ISSN 0105-8517

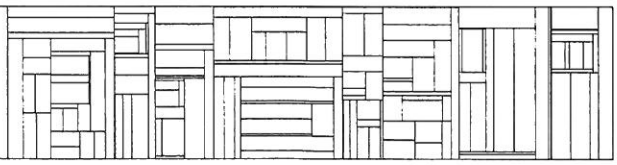
# Workshop on Practical Use of Coloured Petri Nets and Design/CPN

Aarhus, Denmark, 10-12 June 1998

Kurt Jensen
(Ed.)

DAIMI PB – 532
May 1998

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark



# Preface

This booklet contains the proceedings of the 1998 Workshop on Practical Use of Coloured Petri Nets and Design/CPN. The papers are also available in electronic form via the CPN Web pages at University of Aarhus:
http://www.daimi.aau.dk/CPnets/

Coloured Petri Nets and the Design/CPN tools are now used by more than 400 organisations in 38 countries all over the world (including 100 commercial companies). 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 workshop is organised by the CPN group at Department of Computer Science, University of Aarhus, Denmark. It takes place June 10-12, 1998 and it is preceded by a CPN tutorial, June 8-9.

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.aau.dk |
| H. Genrich | Germany | hartmann.genrich@gmd.de |
| N. Husberg | Finland | Nisse.Husberg@hut.fi |
| K. Jensen | Denmark chair | kjensen@daimi.aau.dk |
| D. Moldt | Germany | moldt@informatik.uni-hamburg.de |
| L. Petrucci | France | petrucci@iie.cnam.fr |
| D. Simpson | UK | Dan.Simpson@brighton.ac.uk |
| R. Valette | France | robert@laas.fr |
| R. Valk | Germany | valk@informatik.uni-hamburg.de |
| K. Voss | Germany | klaus.voss@gmd.de |

The programming committee has accepted 17 papers 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 tool extensions.

After an additional round of reviewing and revision, some of the best papers from the workshop will be published as a special section in the International Journal on Software Tools for Technology Transfer (STTT).

Kurt Jensen

# Table of Contents

# Applying Coloured Petri Nets and Design/CPN to an Air-to-Air Missile Simulator

*Steven Gordon and Jonathan Billington*

School of Physics and Electronic Systems Engineering,
University of South Australia, Adelaide, SA 5095, Australia
Email: [sgordon, jb]@spri.levels.unisa.edu.au

**Abstract**

In this paper the communication mechanisms of a missile engagement simulator are modelled and analysed. The simulator is developed as a testing platform for missile guidance and control algorithms. The simulation uses concurrency and remote execution concepts to enhance performance. Coloured Petri nets are a well suited formal approach for modelling and analysis of these concepts. Design/CPN is used to create and analyse the model of the simulation. A new requirement of the graphical user interface is identified for the simulation to operate successfully. The communication mechanisms are without deadlocks and are suitable for the simulator.

**Keywords:** Missile Simulator Design, Distributed Systems, Coloured Petri Nets, Formal Analysis

## 1 Introduction

Computer simulations of a missile engaging its target provide an environment for testing the guidance and control functions of the missile. The accuracy of the tests depends on the detail of the models used (eg. target, missile, etc.) and correct communication between the models. This paper addresses the problem of analysing the communication mechanisms for an Integrated Weapons Simulator (IWS) [CGW97a].

IWS is an Air-to-Air missile engagement simulator with a graphical user interface (GUI). Its development, from requirements specification through to testing and delivery, was part of a final year undergraduate project in Computer Systems Engineering at the University of South Australia. The three project members had joint supervision from staff of University of South Australia and the Defence Science and Technology Organisation (DSTO) Australia.

DSTO Australia, the research arm of Australia's Department of Defence, will use IWS to test various algorithms for the guidance and control of Air-to-Air missiles. To provide accurate simulations, the complexity of these algorithms may be large. Therefore, in designing IWS, two important features are desired so that the performance of the

system is adequate: concurrent execution and remote execution of separate components of the simulation. In providing these features it is necessary to verify that communication between the components is correct. Coloured Petri Nets (CP-nets) [Jen92] are well suited to modelling concurrency and hence are a suitable formal method for this problem. CP-nets have been successfully used to model and simulate concurrency in many applications [Jen97], and also for applications that use concepts of distributed execution [JM96].

In this paper CP-nets are used to model the communication mechanisms in IWS and to analyse them for deadlocks. An overview of the operation of IWS and its components is given. Then the CP-net models of IWS are described. An analysis of the entire simulation model is made, and finally the conclusions drawn from the analysis are presented.

# 2 The Missile Engagement Simulation

## 2.1 Structure and Function of IWS

IWS is developed to give DSTO Australia a testing platform for missile guidance and control algorithms. Hence, IWS is required to provide appropriate testing functionality and mechanisms for easily changing the algorithms. The first requirement is implemented by the GUI. The second requirement is met by structuring IWS into components that represent the relevant functions of a missile engaging a target. Figure 1 shows the physical system IWS models.

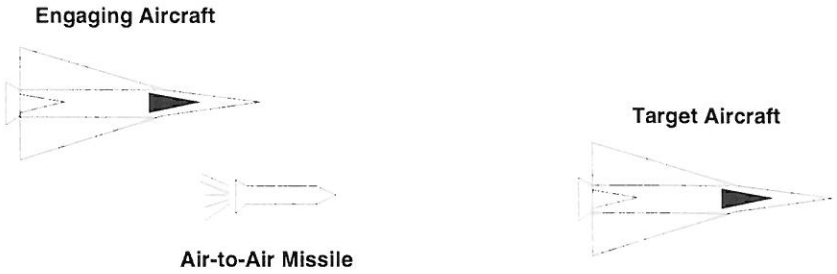



Figure 1: Physical System

When the engaging aircraft detects the target, it launches an Air-to-Air missile. After launching, the missile uses its own guidance system to track the target. IWS begins simulation from this stage - it does not simulate the launching aircraft or procedure.

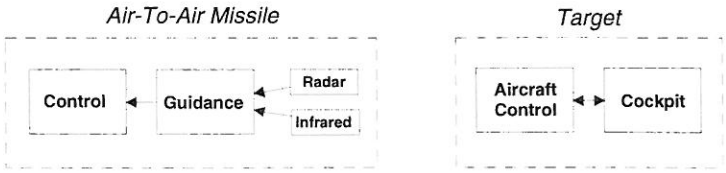



Figure 2: Missile and Target Physical Structure

The physical components of the missile and target that IWS is concerned with are shown in figure 2. The Air-to-Air missile has two detection mechanisms - a Radar (RF) and an Infrared (IR) sensor. These are physical devices on the missile that detect the location of a target. Both mechanisms are used to improve accuracy. For example, the

Radar may give inaccurate data if electronic counter measures are taken by the target. In this case, the Infrared data will be used. The data to be used is determined by the fusion mechanism in the Guidance component of the missile. The Guidance component then calculates the trajectory of the missile required to intercept the target. The Control algorithm controls the thrusters on the missile for it to maintain the calculated trajectory.

To simulate this physical system, IWS is divided into five components: Target, Radar, Infrared, Missile Control, and GUI. The components, and the interaction with the user, are shown in figure 3.

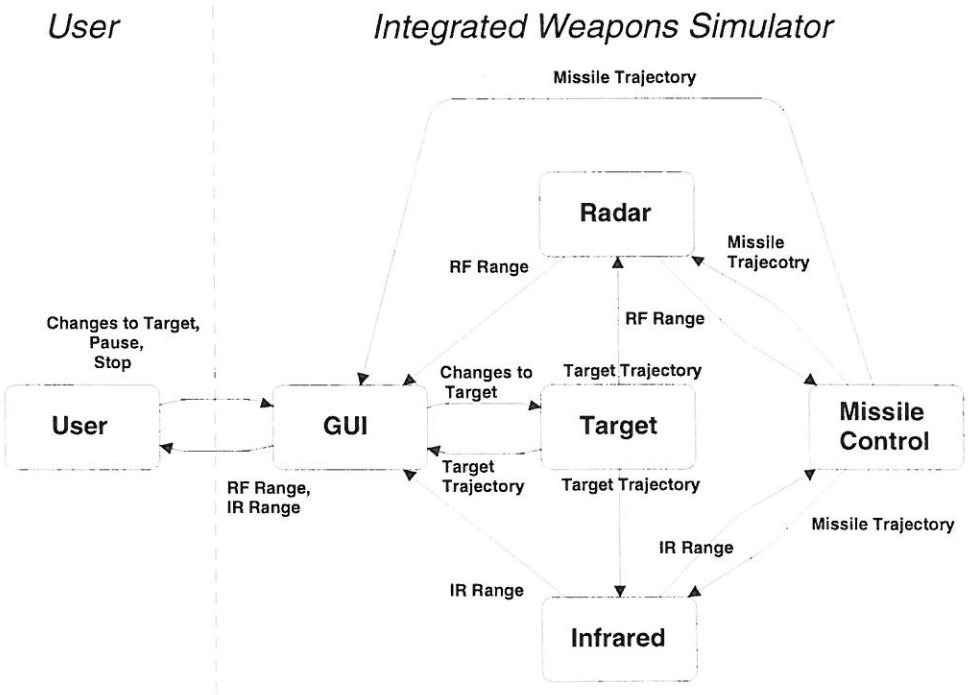



Figure 3: IWS Components

- **User.** The user of IWS plays three roles during the simulation:

  1. running the simulation, eg. issuing "pause" and "stop" commands,
  2. controlling the target by giving required changes to the target's trajectory, and
  3. viewing the target from the missile's point of view, ie. using the radar and infrared ranges.

  To do this, the user interacts with the GUI. (Future versions of IWS could separate the roles by introducing a new user who controls the target, while the first user runs the simulation from the missile's point of view [CGW97b].)

- **GUI.** The GUI component accepts inputs from, and presents the results of the simulation to, the user. Data from the Radar and Infrared systems are shown, and continuously updated, on a display for the missile. Inputs to Target are entered via a target controller window. This window, which allows changes in speed, elevation and azimuth, represents the throttle and control stick of the target aircraft.

The GUI also starts and stops the other four simulation components. It begins by sending the user inputs (changes to target trajectory) to the Target component. Once results from Target and Missile Control are received, the GUI compares the target and missile positions. If the two are the same, ie. the missile has hit the target, the simulation stops. Otherwise, the next user input is sent to Target and the simulation continues.

- **Target.** The Target component models a target for the simulation. The model must be able to perform realistic maneuvers so the guidance and control functions of the missile can be fully tested. A trajectory is calculated by Target based on inputs from the user (via the GUI). The inputs indicate a change in speed and direction. With these changes, the new position and velocity of the target are sent to the GUI, Radar and Infrared. Note that only a single target is modelled - IWS does not consider multiple targets [CGW97a].

- **Radar.** The Radar component simulates the physical radar sensor on the missile. It requires both the missile and target current positions and velocities. The Radar calculates the range (distance, elevation, and azimuth) of the target from the missile. To model the inaccuracies of the radar, small amounts of noise are added to these calculations. The range is sent to Missile Control.

- **Infrared.** The Infrared component simulates the physical infrared sensor on the missile. It requires the same inputs as the Radar and sends a calculated range to Missile Control. Again, the inaccuracies of the infrared sensor (due to, for example, cloud coverage) are modelled by adding noise to the calculations.

- **Missile Control.** The Missile Control component includes the guidance and control functions of the missile. The ranges from the Infrared and Radar systems are used to calculate the required missile trajectory. In IWS, the magnitude of the missile velocity (the speed) is constant. Once these guidance and control functions are performed, Missile Control then simulates the actual motion of the missile. The resulting position and velocity are sent to the GUI, Radar and Infrared. Once successfully tested, it is proposed that the guidance and control algorithms used in Missile Control can be included in the software onboard an Air-to-Air missile.

## 2.2 Communication Mechanisms

To achieve a likeness to the real-life situation, IWS executes each component concurrently. However, as the inputs of some components depend on the output of others, they cannot always be executing at the same time. For example, although in the real world the target and missile are independent, in IWS parts of the missile (Radar and Infrared) are synchronised with the target. There are several advantages of this mode of execution [SG94], the main one being computation speedup. This is particularly useful when the GUI must update its display (a CPU intensive task) and the other components can continue. The concurrent components are known as *threads* (eg. Target thread, Radar thread, etc.).

In addition to concurrency, the four simulation components (Target, Radar, Infrared, and Missile Control) will have the ability to be executed remotely. This will allow the resources for IWS (for the simulation) to be distributed across multiple computers [Tan92]. The section of each component executing remotely is called a *process*. The process will perform the calculations for models used in each simulation component.

Using both threads and processes in IWS introduces three communication mechanisms.

1. *Thread-to-thread* - Each of the five threads use shared memory to communicate, with access guarded by semaphores. This is a simple and common technique for multi-thread communication. The threads are required to run on the same computer.

2. *Thread-to-process* - The four simulation threads communicate with their corresponding processes using TCP/IP sockets. TCP/IP is the most common set of protocols used for communication over the Internet [Los95]. TCP/IP sockets are a standard feature of UNIX and as IWS will be developed for a UNIX environment [CGW97a], this method of interprocess communication allows IWS to run on almost any UNIX-based computer.

3. *Process-to-process* - Direct communication is used between the Radar and Missile Control processes and between the Infrared and Missile Control processes. The ranges are sent directly to Missile Control because the GUI is not required to control the missile components (Radar, Infrared and Missile Control). However, there is a need for the GUI to control the Target and the missile components separately, hence there is no direct communication between the Target and Radar or Infrared.

For successful operation of IWS, it is required that each of the above mechanisms are tested to operate correctly. This includes analysis to determine whether the flow of data is correct and that the terminal state (the state the system is expected to stop in) is always reached.

# 3 CP-nets

CP-nets [Jen92, Jen94] are a class of high level nets that extend the features of basic Petri nets. The two most basic and important differences are that CP-nets use: tokens which are arbitrarily complex data, and inscriptions on arcs and transitions. With the use of hierarchies, these features make CP-nets a powerful modelling tool for large applications.

The CP-nets in this paper were edited, simulated and analysed using Design/CPN [Sof93]. The Occurrence Graph (OG) tool [CJ95] of Design/CPN was used for analysis. This tool allows the creation of the full occurrence (or reachability) graph and also provides the capability to query the OG to determine properties of the system being modelled.

The choice of CP-nets as a formal method for modelling and analysis of IWS was based on: the existing experience with CP-nets, and their natural capability of modelling concurrent events.

The CP-nets were created by one member of the IWS Project group. Previous experience with CP-nets and Design/CPN consisted of: 9 x 2 hour lectures introducing Petri nets, High-level nets and example applications, and a 2 hour practical session introducing Design/CPN.

The project member also had experience with SDL [CCI89], but lack of available software tools for analysis made it an unsuitable method for this problem.

A major component of this work was the exploration of different approaches to modelling IWS and the levels of abstraction that were appropriate. This process was slowed by the lack of experience with some Design/CPN features (eg. hierarchies). Including this initial work and reporting on results, the modelling and analysis consisted of approximately 11 man weeks of work.

# 4 Missile Simulation Model

## 4.1 General Structure

The model of IWS consists of five pages of CP-nets and a global declaration node. This is indicated in figure 4. A top level model of IWS shows the interfaces between the four simulation components (figure 6).

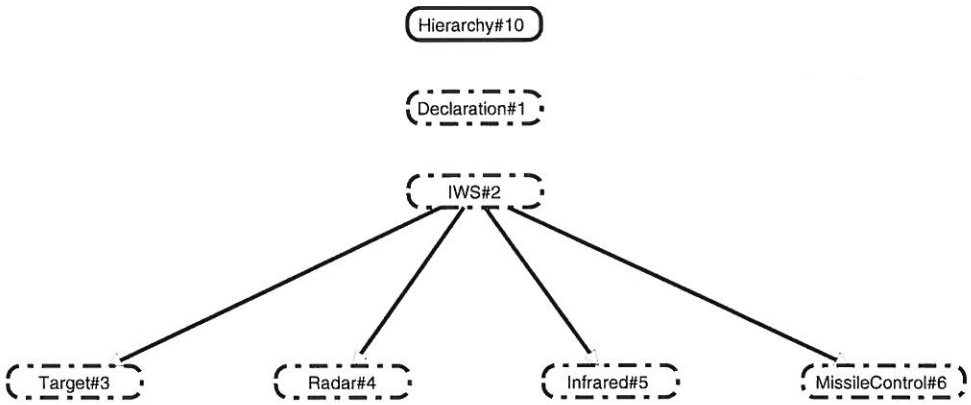



Figure 4: CP-net Hierarchy Page

Four transitions of the top level model are expanded into subpages to represent the simulation components ie. Target, Radar, Infrared, and Missile Control (figures 7 to 10).

## 4.2 Global Declarations

The global declaration page is shown in figure 5.

```
color Status = with Ready | Wait;
color Input = with Target_Delta | Pause | Stop;
color Motion = with Target_Trajectory | Missile_Trajectory;
color Range = with RF_Range | IR_Range;

var cmd:Input;
```

Figure 5: Global Declarations

The colour `Status` indicates the state of a thread when not processing data. For example, when the Radar thread (figure 8) is ready to receive new data from the target and missile, a `Ready` token is sent from place `Thread Ready`.

The colour `Input` represents a single input from the user. `Target_Delta` represents the change to target trajectory the user wants.

The positions and velocities of the target and missile are modelled as a trajectory. These are represented by the colour `Motion`.

The colour `Range` represents the range of the target relative to the missile. The radar (RF) and infrared (IR) ranges are different.

The variable `cmd` is used to represent the next user input. It can be either `Target_Delta`, `Pause` or `Stop`.

## 4.3 IWS

### 4.3.1 Top-level CP-net

The top level CP-net (figure 6) shows the functionality and flow of data in IWS. The user is not represented explicitly, instead the place `GUI inputs` stores the current command from the user (either a `Pause`, `Stop` or `Target_Delta`) and it is processed by one of the three output transitions. If the user issues a pause command the `Pause` transition fires and a new user command is generated. If stop is issued by the user then transition `Stop` will fire leaving no more user commands. If `Target_Delta` is the user command then it is sent to Target and the simulation continues.

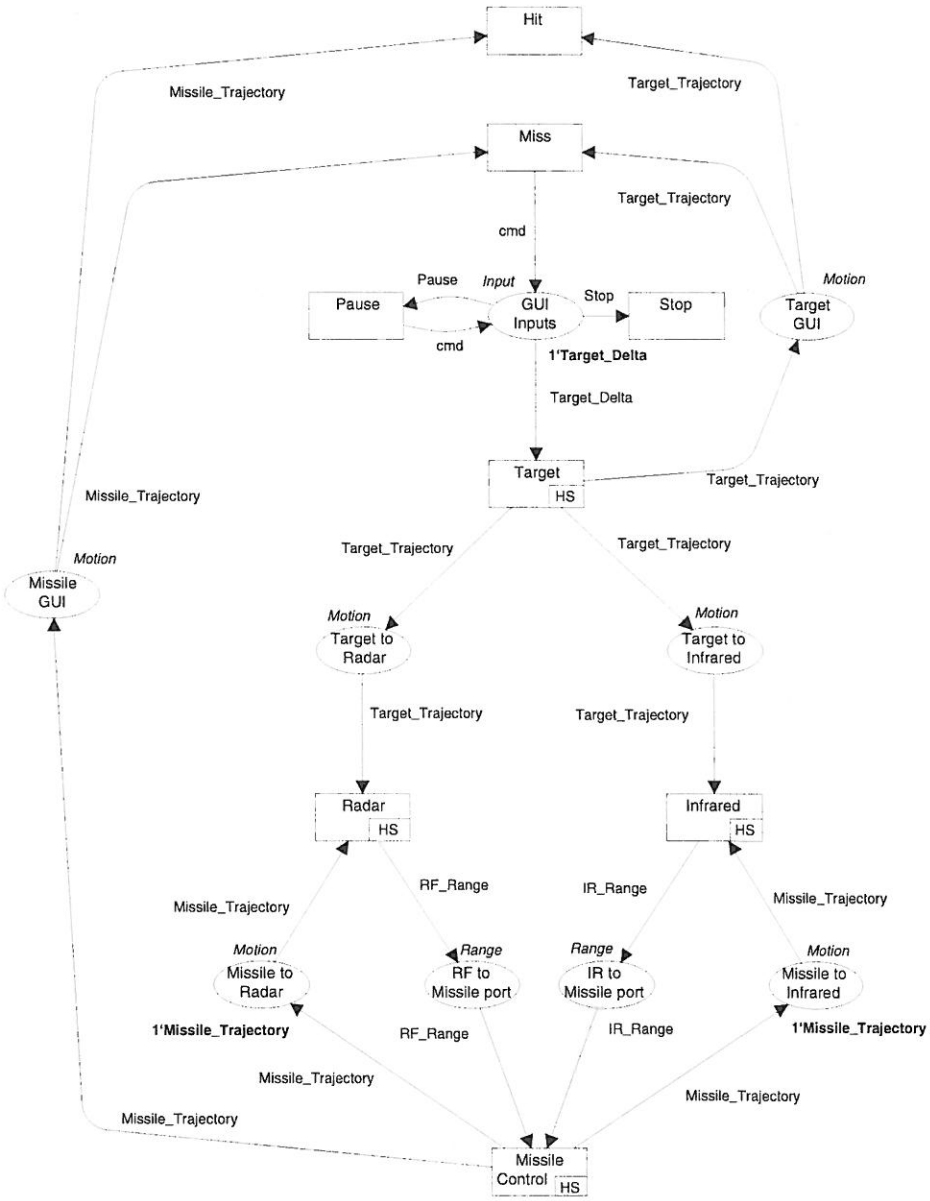



Figure 6: Top-level CP-net

When the new target and missile trajectories have been calculated two conditions can occur: the positions of each are the same (within a given tolerance) meaning the missile had hit the target and the simulation should stop; or the positions are not the same and

the simulation should continue. This is modelled using the two transitions `Hit` and `Miss`. If `Miss` occurs a new user command is generated and the simulation can continue. If `Hit` occurs, no more transitions will be enabled causing the simulation to stop.

### 4.3.2 Structure of CP-net Subpages

The four simulation component pages (figures 7 to 10) are structured into columns to visually separate the shared data, threads, processes, and communication mediums.

The first column represents the shared memory between each of the four threads. The data (or tokens) in this area can be accessed by any of the threads. When implemented, access to the memory will be guarded by the use of semaphores.

The second and fourth columns in the CP-net pages represent the procedures carried out by the threads and processes, respectively.

The third and fifth columns of the models represent the TCP/IP socket connections. The third column is for connections between the thread and its corresponding process, whereas the fifth column represents a connection between two processes. Although TCP/IP connections incorporate queueing mechanisms, they are modelled as simple buffers in the CP-nets. This is because the control of data flow does not allow overtaking of data, hence there are no need for queues.

For all the models, it is assumed that the socket connection has been setup and initial data for the processes has been received.

The functions performed by the simulation components in IWS are described in the following sections.

### 4.3.3 Target

The Target CP-net is shown in figure 7. The Target process begins in the "ready" state. This is indicated by an initial marking of `Target_Trajectory` for the place `Process Ready`. The first user command is also an initial marking for the place `GUI Inputs`.

When a user target change is received from the GUI, it is sent to the Target process. The transition `Calculate Target` represents the calculations made using the user target change and old target trajectory (from `Process Ready`) to produce a new target trajectory. This result is sent back to the Target thread and then to the GUI, Radar and Infrared.

### 4.3.4 Radar

The Radar thread begins in the "ready" state (figure 8). There is also an initial marking `Missile_Trajectory` in the `Missile to Radar` place. This marking represents the initial missile position and velocity so Radar (and Infrared) can calculate an initial range to be used by Missile Control.

When the target trajectory arrives at place `Target to Radar` (and the missile trajectory is in `Missile to Radar`), transition `Get Target data` is enabled. Upon firing, the target and missile trajectories are transferred together to the Radar process and with them a new range can be calculated. The range is sent back to Missile Control via the Radar-Missile socket, and also to the GUI via the Radar thread.

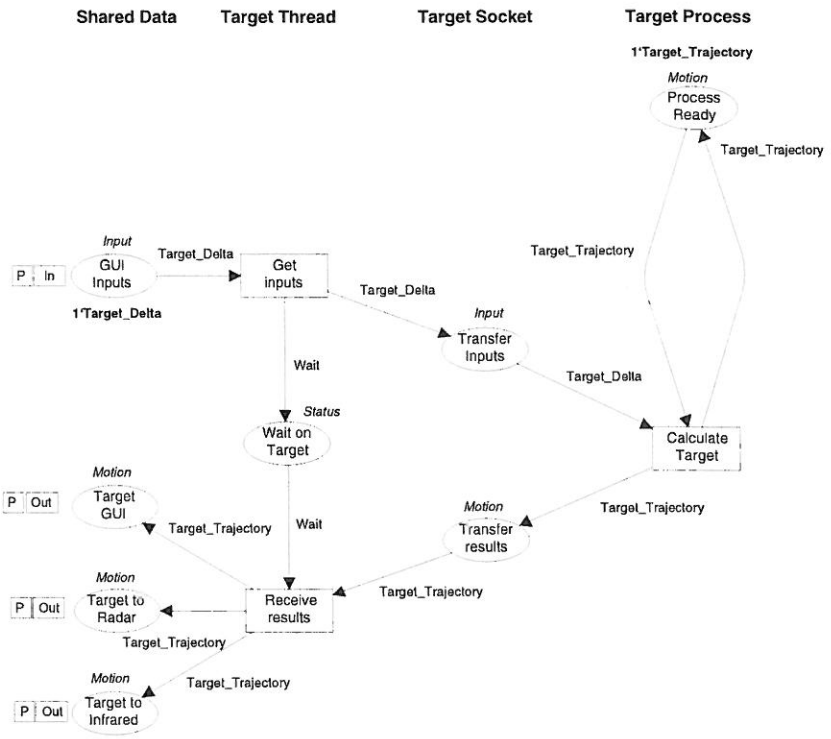



Figure 7: Target CP-net

### 4.3.5 Infrared

The CP-net model of the Infrared component (figure 9) operates in the same manner as the Radar model.

### 4.3.6 Missile Control

The Missile Control is straightforward (see figure 10). An initial missile trajectory is stored in `Process Ready` and when both the Radar and Infrared ranges arrive the new missile trajectory is calculated by the Missile Control process. The results are sent to the Missile Control thread which stores them as shared data to be accessed by the Radar and Infrared.

# 5 Analysis

The missile engagement simulation model was analysed using Design/CPN. Using Design/CPN interactive graphical simulations and automatic simulation modes are available. Simulating was useful when developing the models, and to view the flow of data in IWS. However, it was not suitable for testing all possible states of the models. The formal analysis of the CP-net model involved the OG tool of Design/CPN. The OG tool calculates the occurrence graph for the CP-net model. Rather than visually inspecting all nodes, queries were made of the occurrence graph to examine its properties. Table 1 shows the results of the occurrence graph generation.

The OG was cyclic - each simulation iteration was represented by one cycle of the occurrence graph. Figure 11(a) shows the initial marking (node 1) and its immediate successor and predecessor nodes. Note that the place and transition names used are

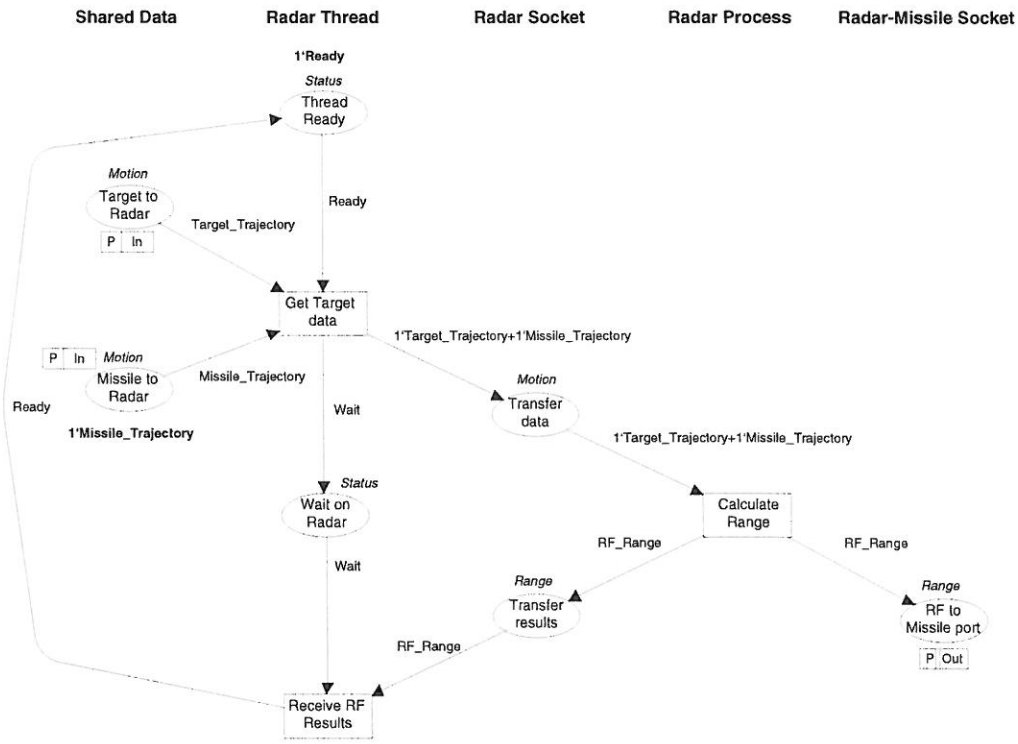



Figure 8: Radar CP-net

| *Nodes* | *Arcs* | *Time* | *TerminalStates* | *Undesired* |
|---|---|---|---|---|
| 57 | 124 | 1 s | 1 | 0 |

Table 1: OG Results

extensions of the labels used in the models. Also, places with empty markings are not shown.

There was one terminal state and no deadlocks (undesired terminal states) for the model. Closer examination of the terminal state and the arcs leading to it confirmed that the simulation stopped by either a "hit" or by the user. Figure 11(b) shows the terminal state (node 43) of the OG. A selection of nodes leading to this node are also shown. The arc from node 31 indicates `Hit` has occurred causing the simulation to stop. The arc from node 41 indicates the user has input a `Stop` command again causing the simulation to stop. The other two input arcs of node 43 show the transitions that can occur after `Hit` or `Stop` occurs. These two transitions are `Receive RF Results` and `Receive IR Results`. When the Radar or Infrared threads receive results from their corresponding processes (ie. these two transitions fire), the data is made available to the GUI so the display can be updated. However, when `Hit` or `Stop` occur the display is frozen so the user can view and analyse the results. Therefore, if `Hit` or `Stop` occur before either `Receive RF Results` or `Receive IR Results` then the latest results will not be displayed. To be sure this does not occur, the GUI must force an update of the display when stopped before allowing the user to perform analysis on the results.

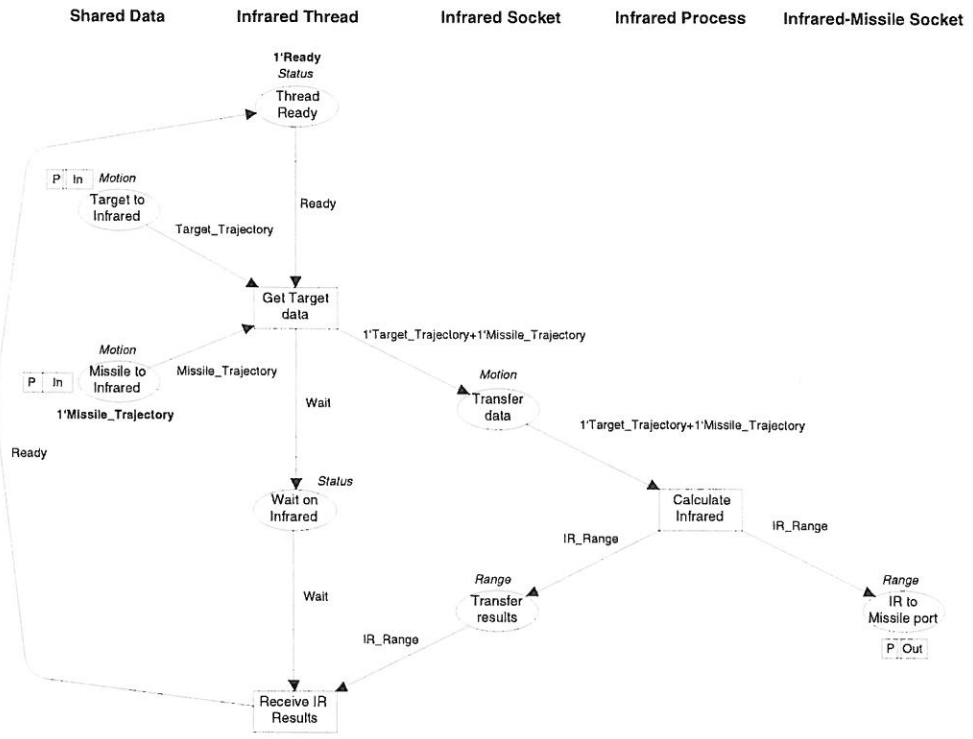



Figure 9: Infrared CP-net

# 6 Conclusions

Coloured Petri nets have been used to test the communication mechanisms for a distributed Air-to-Air missile engagement simulator.

The first step in achieving the goals was to create a top-level CP-net for IWS. This showed the data flow and basic functionality of the system. Using Design/CPN's hierarchy features, the relevant transitions were then expanded into subpages to model the interaction between threads and processes and the calculations performed by IWS. This allowed the concurrent execution of the IWS components to be analysed. This top-down approach to modelling IWS was advantageous because initially it was unclear how much detail was needed. Further functionality could be added when it is required.

Once the models were completed they were analysed using the OG tool of Design/CPN. This allowed occurrence graphs to be calculated and properties of it examined using queries. The analysis showed that the models behaved as required. There were no deadlocks and the individual threads and processes executed correctly. However, a new requirement of the GUI was identified, ie. a display update should be forced when the simulation is stopped.

From the analysis of the CP-nets, under the requirements from DSTO Australia, the communication mechanisms modelled are suitable for use in IWS. However, to take full advantage of the performance enhancement from concurrency and remote execution, it is necessary that the system performance is not restricted in other areas. In particular, a preliminary investigation into the effects of failures in the communication between processes and the methods for coping with such failures has been made. The models developed in this work served as a basis for this investigation.

There are also possibilities for improvements and further modelling and analysis to

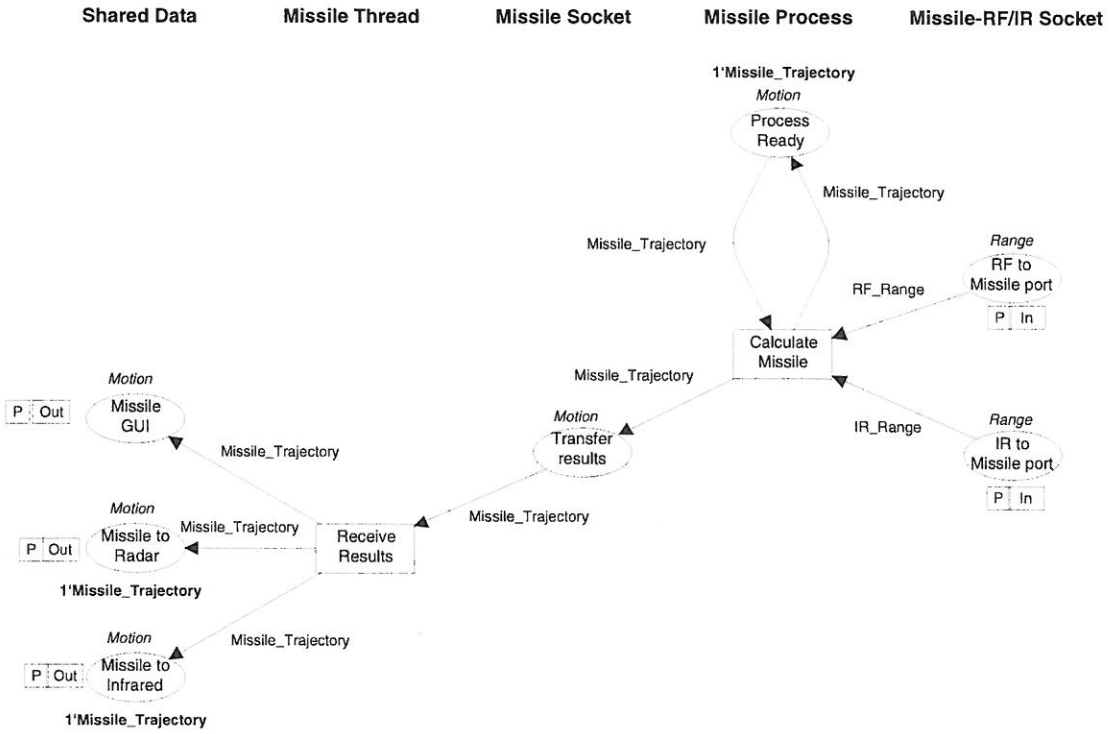



Figure 10: Missile Control CP-net

test other features of the IWS simulation.

- A requirements-level CP-net which specifies: interactions between the user of IWS and the simulation, and interactions between components within the simulation. A comparison can then be made between the requirements- and design-level CP-nets.
- Detail modelling of the GUI (at the design-level) and the interactions with the simulation. This would introduce more levels of concurrency (eg. the GUI updating it's display while the simulation performs the next calculation) and require more detailed analysis.
- Further modelling and analysis if multiple targets are introduced into IWS [CGW97a].

# 7 Acknowledgements

We would like to thank Hatem Hmam and Len Sciacca from DSTO for the technical background on Air-to-Air missiles and IWS, and Chris Steketee for his comments on TCP/IP. Also, the financial support of the University of South Australia's Institute for Telecommunications Research is appreciated.

# References

[CCI89] CCITT. Recommendation Z.100 Functional Specification and Description Language. In *Blue Book*, Geneva, Switzerland, 1989. ITU.

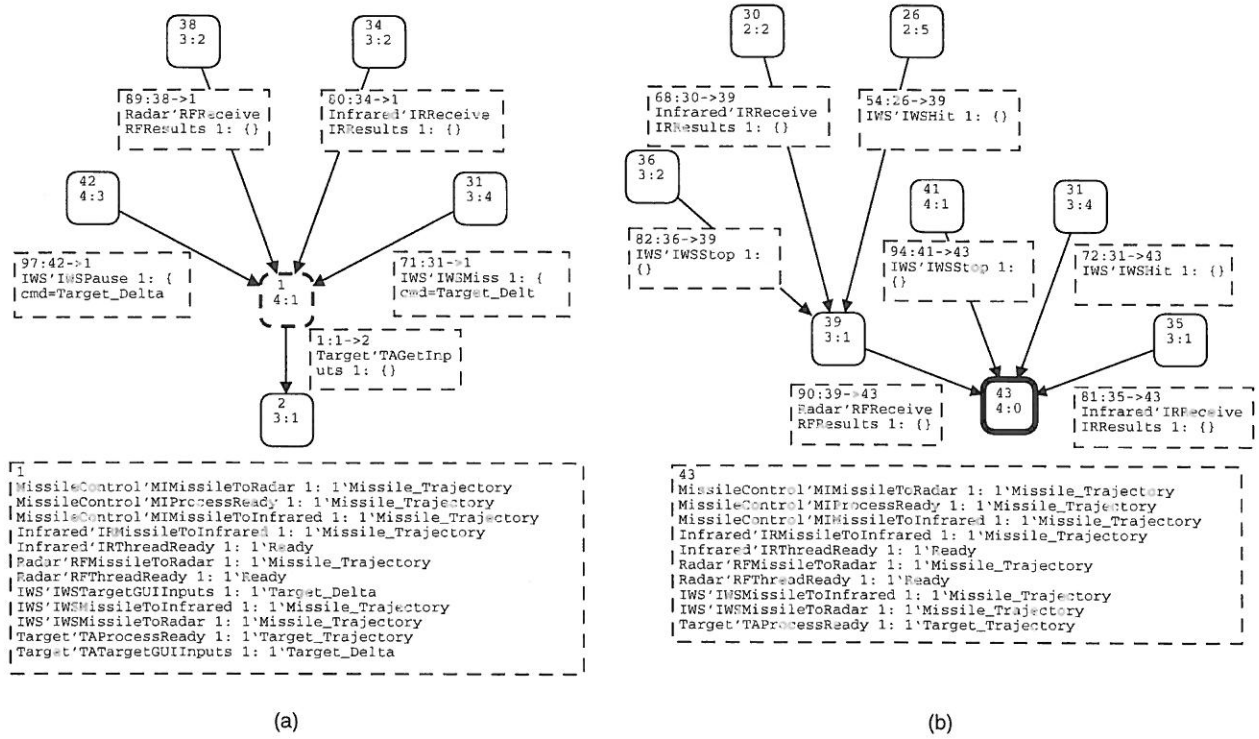



Figure 11: (a) Initial and (b) Terminal States


[CGW97a] B. Collas, S. Gordon, and H. Widjaja. *IWS Requirements Specification.* Number 1.2. University of South Australia, Adelaide, Australia, 1997.

[CGW97b] B. Collas, S. Gordon, and H. Widjaja. *IWS System Manual.* Number 1.0. University of South Australia, Adelaide, Australia, 1997.

[CJ95] S. Christensen and K. Jensen. *Design/CPN Occurrence Graph Tool - User's Manual.* Computer Science Department, Aarhus University, Aarhus, Denmark, 1995.

[Jen92] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use.* Springer-Verlag, 2nd edition, 1992.

[Jen94] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Volume 2.* Springer-Verlag, 1994.

[Jen97] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use, Volume 3.* Springer-Verlag, 1997.

[JM96] J.B. Jorgensen and K.H. Mortensen. Modelling and Analysis of Distributed Program Execution in BETA using Coloured Petri Nets. In *Application and Theory of Petri Nets*, number 1091 in Lecture Notes in Computer Science, pages 249–268. Springer-Verlag, 1996.

[Los95] P. Loshin. *TCP/IP for Everyone.* Academic Press, Chestnut Hill, MA, USA, 1995.

[SG94] A. Silberschatz and P. Galvin. *Operating System Concepts.* Addison-Wesley Publishing Company, Reading, MA, USA, 1994.

[Sof93] Meta Software. *Design/CPN Reference Manual for X-Windows.* Meta Software Corporation, Cambridge, MA, USA, 1993.

[Tan92] A. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1992.

# Modelling of Superscalar Processor Architectures with Design/CPN

Frank Burns, Albert Koelmans and Alexandre Yakovlev
Department of Computing Science
University of Newcastle upon Tyne, NE1 7RU, UK

**Abstract**

We describe aspects of modelling a generic superscalar processor architecture using Coloured Petri nets, for the purpose of analysis of its real-time properties, such as Worst Case Execution Time for a block of instructions. The model can be simulated within the Design/CPN environment. The results of the simulation are displayed using a custom graphics tool written in Tcl/Tk.

## 1 Introduction

Design of real-time systems, which must respond to external events within strict time bounds, involves a mapping between the logical and physical levels of a system specification. This mapping determines the quality of the timing information available at each level of abstraction. A crucial issue, which introduces a significant element of temporal uncertainty, is the way in which the system design is tied together with the use of specific hardware components. Even though some of these components can be standard, off-the-shelf ones, their timing characteristics can be very imprecise for the purpose of acquiring reliable, yet not too pessimistic data to be used at higher levels (e.g., worst execution times of program code to be used for task scheduling). Moreover, some hardware is designed in a special way (e.g., interface circuits), which is increasingly the case for embedded applications. In such situations, realistic timing information about the hardware becomes available only during software development, which creates an additional adequacy problem.

