer are all substitution transitions which means that the detailed behaviours are shown on the subpages with the corresponding names. The four CANAPPs in the system are modelled by use of colour sets. Therefore only one page modelling the CANAPPs is needed in the CPN model. The page CANAPP is further split into two parts modelling two different design alternatives for the CANAPPs. The presentation here concentrates on the parts of the model relevant for experimenting with the two different high level design alternatives of the CANAPPs. Therefore only the pages ApplicationLayer and CANAPP are presented in detail. First the page ApplicationLayer is presented as a basis for understanding the following CANAPP presentation and analysis. Then the desired properties of a flowmeter system are motivated and presented, the page CANAPP containing the design alternatives is presented and it is analysed whether the system modelled fulfils the properties formulated.

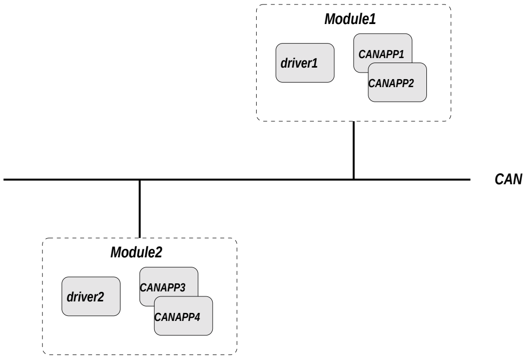


Fig. 9. The flowmeter system modelled

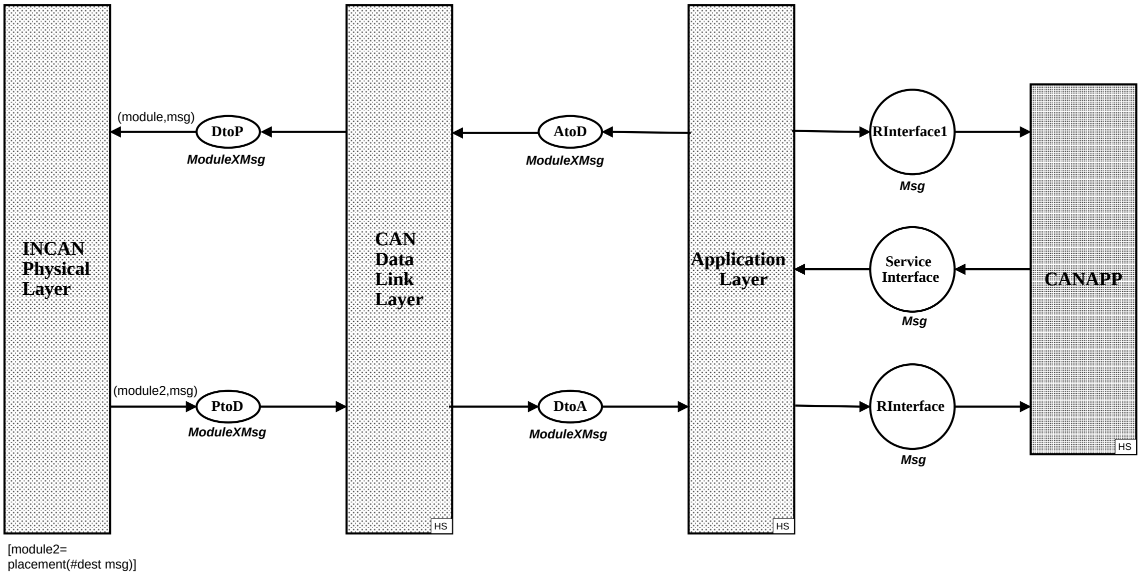


Fig. 10. Page Flowmeter_System

3.1 CPN model of the protocol

The page ApplicationLayer is shown in Fig. 11. This page contains the model of the driver, which is the part of the communication system in each module. The part of the driver responsible for receiving and delivering the messages from and to the CANAPPs in the module is modelled in the page Dispatcher. Messages are received from the CANAPPs via the place Service Interface and delivered to the CANAPPs via the places RInterface and RInterface1. The dispatcher is responsible for the internal communication in the module. There is no need to send messages from a CANAPP in a module to another CANAPP in the same module via the CAN bus. These intra module messages is delivered to the receiving CANAPP by the dispatcher. The inter module messages are handled by the rest of the driver. The dispatcher and the rest of the driver communicate via the 7 places to the left of the substitution transition Dispatcher in Fig. 11. We will return to these places later in this section.

To avoid deadlock between two modules the driver is implemented to support some concurrency. All incoming requests are received and handled to the CANAPPs for processing even if the driver is waiting for a response to an outstanding request from a CANAPP in the module. Specific buffers are allocated to handle incoming and outgoing broadcasts. This means that independent on other activities the modules are always able to transmit and receive broadcast messages. To obtain the wanted concurrency the part of the driver responsible for sending and receiving messages and to wait for responses is implemented as four state machines, two state machines for outgoing requests and the corresponding incoming responses and two state machines for incoming requests and the corresponding outgoing responses. This is reflected in the model by the four parts: BOut, MOut, MIn and BIn. MOut and MIn are substitution transitions, which means that the detailed behaviours are shown on the subpages with the corresponding names. These subpages will be presented later in this section.

Messages received from the data link layer on place DtoA are directed through the IN-buffers MessageIN and BroadcastIN, depending of the type of message. Broadcast messages are directed through BroadcastIN, the rest is directed through MessageIN. If the message is a read, write or action message the corresponding response is directed through the same IN-buffer in the opposite direction. Messages from the CANAPPs are directed through the OUT-buffers MessageOUT and BroadcastOUT to the data link layer on place AtoD. If the message is a read, write or action the corresponding response flows through the same OUT-buffer. In the rest of this section the modelling of the four statemachines BroadcastOUT, BroadcastIN, MessageOUT, and MessageIN is presented.

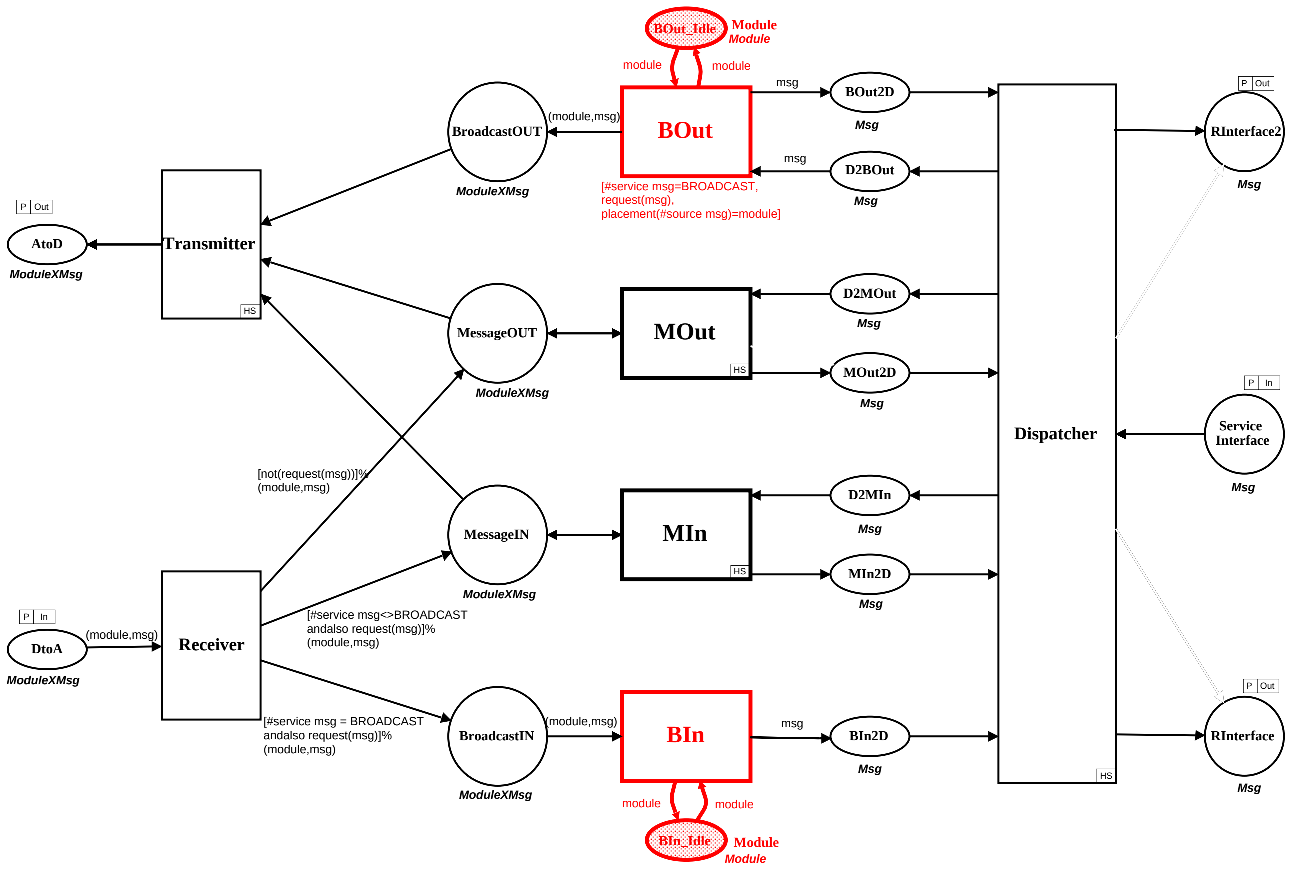


Fig. 11. Page ApplicationLayer

The BroadcastOUT statemachine The BroadcastOUT statemachine for sending broadcast messages can be seen in page ApplicationLayer which is shown in Fig. 11. The statemachine is very simple and consists only of the place BOut_Idle and the transition BOut. Initially there is a token corresponding to each module in the system on place BOut_Idle modelling that all drivers in the system are idle and ready to transmit broadcast messages. When a CANAPP sends a broadcast message a token modelling the message is by the dispatcher put on place D2BOut. This message is to be sent on the CAN bus to all other drivers in the system. The dispatcher is responsible for sending the broadcast message to all other CANAPPs in the same module. When there is a token corresponding to a broadcast message on place D2BOut the transition BOut can occur. The message is put on place BroadcastOUT to be put on the CAN bus and a token modelling a signal is put on place BOut2D. This signal is by the dispatcher delivered to the sending CANAPP. The tokens on place BOut_Idle modelling the modules in the system ensure that each module only transmits one broadcast message at a time.

The BroadcastIN statemachine Also the BroadcastIn statemachine for receiving broadcast messages can be seen in page ApplicationLayer which is shown in Fig. 11. The statemachine consists of the place BIn_Idle and the transition BIn. Initially there is a token corresponding to each module in the system on place BIn_Idle modelling that all drivers in the system are idle and ready to accept broadcast messages from the data link layer to be delivered to the CANAPPs in the module.

When a broadcast message is received from the data link layer, the message is put on the place BroadcastIN. Now the transition BIn can occur and the message is put on place BIn2D. The dispatcher now distributes the message to all CANAPPs in the module. The tokens on place BIn_Idle modelling the modules in the system ensure that each module only receives one broadcast message at a time.

The MessageOUT statemachine The MessageOUT statemachine is modelled by the substitution transition MOut. The page MOut is shown in Fig. 12.

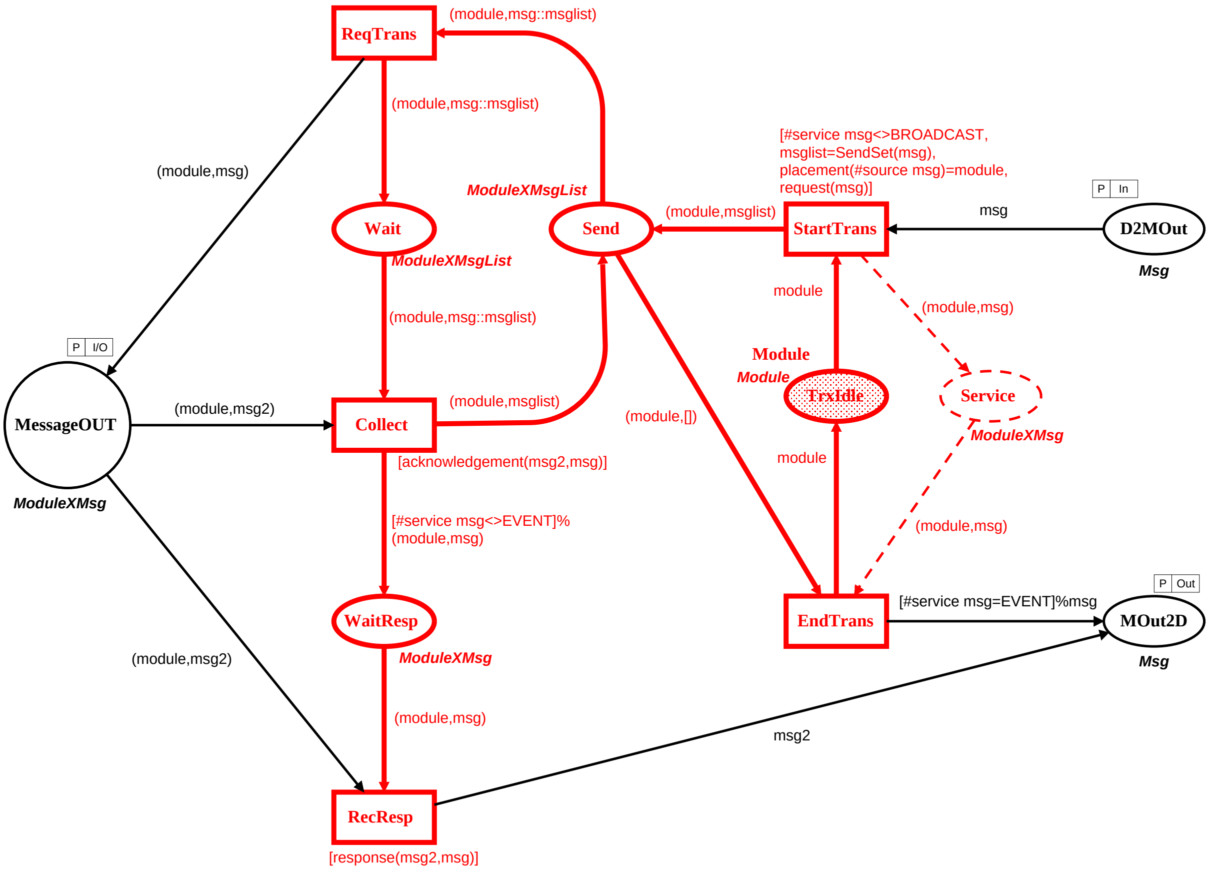


Fig. 12. Page MOut

Initially the driver in each module is idle and ready to accept messages from the CANAPPs in the module to be sent to other CANAPPs in the system. This is modelled the place TrxIdle which contains a token modelling each of the modules in the system. When a CANAPP in a module wants to send a non-broadcast message to a CANAPP in another module, a token modelling the message is by the dispatcher put on place D2MOut. The dispatcher is responsible for delivering intra module messages.
If the driver in the module in which the sending CANAPP is placed is idle the transition StartTrans can occur. The effect of the occurrence of the transition StartTrans is

- The token corresponding to the module is removed from place TrxIdle modelling that the driver is not idle anymore but is in the process of transmitting the message.