This problem arises increasingly due to the following three major factors:

- New processor architectures are becoming ever more complex. They include many stages of pipelining, out-of-order execution, parallel execution of several instructions, and multi-level caching mechanisms;
- System architectures often involve non-standard components whose temporal parameters are not known in advance;
- Future systems will tend to become more asynchronous. Even if their core elements are clocked, the overall system will either be multi-clocked or almost entirely self-timed.

We are developing a methodology and an associated set of software tools for the modelling and analysis of timing specifications of hardware platforms, in particular asynchronous RISC processors, based on the use of Coloured Petri Nets (CPNs) [3]. Previous work in this area (e.g. [6]) has mainly focused on the use of P/T nets. The ability to model asynchronous interaction inherent in Petri Nets would enable us to take into account some fine behavioural issues like the effect of potential 'request–acknowledgement' handshakes in the interfaces between units, and dynamic allocation of instructions on different units. In this paper we present a detailed model of a generic superscalar RISC processor developed using the Design/CPN tool.

## 2 Why ordinary Petri nets are insufficient

Petri nets have traditionally been used for modelling and analysis of digital systems. Processor architectures are no exception [2]. In addition to classical "qualitative" properties for verification, like deadlocks and boundedness, the real-time aspect requires ways of obtaining more detailed, "quantitative", analysis, such as worst-case execution time for a block of instructions. The former type of modelling need only capture fairly general properties (possibilities) of the system behaviour (e.g., the processor execution pipe cannot overflow with instructions). The latter is certainly more specific with regards to the actual paths taken by the modelled system under a given set of data. Hence the "quantitative" analysis often lends itself to simulation, performed in addition to the exploration of possibilities.

Use of ordinary Place-Transition nets, which can describe the control flow quite comfortably and provide efficient ways of "qualitative" verification, cannot themselves (without additional annotation) represent the effect of data, associated with the model states, on the execution of actions in the

processor. Note that in modelling processor architectures, the role of data path is played by instruction flow. Each instruction is a complex (again, depending on the level of detail we want to represent) data object. This object consists of attributes reflecting not only its type and operands, but also important (in the superscalar case) information about dependencies, targets, branching etc. Since all these aspects affect the temporal profile of the instruction in the overall instruction flow, the model should be able to represent them as accurately as possible. Coloured Petri nets appear to be capable of providing the processor model with appropriate mechanisms for data typing. These mechanisms can concretise both the notion of a state (token marking) and the data-dependent conditions of action execution.

Before we proceed to a detailed model of a processor, let us consider a simplified control flow model of an asynchronous processor, described with a Place-Transition net as shown in Figure 1. We believe that this figure is self-explanatory. More details about the use of Place-Transition nets in the design of a processor can be found in [4].

At the highest abstraction level, the behaviour of a processor consists of two actions, Instruction Fetching (IF) and Instruction Execution (IE), which alternate and are therefore performed sequentially.

These actions can be refined into subactions according to our knowledge about the processor architecture. Thus, the IF action can be seen as a process, i.e. a Petri net fragment, consisting of the following subactions: incrementing a Program Counter (PC), loading a Memory Address Register with the new address for memory reading (MAR_r), and reading the new instruction word from Memory (Mem). The IE action can be refined into a process (another Petri net fragment) involving other subactions: loading an Instruction Register (IR), decoding, activating and executing the fetched instruction for two possible instruction formats, a one word instruction (1WdInst and 1WdEx) and a two word instruction (2WdInst and 2WdEx). The part of the process concerned with two word instruction execution requires two memory cycles. As can be observed from the analysis of this Petri net, the initial sequential operation between IF and IE has been refined into a model which allows concurrency between actions with smaller granularity. For example, the PC action can be executed concurrently with instruction reading, decoding and execution. Another paradigm appearing at this level is that of choice between two types of instruction execution. The refined model can be subjected to verification (e.g. for absence of deadlocks or undesirable conflicts between actions) and/or performance analysis (e.g., estimation of the degree of concurrency between transitions, evaluating crit-

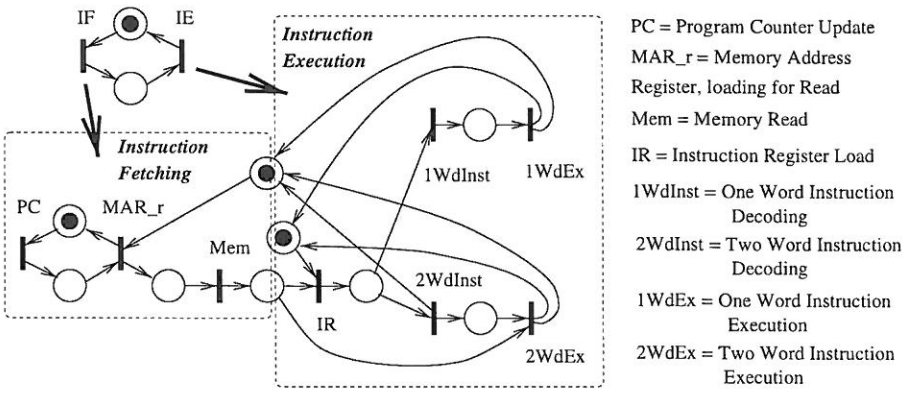



Figure 1: Place-Transition net model of an asynchronous processor.

ical paths, simulation). The process of refining the design can be continued until the designer realises that the abstract behavioural model satisfies the desired functional and quantitative requirements. The result of this design stage is a specification of the control flow in such a form that its actions, i.e. transitions in the labelled Petri net model, can be easily mapped onto the primitive operations of the datapath units. This part of the design process is described in detail in [4].

# 3 Basics of processor modelling

We model instruction types by first defining a set of predefined identifiers using an enumerated colorset as follows:
**Color Instr = with INT | FPADD | MUL | DIV | BRA | NOOP;**
It is then possible to create a record colorset using appropriate fields to model an instruction completely:

```
color Value = record
                no: Line *
                instr: Instr *
                d: Dep *
                d': Dep *
                t: Target timed;
```

assuming that the instruction has dependencies **d, d'** and a destination **t**, which links those instructions sharing the same target register. It must be

noted that the model we are trying to develop here is primarily of timing not hardware.

To model processor timing we need to define updates to Program Counter (PC) values at various points in our model. In Design/CPN variables, guards and arc inscriptions can be used in order to do this. CPN variables can be defined for any predefined type or colour. For example, we can define a variable fetch of type **Value** to represent a fetched instruction:

**var fetch : Value;**

Once variables are defined it is possible to model the occurrence of updates to processor values by referencing the values of tokens arriving at the input places of transitions. Guards provide predicates to check the values of tokens at input places to determine that the correct values are present for enablement. For example, at the **FETCH** stage we can determine the next instruction selection by defining a fetch transition guard which checks the instruction number against the pc count: [**#no fetch = pc**] Here the selector **#no** checks the number of the instruction fetched against the value of the program counter.

An arc inscription can then be defined to update the PC value to the next instruction number ready for the next selection. This is modelled in Design/CPN by using the following multiset arc expression which adds one to the program counter variable.

**1'(pc+1)**

The main graphical part of the CPN model of a superscalar processor is shown in Figure 2. In the following sections we discuss individual aspects of this model.

# 4 Pipelining and In-Order Issues

The basic processor pipeline can be modelled in Design/CPN by using a set of transitions for each pipestage **FETCH, DECODE, EXECUTE** etc. The **EXECUTE** stage is further subdivided into transitions to represent execution units **INT, FPADD, MUL, DIV.** Individual execution units are further pipelined by using a hierarchy of pages over transitions. Places are used between transitions for storage, queues etc.

Instruction flow through functional units needs to be controlled and we need to cope with hazards in processors when they arise. In order to reference processor values at various pipestages in Design/CPN we define instruction variables for the inputs and outputs leading to and from each pipestage

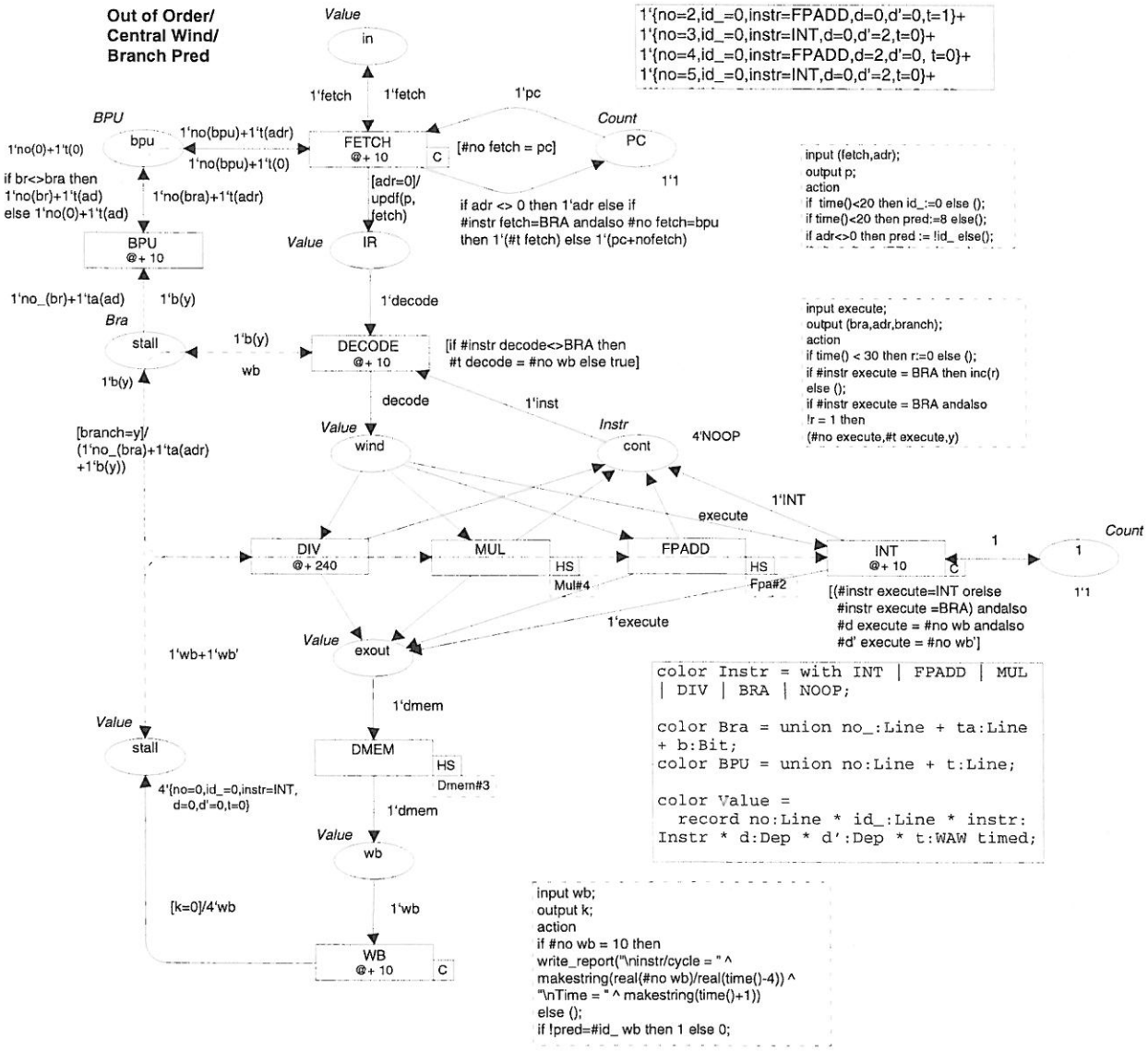



Figure 2: Design/CPN model of a superscalar processor.

and execution unit transitions:
**var fetch, decode, execute, wb, commit : Value;**

Guards are used at execution unit transitions to select correct instructions for execution. For example, the following guard filters an instruction of type **MUL** at the multiplier unit transition:
**[#instr execute = MUL]**
This models the correct instruction flow through execution units, but hazards also need to be modelled. Different types of hazards can be split into dependency and structural hazards. Dependency stalls are modelled by guards which check dependencies against instructions that have been written back. For example an RAW (read after write) or true dependency can be checked at a decode transition using the following guard:
**[#d decode = #no wb]**
This checks that the number of an instruction at writeback satisfies the dependency **d** of a decoded instruction waiting to issue.

Similarly we can check for WAW (write after write) or output dependencies by using the following guard at the **DECODE** stage where **t** is defined as having the same target register as a previous instruction which has already or which is expected to be written back to the same register:
**[#t decode = #no wb]**
Note that we use **t** here as an instruction number and not as a register number.

Aditionally, we need to be able to cope with structural hazards. Structural hazards occur when some combination of instructions cannot be accomodated because of resource conflicts. These need to be detected and issuing instructions need to be stalled. Structural stalls can be modelled in Petrinets by using feedback from functional units to indicate that a functional unit is free.

# 5 Superscalar issues

## 5.1 Multiple issue

Superscalar machines depend on the ability to execute multiple instructions in parallel. This is known as *Instruction Level Parallelism* [1]. Multiple issue exploits instruction level parallelism by fetching and decoding more than one instruction at a time.

The use of a fixed length instruction set enhances parallelism. For our multiple issue model we have decided to fetch four fixed length instructions

at a time. In order to do this the select Guard at **FETCH** needs to be changed to accommodate this. Now we have to fetch instructions as a block (see Subsection 8) To do this we fetch a block of four instructions defined as a record:

```
1'{blockno=1,
      b1 = {no=1,instr=INT,d=0,d'=0,t=0},
      b2 = {no=2,instr=INT,d=1,d'=0,t=0},
      b3 = {no=3,instr=FPADD,d=0,d'=2,t=0},
      b4 = {no=4,instr=FPADD,d=0,d'=0,t=0}
}
```

The PC count must also be updated accordingly:

**1'(pc+nofetch)**

Instructions are fetched into a queue (dispatch stack). We assume a queue of length eight split into upper and lower queues each being capable of holding a block of four instructions.

**var bl : block;**

Instructions from the lower queue are considered for decode and are replaced by the top half when all instructions from the lower queue have been decoded. In Petri nets two places separated by a transition can be used to model the two queue halves:

**var upper, lower: block;**

All instructions must be decoded and issued before another set of instructions is fetched. This is achieved by having a corresponding buffer size control node with four tokens representing the lower queue size. An up/down counter must be used to service incoming requests in order to determine the number of instructions that can be issued.

We can implement the instruction type checking and the dependencies and also the issue order provided the instructions are modelled at decode using an addition multiset:

**1'I1 + 1'I2 + ..;**

When the instructions are modelled like this we can relate the instructions to one another and compare them:

**[#t I1 <> #t I2]**

This is not simply modelled in Design/CPN because the number of instructions issued may vary. In Design/CPN it is difficult to recognise dynamically a variable number of buffer control size tokens and control a specific number of instructions waiting to be issued. Because of this an up/down counter must be used to determine how many instructions are left in the

issue window before a fixed number of requests can be serviced. This aspect is now being investigated.

## 5.2 Out of order issue and execution

So far we have assumed a model of in order issue. This section covers out of order issue and out of order execution. To issue out of order implies using a buffer(s) or window(s) in which to store instructions waiting to execute. The buffer, called an instruction window, is placed between the instruction decoder and the functional units.

There are two ways to implement the instruction window. The first is to centralize the window. We can model this in Petri-nets by using a place after **DECODE** to mimic a central window where collected tokens in the central window represent unissued instructions. To control its size we create a corresponding buffer control size node of colour Instr and initialise it with size tokens or instructions.

**Color Instr = with INT | FPADD | MUL | DIV | NOOP;**
**size'NOOP;**

The buffer control size node is connected to the decode transition by an arc. Each time an instruction token is issued from decode to the central window a corresponding token is removed from the buffer size control node. When size tokens are removed from the buffer size control node the central window reaches its full capacity and no more instructions can be decoded.

Another way of implementing the instruction window is to distribute individual buffers called reservation stations to each of the functional units, buffering instructions destined for a particular functional unit at the input of that functional unit. Multiple windows or reservation stations can be modelled in Petri nets by using a number of buffer places, one for each reservation station. In this case we need to control the numbers of each instruction type entering each buffer place.

To control the size of the reservation stations we make use of the corresponding size control node of colour **Instr** as for the central window but this time initialise it with a multiset which corresponds to the individual sizes of each reservation station:

(1) **2'INT + 2'FPADD + 2'MUL + 2'DIV**

Every time an **INT** instruction is issued from the decode stage a Guard is used to remove the corresponding type of instruction from the buffer size control node.

An arc inscription selector is used from decode to each reservation station

place to control the instruction selected. For example, for the arc leading from the decode transition to the reservation station for the **INT** unit we have the following arc inscription:
**[#instr decode = INT]/1'decode**

If the instruction of type INT is taken from 1) the corresponding multiset will result in the buffer size control node:
(2) **1'INT + 2'FPADD + 2'MUL + 2'DIV**

When two **INT** tokens are removed from the buffer size control node this is equivalent to the reservation station at the **INT** execution unit being full.

# 6 Branch prediction

Branch prediction can reduce the average branch delay by predicting the outcomes of branches during instruction fetching. Branch prediction makes use of a Branch Prediction Unit (BPU) for building up a database of speculative branch information.

Branch instructions are represented in colour using the following format:

```
color Value = record
                no: Line *
                BRA: Instr *
                d: Dep *
                l: loop *
                t: Target timed;
```

where **d** represents a dependency, **l** indicates whether the instruction will loop and the number of times for the loop is predetermined and **t** represents the correct target of the branch instruction which is also predetermined by the user.

For Branch Prediction we assume a branch prediction unit using a target buffer which dynamically collects information about the most recently executed branches. The data structure for the BPU is defined as follows:
**color BPU = union no:Line + t:Line;**

**FETCH** checks the instruction against the target buffer to see if their is a predetermined branch for that branch **no:Line**. If the fetched instruction is a branch the branch target buffer indicates the predicted outcome using the target address **t:Line**.

In the case of a misprediction two values are sent back to the BPU, a branch instruction number defined as the first part of the union: **no_(bra)**, and the correct target defined as the second part of the union: **ta(adr)**. This is achieved with the following arc inscription flowing from the corresponding execution unit to the BPU :

**[branch=y]/(1'no_(bra)+1'ta(adr)+1'b(y))**

This transfers a number for the branch instruction **no_(bra)** and a corresponding BPU update prediction target address **ta(adr)**. The target for a particular address is checked for by the **FETCH** with the following arc insrciption:

```
if adr <> 0
then 1'adr
else if #instr fetch = BRA andalso #no fetch = bpu
       then 1'(#t fetch)
       else 1'(pc+nofetch)
```

# 7 Recovery

Recovery is a process which cancels the effects of instructions that were issued under false assumptions using speculative execution. In asynchronous circuits, recovery from an incorrect branch is accomplished by assigning a tag or colour to each instruction (see e.g., [5]). When a branch is encountered the tag or colour is changed and instructions which do not have the current tag are considered invalid and are terminated further down the pipeline. For speculation it is assumed that a tag needs to cover a range of values between branches and an integer tag is proposed so that each instruction between a branch is represented by a unique **id_**. An additional record value is provided for this in the instruction definition:

```
color Value = record
                    no: Line *
                    id_: Count *
                    instr: Instr *
                    d: Dep *
                    d': Dep *
                    t: Target timed;
```

The value of the **id_**is changed for each instruction by incrementing it modulo some limit over which the range of tags are defined. Each time a branch instruction is fetched the value of **id_** must be changed and a list created of all the invalid **id_** values built up from an incorrect speculation.

To make the recovery process more efficient instructions could be checked and eliminated earlier than the writeback stage to speed up recovery. This is done by using a selector at certain stages to determine if the instruction is valid or invalid:
**[instructionvalid=0]/1'execute**

# 8 Caches

While modelling caches we assume that direct cache modelling is being used. In Design/CPN an instruction cache can be modelled by having an extra **CACHE** place between the instruction memory and the processor. An extra **READ** transition is also inserted after the cache place which models a read access made by the processor to the cache for an instruction. The original **FETCH** now becomes a fetch access to memory to load a block into the **CACHE** place.

When modelling caches we need to be able to reference values in the cache. In order to do this tags need to be added to the instructions to model their relative cache positions. We have done this by defining a new record which defines a block of instructions and adding extra fields called cachetag and cacheblock:

```
color Block = record ct:Line * cb:Line * b1:Value * b2:Value
b3:Value * b4:Value;

{ct=0,cb=0,
      b1={no=0,id_=0,instr=NOOP,d=0,d'=0,t=0},
      b2={no=0,id_=0,instr=NOOP,d=0,d'=0,t=0},
      b3={no=0,id_=0,instr=NOOP,d=0,d'=0,t=0},
      b4={no=0,id_=0,instr=NOOP,d=0,d'=0,t=0}
}
```

The cachetag is used to specify the set of memory blocks which can be mapped to a specific cacheblock. The cachetag value is calculated as the upper part of the pc address. The PC address is formatted as follows:
**pcaddr = cachetagbits;cacheaddrbits;offsetbits**

Here we assume instructions have a fixed length (32 bits). The cachetag for an instruction block is calculated by dividing its instruction address by $\mathbf{2^{cacheaddrbits+offsetbits}}$ where **cacheaddrbits** are used to specify the cacheblock address, and **offsetbits** are used to specify the offset of an instruction within a cacheblock.

Initially all the instructions in the cache are set to **NOOP** and their cachetags set to zero, **cachetag=0**.
To access a cache address at READ we select the cacheblock address by using a transition guard:
$[\#\mathbf{cacheblockread} = ((\mathbf{pcaddrnoDIV2^{offsetbits}})\mathbf{mod2^{cacheaddrbits+offsetbits}})]$

# 9 Preliminary simulation results

Design/CPN simulation output consists of a large text file. In order to display the timing results in a more suitable format, a graphical tool, written in Tcl/Tk, was written to display the timing results of the simulation in a more suitable format. A software filter processes the output from Design/CPN and drives the Tcl/Tk display.

Figure 3 shows the schedule obtained for a set of instructions which have been simulated using the model shown in Figure 2. In the schedule, the second instruction has been delayed from executing by a Write-After-Write dependency on the first instruction. The second instruction then proceeds to execute, but the subsequent instructions which have a Read-After-Write dependency on the second instruction are delayed, and accumulate in the central window before the dependency on the second instruction has been resolved. Subsequently, instructions are shifted out of the central window and execute in parallel. This allows instructions 6,4 and 8 to enter the FPADD pipestage (4 stages) and execute in parallel. It can be seen from the schedule that instructions execute out of order and are written back out of order.

# 10 Discussion

The use of Design/CPN on a complex processor model shows that the tool has many obvious strengths but also weaknesses.

The strength of Design/CPN modelling lies in its use of colours and datatypes which can be be used for modelling basic processor instructions and also more complex memory types such as caches. Because of its typing,

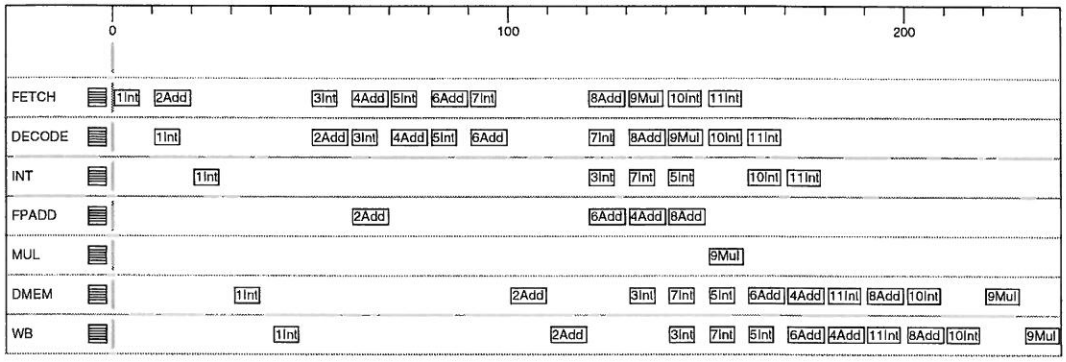



Figure 3: Instruction timing diagram.

Design/CPN can be used to capture dataflow adequately using arc and guard expressions.

Pipelines are easily captured using a sequence of transitions and places. Hierarchy is expressed easily by using pages over transitions.

Dependency hazards can be easily modelled by placing guards at execution units. Structural hazards are easily modelled using feedback. Register locking can also be modelled.

In order and out of order issue modelling is easily captured. The natural flow in Design/CPN is out of order or arbitrary but this can be changed to in order by using Guards. Design/CPN is also good at modelling buffers, and therefore reservation stations can be easily modelled.

Multiple issue can be modelled but not without difficulty. This is because it is difficult to express dynamically the relationship between token flow expressions on different arcs. Compensatory techniques have to be found to overcome this such as counters to recognise how many tokens can be allowed to pass over arcs.

Branch prediction can be modelled. However, there is a possibility of arbitration occurring between the fetch unit and the branch prediction unit. Arbitration is not easily simulated, as it has no priority of choice and cannot make a decision about the correct flow of data. Although the simulator tends to correct this, it is not an adequate solution at the moment. This requires further investigation.

Recovery is difficult to model, but it is naturally difficult to model in

asynchronous designs. The techniques to overcome this are based on assigning tags to instructions, and terminating instructions when they become invalid. The problem with this is that more than one tag needs to be used for out of order issue.

## 11 Conclusions

We have described the main aspects of our approach to the modelling of a superscalar processor with Coloured Petri nets, and some preliminary results. The model is successfully run using the Design/CPN software. Typical speed of simulation is about 3000 instructions per minute under Linux on a 166Mz PC. Design/CPN provides the designer of a real-time system with both qualitative analysis of reachable states and analysis of timing characteristics, such as worst case execution for a block of instructions. A number of modelling issues has been revealed that require further investigation of the descriptive power of Design/CPN. Further tool development should allow extraction of timing parameters from Design/CPN output files, and their display using graphical tools not available within the Design/CPN environment.

## 12 Acknowledgements

This work is supported by EPSRC grant GR/L28098 (project TIMBRE) and Esprit LTR Project 20072(DeVa). The authors also wish to thank the referees for helpful comments on the paper.

## References

[1] D.K. Arvind and V.E.F. Rebello. Instruction-level parallelism in asynchronous processor architectures. Proceedings of the Third Int. Workshop on Algorithms and Parallel VLSI Architectures, M. Moonen and F. Catthoor (Eds), Leuven, Belgium, August 1994, Elsevier Science Publishers, pp. 203–215.

[2] J.B. Dennis. Modular, Asynchronous Control Structures for a High Performance Processor. Proceedings of Project MAC Conference on Concurrent Systems and Parallel Computation, June 1970, pp. 55-92.

[3] K. Jensen. Coloured Petri Nets. Basic concepts, analysis methods and practical use. EATCS Monographs on Theoretical Computer Science, Springer-Verlag 1992.

[4] A. Semenov, A.M. Koelmans, L. Lloyd, and A. Yakovlev. Designing an asynchronous processor using Petri nets. *IEEE Micro*, 17(2):54–64, March 1997.

[5] N.C. Paver. The design and implementation of an asynchronous microprocessor. Ph.D. Thesis, University of Manchester, 1994.

[6] R.R. Razouk. The Use of Petri Nets for Modeling Pipelined Processors. Technical Report 87–29, University of California, Department of Information and Computer Science, 1987.

# ALPiNe: A Hardware Computing Platform for High-Level Petri Nets

S. Bulach, H. Baur, H.-J. Pfleiderer, Z. Kucerovsky*

Department of Microelectronics, University of Ulm,
Ulm, D-89069, Germany

*Department of Electrical & Computer Engineering,
University of Western Ontario
London, Ontario, N6A 5B9, Canada

May 5, 1998

***Abstract**: A motivation for the design of a novel hardware platform for processing algorithms based on High-Level Petri Nets is presented. ALPiNe (Asynchronous High-Level Petri Net) processor is aimed at embedded discrete-event control applications and is characterized by its natural incorporation of external stimuli into the computation flow. The processor consists of two layers of hardware: one for determining when and which computations will take place, and another for effectively performing the actual computations. A hybrid architecture and hardware organization are described in detail. The process of software development is presented, augmented with an illustrative example. In conclusion, comments on advantages and possible future implementations are made.*

## Introduction

The last thirty five years have demonstrated that Petri Nets (PN) have the inherent ability to survive, and not only to survive. The rate at which this concept was applied, modified and extended is perhaps analogous to the exponential PN state explosion phenomenon. One such extended version of a Petri Net is known as a Coloured Petri Net (CPN) [KJ96]. The CPNs could be classified as High-Level PNs, being more abstract than their predecessors. In addition to allowing abstract token types they may also incorporate hierarchical capabilities and time extensions [WA94]. Several commercially available software packages support efficient creation, simulation and analysis of algorithms based on CPNs [AW91]. This paper deals mainly with the Design/CPN [MS93] package and its application to modelling complex discrete-event control systems.

Due to their flexibility Petri Nets have been successfully applied to a wide range of problems [TM89], and were found to be best pertinent in performance evaluation and in the design of communication protocols. In general, PNs are well suited for modelling and analysis of systems that may possess concurrent, distributed, parallel, event-driven, asynchronous, reactive and non-deterministic qualities. In yet another dimension, Petri Nets can be used from gate-level hardware design, through register-transfer level (RTL) modelling, all the way to the design of complex software systems [JP81]. However, it should be noted that because Petri Nets have well defined static and dynamic properties they do well in

modelling processes, systems and algorithms that in some sense exhibit analogous characteristics. In other words, it is not very efficient to model spherical objects using triangles!

A very broad class of systems that Petri Nets handle well is known as *reactive systems* [BM91]. A reactive system, informally defined, is said to have an on-going interaction with its environment. It receives the input stimuli from sensors, processes them, communicates back to the environment through actuators, and keeps track of the state [HP85]. Furthermore, embedded systems could be viewed as reactive systems that must satisfy hardware constraints, whereas real-time systems are reactive systems which must also satisfy timing constraints [MP96]. Design, verification and efficient implementation of reactive (embedded and/or real-time) systems still remains an active research field.

At the gate-level, Petri Nets could be used for both combinational and sequential circuit design, especially when gate delays are taken into account. Asynchronous (operating without a clock) sequential circuits are particularly well suited for modelling with PNs, although attempts have been made to extend PNs to handle synchronous sequential circuits, or even mixed asynchronous-synchronous systems [TT97]. This is mainly due to the asynchronous event-driven nature of processes that PNs represent. Note that asynchronous sequential circuits represent simplest reactive processes. Recently, PN based methods were developed which allow automatic synthesis of asynchronous circuits based on Signal Transition Graphs (derivative of marked PN), or signal switching specifications [JC97]. However, this methodology is restricted to the relatively small controller circuits. There is also ongoing research at utilizing PNs at a register-transfer level for system verification and synthesis [SK97]. However, both gate-level and RTL represent so-called *direct solution* or implementation, where a given algorithm is hardwired and is intended for a specific application, as is the case, for example, in the ASIC (Application Specific Integrated Circuit) design. Another type of solution is known as *indirect* and involves use of programmable devices, such as microprocessors and processor-based systems.

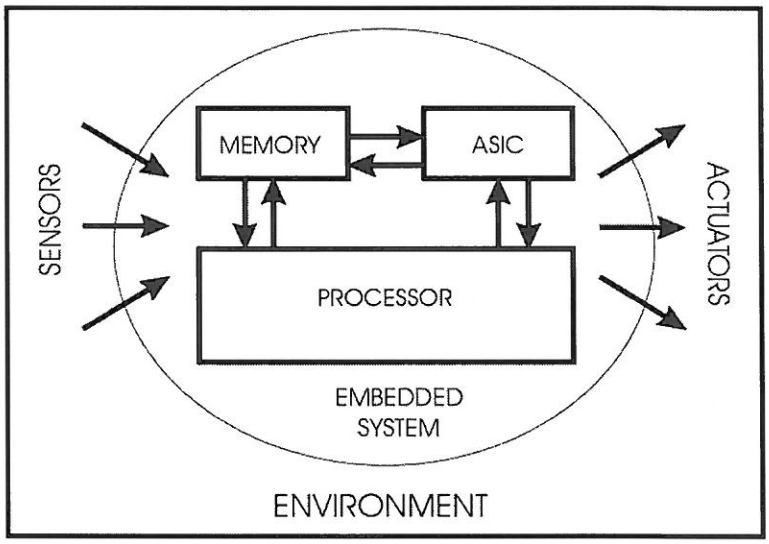



**Figure 1. A Simplified Structural View of an Embedded System.**

An indirect solution is essentially a software design because the hardware computing resources are provided and are fixed, while the algorithm resides in memory. A complex reactive system would normally contain a processor which has I/O capabilities in order to

communicate with the environment through sensors and actuators. A simplified structural view of such a system is given in Figure 1 [GD94]. Much attention has been given to the design of complex software systems using High-Level Petri Net concepts (for examples, refer to [TH92], [RS96]). However, most of the efforts are directed at employing commercially available processors, thus shifting the focus to the efficient computing strategy given the computing resources and I/O capabilities. The weakness of this approach is that while reactive systems, or in other words, most of the embedded systems, are intended to be reactive or event-driven, commercially available processors are predominantly of the control flow architecture. They are excellent number crunchers but inefficient communicators. Thus, processing cores are normally surrounded by a plethora of peripherals to enable interaction with the environment. However, the overall architecture still remains control driven.

In system design it is often the case that the system description at the early stage of the design is done in dataflow, functional, or some other unconventional language. However, the final implementation is forced to be translated into the procedural language equivalent and then compiled into the target code. The point is that even if the desired programmed functionality of a software system is reactive, event or data driven, the final implementation is done on a hardware platform that is strongly control driven. Could these inefficiencies be addressed at an architectural level?

**ALPiNe Architecture**

The universality of Petri Nets could be attributed to their flexibility in ascribing a meaning to the concept of tokens, places and transitions. For different applications both places and transitions may mean different things. Some typical interpretations are given in Table 1 [TM89].

**Table 1. Possible Interpretations of Transitions and Places**

| Input Places | Transitions | Output Places |
|---|---|---|
| Preconditions | Event | Postconditions |
| Input Data | Computational Step | Output Data |
| Input Signals | Signal Processing | Output Signals |
| Resources Needed | Task or Job | Resources Released |
| Condition(s) | Clause in logic | Conclusion(s) |
| Buffers | Processor | Buffers |

Note that the second interpretation with input and output data, and a computational step could be well applied to reactive systems. That is to say that a computational step is invoked only when the required input data is present. This principle is reminiscent of the dataflow architecture with a data driven computing organization [TB82], where the computational step is executed once the necessary data becomes available. The computational step itself may represent the execution of one or more instructions, or in other words, a subroutine of arbitrary size, which is concerned with pure computations.

The above observation leads to the conception of a hybrid architecture comprising both data flow and control flow characteristics. Thus, data flow principle is employed to

determine whether or not (and which) computational step is invoked, while control flow principle is used to efficiently perform number crunching once it has been determined which step to execute. These architectural principles are realized in the proposed ALPiNe (Asynchronous High-Level Petri Net) processor. ALPiNe processor has two primary modules: a Petri Net Decision Unit (PNDU) and a Computing Engine (CE). The PNDU module is responsible for making decisions regarding the PN structure, such as determining which transition fires based on the Precondition, or a special condition that must be satisfied for a transition execution to initiate. The PNDU processes its own code, and coding is described in the following section. The CE module is a conventional processor core, optimized for logic, arithmetic and bit manipulation instructions. The CE module also has its own code. Detailed software and hardware organization of the ALPiNe hybrid architecture is presented below.

**ALPiNe Software**

First of all, the design flow must be thoroughly examined in order to see how High-Level Coloured Petri Nets (in particular, developed with Design/CPN) could be processed by the ALPiNe processor. Clearly, the CPNs manipulated by the Design/CPN package could not be implemented directly on the ALPiNe processor. They are simply too complex, have abstract data types, and the code size is most likely too large. The design flow, shown in Figure 2, should help to see the role of the ALPiNe with respect to CPNs, as well as what type of CPNs could be processed by this processor.

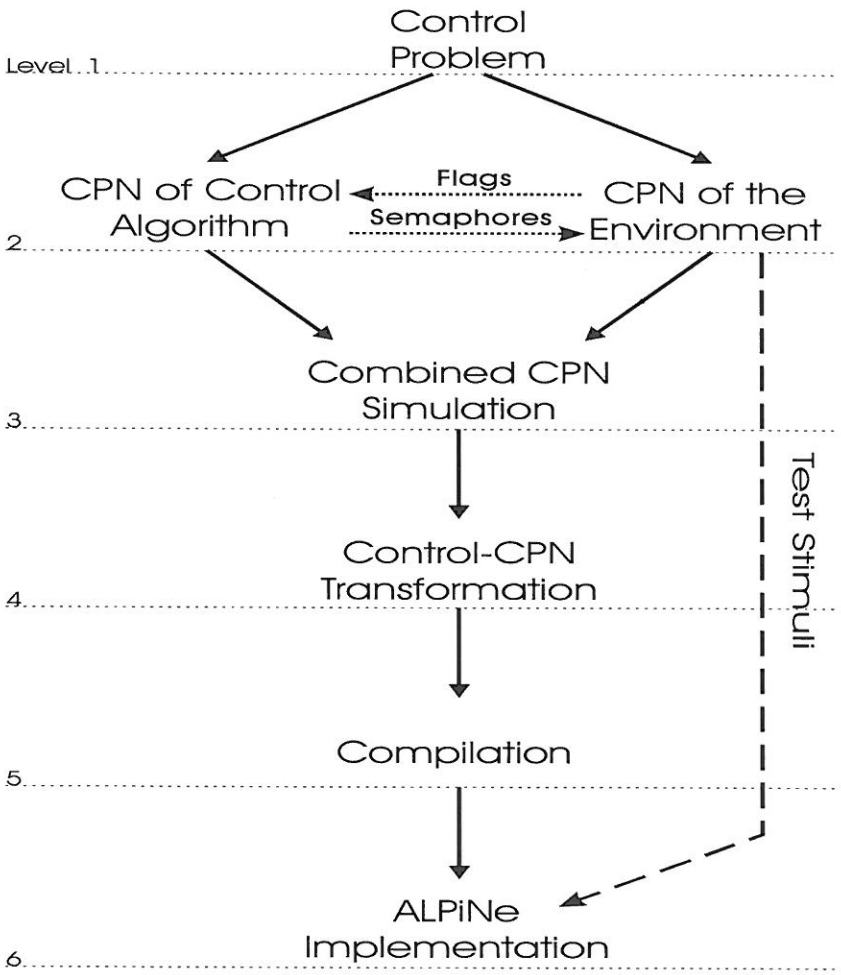



**Figure 2. Design Flow for a Control Problem Solution Using the Coloured Petri Nets.**

Given a control problem, a solution, which involves the CPN of the control algorithm and the CPN of the environment, is developed with the Design/CPN. In this approach, the environment CPN communicates to the control CPN through *Flags*, while the control CPN uses *Semaphores* to signal its state to the environment. This method allows to model the on-going interactions between the two. The combined model can be simulated and formally analyzed to verify the correctness of the algorithm. If the control model is satisfactory, it must be transformed from its *high-level* CPN (level 2,3) to the *intermediate-level* CPN (level 4). The intermediate-level CPN must be of the form which would allow compilation into the ALPiNe machine code. Thus, the high-level CPN (level 2, 3) is in the format understood by the Design/CPN. The intermediate-level CPN must be in the format understood by the compiler. The rules of transformation and compilation must be clearly defined and are based, of course, on the intermediate-level CPN. The important issue is to optimally specify the intermediate-level CPN which will serve as a bridge between the two different worlds of high-level CPN software (Design/CPN) and the hardware (ALPiNe). Furthermore, the intermediate CPN must deal with real-world binary variables which are digital signals on the wires and processor input pins.

It was chosen that the current version of ALPiNe would handle problems modelled with the CPNs of the Finite State Machine (FSM) subclass. That is, at any given time the processor could only be found in one state (or place). This means that one or more transitions could be enabled at the same time. However, the execution algorithms and the CPN structure ensures that only one will fire, remove the enabling token from the input place and produce a token for a corresponding output place. The FSM restriction on CPNs is achieved by allowing at most one input and output place from any given transition. This restriction does not apply to special Flag and Semaphore places.

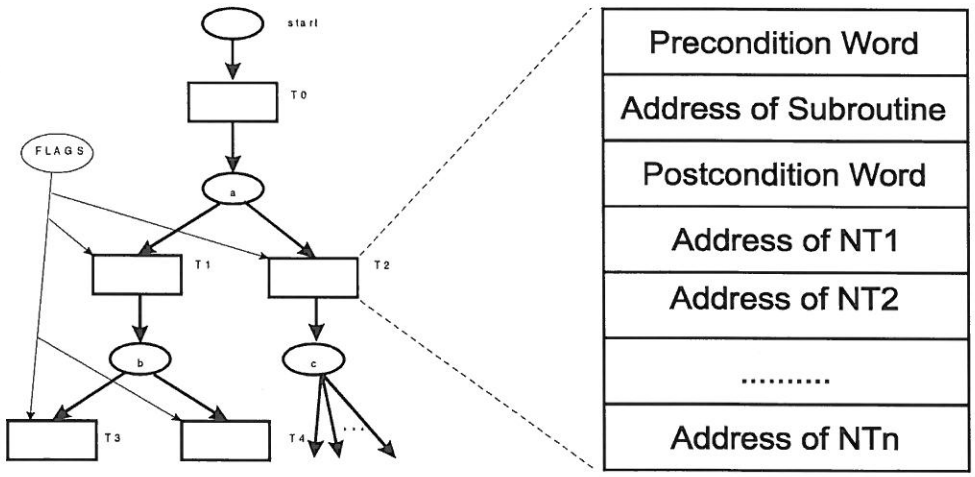



**Figure 3. A General Transition Coding Format**

The program to be executed on the ALPiNe processor (machine code) mimics the intermediate-level CPN and consists of a list of transitions. Each transition is coded according to the general pattern shown in Figure 3. It contains a Precondition Word, an Address of a Subroutine, a Postcondition Word and a list of addresses of Next Transitions (pointers). Note that places do not appear explicitly in the code. The concept of places is dissolved by the presence of preconditions, postconditions and pointers; these three combined precisely determine the unique state of the processor at any given time. General rules of converting a Design/CPN executable intermediate-level CPN into the ALPiNe code are given below.

The principal components of a CP net are: Data, Places, Transitions, Arcs, Input Arc Inscriptions, Guards, and Output Arc Inscriptions [MS93]. A full translation of the CP Net into the ALPiNe machine code requires the specification of conversion for each of these components.

***CPN Data***: Data objects are known as tokens, while datatypes are called colorsets. The high-level CPN has fairly complex tokens and colorsets. The intermediate CPN, on the other hand, must reflect hardware dependence in that all (or most of the) tokens are binary variables. These binary objects could be combined into different binary multisets such as Flags (reflect the current state of the environment) and Semaphores (reflect the current state of the system). The size of Flags and Semaphore multisets is determined by the width of their respective registers (Figure 4). This sets the upper boundary on the number of encoded states. Refer to the Railroad Crossing example presented below to see how these multisets could be arranged.

***Places***: According to the Design/CPN definition, places are locations for holding data. For high-level CPNs places are important components of the algorithm. They represent the state of the modelled system. At the machine code level, places have no real interpretation, as only transitions are encoded as a program. Note, this does not mean that the modelled system becomes stateless! The state may be encoded with Semaphore tokens as required.

***Transitions***: In high-level CPNs these are defined as activities that transform data. At the machine code level these are primary elements of the program. They specify what computational step (i.e. subroutine) is to be executed once the transition is enabled. In addition, at the intermediate-level transitions also encode Preconditions and Postconditions. This procedure could be observed again from the Railroad Example.

***Input Arc Inscriptions (IAI)***: These specify the data that must exist for an activity to occur.

***Guards***: Define conditions that must be true for an activity to occur. In terms of the present ALPiNe encoding for the intermediate-level CPN, both IAI and Guards form a Precondition Word. Ideally, the IAI should indicate which tokens (or binary elements) are important to test in the Precondition. Guard specifies the binary AND operation performed on these variables. If the outcome is TRUE, the transition will fire.

***Output Arc Inscriptions***: Specify data that will be produced if an activity occurs. This CPN component is encoded into the Postcondition Word, in which binary variables of the Semaphore multiset (current state indicators) are set or cleared upon requirements.

Again, refer to the Railroad Crossing Example to observe the relationship between the Design/CPN components and the corresponding elements of the ALPiNe encoding, i.e. Preconditions, Postconditions and Transitions, as shown in Figures 9, 10 and 11. The overall PNDU code structure has the following components: Precondition Word, Address of Subroutine, Postcondition Word, Number of Next Transitions (NNT) indicator, and the actual addresses of Next Transitions, NTA1, NTA2,..., NTAn. Thus, the PNDU has a variable length instruction format. This means that the information about the transition length must be explicitly encoded. The NNT field contains the explicit number of next transitions

The execution of a subroutine is the task of the CE module. The encoding of its logic and arithmetic instructions is done according to the principles described in [HP90].

## ALPiNe Hardware

As mentioned above, ALPiNe consists of two modules, each optimized for its own unique function. The block diagram of the ALPiNe processor is shown in Figure 4. Both PNDU and CE have their dedicated memory, with their own data and address buses. This ensures optimum concurrency of the operation.

The PNDU is responsible for processing PN structure related information. It receives new information from the environment through the FLAGS pins. Every new event or change on FLAGS is registered and queued in the FIFO block. At the same time New Event Detector produces a signal to indicate that the new event has been detected. The PNDU communicates to the outside world through Semaphores. The Semaphores are combined with "oldest" Flags to form a State Word. The Comparator checks for equivalence between the State Word and the Precondition Word. A Transition Register File is where Transition information is temporarily stored while a comparison is performed. The Precondition Word, Subroutine Address and the Postcondition Word of the currently executed transition become available to the Computing Engine. The CE also receives a prompting signal to start processing a subroutine and upon execution indicates that it is finished. It has an option of altering the Postcondition Word based on the results of computation.

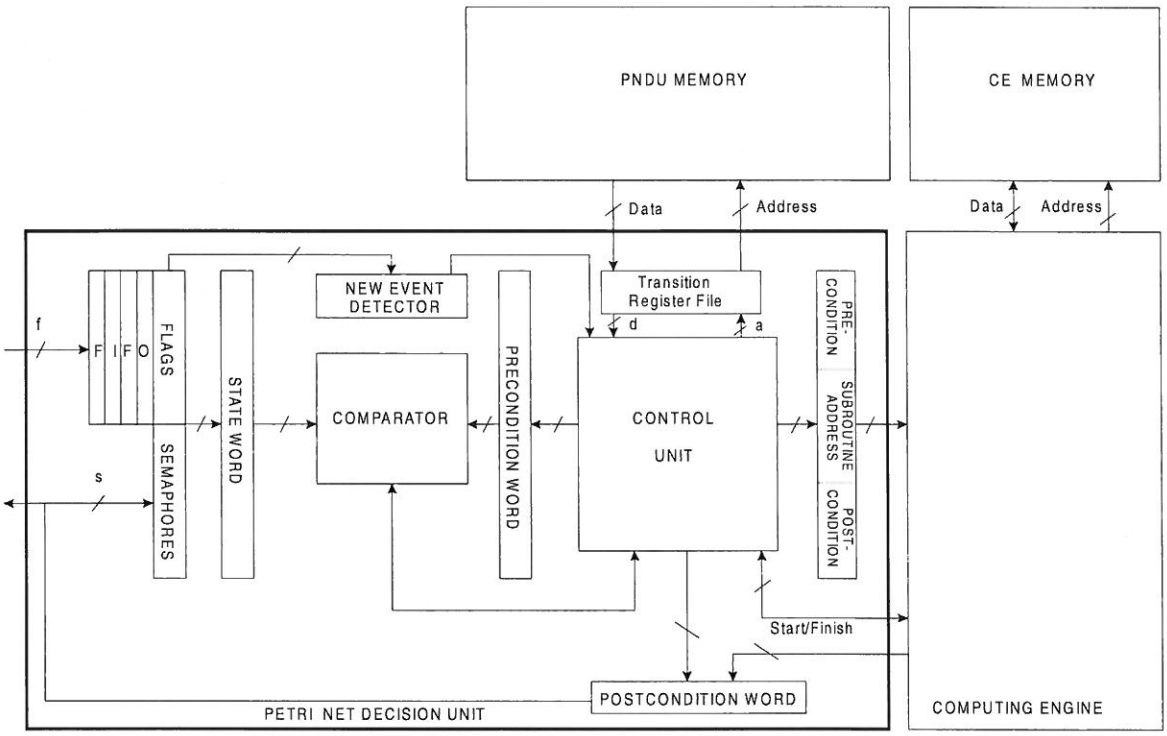



**Figure 4. Simplified Block Diagram of the ALPiNe Processor.**

The functionality of the PNDU module is flowcharted in Figure 5. For the following discussion refer to Figure 10, transitions T0, T1 and T2. Upon Power On, the Transition Pointer (equivalent of the Program Counter) is initialized to point at the memory location of the first transition in the program (Step 0). The PNDU Memory is accessed and the transition is loaded into the Transition Register File (Step 1). The Precondition Word is then compared to the Status Word (Step 2). If they are equal, the transition is said to fire. The Subroutine Address is passed on to the Computing Engine along with the Start signal (Step 3). The Precondition is also available to the CE for examination, and the Postcondition could even be

modified by the CE. After the CE completes the subroutine, it asserts the signal Finished and may overwrite the Postcondition (Step 4). If the Postcondition was not modified by the CE, the SEMAPHORES register receives the original Postcondition. At the end of this cycle the Transition Pointer is given the first address contained in the Next Transition List of the fired transition (Step 5). The execution then continues in the same fashion from Step 1 to Step 2.

If upon testing the Precondition against the State Word it is found that they are not equal, the Transition Pointer is given the second address from the Next Transition List of the previously fired transition. If the second Next Transition does not fire the subsequent transitions are all tested until the Next Transition List is exhausted. This is represented by the cycle of Steps 1,2,6 and 7 until all the transitions on the list are tested. In this case the execution cycle suspends (at place p6) and waits for new Flags or new events to arrive. Once a new event arrives, the Transition Pointer is updated to the first address on the Next Transition List and the testing of Next Transitions starts all over again until either one of them fires, or after testing them all the execution suspends and awaits a new Flag.

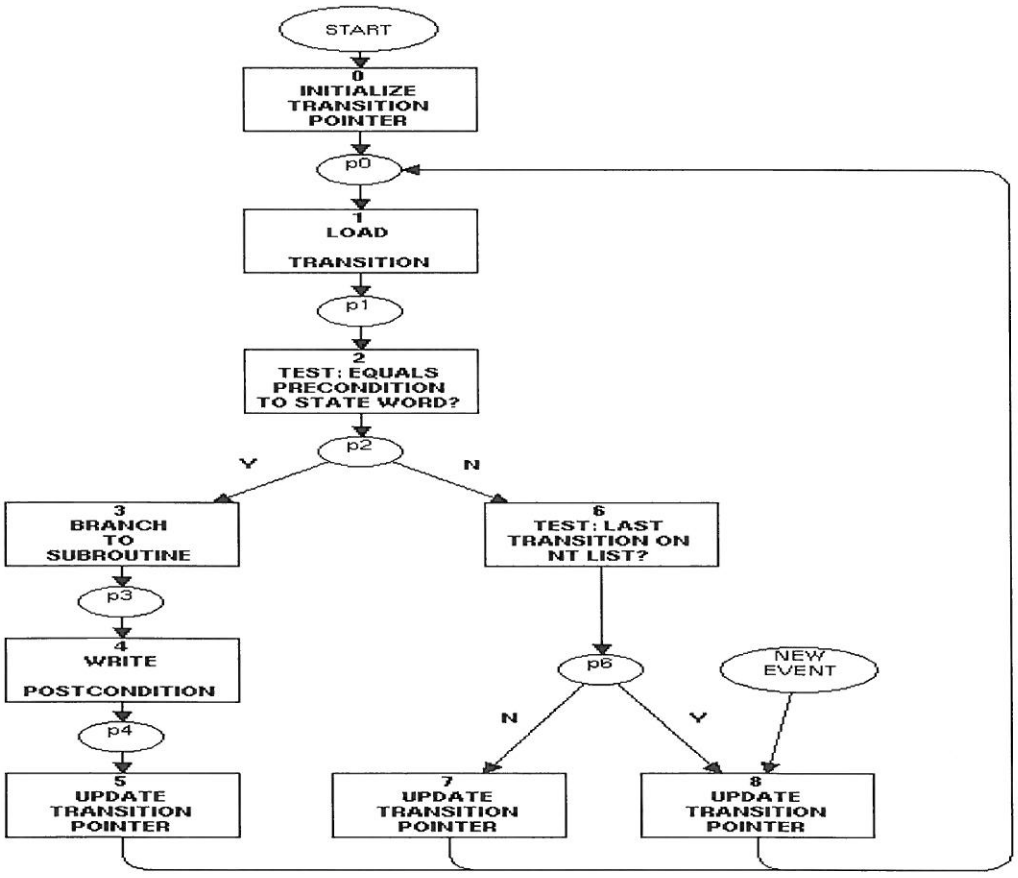



**Figure 5. Execution Cycle of the ALPiNe Processor.**

The architecture and organization of the CE is that of a simple processor core, as outlined in [HP90]. It is a processor with typical Fetch, Decode and Execute stages and a small instruction set (RISC), extended with bit manipulation instructions. The overall organization is based on the implementation described in [MG95]. Additions include the interface to the PNDU with ability to read Precondition, Subroutine Address and Postcondition, and to write Postcondition, and to assert control signals Start and Finished. In the execution cycle of Figure 5, the CE task is embedded into the Step 3 - Branch to

Subroutine. This is where the CE receives the Address of Subroutine from the PNDU along with the Start prompt, executes the code contained in the subroutine, signals Finished to the PNDU, and may return a modified Postcondition.

**Railroad Crossing Example**

A simple example is used to demonstrate the full translation of the CPN into the proper ALPiNe encoding. Figure 6 shows four sets of waveforms that represent operation of the railway crossing controller [CP96]. There are two sensors "r" and "l" located on the right and on the left of the crossing respectively. Long and short trains may come from either direction. For example, a long train coming from the right would produce a waveform shown in Figure 6 a), whereas a short locomotive coming from the left is shown in Figure 6 d). Based upon the sensor detection of the train the output variable "z" is set high in order to activate the closure of the railway crossing.

The control algorithms developed for this problem is given by the CPN of Figure 7. There are three main places (a, b and c) and the corresponding transitions T1, T2 and T3. The execution starts from the extra place START which introduces a machine token "m". This token is used to symbolically follow the flow of the execution of the net. The extra transition T0, which is always enabled at the start of the execution, is used to set or clear appropriate binary variables (Semaphores) before the execution of the main algorithm. The global fusion place FLAGS introduces tokens received from the environment CPN into the control CPN.

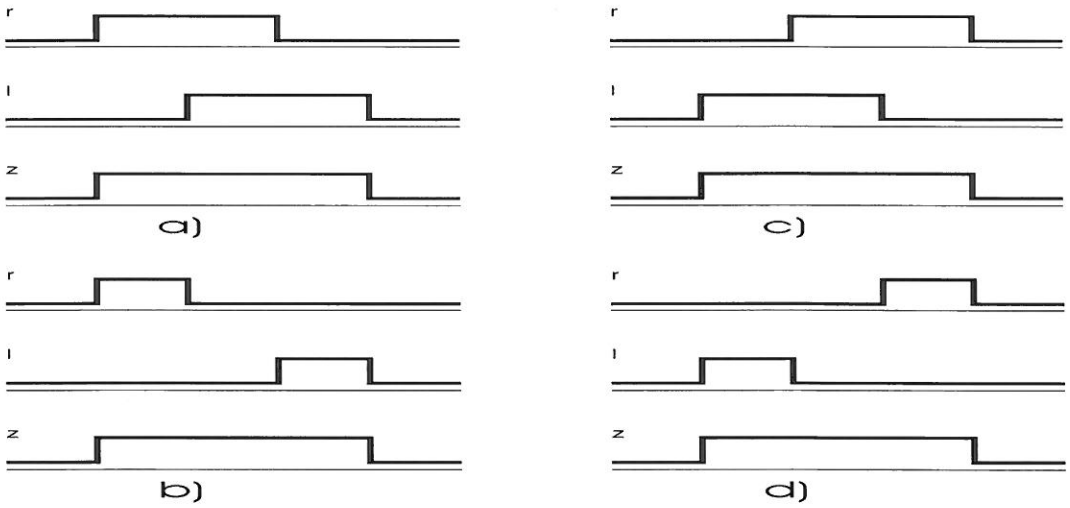



**Figure 6. Signal Waveforms for the Control of the Railway Crossing.**

A Coloured Petri Net of the environment is shown in Figure 8. The given net describes the detection of a long train coming from the right. The first transition initializes the net by producing (r,l) vector equal to logical (0,0). Then there are four events corresponding to the four transitions at times 20, 40, 70 and 90 (arbitrary time units). The token values are derived from the sensor waveforms and are deposited into the global fusion place FLAGS. The very instance they are deposited, they are picked up by the control CPN and processed. Note that this example demonstrates one way communication from the environment to the control CPN. However, two way communication can be easily achieved by introducing another global fusion place SEMAPHORES which would transfer status indicator tokens of the control CPN to the environment CPN.

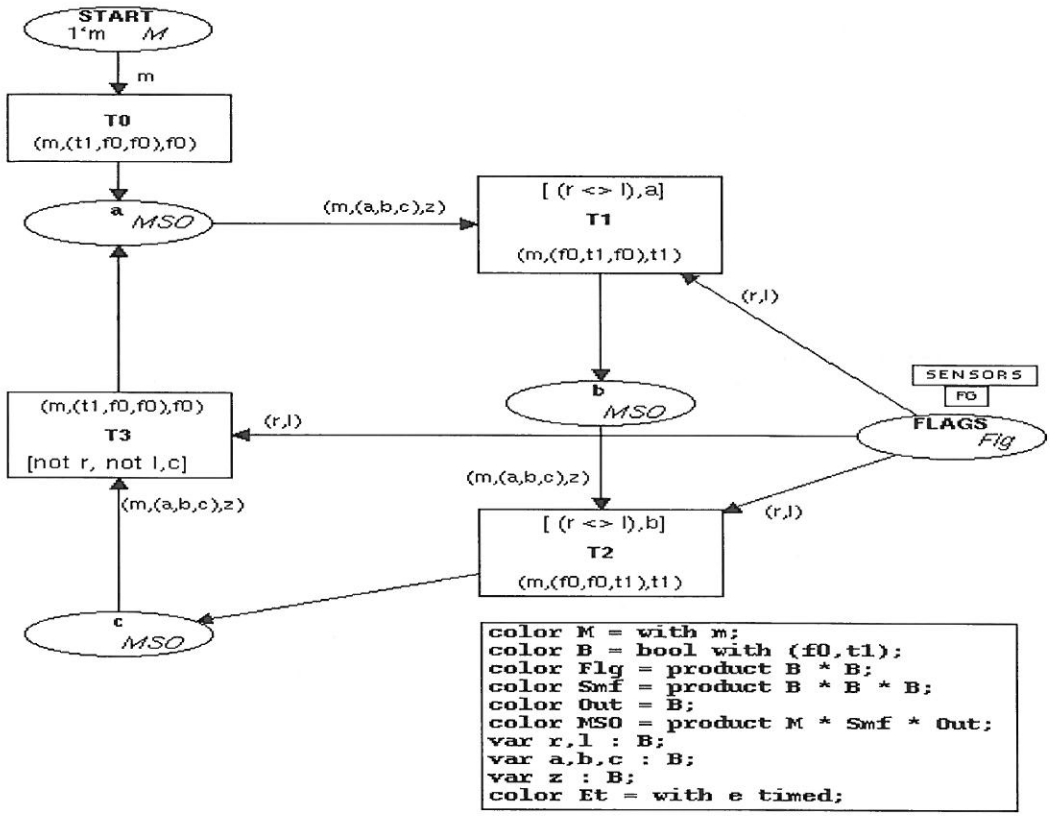



**Figure 7. The CPN of the Control Algorithm for the Railway Crossing.**

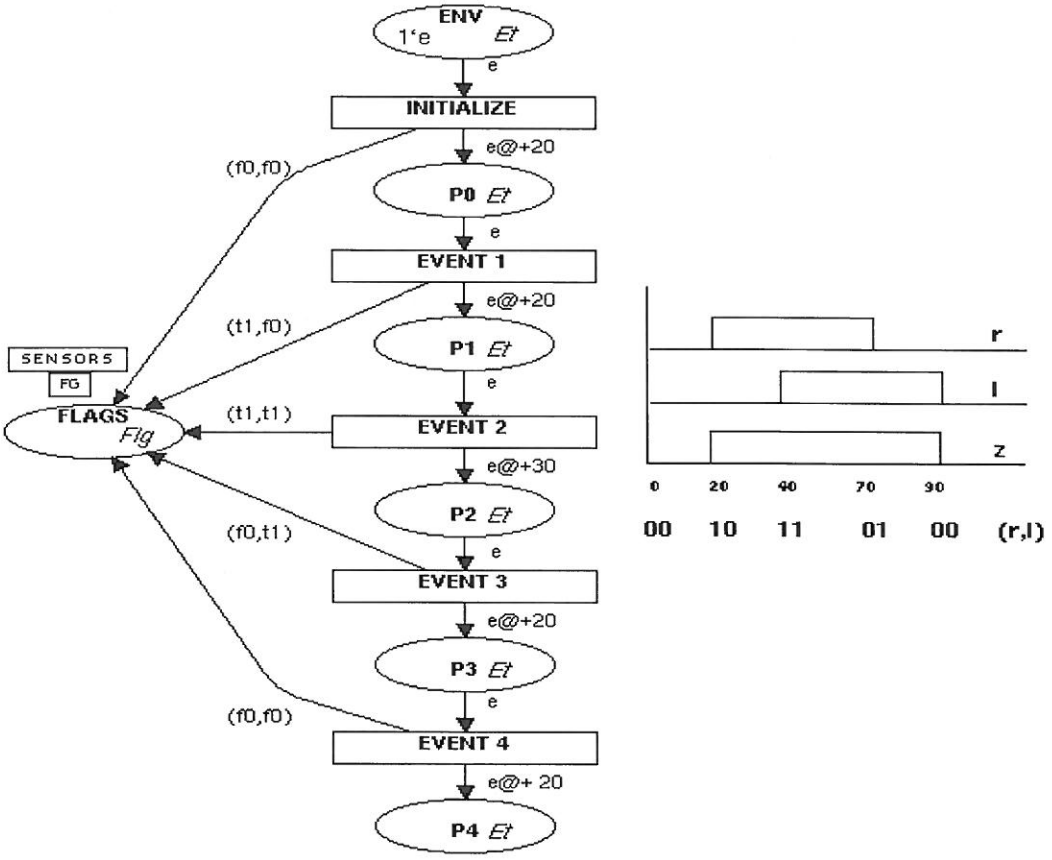



**Figure 8. The CPN of the Environment for the Railway Crossing Problem.**

The global declaration node contains a description of the variables used in this problem. Color M is assigned to the simple machine token *m* which flows from place to place and indicates in what place (or state) the algorithm is at a given time. Color B defines Boolean type with *false* assigned to an alphanumeric *f0* and *true* to *t1*. Each place on the main program is of the MSO type, which is a tupple of Machine, Semaphore and Output colors. Semaphore variables *a*, *b* and *c* are used to indicate the state, while the Output variable *z* sets the proper output value. In the hardware sense, there is no difference between the Semaphore and Output variables because both are the elements of the Postcondition Word register. However, for programming clarity it helps to separate them. The Flg color is a tupple of the Flags variables *r* and *l*.

The control CPN is an executable net. It allows to verify the correctness of the control algorithm and to perform formal analysis, for example, by constructing an occurrence graph. Note that the Guards are placed inside the transition on the side of the input arc to hint that they form a Precondition. Also, the output arc inscriptions are placed inside the box to signal that they are, in fact, Postconditions. The verification is done by processing the tokens which get introduced to the FLAGS place during the execution of the environment CPN. For example, the sequence for the long train coming from the right is a product (r,l) with binary values {(t1,f0),(t1,t1),(f0,t1),(f0,f0)} which represent four distinct events corresponding to the waveform of the environment CPN of Figure 8.

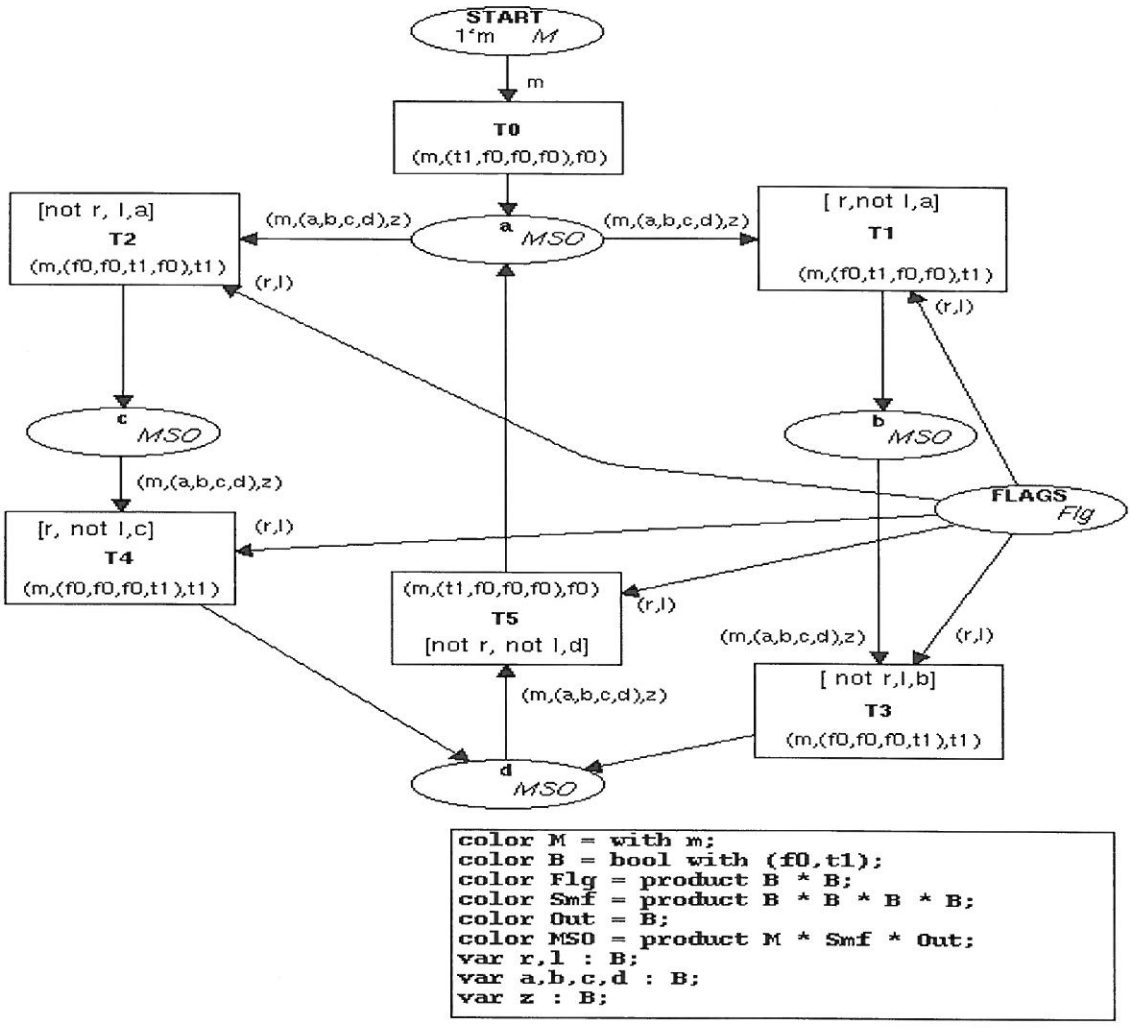



**Figure 9. Intermediate-Level CPN for the Railroad Crossing Problem.**

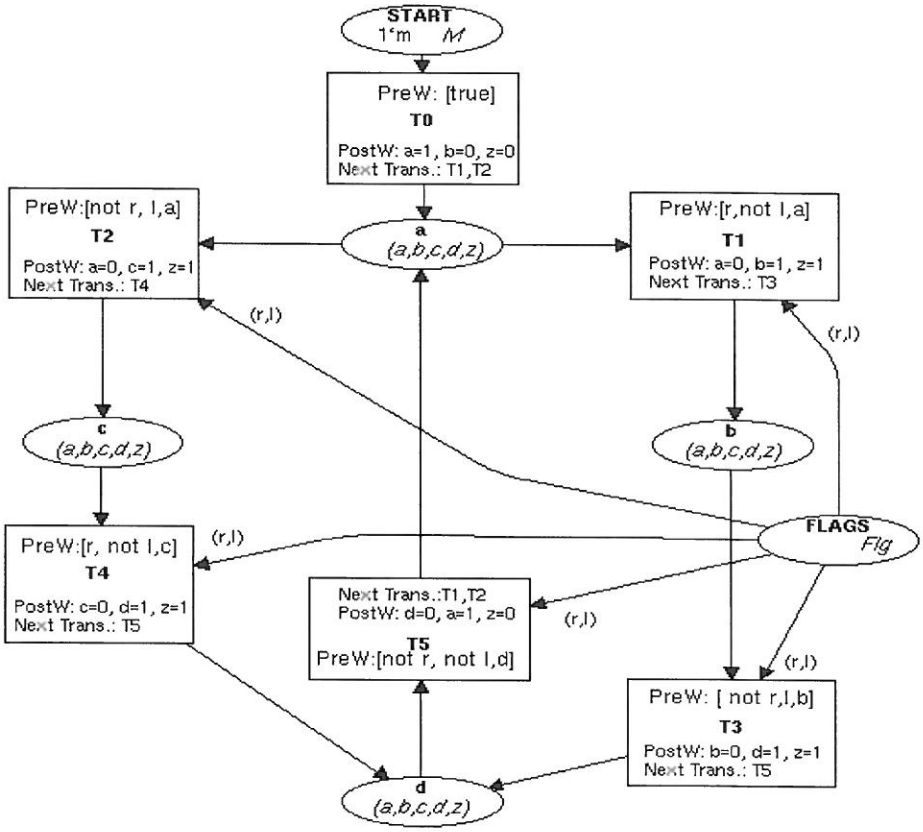



**Figure 10. Encoding of the CPN into the ALPiNe Format.**

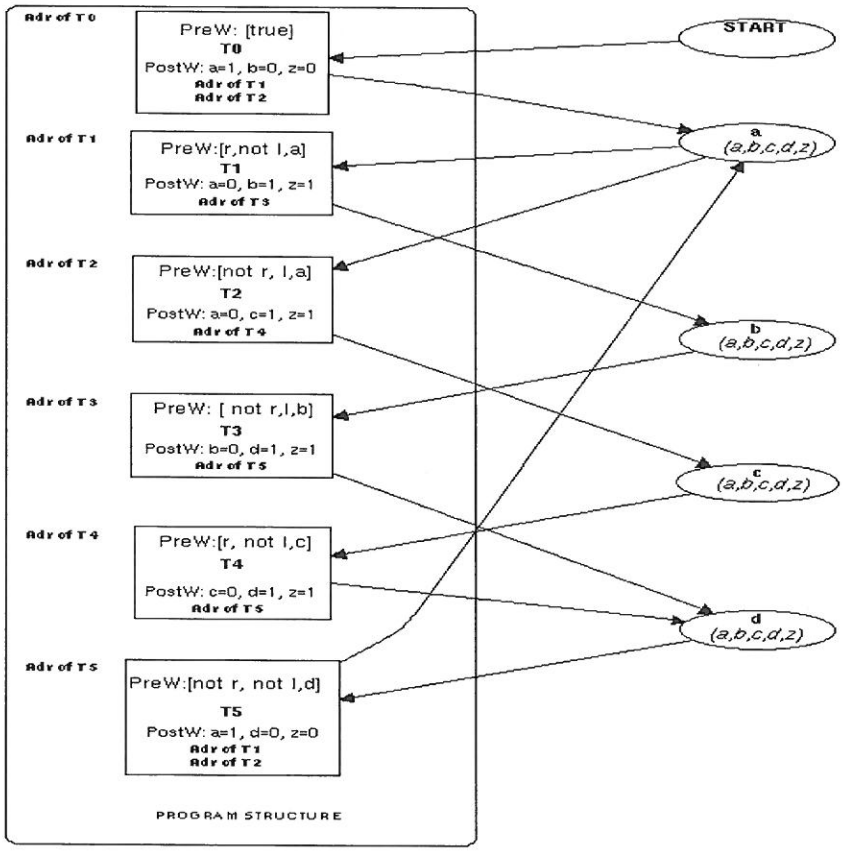



**Figure 11. Final Step of Encoding the CPN into the ALPiNe Format.**

Taking into account that presently ALPiNe is capable of testing only Boolean-AND operation of the Guard, the CP Net of Figure 7 could not be directly encoded into the ALPiNe machine code. This is observed from the Guard of T1, G = [(r not equal l) and a]. The Guard of T2 is also complex. Only the Guard of T3 is of the required Boolean-AND format. Thus, this CP net is "somewhat" high-level CPN. Enabling more complex Guards to be tested would solve this problem. However, at this point ALPiNe does not support complex Guards. Therefore, the CP net must be transformed into the intermediate-level CP net with more transitions to explicitly test complex Guards. This lower level CP net is shown in Figure 9, where Guards of each transition are of the required format. This CP net could be encoded into the ALPiNe machine code. The process of encoding is given by Figures 10 and 11.

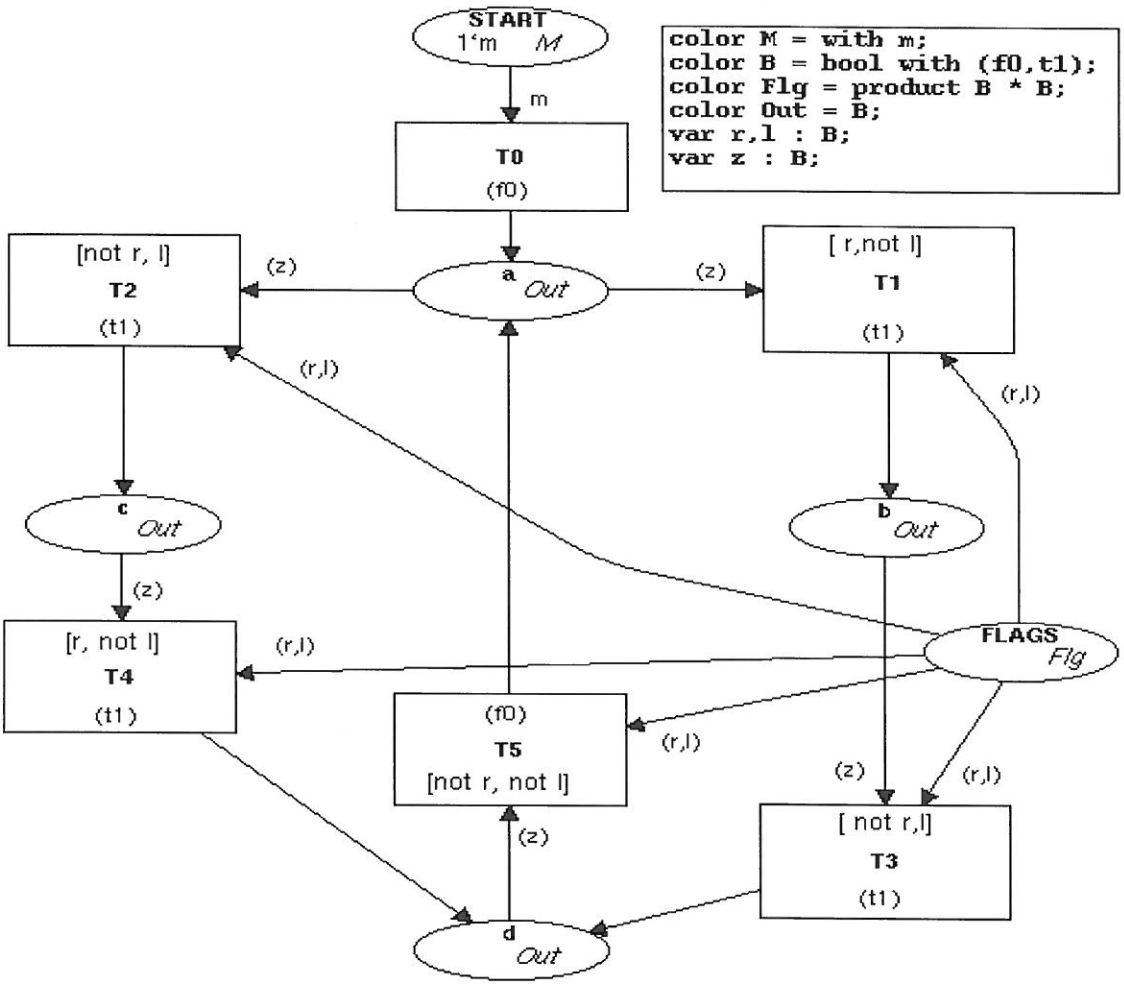



**Figure 12. Simplified Intermediate-Level CPN.**

The encoding proceeds as follows. Information provided by the Input Arc Inscription and the Guard is combined to form a Precondition Word, PreW. The Output Arc Inscription is converted into a Postcondition Word, PostW. In this example there is no subroutine associated with transitions, only the output variable is set or cleared. Subsequently, the corresponding addresses of the Next Transitions are appended to each transition. For example, the Next Transition List section of T0 will contain the addresses of T1 and T2. This process is shown in Figure 11. At this point, places and arcs lose their meaning as only transitions constitute the body of the program. This program can be executed on ALPiNe according to the algorithm described in Figure 5.

Careful analysis of the above coding scheme will show that encoding the Semaphore (current state) information is superfluous and could be omitted because for the correct solution of this control problem it is sufficient to test Flags tokens only. In the example presented the CPN of the environment does not need the information about the present state of the control algorithm. Therefore, the control CPN of Figure 9 could be simplified to produce a net with simpler arc inscriptions as shown in Figure 12. However, for some more complicated processes it may be crucial to explicitly know the state of the control. In such cases, the state could be encoded as already explained.

## Advantages of ALPiNe

ALPiNe is an interesting processor from both academic and commercial perspectives. First of all, it is a dedicated platform optimized for fast implementation of control algorithms developed using High-Level Petri Nets. It is a programmable device, and thus it is not limited to one specific application, rather it is suitable for the broad class of reactive systems. ALPiNe has two layers of hardware: one for deciding when and which computations should take place, and another one for performing those computations. This results in an efficient hardware utilization, and promises power efficiency depending on the final implementation.

One aspect not addressed at this time is the absence of interrupts. This would make it impossible to use ALPiNe in hard real-time applications, excluding some special cases. However, this issue will be addressed at the subsequent versions of the processor, with inclusion of a timer. This may be particularly beneficial for implementing Time-Extended CPNs.

## Implementation Strategy

ALPiNe processor described above is currently being implemented using the VHSIC Hardware Description Language (VHDL). The first version of ALPiNe has a global clock for both PNDU and CE modules. There are plans to experiment with an asynchronous (self-timed) realization of the PNDU, which naturally achieves the event-driven nature of operation, and a gated-clock CE. The ultimate goal is to have a complete self-timed version of ALPiNe and to analyze its performance against its synchronous counterpart. In addition, work is in progress on overcoming the FSM restriction and allowing more complex Guards.

## Conclusions

ALPiNe is a dedicated processor whose execution algorithm is based on the event-driven property of Petri Nets. Its unique architecture is geared to ease the development of complex reactive systems, such as embedded control systems, by providing special hardware to handle High-Level Petri Net algorithm and by incorporating external events into the program execution flow. This paper describes a detailed procedure for software development and presents an overview of the ALPiNe architecture. The successful completion of the ALPiNe project is expected to increase control system designer's awareness of the modelling power of Petri Nets and open new implementation horizons. Another anticipated positive side effect is the encouragement of new developments in CPN based software tools that would automate the transformation and compilation of the High-Level Petri Nets into the ALPiNe machine code.

**Acknowledgements**

The authors wish to thank Soren Christensen and Kurt Jensen for their helpful comments regarding this work. This work has been partially supported by the Natural Sciences and Engineering Research Council of Canada.

**References**

[AW91] W. M. P. van der Aalst, A. W. Waltmans, "Modelling Logistic Systems with EXPECT", In *Dynamic Modelling of Information Systems*, Eds. H. G. Sol and K. M. van Hee, Elsevier Science Publishers, Amsterdam, pp. 269-288, 1991.

[BM91] A. D. Ben, I. Miron, "Concurrent Modelling and Simulation of Reactive Manufacturing Systems Using Petri Nets", *Computers and Industrial Engineering*, Vol.20, No. 1, pp.45-54, 1991.

[CP96] C. Piguet, "Low-Power Design of Finite State Machines", in *Proceedings of PATMOS'96*, Eds. W. Nebel and B. Ricco, Pitagora Editrice Bologna, Italy, pp. 25-34, 1996.

[GD94] Giovanni De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill Inc., Chapter 11, 1994.

[HP85] D. Harel, A. Pnueli, "On the Development of Reactive Systems", in NATO ASI Series, Vol. 13, *Logics and Models of Concurrent Systems*, K.R. Apt, Editor, Springer-Verlag Berlin Heidelberg, pp. 447-498, 1985.

[HP90] J. H. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufman Publishers, 1990.

[JC97] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, A. Yakovlev, "Petrify: a Tool for Manipulating Concurrent Specifications and Synthesis of Asynchronous Controllers", *IEICE Transactions on Information and Systems*, E80-D(3):315-325, 1997.

[JP81] James L. Peterson, *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Chapter 3, 1981.

[KJ96] Kurt Jensen, *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use*, Volume 1, EATCS Series Monographs in Theoretical Computer Science, Springer-Verlag, Chapter 1&2 , 1996.

[MG95] Martin Gumm, *VHDL - Modelling and Synthesis of the DLXS RISC Processor*, University of Stuttgart, Institute of Parallel and Distributed High-Performance Systems, Dept. of Integrated Systems Engineering, December 1995.

[MP96] Olivier Maffeis, Axel Poigne, "Synchronous Automata for Reactive, Real-Time or Embedded Systems", Technical Report No. 967, German National Research Centre for Information Technology (GMD), January 1996.

[MS93] Kurt Jensen *et al*, *Design/CPN Reference Manual*, Meta Software and Computer Science Department, University of Aarhus, Denmark, 1993. On-line version: http://www.daimi.aau.dk/designCPN/.

[RS96] J. L. Rasmussen, M. Singh, "Designing a Security System by Means of Coloured Petri Nets", *Lecture Notes in Computer Science*, Vol. 1091, Springer-Verlag, pp. 400-419, 1996.

[SK97] A. Semenov, A. M. Koelmans, L. Lloyd, A. Yakovlev, "Designing an Asynchronous Processor Using Petri Nets", *IEEE Micro*, Vol. 17, No. 2, pp. 54-63, March/April 1997.

[TB82] P. C. Treleaven, D. R. Brownbridge, R. P. Hopkins, "Data-Driven and Demand-Driven Computer Architecture", in *Dataflow and Reduction Architectures*, Editor S. S. Thakkar, Computer Society Press of the IEEE, pp.4-54, 1987.

[TH92] Kenneth Hinz, Daniel Tabak, *Microcontrollers - Architecture, Implementation and Programming*, McGraw-Hill Inc., Chapter 3, 1992.

[TM89] Tadao Murata, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE*, 77(4), pp.541-580, 1989.

[TT97] J. Teich, L. Thiele, S. Sriram, M. Martin, "Performance Analysis and Optimization of Mixed Asynchronous Synchronous Systems", *IEEE Transactions on CAD of Integrated Circuits and Systems*, Vol. 16, No. 5, pp. 473-484, May 1997.

[WA94] W. M. P. van der Aalst, "Putting High-Level Petri Nets to Work in Industry", *Computers in Industry*, 25(1):45-54, 1994.

# Textual Interchange Format for High-level Petri Nets

Regnar Bang Lyngsø and Thomas Mailund

Computer Science Department
University of Aarhus
DK-8000 Aarhus C
Denmark
E-mail: {rblyngso,mailund}@daimi.aau.dk

**Abstract.** In this paper a text format for High-level Petri Net (HLPN) diagrams is presented.
The text format is designed to serve as a platform-independent file format for the Design/CPN tool. It is consistent with the forthcoming standard for High-level Petri Nets. The text format may also be seen as our contribution to the development of an open, tool-independent interchange format for High-level Petri nets.
The text format will make it possible to move *Design/CPN* diagrams between all supported hardware platforms and versions. It is also designed to be a bridge to other Petri Net tools, e.g., other analysis tools which the user may want to use with *Design/CPN* diagrams. The proposed text format does not address any standardisation for the inscription language used in the diagram. It is, however, possible to extend the format to incorporate such a standardisation.
The text format is designed for the exchange of *Hierarchical Coloured Petri Nets* but the structure is general enough to cope with other High-level Petri Nets as well.
The text format presented here has been implemented as part of *Design/CPN* version 3.1.

## 1 Introduction

*Design/CPN* is a widely used tool within the Petri Net community and has been developed for more than 10 years. The tool has been used in many projects in a broad range of application areas [8].

*Design/CPN* supports Hierarchical Coloured Petri Nets (CP-nets or CPNs) [7] with complex data types (colour sets) and complex data manipulations (arc expressions and guards) - both specified in the functional programming language CPN ML [3]. It also supports hierarchical CP-nets, i.e. net diagrams that consist of a set of separate modules (subnets) with well-defined interfaces.

### 1.1 The Reasons for Choosing a Text Format

Until now *Design/CPN* diagrams have been saved in a binary file format, but there are several reasons why we would prefer to use a text format. The cur-

rent format is a properitary file format which poorly supports the possibility of making diagrams distributed in multiple files. With the text format it should be possible to add some sort of modularisation system making modularisation of diagrams easier. This is not possible with the current file format of *Design/CPN*.

Furthermore the current file format of *Design/CPN* has certain limitations when it comes to moving a diagram between different hardware platforms of the tool. Specifically a user might not be able to save a diagram on one hardware platform (e.g., the Intel family) and load it on another (e.g., Macintosh). Another point is compatibility between different versions of *Design/CPN*. With the binary file format it is more difficult to add and remove features to the language than it is with a text format.

We also hope to make it easier to work independently on different parts of a diagram, and then later on put the pieces together. By adding merging/replacing facilities and some sort of version-control we hope to make *Design/CPN* more attractive for groups working on larger projects. The text format should also make it possible to create libraries of commonly used net structures.

Another desirable feature of a text format is readability, since this will ease debugging and inspection by users.