- The message is removed from place D2MOut
- A token consisting of a pair where the first component is the module and the second component is a list of the messages which are to be sent is put on the place Send. The list is constructed by use of the function `SendSet`. Depending on the type of the message being sent different actions occur. If the message is a read, write or action message, the list has the message sent from the CANAPP as the only element. If the message instead is an event or command message the elements in the list are a message to each of the other CANAPPs in the system witch is placed on another module as the sending CANAPP. All of these messages are to be sent one by one and an acknowledgement is awaited between each individual transmission.
- A token modelling the message is put on place Service. The function of this place is to "remember" the type of message being sent. In this way it can later be decided whether a response should be awaited (in case of read, write or action messages) or a signal should be sent to the sending CANAPP (in case of event and command messages).

If the list of messages on place Send is non-empty the transition ReqTrans can occur. The effect is

- The pair `(module,msg::msglist)` is removed from the place Send.
- The first message in the list is put on place MessageOUT to be put on the CAN bus.
- A token corresponding to the pair of the module and the list of messages (including the message just send) is put on place Wait modelling that an acknowledgement is awaited between each transmission.

The acknowledgement will be directed through the same OUT-buffer. When an acknowledgement to the message just sent arrives on the place MessageOUT the transition Collect can occur. The effect is

- The pair `(module,msg::msglist)` is removed from the place Wait.
- The pair `(module,msglist)` is put on place Send to have the rest of the messages in the list sent.
- If the message is a read, write or action message a response which will be directed through the same OUT-buffer has to be awaited and a the pair `(module,msg)` is put on place WaitResp.

When the transmission is finished (i.e, the list of messages on place Send is empty), the transition EndTrans can occur. The effect is

- A token corresponding to the module is put on place TrxIdle modelling that the driver is now idle and ready to send a new message.
- The pair of the module and the message on place Service is removed. If the message is an event or command message a token modelling a signal is put on place MOut2D to be delivered to the sending CANAPP by the dispatcher. The signal indicates that the transmission is finished, not that the individual messages actually have been received by the CANAPPs.

When a response arrives the at the place MessageOUT transition RecResp can occur. The effect is

- The pair `(module,msg)` is removed from place WaitResp modelling that no response is awaited anymore.
- The token corresponding to the response message is removed from MessageOUT.
- The response is put on place MOut2D to be delivered to the receiving CANAPP by the dispatcher.

The MessageIN statemachine The statemachine MessageIN is modelled by the substitution transition MIn. The page MIn is shown in Fig. 13.

Initially the driver in each module is idle and ready to accept request messages from the data link layer on place MessageIN to be delivered to the CANAPPs in the module. This is modelled by the place RecIdle which contains a token modelling each of the modules in the system.
If the driver in the module in which the receiving CANAPP is placed is idle and a request message arrives on place MessageIN the transition Collect can occur. The effect of the occurrence is

- The token corresponding to the module in which the receiving CANAPP is placed is removed from the place RecIdle modelling that the driver in the module is no longer idle but in the process of receiving a message.
- The token modelling the message is removed from the place MessageIN

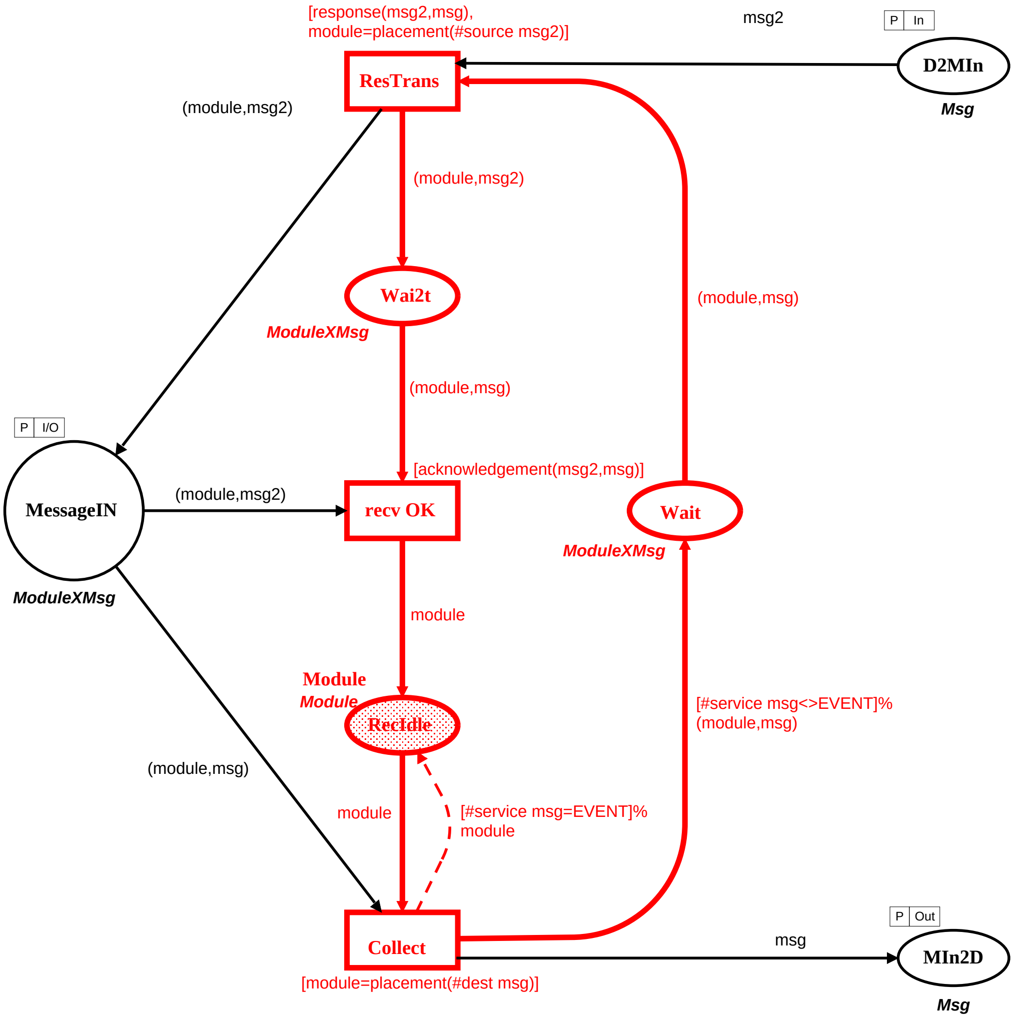


Fig. 13. Page MIn

- A token modelling the message is put on the place MIn2D to be delivered to the receiving CANAPP by the dispatcher.
- If the message is an event or command message no response is to be returned and a token corresponding to the driver is put back on place Wait modelling that the driver is again idle and ready to receive non-broadcast messages from the data link layer. If instead the message is a read, write or action message the driver has to wait for an response message to deliver to the sending CANAPP. In this case a token is put on the place Wait.

When the receiving CANAPP has generated the response a token corresponding to the message is by the dispatcher put on the place D2MIn. The transition ResTrans can now occur. The effect is

- The message is removed from the place D2MIn.
- The pair of the module and the request message is removed from the place Wait modelling that the response from the CANAPP is no longer awaited.
- The message is put on place MessageIN to be put on the CAN bus.
- A token corresponding to the pair of the module and the message is put on place WaitAck modelling that an acknowledgement is awaited.

The acknowledgement will be directed through the same IN-buffer. When an acknowledgement to the response message sent arrives the transition recv OK can occur. The effect is

- The pair (module,msg) is removed from the place WaitAck modelling that an acknowledgement is no longer awaited.
- A token corresponding to the module is put on place RecIdle modelling that the module is now idle and ready to receive messages from the data link layer to be delivered to the CANAPPs in the module.

The CPN model for the protocol defining the transaction procedure for message passing among the CANAPPs in a flowmeter system has now been introduced. In the following sections the CPN model of two different design alternatives for the CANAPPs are presented and analysed.

3.2 CPN model of the CANAPPs

When two CANAPPs communicate they do it in an asynchronous way: when receiving a read, write or action message a response message is sent back to the sender of the request. As we will see in this section the CANAPPs need to be carefully designed to avoid deadlocks and other problems.

When designing the flowmeter system Danfoss used the OCTUPUS method [1], where two basic design alternatives are presented: internal wait point and primary wait point.

As it is most important in collaboration projects to relate to existing knowledge and concepts in the field examined, this section will combine these two well known design alternatives with the CPN modelling of the system. Therefore a design of the CANAPPs based on each of the wait points above is modelled.

After the modelling each of the two models has to be analysed concerning the presence of the desired properties of a flowmeter system. These are, as stated by the producer:

- No deadlocks in the flowmeter system
- No change in the variables of a CANAPP when it is in the process of doing asynchronous communication

Internal wait point When a request is sent the CANAPP is blocked. If the CANAPP cooperates with other CANAPPs on the same module all CANAPPs in the task are blocked. The CANAPPs in the task are not released until the response message (or signal) has been received.

A CPN model of the CANAPPs based on the internal wait point approach is modelled on page InternalWaitpoint which shown in Fig. 14. The system modelled is the system in Fig. 9 of two modules each containing two CANAPPs. CANAPP 1 and 2 cooperate in the same task in module 1 and CANAPP 3 and 4 cooperate in the same task in module 2.

Initially all CANAPPs in the system are idle and ready to generate request messages to or receive messages from the other CANAPPs in the system. This is modelled by the place Idle which contains a token for each of the CANAPPs in the system. The place Attr models the attributes of the CANAPPs. Each of the CANAPPs modelled has an attribute (a local variable) which is modelled as an integer. When a CANAPP receives a write request or receives a response to a previously sent read request the CANAPP updates it's attribute according to the value sent in the message.
If the CANAPP is idle the transition Request can occur. The effect is

- The tokens modelling the sending CANAPP and the other CANAPP in the task are removed from the place Idle modelling that both of the CANAPPs in the task are blocked until a response message or a signal returns.
- A token modelling the message is put on the places Service Interface to be delivered to the receiving CANAPP.
- A pair of the sending CANAPP and the message sent is put on place Wait modelling that the message is sent and the sending CANAPP has to wait for a response message or a signal.
- The places Packet and Attr1 are inspected to generate a packet with the value of the attribute of the sending CANAPP.

When a response message arrives on place RInterface1 the transition Confirm can occur. The effect is

- The message is removed from place RInterface1.

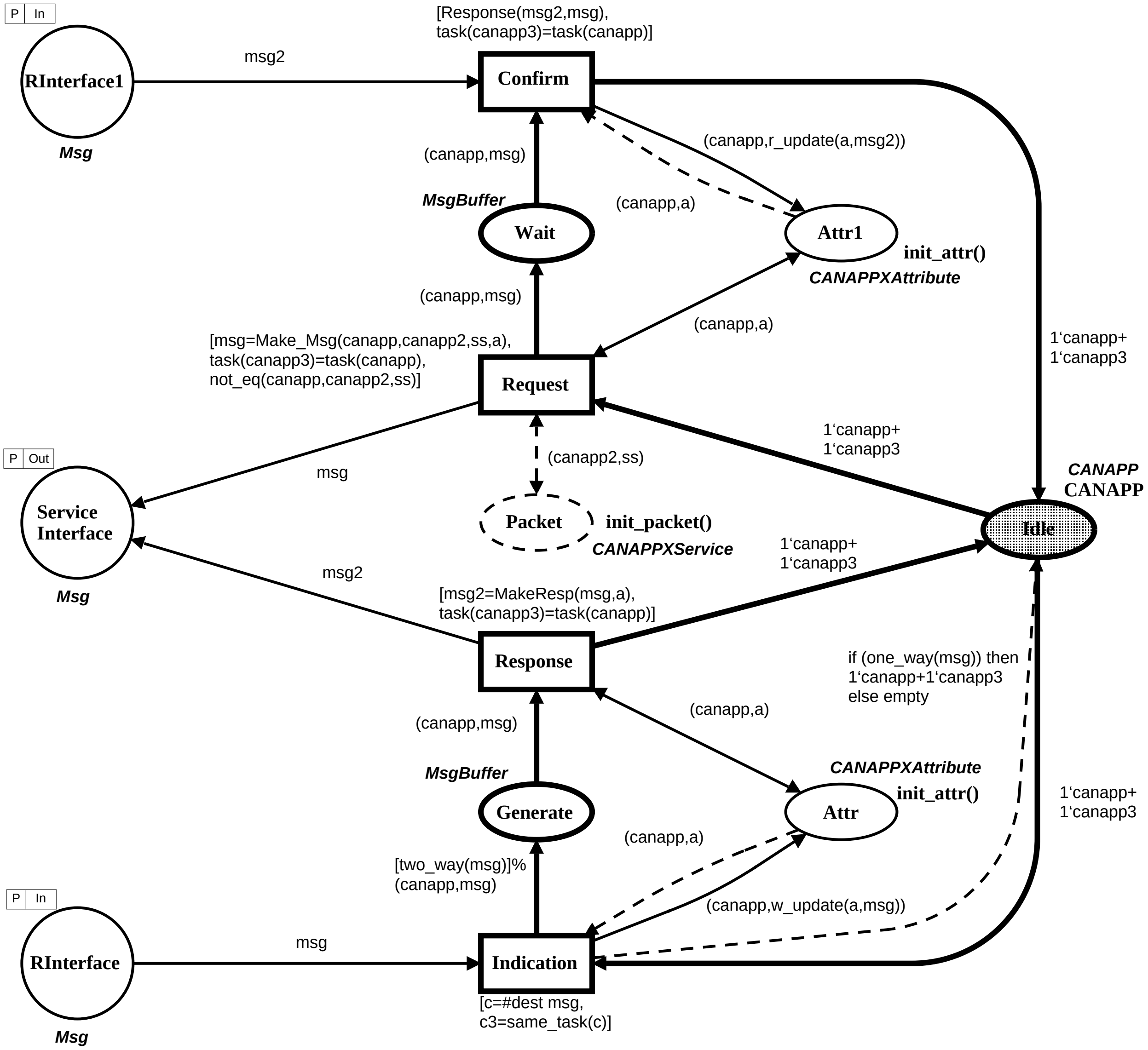


Fig. 14. Page InternalWaitpoint

- The pair modelling the sending CANAPP and the message sent is removed from place Wait modelling that the CANAPP no longer waits for a response.
- The token modelling the attribute of the CANAPP is removed from the place Attr. If the response message is a read message the attribute is updated to the value of the message. Otherwise a token with the original value of the attribute is put back on the place Attr.
- Two token corresponding to the two CANAPPs in the task are put back on place Idle modelling that both CANAPPs in the task are now idle and ready to either generate a message to another CANAPP in the system or to receive a message from another CANAPP.

When a message is sent to a CANAPP a token modelling the message is put on place RInteface. If the both of the CANAPPs in the task of the receiving CANAPP are idle the transition Indication can occur. The effect is

- The message is removed from place RInterface1.

- The tokens modelling the two CANAPPs in the task are removed from the place Idle modelling that they are no longer Idle but in the process of receiving a message.
- The token modelling the attribute of the receiving CANAPP is removed from the place Attr. If the message is a write message the attribute is updated to the value of the message. Otherwise a token with the original value of the attribute is put back on the place.
- If the message is a one-way message (i.e., a broadcast, event or command message) no response has to be generated and a two tokens modelling the two CANAPPs in the task are put on place Idle. If the message instead is a read, write or action message a response has to be generated and a pair of the CANAPP and the message is put on place Generate.

Now transition Response can occur. The effect is

- The pair is removed from the place Generate modelling that the response has been generated.
- Two tokens modelling the two CANAPPs in the task are put on place Idle modelling that the CANAPPs in the task are now idle and ready to either generate a message to another CANAPP in the system or to receive a message from another CANAPP.