The advantages of having a widely accepted interchange format supported by different Petri Net tools are obvious and well known. With $n$ Petri net tools, each with its own diagram representation, it would be necessary to have $n^2 - n$ tools to translate between all pairs of languages. By using an intermediate language it would only be necessary with $2n$. Furthermore, *each tool* would only need *one* translator to and from the interchange format, instead of the $n - 1$ to be able to load and save each of the different formats.

To fulfil these requirements the text format must necessarily be expandable in such a way that it matches the capabilities of any format used by any Petri Net tool i.e. a loaded diagram must look and behave exactly as the one that was saved. The text format described here is a proposal for such a common format.

To sum up the goals of the text format, which would be either difficult or impossible to achieve by further development of the current binary file format:

- Modularisation (on file basis) allowing development of libraries.
- Hardware independency.
- Compatibility between different versions of the tool.
- Readable by humans.
- Ease of further development/enhancements of the tool.
- Easier communication between different tools.

### 1.2 Design Decisions

From the very beginning it was decided that the text format should be independent of the inscription language. This decision was made for numerous reasons. First of all we wanted it to be easy to implement the text format in an existing tool as we have done with *Design/CPN*.

The considerations mentioned earlier concerning different tools interchanging diagrams are of course true for the inscription language as well. The same benefits can be obtained by sharing an inscription language. We believe however that the inscription language is a far more integrated part of each tool than the graphical layout, thus translation to and from a common inscription language was considered beyond the scope of this text format.

On the other hand it was feasible to implement known standards for both naming conventions and syntax in order to make it easier for humans to read the description of the diagram. To that end we have chosen to use the terminology presented in the current version of the committee draft of the HLPN standard [1]. This means that the entities known in *Design/CPN* as arc expressions, colour regions and guards are called arc annotations, type region and transition conditions, respectively. Furthermore we chose to use the SGML (*Standard Generalized Markup Language*) ISO standard, which will be described in Sect. 3.

The structure of the text format is chosen to be based on the semantic properties rather than the graphical appearance. It is thus considered more important to know that a certain object is a place than it is to know that the object has the shape of an ellipse.

We want the text format to be both general enough to be used by various different tools and specific enough to save the same information about a diagram as the binary format used so far. Different tools need to add different kinds of information to the text format, and later versions of *Design/CPN* will probably add to the format as well.

On the other hand, each tool will need certain information, e.g. graphical attributes such as line thickness or fill pattern, which has no counterpart in some other tools. A simulator need no information about the graphical layout. Thus we cannot expect to get all the information needed by one tool from text format exported by another tool.

The solution to these seemingly conflicting goals, was found to be a very simple structure which could be easily parsed and subsequently weeded for unwanted information. Furthermore, it should be possible to describe a diagram with a minimum of extra information, leaving most of the graphical layout and some of the semantics to defaults. The default shape of a place is an ellipse, the default of a condition is true, etc. In our implementation, when loading a diagram, unknown parts are ignored and missing information is substituted by defaults.

We have implemented routines for loading and saving the text format in *Design/CPN*. The load routines, however, are almost tool independent, and it should be fairly easy to incorporate this module into other tools. For further detail we refer to [9].

## 2 An Example Diagram

In this section we will give an example of a small diagram described in the text format. A description of the text format will follow in the next section.

Figure 1 shows a small diagram with two places and one transition. Example 1 shows the text description of the same diagram. The text description is slightly edited from text generated by the tool.[1] It is followed by an explanation.

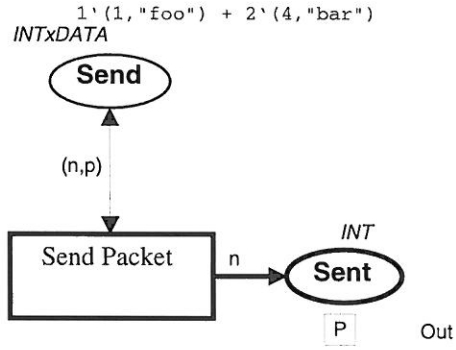



**Fig. 1.** A simple diagram

*Example 1 (Text description of Fig. 1).*

```
1    <!-- File /users/cpn/nets/Sender.int -->
2    <page id=id10>
3      <trans id=id34>
4        <text>Send Packet</text>
5        <textattr font=Times size=12 just=Centered colour=Lime>
6        <lineattr type=Solid thick=2 colour=Yellow>
7        <fillattr pattern=White colour=White>
8        <posattr x=-644 y=779>
9        <box h=134 w=321>
10     </trans>
11
12     <place id=id25>
13       <text>Send</text>
14       <textattr font=Helvetica size=12 just=Centered colour=red bold=TRUE>
15       <lineattr type=Solid thick=1 colour=red>
16       <fillattr pattern=White colour=red>
17       <posattr x=-644 y=1090>
18       <ellipse h=84 w=194>
19
20       <type id=id24>
```

[1] A few *Design/CPN* specific attributes have been removed.

```
21          <text>INTxDATA</text>
22          <textattr font=Helvetica size=9 just=Left colour=red italic=TRUE>
23          <posattr x=-722 y=1167>
24          <label>
25        </type>
26
27        <initmark id=id22>
28          <text>1`(1,"foo") + 2`(4,"bar")</text>
29          <textattr font=Courier size=9 just=Left colour=red>
30          <posattr x=-450 y=1200>
31          <label>
32        </initmark>
33      </place>
34
35      <place id=id7>
36        <text>Sent</text>
37        <textattr font=Helvetica size=12 just=Centered colour=navy bold=TRUE>
38        <port type=out>
39        <lineattr type=Solid thick=2 colour=navy>
40        <fillattr pattern=White colour=navy>
41        <posattr x=-275 y=779>
42        <ellipse h=84 w=184>
43
44        <type id=id11>
45          <text>INT</text>
46          <textattr font=Helvetica size=9 just=Left colour=navy italic=TRUE>
47          <posattr x=-260 y=851>
48          <label>
49        </type>
50
51        <portkreg id=id15>
52          <text>P</text>
53          <textattr font=Helvetica size=9 just=Centered colour=navy>
54          <lineattr type=Solid thick=0 colour=navy>
55          <fillattr pattern=None colour=navy>
56          <posattr x=-282 y=694>
57          <box h=49 w=49>
58
59          <portreg id=id14>
60            <text>Out</text>
61            <textattr font=Helvetica size=9 just=Left colour=navy>
62            <lineattr type=None thick=1 colour=navy>
63            <fillattr pattern=None colour=navy>
64            <posattr x=-128 y=694>
65            <box h=52 w=84>
66          </portreg>
67        </portkreg>
68      </place>
69
70      <arc id=id16 orientation=BOTHDIR>
```

```
71        <placeend idref=id25>
72        <transend idref=id34>
73        <lineattr type=Solid thick=0 colour=navy>
74        <fillattr pattern=Black colour=navy>
75        <seg-conn>
76
77        <annot id=id13>
78          <text>(n,p)</text>
79          <posattr x=-686.15289 y=957.00000>
80          <label>
81          <textattr font=Helvetica size=9 just=Left colour=navy>
82        </annot>
83      </arc>
84
85      <arc id=id19 orientation=TtoP>
86        <placeend idref=id7>
87        <transend idref=id34>
88        <lineattr type=Solid thick=2 colour=navy>
89        <fillattr pattern=Black colour=navy>
90        <seg-conn>
91
92        <annot id=id17>
93          <text>n</text>
94          <textattr font=Helvetica size=9 just=Left colour=navy>
95          <posattr x=-447 y=809>
96          <label>
97        </annot>
98      </arc>
99    </page>
100
```

Line 1 is a comment and has no semantic meaning. Line 2 identifies the start of the page. The page ends in line 99. Lines 3–10 represent the transition *Send Packet*. This is done by means of a `trans` block. The `id` in line 3 is a unique identifier which which it is possible to refer to the transition. Line 4 describes the contents of the text within the transition, i.e. the text "Send Packet". Line 5 describes the format of the text. The font name is Times while the font size is 12. The text is Centered and the colour is Lime. Lines 6–7 describes the borderline and the fill pattern within the transition. The borderline is solid, (i.e. a full-drawn line) with thickness 2 and colour Yellow. The fill pattern is White, which is a non-transparent pattern. The fill colour is White. Line 8 describes the position of the centre of the transition, while line 9 describes the shape of the transition, i.e. box shape together with the width and height of the transition.

Line 12–37 represent the place *Send*. This is done by means of a `place` block. Lines 12–18 play a similar role as lines 3–9 in the `trans` block, but the contents are of course different. It can, e.g., be seen that the text is set in Helvetica, that the text style is boldface (line 14), that the line thickness is 1 (line 15), and that the shape is ellipse instead of box (line 18). The colour of the place is red (lines

14, 15 and 16). It can also be seen that the place has the same $x$ coordinate as the transition which means that the two objects are vertically aligned.

Lines 20–25 describes the type (colour set) of the place. The type block is nested within the place block. This corresponds to the fact that in *Design/CPN* the type is declared in a region under the place object. The text in the type block is smaller (font size 9), left justified and italic (line 22). Line 24 indicates that the type region has the shape of a label (i.e. a box where the size is automatically determined by the text). The initial marking of the first place (lines 27–32) is described similar to the type. The text is however set in Courier font.

The second place of the diagram is described in lines 35–68. It is horizontally aligned to the transition, i.e. they have the same $y$ coordinate (line 9 and 41). It is furthermore shaped as an ellipse. This place is also an output port (line 38), hence it contains a `portkreg` block (port key region), which again contains a `portreg` block (port region). These are *Design/CPN* specific and hence will not be described in any detail.

Lines 70–98 describe the arcs of the diagram. The first arc (lines 70–83) is orientated in both directions (line 70) and connects the *Send* place and the *Send Packet* transition, i.e., the `placeend` (line 71) refers to id *id25* which was the id of the *Send* place (line 12), and the `transend` (line 72) refers to the id of the transition, i.e. *id34*. The shape of the arc is seg-conn, one of the different connector shapes in *Design/CPN* . The line thickness is 0, which is hairline. The annotation (arc inscription) of the arc is found in the `annot` block, lines 77–82.

The second arc (lines 85–98) is very much like the first. There are, however, a few differences. It connects the transition and the *Sent* place. The orientation is from the transition to the place (line 85) and the line thickness is 2 instead of 0 (lines 88 and 73).

## 3 Description of the Text Format

The format is based on SGML *(Standard Generalized Markup Language)* [6] for several reasons:

SGML

- is designed specifically to support text interchange,
- has a rigorously described structure which makes it easily understood by both computers and humans,
- represents hierarchies,
- sets a standard for communication between different platforms, versions and tools,
- is a well documented, wide spread and accepted standard,
- is the framework upon which numerous well known markup languages are founded, for instance HTML.

### 3.1 Short Introduction to SGML

In SGML the basic element is the *document element*. It is the root of a tree of elements which as a whole makes up the document's *content*.

As stated in Annex B of the SGML ISO standard [6] a cornerstone of a SGML is the Document Type Definition (DTD for short). The following is a light version of the full DTD which describes the structure of the text format. It will be followed by an in-depth explanation[2].

```
1   <!DOCTYPE cpn [
2
3   <!-- "Macros" -->
4   <!ENTITY % colours              "(Black | Maroon | Gray | Olive |
5                                   Purple | Silver | Red | Teal |
6                                   Navy | Fucia | Blue | Aqua |
7                                   Green | Lime | Yellow | White)">
8   <!ENTITY % arcdirs              "(BOTHDIR | NODIR | PtoT | TtoP">
9   <!ENTITY % linetypes            "(Solid | Dash | None)">
10  <!ENTITY % fillpatterns         "(Black | None | White)">
11  <!ENTITY % boolean              "(TRUE | FALSE)">
12  <!ENTITY % fonts                "(Helvetica | Times | Courier)">
13  <!ENTITY % textjust             "(Centered | Left | Right)">
14  <!ENTITY % porttypes            "(in | out | inout | general)">
15
16  <!-- The basic elements of a net -->
17  <!--       ELEMENTS      MIN             CONTENT                    >
18  <!ELEMENT cpn            0 0             page*>
19  <!ELEMENT page           - -             (arc | place | trans)*>
20  <!ELEMENT arc            - -             (placeend? &  transend? & lineattr?
21                                           & fillattr? & seg-conn? & annot?)>
22  <!ELEMENT place          - -             (text? & type? & initmark?
23                                           & portkreg? & port?)>
24  <!ELEMENT trans          - -             (text? & textattr? & lineattr? &
25                                           fillattr? & posattr? & box?)>
26
27  <!--       ELEMENTS      NAME            VALUE           DEFAULT>
28  <!ATTLIST page           id              ID              #IMPLIED>
29  <!ATTLIST arc            id              ID              #IMPLIED
30                           orientation     %arcdirs;       #IMPLIED
31  >
32  <!ATTLIST place          id              ID              #IMPLIED>
33  <!ATTLIST trans          id              ID              #IMPLIED>
34
35  <!--    Identifiers related to the look of an object and the
36          corresponding attributes -->
37  <!--       ELEMENTS      MIN             CONTENT                    >
38  <!ELEMENT posattr        - 0             EMPTY>
```

[2] The full DTD for the *Textual Interchange Format* can be found in the *Textual Interchange Format for High-level Petri Nets - Developers Guide* [9].

```
39  <!ELEMENT lineattr      - O             EMPTY>
40  <!ELEMENT fillattr      - O             EMPTY>
41  <!ELEMENT textattr      - O             EMPTY>
42
43  <!--      ELEMENTS    NAME              VALUE           DEFAULT>
44  <!ATTLIST posattr       x               CDATA           #IMPLIED
45                          y               CDATA           #IMPLIED
46  >
47  <!ATTLIST lineattr      type            %linetypes;
48                          thick           CDATA
49                          colour          %colours;
50  >
51  <!ATTLIST fillattr      pattern         %fillpatterns;
52                          colour          %colours;
53  >
54  <!ATTLIST textattr      font            %fonts;         #IMPLIED
55                          size            NUMBER          #IMPLIED
56                          just            %textjust;      #IMPLIED
57                          bold            %boolean;       #IMPLIED
58                          italic          %boolean;       #IMPLIED
59                          underline       %boolean;       #IMPLIED
60                          outline         %boolean;       #IMPLIED
61                          shadow          %boolean;       #IMPLIED
62                          condense        %boolean;       #IMPLIED
63                          extend          %boolean;       #IMPLIED
64                          colour          %colours;       #IMPLIED
65  >
66
67
68  <!-- Identifers related to the shape of an object -->
69  <!--      ELEMENTS      MIN             CONTENT                 >
70  <!ELEMENT label         - O             EMPTY>
71  <!ELEMENT seg-conn      - O             EMPTY>
72  <!ELEMENT box           - O             EMPTY>
73  <!ELEMENT ellipse       - O             EMPTY>
74
75  <!--      ELEMENTS      NAME            VALUE           DEFAULT>
76  <!ATTLIST box           x               PCDATA          #IMPLIED
77                          y               PCDATA          #IMPLIED
78  >
79  <!ATTLIST ellipse       x               PCDATA          #IMPLIED
80                          y               PCDATA          #IMPLIED
81  >
82
83  <!-- Identifiers related to text -->
84  <!--      ELEMENTS      MIN             CONTENT                 >
85  <!ELEMENT text          - -             CDATA>
86
87  <!-- Identifiers related to arcs -->
88  <!--      ELEMENTS      MIN             CONTENT                 >
```

```
89  <!ELEMENT placeend      - O             EMPTY>
90  <!ELEMENT transend      - O             EMPTY>
91  <!ELEMENT annot         - -             (text? & textattr? & posattr?
92                                          & label?)>
93
94  <!--      ELEMENTS      NAME            VALUE           DEFAULT>
95  <!ATTLIST placeend      idref           IDREF           #IMPLIED>
96  <!ATTLIST transend      idref           IDREF           #IMPLIED>
97  <!ATTLIST annot         id              ID              #IMPLIED>
98
99  <!-- Identifiers related to places>
100 <!--      ELEMENTS      MIN             CONTENT                   >
101 <!ELEMENT type          - -             (text? & textattr? & posattr?
102                                         & label?)>
103 <!ELEMENT initmark       - -             (text? & textattr? & posattr?
104                                         & label?)>
105 <!ELEMENT portkreg       - -             (text? & textattr? & lineattr?
106                                         & fillattr? & posattr? & box?
107                                         & portreg?)>
108 <!ELEMENT portreg        - -             (text? & textattr? & lineattr?
109                                         & fillattr? & posattr? & box?)>
110 <!ELEMENT port           - -             EMPTY>
111
112 <!--      ELEMENTS    NAME            VALUE           DEFAULT>
113 <!ATTLIST type          id              ID              #IMPLIED>
114 <!ATTLIST initmark      id              ID              #IMPLIED>
115 <!ATTLIST portkreg      id              ID              #IMPLIED>
116 <!ATTLIST portreg       id              ID              #IMPLIED>
117 <!ATTLIST port          type            %porttype;      #IMPLIED>
118 ]>
```

In line 1 of the DTD the type of the document is defined to be cpn. On the lines 4–14 a number of "macros" are defined. This means, for example, that the macro boolean in line 11 is expanded to (TRUE | FALSE) in the lines 57–64. Note that a macro substitution is invoked by prepending the macro name with '%' and appending ';' to it.

In line 18 the Generic Identifier (GI) cpn is defined. Since this GI equals the doctype in line 1 an element of this type is the root of the document (file). It is called a root because a SGML document represents a hierarchical structure. By defining the cpn GI, two so called *tags* are defined. The first one is a start tag which in a document is <cpn>. The second one is </cpn> which is an end tag. These two tags indicate the beginning and the end of the whole document. Generally a piece of text delimited by a start tag and an end tag is called a block. Since nothing but comments can come before the <cpn> tag it might be omitted. Similarly, the </cpn> tag might be omitted since nothing follows it. This is indicated by the two Os just below the MIN in line 17. The Os mean "omit". The first O indicates that the start-tag might be omitted. The second one that the end-tag might be omitted. By writing page* in the content field

on the end of line 18 we define that a cpn-block might contain zero or more elements of type page. The '*'-operator means "zero or more".

In example 1 the <cpn> and </cpn> tags are omitted. In line 2 of example 1 a page block starts. This is legal since, as we just saw, page is a valid subelement GI of cpn. In line 19 of the DTD the GI page is defined. Thereby we define that the beginning of a page block is identified by the start tag <page>. The end of a page block is identified by the corresponding end tag </page>. The two dashes in line 19 indicate that the start and end tags can't be omitted. The content field at the end of line 19 of the DTD indicates that a page element consists of zero or more elements of type arc, place or trans. The '|'-operator means "or". In example 1 the page block contains a trans block (in lines 3–10), two place blocks (in lines 12–33 and lines 35–68), and two arcs (in lines 70–83 and lines 85–98). This is legal according to the definition in line 19 of the DTD. Notice that it is perfectly legal to have e.g. an arc block between two trans blocks. In line 2 of example 1 the start tag of the page block also contains the text id=id10. This text identifies an attribute. Attributes are only allowed in the start tag. In this case the attribute name is id and the attribute value is id1. It is defined in line 28 of the DTD that a page might have an attribute named id. Moreover the value range of this attribute is defined to be ID. Line 29 defines that the arc GI also can have an attribute whose value range is ID (and whose name is also id). By defining both value ranges to be ID we express that a page and an arc can't have same id value.[3] The id attribute is a legal attribute of all GIs representing objects. Thus, in the text format the value of each id attribute must be unique.

ID is a SGML reserved keyword. This uniqueness property of ID is defined by the SGML standard. Thus an object has a unique identification within the document. At the end of line 28 the default value of the attribute is defined to be #IMPLIED. IMPLIED is also a SGML keyword. That the default is IMPLIED means that it is implementation dependent what to do when processing a <page> tag in which the attribute id does not appear. In our implementation the page will be created.

The arc GI declaration in line 29 of the DTD introduces some new operators for the content declaration. The '&'-operator means that all of the operands must appear. The order in which the operands appear is however not determined. The '?'-operator means zero or one occurrence. This means that an arc block can contain at most one of each of the following blocks: placeend, transend, lineattr, fillattr, seg-conn, and annot. The six blocks are not restricted to appear in the listed order.

The placeend GI is defined in line 89 of the DTD. The content field contains the SGML keyword EMPTY. This means that a placeend has no valid subelement GIs. Hence there is no need for an and tag. In fact an end tag is disallowed in this case. The O indicating that the end tag can be omitted is just a reminder. The placeend GI has the attribute named idref defined. It is used in example 1 in which the value of the idref attribute in line 86 refers to the

[3] Even if the attributes had different names, say pageid and arcid, the common value range ID would insure uniqueness

`id` attribute value declared in the `place` start tag in line 35. The value range of this attribute is IDREF. IDREF is a SGML keyword by which the attribute value range is defined to be a reference to an attribute value of the type ID. The attribute value referenced must be declared within the document.

In line 44 of the DTD the attributes of the GI `posattr` are defined. The value range of both attributes is CDATA. This means that the attributes `x` and `y` can assume arbitrary character values, which does not conflict with the markup.

In the definition of the `text` GI in line 85 the content is defined to be CDATA. This means that everything within a `text` block is processed and no markup within the `text` is processed. Hence the sequence `<place></place>` is illegal inside a `trans` block whereas it is legal in a `text`. In the `text` block the sequence is not identified as a block but just as a stream of characters.

The rest of the definitions in this light version of the real DTD make use of the constructs discussed above.

For a thorough description of SGML confer with the ISO 8879 standard [6] or [5] which includes an annotated version of the ISO 8879 standard.

### 3.2 The Text Format as Description of a Petri Net Diagram

We have now presented and explained how the text format might represent a Petri Net diagram. We have also presented the formalism defining the structure of the text format. We now focus on *how* the structure defined by the DTD corresponds to the structure of a Petri Net diagram. We divide the GIs in two categories: the ones where content is allowed, and the ones where content is not allowed, e.g. the content is defined EMPTY. By using GIs of the first type it is possible to make *blocks*. The GIs with empty content do not define blocks.

**The Blocks.** In the DTD we defined some GIs with non-empty content, e.g. the definition of `page` in line 19. These GIs with non-empty content corresponds to *blocks* in example 1, e.g. the `page` block in lines 2–99 of the example. We have defined a block type called `ignore`. It is used for comments. These comments may extend over several lines. It may encompass other blocks (and other comments). Hence it can be used to ignore a large contiguous part of a document. The rest of the blocks represent five different types of information about the CP-net.

- Actual objects of the CP-net (pages, transitions, places, etc.).
- The text within an object.
- Additional information about a place, transition or arc (place type, transition condition, arc annotation, etc.).
- The text which is part of the additional information.
- Different kinds of defaults.

The first type is represented in example 1 where lines 3–10 represents the transition of Fig. 1. The `text` block in line 4 of example 1 is of the second type. It represents the text within the transition on Fig. 1. An example of the third type is the `initmark` block in lines 27–32. This block represents the initial marking of the

*Send* transition on Fig. 1. In *Design/CPN* the initial marking is a region of the place. The text format does not assume that this is the case. If the `id` attribute in line 27 was omitted along with the lines 29–31, the `initmark` block would still be valid according to the DTD. If *Design/CPN* parses a block where the attributes are missing it just creates an initial marking region according to the defaults. The `id` is a valid attribute of the `initmark` GI because in *Design/CPN* an initial marking *is* represented by a graphical object. If a connector was created between the initial marking of the *Send* place in Fig. 1 and one of the arc annotation regions we would need to reference the corresponding blocks in the text format in order to represent the connector. The arc between the transition and the *Send* place on Fig. 1 is represented in lines 70–83 in example 1. To identify which place/transition pair is connected by the represented arc, line 71–72 refers to the ids introduced by the representation of the place and the transition. A connector connecting two object is represented in a similar way.

It is even possible to leave out the `text` block in line 28 of example 1. That block is of the third type according to the block type partitioning above. If it is left out the initial marking is implementation dependent. In *Design/CPN* an initial marking region without any contents is created if such a `initmark` block is parsed

**The GIs with no Valid Subelement GIs.** A GI which have no valid subelement GIs represents a (named) set of attributes. The attribute sets are primarily used to represent graphical attributes in diagrams. In line 5 of example 1 we have a set of four attributes:

```
<textattr font=Times size=12 just=Centered colour=Lime>
```

In lines 71 and 72:

```
<placeend idref=id25>
<transend idref=id34>
```

we see examples of attribute sets which have only a single attribute each.

In line 31 we have the empty attribute set `<label>`. This set represents the object represented by the enclosing block has box shape and that the size of that object depends entirely on the text within the object.

**Representation of Graphical Layout** One of the strengths of the Petri Nets is the graphical representation, which is intuitive and easy to understand. To keep this advantage it is essential that the nets are *drawn* in a readable fashion. Much care must be put into creating the graphical layout, and this work we do not want to lose. Thus we need to be able to represent the graphics in the text format.

First of all we need to be able to specify positions. We use a coordinate system with the x-coordinate running from left to right and the y-coordinate bottom-up. The centre of the diagram is at coordinate $(0, 0)$, and the measurement unit

used is millimetres and fractions of millimetres. It is not possible to express this requirement in the DTD. In other words, the text format is not described entirely by the DTD.

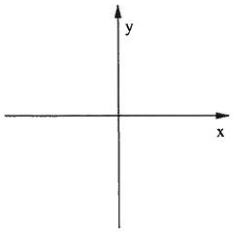



**Fig. 2.** The coordinates

The position of a object represented by a block is specified in the position attribute set `posattr`. The transition represented in the example is positioned at coordinate $(-644, 779)$ according to line 8 of example 1.

The text format does not make any demands on the shape of net objects. It is thus possible to specify any shape, which a given tool supports, as an attribute set of a block, to keep this information. The transition in the example is box shaped, and the first place has shape ellipse. Line thickness, fill pattern and colours are important. Thus this information is a part of the text format. The first transition in the example has a solid line, yellow coloured, and with line thickness 2pt, as seen in line 6.

## 4 Benchmarks

When considering a change of the file format used within *Design/CPN* , it is essential that performance is not jeopardised with respect to size and especially speed. To investigate this we have compared the time and space usage when loading and saving two "typical" CP-nets. No systematical benchmarking has been made yet.

| Diagram | | Binary | Textual | Ratio |
|---|---|---|---|---|
| **Size** | uncompressed | 161 KB | 165 KB | 1.02 |
| | compressed | 35 KB | 16 KB | 0.46 |
| **Time** | save | 1 sec | 2 secs | 2.00 |
| | load | 1 sec | 5 secs | 5.00 |

**Fig. 3.** 7 pages, 22 places, 12 transitions

| Diagram | | Binary | Textual | Ratio |
|---|---|---|---|---|
| **Size** | uncompressed | 1371 KB | 2221 KB | 1.61 |
| | compressed | 302 KB | 137 KB | 0.43 |
| **Time** | save | 3 secs | 39 secs | 13.00 |
| | load | 13 secs | 73 secs | 5.62 |

**Fig. 4.** 33 pages, 237 places, 109 transitions

As seen in Figs. 3 and 4, the increase in size of the files is not significant, even though the size increases as much as 61%. This is not important, when considering the disk space available on an average workstation. This is especially true since the size of a diagram saved in the text format can be compressed to less than what is possible when using the binary format. This is done by using commonly known compression programs.[4]

When it comes to speed, the text format does not match the binary format. This comes as no surprise since it is possible to do a "memory dump" when using the binary format, whereas the diagram is traversed object-by-object when saved and loaded. Experiments indicate that the time used when saving a diagram in the textual format is linear in the number of objects. If modularisation and/or version control is integrated with the text format a substantial decrease in save times is expected. In that case it is only necessary to save the changed parts of a diagram. The same decrease is expected when editing diagrams. The whole diagram is only needed for simulation.

## 5 Future Work

The text format presented here makes it possible to save and load (parts of) diagrams within *Design/CPN*. When modelling complex systems two additional features would be feasible: version control and a modularisation system.

### 5.1 Version Control

The text format opens up for libraries of common net structures and distribution of nets over several files, and thus makes it easier for groups to work on different parts of larger projects. Large projects, however, often demand some sort of version control of files.

Common version control systems, e.g. CVS [2], works well on text but, perhaps surprisingly enough, less than perfect on the text format proposed here. Writing diagrams in a text editor will of course work as well as any source code version control, but when editing the diagrams in HLPN tools, as will often be the case, two major problems arise: we have no guaranty of consistency in identifiers, and different tools might export the objects in a different order.

[4] The examples in this paper were all compressed using `gzip` [4].

Tools will have different ways of identifying objects internally which might reflect on the exported identifiers. Editing one file in the project can thus change the identifiers in that file making it inconsistent with the other files. This can of course be solved by simply demanding that blocks should be saved with the same identifier as they were loaded, forcing the tools to remember these "external" identifiers.

Merging two files is easily done with CVS, but different ordering of the blocks will result in conflicts in most of the file, nullifying the advantages of being able to edit separate copies. A solution could be restricting the order of blocks somehow. A better approach could be to add version control to the text format. With a fine-grained version control system as part of the text format it would be possible for different people to edit different parts of a diagram, perhaps in different tools, and then to merge the changes together, fully automatically. This could be done by adding unique time stamps or version numbers to each block. Merging two files would then simply be a matter of comparing these numbers. Whenever the version numbers differ the user could be prompted to take action, or the newest version would simply be selected.

Such a system could be the next step in making *Design/CPN* more attractive to large project groups.

### 5.2 Modularisation Management System

For the time being, the only way of dividing a diagram in modules is by saving different modules (subpages) in different files and later loading these separately. This is far from optimal. It would be better if it was possible to let the text format manage modularisation of a diagram. It should be possible to specify some sort of modularisation of the diagram, such that modules can be edited independently *without* having to combine all modules manually afterwards.

One solution is to tag substitution transitions as "modules" and to specify a file name for each of these. When saving the diagram, the subdiagram with the tagged substitution transition as root should be saved in a different file, with the specified file name, and only the file name and the substitution transition should be kept in the original diagram. If the tool discovers a "module" tagged transition during load, it simply loads the subdiagram from the file. Although not essential, this is definitely worth considering.

## 6 Conclusions

What we have presented here is not just a file format for *Design/CPN* but a framework for representing *High-level Petri Nets*. The benefits of this is that the format is easily changed/enhanced without losing the opportunity to develop a given diagram further in another tool (or another version of a given tool). Furthermore the format makes it possible to split diagrams into different files. This makes it easier for groups to work on the same diagram and also makes it possible to have standard libraries which can be used in different diagrams. The

drawbacks of the format are mainly related to the speed for saving and loading a diagram. With current technology these penalties are not considered severe and are believed to become less significant within a short time frame due to hardware development.

We believe that SGML provides a robust framework for representing *High-Level Petri Nets*. The format presented here has proved that it is indeed possible to use SGML to represent *High-level Petri Nets*. The text format will evolve due to future development of *Design/CPN* and hopefully due to input from the Petri Net community.

## 7 Acknowledgements

We would like to thank the CPN group at University of Aarhus for their feedback to both the text format and this paper. A special thank to Kurt Jensen for his help in structuring this paper.

## References

1. Jonathan Billington et al. High-Level Petri Nets – Concepts, Definitions and Graphical Notation. Committee Draft 15909, International Standards Organization and International Electrotechnical Committee, oct 1997. ISO/IEC JTC1/SC7/WG11 N-3.
2. Per Cederqvist. *Version Management with CVS.* Signum Support AB, Box 2044, S-580 02 Linkoping, Sweden, November 1993.
3. CPN ML: Relation with Standard ML. <URL:http://www.daimi.aau.dk/~desgncpn/sml/cpnml.html>.
4. L. Peter Deutsch. RFC 1952: GZIP file format specification version 4.3, May 1996. Status: INFORMATIONAL.
5. Charles F. Goldfarb and Yuri Rubinsky. *The SGML handbook.* Clarendon Press, Oxford, UK, 1990.
6. Information processing: text and office systems: Standard generalized markup language (SGML). ISO standard 8879, International Organization for Standardization, Genève, Switzerland, 1986. Amendment from 1988 exists.
7. Kurt Jensen. *Coloured Petri nets - Basic concepts, Analysis Methods and Practical Use. - Volume 1: Basic Concepts.* Monographs in theoretical computer science, 234 pages, Springer-Verlag, Berlin, 1992.
8. Kurt Jensen. *Coloured Petri Nets - Basic concepts, Analysis Methods and Practical Use. - Volume 3: Practical use.* Monographs in theoretical computer science, 265 pages, Springer-Verlag, Berlin, 1997.
9. Regnar Bang Lyngsø and Thomas Mailund. *Textual Interchange Format for High-level Petri Nets - Developers Guide*, 1998.

# SNIFF: An Input/Output Library for Design/CPN

Christoph Maier * Daniel Moldt and Heiko Rölke †

**Abstract**

In our group several projects used and use Coloured Petri nets as their main technique for modeling. Design/CPN has been one of the tools to edit and simulate the Petri net diagrams. However, Design/CPN has to be considered as a closed tool. To enlarge its funcitonality the desire is to open the tool for its environment. This is achieved by a library of functions which are implemented using Design/ML and Mimic allowing selective import and export of single or all pages of a Petri net diagram. Its input/output format is a human-readable text file. The text has a well defined syntax and maintains the nets structure and its content. Hence the interface is public other tools can use the diagrams produced by Design/CPN and Design/CPN can use the output of other tools.

## 1 Introduction

Today's software development needs special and powerful tools which support the underlying concepts and techniques. In the area of specification and prototyping (Coloured) Petri nets are a well known formalism for which the tool Design/CPN [Design/CPN 1993b] has specifically been developed, besides some others. Building such tools is a very expensive task as can be seen in all parts of computer science. Therefore existing tools are adopted or extended to fit the specific needs within projects.

In Hamburg one of the main techniques used within projects were Coloured Petri Nets [Jensen 1992]. Especially in the context of prototyping projects [Beckmann 1993] as well as object-oriented Petri net projects [Becker und Moldt 1993; Maier 1996] the need for tool support arose. The tool Design/CPN has been used since its first distribution. However, especially the introduction of new concepts and structuring rules for Coloured Petri nets diagrams required to use an open tool. Some former projects in Hamburg showed that the required man-power to build an own special version of a Petri net tool would have been expensive with respect to time and resources. Therefore Design/CPN has been extended

*Forschungsinstitut für Angewandte Software-Technologie (FAST) e.V., Arabellastr. 17, D-81925 Muenchen, cma@fast.de

†Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Theoretische Grundlagen der Informatik, Vogt-Kölln-Str. 30, D-22527 Hamburg, {moldt,3roelke}@informatik.uni-hamburg.de

by some functions which are now bundled in the SNIFF library[1]. SNIFF allows to connect Design/CPN to the output of other tools, e.g. diagrams which contain Coloured Petri nets transformed from class diagrams (for class diagrams see e.g. [Fowler und Scott 1997]). Also some extensions were necessary due to some mistakes, gaps, or design flaws within Design/CPN. The following list provides some functionality requirements for Design/CPN which influenced our design decisions for the SNIFF architecture.

- Within the prototyping projects many cycles of editing and simulating the nets occurred. During that process the internal representation got somehow corrupted without being recognized by the modelers. In the later process this turned out to be a reason for crashing of the program. Due to the unreadable file format the large nets had to be rebuilt by hand. Therefore a reasonable backup facility was required.

- The binary file format of Design/CPN did not allow to modify the net diagrams with other tools, e.g. changing some values for whole netsusing emacs for textual replacement of regualar expressions.

- The import and export of the whole net or some parts were very limited. Desirable features are to select pages or to blind out attributes of diagrams, e.g. graphical attributes for an external analysis tool.
  To load other diagrams only loading of IDEF diagrams was available. However, in some projects it was of interest to load nets which were generated by other tools, e.g. object-oriented Coloured Petri nets as defined in [Maier 1996; Moldt 1996].

- A library of Design/CPN diagrams could not be build due to the insufficient import/export functionality. Independent development of parts of a Petri net e.g. by different people was not supported. Models build separately could not be merged, except for the restricted use of loading and saving one single diagram page. This implied many difficulties for teamwork.

- There was no pretty printer for the net diagrams. Modelers have to ensure themselves the use of the same attributes for the same kind of element within one Design/CPN diagram.

- Porting diagrams between different platforms (Machintosh and Sun) was insufficient.

- For some diagrams the concepts of place refinement was of interest, even when these diagrams should not be simulated, the automatic consistency was desired. Unfortunately only the versions before Design/CPN version 2 supported this. Diagrams already drawn using version 2.0 had to be redrawn by hand. Therefore a support for different versions was desirable. The same is true when moving from version 3.xx to version 2.xx.

[1] SNIFF stands for "'Structured Net Interchange File Format"'

First of all SNIFF is the name for the library given here. At the same time it is the name of a project which runs now for several years besides our main activities. The phrase SNIFF is for obvious reasons strongly related to the SNIFF grammar and the SNIFF textual format.

### Design/CPN

Design/CPN integrates three kinds of Petri net tools: editor, simulator, and analyzer (see [WWW-Design/CPN 1998b]): "The CPN Editor supports construction, modification and syntax check of CPN models. The CPN Simulator supports interactive and automatic simulation of CPN models. The Occurrence Graph Tool supports construction and analysis of occurrence graphs for CPN models (also known as state spaces or reachability graphs/trees). All three parts can be used for large hierarchical CP-nets - with or without time."

#### Functionality

A Petri net is called a diagram within the context of Design/CPN. A diagram is the combination of all net components that can be accessed at the same time. The diagram is divided into single pages, each shown in a window of its own. The pages can be connected by substitution transitions and fusion places. Main elements of the net structure are places, transitions and arcs. In Design/CPN diagrams (and therefore in the SNIFF grammar) all these elements can be further specified by regions and auxiliary regions: places need a colour region, transitions may have a guard region or a code region, arcs should have an arc expression etc. Hierarchy information for the already mentioned substitutions and fusions are located in such regions, too, as well as simulation specific and other information. For a deeper introduction to Design/CPN see [Design/CPN 1993b] or [WWW-Design/CPN 1998a].

#### Design/ML

Design/CPN is based upon the functional programming language ML (see [Design/CPN 1993a; Harper 1986; Paulson 1991]). ML is not only used in code regions of more complex transitions, but it is also possible to execute any program without leaving the tool. The language itself is extended mainly by colors of places and variables to satisfy special Petri net needs. Furthermore an extensive set of functions is provided. These functions have been used intensively for SNIFF.

#### Mimic

Mimic is an extension to Design/CPN which allows to build simple graphical user interfaces using only functions from the built in Design/ML library. The mouse is supported for user interaction. The functions can be used within the edit-mode of Design/CPN. Mainly the selection possibilities for the mouse device are used within SNIFF. For more details about Mimic see [Rasmussen und Singh 1995] and [WWW-Mimic 1998].

After this introduction section 2 explains basic ideas, architecture, functionality, grammar, and examples files of SNIFF and closes with a discussion of gained experience. The general conclusion and outlook is followed by SNIFF's grammar and an example in the appendices.

# 2 Textual Output for Design/CPN: SNIFF

To open Design/CPN to its environment, general import and export functions have to be implemented. These can be used in different contexts. Therefore the provided library can be used to backup, to overcome the problem of corrupted diagrams, to import from other tools, to export to other tools, to build Coloured Petri net diagram libraries, to pretty print Coloured Petri net diagrams, e.g. according to project standards, and to use selective filters. As the conceptual basis the Structured Net Interchange File Format (SNIFF) is used.

## 2.1 Basic Idea

Hence the internal representation of the diagrams is hidden, the Design/ML functions are used to access the internal representation to build an SNIFF internal representation which is then exported in ASCII format using a reasonable, human-readable notation. Generally the Design/CPN tool becomes an open tool, hence the exported ASCII text diagrams have a public interface, the SNIFF grammar. Once we have such an internal SNIFF representation and the exported diagrams, we can manipulate the Coloured Petri net diagrams to implement the functional requirements mentioned above.

The SNIFF text diagram representation[2] can be written to a text file, reread from a file and converted again to a Design/CPN diagram. It is important to notice that this includes the hierarchical structure of Design/CPN diagrams, like substituion transition or fusion place relations in the same way as all other regions. Separate im- and export functions are available which can be used independently, e.g. to filter diagram output. To allow for flexibility, parameter files are used for filter and default options. The non-terminals in the grammar are easily changeable keywords, allowing to adopt the output file, e.g. to new evolving international standards.

The textual description of a net allows the access to a net by external programs. These programs may further process an existing net, but also generate a new net description which can be converted to a Design/CPN net. This can be graphically edited and simulated as usual.

## 2.2 Architecture of SNIFF

SNIFF is realised as a function library. There are four major function blocks. Figure 1 shows the functions and which functions are used for what purposes. A Coloured Petri net is applied as the description technique for the architecture.

One function block converts a Design/CPN diagram to SNIFF's internal representation (*read diagram*), another converts the internal representation to text file format (*write text*). The two remaining function blocks handle the other direction (*read text*) and (*write diagram*). The figure is explained by desribing a scenario for exporting a Design/CPN diagram to SNIFF's text format. The small dashed box annotated with **Export** indicates the

[2] A direct backup of the net to a text file would also be possible, but we prefer a modular approach.

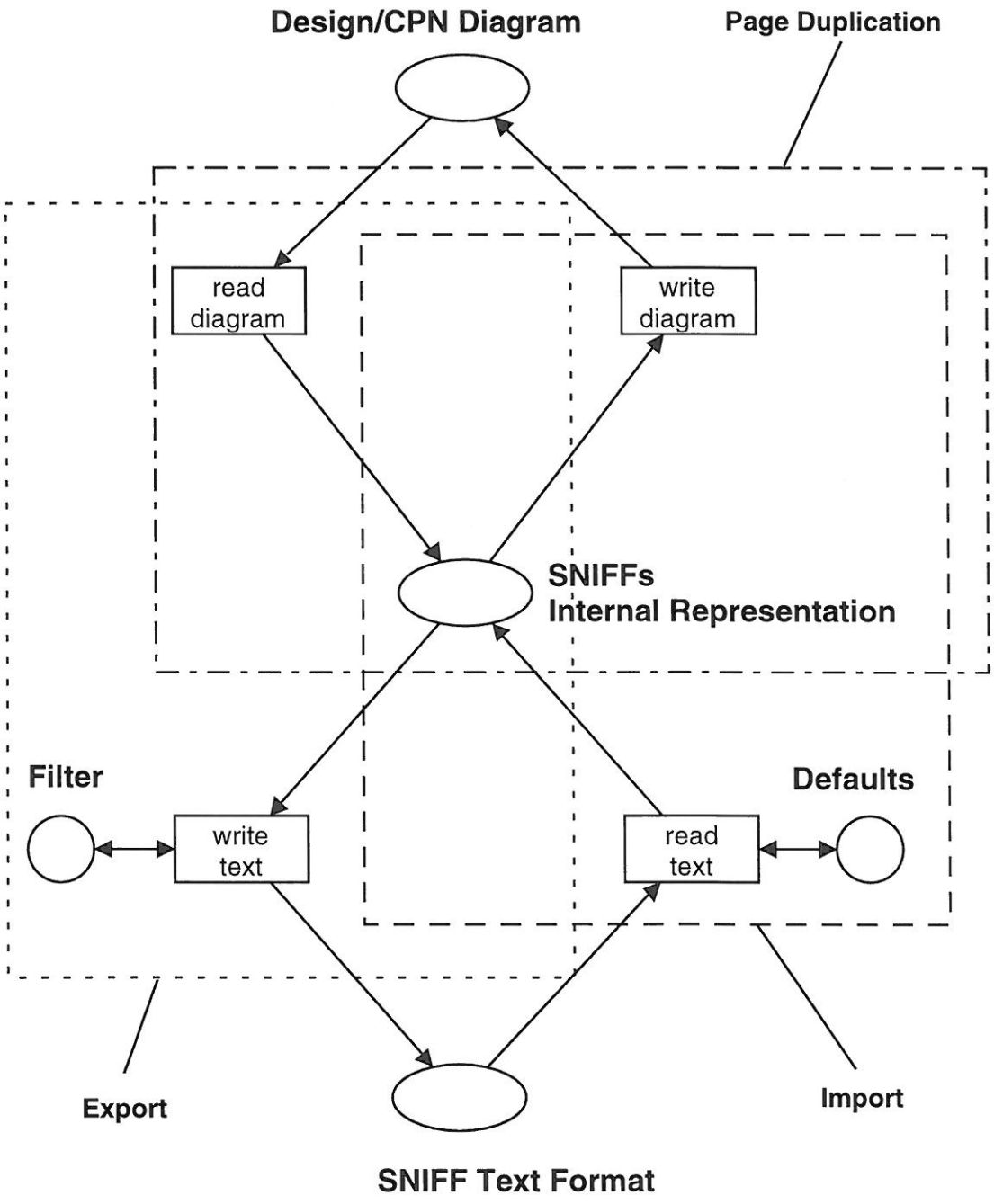



Figure 1: SNIFF's Architecture

involved parts.