Primary wait point In the primary wait point approach the return message is treated as an event and the sending CANAPP or any other CANAPPs in the task are not blocked. This means that a CANAPP can receive a request even if it is temporarily waiting for a response to a previously sent request.

A CPN model of the CANAPPs based on the primary wait point approach is modelled on page PrimaryWaitpoint which is shown in Fig. 15. The system modelled is the system in Fig. 9 of two modules each with two CANAPPs. CANAPP 1 and 2 cooperate in the same task in module 1 and CANAPP 3 and 4 cooperate in the same task in module 2.

The CPN model in Fig. 15 differs from the CPN model in Fig. 14 by having two idle places (Idle and Idle2) instead of just one. This means that a CANAPP can receive and send messages independently. Furthermore are the other CANAPP in the not blocked when a CANAPP is sending or receiving messages.

Properties of the model based on each of the two designs The model has been analysed with both of the two designs of the CANAPPs. The model has been analysed by means of the Design/CPN OG tool. The flowmeter system analysed is the system consisting of four CANAPPs - two in each module. Furthermore the total number of messages sent is limited in order to make the occurrence graphs finite. The analysis shows that even in this simple setting undesired properties of the two models can be found. Other configurations have been analysed. The results can be found in Sect. 4.

The internal wait point approach presents the problem, that it is not possible to access any CANAPP in a task, if one CANAPP is waiting for an asynchronous return message. This may result in deadlock situations as the following analysis shows.

The analysis reveals that the occurrence graph of the model, when the CANAPP design in Fig. 14 based on the internal wait point approach is used, has several dead markings. Not surprisingly the analysis shows that a deadlock occurs if two CANAPPs on different modules simultaneously send a request to each other. They both wait for a response but both are blocked and are not able to receive and process the request.

We will concentrate on another and more problematic kind of deadlock which is found in the analysis of the model based on the internal wait point approach. Figure 16 visualises a path to a node in the occurrence graph representing a dead marking of the CP-net.

CANAPP 2 sends a request to CANAPP 4. As the model is based on the internal wait point approach and CANAPP 1 and 2 are in the same task, both CANAPP 1 and 2 block until a response is received. CANAPP 3 sends a request to CANAPP 1. Of the same reason as before both CANAPP 3 and 4 block until a response is received. Now neither CANAPP 4 nor CANAPP 1 can receive the requests and the system is deadlocked.

All deadlocks are of course unwanted, but especially the kind of deadlock visualised in Fig. 16 is problematic in a flowmeter system because it it very hard to identify. The reason is that the CANAPPs are freely movable between the modules in the system but the concrete placement of the CANAPPs should not affect the functionality of the system. If CANAPP 4 in the example above instead is placed in a separate module (the flowmeter system consists of three modules instead of two) no deadlock would have occurred.

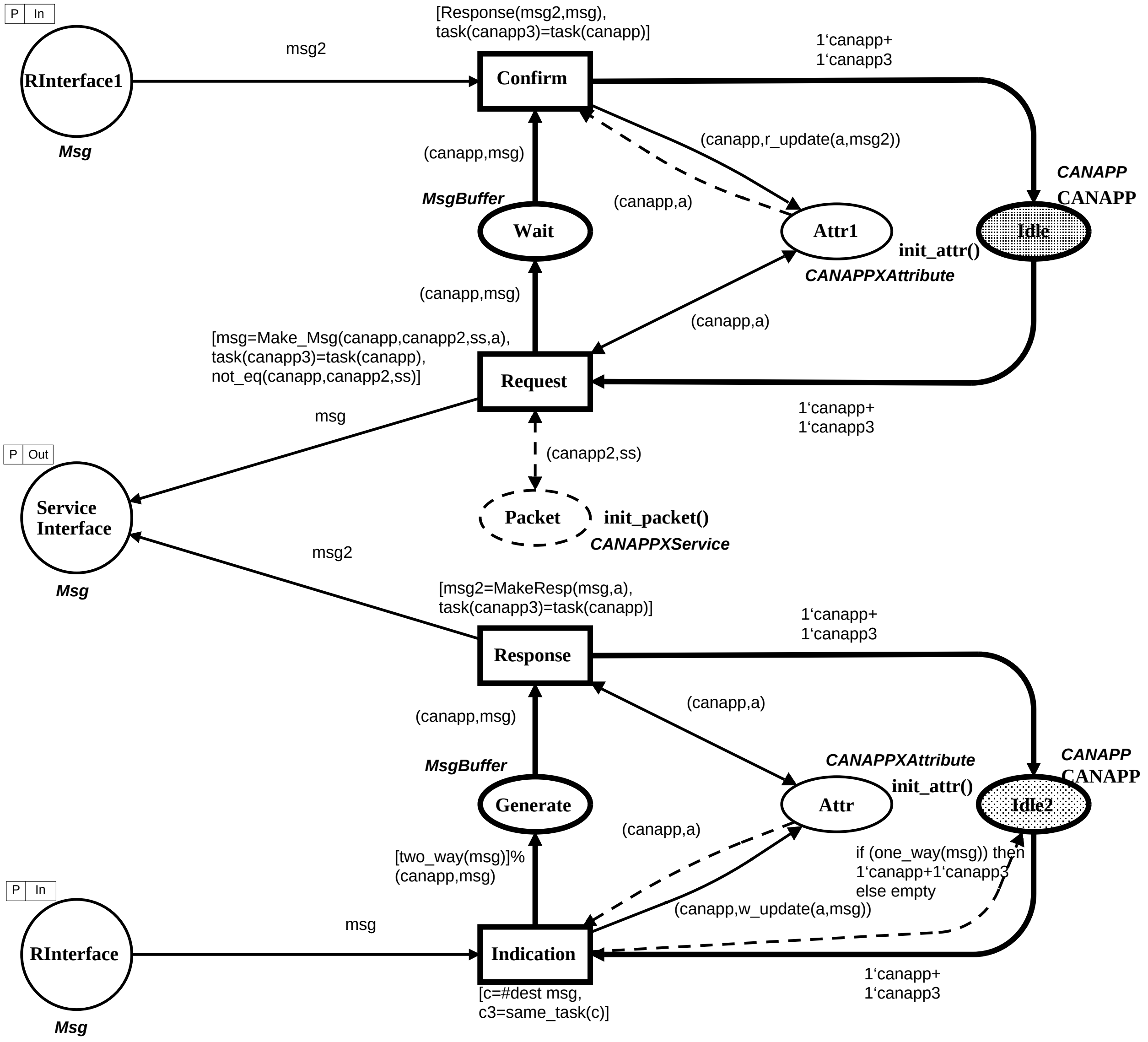


Fig. 15. Page PrimaryWaitpoint

One of the desired properties of a flowmeter system is, as listed in section 3.2, that it should not have any deadlocks - the analysis shows that the design of the CANAPPs based on the internal wait point approach does not fulfil this property.

The occurrence graph of the model, when the CANAPP design in Fig. 15 based on the primary wait point approach is used, only has dead markings due to the limitations of the number of messages sent. This means that no deadlock in the communication occurs - the design of the CANAPPs based on the primary wait point approach therefore fulfils the first desired property of the flowmeter system.

The second desired property as stated in section 3.2 is that the internal state of a CANAPP should not be corrupted while the CANAPP is involved in asynchronous communication, i.e., the attributes of a CANAPP

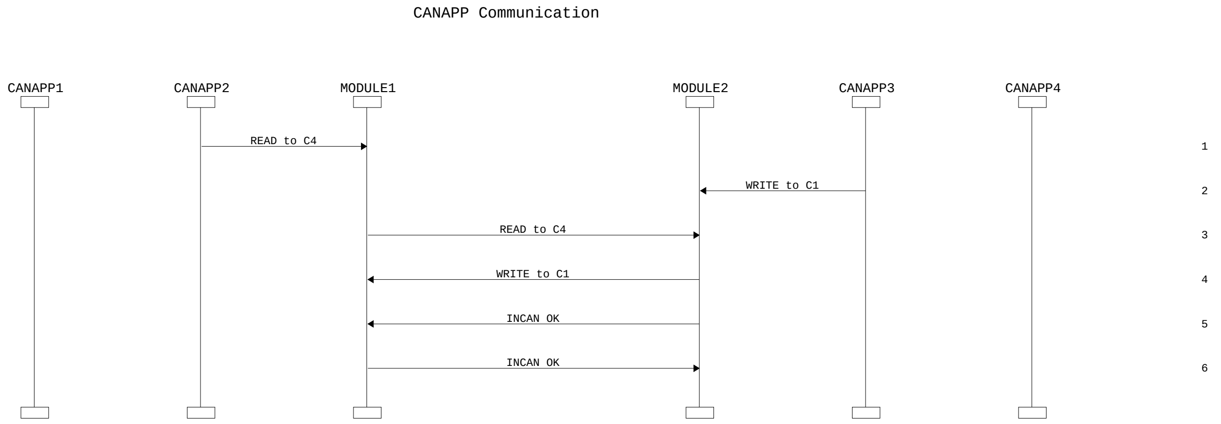


Fig. 16. Visualisation of a path to a dead marking

should not be changed while the CANAPP is waiting for a response to a previously sent request. In Fig. 17 the negation of the desired property is formulated in CPN ML [3]. All reachable markings are investigated via the predefined function `SearchAllNodes` and it is by a predicate function checked whether a CANAPP is waiting for a response in the marking and the marking has a successor marking in which the CANAPP is still waiting and the attributes have changed. If it is the case then the design does not fulfil the second desired property of the flowmeter system.

```
(*********************************************************************)
(* predicate testnext:                                               *)
(* A CANAPP is waiting in the marking and there is a successor marking *)
(* in which the CANAPP is still waiting but the attibutes have changed *)
(*********************************************************************)


fun testnext (n,l::li) =
  ((Mark.PrimaryWaitpoint’Wait 1 n)==(Mark.PrimaryWaitpoint’Wait 1 l))
      andalso
    ((Mark.PrimaryWaitpoint’Attr 1 n)<><>(Mark.PrimaryWaitpoint’Attr 1 l))
    orelse (testnext (n,li))
  | testnext (n,[]) = false;
```

Fig. 17. The negation of the second desired property formulated in CPN ML

A search in the occurrence graph of the model in Fig. 14 based on the internal wait point approach returns the empty list - the predicate function returns false for all reachable markings, which means that the negation of the property holds in all markings; the attributes of a CANAPP are not corrupted while the CANAPP is waiting for a response to a sent request.

A search in the occurrence graph of the model in Fig. 15 based on the primary wait point approach returns a non-empty list. This means that a corruption of the attributes of a CANAPP can occur while the CANAPP is doing asynchronous communication. The design of the CANAPPs based on the primary wait point approach therefore does not fulfil the second desired property of the flowmeter system. Fig. 18 shows a sequence of events which leads to a node in the occurrence graph which represents a marking in which the attribute of a CANAPP has been corrupted.

CANAPP 2 sends a request to read an attribute of CANAPP 3. As the model is based on the primary wait point approach CANAPP 2 is now waiting for a response but is not blocked. CANAPP 3 sends a request

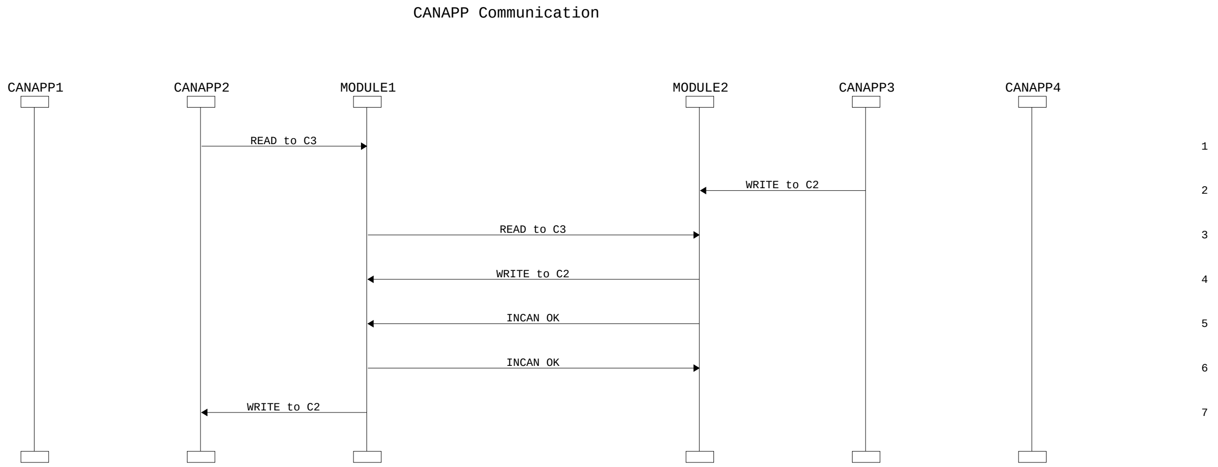


Fig. 18. Visualisation of a path to a marking where the attribute of a CANAPP has been corrupted

to write an attribute of CANAPP 2. CANAPP 2 receives the message from CANAPP 3 and changes its attribute. Now the attribute of CANAPP 2 has been corrupted while CANAPP 2 is waiting for a response to a previously sent message.

To summarise, the design based on the internal wait point approach violates the first property (no deadlocks) and fulfils the second (no corruption of attributes). The design based on the primary wait point approach on the other hand fulfils the first property but violates the second. Therefore, none of the two approaches above are suitable for the final design of the flowmeter system. In their construction of the final design the company has in fact chosen to work with a combination of the two described approaches, combining, as stated by the company, the best parts from both approaches.

4 Configurations

In Sect. 3.2 a small configuration of the flowmeter system was analysed. Even in this small setting deadlocks and corruption of attribute values could be detected thereby pointing out the problems of the, for the engineers in Danfoss, well known designs based on internal wait point and primary wait point. However the occurrence graphs get very large as the configurations analysed get bigger. This is as expected due to the asynchronous communication among the CANAPPs, which means that all possible interleavings are represented in the occurrence graph.

Below a overview of the size of the occurrence graph for different small configurations of the system is given to illustrate how the size of the occurrence graphs grows very fast as asynchronous communication over modules and more CANAPPs are introduced in the model. The results from Sect. 3.2 applies to all the configurations analysed.

Consider the flowmeter system with the configuration shown in Fig. 9, which is a system of two modules each containing two CANAPPs. The occurrence graph of this system where one message can be sent has 18,397 nodes and 48,930 arcs. If two messages can be sent, but with the restriction that only one CANAPP in each task is the sender of the messages, the occurrence graph has 45,939 nodes and 159,688 arcs. This means that the system with two messages (but the restriction that only one CANAPP in a task can be the sender of a message) instead of one message the occurrence graph grows by approximately a factor 3. The occurrence graph for the system with two messages without restrictions on the sending CANAPPs has more than 500,000 nodes and 1,500,000 arcs.

In this concrete project the use of CPNs in the design of the CANAPPs is well illustrated through small configurations. This means that the large occurrence graphs is not actually a problem. However, if a final design of a product is going to be based on a CPN model larger configurations have to be analysed to increase confidence in the system designed.

A subject for further work is therefore to use different techniques to reduce the size of the occurrence graph thereby making it possible to analyse even larger configurations. An interesting approach could be to consider the CANAPPs as symmetric in a task [6] and use stubborn sets [10] to reduce the size of the part of the occurrence graph related to the asynchronous communication among the CANAPPs.

5 Conclusions

The distributed flowmeter system studied in this paper is today a product sold by Danfoss.

The new design of a flowmeter system as a distributed system, relying on asynchronous communication among the processes in the system, raises a need for new ways of reasoning about the system and of validating the presence of the desired properties in a given flowmeter system. The project described in this paper demonstrates through concrete modelling and analysis of the constructed models by means of the Design/CPN OG tool that Coloured Petri Nets and occurrence graphs may indeed be relevant new techniques to be used in industrial settings, not only related to this product but also related to the design process of future products.

The deadlock found by Danfoss in the design phase of the product was found in a practical test of the product. The analysis performed in this project is able to detect the same deadlock and also show how this kind of deadlocks is unavoidable if the design of the CANAPPs is based on the internal wait point approach. Furthermore the project shows that the desired properties of a product as stated by the producer can be expressed and easily analysed when a CPN model of the system is constructed.

Also the graphical qualities of Coloured Petri Nets have proven to function well among engineers and between engineers and other groups involved in the design process.

6 Acknowledgements

I would like to thank Danfoss Instrumentation, especially Arne Peters, for their contributions to the project. I would also like to thank Lone Haudrum Olesen for her work on a early CPN model of the protocol.

References

[1] M.Awad, J.Kuusela and J.Ziegler, *Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion*, Prentice Hall, 1996.

[2] S.Christensen. *Design/CPN Message Sequence Charts Library Manual.* Computer Science Department, University of Aarhus, Denmark.

[3] K.Jensen, S.Christensen, P.Huber & M.Holla. *Design/CPN Reference Manual.* Computer Science Department, University of Aarhus, Denmark. Online: `http://www.daimi.au.dk/designCPN`

[4] J.D.Day & H.Zimmermann. *The OSI Reference Model*, Proceedings of the IEEE, vol.71, Dec.1983.

[5] K.Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol.1, Basic Concepts.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[6] K.Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol.2, Analysis Methods.* Monographs in Theoretical Computer Science. Springer-Verlag, 1994.

[7] K.Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol.3, Practical Use.* Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[8] K.Jensen, S.Christensen and L.M.Kristensen, *Design/CPN Occurrence Graph Manual*, Computer Science Department, University of Aarhus, 1996. Online: `http://www.daimi.au.dk/designCPN`

[9] W.Lawrenz, *CAN Controller Area Network, Grundlagen und Praxis*, Hüttig Buch Verlag, Heidelberg, 1994.

[10] A.Valmari, *Error Detection by Reduced Reachability Graph Generation*, Proceedings of the 9th European Workshop on Application and Theory of Petri Nets, pp.95-112.

Performance Prediction Model Generator Powered by Occurrence Graph Analyzer of Design/CPN*

Insub Shin and Alexander H. Levis

System Architectures Laboratory
School of Information Technology and Engineering
George Mason Unversity, MS 4D2
Fairfax, VA 22030, USA
E-mail: {ishin,alevis}@gmu.edu

ABSTRACT

This paper describes a methodology for generating a performance prediction model so that both qualitative (logical correctness) and quantitative (timeliness) properties of a real-time system can be evaluated. The methodology uses two types of tools: the reachability analysis tool of a Petri net and a discrete event-driven simulation technique of a queueing net. Typically, the complete specification of a complex system is obtained through a combination of multiple architectural perspectives. This paper focuses on the synthesis of a functional architecture and a physical architecture. Given that the logical behavior of a real-time system is described as a functional architecture, the logical property is verified using the Occurrence Graph Analyzer of Design/CPN [1]. Then, the timing property of the system, running on a communication network, is measured by simulation using NS (Network Simulator [2]). During the simulation, the partially ordered messages from the logical model are processed in the total order in the network traffic-scheduling scheme. The uniqueness of this methodology is the “off-line” simulation of both models, rather than “run-time” simulation. The techniques developed offer a means to use the Petri net properties for analysis. This is not possible in run-time simulation due to the open interfaces between the two models. This methodology provides the strengths of both a formal method and simulation techniques for performance evaluation of a real time system. The paper includes a description of the automated algorithms that transform a logical model (i.e., untimed Petri net model) of a real-time system into a performance prediction model (i.e., timed Petri net model).

1. INTRODUCTION

1.1 Motivation

In the systems engineering approach, a mission is decomposed into tasks, and resources are allocated (e.g., sensors, human decision makers, computing processors, communications networks, and weapon systems, etc.) to perform the task. The mission is structured as a set of tasks connected by synchronous/asynchronous channels and accomplished by the execution of tasks in series and/or in parallel. The execution of a task is constrained by external stimuli such as higher authority’s mission requirements or by the capabilities of assigned resources. For a time critical C3 (command and control, communication) system, analysis of the timing of each task and time management in a series of task executions are essential to guarantee that the system responds at the right time, or before but not after a due time.

* This work was supported in part by the Air Force Office of Scientific Research under Grant No. F49620-98-1-0179

In general, a real time system is characterized by its timely response to external stimuli [3]. A response consists of a series of task executions. A task execution is usually characterized by its start-time, execution-time, and deadline [4]. Timing constraints are then postulated between the external stimuli and the response. Those constraints express properties of end-to-end timing propagation delay. Maintaining functionally correct end-to-end values may involve a large set of interacting components. To ensure that the end-to-end constraints are satisfied, each of these components will be subject to its own intermediate timing constraints [5]. For a distributed decision-making organization dealing with time critical missions, the estimates of the elapsed time of each interacting component and the message delay can play a critical role in developing combat engagement rules, establishing decision thresholds, and battle time management.

At the early stage of the system development cycle, however, there are significant uncertainties in the performance parameters, task execution time, and network connection delays. They are due to incomplete system specifications and/or complexity of the problem. If the physical communication link between tasks is simply an immediately adjacent transmission link, the modeling effort may be easy. When multiple connections share common communications network resources other than single, immediate, adjacent transmission link, the cost of modeling is very high. In some cases, modeling the contention of communication resources may be practically impossible. The performance model for time critical missions must be well formalized. It also requires accurate timing estimate methods.

1.2 Approach

A C3 system for a high level military organization is a system-of-systems consisting of heterogeneous subsystems operating in distributed operational environments. The property or behavior of a system is specified and expressed in multiple perspectives or views. Figure 1 shows an example of those architectures.

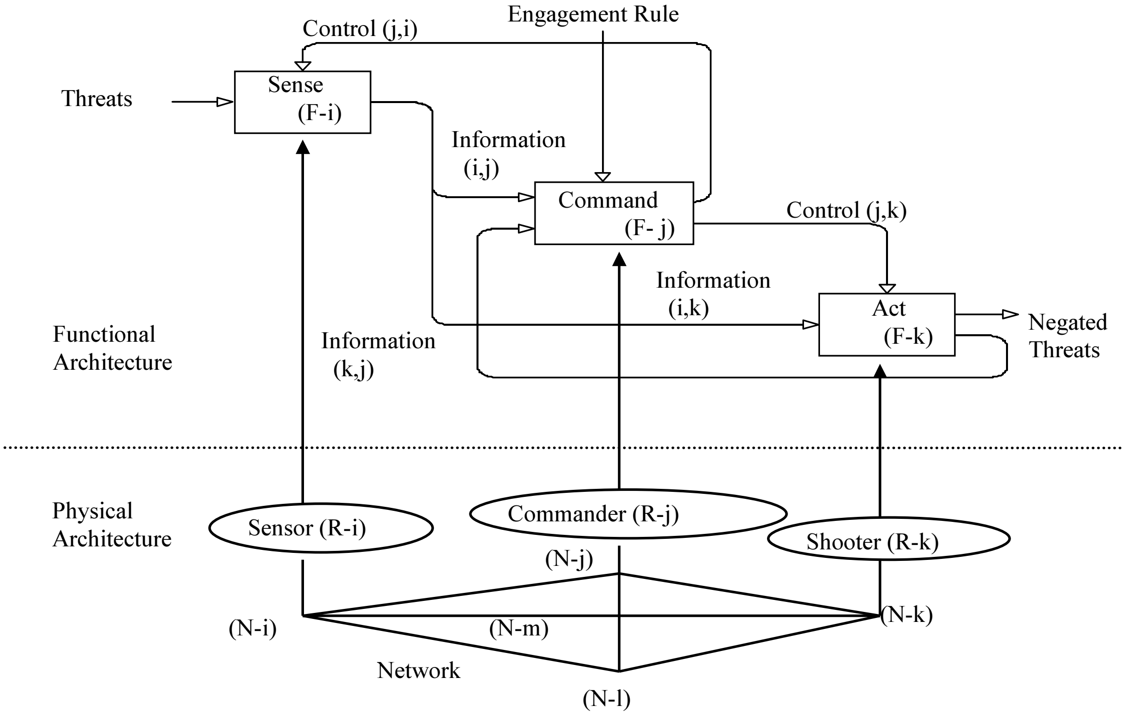


Figure 1. An Example of System Architecture

The logical behavior (e.g., computational correctness or control structure) of a system is specified in a functional architecture and the supporting resources are specified in a physical architecture. Predicting the performance of the system requires synthesis of the multiple views. Both functional and physical architectures describe or specify the same system. So they are not separate systems in a global sense. In this methodology, however, both architectures are considered as separate state machines. Then, the performance parameters are obtained by exchanging the necessary state information between two machines.

Developing an executable model is a way to predict the performance of a real time system. Operational formalisms [6,7] such as Petri nets and State Machines are suitable to specify and express executable models. One of the most important principles of military doctrine is synchronization. Using the Petri net formalism, the synchronization problem is resolved by its structure. In this methodology, the target performance prediction model is developed as a Timed Petri net. However, the methodology does not directly model the communications demands between tasks within the Petri net. The time values are estimated by simulations of a separate communications network model using realistic traffic messages that are generated in partial order from an untimed Petri net. We use the occurrence graph of a Petri net to generate those messages. While simulating the communication network model, the partially ordered messages shall be processed in a total order in its traffic-scheduling scheme.

1.3 Scope of Work

Finding all sequences of message passing from a full occurrence graph is very costly since the full occurrence graph contains all the reachable markings. However, if the functional architecture is specified as a deterministic conflict-free Petri net, the net has a single final state after firing all enabled transitions given initial markings. Also the markings of the net after firing a set of transitions are same regardless of the sequence of firings. So, if we can develop a functional architecture as a conflict-free Petri net, a partial occurrence graph with one sequence of firings can provide the order of message passing in the system.

Once a conflict-free Petri net is created using Design/CPN, the tool developed generates the whole state space information automatically. Next, the network topology generation tool is invoked to build the physical architecture. Once the physical architecture (communication network model) is created, the Network Simulator reads the state information, runs the simulation, and generates delay values. Finally, these values are introduced as an arc inscription to express the time delays in the Timed Petri net (performance prediction model).

The methodology has been implemented under the assumption that the system is an information system with a conflict free net structure. But the methodology and tools developed are neither limited to information systems nor conflict-free nets. If the Petri net is not conflict-free and if there exist multiple final states, the performance parameters will be obtained by repeated simulation. The tool developed can compute and select either a shortest path or a longest path from initial state to final states.

2. A METHODOLOGY TO DEVELOP A PERFORMANCE PREDICTION MODEL

The performance prediction model is generated using 6 major steps:

- Develop executable functional architecture
- Identify communications service access interfaces
- Create message dependency relations
- Simulate communications network
- Estimate network delays
- Introduce time to the executable functional architecture.

2.1 Step 1: Develop Executable Functional Architecture

The first step in the development of a functional architecture is the functional decomposition. The functional decomposition can be carried to any level of detail. It may depend on the level of abstraction and modeling purpose. In practical terms, it is carried out to the point that the lowest level tasks must be executed by a single resource [8]. To estimate the network delays, we need to identify the physical resources in a communications network architecture. So, the decomposition process may stop at the level at which the physical resources can be identified at the communication network and its elemental performance parameters are known.

Suppose that the functional architecture in Figure 1 describes a system to be used to clear threats in which a man-machine decision making system makes decisions based on input sensor values to respond to the threats. A corresponding Petri net structure can be depicted as shown in Figure 2.

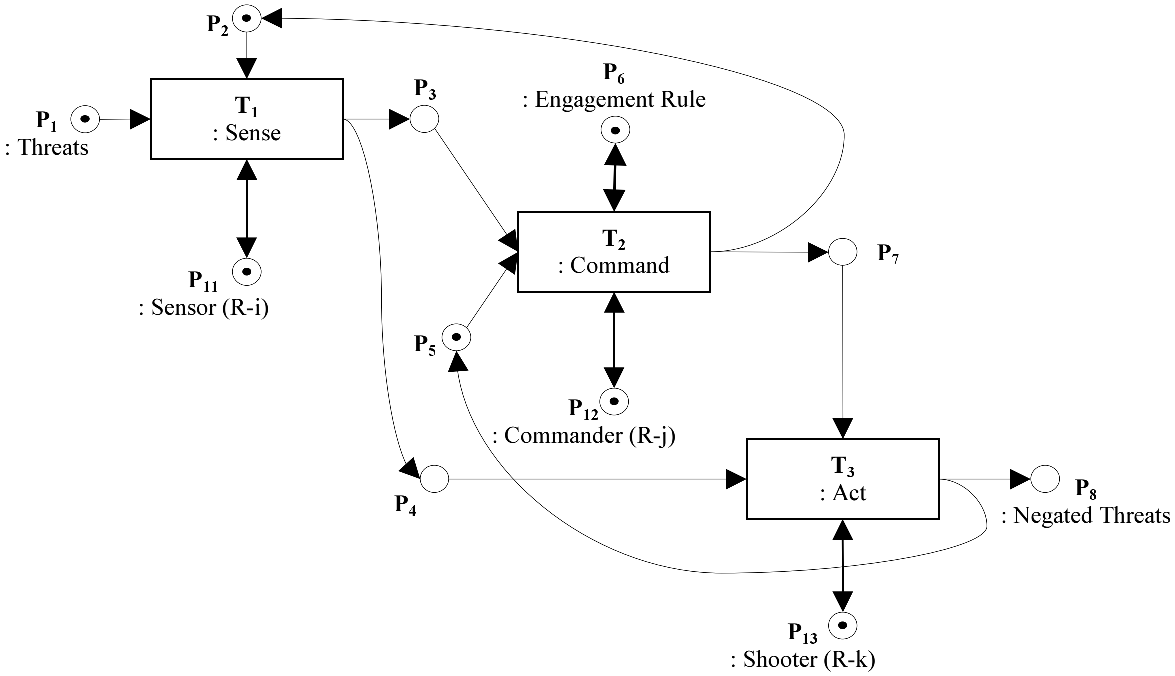


Figure 2. An Example of Executable Functional Architecture

Once a set of tasks is represented by "transitions", the allocation of resources to perform the tasks is represented by tokens at "places". Places P11, P12, and P13 in Figure 2 are used for assignment of resources to tasks. Resources R-i, R-j, and R-k in Figure 1 are instances of resources. These physical resources will have dual roles: one role as computing devices in the functional architecture and the other role as traffic generators in the communications network architecture. In Figure 1, Resources R-i, R-j, and R-k are the physical resources that play the dual roles. As a computing device, the physical resource can be interpreted as a server to carry out a task. Adding these physical resources to the executable functional architecture denotes that the task uses the service of the resource to execute the task. If those resources at places P11, P12, and P13, are used only one time and consumed, those thick arcs in Figure 2 will be uni-directed arcs from P11 to T1, P12 to T2, and P13 to T2. Similarly, if the engagement rule at place P6 is used only one time and consumed, the thick arc from P6 to T2 will be uni-directed. If the rule is used repeatedly and updated dynamically, the arc must express self-loop. Also the rule must have a time stamp and hence the color set of place P6 must be timed.