Imagine a Design/CPN diagram lying as a token on the place[3] at the top of the figure. The upper left transition takes this token and generates a new token of the type '*Internal Representation*' and puts it on the place in the middle. The token represents the whole description of the net in a SNIFF specific format. Now there are two ways to go: The upper right transition and the lower left transition are activated as well. This means that one may call the function for producing a text format diagram or for generating new pages in the given Design/CPN diagram. (This would be the page duplication function, indicated by the dotted dashed box in figure 1.) To achieve our goal of exporting the diagram we choose the *write text* transition.

Our example scenario is simplified in terms of some adjustment possibilities. There is a text file influencing in detail which diagram elements and even which attributes are taken over from the internal representation to the text format diagram and which are dropped. In the figure it is shown as a side condition to the *write text* condition (place **Filter**). Another text file (place **Defaults**) controls the transformation of the internal representation back to a Design/CPN diagram. Attributes may be overridden by default values.

This functionality is not realized as a monolithic function but is done via function composition: A function's result is the next function's parameter. This allows for a flexible application of the SNIFF functions.

## 2.3 Functionality

SNIFF is started by a function call in an auxiliary box. Its outer appearance is a kind of graphical user interface. The user will see only the usual Design/CPN graphical interface, e.g. the file menu to select a backup file. To provide easy expandability the advanced user may call some of SNIFF's functions directly in any combiantion with other Design/ML functions.

The ordinary user gets a simple menu containing the following topics:

1 → Save Net to Text
2 → Restore Net from Text
4 → Save selected Pages
5 → Restore selected Pages
7 → Duplicate Pages
9 → Exit

The selection results in the direct execution of the function. Selection one and two are related to the whole diagram. After selecting them the user is asked for the name of the new backup file or for an already existing one respectively. By choosing selection four or five one gets the file menu as well as a new window with a list of all pages of the diagram. Now the mouse can be used to select any subset of pages for further processing. This page list is being examined for integrity and automatically supplemented by all pages which are

[3]So this place's colour is '*Design/CPN Diagram*'.

below the selected pages in the substitution hierarchy. Duplication of pages only needs a selection of pages, copies of these pages, including the subordinated pages, are integrated into the present diagram. The creation of the new pages takes care of already existing pages and their numbers. If necessary, the new pages get assigned new pages numbers.

The menu was implemented using Mimic for Design/CPN [Rasmussen und Singh 1995]. All other parts of SNIFF only use functions available in Design/ML. No external function library is needed for execution of SNIFF. So every user of Design/CPN can easily utilize SNIFF, too.

## 2.4 Grammar and Textual Format

The major goal of mapping a diagram is a complete description of all net contents to ensure the possibility of a one to one copy. Furthermore it is desirable to be able to translate the internal representation to a text file without too much effort. The text file itself should maintain the structure and should be human readable. This is because simple changes ought to be feasible in an ordinary text editor by hand. e.g. to correct corrupted diagrams. These requirements again influence the internal representation which in large parts resembles the text format. The internal representation is implemented as an ML datatype.

The functionality of SNIFF is closely connected to the internal structure of Design/CPN. Thus a central feature is a unique integer as ID assigned to each page, object, arc, and attribute. These IDs are used whenever a graphical object is referenced e.g. socket-port assignment. SNIFF processes diagrams page by page: The internal representation is a list of pages. Each page contains several fixed attributes and a list of its elements. As an analogy each element contains fixed attributes and a list of its subordinated elements like auxiliary regions, hierarchy informations, code segments and so on. The subordinated elements always contain a fixed number of fields. The integer constants for some attributes like '`shape=2`' could easily be replaced by symbolic constants for which one can find reasonable names in the Design/ML handbook [Design/CPN 1993a]. The complete SNIFF grammar can be found in appendix A. All non-terminals of the grammar are constants within the program code and are read in from a text file when starting SNIFF. Modification for keywords is therefore easy. The structure of the grammer requires more effort. Section 2.5 provides an introduction to this data structure by discussing two examples.

## 2.5 Examples

First a very simple diagram is presented as an example, which consists only of a single page (see figure 2). This page shows the three main elements of all Petri nets: place, transition, and arc. The arc is directed from place to transition. Beside the figure SNIFF's text output is shown. Each page block is encapsulated by a begin-end. Fixed attributes such as name or page number are followed by the blocks for the page's content. Places (the first block) and transitions (the last block) are somewhat similar in their attributes. The arc's block looks a little bit different because an arc needs extra fields for its orientation and its style. The example net has been scaled up so that its parts fit the text counterparts.

```
begin
    name=/*New*/;
    OID=4;
    subtyp=/*Simple*/;
    pagenr=1;
    visible=1;
    coord=[0,0,533,776];
    begin
        OID=7;
        typ=/*Place*/;
        subtyp=/*Simple*/;
        name=/**/;
        shape=2;
        visuals=[0,2,0,1];
        coord=[~15,~268,120,60];
    end
    begin
        OID=10;
        typ=/*Arc*/;
        subtyp=/*Simple*/;
        name=/**/;
        shape=20;
        visuals=[0,2,1,1];
        pointlist=[~15,~268,~15,65];
        coord=[7,8];
        begin
            orientation=1;
            connprops=[6,12,20,64,32];
        end
    end
    begin
        OID=8;
        typ=/*Trans*/;
        subtyp=/*Simple*/;
        name=/**/;
        shape=1;
        visuals=[0,2,0,1];
        coord=[~15,65,120,60];
    end
end
```

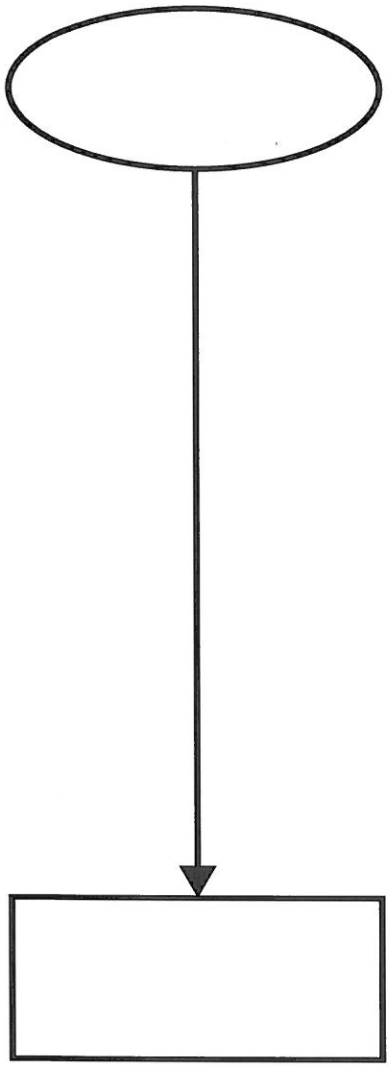


Figure 2: Example 1 for SNIFF text file and Petri net diagram

Appendix B contains an example for a Coloured Petri net which shows the two putative seasons of Hamburg (from the point of view of some pessimists). It exemplifies how regions are assigned to nodes and some other essential elements of Design/CPN diagrams. Both places have a name and a colour region. The 'Summer' place has an initial marking. Transitions are named, they do not have any guards or code regions. The arcs are inscribed with simple arc expressions. In appendix C the textual representation as produced by the export function of SNIFF can be found.

## 2.6 Discussion

Both examples are rather simple Coloured Petri nets. The size of textual representations of more complex nets prohibits larger examples here, however, the examples give an idea of the textual representation, its advantages and disadvantages. The benefits of SNIFF have already been mentioned above. It can be added, that the usual Design/CPN feature of saving a simulation state is still preserved, hence SNIFF is an add-on. The option to modify Design/CPN to use SNIFF as one save/load option in the file menu is independent from this.

However, it must be mentioned that the strong closeness to Design/CPN causes some problems, which are discussed in the following. The special tayloring to Design/ML with the specific Design/CPN functions allows no portation to other tools which use other platforms, even if the design of SNIFF tried to avoid this. The implementation had to obeye runtime requirements.

The grammar has been designed some years ago without the aim to fit any upcoming standard for High-Level Petri nets (HLPN). However, it should be possible to adopt it. The import-functions of SNIFF has to be adopted to be able to drop other information which is not present in usual Design/CPN diagrams. E.g. an extension of Design/CPN by test-arcs in a newer version would also require an appropriate representation in the internal data structure. For new functions there will be internal Design/CPN functions which could be easily integrated into the SNIFF functions. While new features, not present in Design/CPN had to be dropped or transformed by special functions.

The direct use of integers e.g. for the shape of an object (for transtions usually: `shape = 1`, for places usually: `shape = 2`), can be replaced by defined textual constants such as `shape = rectangle` or `shape = ellipse`, if this is desired. Here the internal values are used.

One problem using the Design/ML functions is that they do not provide all informations. Sometime values are returned e.g. as `unknown`, even if the behavior of Design/CPN clearly shows that in is known. This caused many time consuming work arounds. Even if the design could mainly be kept, the implementation had to be more complex than necessary otherwise.

The speed of the save and load functions is reasonable and at a linear speed compared to the overall number of objects, arcs, and attributes. However, burdened with a fixed amount of setup time. The loading of a file can be directly watched, hence the nets are build up using the usual Design/ML functions. The appearence is quite impressive and,

due to the rapidly drawn nets, does not disturbe the user while waiting for the completion of the diagram, since the progress can be directly followed.

The last versions of the SNIFF library have been tested on Sun and Linux platforms only. Only very few problems were encountered between the different version. E.g. the hierarchy page of the Sun version has a bug, which prevents the preservation of its graphical structure, while the Linux version does not have this problem. The Machintosh platform was only used at the beginning of the SNIFF project. Mainly the insufficient performance of the available Machintosh computers prevented the portation for the newer Design/CPN versions to those machines. Also there were some minor problems with the compatability of the ML versions. Furthermore, the graphical appearance on the Machintosh/Sun platforms can be different. The colours used at a Machintosh have to be mapped to black/white attributes such as dotted lines. This caused the provision of the text files for filters and defaults as can be seen in figure 1.

Altogether a new tool has been created which gives Design/CPN users the possibility of using their diagrams in a more widespread way. Figure 3 gives an idea of SNIFF's possibilities. The ASCII text allows to provide Design/CPN diagrams to other tools.[4] They have to provide an import and an export function. Due to the SNIFF text format the exchange between other tools also becomes possible, if they stick to the format. What was of special importance for us, was the possibility to allow external generation of Coloured Petri nets, especially for class diagrams which have a considerable amount of components when described as Colured Petri nets. What is interesting at this point is that Design/CPN itself can be used as the generating tool, hence it can draw arbitrary diagrams, which do not have to be Coloured Petri nets. The Design/ML allows the expanding or transformation of the other diagrams, for which again SNIFF functions can be used. This has been used in the SIMON[5] project [Moldt 1996] and worked quite well. The generation of special Petri nets is especially important for net formalisms with a high degree of administrative overhead like pages for class diagrams or message schedulers in object-oriented Petri nets.

# 3 Conclusion and Future Plans

SNIFF has been used within several projects. Due to the difficult access to the internal representation several versions had to be build to cope with the available internal Design/ML functions. The availability of SNIFF has widened the range of possible applications considerably, covering the before mentioned features. It would be interesting to see how the interchange format for Design/CPN could be used in the context of the evolving international standard (see [WWW-PetriNetStandard 1998]).

At the Computer Science department of the University of Hamburg SNIFF is at present used within projects for diagram merging, for backups as well as for importing automatically generated net descriptions into Design/CPN. These external functions for generating textual Coloured Petri nets outside Design/CPN are still at prototype level.

[4] XPN in figure 3 stands for all other tools.

[5] SIMON stands for *SIMulator for Object-oriented Petri Nets*

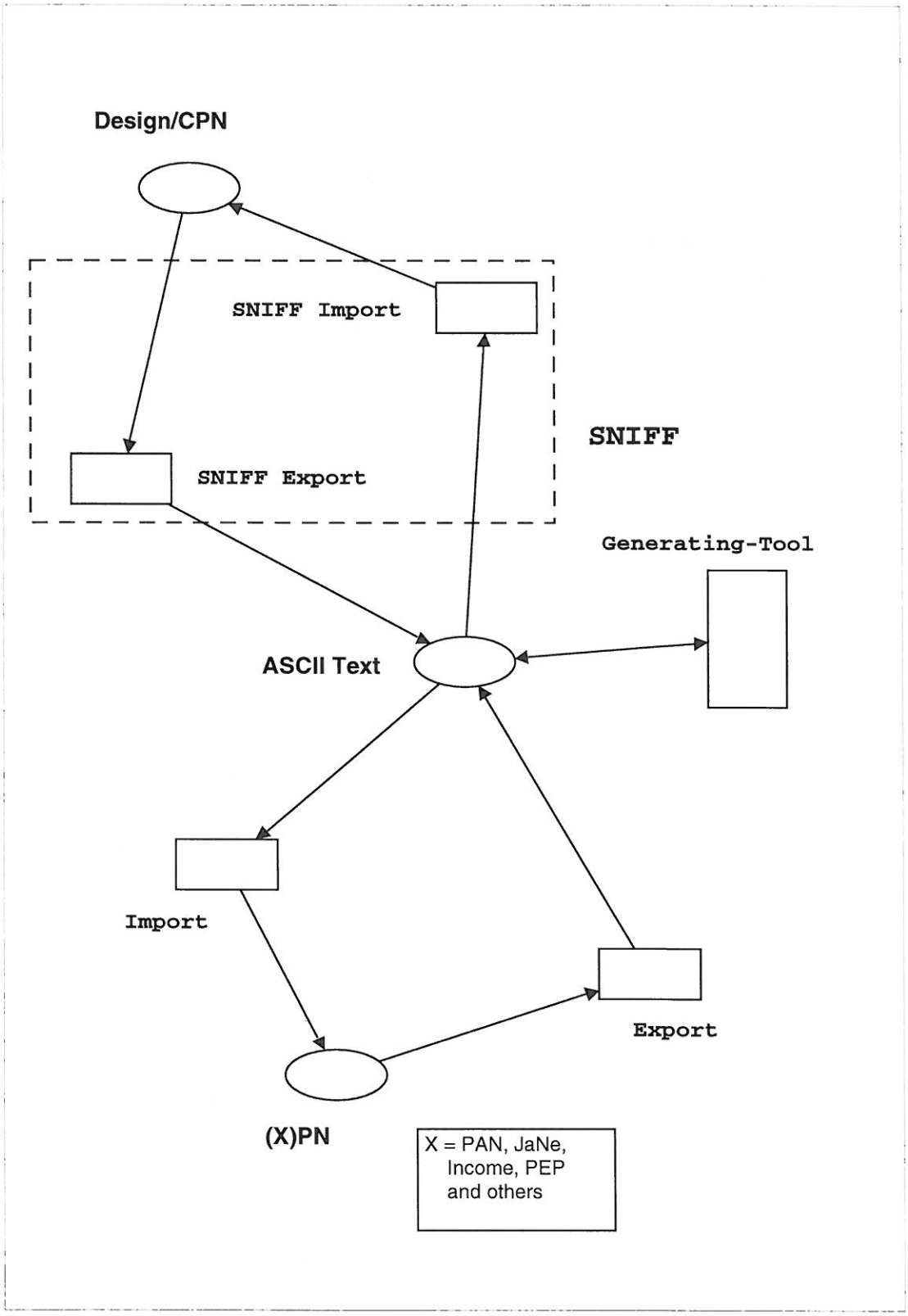



Figure 3: Possible use of SNIFF

# A Grammar of SNIFF's text file output

```
net             ::= <page>*

page            ::= begin
                        name            = <string>;
                        OID             = <int>;
                        subtyp          = <ObjSubType>;
                        pagenr          = <int>;
                        visible         = <int>;
                        coord           = <int list>;
                        { <element>* }
                    end

element         ::= <object> | <arc>

object          ::= begin
                        name            = <string>;
                        OID             = <int>;
                        typ             = <ObjType>;
                        subtyp          = <ObjSubType>;
                        shape           = <int>;
                        visuals         = <int list>;
                        fontsize        = <int>;
                        coord           = <int list>;
                        { begin <attribute>* end }
                    end

arc             ::= begin
                        name            = <string>;
                        OID             = <int>;
                        typ             = Arc;
                        subtyp          = <ObjSubType>;
                        shape           = <int>;
                        visuals         = <int list>;
                        pointlist       = <int list>;
                        coord           = <int list>;
                        begin
                                { <attribute>* }
                                orientation     = <int>;
                                connprops       = <int list>;
                        end
```

```
                    end

attribute       ::= <attr_sort> = <string> with
                        OID             = <int>;
                        parentid        = <int>;
                        shape           = <int>;
                        visuals         = <int list>;
                        fontsize        = <int>;
                        coord           = <int list>;
                    end

ObjType         ::= Unknown | DefBox | Trans | Place

ObjSubType      ::= Simple | Subtr | SubPl | Glob | Temp | Loc
                  | In | Out | InOut | Gen | Sub | Inv
                  | GlobFus | PageFus | InstFus

attr_sort       ::= name | region
                  | namereg
                  | color | initmark | port
                  | globalfusion | pagefusion | instfusion
                  | guard | codeseg | time | hierarchy |
                  | expr

string          ::= /* <ident> */

int             ::= {~} <digit>+

int list        ::= [] | [ <int> {, <int>}+ ]
```

# B Design/CPN Petri Net Example

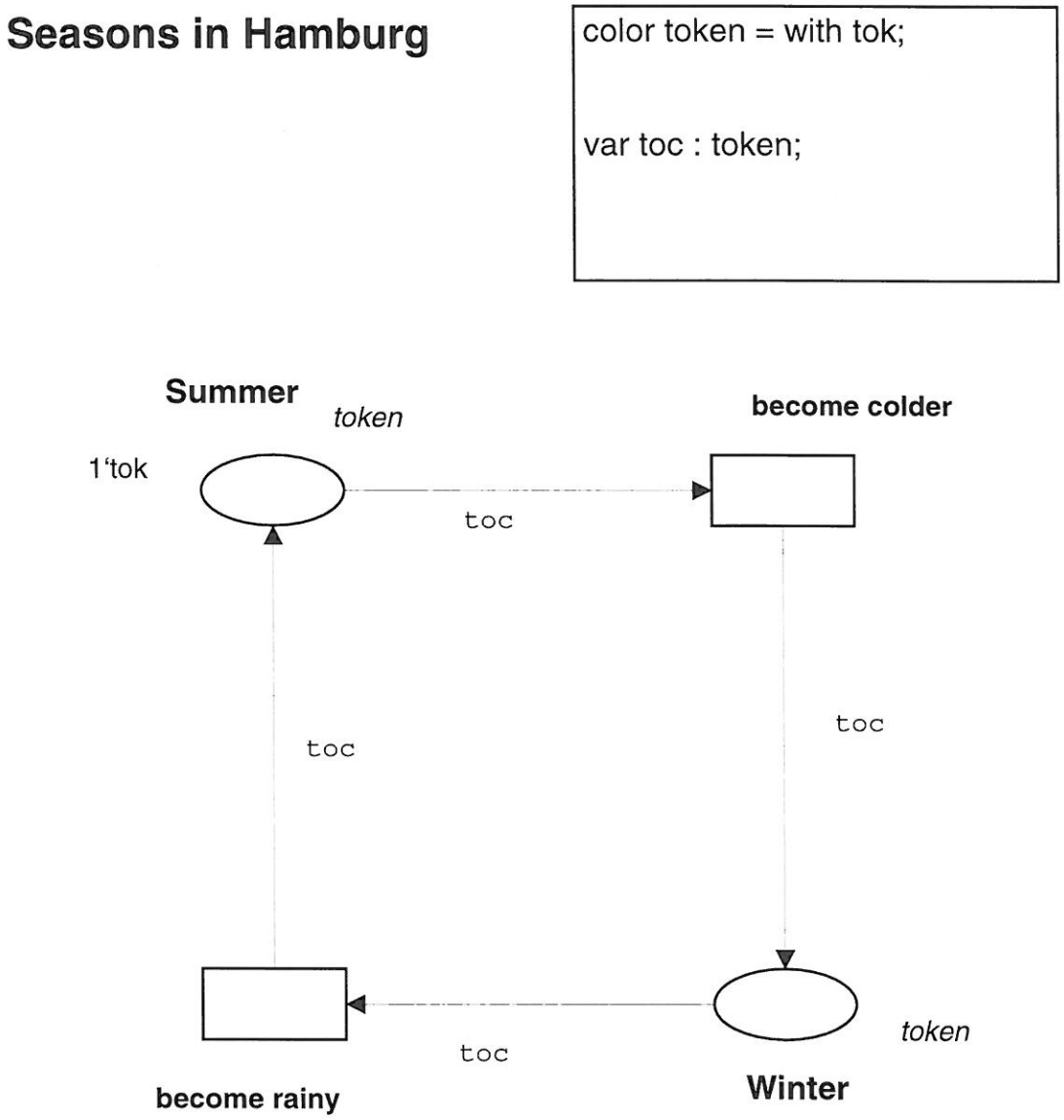



Figure 4: A more "complex" net

# C Design/CPN Petri Net Example as SNIFF text file

```
begin
    name=/*New*/;
    OID=18;
    subtyp=/*Simple*/;
    pagenr=1;
    visible=1;
    coord=[0,0,533,776];
    begin
        OID=36;
        typ=/*Unknown*/;
        subtyp=/*Simple*/;
        name=/*Seasons in Hamburg*/;
        shape=1;
        visuals=[0,1,0,1];
        fontsize=18;
        coord=[~145,~347,188,23];
    end
    begin
        OID=34;
        typ=/*DefBox*/;
        subtyp=/*Glob*/;
        name=/*color token = with tok;
               var toc : token;
             */;
        shape=1;
        visuals=[0,1,0,1];
        fontsize=14;
        coord=[106,~301,204,116];
    end
    begin
        OID=21;
        typ=/*Trans*/;
        subtyp=/*Simple*/;
        name=/**/;
        shape=1;
        visuals=[0,1,0,1];
        coord=[~124,64,60,30];
        begin
            namereg=/*become rainy*/ with
                OID=32;
                parentid=21;
                coord=[~134,104,86,16];
                shape=1;
                visuals=[0,0,0,1];
                fontsize=12;
            end
        end
    end
begin
    OID=2;
    typ=/*Trans*/;
    subtyp=/*Simple*/;
    name=/**/;
    shape=1;
    visuals=[0,1,0,1];
    coord=[91,~154,60,30];
    begin
        namereg=/*become colder*/ with
            OID=30;
            parentid=2;
            coord=[121,~190,92,16];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=12;
        end
    end
end
begin
    OID=5;
    typ=/*Place*/;
    subtyp=/*Simple*/;
    name=/**/;
    shape=2;
    visuals=[0,1,0,1];
    coord=[91,64,60,30];
    begin
        namereg=/*Winter*/ with
            OID=28;
```

```
            parentid=5;
            coord=[97,99,52,18];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=14;
        end
        color=/*token*/ with
            OID=38;
            parentid=5;
            coord=[154,75,36,16];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=12;
        end
    end
end
begin
    OID=7;
    typ=/*Place*/;
    subtyp=/*Simple*/;
    name=/**/;
    shape=2;
    visuals=[0,1,0,1];
    coord=[~124,~154,60,30];
    begin
        namereg=/*Summer*/ with
            OID=26;
            parentid=7;
            coord=[~141,~196,63,18];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=14;
        end
        color=/*token*/ with
            OID=37;
            parentid=7;
            coord=[~84,~185,36,16];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=12;
        end
        initmark=/*1`tok*/ with
            OID=39;
            parentid=7;
            coord=[~189,~163,33,16];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=12;
        end
    end
end
begin
    OID=25;
    typ=/*Arc*/;
    subtyp=/*Simple*/;
    name=/**/;
    shape=20;
    visuals=[0,0,1,1];
    pointlist=[~124,64,~124,~154];
    coord=[21,7];
    begin
        expr=/*toc*/ with
            OID=60;
            parentid=25;
            coord=[~101,~45,27,15];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=12;
        end
        orientation=1;
        connprops=[4,8,20,1,1];
    end
end
begin
    OID=24;
    typ=/*Arc*/;
    subtyp=/*Simple*/;
    name=/**/;
    shape=20;
    visuals=[0,0,1,1];
    pointlist=[91,64,~124,64];
    coord=[5,21];
    begin
        expr=/*toc*/ with
            OID=66;
            parentid=24;
```

```
            coord=[~37,85,27,15];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=12;
        end
        orientation=1;
        connprops=[4,8,20,1,1];
    end
end
begin
    OID=23;
    typ=/*Arc*/;
    subtyp=/*Simple*/;
    name=/**/;
    shape=20;
    visuals=[0,0,1,1];
    pointlist=[91,~154,91,64];
    coord=[2,5];
    begin
        expr=/*toc*/ with
            OID=67;
            parentid=23;
            coord=[122,~56,27,15];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=12;
        end
        orientation=1;
        connprops=[4,8,20,1,1];
    end
end
begin
    OID=22;
    typ=/*Arc*/;
    subtyp=/*Simple*/;
    name=/**/;
    shape=20;
    visuals=[0,0,1,1];
    pointlist=[~124,~154,91,~154];
    coord=[7,2];
    begin
        expr=/*toc*/ with
            OID=64;
            parentid=22;
            coord=[~34,~141,27,15];
            shape=1;
            visuals=[0,0,0,1];
            fontsize=12;
        end
        orientation=1;
        connprops=[4,8,20,1,1];
    end
end
end
```

# References


[Becker und Moldt 1993] Ulrich Becker and Daniel Moldt. Object-Oriented Concepts for Coloured Petri Nets. In IEEE, editor, "Conference Proceedings, IEEE International Conference on Systems, Man and Cybernetics", volume 3, pages 279–286, Le Touquet, Frankreich (17.–20. October 1993). IEEE.

[Beckmann 1993] Nicole Beckmann. Prototyping mit gefärbten Petrinetzen. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg, Germany (November 1993).

[Design/CPN 1993a] "Design/CPN Handbook: CPN ML – CPN Palette; Part 1, Version 2.0". Cambridge, MA, USA (1993).

[Design/CPN 1993b] Meta Software Corporation, Cambridge, MA, USA. "Design/CPN Handbook Version 2.0" (1993).

[Fowler und Scott 1997] Martin Fowler and Kendall Scott. "UML Distilled – Applying the Standard Object Modeling Language". Addison-Wesley, Reading, Massachusetts (1997).

[Harper 1986] Robert Harper. Introduction to Standard ML. LFCS Report Series ECS-LFCS-86-14, University of Edinburg (1986).

[Jensen 1992] 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).

[Maier 1996] Christoph Maier. Darstellung von Konzepten der objektorientierten Modellierung und Programmierung mit Petrinetzen. Studienarbeit, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg, Germany (May 1996).

[Moldt 1996] Daniel Moldt. "Höhere Petrinetze als Grundlage für Systemspezifikationen". Dissertation, Universität Hamburg, Fachbereich Informatik, Vogt-Kölln Str. 30, 22527 Hamburg, Germany (August 1996).

[Paulson 1991] Lawrence C. Paulson. "ML for the Working Programmer". Cambridge University Press, Cambridge, England (1991).

[Rasmussen und Singh 1995] Jens L. Rasmussen and Mejar Singh. "Mimic/CPN A Graphic Animation Utility for Design/CPN". Computer Science Department, Aarhus University, DK-8000 Aarhus C, Denmark, 1.5 edition (1995).

[WWW-Design/CPN 1998a] "http://www.daimi.aau.dk/designCPN/" (1998).

[WWW-Design/CPN 1998b] "http://www.daimi.aau.dk/PetriNets/tools/db/designcpn.html" (1998).

[WWW-Mimic 1998] "http://www.daimi.aau.dk/designCPN/libs/mimic/" (1998).

[WWW-PetriNetStandard 1998] "http://www.daimi.aau.dk/PetriNets/standard/" (1998).

# On Modelling Train Traffic in a Model Train System

Wolfgang Hielscher, Lars Urbszat, Claus Reinke, and Werner Kluge
Department of Computer Science
University of Kiel
D–24105 Kiel, Germany
E–mail: wk@informatik.uni-kiel.de

May 15, 1998

**Abstract**

*The paper describes the design of a coloured Petri net model for a rather complex model train system. The purpose of this system is to teach graduate CS students net modelling and analysis techniques, and the systematic concersion of non–trivial net models into fully operational real systems.*

*The track layout of this system currently includes three main cyclic tracks, each subdivided into several sections, three switchyards of several sidings, and also interconnecting tracks via which trains may change main tracks and directions.*

*The idea is to equip each of several trains - currently up to ten - with its own travel plan. It specifies a sequence of tracks through which the train must be routed in the given order. Execution of these plans must be dynamically coordinated based on locally made decisions about the allocation of track sections to requesting trains so that essential safety and liveness properties are met.*

*The paper first introduces the basic net components necessary to model train movement along track sections and across branching and merging switches, then describes the composition of a complete track model from these components, including the controls necessary to enforce an orderly behaviour, and then outlines the composition of the complete system model. It also addresses some of the as yet unsolved problems of deadlock prevention in the system.*

## 1 Introduction

This paper relates to the organization of an orderly train traffic in a complex model train system which serves to teach graduate students how to model, by means of coloured Petri–Nets, the dynamic behaviour of non–trivial real life systems with concurrent activities and how to translate these models into working control programs.

The track layout of this train system is depicted in fig. 1. It consists of

- three main circular tracks called the outer circle (labeled OC_LN), the inner circle (labeled IC_LN), and the kicking horse pass [1] (labeled KH_LN) along which trains may move counterclockwise, clockwise, and in both directions, respectively, as indicated by the arrows;
- switchyards of three to five sidings included in each of the main tracks (the outer circle station (labeled OC_ST), the inner circle station (labeled IC_ST) and the kicking horse pass station (labeled KH_ST)), and also

[1] This name is adopted from a section of the Canadian Pacific Railways mainline in the Rocky Mountains which includes a similarly spiral–shaped track layout to negotiate a rather steep uphill / downhill passage [Po95].

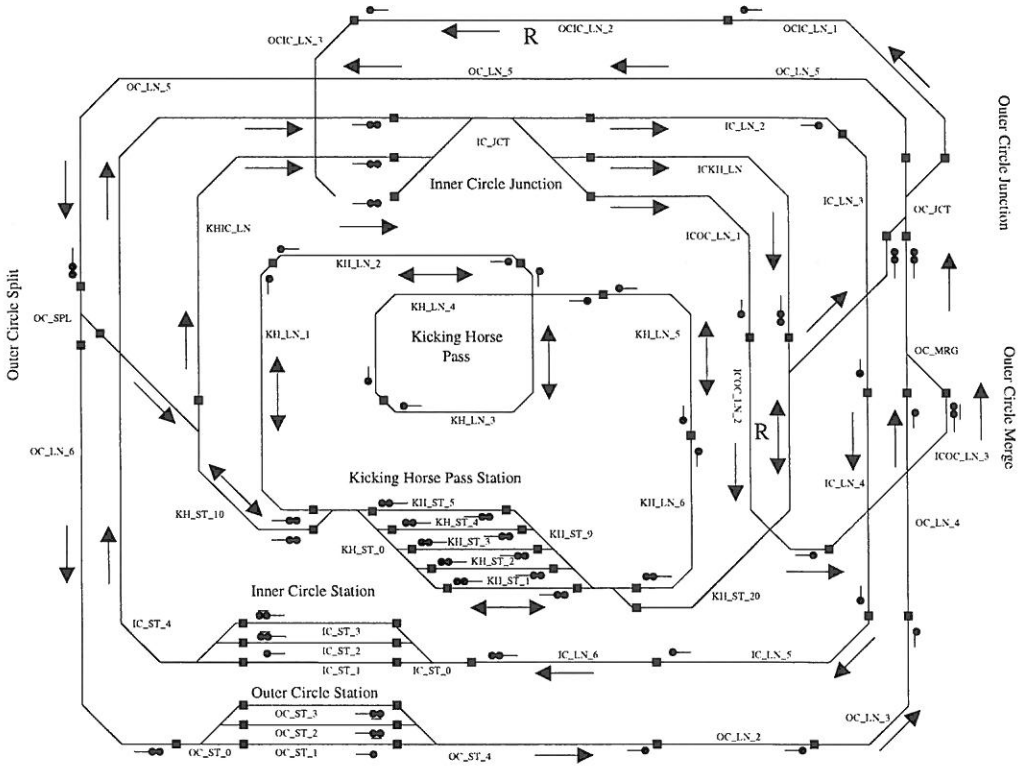



Figure 1: Complete track layout of the system

- interconnecting tracks, labeled OCIC_LN and ICOC_LN, through which trains may change main tracks and thereby also change directions without needing to shunt engines;

Each of the tracks is subdivided into several sections (or blocks), in the figure separated by little square–shaped dots. Light signals, symbolized as little dots extended by short bars, visualize permission to cross section boundaries. Actual train positions are detected by magnetic sensors placed along the tracks, and the speed of trains is controlled by applying appropriate voltage to track sections.

Sensor data are, by a dedicated digital interface, scanned and passed on, as a stream of 256 bytes, to a SUN-Workstation. These sensor data are translated into a description of the current system state, from which a control program executed by the SUN derives the control signals that effect transition to a next state and issues them, through the same interface, as a stream of 256 bytes to the switches, signals and the voltage supplies. This control cycle is executed in periods of a few milliseconds to react quickly enough to critical situations, with trains moving at most some 10 mm during that time.

The track system is laid out on an area of $4.5 * 3.5$ square meters. It includes 145 meters of HO gauge tracks, subdivided into 37 sections and sidings, 28 two-way switches and 51 signals. The inner and outer circles are configured as intertwined loops of about 28 meters length each, running in parallel, and the kicking horse pass includes two double spirals on which trains climb up to (and down from) about 30 centimeters above ground level on a track length of about 25 meters. It takes about two minutes for trains to complete a single lap on the inner and outer circle, and about two and a half minutes over the kicking horse pass.

The photograph of fig. 2 gives an areal view of the track layout, showing the harp-like structures of the station sidings on the left, the tracks of the outer and

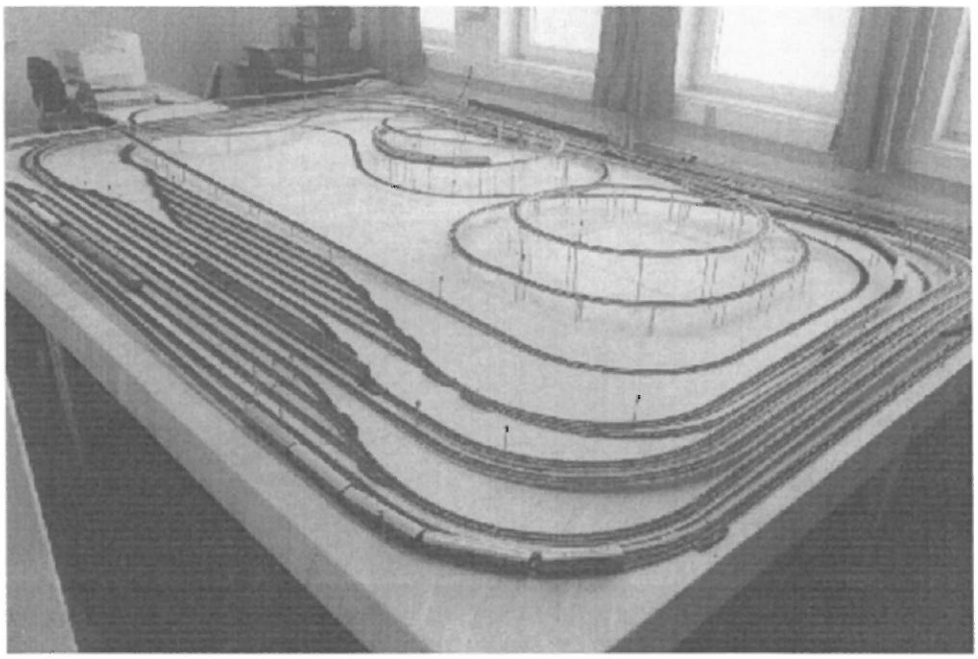


Figure 2: A birds eye's view of the system

inner circles running along the periphery, and the spirals of the kicking horse pass in the center part.

The design of this track system was guided by the objective of providing, within the confines of limited real estate, sufficient complexity with regard to train movement - currently up to 10 trains can be operated simultaneously - so that all the phenomena that typically occur in systems with concurrent activities can be studied and experimented with in a real-life setting.

To this end, train movement is governed by travel plans which for each train individually specify which (sections of) tracks it has to pass through in which order at which speed, and at which stations it has to stop for some time. Each plan sets out with some initial positioning of the train in a specific station siding, and after some finite number of passages through main tracks terminates with the train arriving at the same or some other siding [2]. A typical plan for a train that passes through all three main tracks is given in fig. 3.

The control program that receives these travel plans as input must coordinate their execution 'on the fly', based on local decision making as conflicts and potential deadlock situations arise. In doing so, it should obey first principles of an orderly system behavior, defined in terms of essential safety and liveness properties:

- each track section must be occupied by at most one train at a time (the mutual exclusion principle);
- a train that occupies a particular section of track, including station sidings, must always find a way out unless it has reached its terminal position (completed its plan);
- no train must be unduly delayed in pursuing its travel plan: entry into some track section must be granted eventually to all trains competing for it;

[2] Choosing the same siding as the starting and terminal position has the advantage that the same plan can conveniently be repeated several times in succession (or even forever).

```
train "freight train #17":
   operation speed   6
   minimum   speed   1
   priority          10

   # initial position: siding #1 in the kicking horse Pass station
   # initial heading:  clockwise
   from KH_ST_1 CLK

   # Move to Inner Circle Station and stop there for (at least) 10 seconds.
   goto IC_ST  stop 10 sec

   # Take 3 turns on the Inner Circle without stopping. Upon
   # completing the last turn, stop for 15 seconds.

   loop 2
      goto IC_ST
   endloop
   goto IC_ST  stop 15 sec

   # Go to the kicking horse Pass Station heading counterclockwise,
   # go over the pass once, then move to the Outer Circle Station,
   # stop there for 17 seconds, and finally return to the initial
   # position KH_ST_1.

   goto KH_ST CNTCLK
   goto KH_ST CNTCLK
   goto OC_ST         stop 17 sec
   and return
```

Figure 3: Travel plan for a single train

- a train requesting entry into a track section must be permitted to proceed immediately if no other train competes for it.

A system which meets these essentials can usually be modelled as an ordinary Petri–net and by proving that it satisfies certain invariance properties [GeLaTh80, KlLa82]. However, things are not that simple if, in addition to these essentials, individual travel plans and other train–specific parameters have to be included into the model. Such parameters may be priorities, e.g., of passenger trains over freight trains, time tables which need to be followed (with trains falling behind getting their priorities dynamically stepped up), or pre–specified speed profiles along the travel routes. It takes higher–order Petri–nets [GeLa78, GeLa81, Rei85, Jen90, Jen92] to represent trains as tokens which carry along with them inscriptions to this effect and to have the firing of transitions in some consistent form controlled by these parameters, and the parameters changed if necessary.

The sequel outlines the design of such a net model for the entire system, using the Design/CPN tool version 3.0.4 of Aarhus University for coloured Petri–nets. The design is described from the bottom–up, setting out in the next section with basic net components which model train movement across section boundaries and across merging and branching switch configurations. Section 3 describes in some detail the net model for the kicking horse pass, including the station and the track sections that lead into and out of it, and section 4 briefly explains how the complete system model is assembled from net abstractions (substitution transitions) for the inner and outer circle tracks and for the kicking horse pass track. Section 5 discusses some open problems of deadlock prevention within the system, and the conclusion summarizes some of the experiences we had with the Design/CPN tool.

Since the system model is rather complex – its full net specification requires some 28 pages and a total of about 400 places – the paper merely attempts to

convey some basic features of its construction and its interpretation (semantics), but it cannot be very specific on all details. In fact, the net models of the kicking horse pass and of the complete system have been cleaned of some components which relate to rather special control problems in order to get the underlying ideas across more clearly.

# 2 The Basic Net Components

This section is to introduce the net components by which the basic building blocks of the track system and the train movement therein can be modelled.

## 2.1 Track Sections and Train Descriptions

For purely organizational purposes it suffices to model each section of track by a place with a color specifying the type of tokens it may carry. As these tokens must represent the presence or absence of trains in the section, and a train must be characterized by a number of attributes (or parameters), the color `block_descr` associated with these places must be of the general form

```
color block_descr = union u_train : train_descr + no_train,
```

where the colors `train_descr` and `no_train` respectively specify the type of a train description and the absence of a train in the section. For reasons that will become clear when specifying the net component which controls train movement from one section to the next it must be guaranteed that a token of either type resides in such a place under all markings (token distributions) of the net. The color `train_descr`, in turn, is defined as [3];

```
color train_descr = product t_state * t_train_id * t_dir * t_sched
```

with

- `color t_state = with moving | stopped` distinguishing between the train moving through or temporarily stopping in the section;
- `color t_train_id = ( 1 .. n )` serving as indices to distinguish the `n` trains moving about the tracks;
- `color t_dir = with clk | co_clk` distinguishing between clockwise and counterclockwise movement along the tracks;
- `color t_sched = list t_dest` defining the trains travel plan in terms of switch positions of type `color t_dest = with left | straight | right` .

Specifying travel plans as lists of switch positions which must be interpreted from head to tail is a low–level description which suffices to route trains from start to destination: choices among alternative routes must only be made at sections where tracks branch into two or more, otherwise next sections are unique [4].

The color `train_descr` is minimal, in terms of numbers and types of its components (attributes), with respect to an orderly coordination of the train traffic about the track system. No provisions are as yet made to specify priorities or speed profiles.

[3] The prefixes `t_` are to distinguish the colors (types) within the type product from the variables used in the nets to identify the components of train descriptors.

[4] These low–level travel plans can be readily derived from high–level specifications as exemplified in fig. 3.