2.2 Step 2: Identify Communications Service Access Interfaces

The communication network simulation model can be generalized as a network of a set of nodes, namely N_1, N_2, …, $N_{n,}$ and a set of links which are node pairs (i,j) between node i and node j. Each link has a capacity $C_{i,j}$, counted in bandwidth units. Links are undirected; (i,j) and (j,i) denote the same link. In this methodology, the communication network consists of four layers: Physical Layer, Data Link Layer, Network Layer, and all the Higher Layers [9]. The Higher Layers layer generates the application traffic, and the other 3 layers process this traffic. The message traffic has 7 attributes; message identification (ID) number, source node, destination node, message size, priority, protocol, and message dependency relation. The physical resources allocated to the functional architecture model generate the application network traffic. The same physical resources connected to a communication network model as its leaf nodes can be used to identify the communication service access points in the network as shown in Figure 3.

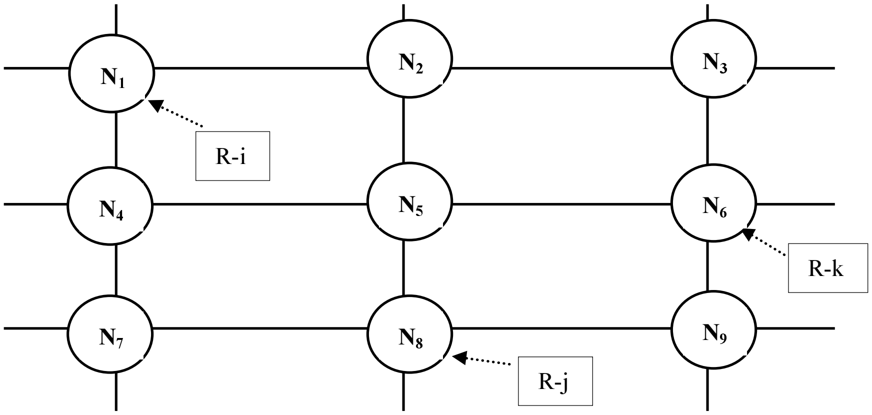


Figure 3. Communication Network Access Points

The appearance of tokens in places P3, P4, P7, P2, and P5 in Figure 2 represents the information flow between tasks: ‘information (i,j)’, ‘information (i,k)’, ‘control (j,k)’, ‘control (j,i)’, and ‘information (k, j)’ in Figure 1. In turn, traffic messages with source-destination node pairs (N_1, N_8), (N_1, N_6), (N_8, N_6), (N_8, N_1), and (N_6, N_8) in Figure 3 represent the physical message passing, respectively.

2.3 Step 3: Create Message Dependency Relations

When the firing sequence of transitions is known, we can determine the sequence of message passing. When the firing sequence is not known, we have to consider all possible sequences of transitions. These all possible sequences are captured in a full occurrence graph of a Petri net. Generating a full occurrence graph and searching it exhaustively are not practically acceptable when the state space is large.

In a conflict free net, firing of one transition does not disable other transitions. Once tokens in the preset of a transition fulfill the enabling conditions, the transition will definitely fire under a “must” firing rule. So, even though multiple transitions in a Petri net are enabled concurrently, the final markings of the net are the same regardless of the firing order of the concurrently enabled transitions. Also, the group of messages that are produced after firing of concurrently enabled transitions have no precedence relations inside the group of messages. Message dependency relations occur only between different sets of messages that are produced after subsequent firings other than concurrent firings. Thus, given a conflict free net with initial markings, one sequence of occurrence of markings in a partial occurrence graph has all the information necessary to count the number of messages in the system and their precedence relations. If we generate a partial occurrence graph in the depth-first search algorithm and there is no circuit in the net, it is not costly to capture the message dependency relations, and the methodology developed can be used effectively.

If a Petri net is not conflict free, however, the net may have multiple final markings. Even if the net has a single final marking, there may exist multiple sequences of occurrences of markings that have different precedence relations amongst different sets of messages. To handle those non-conflict free nets, the methodology may require repeated simulations of the communications network corresponding to the different paths in the full occurrence graph. One alternative way to estimate network delays between all transitions is to set the initial marking so that all transitions are covered in each path of the full occurrence graph. Then, the methodology can estimate the network delay between tasks by selecting one path. Assume that the communications network has the following characteristics: (1) network delays are longer at its heavier traffic loads than its lighter traffic loads, and (2) the degree of traffic loads is defined as the amount (e.g., number or size) of messages in a given time interval. By selecting the path that has the maximum number of messages (most likely the longest path), the methodology can estimate the upper bound of network delays given the initial marking. Similarly, by selecting the path that has the minimum number of messages (most likely the shortest path), the methodology can estimate the lower bound of network delays given the initial marking. Instead of repeated simulations of a communications network as many as the number of paths, the alternative method may be efficiently used for a worst-case timing analysis.

The newly generated tokens are identified by subtraction of multi-sets between subsequent markings. However, untimed occurrence graphs can not distinguish the new tokens in a self-loop. In a model of a real-time system, new tokens in a self-loop can be distinguished by their time stamps. So we can have a tool to distinguish new tokens in a self-loop by adding a counter or a logical time [10] unit. In our methodology, we used a logical time unit because it is easy to analyze the logical behavior of a system. Using the Occurrence Graph Analyzer (OGA) embedded in Design/CPN, a timed Occurrence Graph can be converted into an untimed Occurrence Graph easily. By adding time delay “1”on the arc from transition “T1” to place “P11”, if necessary, we can distinguish the new tokens in the self-loop. If we generate an occurrence graph after assigning time delays to transitions, the occurrence graph becomes a timed occurrence graph. But the order of message passing does not change in a conflict free net. Also a timed occurrence graph is a subset of an untimed occurrence graph [11]. So in a conflict free net, we can extract the number of messages and their precedence relations without affecting the logical behavior. Now we are ready to extract the message precedence relations by a simple set operation; subtraction of multi-sets.

Suppose that an Ordinary Petri net is represented by a quadruple (P, T, I, O) where:

P = $\{p_1, p_2, .., p_n\}$ is a finite set of places.

T = $\{t_1, t_2, ..., t_m\}$ is a finite set of transitions.

I is an input mapping P x T → {0, 1} corresponding to the set of directed arcs from P to T.

O is an output mapping T x P → {0, 1} corresponding to the set of directed arcs from T to P.

Given the current marking M, firing of the transition *t* results in a new marking M' where:

$M'(p_i) = M(p_i) - I(p_i, t) + O(t, p_i)$, For i = 1, ..., n.

So, if we perform an operation of subtraction; $M'(p_i) - M(p_i)$, the computation results in {1, 0, –1} from

$O(t, p_i)$	-	$I(p_i, t)$	=	$M'(p_i) - M(p_i)$
0		0		0
0		1		-1
1		0		+1
1		1		0

The “+” and “-“ signs tell whether the token is a newly placed one or a removed one. The “0” at the fourth row occurs when the structure has a self-loop. It must be detected because one token is removed and one token is newly generated. But it is not distinguished from the “0” at the first row. That represents no change after firing of the transition. By adding time stamps, the “0” at the fourth row can be distinguished from the “0” at the first row.

Let the service times of physical resources R-i, R-j, and R-k (i.e., an execution time of a task executor) be τ_{11}, τ_{12}, τ_{13} that are associated with each place P11, P12, and P13, respectively. Also let the network delays of messages *m*(p3), *m*(p4), *m*(p7), *m*(p2), and *m*(p5) be δ_3, δ_4, δ_7, δ_2, and δ_5 associated with each place P3, P4, P7, P2, and P5, respectively. A message dependency graph corresponding to Figure 2 can be depicted as shown in Figure 4.

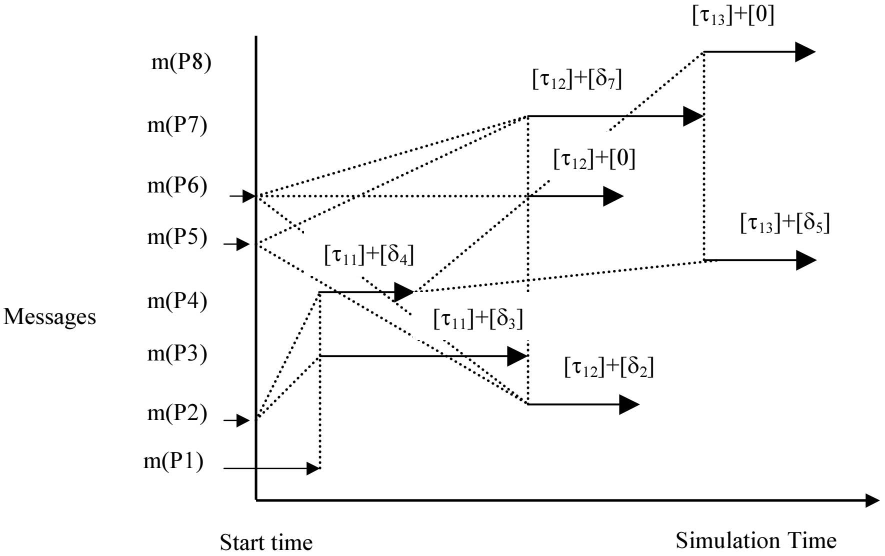


Figure 4. An Example of Message Dependency Graph

2.4 Step 4: Simulate Communications Network

The role of message dependency (or precedence) relations is to determine the order of message passing to schedule every message over a single time line during the simulation. If we know the network delay, the messages can be linearly ordered over a single time line: total order. Since the delays are not known at the logical model, the messages from the logical model only show a partial order. Network delays are estimated during simulation and they are used for scheduling the next messages.

The figure shows that two messages *m*(p3) and *m*(p4) (i.e., tokens produced at places P3 and P4, respectively, after firing of transition T1) will be triggered by two initial messages *m*(p1) and *m*(p2). In the figure, the initial time stamp of message *m*(p1) is bigger than the initial time stamp of message *m*(p2). If transition T1 takes an action delay (e.g., τ11), messages *m*(p3) and *m*(p4) must be generated after the action delay. In turn, each message *m*(p3) and *m*(p4) will be used for triggering the next messages. The second message of place P2 *m*(p2) will be generated after the arrival of *m*(p3) when the time values of initial messages *m*(p5) and *m*(p6) are passed. Place P6 in Figure 2 is self-looped. So message *m*(p6) does not involve network delay (0 delay). Message *m*(p7) will be generated after arrivals of *m*(p3), *m*(p5), and *m*(p6) at the latest arrival time of all 3 messages. Finally, messages *m*(p5) and *m*(p8) will be triggered by two messages *m*(p7) and *m*(p4). In this specific example, it shows a case that message *m*(p7) generated at resource R-j (i.e., source node N8) arrives at resource R-k (i.e., destination node N6) later than message *m*(p4) generated at resource R-i (i.e., source node N1) arrives at resource R-k (i.e., destination node N6). But *m*(p8) has no network delay (0 delay) because it is a sink of information in the model.

Table 1 is an example of a traffic log file. Instead of having a message generation time, which is usually specified by a probabilistic form such as Poisson or Exponential in a typical communication network simulation approach, each message is specified by its dependency relations.

Table 1. Message Traffic Log

ID	source	destination	size	priority	protocol	dependency
1: *m*(p1)	INITIAL	N1	1000byte	0	UDP	[]
2: *m*(p2)	INITIAL	N1	1000byte	0	UDP	[]
3: *m*(p5)	INITIAL	N8	1000byte	0	UDP	[]
4: *m*(p6)	INITIAL	N8	1000byte	0	UDP	[]
5: *m*(p3)	N1	N8	1000byte	0	TCP	[1,2]
6: *m*(p4)	N1	N6	1000byte	0	TCP	[1,2]
7: *m*(p2)	N8	N6	1000byte	0	TCP	[3,4,5]
8: *m*(p6)	N8	N8	1000byte	0	TCP	[3,4,5]
9: *m*(p7)	N8	N6	1000byte	0	TCP	[3,4,5]
10: *m*(p5)	N6	N8	1000byte	0	TCP	[6,9]
11: *m*(p8)	N6	EXTERNAL	1000byte	0	UDP	[6,9]

2.5 Step 5: Estimate Network Delays

After running the simulation, two types of delay information can be obtained. One is the statistical data, and the other is a pair of bounded network delays, if it exists. Assume that the communications network has the characteristics explained in section 2.3. If the delays with light traffic loads (lower bound network delays) are smaller than the delays with heavy traffic loads (upper bound network delays), and if they are consistent for various degrees of traffic loads, the two assumptions regarding the characteristics of the communications network may be considered reasonable. Then, we can determine the minimum and maximum network delay pairs for each message and use them to verify the end-to-end timing constraints.

If the simulation results show a random pattern, we can not determine the bounds. Instead, the network delay pattern will be formalized in a statistical form such as average delay time and variance. This probability distribution does not express the lower or upper bound delays. Consequently it is not decidable whether the end-to-end timing constraint of a system is met or not. The timing behavior of the system will be predicted by only repeated simulations of the Petri net model or the communications network model.

2.6 Step 6: Introduce Time to Executable Functional Architecture

Once we obtain the network delay pattern with either a probabilistic distribution or bounded delays, these network delay values and service times are specified as arc expressions of a Petri net. The resulting Timed Petri net model has all information to predict the performance of the system. This is the last step to develop the performance prediction model. If we care only about the end-to-end timing delay, this step is not necessary since the output data from communication model has the same information. However, this step helps to analyze the logical behavior and timing behavior together using the resulting timed occurrence graph.

3. DESIGN OF PERFORMANCE PREDICTION MODEL GENERATOR

The Performance Prediction Model Generator (P2MG) was designed based on the requirements so that it can be used efficiently and effectively for performance prediction of a real-time system for various scenarios or design options. Figure 5 shows the workflow model. In this methodology, the functional models are assumed to exist.

The scenario is expressed as initial conditions of the functional model (i.e., initial markings of the Petri net). The state of a functional model is represented by an occurrence graph of the Petri net. Vector $[m(p_1), m(p_2), m(p_3),..]^T$ in the functional model represents the state of a system in the functional view. Using the OGA of Design/CPN, the logical behavior of the system is verified first. Then, from the occurrence graph, the order of message passing is captured and generated in a traffic log file. The order of message passing [{a,b}→{c,d}→{e}, ...] has information about the number of messages and the precedence relations between sets of messages. The communication network simulator processes each message in a sequential order over a single time line, and generates all the delay values for each message. The result shows node processing delays (i.e., pre- and post- processing delays) and network transmission delays for all messages. Finally, all the delay values are mapped back to the functional model for analysis of both logical and timing properties using a timed occurrence graph.