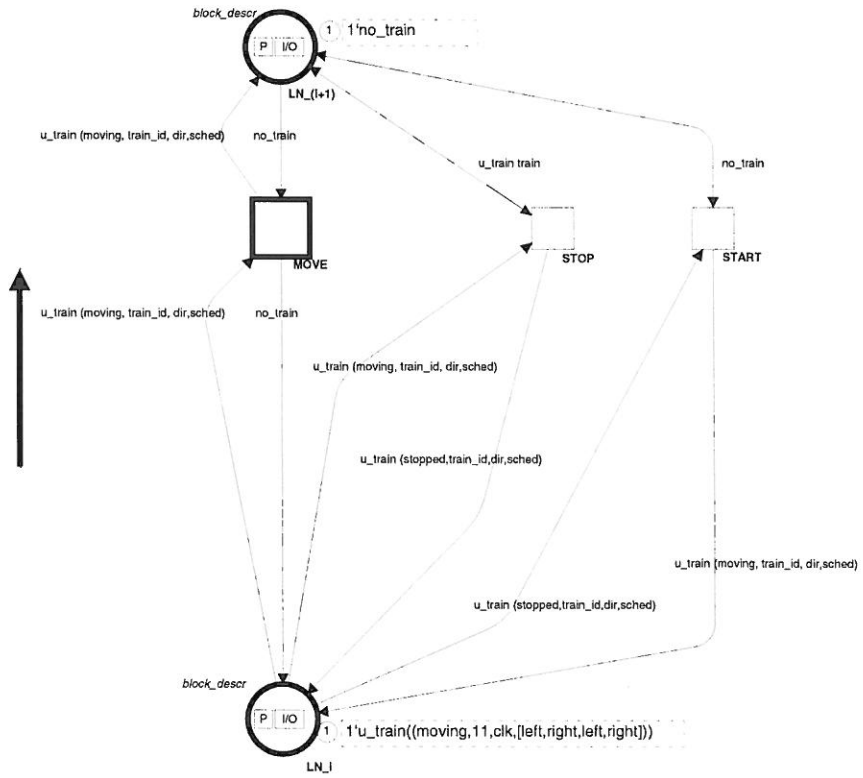



Figure 4: Net component for unidirectional train movement between adjacent track sections

## 2.2 Moving Trains from one Section to the Next

Modelling the unidirectional movement of a train along track sections requires a net component as shown in fig. 4. It includes two places LN_$i$ and LN_$(i+1)$ which model the sections from where to where a train is supposed to proceed, and three transitions which analyze the tokens in both places to take appropriate actions [5] . In particular, the transition

- MOVE is enabled if a train is in section LN_$i$ and moving, and no train is in LN_$(i+1)$, as indicated by the token inscriptions, in which case the transition fires and exchanges the two tokens (i.e., the train moves on);
- STOP is enabled if a train moves through section LN_$i$ and section LN_$(i+1)$ is occupied by another train, in which case the state of the train in LN_$i$ must be changed to `stopped`.
- START returns the status of a stopped train in section LN_$i$ back to `moving` if the train that occupied section LN_$(i+1)$ has moved on and left.

There can be at most one of these transitions enabled as the respective markings are mutually exclusive.

A minor extension of this net component is necessary to deal with track sections in which trains are not allowed to stop. They include the inner and outer circle junctions through which trains may change main tracks, but also other sections

[5] Three transitions are used for structural clarity here. They can, of course, be folded into one in which the three situations that need to be taken care of are distinguished by appropriate inscription.

with switches in them. Here it is necessary to make sure that not only the no-stop section but also the section which the train must reach after it are free. This can be accomplished by means of a so-called look–ahead place, denoted as LA_ ..., of `color t_la = with free | occ` connected to a CTR_ ... transition which imposes an additional condition on its firing: it is disabled if the token in this place carries the color `occ` and, depending on the tokens in the places LN_$i$ and LN_$(i+1)$, may be enabled if the color is `free`. The token that actually resides in place LA_... is generated by some look–ahead subnet which inspects the tokens in the relevant section places.

To model adjacent track sections along which trains may move in both directions (as in the kicking horse pass), the transitions MOVE, STOP and START must simply be duplicated, with one set each taking care of clockwise and counterclockwise direction.

## 2.3 Controlling Train Movement Across Switches

Switches in both unidirectional main tracks (the inner and outer circles) are operated as either

- branching switches through which trains are routed from an incoming track section to one of several outgoing track sections;
- merging switches through which trains are routed from one of several incoming sections to one outgoing section.

Switches within the kicking horse pass are operated both ways, depending on the direction in which trains are going. Inner and outer circle junctions are made up from a set of merging switches followed by a set of branching switches. The switches themselves belong to no-stop sections between the incoming and outgoing sections [6].

In the simple setting that nothing other than the presence (or absence) of trains matters, merging switches must be operated as follows: If there is only one train in any of the incoming sections and both the no-stop switch section and the outgoing section are free, then the train may move on without delay by setting the switch(es) accordingly. If there are two or more trains in the incoming sections competing for passage, then the conflict is resolved by arbitration. A typical example is the net component of fig. 5 which models the merge switch through which trains leave the sidings of the outer circle station [7]. It consists of a place each for the three incoming sections (the sidings) OC_ST_1, .. , OC_ST_3, for the no-stop section OC_ST_4, and for the outgoing section OC_LN_2. The transitions CTR_OC_ST_1, .. , CTR_OC_ST_3 which are to swap train tokens between the sidings and the no-stop section are controlled by look–ahead places LA_OC_ST_I which receive control tokens from some look–ahead subnet encapsulated in the substitution transition LA_OC_ST[123]. This subnet monitors whether there are trains in any of the sidings and both the no-stop section and the outgoing section are free. In this case, the look–ahead subnet injects a `free` token as a selector in one of the look–ahead places which has the respective siding occupied by a train, whereas all other look–ahead tokens remain set to `occ`. This enables the selected train token to proceed to the no-stop section (whereupon the look–ahead token is reset to `occ`) and, via the MOVE transition, on to the track section OC_LN_2. The look–ahead subnet selects competing trains based on fair simulation in order to prevent waiting trains from being unduly delayed. As an additional safety measure which relates to implementation details of the controls,

[6] $n$–fold switches of either kind are technically implemented as cascades of $n-1$ two–way switches.

[7] In this and all the following nets the tokens are omitted since inscriptions that are readable would take up too much space.

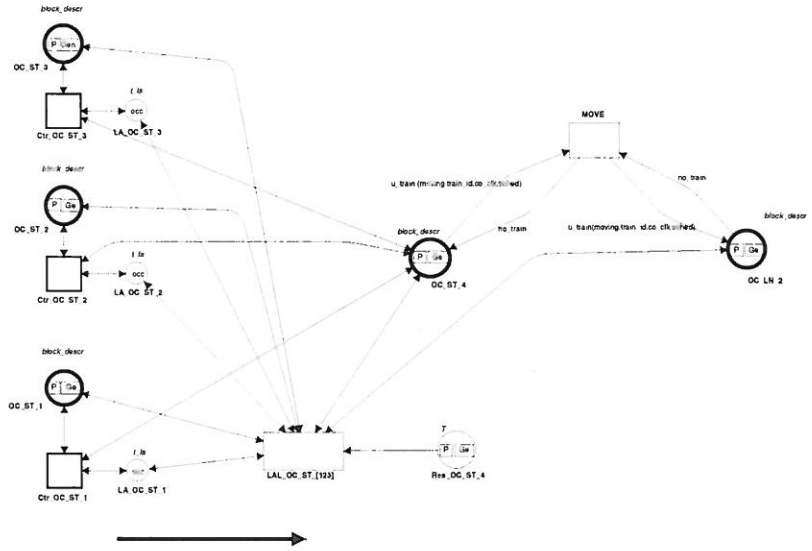



Figure 5: Net model of a three–way merge switch

the firing of the look–head transition is also made dependent on the consumption of a plain control token from the place RES_OC_ST_4, which is returned if the train token is removed from place OC_LN_2, i.e., the train leaves the outgoing section. This measure ensures that no train is in either the no-stop or the outgoing track section before one of the CTR_OC_ST_$i$ transitions can be enabled.

Branching switches come in a non–deterministic and in a deterministic version. The former is to control entry of trains into any of the free sidings of a station where it does not matter which one is actually chosen, the latter is to control selection of a specific outgoing track.

Consider, as an example, the net component for the branching switch as depicted in fig. 6 which non–deterministically controls entry of trains into the outer circle station. It comprises a place each for the incoming track section OC_LN_6, for the three outgoing track sections (the sidings) OC_ST_1, .. , OC_ST_3, and for the no-stop switch section OC_ST_0. The firing of the entry transition CTR_OC_LN_6 is made dependent on a look–ahead place LA_OC_LN_6 whose marking is worked out by some look–ahead logic inside the substitution transition LA_OC_ST_0. This logic checks whether any of the outgoing sections and the no-stop section are free. This being the case, it injects a `free` token into LA_OC_LN_6 to allow the train token to proceed to OC_ST_0 and from there across one of the enabled MOVE transitions to a free siding; otherwise it is not enabled, i.e., the `occ` token remains in place, which stops the train in the incoming track section.

An example for the net model of a deterministic branching switch is the one that allows trains to either change from the track section OC_LN_5 of the outer circle to section KH_ST_10 of the kicking horse pass station or to continue along the outer circle track to section OC_LN_6 (see fig. 7). Apart from different labels for places and transitions, it basically differs from the net model for the non–deterministic branching switch in that it includes another place DEST which is fed with a selector token `left` or `right` extracted from the train token as it enters OC_LN_5 (which is not included in this subnet). This token enables either of the transitions CTR_SPL_$left$ or CTR_OC_SPL_$right$ to route the train token in OC_LN_5 to either KH_ST_10 or OC_LN_6, respectively.

Deterministic switches are in the entire track system used only unidirectionally. Non–deterministic switches may also be used, in reverse direction, as merging

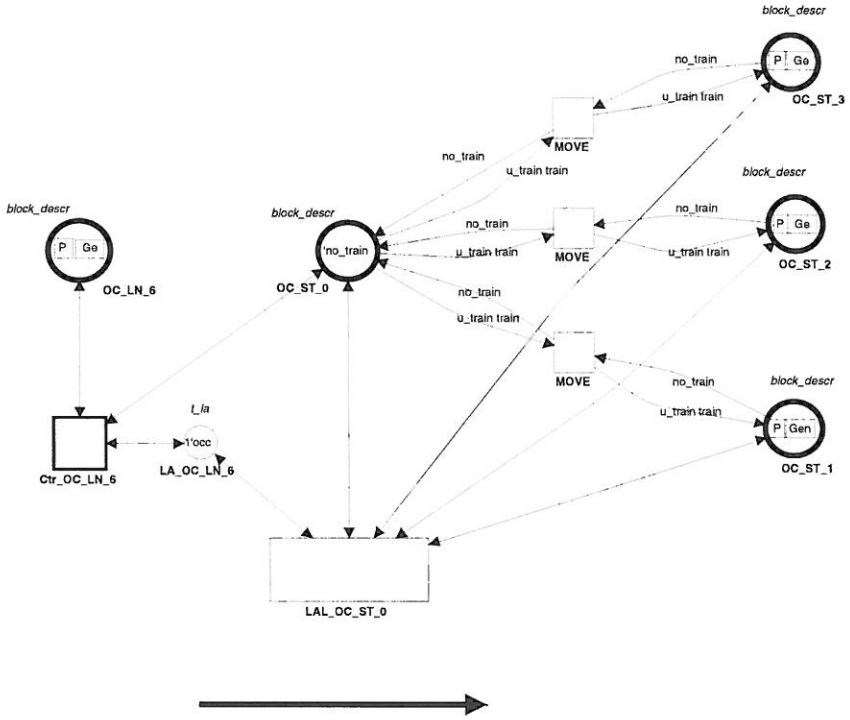



Figure 6: Net model of a three–way non–deterministic branching switch

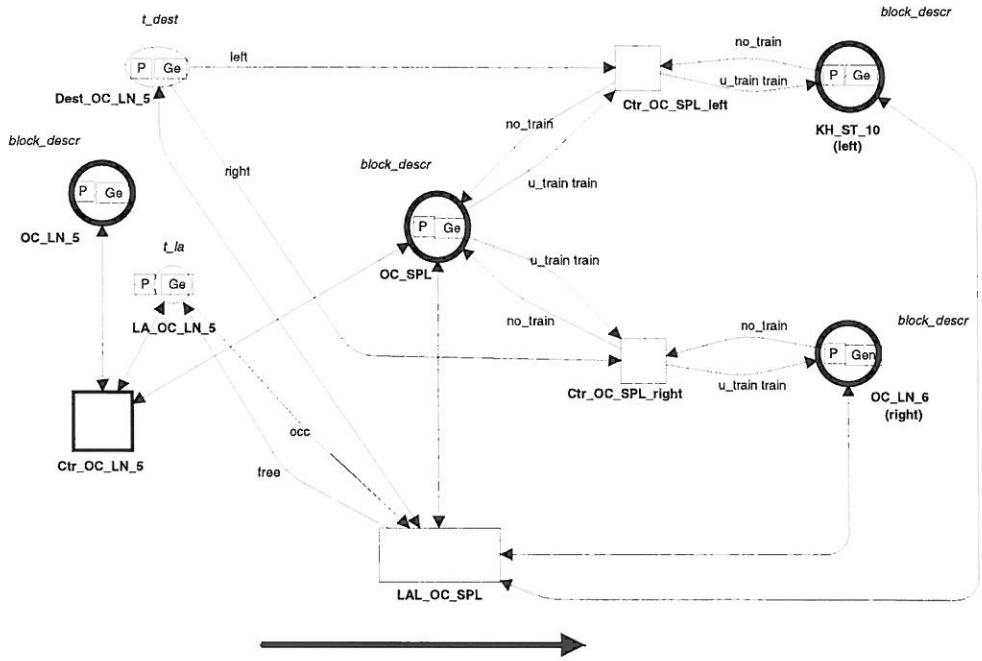



Figure 7: Net model of a two–way deterministic branching switch

switches, and vice versa as, for instance, at both sides of the kicking horse pass station. All it takes to operate these switches both ways is to equip the respective nets with the controls of both figs. 5 and 6.

# 3 Modelling the Kicking Horse Pass

With the net components as introduced in the preceding section, it is fairly straightforward to compose larger subnets which model, say, complete circular tracks, including stations and switch configurations through which trains may change main tracks, and the necessary controls.

The most interesting of these tracks is the kicking horse pass as it allows trains to travel in both directions and its station, besides being connected to the pass itself, includes entries from and exits to the other two main tracks (compare fig. 1). Moreover, train movement over the track must be organized so that

- there are never two trains in the track that go in opposite directions (to prevent deadlocks);
- there may be several trains moving in the same direction over the pass (each occupying one of the six track sections);
- some measure of fairness is enforced which prevents the monopolization of the track by trains going in one direction, while trains trying to go in the other direction are starving.

Controls to this effect may be realized by means of ordinary Petri–nets attached to the station model as they merely require keeping track of the number of trains that have left the station to cross the pass in one of the two possible direction (and have not yet returned) [Kl98].

Fig. 8 shows the net model of the complete pass, with the station abstracted to a substitution transition which is connected to places modelling

- the pass sections KH_LN_1 on the left and KH_LN_6 on the right;
- the sections KH_ST_10 on the left and KH_ST_20 on the right through which trains may move in from and out to both the inner and outer circles;
- the sections KHIC_LN and ICKH_LN to which trains may proceed past KH_ST_10 on their way out to the inner circle and from which they may enter section KH_ST_20 on their way in from the inner circle, respectively.

The types of tokens that are actually in these places are used by the controls inside the substitution transition to decide which of the trains in these sections or in one of the sidings inside the station gets permission to proceed.

The track over the pass is modelled by the six places KH_LN_$i$ representing the six sections and by the six substitution transitions CTR_KH_LN_$i$ which pass train tokens along the track in both directions.

The internal structure of the HS–transition that substitutes for the kicking horse station is depicted in fig. 9. It interfaces with the surrounding places through the substitution transitions KH_ST_0 and KH_ST_9 which represent the five–way switch configurations through which trains may enter and leave the station on both sides. Both transitions are interconnected through four places which model various aspects of the occupation by trains of the five sidings inside the station. In particular, the place

- KH_ST_[12345] contains tokens inscribed with `train_descriptors` paired with the indices of the sidings which they actually occupy;

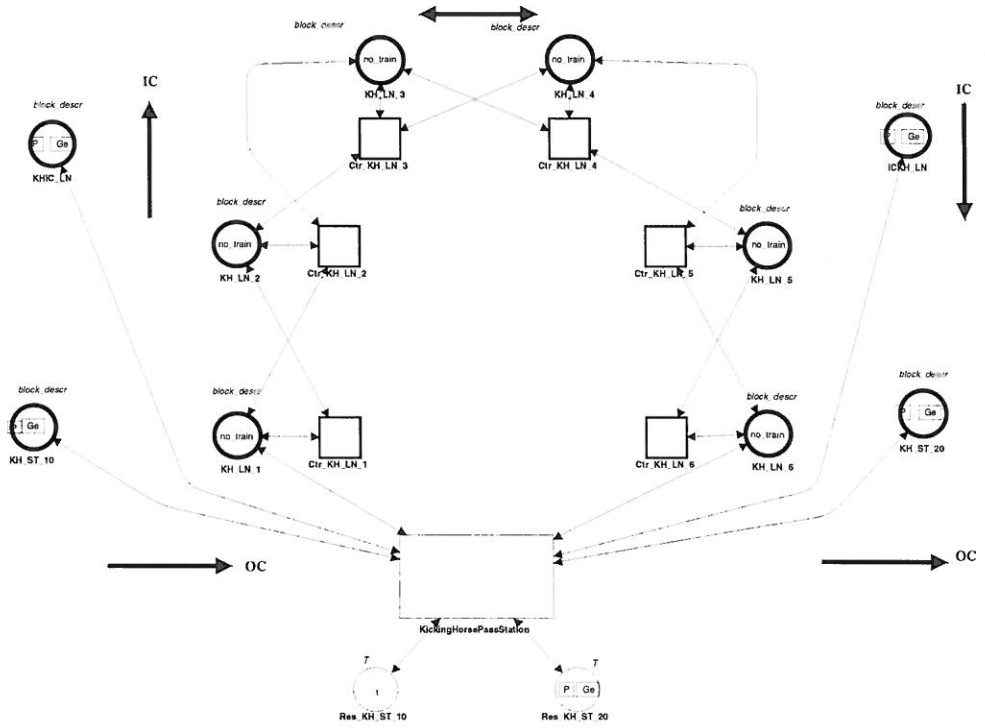



Figure 8: Net model of the kicking horse pass track

- RES_KH_ST_[12345] contains tokens inscribed with indices $i \in [1 \ldots 5]$ which represent free sidings;
- DEST_KH_ST_[12345] contains switch selector tokens paired with siding indices which determine in which direction trains in the particular sidings have to leave the station (up the pass or alternatively through section KH_ST_10 on towards the inner circle when going clockwise or through section KH_ST_20 and on towards the outer circle when going counterclockwise);
- LA_KH_ST[12345] contains look–ahead tokens generated by either of the HS–transitions which determine the siding(s) from where train(s) may leave the station next (with a train each being able to leave concurrently in either direction);

Four more places RES_KH_ST_ 0|10 | 20| 9 for plain tokens are necessary to ensure that trains arriving from or leaving towards the inner or outer circles have free passage through the track sections KH_ST_10 and KH_ST_20 , and that incoming trains from either of these circles find free sidings before entering these sections in order to prevent potential deadlocks with trains trying to use them in the opposite direction. These places are also connected to controls, specifically look–ahead subnets, for the respective parts of the inner and outer circle net models (not shown here) with which the HS–transitions KH_ST_0 and KH_ST_9 need to interact.

Another important part of this net model is the subnet contained in the substitution transition at the bottom of fig. 9 which exercises control over the direction in which trains are permitted to move across the pass and also enforces fairness between trains trying to go in opposite directions.

Roughly, these controls work as follows: A conflict between trains trying to enter the empty track in the same or opposite direction(s) is generally resolved by arbitration. Once permission is given to a train to leave the station in one

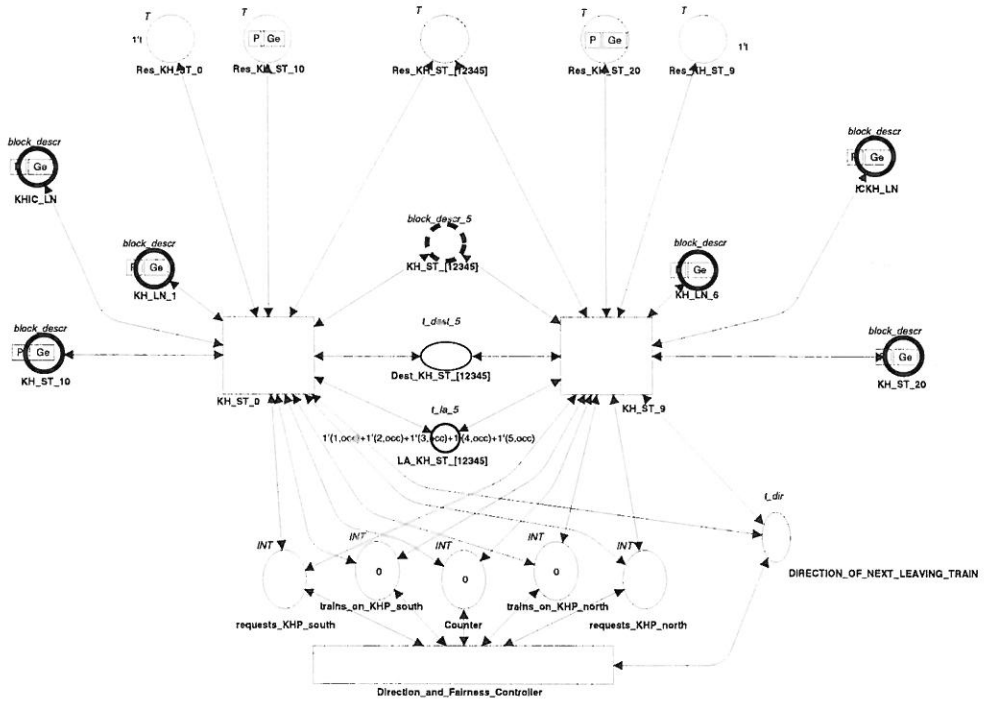



Figure 9: Net model of the kicking horse Pass station

direction, a lock is set for trains waiting to go over the pass in the other direction. A train trying to enter the empty track without competitor in either direction gets permission to leave immediately, thereby again locking the other direction. Once a train moves along the track, more trains may follow in the same direction, while the other direction remains blocked. Fairness is enforced by putting an upper limit on the number of trains which in one direction may move over the pass in succession. If this limit is exhausted, further trains are blocked. As soon as all trains have left the track, the direction is reversed, and trains waiting in the station to go in the new direction are given permission to do so. Directions may also freely be changed if the track is empty and neither of the limits is exhausted. This measure in fact establishes a finite synchronic distance between trains going in opposite directions [GeLaTh80, KlLa82].

To this effect, the DIRECTION_AND_FAIRNESS_CONTROLLER transition is connected through a place for tokens of type (color) `t_dir` and five places for tokens of type `t_integer` with the HS–transitions that control entrance to and exit from the sidings for trains scheduled to go over the pass. These places serve the following purposes:

- the token in DIRECTION_OF_NEXT_LEAVING_TRAIN, as the name indicates, specifies the direction in which the next train is permitted to move across the pass;
- REQUESTS_ON_KHP_CLK | _CO_CLK keep track of the number of trains requesting entry into the track in clockwise and counterclockwise direction, respectively;
- TRAINS_ON_KHP_CLK | _CO_CLK follow up on the number of trains that are actually in the track in either direction (one of these counter values must always be zero);

- COUNTER counts the trains which in succession have left the station in the direction indicated by the token in place DIRECTION_OF_NEXT_LEAVING_TRAIN (of which some may already have completed their passage over the pass, others may still be in the track).

A control structure similar to the one that needs to be realized inside the DIRECTION_AND_FAIRNESS_CONTROLLER transition is, on the basis of an ordinary Petri–net model, described in [Kl98].

In a larger context, the complete kicking horse pass model can be abstracted to another substitution transition KICKINGHORSEPASS which interfaces with its surroundings, the inner and outer circle net models, essentially via the places KHIC_LN, ICKH_LN and KH_ST_10, KH_ST_20, respectively. These interfaces must, of course, be complemented by small subnets which model the look–ahead controls for the passage of train tokens across these places. However, as these subnets feature basically the same elements as in figs. 5, 6 and 7, they have not been included in the nets of figs. 8 and 9 to keep them clear of details that add nothing new.

# 4 The Complete System Model

The net models for both the inner and outer circle tracks, including the stations, the inner and outer circle junctions and the two tracks by which they are interconnected, are quite similar to each other and slightly less complicated than the kicking horse pass model since trains (train tokens) are allowed to move only in one direction in both of them.

Fig. 10 shows, merely for illustration purposes and without further explanation, how the net for the outer circle looks like. Of primary interest for the construction of the full system model are the places KH_ST_10, KH_ST_20 and ICOC_LN_3, OCIC_LN_3 through which this net respectively interfaces with the kicking horse pass net and with the net for the inner circle.

Likewise, the net for the inner circle interfaces with the kicking horse pass through the places ICKH_LN (for trains leaving towards the pass) and KHIC_LN (for trains arriving from the pass) and with the outer circle through the places ICOC_LN_3 (for trains leaving) and OCIC_LN_3 (for trains arriving).

When abstracting both nets to substitution transitions OUTERCIRCLE and INNERCIRCLE, and using the HS–transition KICKINGHORSEPASS, the net model of the entire system can simply be constructed by overlapping the respective interfacing places of these three components, as is shown in fig. 11 (the look–ahead controls associated with the interfaces between the KICKINGHORSEPASS transition on the one hand and the OUTERCIRCLE and INNERCIRCLE transitions on the other hand are again omitted here).

There is some asymmetry, though, between the interfaces of the OUTERCIRCLE and INNERCIRCLE transitions with the KICKINGHORSEPASS, which is not visible on this level of abstraction. It relates to the chosen modus of operation for the track sections KH_ST_10 and KH_ST_20 which get involved in all train movements between the kicking horse station and both the inner and outer circles. Trains moving into this station from the outer circle or leaving it towards the outer circle may be stopped in these sections, whereas for trains that are being exchanged with the inner circle they are operated as no-stop sections. This implies that in the former case both sections may be allocated directly if trains demand entry and they are free, whereas in the latter case they must be reserved by look–ahead controls which ensure that subsequent sections are also free. The controls to this effect are in large parts integrated into the HS transitions KH_ST_0 and KH_ST_9 which model the five–way switches of the station (compare fig. 9).

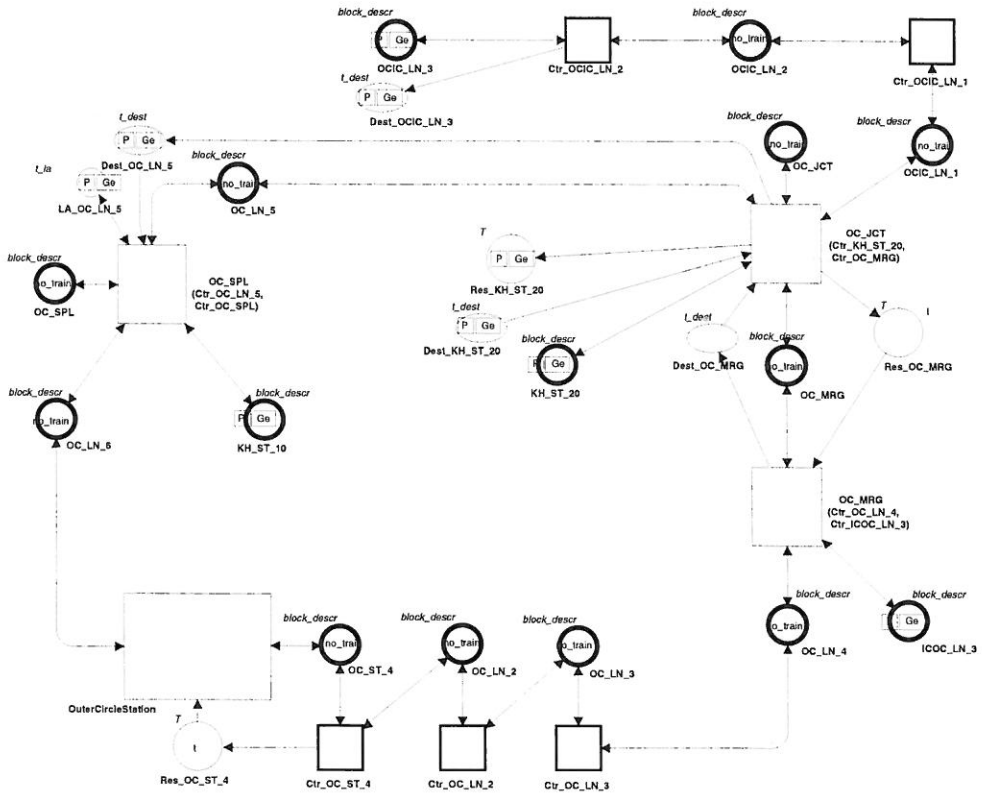



Figure 10: Net model of the outer circle track

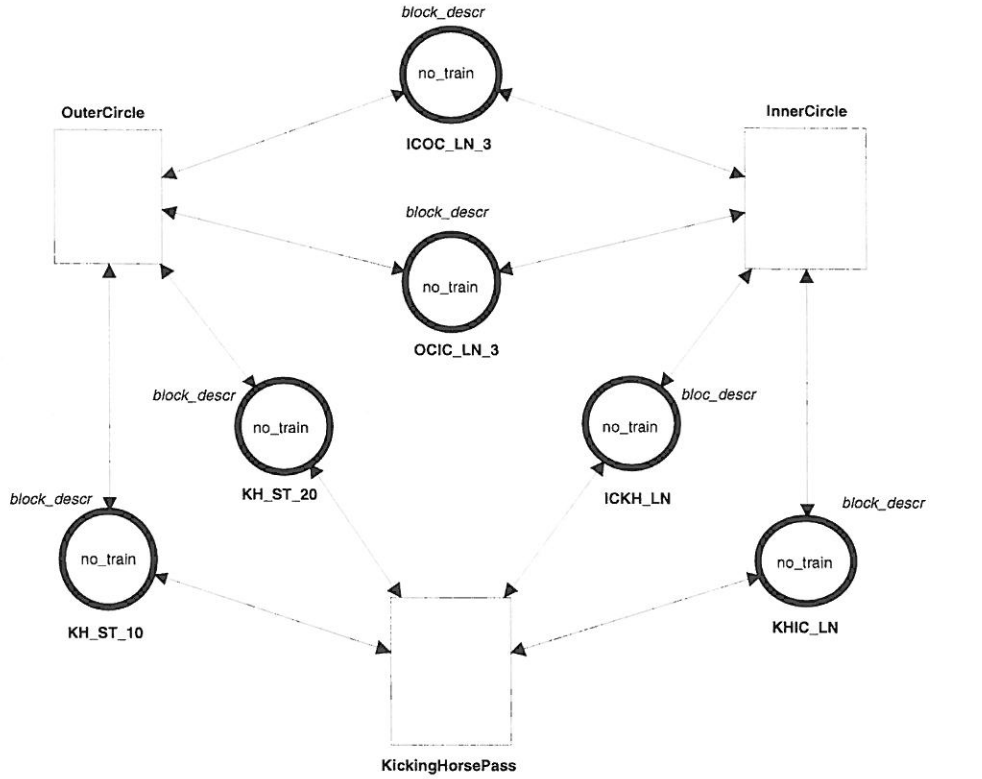



Figure 11: Complete net model of the entire track system

# 5 Discussion

The net model outlined in this paper was a first attempt to explore and analyze the dynamic aspects of the train system, in which several concurrent activities - the routing of up to ten trains as prescribed by individual travel plans - have to be coordinated 'on the fly'. Though not exactly a blueprint, it also served as a guideline for the development of a prototype control program, written in C, which runs the trains in the real system. This program builds on an internal representation, in the form of a large data structure, of the system state in terms of travel plans, actual train positions (sections). the acctual settings of switches and light signals, and of speed levels (voltages). The program itself consists of a set of C–functions which more or less directly implement the various substitution transitions. They are called upon receiving sensor signals triggered by trains crossing section boundaries to inspect and update the parts of the state representation actually affected and to issue the control data necessary to set switches and light signals, and to apply voltages to track sections.

As several people were involved in various aspects of the project (track design and construction, development of hard- and software, net modelling), and not all of them had a background in Petri nets, the basis for communication was the schematic track layout given in figure 1. In the early stages of the project, the CPN model helped to identify fairly quickly flaws in the conceptual design and potential trouble spots by generating extreme system states (token distributions representing trains in track sections), rather than doing these tests in the real system where critical situations might be difficult and rather time–consuming to bring about and to reproduce. Also, a C implementation of software that controls concurrent activities under real-time constraints includes too many low–level details to permit an abstract view of the (idealized) system dynamics. In contrast, translations between the CPN model and the track layout are straightforward, i.e., problems found in the model could be communicated using the layout, and issues unclear in the layout could be investigated in the model.

A major issue that cannot be properly addressed using the track layout alone is the identification of potential deadlocks (in the non-technical meaning of the term). The layout could easily be mapped to a condition-event net, but this would fail to account for the individual travel plans. In general, none of the trains "sees" the complete track layout, but rather each of them has its own partial view, or its own private net model, of the system as defined by its travel plan.

To convey the nature of the problem, consider as an example the condition-event net of figure 12 which models just the outer and inner circles without sidings and, in rudimentary form, the two interconnecting tracks. The token distribution depicts 6 trains on the inner circle and another 7 trains on the outer circle, and it is assumed here that no trains can move in or out of this particular subsystem (tokens may neither be added nor disappear).

Trains can obviously move along both tracks – though not very smoothly in reality – as there is one free section in each of the circles. However, if the travel plan of one of the trains on the outer circle would prescribe changing into the inner circle but none of the travel plans of the trains already in it would prescribe changing in the opposite direction, the last free slot in that track would be occupied, and all trains would come to a halt. Trains could still move along the outer circle, but if further trains would try to follow their travel plans into the – now blocked – inner circle, the deadlock would spread out over the outer circle as well. Thus, as far as these particular travel plans are concerned, there exists just the interconnecting path from the outer to the inner circle through the place OCIC_LN, but not the path through the place ICOC_LN, i.e., the inner circle constitutes a classical structural deadlock.

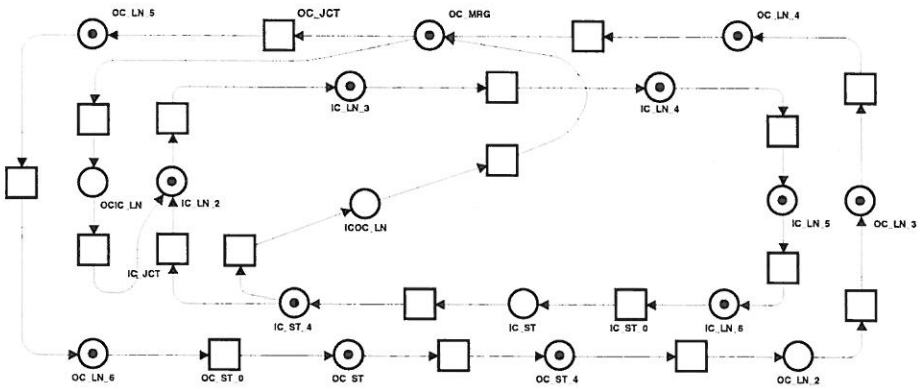



Figure 12: Illustration of a deadlock problem

The constraints imposed on the net by the travel plans do not necessarily increase the potential for deadlocks, even though they do decrease the available resources in terms of usable track sections. In our example, if the travel plans of the trains in the outer circle would not include the connection to the inner circle, the path through the place OCIC_LN would become invisible as well, leaving two separate and deadlock-free circles.

A conservative approach to deadlock avoidance would limit the number of trains on each track. With $n$ denoting the number of sections per track, this seems to allow for $n-1$ trains to move along each track at the maximum. In a worst case scenario, however, the individual travel plans could be such that all trains try to get into the same track at about the same time to make some turns there before leaving again. Therefore, the restrictions would have to include those trains that just pass through a track while following their travel plan, leaving only $min\{n_{IC}, n_{OC}\} - 1$ trains moving about inner and outer circle. The same consideration applies if the kicking horse pass is included, so that the total number of trains that can be guaranteed to move freely about the entire track system without causing deadlocks would have to be limited to $min\{n_{IC}, n_{OC}, n_{KH}\} - 1$. Other circular tracks of restricted capacity are formed by the interconnecting tracks between the inner and outer circles and the kicking horse pass, bringing down the number of trains in the complete track system to no more than about five trains.

With more trains, deadlock prevention requires that some restrictions be placed on specifying individual travel plans, e.g., by allowing each of the circular tracks to be used in at most $n_{track} - 1$ of them. With ten trains to deal with, this could for instance be accomplished by travel plans which allow three trains to use all three main tracks in any order, while of the remaining seven trains two each could cycle just over the kicking horse pass and just around the inner circle, and three trains could make turns just around outer circle.

Travel plans in which all ten trains use all three main tracks in some arbitrarily chosen sequences may or may not end up in deadlocks, depending on the order in which trains happen to get into critical situations. Preventing these deadlocks requires sophisticated global controls which must monitor actual distributions of trains over track sections and next actions prescribed by travel plans to decide which trains may proceed without getting trapped. This is still an open problem which requires further simulation and analysis to find a satisfactory solution.

Another source of deadlocks are trains that have completed their journeys and arrived at pre-defined terminal positions (station sidings). The simpler of these situations may occur at stations of the inner or outer circles, along which trains are moving in just one direction. If all three sidings in these stations are terminal

positions and the respective trains have ended up there, no other trains still on the move can pass through, i.e., they inevitably deadlock. The way out of this problem consists either in using only two of the three sidings as terminal positions, or in devising a travel plan for one of the three trains ending there which can be expected to terminate after all others trains have passed through.

Deadlocks that involve the kicking horse station, due to trains moving in both directions, are more difficult to prevent. There may be trains arriving in the station scheduled to go, say, counterclockwise over the pass while other trains are actually moving clockwise over the pass. If there are more such trains than there are sidings left in the station, the entire pass deadlocks. The same happens if a train arriving from the pass to end its journey in the station finds its terminal position occupied by another train scheduled to go over the pass in the opposite direction.

These situations could be brought about quite frequently by simulating various travel plans in the net model but also by running them in the train system itself. The remedies found sofar are not very satisfactory from a systems design perspective as they either require careful coordination of travel plans or putting rather conservative restrictions on train movement which often bring to a grinding halt all but one or two of the ten trains.

As all of this is only the prelude to the problems to be solved when designing and implementing the control software for the real system, the reader may wonder why the information gathered from the CPN model was not used to simplify the track layout. The rationale for this comes from the teaching context of this project: the system has to have rough edges for the modelling process to be interesting, and while minor corrections were made to the initial layout, most trouble spots had to remain, so that the students can employ their theoretical knowledge about Petri Nets in practice to find them. We do not claim that we have consciously designed the track system with all its problems in mind, but the current system seems to provide just the right level of complexity. The non-local interactions between its parts make it challenging even for good students, but a great deal can be achieved by finding the right abstractions and by looking for modular solutions.

# 6 Conclusion

The entire net model was designed by two graduate students (the first two coauthors of this paper) as a term project accompanying a graduate course on Petri–nets. Having neither been familiar with the Design/CPN tool nor having had any prior experience with system architecting and modelling, i.e., starting more or less from scratch, it took them about four months of hard work to complete the model and to run some simulations. A considerable part of this time was spent on getting used to handling the tool rather than on the net design itself.

One problem area is editing. It takes pages after pages (and diagrams and menu screen dumps) of a rather voluminous tutorial (which would take students the better part of a term to digest) to explain things that should be self–explanatory with a decently designed graphical interface; and although the manuals are available online, there is no help functionality integrated into the tool. Another problem is the need to switch back and forth between the editor and the simulator which takes way too much time (even for an early and incomplete prototype of the full net model it took some 10 minutes on a SUN–SPARC equipped with 48 MBytes of memory, 4 minutes if the memory was stepped up to 64 MBytes, and still 2 minutes on an Ultra–SPARC with 512 MBytes of main memory). For all practical purposes, this rules out going repeatedly through cycles of designing nets in incremental steps, testing and modifying them until an acceptable solution is found, as one would often wish (or need) to do it in early phases of architecting complex systems.

Executing the simulator in a stepwise manner is equally frustrating since changes of the token distribution are very hard to recognize on the graphical display and, what is even worse, each step requires several seconds, sometimes even minutes, to complete. It renders testing large nets in this way a time–consuming affair. Surprisingly, most of the time is spend in the graphic updates, whereas the simulation itself seems to be reasonably fast. As a consequence, automatic simulation has to be used in practice instead of the animated, interactive simulation.

These deficiencies render Design/CPN in its current form a tool that can hardly be used for teaching as students are easily turned off.

What is also (as yet ?) missing are means to verify net designs with regard to essential safety and liveness properties. It would already help if the existence of S– and T–invariants could be verified by analytical methods at least on the level of plain Petri–nets, i.e., without regard for the constraints imposed by inscriptions (which would be partially sufficient in the case of our train system model and presumably also in many other areas, e.g., process coordination languages [CaGe89, Ass95]). This being a problem which, for complexity reasons, may be hard to crack if the nets become large, one could alternatively think of simply checking whether or not invariants pre–specified by the designer do indeed hold.

While the improvements planned for the new CPN simulator (see the respective web page) seem to point in the right direction, doing away with at least some of the performance problems, it is not at all clear why memory demand still remains excessive, why compilation of complete nets still takes several minutes, and why users are not given the choice of compiling to an SML implementation of their own.

# References


[Ass95] Aßmann, C.: *A Coordination Language for Systems of Cooperating Processes*, Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95), pp. 738 - 747

[CaGe89] Carriero, N.; Gelernter, D.: *Linda in Context*, CACM Vol. 32, No. 4, 1989, pp. 444–458

[GeLa78] Genrich, H.J.; Lautenbach, K.: *The Analysis of Distributed Systems by Means of Predicate/Transition-Nets*, in: Semantics of Concurrent Computation, Lecture Notes in Computer Science, No. 70, Springer, 1979, pp. 123–146

[GeLaTh80] Genrich, H.J.; Lautenbach, K., Thiagarajan, P.S.: *Elements of General Net Theory*, in: Net Theory and Applications Lecture Notes in Computer Science, No. 84, Springer, 1980, pp. 21–163

[GeLa81] Genrich, H.J.; Lautenbach, K.: *System Modelling with High–Level Petri–Nets*, Theoretical Computer Science 13, 1981, pp. 109–136

[Jen90] Jensen, K.: *Coloured Petri Nets: A High-Level Language For System Design and Analysis*, in: Rozenberg, G. (Ed.): Advances in Petri Nets 1990, Lecture Notes in Computer Science, No. 483, Springer 1991, pp. 342–416

[Jen92] Jensen, K.: *Coloured Petri Nets*, Monographs on Theoretical Computer Science, Springer 1992

[KlLa82] Kluge, W., Lautenbach, K.: *On the Orderly Resolution of Memory Access Conflicts among Competing Channel Processes*, IEEE–TC Vol. C–31, No. 3, 1982, pp. 191–207

[Kl98] Kluge, W.: *The Kicking Horse Pass Problem*, to be published in Petri–Net News Letters

[Po95] Pole, G.: *The Spiral Tunnels and the Big Hill*, Altitude Publishing Canada Ltd. 1995

[Rei85] Reisig, W.: *Petri Nets*, EATCS Monographs on Theoretical Computer Science, Vol. 4, Springer, 1985

# TECHNICAL ISSUES IN MODELLING THE EUROPEAN TRAIN CONTROL SYSTEM (ETCS) USING COLOURED PETRI NETS AND THE DESIGN/CPN TOOLS

L. Jansen, M. Meyer zu Hörste, E. Schnieder

Technical University of Braunschweig,
Institute of Control and Automation Engineering,
Langer Kamp 8, D-38106 Braunschweig, Germany

Email: {jansen | meyer | schnieder}@ifra.ing.tu-bs.de

**Abstract**

At the Institute of Control and Automation Engineering Design/CPN has been used to model the European Control System (ETCS) within a project for the Deutsche Bahn AG (German railways). This paper reports of experiences in modelling this complex, distributed automation system using Coloured Petri Nets and the Design/CPN tools. We will concentrate on some technical issues. However, for motivation we will give a brief overview of the application and will describe the modelling paradigms that we applied.