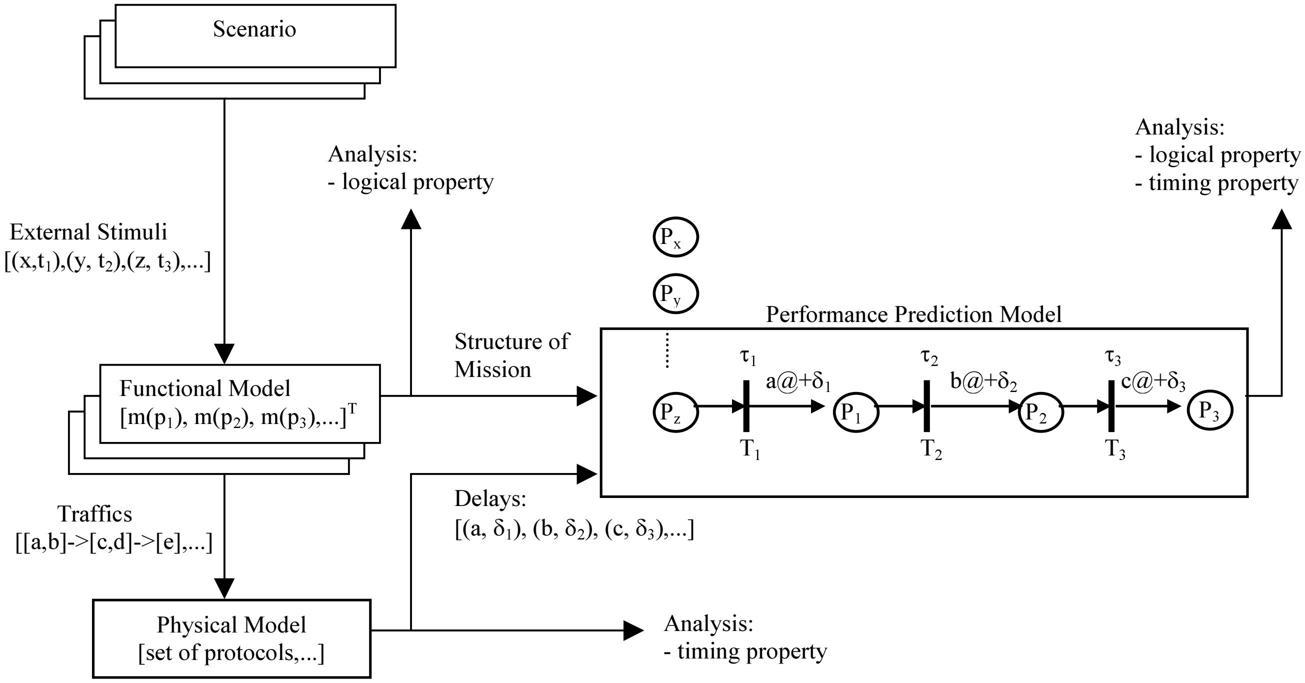


Figure 5. Workflow Model for P2MG

To implement the methodology, we used Design/CPN for the description of the functional model and NS for the description of the physical model. The P2MG consists of 5 major software components; Reachability Tree Generator (RTG), Message Generator (MG), Performance Prediction Model Converter (PPMC), Network Topology Generator (NTG), and Communication Network Simulation Script (CNSS). The four components RTG, MG, PPMC and NTG were implemented using the ML (Meta Language) imbedded in Design/CPN. CNSS was implemented using the TCL (Tool Command Language) embedded in NS.

To describe each component, we will use a simple Petri net as shown in Figure 6 (with the global declaration of Table 2). The Petri net corresponds to Figure 1, but very simplified for ease of understanding. The structure of the model describes a typical command and control process. Let us develop an arbitrary scenario. The system senses threats using radar. It scans the battlefield periodically, 30 times per minute (e.g., the minimum action delay of sensing is 0 seconds and the maximum action delay is 2 seconds). It disseminates the tactical picture to its missile Fire Direction Center (FDC). Once the FDC receives information of a new threat, the FDC checks the availability of weapons, and issues an order to a Missile Launcher following the engagement rule, and issues the surveillance directive. This action is invoked automatically by embedded algorithms in the automated missile controller. It takes some time for data fusion, but the delay is assumed to depend only on the size of the tactical picture received and the bit processing power of the computing device. The missile launcher platform requires 10 seconds warming time to fire a missile (e.g., minimum delay of 0 seconds when it is already warm and maximum action delay of 10 seconds). All messages in the system are 1000 bytes (8000bits) long.

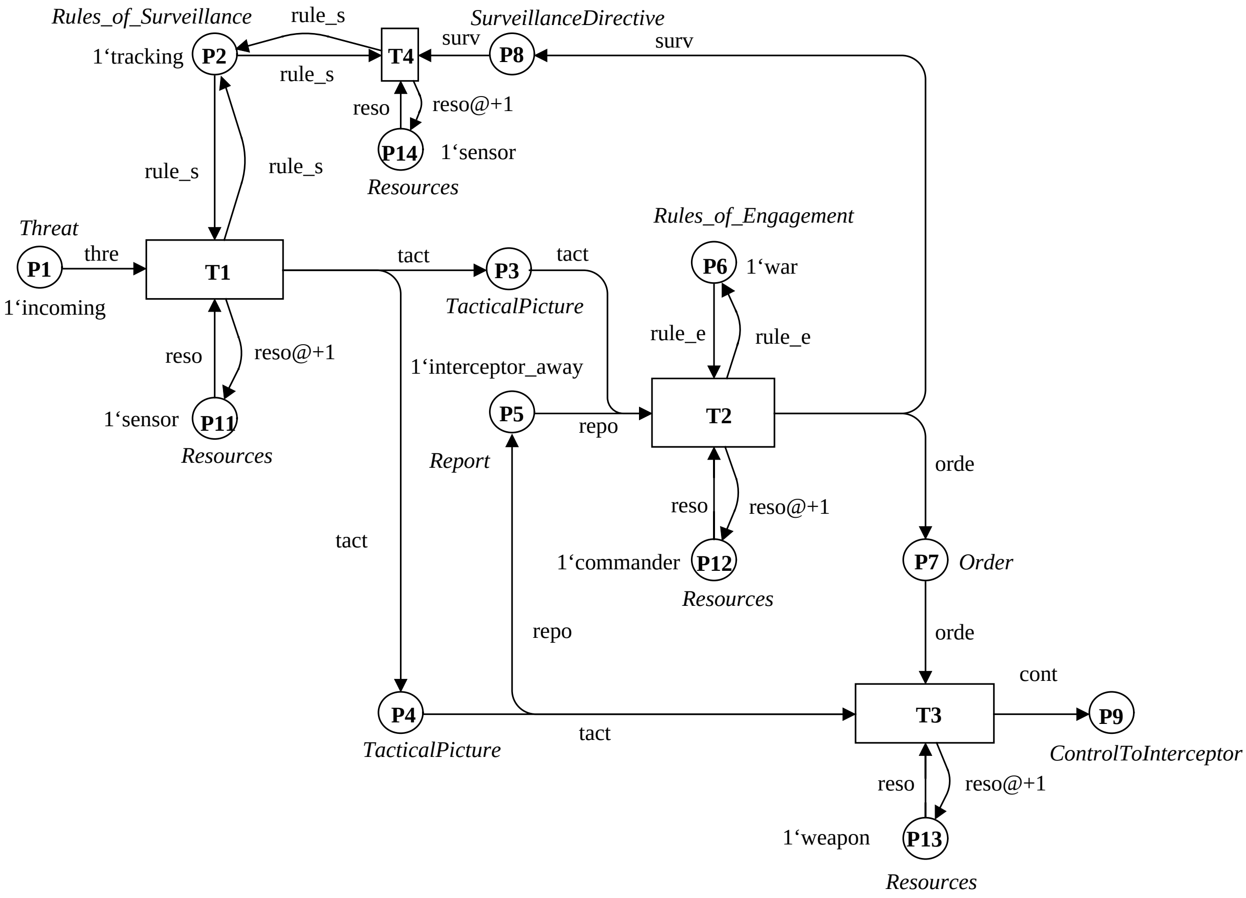


Figure 6. An Example of Executable Functional Architecture

Table 2. Global Declaration Node

```
(* global declaration *)
color Rules_of_Engagement = with war  ;
color Rules_of_Surveillance = with tracking ;
color Threat = with incoming  timed ;
color SurveillanceDirective = with default  timed ;
color TacticalPicture = with new timed ;
color Order = with intercept  timed ;
color Report = with interceptor_away  timed ;
color ControlToInterceptor = with take_off  timed ;
color Resources = with sensor | commander | weapon timed ;
var rule_e : Rules_of_Engagement;
var rule_s : Rules_of_Surveillance;
var thre : Threat;
var surv : SurveillanceDirective ;
var tact : TacticalPicture;
var orde : Order;
var repo : Report ;
var cont : ControlToInterceptor;
var reso : Resources ;
```

3.1 Reachability Tree Generator (RTG)

The RTG primarily uses software function codes provided by the OGA of Design/CPN. However, this component was implemented to be interactive with users. It generates a reachability tree, searches final states, and calculates the shortest and longest paths from the initial state to final states. This occurrence graph (OG) generation module generates a partial reachability tree until it finds at least one final state. If the net is conflict free, this partial reachability tree can be used to capture all the necessary state information. The current version of the OGA in Design/CPN uses "breadth-first" search algorithm to build the OG. But it provides "narrow" search; a kind of "depth-first" search. Using this option can reduce computation time to find a final state. If the net is not conflict free, the user will be provided an option to search all final states and select one of the paths from the initial state to multiple states.

3.2 Message Generator (MG)

The main feature of MG is to generate the order of message passing containing their precedence relations. It generates a traffic log file in the format shown in Table 1 so that it can be used by the CNSS. MG provides the user with a Graphic User Interface (GUI) developed in the Design/CPN environment in order to obtain information necessary to make the traffic log file.

The MG searches what resources are allocated to which transitions in order to generate its source and destination nodes of the communication network. The mapping mechanism is based on the dual roles of resources as described in Section 2 (places P11, P12, and P13 in Figure 2 are placeholders to map functional architecture and physical architecture). The process of selecting alternative resources is an issue of resource allocation or an issue of design. So we can compare the alternatives in order to execute the same scenario. This GUI helps to select instances of resources that are connected to the communication network as leaf nodes.

At this module, the user enters the attributes of actors, which are the performers of tasks. The main attribute is the action time. This action time is differentiated from the node processing delay and network transmission delays. Basically, those node processing and network delays are associated with the capability of resources whose performance parameters are specified in the physical model. The delay of an actor is associated with the delay of task executor(s). If every task is an autonomous data processing task, the actor's delay may be set to "0". Once this attribute is obtained, it is maintained at a special region of each transition and arc. It can be alternatively selected depending on scenarios. Another attribute is the demand protocol to transmit each message.

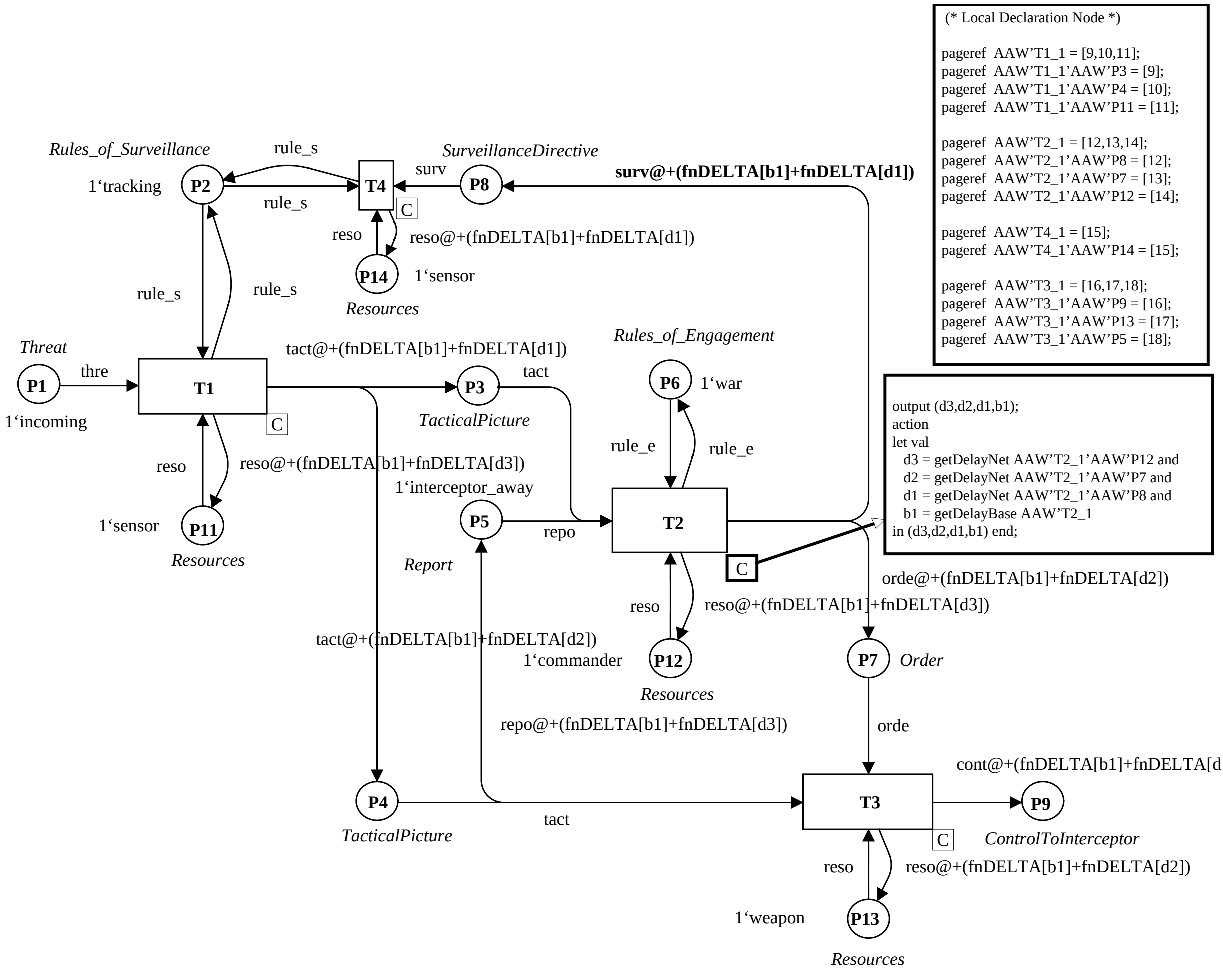
3.3 Performance Prediction Model Converter (PPMC)

The main purpose of the Performance Prediction Model Converter is to transform the logical Petri net into a Timed Petri net (i.e., Performance Prediction Model). An example of the transformed Timed Petri net model is shown in Figure 7. Compare the two logical and Timed Petri nets; Figure 6 is the logical Petri net and Figure 7 is the transformed Timed Petri net. Figure 7 additionally has a local declaration node, a time delay arc expression, a code segment, and a temporary declaration node. They are created automatically by PPMC.

The local declaration node defines the variables of delay values. Each variable name tells a logical connection link name between two tasks. It consists of a transition name (source) and a place name (sink). For example, a variable name AAW'T1_1'AAW'P3 means that the first instance of transition T1 at the AAW page sends messages to place P3 at the AAW page. Those assigned integer values to each variable are message identification numbers. So the total number of messages in a link represents the total number of newly created messages. For example, the transition T1 generated 3 tokens (i.e., 9,10,11) and places P3, P4, and P11 received 1 token each from transition T1.

The existence or non-existence of code segments tells whether that transition fired at least one time or never fired in the scenario (i.e., initial marking). The non-existence of the code segment at each transition means that the transition has not fired at all given the initial marking, or the transition has self-loop. This helps visualizing the logical behavior of the model.

The delay variables are automatically added at the tail of each arc expression with time expression. The "fnDELTA" on arc expression is the delay selection function. If a time delay expression appears on an arc, the arc's ending place must be declared as a timed color set.



```
(* Local Declaration Node *)

pageref  AAW'T1_1 = [9,10,11];
pageref  AAW'T1_1'AAW'P3 = [9];
pageref  AAW'T1_1'AAW'P4 = [10];
pageref  AAW'T1_1'AAW'P11 = [11];

pageref  AAW'T2_1 = [12,13,14];
pageref  AAW'T2_1'AAW'P8 = [12];
pageref  AAW'T2_1'AAW'P7 = [13];
pageref  AAW'T2_1'AAW'P12 = [14];

pageref  AAW'T4_1 = [15];
pageref  AAW'T4_1'AAW'P14 = [15];

pageref  AAW'T3_1 = [16,17,18];
pageref  AAW'T3_1'AAW'P9 = [16];
pageref  AAW'T3_1'AAW'P13 = [17];
pageref  AAW'T3_1'AAW'P5 = [18];
```

```
output (d3,d2,d1,b1);
action
let val
   d3 = getDelayNet AAW'T2_1'AAW'P12 and
   d2 = getDelayNet AAW'T2_1'AAW'P7 and
   d1 = getDelayNet AAW'T2_1'AAW'P8 and
   b1 = getDelayBase AAW'T2_1
in (d3,d2,d1,b1) end;
```

Figure 7. Intermediate Performance Prediction Model

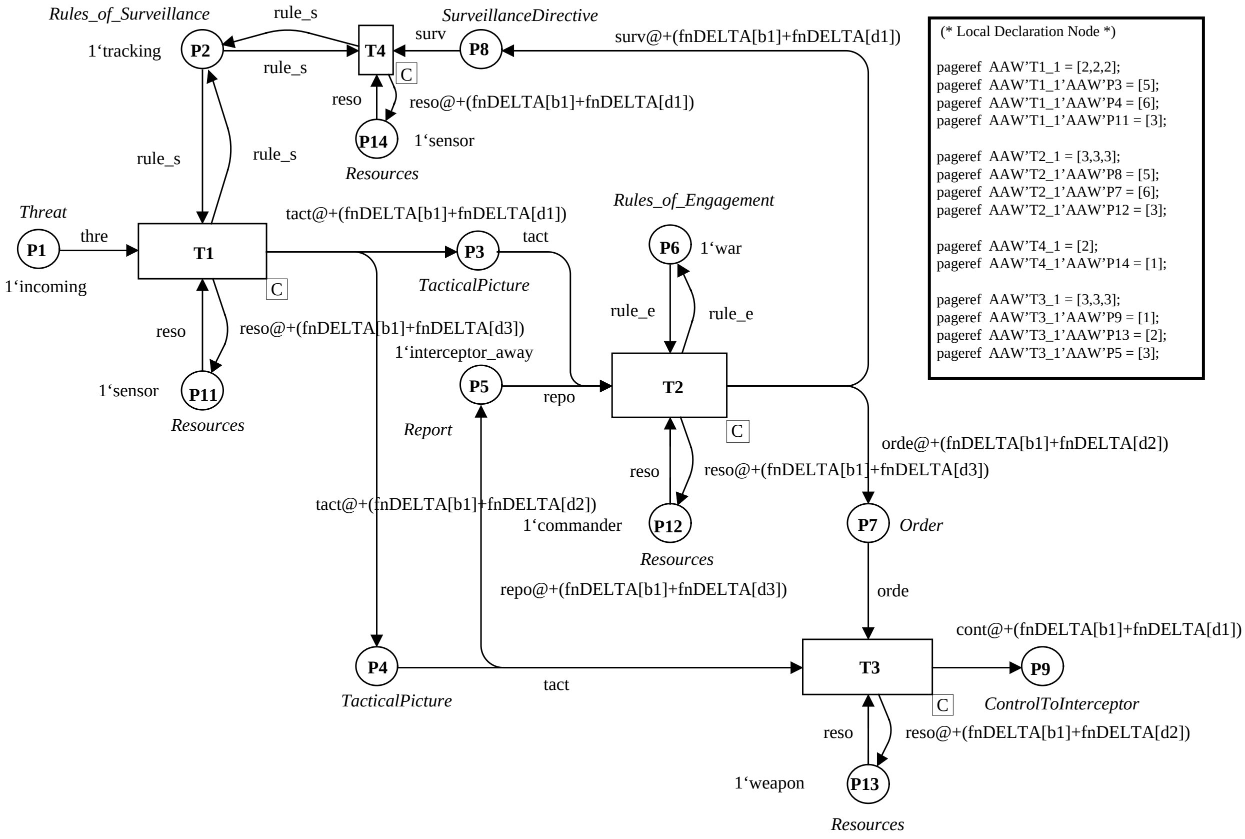
The delay variables of the local declaration node and the delay variables of arcs are mapped in the code segment. In the code segment, another two delay retrieving functions are used with names "getDelayBase" and "getDelayNet". These are used to retrieve the delay values from the local declaration node and send them to each arc. The functions associated with delays (i.e., getDelayBase, getDelayNet, and fnDELTA) are defined at the temporary declaration node (Table 3). These functions can be also defined as user's preference. By default, they were defined to get the minimum delay, maximum delay, zero delay, and random delays. The user can select this option anytime during simulation of the Petri net.

Once the net is converted into a performance prediction model, the net can be executed "as is". In this case, the net will consider the message ID as its own delay value. By doing this, the user can check if the system is modeled correctly before stepping forward. One example of this is as follows. Assume that the scenario changes so that the FDC changes the surveillance directive from the scanning mode to the tracking mode once the object is identified as a hostile flight. Transition T4 updates the new directive from the old surveillance directive to the new surveillance directive by changing the arc inscription. The resulting performance prediction model will create a time expression on the arc from T4 to P2. If we have the place P2 untimed, it will raise a syntax error. Even if the place represents a rule, the rule changes over time, involves updating time of the rule, and hence the place must be defined as a timed color set.

Table 3. Temporary Declaration Node

```
(* Temporary Declaration Node *)

globref  MinOrMaxBase = "Max" ;
globref  MinOrMaxNet = "Max" ;

color  timescale = int;
val unittime = (0: timescale) ;
val minmaxdef = (~1: timescale) ;
var d3,d2,d1,b1 :  timescale ;

fun Zero (xs: timescale list)  = unittime;
fun sum (nil) = unittime
   | sum ((x:timescale) ::xs) = x+sum(xs);
local
     fun min value  nil =  if value = minmaxdef then unittime else value
        | min value ((x:timescale)::xs) =
          let val new_value =  if value > x orelse value = minmaxdef then  x else value
               in min new_value xs end;
in fun Min (xs: timescale list) = min  minmaxdef xs
end;
local
     fun max value  nil =  if value = minmaxdef then unittime else value
       | max value ((x:timescale)::xs) =
          let val new_value =  if value < x orelse value = minmaxdef then  x else value
               in max new_value xs end;
in fun Max (xs: timescale list) = max  minmaxdef xs
end;

exception TimeNotDetermined ;
fun getDelayBase (xs: timescale list ref) =  if !xs=[] then raise TimeNotDetermined
  else  if  (!MinOrMaxBase) = "Zero" then Zero (!xs)
  else  if  (!MinOrMaxBase) = "Min" then Min (!xs)
  else  if  (!MinOrMaxBase) = "Max" then Max (!xs)
  else random (list_to_ms (!xs));

fun getDelayNet (xs: timescale list ref) =  if !xs=[] then raise TimeNotDetermined
  else  if  (!MinOrMaxNet) = "Zero" then Zero (!xs)
  else  if  (!MinOrMaxNet) = "Min" then Min (!xs)
  else  if  (!MinOrMaxNet) = "Max" then Max (!xs)
  else random (list_to_ms (!xs));

fun fnDELTA (xs: timescale list) = nth (xs,0);
```

Once the delay values are obtained from the physical model, the message IDs are replaced with their delay values. Figure 8 shows the final performance prediction model that was derived from the logical Petri net model (i.e., Figure 6) using the output delay data (shown in Table 4).



```
(* Local Declaration Node *)

pageref  AAW’T1_1 = [2,2,2];
pageref  AAW’T1_1’AAW’P3 = [5];
pageref  AAW’T1_1’AAW’P4 = [6];
pageref  AAW’T1_1’AAW’P11 = [3];

pageref  AAW’T2_1 = [3,3,3];
pageref  AAW’T2_1’AAW’P8 = [5];
pageref  AAW’T2_1’AAW’P7 = [6];
pageref  AAW’T2_1’AAW’P12 = [3];

pageref  AAW’T4_1 = [2];
pageref  AAW’T4_1’AAW’P14 = [1];

pageref  AAW’T3_1 = [3,3,3];
pageref  AAW’T3_1’AAW’P9 = [1];
pageref  AAW’T3_1’AAW’P13 = [2];
pageref  AAW’T3_1’AAW’P5 = [3];
```

Figure 8. Final Performance Prediction Model

Table 4. Output Delay Data from Communication Model produced by CNSS

```
model name: /demo/
Original number of message:  /18/
Valid number of message: /16/
variables: /("msg_id", ["pre_proc_delay", "action_time","post_proc_delay", "net_delay"])/

(1,[0.000000,0.000000,0.000000,0.000000])
(3,[0.000000,0.000000,0.000000,0.000000])
(5,[0.000000,0.000000,0.000000,0.000000])
(6,[0.000000,0.000000,0.000000,0.000000])
(7,[0.000000,0.000000,0.000000,0.000000])
(8,[0.000000,0.000000,0.000000,0.000000])
(9,[2.000000,0.000000,1.000000,4.000000])
(10,[2.000000,0.000000,2.000000,4.000000])
(11,[2.000000,0.000000,3.000000,0.000000])
(12,[3.000000,0.000000,1.000000,4.000000])
(13,[3.000000,0.000000,2.000000,4.000000])
(14,[3.000000,0.000000,3.000000,0.000000])
(15,[2.000000,0.000000,1.000000,0.000000])
(16,[3.000000,0.000000,1.000000,0.000000])
(17,[3.000000,0.000000,2.000000,0.000000])
(18,[3.000000,0.000000,3.000000,0.000000])
```

The delay values were imported into the Timed Petri net model as a list of values. CNSS can simulate the physical model repeatedly with the same order of message passing with various options and can generate multiple instances of delay values. However, the resulting delay values are imported into the functional model without processing them in a statistical form. The main purpose of using raw delay data is to generate the timed occurrence graph effectively. Another reason is to allow the user to be able to select those values as his analysis purpose. The converted performance prediction model has options how to use those values as specified in the temporary declaration node.

3.4 Network Topology Generator (NTG)

The purpose of the NTG is to generate network topology since the current NS does not provide a GUI for topology generation. The main use of this tool is, however, to allow the user to select the physical resources interactively depending on the design option or resource allocation option for each scenario. NTG provides the user with a GUI to enter the number of nodes and their attributes; the number of switching nodes of the Wide Area Network (WAN), the number of terminal nodes of each Local Area Network (LAN), and the number of leaf nodes at which the physical resources are connected to the WAN node. It automatically lays out the nodes in a two-dimensional grid and connects them in a default network topology of tactical communication networks. Figure 9 is an example of a network topology drawn by NTG.

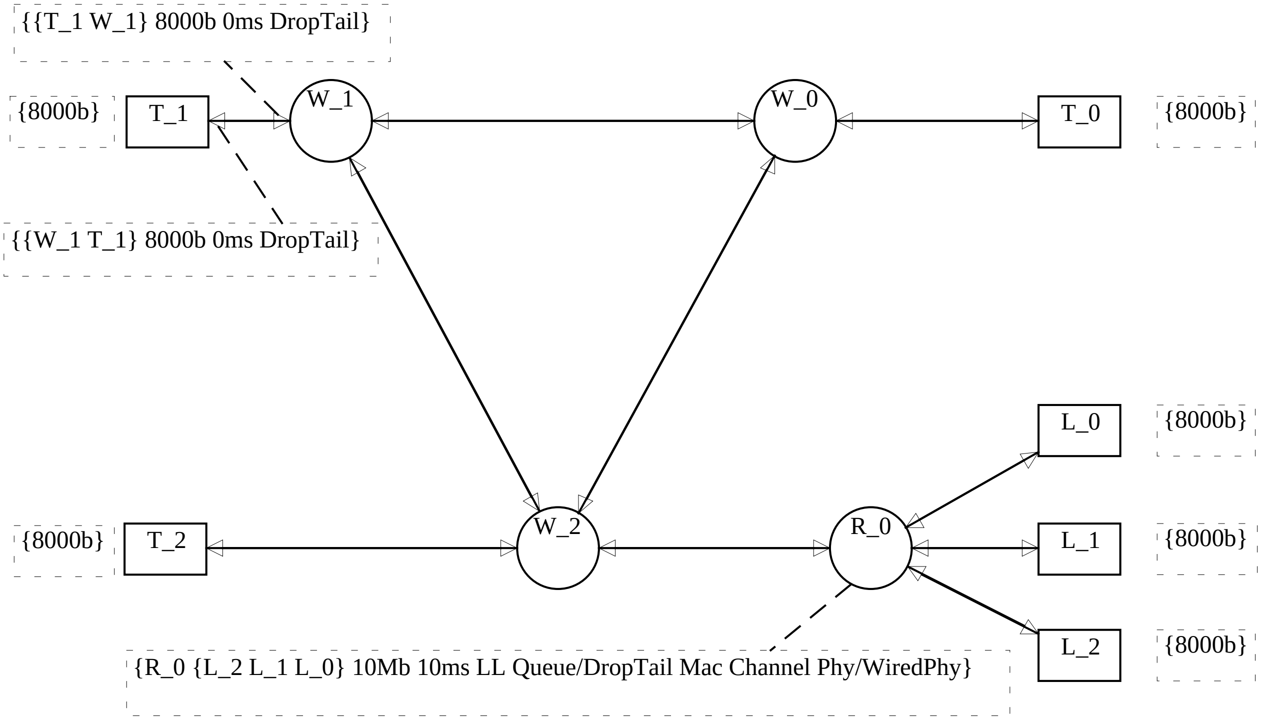

{{T_1 W_1} 8000b 0ms DropTail}
{8000b}
T_1
W_1
W_0
T_0
{8000b}
{{W_1 T_1} 8000b 0ms DropTail}
L_0
{8000b}
{8000b}
T_2
W_2
R_0
L_1
{8000b}
L_2
{8000b}
{R_0 {L_2 L_1 L_0} 10Mb 10ms LL Queue/DropTail Mac Channel Phy/WiredPhy}

Figure 9. An Example of Network Topology drawn by NTG

Figure 9 shows three WAN nodes (W_0, W_1, and W_2), one LAN router (R_0) with three terminals (L_0, L_1, and L_2), and another three terminal leaf nodes (T_0, T_1, and T_2). All links except LAN have link attributes with data transmission rate of “8000bps” and “0ms” electronic propagation delay. The example specification of the LAN is a wired, 10Mbps Ethernet protocol. Since the size of every message in the system was specified as 1000 bytes (8000bits) long in section 3, every delay of one hop transmission between two nodes will have 1 second delay. Also the figure shows that every leaf node (T_i or L_i) has bit processing power of “8000bps”. So it will take another 1 second for data processing at each node.

NTG defines and uses special node and region types that are different from “PetriNode” types.

Once nodes or links are constructed, the user can modify the topology of the network and its attributes. The user can add and delete a link using the drawing function ("connector") of Design/CPN. Once the topology is constructed, the tool performs syntax check and produces a network topology file in a format so that the CNSS can read it directly.