**Keywords:** European Train Control System (ETCS), Modelling, Scenarios, Distributed Simulation, Communication, Synchronization

## 1 Introduction

### 1.1 The Aims of the Interoperable European Train Control System (ETCS)

The process of European harmonization is still in progress, which also concerns the railway companies. Each European railway company has one or more train control systems, which are mostly incompatible. So every train which shall run through different countries has to be equipped with several systems or the train traction units and drivers have to be changed at every border. As a consequence, the first solution requires a complex system on-board, which leads to high installation and maintenance costs. The second one is a time consuming solution, which leads to increased operational costs. Thus, it is important to define a train control system which is standard for all countries and provides a uniform and language-independent signalling information for the man-machine-interface (MMI). [Frosig95]

Moreover, the increase of new developed trains' maximum speed demands a communication-based train control system, because at a travelling speed of more than 160 km/h the trackside

signals aren't surely recognisable any more. So for high-speed railway service the application of a communication-based train control system is necessary.

The European Train Control System (ETCS) matches these two targets: It is a communication-based train control system, which is useful for both high and low speed railway services and it defines a standard for a uniform signalling system on an MMI (Man-Machine Interface), so neither locomotives nor drivers have to be replaced when crossing borders. The textual display on the MMI can be made to comply with the driver's language.

The ETCS equipment consists of two main subsystems: Speed supervision and brake intervention are performed by the on-board system. The permission to run is given by the trackside system called "Radio Block Centre" (RBC) within a movement authority (MA). The task of this second system is to ensure that all trains in the related area are keeping the safety distance to the neighbouring trains. Also the supervision of special operational procedures like joining or splitting of trains is a task of this system. For lines with different capacities three application levels with different types of trackside and on-board equipment are possible.

# 2 Modelling the ETCS with CP-nets and Design/CPN

## 2.1 Modelling Paradigms

Modelling of control systems starts from different modelling paradigms. Before starting to model it is necessary to make sure that the methodology is suitable for achieving the modelling objective. A specific model for system functionality will be quite different from a model used for availability considerations. In modelling the European Train Control System, different modelling aspects have been integrated [JaLePtSc97]:

- components
- scenarios
- functions,

are shown on different model levels.

When modelling the component view, the focus is on communication and interaction of different subsystems. A general representation of these nets shows the subsystems and their interfaces, every subsystem being detailed on additional levels. In [JaLePtSc97] this was called the process aspect.

The scenario-based view is the modelling of operational procedures. Its main elements are the interaction between on-board and trackside equipment and the sequence of events required to maintain operation. Individual scenarios are connected to form groups and in this way they are integrated into the component model.

The functions are represented at lower model levels. The functions are specifically associated with the objects of the process aspect and represent the activities or the response to interaction requirements following from the scenarios. Some functional modules can be used in different objects and are hence modelled in so-called "functional blocks". These functional blocks are modelled as separate nets and can serve as functions in the different scenarios, without infringing

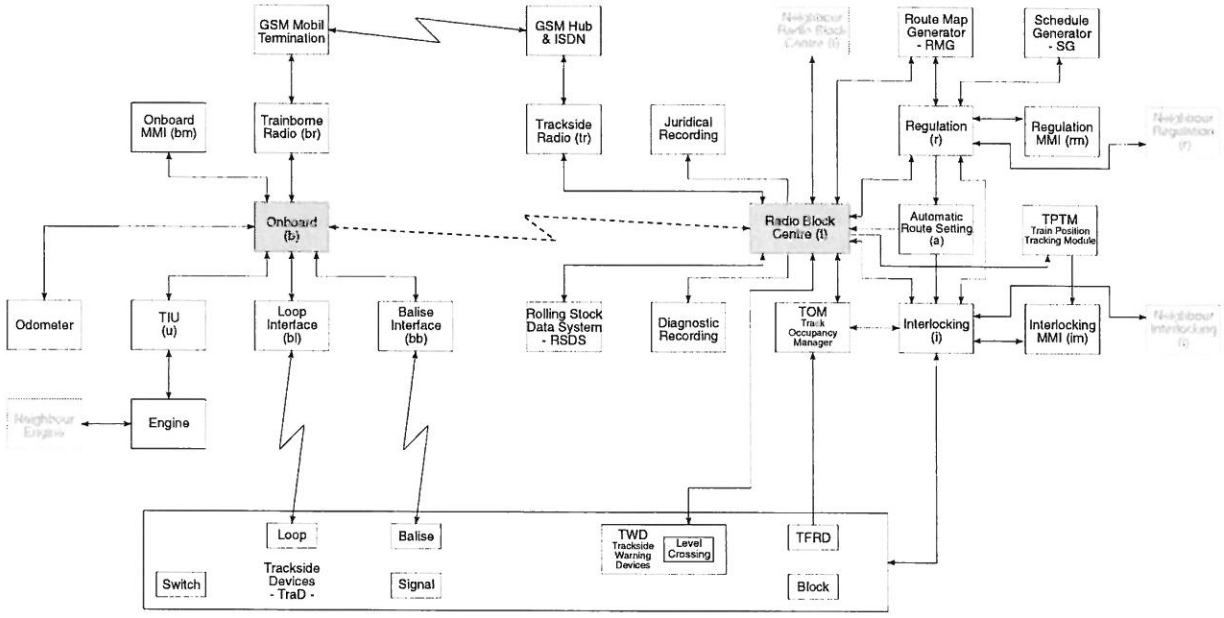



Figure 1: System Architecture of the ETCS

the modelling principle of hierarchic decomposition. This at the same time ensures that sub-nets and implementation components can be reused.

Systems modelled in this way by necessity are modular in structure – the hierarchical aspect being maintained at the same time. Because of their structure, models are easier to comprehend, easier to adapt, and they can be reused. Partitioning aspects can be considered, and symmetries can be utilised.

## 2.2 System Architecture

The goal of the first phase of our project was to model the two main components of ETCS, the onboard system and the radio block centre. Special emphasis was laid on the communication between both systems, that should be regarded according to both the more abstract interaction, represented by the sequence of telegrams, and the more functional aspects, represented by the data inside the telegrams.

Besides the onboard system and the radio block centre, there are more than a dozen of other components directly interfacing to one of the above main components or being related to them in some other way. For some of those systems, preliminary models have been built, but apart from a basic environmental model of the train none of them is fully executable by now. For an overview of the overall architecture of ETCS and the main components see figure 1. Further information on the topic can be found in [JaLeMeSc97].

For simulation purposes, we substitute the missing components by dummy solutions or by manually providing the missing stimuli directly to the interfaces of the receiving component nets.

ETCS is a modular and communication based approach to train control systems. Communication plays an important role in ETCS and therefore communication systems should be regarded as any

other kind of system with respect to structure, functionality and dynamics. By now, we modelled communication only with respect to the application layer of the well known ISO/OSI protocol stack. We did not actually model the underlying communication systems due to the restricted scope of our project, but the main component modells are designed to be easily adaptable and a general framework for later supplementing of separate models of underlying communication systems to our distributed modelling is provided. This opportunity is indicated in figure 1 for the refinement of the radio link (between the onboard system and the radio block centre) by adding a radio subsystem and a GSM modul at each side.

## 2.3 Vertical Decomposition of Component Models

Structuring a component model is important for readability, maintainability and last but not least for acceptance by the user. Integrating different aspects of a system description and combining formerly isolated views on modelling is a demanding challenge for working with Petri nets, although Coloured Petri Nets and Design/CPN provide a good basis for modelling of complex systems.

For structuring the main component models of the onboard system and the radio block centre, we adopted a layered approach, proposed by [JaLePtSc97]. Dynamics and functionality are described while distinguishing between scenarios and functions. Scenarios show the behaviour of a system in it's environmental context, especially regarding the technical process and railway operations. Functions are used to process input data that has been received from external components or internal sources. In contrast to scenarios, functions do not regard processes in the environmental context and therefore can be used within arbitrary scenarios.

Functions in our sense are not restricted to the mathematical notion but can represent processes with internal states and transitions as well. To be more precise, we could call them complex functions or tasks, but we will stick to the name and instead will speak of mathematical functions or ML functions, respectively, when we have in mind the mathematical notion or their implementation.

### 2.3.1 First Approach: Integrating Scenarios and Functions

In the beginning of the modelling we aimed at representing the application of functions within scenarios explicitly on the scenario level. Our intuition was to represent the invocation of a complex function graphically in the representation of a scenario, analogous to a function call in a functional or procedural language for sequential programming. We tried to model this aspect by making use of the hierarchy concept of Design/CPN in a straight forward way. The corresponding hierarchy or vertical decomposition, as we prefer to call it, is depicted in figure 2.

First of all, there exists a top level net $I$ for each component model. It shows the connections of a component to other components and the corresponding (unidirectional) communication channels on an abstract level. It is possible to substitute the terminating transitions by CP-nets, that are models of the neighbouring components. But as for a distributed modelling we prefer to substitute them with interfacing functions, that realize the communication with those models implemented separately.

On the next level we have a net $D$, that represents the decomposition of the component model

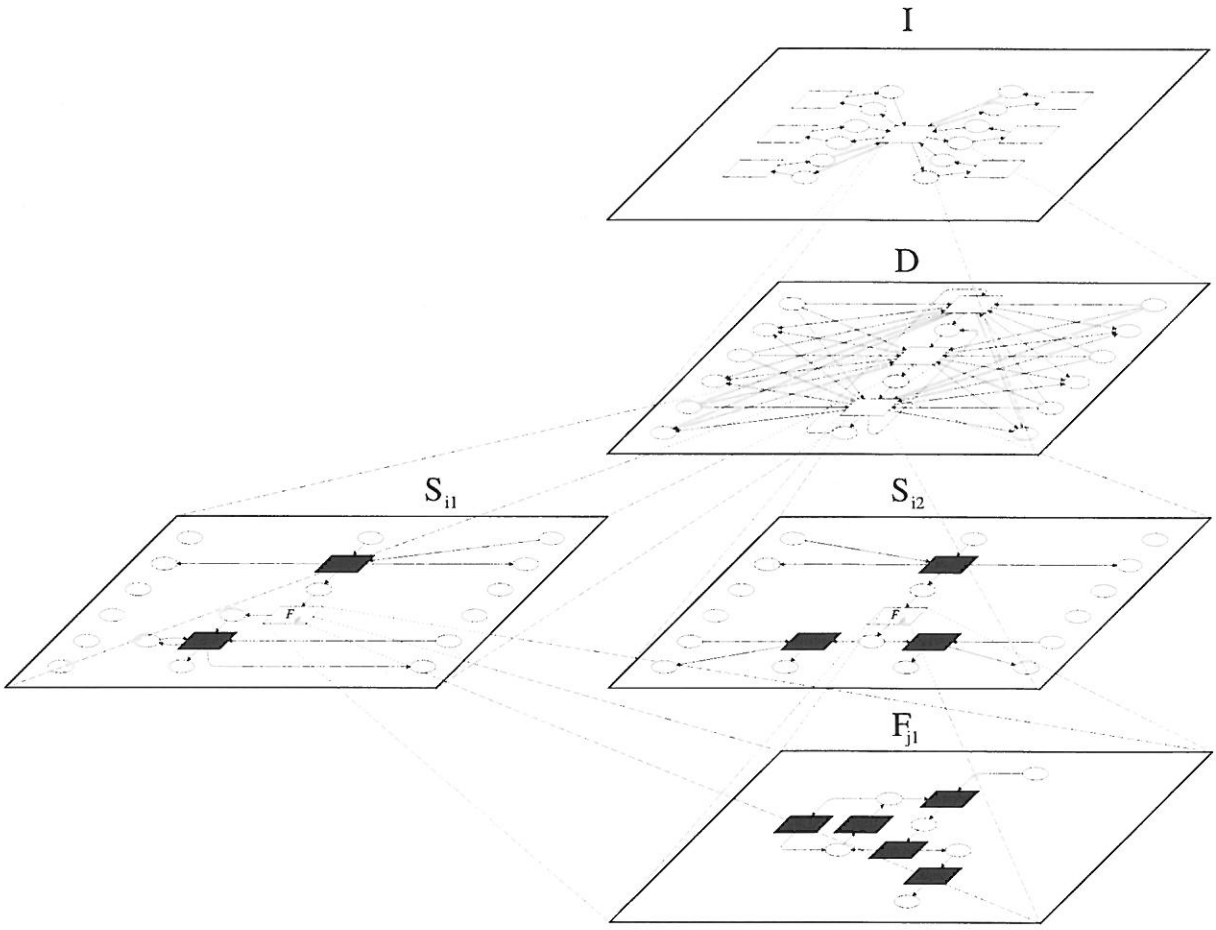



Figure 2: **Generic structure of the main component models (first approach).** Substitution transitions are depicted white, elementary transitions black and places white. Port places are not marked up, but can be easily identified by comparing the net structures.

with respect to relevant scenarios. This decomposition may be reached traversing several sublevels, that are combining scenarios to so called *scenario groups.*

The two bottom levels are used for representing scenarios or functions, respectively. A net, representing a scenario $S_i$ is called a *scenario net.* A net representing a function $F_j$ is called a *functional block.* Both may be further decomposed. A scenario net $S_i$ that makes use of a function $F_j$ contains a substitution transition, establishing the connection to the corresponding functional block.

Later on, this approach has come out to be against our intuition, as the semantics of hierarchical refinement within Design/CPN is of an instantiating kind. That is, if in the Design/CPN editor the same subpage, e. g. the top level net of a functional block $F_j$, is connected to several superpages (each with one or more corresponding supernodes), e. g. the scenario nets $S_{i1}$ and $S_{i2}$, the effect after switching to the Design/CPN simulator is that the subpage is instantiated with the number of supernodes, resulting in a 1:1-relation between the supernodes and instances of the corresponding subpage. For further details concerning the concept of hierarchy in Design/CPN

see [Jensen92].

At a first quick glance, we identified the problem as a need of sharing a local process by a number of other processes, sometimes even at the same time. The relation between the shared process and those latter processes is of an orthogonal kind and, thus, unusual for hierarchical decomposition. Within Design/CPN the problem can be solved in two ways:

- declaring all local places as fusion places in the shared net with the effect of synchronizing all its instances;
- not sharing the common net but implementing the calling and returning of the functional block via communication over a special fusion place.

The decision was made for the second solution, mainly because of the extensive use of fusion places required for the first solution, and second because of tool problems with Design/CPN 2.0, encountered at that time, with managing a restricted number of fusion places.

We think that with respect to the point of integrating (complex) functions and scenarios in a Petri net model, a semantics of shared subtrees would better accomplish our needs in this special case, although it probably would compromise the strict hierarchical decomposition paradigm. However, analyzing the abstract nature of this "irregularity" in terms of hierarchical decomposition is an open matter, important for integrating different views and paradigms for modelling with hierarchical Petri nets in general.

### 2.3.2 Improved and Extended Approach: Coordinating Szenarios and Functions

The improved generic structure of the main component models is depicted in figure 3. According to the afore mentioned problem of integrating scenarios and functions, and aiming at the prefered solution, we adopted the generic structure. Now functions are not any more subordinated to scenarios but, speaking in terms of structure, coordinated to scenarios. The dynamic coordination is managed by means of a newly introduced fusion place. This place serves as an internal message channel for the calling of complex functions and returning results to scenario nets.

Comparing with the first approach, the model was restructured mainly effecting the decomposition level formerly denoted as $D$. Besides the decomposition of scenarios $D_S$ the decomposition of functions $D_F$ is now represented explicitly. The net $D$ is used to distinguish between these two decomposition structures.

In the course of this restructuring we have introduced a relaxation of another modelling constraint, that a function net should not be directly connected to the component's interface. The schematic representation of the function $F_j$ slightly differs from that one in figure 2 in adding the interface places to the function net. This shall indicate, that a function may directly receive/send data to/from the component's interface.

For pragmatical reasons we admit a paradigmatic shift concerning the relation between scenarios and functions. Functions $P_k$ for preprocessing incoming messages and functions $P'_k$ for postprocessing outgoing messages that both are closely related to a certain interface $k$ are represented separate from the other application functions $F_j$. For this purpose an additional layer was introduced between the top level and the decomposition level. The corresponding net $A$ now represents all the functionality and dynamics of a component.

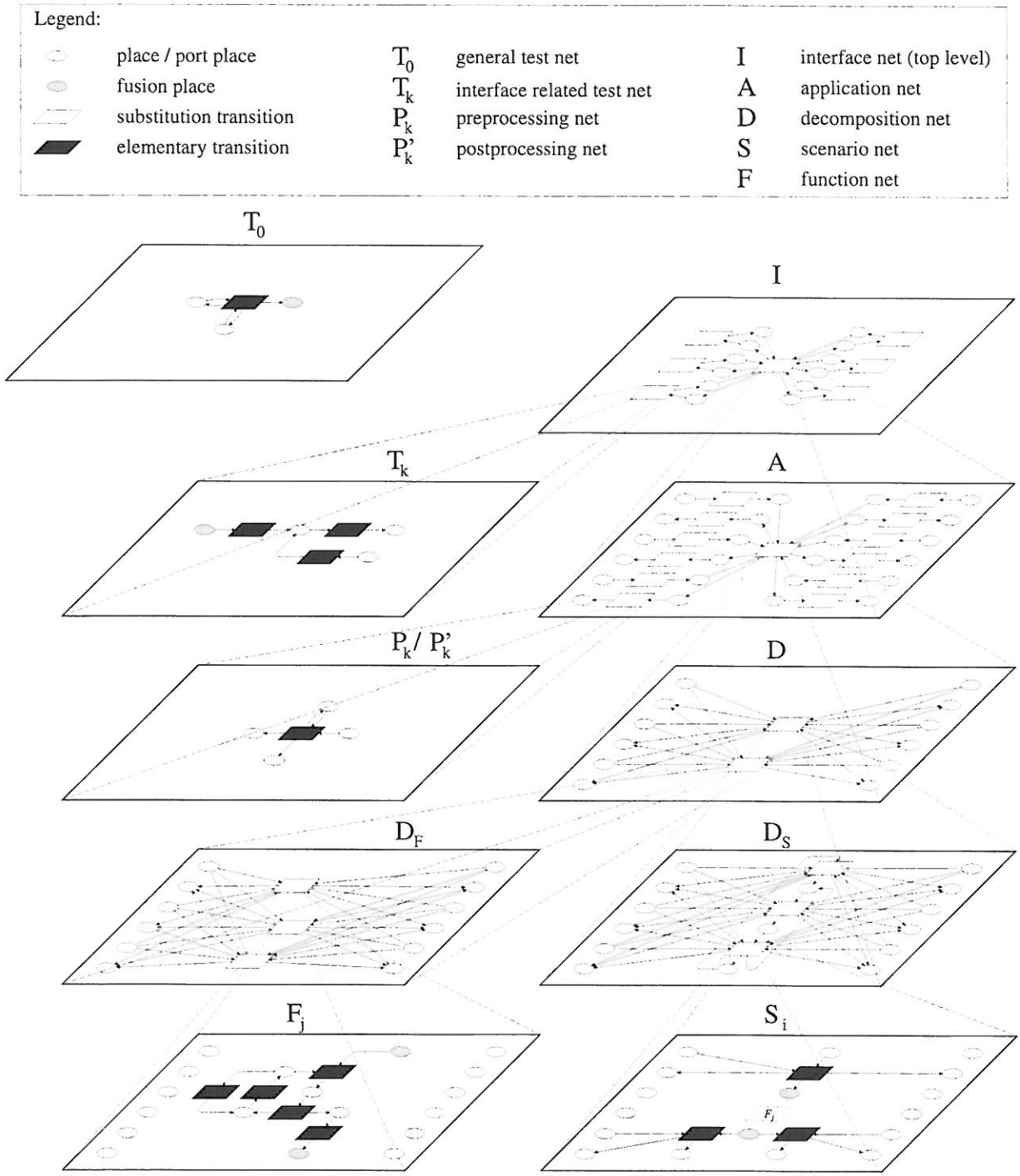



Figure 3: **Generic structure of the main component models (improved and extended approach).** Substitution transitions are depicted white, elementary transitions black, fusion places grey and other places white. Port places are not marked, but can be easily identified by comparing the net structures.

Moreover, the nets $T_0$ and $T_k$ have been added for preliminary simulation purposes. The latter nets $T_k$ substitute the terminal transitions of the top level net, i. e. one for each neighbouring component or its corresponding interfacing system, respectively. They implement special testing interfaces for producing stimuli, i. e. incoming messages, with respect to corresponding interfaces of certain neighbouring components. The net $T_0$ is used to coordinate the nets $T_k$ with regard to a sequence of stimuli, that can be manually defined in $T_0$. The stimuli are executed step by step while incrementing the simulation clock after each stimulus has been processed.

# 3 Technical Issues

## 3.1 Distributed Modelling

In terms of nets, a *distributed model* can be viewed as a Petri net whose net graph is not connected. There are pairs of nets with no path from one net to the other net and vice versa, their sets of places are disjunct and they have no fusion places in common. We call these nets *component nets*. The easiest way in order to obtain a distributed model is to cut the *whole net* at places, where the modelled system can be decomposed into components. Components are weakly connected (i. e. by communication) and do not share memory or control flow. For the cutting purpose, those places should be selected which represent communication channels at the border (interface) of a component to its surrounding. (Remark: Throughout this paper we only regard channels for unidirectional point-to-point communication). The selected places first are duplicated. Then it is decided, which place shall belong to which component net. The places should be renamed according to their assigned component while the arcs from or to transitions of the opposite component net have to be deleted.

## 3.2 Distributed Simulation

For a distributed simulation, we execute the separate component process models by means of separate simulation tasks. The simulation tasks themselves are programs that are run on operating systems and underlying hardware. So, first of all, we have to distinguish between the two words: process and task. We use the words in the way that we call the execution of a CP-net within the Design/CPN simulator a (simulation) task whereas the sequence of occurrences of transitions in a CP-net model is called a process.

The above component CP-nets can be separated by assigning each component net to a Design/CPN diagram. Thus, loading these Design/CPN diagrams to different Design/CPN tasks and starting them to simulate, we get a *distributed model* running on a *distributed system*, e. g. a cluster of workstations. As we formerly noted, the whole net was cut at the component's interfaces. Thereby the communication channels in the whole model were broken up and, in addition, the component models were assigned to different simulation tasks. As a result, we have to supply a communication mechanism on the level of Design/CPN simulation tasks, enabling two Petri nets to exchange data while being executed. We call this *implemented communication*. It must not be confused with the *modelled communication* being subject to the modelling of the application.

As a second aspect, distributed systems are generally characterised by the absence of a *global time*. Time is measured by clocks belonging locally to the tasks. There may be several kinds of

clocks concerning hardware, operating system and simulation tasks, each measuring a different concept of time. Tasks are distributed over different hardware, they run on different operating systems, they show different work load and share their resources (processor, memory, communication) with other tasks. Thus, we have to deal with the effect that distributed simulation tasks generally will not run synchronously without additional measures. This raises two questions: In which cases is synchronization of simulation tasks necessary? And if so, how can it be done? These questions shall be answered in the section about synchronization mechanisms.

## 3.3 Communication between Seperate CP-Nets

In order to have separate CP-net simulation tasks communicating with each other in a distributed simulation run, a family of basic communication protocols was defined.

The protocols offer three service primitives:

**SETUP:** for establishing a logical communication channel, generating the corresponding transfer file and setting up the connection to the other component model.

**SEND:** for sending a telegram to another system.

**TERMINATE** for closing down the connection.

A common protocol instance for all defined protocols is about to be realized for Design/CPN. Its implementation consists of seven hierarchical CP-nets and some ML routines. The protocol instance may be reused by defining a supernode for each interfacing component. Failures occurring during the execution of service primitives are detected and an error indication is returned to the service caller.

The protocol instance has been implemented and successfully tested for a first simple protocol, enabling a static point to point connection without the need of setting up or terminating the connection.

Generally, the communication is implemented via reading and writing to a common file system, accessible for all simulation tasks. Logical communication channels are mapped to physical files in the file system. These files are called *transfer files*. Some further files are used for synchronizing the protocol instances for more advanced protocols which are not implemented yet.

As a means of communication files have been prefered for two main reasons: First, the communication via files can be easily observed externally and can be specificly manipulated for simulating communication errors, and second, nearly every simulation tool offers a run-time interface to the file system. Thus, transfering of data via files opens the opportunity to implement the protocol for other tools so as to enable a combined simulation in a diverse tool environment.

## 3.4 Synchronization

Regarding a distributed simulation with time, the aim of synchronization is to establish a common control over the distributed flow of simulated time. The simulated time is measured by clocks, one for each simulation task. These clocks now have to be addressed as *local clocks*, one for each component simulation being run on a Design/CPN simulator or any other simulation

tool. The synchronization mechanism depends on the representation of time. There are three alternatives:

1. no time
2. discrete time (normally with equidistant time intervals)
3. event time (with non-equidistant time intervals)

The first alternative leads to a causal model, not considering timing aspects at all. There is no need of a timed simulation neither for modelling nor for implementing the communication between Petri nets. Under these circumstances synchronization is no matter of concern.

The second alternative leads to a time based model. The behaviour of a system is computed for predetermined times and thus can be observed according to a schematic time grid. For a distributed simulation a common time base is defined the local clocks have to be synchronized with. For defining the time base a trade off between accuracy and efficiency is inevitable since both dimensions have their optima in the opposite extrems, maximum accuracy for a small time base and maximum efficiency for a huge time base.

The third alternative is mentioned for completeness but was not yet studied in our project and therefore will not be considered in this paper.

### 3.5 Synchronization Mechanism for Discrete Time

For the distributed model execution we developed a preliminary version of a synchronization mechanism for discrete time simulation, adopting the method of *barrier sychronization* (see [Lange97]) for CP-nets and Design/CPN.

The mechanism is based on a central process serving the global time to the local Design/CPN clocks. These clocks are maintained by the distributed Design/CPN simulation tasks. The synchronization of a Design/CPN simulation is realized by adding a special net page to each component net. This page contains a so called client process, that realizes the synchronization for the component net. The clients half of the synchronizing process is identical for each component net and apart from its synchronizing aspect does not interfere with the rest of the component net.

The mechanism consist of four steps performed cyclically as follows:

1. The server process sends a synchronization signal via a special synch message channel to the other simulation tasks.
2. The client process within each component simulation task polls to this synch message channel, until it receives the synchronization signal.
3. When there are no more transitions enabled for a client process under the current value of the local clock, its value is incremented towards the next discrete time. Among the newly enabled transitions there is at least one that makes the client process send back a synchronization acknowledgement to the server process and restarts the polling to the synch message channel for the next synchronization signal.

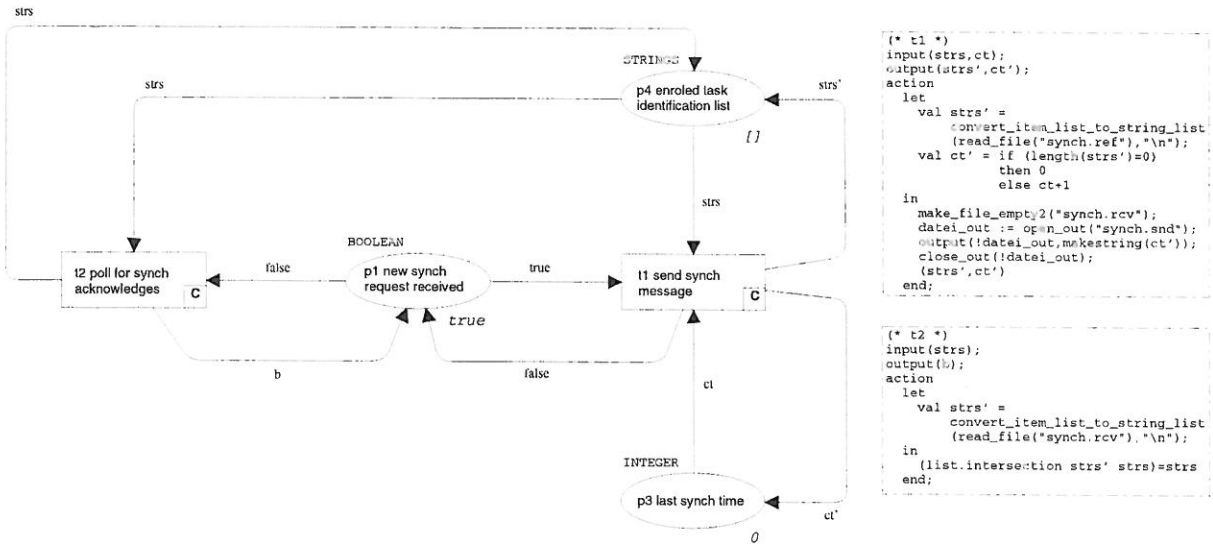



Figure 4: CP-net with server process for synchronization

4. The server process waits until the acknowledges of all client processes have been received and then sends the next synchronization signal. Thus the cycle begins from the start again.

An implementation of the server process for this synchronization mechanism with CP-nets is depicted in figure 4. The implementation uses three files: *synch.snd* as the synch message channel, *synch.rcv* for the synch acknowledgements of the client processes and *synch.ref* containing the list of identities of all enroled simulation tasks as a reference for comparison. The aim of the file *synch.ref* is to cope with dynamically adding and completing of simulation tasks.

The CP-net consists of two transitions and three places. Transitions $t1$ and $t2$ implement the first and fourth of the afore mentioned steps. In the initial state, the place $p3$ contains an initial token with default value 0 for the synchronization time. There are no enroled simulation tasks known, as represented by an initially empty list in place $p4$, and the place $p1$ contains a token with the value $true$, which means that a new synchronization signal has to be sent to all the enroled client processes.

The sending of the synchronization signal is done by transition $t1$. It updates $p4$ with the list of enroled simulation tasks read from the file *synch.ref*, deletes old synch acknowledges from the file *synch.rcv* and writes the new synchronization message to the file *synch.snd*. The message contains the actual synchronization time that is incremented from the last stored value in place $p3$ if there are any simulation tasks enroled at all. The transmission of the synchronization time with the synch signal is intended for later use regarding event time. Finally a token with the value $false$ is generated in the place $p1$ which disables transition $t1$ and enables transition $t2$.

The transition $t2$ cyclically reads (i. e. polls) the file *synch.rcv* for acknowledges from the client processes. An acknowledge consists of the identification of the corresponding simulation task. As each client appends it's task identification to the file *synch.rcv*, successful completion of the current synchronization request can be checked by comparing that file to the list of enroled task identifications in place $p4$. Transition $t2$ stores the result of this comparison in place $p1$, and in

case of successful completion the next synchronization cycle can begin.

It should be obvious, without giving the exact declaration of the used colorsets, that the implementation of the server process does not require timed simulation. However, for the client process, a timed simulation is necessary. As the net structure is quite similar to that of the server process, we will not give a graphical net representation of the client process. The list of identities of the enroled simulation tasks is not used for the client process. The transition, which is sending the synch acknowledge, has to generate a delayed token with an incremented time stamp.

# 4 Conclusion and Outlook

For the application of formal analysis methods to the ETCS, two models have been developed. The partition of the model follows the partition of the real system, both the on-board and the trackside system are represented each by one hierarchical model. Like the real system these models have interfaces where the data-exchange can be observed. The nets are divided into different types according to their position in the structure and their content. The component models are implemented separately, so different CP-nets may be simulated on separate workstations. A simulation runs through a sequence of scenarios which call several functions.

In the actual state the two models consist of circa 200 nets and 2500 transitions and places. The application logic is supplemented by some constructs which perform supporting tasks like time synchronization, stimuli generation or implementing communication between separate CP-nets. Also a smaller model has been developed, which is simulating the environment, i.e. the track, the physical train etc. Two more models of the interlocking and the regulation are still in working progress.

**Future Plans** The aim of our formal model is

1. to check the completeness of the specification of the ETCS,
2. to use it for a systematic derivation of test cases and
3. to evaluate at an early stage the specification of the european stardized interfaces of ETCS according to the national railway environment.

The future work will be directed to identifying ways of using the models for hardware in the loop tests with prototypes of the main components. Another aspect of intent is the derivation or automatic generation of code for an implementation of the modelled components.

# References

[Frosig95] P. Frosig. ETCS: Requirements to Seamless Cross-Border Operation. *International Railway Journal and Rapid Transit Review*, 9 (September), 1995, New York.

[JaLeMeSc97] A. Janhsen, K. Lemmer, M. Meyer zu Hörste, E. Schnieder. Migration Strategy for Different Level of the European Train Control System to Existing Railway Environment. In *Proc. of WCRR 97 - World Congress of Railway Research, Volume C: Power Supply, Signalling, Telecommunications and Non-conventional Systems*, pages 335–341, Florence, 1997.

[JaLePtSc97] A. Janhsen, K. Lemmer, B. Ptok, E. Schnieder. Formal Specification of the European Train Control System. In M. Papageorgiou and A. Pouliezos, editors, *Proc. 8th IFAC Symposium on Transportation Systems*, pages 1215–1220, Chania, 1997. IFAC.

[Jensen92] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* Volume 1. Springer-Verlag, 1992.

[Lange97] K.-J. Lange. On the Distributed Realization of Parallel Algorithms. In F. Plasil, K. G. Jeffery, editors, *Proc. of the 24th Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'97), Milovy, Czech Republic, November 1997*, pages 37–52. Springer, Berlin, LNCS 1338, 1997.

# Simulation Based Performance Analysis in Design/CPN

Bo Lindstrøm and Lisa Wells

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

**Abstract.** This paper describes the design of facilities for doing simulation based performance analysis. The performance facilities have three main components: functions for generating random numbers from different distributions, statistical variables for collecting different data while simulating, and reporting facilities for generating output from the statistical variables. We also describe the integration of the performance facilities into the Design/CPN tool. To illustrate the usability of the performance facilities, we give a non-trivial example of simulation based performance analysis by analysing a multiaccess protocol.

**Keywords.** Performance analysis, Coloured Petri Nets, simulation, random distributions, networks and multiaccess protocol.

## 1 Introduction

Most applications of Coloured Petri Nets (CP-nets) [Jen92,Jen94,Jen97] are used to investigate the logical correctness of a system. This means that we consider the dynamic properties and the functionality of the system. However, CP-nets can also be used to investigate the performance of a system, e.g., the maximal time used for the execution of certain activities and the average waiting time for certain requests. To perform this kind of analysis we often use timed CP-nets. A timed CP-net is a CP-net extended with a *global clock* (continuous or discrete) which represents the model time. Each token is now allowed to carry a *time stamp*. A binding element is *ready* when the time stamps of the removed tokens are less than or equal to the current model time. A binding element is *enabled* if it is colour enabled (enabled in an ordinary CP-net) and ready. The global clock advances when none of the colour enabled binding elements are ready. See Chap. 5 in [Jen94] for further details.

While the Design/CPN tool [Jena] supports state space analysis [CK97], timed simulations and functional analysis [CJ97], it lacked integrated support for performance analysis of a CPN model. Previously, all collection of data had to be explicitly defined and coded by the user. This, in turn, meant that the user had to be familiar with untimed statistical variables[1] and the use of code segments. An untimed statistical variable is a data type with which it is possible to collect some values and later on extract different statistical information such as sum or average [Jenb]. Examples of CPN models with explicitly coded performance analysis can be found in Chap. 2, 4, 12 and 15 in [Jen97].

The aim of this paper is to present the performance facilities [LWb] which have been integrated into the Design/CPN tool and which remedy the above shortcomings of the tool. These performance facilities provide distribution functions[2] for generating random numbers and high-level support for collecting statistics during simulation. Finally it is possible to create reports containing these statistics.

[1] Originally called statistical variables.

[2] In this paper a *distribution function* refers to a function returning a sample (integer or real) from the given probability distribution function.

When constructing a CPN model of real world phenomena, we are often interested in using random numbers from a specific distribution due to our knowledge of the behaviour of the modelled phenomena. The implemented distribution functions allow the user to draw random samples from several different kinds of distributions. These functions allow the user to construct, e.g. delay or input to the CPN model with values from a specific distribution. For example, when constructing a CPN model of the arrival of packages to a network we often assume that this arrival is Poisson distributed.

During simulation of a CPN model, we are often interested in evaluating the performance of the CPN model. To do this it can be necessary to extract different values from the markings of the CPN model during simulations. By using the statistical variables which are part of the performance facilities it is possible to collect these values from the CPN model while simulating. An example illustrating the use of statistical variables is a CPN model simulating a physical network. In such a CPN model we could be interested in knowing the average number of packages on the network per time unit. Hereby, we get a performance measure of the load of the network.

The performance facilities provide an easy way to specify the values from a marking and a binding element to be examined during a simulation. This can be done by using some of the predefined functions which measure, for example, number of tokens on a place or average length of the lists on a place. The user is also able to make his own functions to extract a value from a marking and a binding element of a CP-net to update the statistical variable. Other facilities such as inserting, deleting and initialising a statistical variable and maintaining a log file with the observed values during a simulation are also available.

This paper is organised as follows: In Sect. 2 we first give an overview of the incorporation of the performance facilities into the Design/CPN tool. Section 3 contains a model which will serve as an example throughout the rest of the paper. In Sect. 4 we briefly describe the distribution functions. The statistical variables are described in Sect. 5. In Sect. 6 we explain how to use the performance facilities. Finally in Sect. 7 we draw some conclusions including ideas for future work.

## 2 Performance Analysis in Design/CPN

In this section the incorporation of the performance facilities into the Design/CPN tool is described. The overall structure of the facilities is illustrated in Fig. 1. The items (except the item *Simulator*) illustrate the new performance facilities in the Design/CPN tool. The rectangles illustrate the main components of the tool while the dashed ellipses indicate the existing libraries which have been extended and integrated into the tool. The solid line ellipses indicate output produced by the performance facilities.

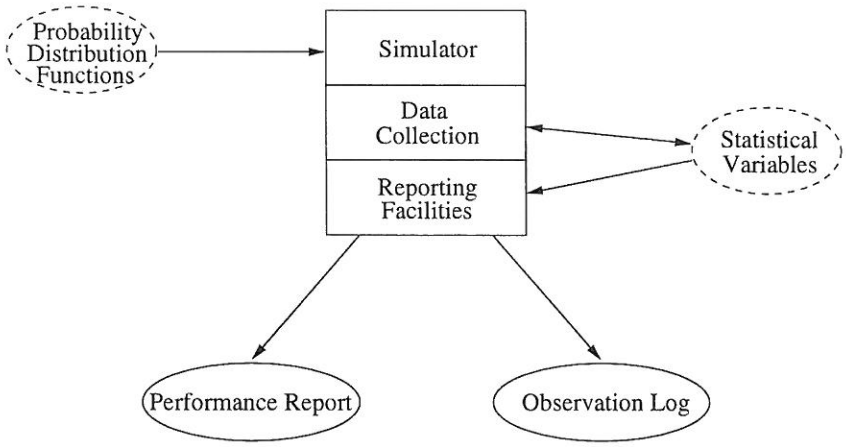



**Fig. 1.** Overview of the Tool.

The execution of the model in the Design/CPN tool using the performance facilities is as follows: the *Simulator* simulates the CPN model possibly using the *Probability Distribution Functions* for generating random numbers. At the same time, the *Simulator* does *Data Collection* and updates the *Statistical Variables* with the new observed values. Finally, the *Reporting Facilities* dump all of the observed values in a detailed *Observation Log* file. At any point in the simulation, the user can also request a *Performance Report* to be generated using the *Reporting Facilities*. A performance report shows the current status of the statistical variables, e.g. sum, sum of the squares, average and variance. In this way a *Performance Report* gives a more abstract view of the observed values than the view of an *Observation Log* file.

The performance facilities are mainly implemented in Standard ML [AM91]. The graphical interface is currently being implemented in C.

## 3 Example: Multiaccess Protocol

This section contains a non-trivial example of a CPN model that uses the performance facilities of Design/CPN. The example is a model of a protocol from one layer of a network architecture. The model presented here will be used in the following sections to illustrate how the performance facilities can be used. A detailed description of the protocol can be found in [BG92], and a brief outline of the protocol follows in Sect. 3.1. Section 3.2 contains detailed description of the model.

### 3.1 Aloha Protocol

The protocol to be modelled is the Aloha protocol which is a multiaccess protocol from the medium access control (MAC) layer of a network architecture. Multiaccess communication is communication between several sources, or nodes, across a shared communication medium, e.g. communication via an Ethernet or a satellite system. The purpose of the MAC layer is to allocate use of the shared communication medium among the competing nodes.

The protocol is used to coordinate communication between $m \geq 1$ transmitting nodes. For the sake of simplicity, we will assume that all messages are sent to one receiver and that the receiver is responsible for providing feedback. The nodes communicate by sending packets via a shared communication channel. The following assumptions are made about the system:

1. *Slotted system.* All packets have the same size, and each packet can be transmitted in one time unit, in the following referred to as a slot.
2. *Poisson arrivals.* Packets to be transmitted arrive at each node according to independent Poisson processes. Let $\frac{\lambda}{m}$ be the arrival rate for each node.
3. *Collision or perfect reception.* A *collision* occurs if two or more nodes send packets in a given time slot. In this case, the receiver obtains no information about the contents or senders of the packets. If only one node transmits a packet in a slot, then it is correctly received by the receiver.
4. *0,1,e Immediate feedback.* At the end of each slot, each node receives feedback from the receiver indicating whether 0 packets, 1 packet, or more than one (*e* for error) were transmitted in that slot.
5. *Retransmission of collisions.* Each packet that collides with another must be retransmitted. A packet is retransmitted until it is successfully received. A node is said to be *backlogged* if it has a packet that must be retransmitted.
6. *Buffering.* Each node has a buffer containing packets to be transmitted to the receiver. These packets have been received from the datalink control (DLC) layer.

The purpose of the protocol is to coordinate and make effective use of the communication channel. This is achieved by minimising both the number of collisions and the number of slots in which no packets are sent, or alternatively, by maximising the number of slots in which exactly one packet is sent. When a collision occurs, the transmitting nodes should not automatically retransmit their packets in the following slot because the packets would obviously collide once again. Thus, the basic idea of the Aloha protocol is to determine when backlogged and unbacklogged nodes should transmit packets. If a node is not backlogged, and its buffer is not empty, then it transmits a packet at the beginning of the next slot. Backlogged nodes retransmit in the following slots with some fixed probability $q_r > 0$ until the packet is successfully transmitted.

### 3.2 The CPN Model

It is first necessary to introduce some information about the complete model which is not included in the following figures. The model is a timed model with integer time. Each slot requires four units of time, i.e. each slot contains four subslots. Each transition is enabled in only one of these four subslots. A global reference variable, $m$, is used to indicate how many nodes there are in the system, and another, *lambda*, is used to calculate packet arrival from the DLC layer. In this model, the relevant parameters and their values are $m = 4$, $lambda = 1.0$ and $q_r = 0.25$, i.e. there are four nodes, the arrival rate to the system is approximately 1.0 packet per slot, and the probability for retransmission is 0.25.