3.5 Communication Network Simulation Script (CNSS).

NS provides an object-oriented, event-driven simulation engine with various protocol modules. CNSS basically uses those facilities except for the node processing module. The developed methodology and tools support description of 3 types of delay: Node processing delay (pre- and post- processing delays) of the computing device, action delay of the task executor, and transmission delay of the network. Node processing time describes two types of data processing delays: pre-processing delay and post-processing delay. Pre-processing delay is a delay between the time at which all conditioned information arrives and the time at which the decision making task starts. Assume that sensor *A* reports a large image data for the feature of a detected object and sensor *B* reports a small signal data for the location of the object. The time to display the information to decision maker will vary. Post-processing delay is the delay between the time at which decision is made and the time at which the order is to start broadcasting. To handle those pre- and post- processing delays, CNSS creates another terminal node and attaches to the corresponding network node. The bit processing power of the computing devices is specified as another bandwidth of a special link between those two nodes. The node processing delay is now measured by the link delay. Assume that a transition supported by the corresponding computing device has 3 input places and 1 token at each place and 3 tokens are 1000, 2000, and 3000 bytes long each. If the bit processing power of the computing device is 1000bps, the pre-processing time will 48 seconds; 6000 *(byte)* * 8 *(bit/byte)* / 1000 *(bit /second)* = 48 seconds. If the same transition has 2 output places, produces 2 tokens at each place, and the token at the first place is 100 bytes long and that at the second place is 1000 bytes long, then the post-processing delay will be 17.6 seconds; {2 *(token)* *100 *(byte/token)* * *8 (bit/byte)* + 2 *(token)* * *1000 (byte/token)* * 8 *(bit/byte)* } / *1000 (bit/second)* = 17.6 seconds. By expressing distance and speed of a vehicle as *meter* and *meter per second* respectively, it can model a force maneuvering in a war-game model. Similarly, by expressing the data as workload, it can model a manufacturing system, business processing system, or logistics system.

NS produces a large amount of simulation data because it executes protocols at each layer in detail. For example, the resulting simulation data of the detailed model of Figure 8 was about 250Mbs long. CNSS creates a monitoring agent class and measures the network delays during simulation, and produces only that delay information. These monitored delays are also used for scheduling the next messages in compliance with message dependency relations produced by MG of section 3.2 in the format of Table 1.

4. RESULTS

Now let us specify all action time delays of task executors to be "0" delay. That is to select the minimum delay from {0s, 2s} for sensor execution time and the minimum delay from {0s,10s} for Missile's warming time. Resources "sensor", "commander" and "weapon" in Figure 6 are mapped to terminals T_0, T_1, and T_2 in Figure 9 respectively. Under those specifications and scenario in the previous section, Table 5 was produced from the Performance Prediction Model (Petri net). Table 6 was produced from the Physical Model (NS). The time stamps of messages in Table 5 and the arrival times of messages in Table 6 show how the system responded to external stimuli. For an incoming threat (message id 1) at time 0 second, a weapon fired at time 20 second (message id 16), the missile launcher repositioned at time 21 second (message id 17), and the after-action-report arrived at time 22 second (message id 18) at the FDC. Table 5 and Table 6 have the same information regarding the timing behavior.

Table 5. Message List produced from Figure 8

```
*** Numbered Message List of model : /home/ishin/funcbin/data/../../workshop/demoafter***
Msg_ID   Content      Time_Stamp   From_Transition   To_Place

1        incoming           0          INITIAL         AAW'P1 1
2        war                0          INITIAL         AAW'P6 1
3        weapon             0          INITIAL         AAW'P13 1
4        tracking           0          INITIAL         AAW'P2 1
5        sensor             0          INITIAL         AAW'P11 1
6        interceptor_away   0          INITIAL         AAW'P5 1
7        commander          0          INITIAL         AAW'P12 1
8        sensor             0          INITIAL         AAW'P14 1
9        new                7          AAW'T1 1        AAW'P3 1
10       new                8          AAW'T1 1        AAW'P4 1
11       sensor             5          AAW'T1 1        AAW'P11 1
12       default            15         AAW'T2 1        AAW'P8 1
13       intercept          16         AAW'T2 1        AAW'P7 1
14       commander          13         AAW'T2 1        AAW'P12 1
15       sensor             18         AAW'T4 1        AAW'P14 1
16       take_off           20         AAW'T3 1        AAW'P9 1
17       weapon             21         AAW'T3 1        AAW'P13 1
18       interceptor_away   22         AAW'T3 1        AAW'P5 1
```

Table 6. Delay Output from Ns

```
modelname: demo
Original number of message:  18
Valid number of message: 16
msg-id platform start-pre  start-p    send-m     recv-m  pre-proc-dly  action-t    post-proc-dly  net-dly

1   INITIAL   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
3   INITIAL   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
5   INITIAL   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
6   INITIAL   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
7   INITIAL   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
8   INITIAL   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000   0.000000
9   T_0       0.000000   2.000000   3.000000   7.000000   2.000000   0.000000   1.000000   4.000000
10  T_0       0.000000   2.000000   4.000000   8.000000   2.000000   0.000000   2.000000   4.000000
11  T_0       0.000000   2.000000   5.000000   5.000000   2.000000   0.000000   3.000000   0.000000
12  T_1       7.000000   10.000000  11.000000  15.000000  3.000000   0.000000   1.000000   4.000000
13  T_1       7.000000   10.000000  12.000000  16.000000  3.000000   0.000000   2.000000   4.000000
14  T_1       7.000000   10.000000  13.000000  13.000000  3.000000   0.000000   3.000000   0.000000
15  T_0       15.000000  17.000000  18.000000  18.000000  2.000000   0.000000   1.000000   0.000000
16  T_2       16.000000  19.000000  20.000000  20.000000  3.000000   0.000000   1.000000   0.000000
17  T_2       16.000000  19.000000  21.000000  21.000000  3.000000   0.000000   2.000000   0.000000
18  T_2       16.000000  19.000000  22.000000  22.000000  3.000000   0.000000   3.000000   0.000000
```

Figure 10a is the full occurrence graph of the functional architecture (logical Petri net, Figure 6). Figure 10b is the full occurrence graph of the performance prediction model (Timed Petri net, Figure 8). The reachable state of the system has been reduced from two paths to one path. Notice that Figure 10b shows the same value of time stamps as shown in Table 6.

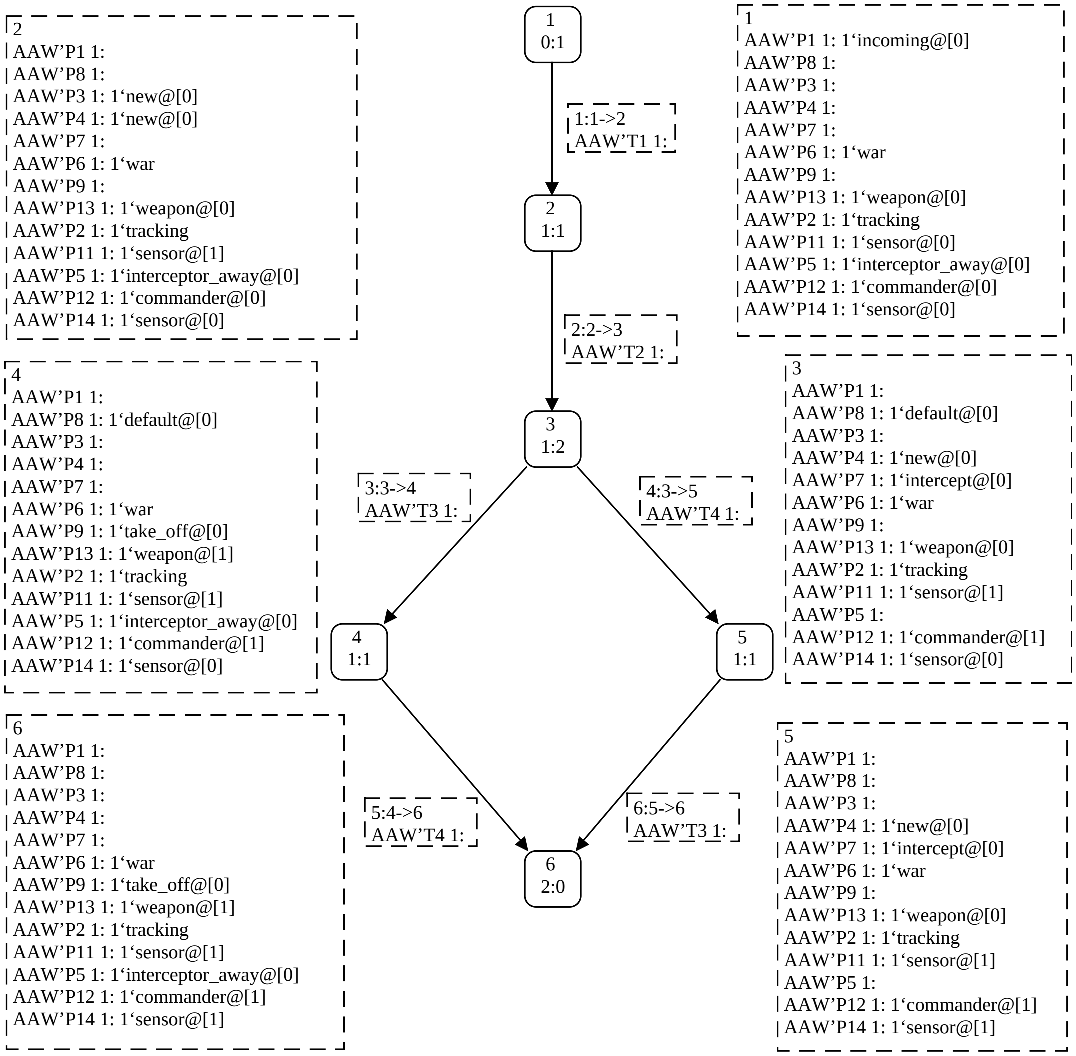


Figure 10a. A Full Occurrence Graph of Logical Model (Petri net in Figure 6)

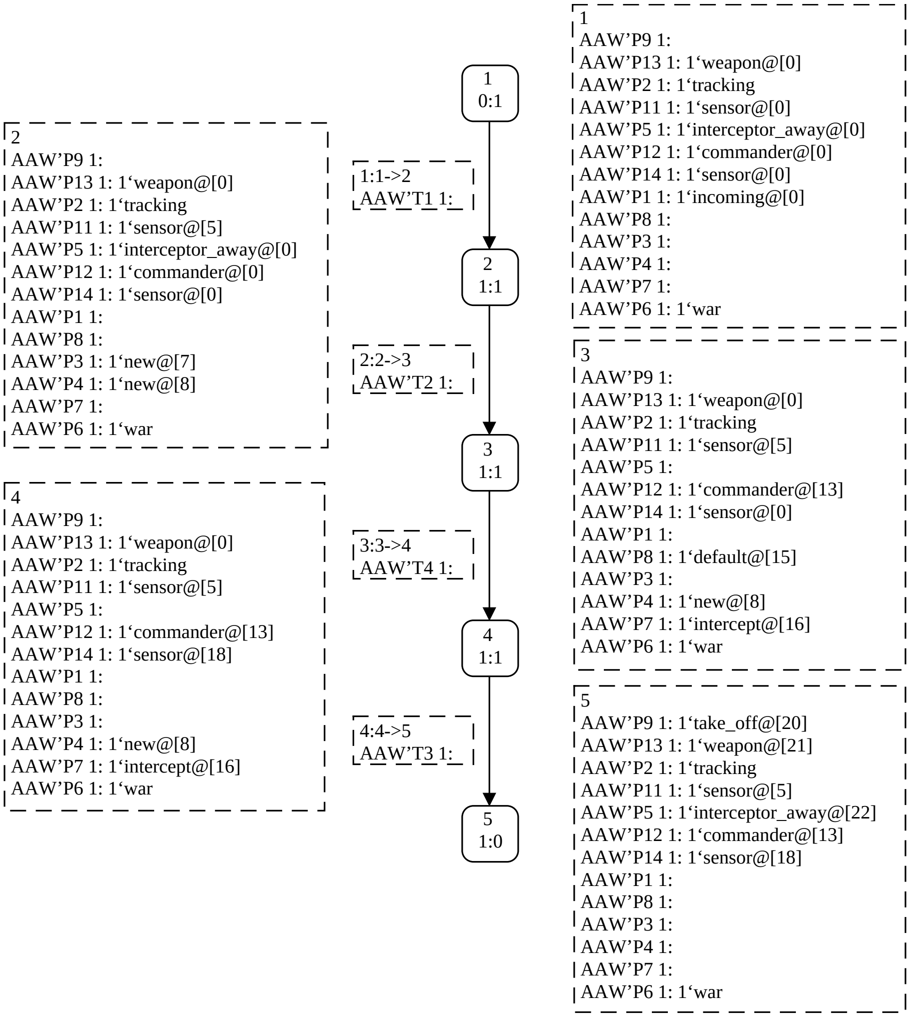


Figure 10b. A Full Occurrence Graph of Performance Prediction Model (Petri net in Figure 8)

5. CONCLUSION

The methodology developed provides an effective performance evaluation method for both qualitative (logical property) analysis and quantitative (timing property) analysis. The P2MG was implemented in the Design/CPN environment for automatic generation of performance prediction model. It is powered by the OGA of Design/CPN. The application of this methodology and the software tool needs only one simulation of each model, if the Petri net is conflict free. However, it is not limited to conflict free nets. It can be used for other Petri nets by repeated simulations. The current version of the software tool is limited to a system that uses a common physical architecture with multiple functional architectures or missions. The next step is to expand the tool to apply to heterogeneous physical architectures.

ACKNOWLEDGMENT

We appreciate the valuable comments of the anonymous reviewers.

REFERENCES

[1] Design/CPN, http://www.daimi.au.dk/designCPN/

[2] Network Simulator - ns (version 2), http://www-mash.cs.berkeley.edu/ns/ns.html

[3] Tsai, Jeffrey J.P. and S.J. Yang (1995), *Monitoring and Debugging of Distributed Real Time Systems*, Washington, D.C., IEEE Computer Society Press.

[4] Stankovic, J.A., K. Ramamritham and S. Cheng (1985), "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems," *IEEE Transactions on Computer*, Vol. 34, Dec., pp. 1130-1143.

[5] Gerber, Richard, Seongsoo Hong and Manas Saksena (1994). "Guaranteeing End-to-End Timing Constraints by Calibrating Intermediate Processes," *Proceedings of Real-Time Systems Symposium*, 7-9 Dec., pp. 192-203.

[6] Deng, Yi, Jiacun Wang and R. Sinha (1998). "Incremental Architectural Modeling and Verification of Real-Time Concurrent Systems," *Proceedings of Second International Conference on Formal Engineering Methods*, 9-11 Dec., pp. 26-34.

[7] Felder, M., D. Mandrioli and A. Morzenti (1994). "Proving Properties of Real-Time Systems Through Logical Specifications and Petri Net Models," *IEEE Transactions on Software Engineering*, Vol. 20, No. 2, Feb., pp. 127-141.

[8] Levis, Alexander H. (1992). "A colored Petri Net model of intelligent nodes", *Robotics and Flexible Manufacturing Systems*, J.C. Gentina and S.G. Tzafestas (Editors), Elsevier Science Publishers B.V. (North-Holland), pp. 369-379.

[9] Fahmy, Hany I. and C. Douligeris (1996). "NAMS: Network Automated Modeler and Simulator," *Proceedings of the 29th Annual Simulation Symposium*, SIMULATION '96, April, pp. 65-70.

[10] Lamport, L (1978). "Time, Clocks and Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, pp. 558-565.

[11] Jensen, Kurt (1997). *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Volume 2, Springer-Verlag.