Two pages from a CPN model of the Aloha protocol can be found in Figs. 2 and 3. These figures contain the models of the node and the receiver parts of the protocol, respectively. These are the most interesting pages of the model for the purpose of this paper because they use the performance facilities found in Design/CPN [LWb]. A complete model of the protocol was used to generate the performance results which will be presented in Sect. 6, but only these two pages of the model will be discussed in some detail. The parts of the model that are not included here are initialisation of various values, message passing from the DLC layer and the connection (physical network) between the nodes and the receiver.

The model makes use of all three parts of the performance facilities, i.e. distribution functions, statistical variables, and reporting capabilities. Several different statistics relevant to the performance of the modelled protocol are collected. Statistics regarding buffer size for each node and cumulative number of backlogged packets in the system are calculated. The total number of backlogged nodes is similarly recorded, as is delay for each successfully transmitted packet.

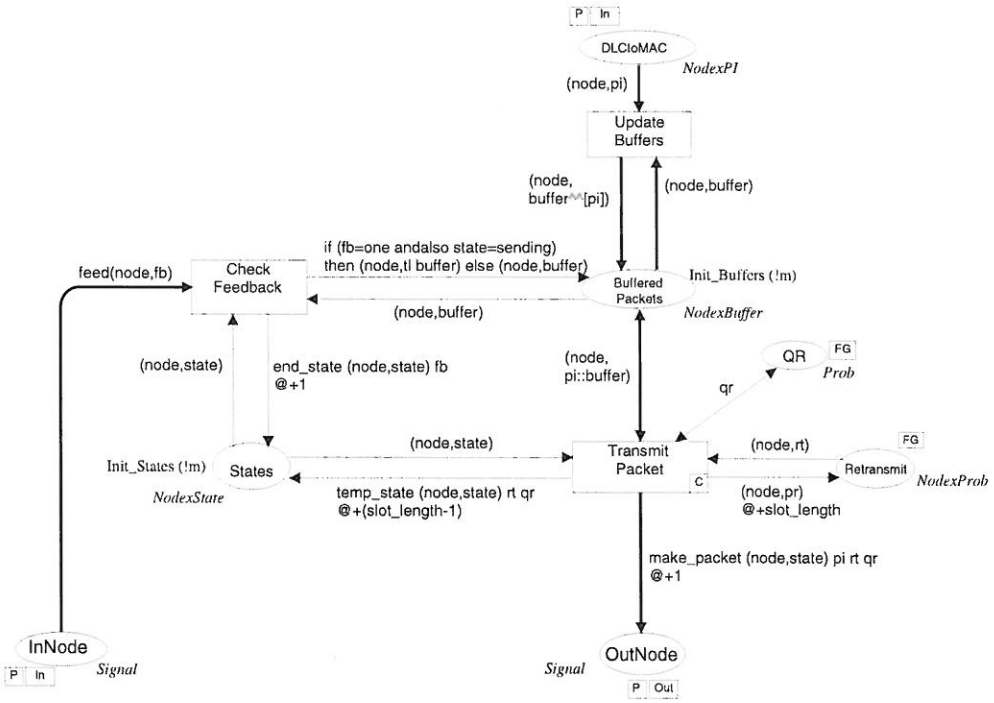



**Fig. 2.** Node in Aloha protocol.

Figure 2 is used to model all nodes. Packets being passed from the DLC layer to a node can be found on the place *DLCtoMAC*. The tokens on the place *Buffered Packets* are products: the first element is a node number, and the second is the node's message buffer. Statistics about buffer lengths and backlogged packets are calculated from the marking of *Buffered Packets*. The place

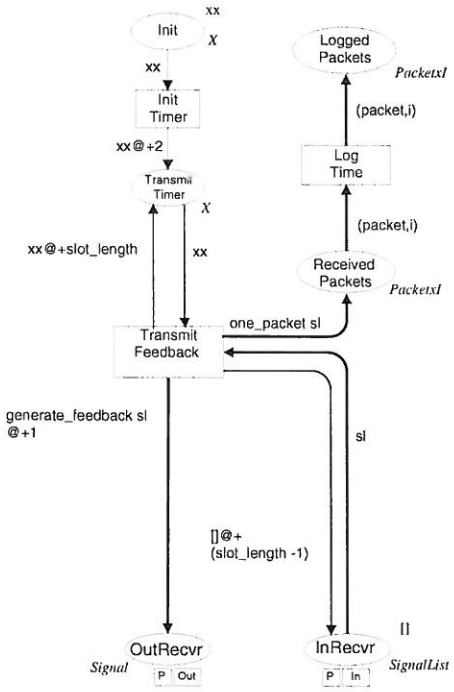



**Fig. 3.** Receiver in Aloha protocol.

*States* models the state of a node: whether it is *ready* (unbacklogged), *backlogged*, or in the process of *sending* a packet. The number of backlogged nodes in the system is calculated from the marking of this place. The values on *Retransmit* are compared to $q_r$ and are thus used to determine whether a backlogged node can retransmit in a given slot.

Whether or not a node can transmit a packet in the current time slot is determined by its state, its buffer, and possibly by the retransmission probability. If there are packets on $DLCtoMAC$ for node *node*, then they can be added to the end of *node*'s buffer. At the beginning of a slot, a node is either *ready* or *backlogged*. If a node's buffer is empty, then it must be *ready*, however, the transition *Transmit Packet* is not enabled, so no packets are sent from that node. When a node's buffer is not empty, and the node is *ready*, then the packet at the front of the buffer is automatically transmitted, and the node is transferred to *sending* state. In the last case, a *backlogged* node is allowed to send a packet only if the value from the place *Retransmit* is less than the predetermined retransmission probability which can be found on the place *QR*. When a *backlogged* node is allowed to retransmit a packet, its state is also changed to *sending*, while a *backlogged* node that cannot retransmit remains *backlogged*. The function `temp_state`[3] will change a node's state to *sending* when necessary. Each time a node attempts to transmit a packet, a token is removed from *Retransmit*, and then a new token with a time stamp which corresponds to the beginning of the next slot is returned to the place. In this way, nodes are prevented from attempting to transmit more than once per slot by the new time stamp.

Once all packets have been received from the DLC layer, and all nodes with packets have attempted to transmit a message, the receiver (Fig. 3) becomes active. The place *InRecvr* contains a list of the packets that were transmitted in the beginning of the current slot. The receiver uses this list to determine which type of message to broadcast to all nodes. If the list is empty, then the receiver sends a *zero* message indicating that no packets were received. A collision occurred if the list has length $\geq 2$, in which case, the receiver broadcasts an *e* (for error) message to all nodes. Finally, when the list contains only one packet, then the receiver broadcasts a *one* and saves the

[3] This function can be found on the arc from *Transmit Packet* to *States* in Fig. 2.

packet. The packet is saved in order to register delay. Again, a time stamp is used (on the token on *Transmit Timer*) to restrict the receiver to broadcasting messages exactly once per slot.

The nodes (Fig. 2) become active again after the receiver has broadcasted the appropriate messages. These messages can be found on place *InNode*. Recall that the nodes can be in *sending, backlogged* or *ready* states. If a node is either *ready* or *backlogged*, then it is only necessary to remove its feedback message from *InNode* because it has not attempted to send a packet in the current slot. If a node is in the *sending* state and the feedback from the receiver is *one*, then the packet at the head of the node's buffer was successfully transmitted, which means that it can be removed from the buffer. At the same time, the node's state must be changed to *ready*, indicating that it can transmit a packet in the next slot. Feedback *e* comes in response to a collision, so every node that is *sending* must become *backlogged*, and no changes are made to buffers because no packets were successfully transmitted. The function `end_state`[4] changes a node's state accordingly.

## 4 Distribution Functions

In this section we give a short description of the implemented distribution functions[5]. When modelling a physical phenomenon we often know that this particular phenomenon has a specific behaviour. This knowledge leads to a need for different functions that return random numbers from different distributions. By using such functions we are able to construct a more precise model of the physical phenomenon which can lead to better results when simulating and analysing the model. An example of a phenomenon which we know has such a specific behaviour can be the delay between busses arriving at a bus stop.

While analysing the built-in random number generator in Design/CPN, it became clear that this random number generator is not completely satisfactory. It is shown in [Dri] that this random number generator produces numbers that can not be considered especially random. This lead to constructing a new random number generator which has been proven to be better than the old one.

The distribution functions have then been implemented on 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.

The implemented distribution functions are: `Bernoulli`, `binomial`, `chi-square`, `discrete uniform`, `Erlang`, `exponential`, `normal`, `student` and `Poisson distribution`. See [Rip87] for further information about the distributions. A detailed description of the analysis, implementation and interface can be found in [Dri].

To illustrate the use of the distribution functions we refer to the example model in Sect. 3.2, where two distribution functions have been used to generate random values. The first one is used to generate the packets from the DLC layer arriving at the nodes. These packets are placed on the input place *DLCtoMAC* in Fig. 2. The packets are generated by the function `generate_packets` (see below) using the Poisson distribution function with arrival rate $\frac{lambda}{m}$ for each node (in accordance with assumption 2, Sect. 3.1).

```
fun generate_packets 0 = empty
  | generate_packets node =
      let
          val num = poisson((!lambda)/(real(!m)))
      in
          num'(node, (pack,time())) + generate_packets (node-1)
      end;
```

[4] This function can be found on the arc from *Check Feedback* to *States* in Fig. 2.

[5] Theo van Drimmelen [Dri] is solely responsible for the design and implementation of the distribution functions presented in this section.

As mentioned previously, the values of the tokens on the place *Retransmit* are used to determine whether a backlogged node can retransmit a packet in a given slot, therefore new values must be generated for each node every time it attempts to send a packet. Each time the transition *Transmit Packet* occurs it is necessary to find a new random value in the interval $(0, 1)$. These values are generated using the uniform distribution function, and the code segment for *Transmit Packet* sets $pr = \texttt{uniform}(0.0, 1.0)$, where $pr$ can be found on the arc from *Transmit Packet* to *Retransmit*.

## 5 Statistical Variables

This section describes the concept of statistical variables. Statistical variables are data structures providing the ability both to collect values from a simulation and to access statistics about these values during the simulation of a model. The values accumulated in a statistical variable can be integers or reals. Two types of statistical variables with different behaviour are available: timed and untimed.

The original implementation and design of untimed statistical variables was done by Alain Karsenty (see [Jenb]). We have modified the implementation of the untimed statistical variables to be more time-effective. Furthermore, we have added the timed statistical variables.

The values and statistics that can be accessed in both types of statistical variables are: minimum and maximum observed value, number of observations, sum, sum of squares, average, sum of squares of deviation, standard deviation and variance. Timed statistical variables include additionally: time of first update, time of last update and time interval (which indicates how much time has elapsed since the statistical variable was first updated).

Updates and accesses of a statistical variable can be freely intermixed. A statistical variable may also be reinitialised at any time, allowing a new set of values to be accumulated.

In our implementation of statistical variables [LWa], we do not save all the individual values but instead only calculate the values needed to be able to access the above mentioned statistics. This approach saves a lot of memory compared to accumulating all the values used for calculating the statistics.

**Untimed Statistical Variables.**
Figure 4 shows how an untimed statistical variable is updated with different observed values. If

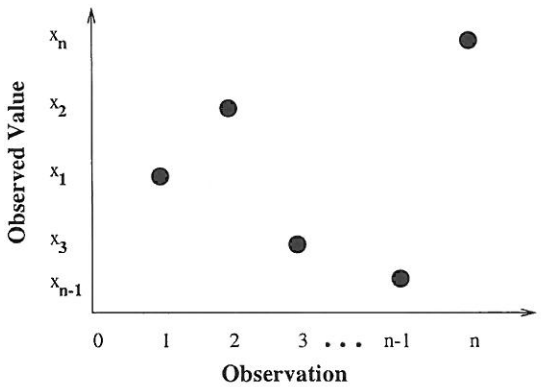



**Fig. 4.** Observed values for untimed statistical variables.

we update an untimed statistical variable with the the same value twice then the value influences the statistics twice, as expected. The sum is calculated in the following way:

$$Sum_n = \sum_{i=1}^{n} x_i$$

**Timed Statistical Variables.**
Timed statistical variables differ from untimed statistical variables in that an interval of time is used to weight each observed value. Assume that at time $t_n$, a timed statistical variable is updated with a new value $x_n$. At precisely time $t_n$, $x_n$ has no influence on sum, sum of the squares, average, sum of the squares of deviation, standard deviation or variance – the weight is zero, but for all time $t > t_n$, $x_n$ will influence these values. As an example, consider the graph in Fig. 5.

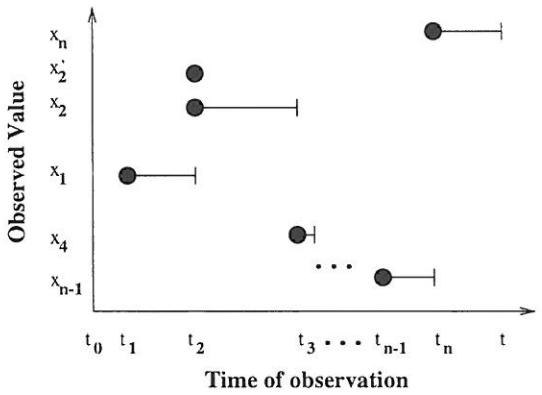



**Fig. 5.** Observed values for timed statistical variables.

Assume that a timed statistical variable was created at the time $t_0$ and that the circles indicate when the timed statistical variable was updated. The variable was last updated at time $t_n$ with value $x_n$. Let us look at how the sum and the average of the values are calculated at time $t \geq t_n$.

$$Sum_t = (\sum_{i=1}^{n-1} x_i * (t_{i+1} - t_i)) + x_n * (t - t_n) \text{ if } t \geq t_n$$

$$Average_t = \frac{Sum_t}{t - t_1} \text{ if } t \geq t_n$$

With timed statistical variables, it is possible for a value to exist for zero time, see for example Fig. 5 where the value $x'_2$ at time $t_2$ exists for zero time because it is followed by an update with the value $x_2$ at the same model time. In this situation we note that, in contrast to the above mentioned functions, the maximum, minimum and number of observations take into account all the values with which the statistical variable has been updated, i.e. including the ones which have existed for zero time (e.g. the value $x'_2$ in Fig. 5). This is due to the fact that the functions maximum, minimum and number of observations are the only ones not weighted with the time elapsed since the last update.
*Technical Remark.* Consider a timed simulation with integer time. If the time advances with value one for each update of a timed statistical variable, then the statistics for the timed and untimed statistical variables are equal, assuming that we access the statistics one time unit after the last update. The reason is that in this situation the weight is one for each value in the timed statistical variable.

## 6 How to use the Performance Facilities

In this section we describe the performance facilities provided for defining and manipulating statistical variables. The statistical variables described in Sect. 5 are incorporated into the Design/CPN tool. The performance facilities provide an easy way for defining when and how the statistical

variables should be updated. Reporting capabilities and maintenance of log files are additional components in the performance facilities. It is possible to maintain several different statistical variables (timed or untimed with integer or real values) – each being updated by extracting different values from markings and binding elements during a simulation. It is also possible to create and delete statistical variables after having simulated some steps. Furthermore, a statistical variable can be reinitialised at any time during a simulation.

When using the performance facilities, it is necessary to define two functions for each statistical variable. Definitions and examples of these two functions follow. Sections 6.1 and 6.2 introduce predicate functions and observation functions, respectively. In Sect. 6.3 observation log files are discussed. Finally, Sect. 6.4 contains an explanation of the performance report.

### 6.1 Predicate Function

A predicate function determines how often a statistical variable is updated. This function is a mapping from a marking and a list of binding elements to a boolean value. A predicate function is evaluated after each step of the simulator. If it evaluates to *true*, then the corresponding statistical variable is updated with an observed value, otherwise no update is made. If a function that always returns *true* is used as a predicate, then the statistical variable will be updated after each step in the simulation. When creating a statistical variable and later examining the performance report generated by the reporting facilities, it is important to be aware of when and how often an update of the statistical variable is performed.

The timed statistical variables have an additional note. An implication of the definition of a timed statistical variable (see Sect. 5) is that only the value observed at the time of the last occurrence of a transition before the time advances influences the values (sum, sum of the squares, etc.) of the *timed* statistical variables. As mentioned above, there are a few exceptions which are influenced: number of observations, minimum and maximum. This means that one has to be very careful when designing the predicate function.

We now take a closer look at some of the predicate functions that were used with the model presented in Sect. 3.2. The function `isSubSlot4` is a predicate function which returns *true* whenever the model time has reached the last subslot of a slot. The transition *Check Feedback* in Fig. 2 is enabled only in the last subslot, so it makes sense to record the number of backlogged nodes right after the states of the nodes have been updated to reflect whether or not they are backlogged.

```
fun isSubSlot4 (ogrec:CPN'OGrec, bes: Bind.Elem list) =
        ((time() mod 4) = 3);
```

The function `Transmit_Feedback_Occurred` is another example of a predicate function. This particular predicate function is interesting because it returns *true* only when a certain transition has occurred, i.e. the return value depends on the binding element from the current step. The local function `Transmit1` evaluates to *true* only when the transition *Transmit Feedback* occurs and *sl* is a list with one element. This predicate function is used when calculating delay of a packet, since it evaluates to *true* precisely when one packet has been successfully transmitted.

```
fun Transmit_Feedback_Occurred (ogrec:CPN'OGrec, bes:Bind.Elem list) =
  let
      fun Transmit1 [] = false
        | Transmit1 ((Bind.MAC_Receiver'Transmit
                        (1,{sl=((node_pack _)::[]),...}))::rest) = true
        | Transmit1 (_::rest) = (Transmit1 rest)
  in
      Transmit1 bes
  end;
```

We now take a closer look at some of the predicate functions that were used with the model presented in Sect. 3.2. The function `isSubSlot4` is a predicate function which returns *true* whenever the model time has reached the last subslot of a slot. The transition *Check Feedback* in Fig. 2 is enabled only in the last subslot, so it makes sense to record the number of backlogged nodes right after the states of the nodes have been updated to reflect whether or not they are backlogged.

```
fun isSubSlot4 (ogrec:CPN'OGrec, bes: Bind.Elem list) =
        ((time() mod 4) = 3);
```

The function `Transmit_Feedback_Occurred` is another example of a predicate function. This particular predicate function is interesting because it returns *true* only when a certain transition has occurred, i.e. the return value depends on the binding element from the current step. The local function `Transmit1` evaluates to *true* only when the transition *Transmit Feedback* occurs and *sl* is a list with one element. This predicate function is used when calculating delay of a packet, since it evaluates to *true* precisely when one packet has been successfully transmitted.

```
fun Transmit_Feedback_Occurred (ogrec:CPN'OGrec, bes:Bind.Elem list) =
  let
      fun Transmit1 [] = false
        | Transmit1 ((Bind.MAC_Receiver'Transmit
                        (1,{sl=((node_pack _)::[]),...}))::rest) = true
        | Transmit1 (_::rest) = (Transmit1 rest)
  in
      Transmit1 bes
  end;
```

### 6.2 Observation Function

An observation function is used to calculate the values with which the corresponding statistical variable is updated. This function is a mapping from a marking and a binding element to either an integer or a real. The return value of an observation function is used to update a statistical variable, provided that the corresponding predicate function evaluates to *true* after a step in a simulation.

It is important to remember that a predicate function determines how often a statistical variable is updated. Consider an observation function that returns a value that is solely dependent on the marking of a particular place. If the predicate function returns *true* after each step, then the statistical variable will be updated with the observed value even though the marking of the place from which the observed value has been extracted may not have changed.

Three predefined observation functions for performance measures are currently available in Design/CPN. The first one, *Number_of_Tokens*, records how many tokens are on a place. The other two are concerned with list length. For a place that has a colour set which is a list, *Average_List_Length* calculates the average list length given a multi-set of lists from that place. *Combined_List_Length* will return the sum of the lengths of lists.

For each statistical variable, the user can either use one of the predefined observation functions or define his own function to extract a specific value from a marking and a binding element. There are no limitations on how a value is computed. For example, the user-defined function can be a function which calculates the average length of lists from a number of places.

Another example showing the generality of the observation function could be a statistical variable counting the number of tokens that arrive to a place. In this way it is possible to get the average number of arrivals per step (or time-unit, if simulating with time and having chosen a timed statistical variable). This could be done by letting the observation function extract from the binding element the number of tokens arriving to the particular place.

The function `Backlogged_nodes` is an observation function that is used in the example model. In this function, the local variable *mark* refers to the marking of the place *States*.The local function

countBacklogged[6] determines how many nodes are *backlogged* for a given marking from the place *States*. In order to keep track of the number of backlogged nodes, it is only necessary to define Backlogged_nodes and isSubSlot4 for updating a timed statistical variable. In other words, a statistical variable is updated with the number of backlogged nodes each time a final subslot is reached in the simulation.

```
fun Backlogged_nodes (ogrec:CPN'OGrec, be: Bind.Elem) =
  let
     val mark = OEMark.MAC_Node'States 1 ogrec
     fun countBacklogged tempty = 0
       | countBacklogged ((coef,(node,backlogged),ts) !!! rest) =
             coef + (countBacklogged rest)
       | countBacklogged ((_,_,_) !!! rest) = (countBacklogged rest)
  in
     countBacklogged mark
  end;
```

### 6.3 Observation Log

It is possible to associate an observation log file with each statistical variable. This log file will be updated each time the statistical variable is updated. For timed simulations, it is possible to record the time of the update as well as the new value. In this way we get a file with all the observed values which can be examined after the simulation. This file could be used, for example, to draw a graph of the values with which the statistical variable has been updated (relative to the time of the update – if using a timed simulation).

The current observation log file format can be plotted by Gnuplot [Gnuplot]. Figure 6 is a Gnuplot graph created from observation log files produced when tracking buffer length for nodes 1 and 2 in the model. As one would expect, the buffer lengths continued to grow since the nodes were frequently backlogged and, therefore, unable to send packets and reduce the number of packets in their buffers.

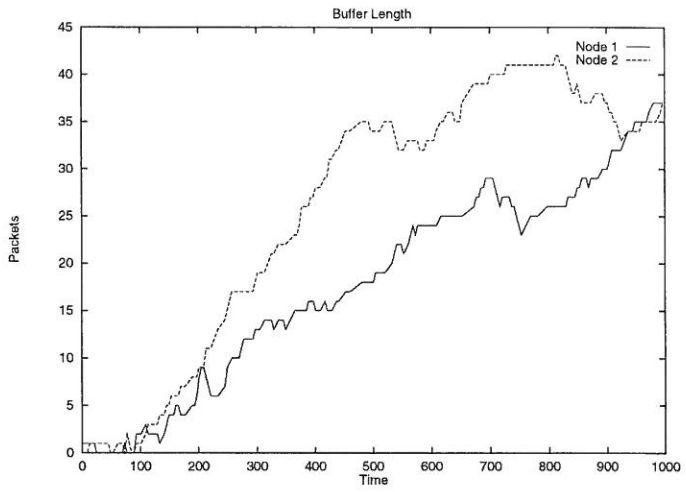



**Fig. 6.** Buffer length from log files.

[6] The multi-set operator !!! is used in the internal representation of timed multi-sets in Design/CPN.

### 6.4 Performance Report

After simulating a CPN model and thereby collecting some statistics, it is possible to print a report containing the different values of the statistical variables, e.g sum, last value, average, etc. This is done by specifying which of these values the user wants to be included in the report. It is also possible to specify both how many decimal positions should be shown and how wide the columns should be.

An example of a performance report which can be generated from the statistical variables can be found in Table 1. The first column, **Observation function**, is always part of the performance report. The values in this column are the names of the observation functions that were used to update the statistical variables. Each untimed statistical variable can produce nine values (sum, average, ...) while timed statistical variables can produce twelve (see Sect. 5). However, one is not always interested in knowing all of the values, so it is possible to indicate which subset of the values should be contained in the report. Here, for example, we chose to include number of updates, sum, average, and maximum, while disregarding the rest.

TIMED STATISTICAL VARIABLES

| Observation function | #Updates | Sum | Average | Max |
|---|---|---|---|---|
| 1 Backlogged_nodes | 2000 | 3379 | 3.39 | 4 |
| 2 Backlogged_packets | 1460 | 74836 | 74.84 | 142 |
| 3 Buffer_length_node_1 | 303 | 17739 | 17.76 | 37 |
| 4 Buffer_length_node_2 | 303 | 25795 | 25.82 | 42 |
| 5 Buffer_length_node_3 | 303 | 15223 | 15.24 | 37 |
| 6 Buffer_length_node_4 | 303 | 15533 | 15.55 | 32 |
| 7 Number_of_Tokens_OutNode | 4122 | 303 | 0.30 | 4 |
| 8 Combined_List_Length_InRecvr | 4122 | 303 | 0.30 | 4 |
| 9 Average_List_Length_InRecvr | 4122 | 303.00 | 0.30 | 4.00 |

UNTIMED STATISTICAL VARIABLES

| Observation function | #Updates | Sum | Average | Max |
|---|---|---|---|---|
| 10 Packet_delay | 109 | 30958 | 284.02 | 642 |

**Table 1.** Performance Report

*Contents of Statistical Variables.* In this particular model, ten statistical variables were used. Three of these (Lines 7-9, Table 1) track the predefined performance measures that are available, the other seven record values that are of specific interest in this model. Line 7 in Table 1 indicates that a total of 303 packets were recorded on place *OutNode*. Here it is interesting to recall how a timed statistical variable is updated. More than one step in a timed simulation can occur before the time advances, this means that only the last value inserted before the time changes will have any influence on the sum, sum of the squares, etc (as discussed in Sect.5 ). Thus, if at a given time there were one, two and finally three tokens on a place, only the value 3 will have influence on the sum, sum of the squares, etc. In this model, there are no tokens on *OutNode* for three out of four subslots (time units). Since the statistical variable which records the number of tokens on this place is updated after every step, many zeros will be added to the total sum of tokens. This accounts for the relatively low (0.30) average number of tokens on the place.

Two statistical variables were used in this model to collect data about the average list length and combined list length of lists on the place *InRecvr*. Lines 8 and 9 in Table 1 show the values accumulated in these variables. Since there is always exactly one list on this place, the values for combined list length and average list length are necessarily the same.

The seven other rows in Table 1 correspond to the model specific performance measures. The number of backlogged nodes in the system can be found in Line 1. On average there were 3.39 (out

of a possible 4) backlogged nodes, which indicates that the parameters chosen in this model of the Aloha protocol do not produce a very effective means for allocating use of the communication channel.

Since there were so many backlogged nodes, there must have been backlogged packets in the system. The total number of backlogged packets at a given time is the cumulative number of packets in the buffers belonging to the nodes. One might think that it would be possible to track this value using the predefined performance measure *Combined_List_Length*. However, this is not possible because the lists corresponding to buffers are one element in a product (see Sect. 3.2), and *Combined_List_Length* can only be used when the tokens on a place are lists. The length of a single node's buffer can be found in Lines 3-6. It is not surprising that the average buffer lengths are quite high (between 15.24 and 25.82), knowing that the nodes were frequently backlogged and that new packets continued to arrive in each slot. The total number of backlogged packets was updated in the first subslot of each slot, while buffer lengths were updated in the second subslot of each slot. It is possible that a different number of simulation steps occur in different subslots (time units), and this accounts for the discrepancy between the number of updates for *Backlogged_packets* and *Buffer_length_node_x*.

The final performance measure of interest is the average delay for packets which were successfully transmitted. The delay was measured in time units. This value was recorded in an untimed statistical variable, and the results can be found in Line 10. The appropriate statistical variable was updated every time a packet arrived on the place *Received Packet* in Fig. 3.

## 7 Conclusion

The motivation for writing this paper came from our work with developing the performance facilities. Our original focus was on extending the untimed statistical variables to timed versions. While working with the statistical variables, it seemed like a logical next step to provide a better, high-level interface to the statistical variables. Previously, it was frequently necessary to add code segments and/or extra places and transitions to a CPN model in order to extract the desired statistics. It is obviously time consuming to add these extra structures. Adding new structures can also lead to more error-prone CPN models. By using the performance facilities described in this paper, the means for collecting data is not directly present in the CPN model, i.e. very little, if anything needs to be added to the model. Thus, it is possible to keep the semantics of the model and the collection of data from the model separate. We think that this is clearly an improvement.

Once it was possible to easily create and update statistical variables independently of a CPN model, it seemed useful to have a reporting facility which could present the results in a simple report. Combining these capabilities with an existing Design/CPN library of distribution functions resulted in a performance library which a user could include and use when simulating models in Design/CPN. The interface to this library consisted solely of fairly low-level function calls. A graphical user interface to the new capabilities, in the form of menus and dialog boxes, would obviously provide a higher-level interface. This interface has not been completed, but it is in the process of being implemented. During development of the library, the advantages of integrating it into the Design/CPN tool became obvious, and the result is the new performance facilities in Design/CPN described in this paper.

Some improvements to the design of the performance facilities are interesting for further research. One of the areas is the predicate function. A possible improvement could be to give the user the ability to specify a list of transitions for each statistical variable. Then we could check if one of these transitions has occurred after each step. If this is the case, then we could update the statistical variable with the observed value, otherwise not. This would be another step towards providing a high-level interface to the performance facilities. The issue of performance analysis based on occurrences of transitions also deserves more attention in future work.

Another possible extension of the performance facilities is the ability to export the defined statistical functions and the corresponding predicate and observation functions to a file. The facility of being able to export these definitions in a library of statistical variables would be

useful. In this way it would be possible to import and re-install the runtime table containing these statistical variables when restarting the simulator. This feature is also in the process of being implemented, but it too is not quite finished yet. One limitation to having a run-time table is that no places or transitions referred to in the predicate or observation functions may have been removed or renamed since the definitions were saved.

It is still possible to use the statistical variables independently of the performance facilities, but then the user is completely responsible for updating the statistical variables and extracting the appropriate values at the end of a simulation. The user is also required to open, close and update files if he is interested in maintaining a log file for each statistical variable.

It has been our experience that the performance facilities provide a user-friendly interface for collecting data, calculating statistics and reviewing the collected data. In conclusion, we do believe that these performance facilities can help doing performance analysis of CP-nets created in Design/CPN.

**Acknowledgements.** We would like to thank Lars Michael Kristensen for his help in writing this paper. We thank the CPN group at the University of Aarhus for comments both on the design of the performance facilities and on this paper.

## References

[AM91] Appel, A.W. and MacQueen, D.B. Standard ML of New Jersey. In J. Maluszyński and M. Wirsing, editors, *Third International Symposium on Programming Languages Implementation and Logic Programming*, volume 528 of*Lecture Notes in Computer Science*. Springer-Verlag, 1991.

[BG92] Bertsekas, D., Gallager, R.: *Data Networks*. Prentice Hall, 1992.

[CJ97] Christensen, S. and Jørgensen, J.B. Analysing Bang & Olufsen's BeoLink® Audio/Video System Using Coloured Petri Nets. In P. Azema and G. Balbo, editors, *Proceedings of the 18th International Conference on Application and Theory of Petri Nets, Toulouse, France*, volume 1248 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[CK97] Christensen, S. and Kristensen, L.M. State Space Analysis of Hierarchical Coloured Petri Nets. In: B. Farwer, D. Moldt and M-O. Stehr (Eds): *Proceedings of Workshop on Petri Nets in System Engineering (PNSE'97) Modelling, Verification, and Validation, Hamburg, Germany*, Publication No. 205, University Hamburg, Fachberich Informatik, pp. 32-43, 1997.

[CJK97] Christensen, S., Jørgensen, J.B. and Kristensen, L.M. Design/CPN - A Computer Tool for Coloured Petri Nets. In E. Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Twente, The Netherlands* volume 1217 of *Lecture Notes in Computer Science* pages 209-223. Springer-Verlag, 1997. Also available as DAIMI PB-511, ISSN 0105-8517, February 1997.

[Dri] Drimmelen, T. Implementation of Statistical Functions in Design/CPN. Online: http://www.daimi.aau.dk/designCPN/libs/pdf/

[Gnuplot] Online: http://www.cs.dartmouth.edu/gnuplot_info.html

[Jena] Jensen, K. et al *Design/CPN Online*, Department of Computer Science, University of Aarhus, Denmark. Online: http://www.daimi.aau.dk/designCPN/.

[Jenb] Jensen, K. et al *Design/CPN Reference Manual*, Department of Computer Science, University of Aarhus, Denmark. Online: http://www.daimi.aau.dk/designCPN/man/

[Jen92] Jensen, K. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts*. Monographs in Theoretical Computer Science. Springer-Verlag, 1992.

[Jen94] Jensen, K. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods*. Monographs in Theoretical Computer Science. Springer-Verlag, 1994.

[Jen97] Jensen, K. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 3, Practical Use*. Monographs in Theoretical Computer Science, chapter 2, pages 21–37. Springer-Verlag, 1997.

[LWa] Lindstrøm B., Wells, L. The Design/CPN Statistical Variable Library, Department of Computer Science, University of Aarhus, Denmark. Online: http://www.daimi.aau.dk/designCPN/libs/

[LWb] Lindstrøm B., Wells, L. User Manual for the Performance Analysis Package, Department of Computer Science, University of Aarhus, Denmark. Online: http://www.daimi.aau.dk/designCPN/

[Rip87] Ripley, B.D. *Stochastic Simulation*. Wiley series in probability and mathematical statistics. Applied probability and statistics, 1987.

# A Framework for Interacting Design/CPN- and Java-Processes

Olaf Kummer Daniel Moldt Frank Wienberg

Universität Hamburg, Fachbereich Informatik
Vogt-Kölln-Straße 30, D-22527 Hamburg
{kummer,moldt,wienberg}@informatik.uni-hamburg.de

**Abstract**

In order to widen the applicability of Design/CPN for the specification and design of large scale distributed applications, a framework has been developed that supports the interaction of Design/CPN and Java processes. Thereby a seamless embedding of the two worlds of Petri nets and object-oriented programming is achieved, allowing problem oriented modeling at different abstraction levels in a fully distributed environment.

The general possibilities to connect Design/CPN with remote processes are discussed and a specific implementation of the required framework is sketched. Promising application areas are named and for some of them concrete example models are provided.

**Keywords:** Coloured Petri Nets, Design/CPN, Distributed Simulation, Framework, Java, Workflow, Prototyping

## 1 Introduction

The specification of systems, its evaluation, and its transfer to implementation is still a major task for computer science. One very promising technique in the area of specification, especially when concurrent and distributed systems are involved, are Coloured Petri Nets (see [7]). Because a strong interconnection of specification and implementation is very useful when developing a system, it is desirable to bring together the worlds of Coloured Petri Nets and some popular programming language.

In the area of implementation Java (see [6]) aroused special interest for building applications for the Internet, as it is an object-oriented, reasonably portable programming language that supports additional features such as high-level networking and easy multithread programming.

As a specification tool based on Petri nets, we chose Design/CPN (see [3] and [9]), because it is flexible and powerful and comes with a specially adapted graphical editor. Design/CPN supports the development of large systems by means of hierarchical models. Furthermore, an internal programming language, namely ML, can be used to extend the tool for our needs.

In the environment proposed here, Java should be used for the implementation of graphical user interfaces, database connectivity, and other applications for which one can fall back upon reusable implementations, while Design/CPN serves as the graphical specification tool that is powerful in designing and executing models of concurrent, distributed systems.

But there are more reasons why an interaction of Design/CPN and Java has been expected and indeed turned out to be fruitful, namely to overcome some limitations of Design/CPN: The tool is designed for single-user mode only, but especially large-scale projects are dependent on group- or teamwork. Although there is a flexible hierarchy concept, Design/CPN does not really support component re-use, as the exchange of parts between

different models is difficult. In the simulator, interfaces for calling programs implemented in languages other than ML are supported on a very low level only. Although the tool itself, especially with the extension Mimic (see [13]), offers graphical user interface routines, these are also very rudimentary compared to state-of-the-art tools. Last but not least the process of implementing a system for which a Design/CPN prototype has been designed is not well supported by the tool. Since there is no way to execute a Design/CPN model in a stand-alone fashion and also simulation of large-scale models is not quite as efficient as a (compiled) program, it would at least be desirable to provide a stepwise migration of the system from net models to some programming language.

Motivated by all these considerations, a framework has been designed that establishes new possible fields of application where Design/CPN may be employed. The framework extends Design/CPN in several ways: Distributed simulation is achieved by different technical means, namely sockets and pipes. The implemented architecture and alternative concepts for the interconnection of multiple Design/CPN processes are described in section 2. In section 3 it is shown how Design/CPN can communicate with Java processes during simulation. This allows invoking Java methods and even the creation of Java objects. Section 4 explains the reverse communication direction where Design/CPN processes are invoked by Java, viewing the whole Design/CPN process as an object. Possible applications of the extensions and benefits gained from them are discussed in section 5. Section 6 provides an outlook on how the framework can be extended further and how it can be exploited for other approaches.

# 2 Distributed Design/CPN Processes

If we want to achieve a distributed simulation using Design/CPN, we have to run multiple instances of the program and allow synchronization of the multiple nets that are being executed.

Distributed simulation in our sense does not mean the distributed simulation of a single Petri net, which would imply the distributed solving of conflicts. This problem has been treated in e.g. [5]. Instead, we will distribute our models in such a way that only local conflicts can occur, so that they can be handled within one Design/CPN process.

## 2.1 Possible Communication Channels

Running multiple instances of Design/CPN in a single simulation run requires communication and synchronization. Since shared memory is not provided, we have to use some form of message passing.

Whatever I/O-channel is used, a blocking I/O call will stop the current net simulation, because the simulator is strictly single threaded. Hence it is usually not advisable to perform blocking I/O, although this might be an option in some cases. Instead, we have to do polling I/O, checking for new input from time to time.

There are essentially three possibilities to send and receive messages from within a Design/CPN simulation process:

- Ordinary files. One Design/CPN process might write to a file, while another process might read the file. This is an inefficient method, especially if the processes access a file by NFS. Almost all network file systems cache file contents so that a change does not immediately propagate through the whole network. This makes ordinary files practically worthless for communication purposes.

- TCP connections. A TCP connection involves a server and a client. The server will setup a server socket under a well-known port number, so that it can accept an arbitrary number of connections from clients. Each server socket is identified by the port number and the server's host at the time the client socket is created.

  Using the ML library `SysIO` Design/CPN supports client sockets with the help of low-level file descriptor I/O.

- Pipes. In Design/CPN it is possible to generate an external process on the same machine by means of the `execute` function. The call will provide input and output pipes from Design/CPN to the created process. These can be handled by the usual ML stream I/O library, which is significantly easier to use than the raw file I/O. Of course, after a few routines have been written to handle the interprocess communication, the designer of a model does not get into contact with these routines very often anyway.

  This approach is suggested in the `process` example that is distributed with Design/CPN 3.02 which starts a Tcl/Tk process as an example.

One the one hand, pipes are most useful when we want to communicate with other processes on the local machine that can be started during the simulation. On the other hand, the more versatile TCP connections have to be used if the processes is required to run on different machines or to be started before the net simulator.

## 2.2 Possible Communication Architectures

The process started by an `execute` could of course be another Design/CPN process, but a Design/CPN simulation cannot access its own standard input and output streams. Moreover, at present there is no way to start a simulation automatically within a Design/CPN process with which we could interact. These two difficulties practically rule out direct pipe communication between the two nets.

Moreover, we cannot use direct TCP communication, because it is not possible to implement server sockets within Design/CPN. This leaves us with three basic options:

- One dedicated communication process is started (the communication server). It opens a server socket that can be accessed by an arbitrary number of Design/CPN processes with one TCP connection each. For a visualization of this architecture see Fig. 1.

- Every Design/CPN process starts a messenger subprocess and accesses it via pipes. The messenger subprocesses is responsible for transporting the messages to and from the server. It will handle the protocol with the communication server as before (see Fig. 2).

- Only the subprocesses are present and implement a suitable algorithm for direct message exchange, typically using TCP (see Fig. 3).

In principle it would be possible to organize the processes like in Figs. 2 or 3 using TCP connections instead of pipes. However, this would imply programming more difficult ML functions without any obvious benefits.

Of course the communication processes can be implemented in any language that supports the necessary networking ability, but among the many options the most promising seems to be Java for the reasons given in section 1.

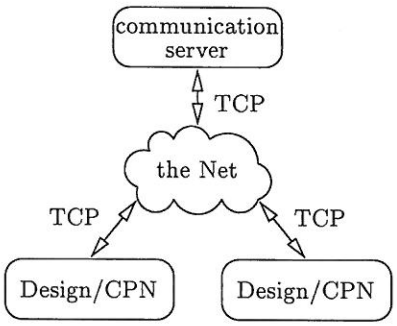



Figure 1: Server solution

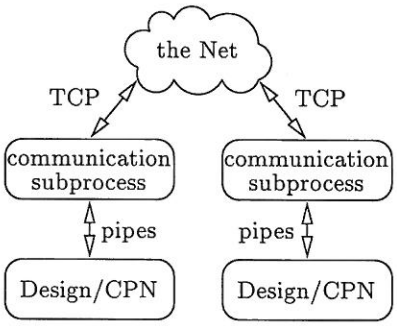



Figure 3: Fully distributed solution

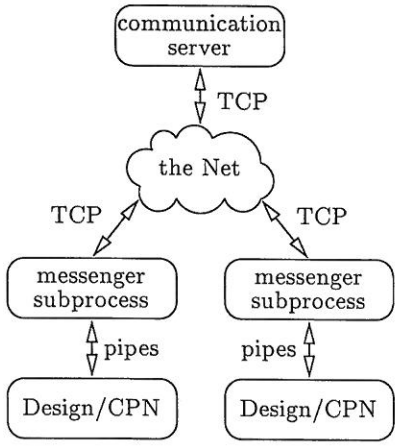



Figure 2: Messenger solution

### The Server Solution

In Fig. 1 we can imagine the Design/CPN processes as actors that exchange messages via a mailbox (the server). Each actor has a unique mail address, which may be generated by the system at runtime and which will usually exploit the unique network name of the local computer. Some actors may also be assigned a unique well-known address chosen at programming time.

In its simplest form, the mailbox only has to provide commands to send a message to a mail address and to receive a message, if one is available. For performance reasons it may be useful to receive all pending messages. More complex implementations would allow a blocking wait for messages, but this is of little use for a Design/CPN process, since it would suspend the simulation completely. Other options include a forwarding mechanism or a test if a mail address is valid.

The host and the port where the mailbox is located must be public for all Design/CPN processes, so that they can establish a connection.

The mailbox can quickly become the system's bottleneck, especially because polling access is required on the side of Design/CPN. Nevertheless, this method is surprisingly practical and will perform well provided there are only few processes.

### The Messenger Solution

In this case there is still a mailbox, but every Design/CPN process starts a special messenger process by means of the `execute` function, as in Fig. 2. The messenger process can communicate with Design/CPN locally, which is usually much faster than remote interaction. Also, since the messenger is only responsible for a single Design/CPN process, we can use a significantly simpler protocol. Moreover, the messenger can communicate with

the mailbox using efficient blocking I/O, thereby taking some load off the bottleneck. Of course, the price to pay is a further indirection leading to some communication overhead.

### The Fully Distributed Solution

Here we no longer have a central instance; instead each subprocess spawned by Design/CPN interacts directly with other subprocesses. This would lead to further performance benefits at the price of a vastly increased complexity of the message-handling algorithm.

Even in a fully distributed architecture there might be a kind of centralized name service, which would allow the distributed nets to communicate without knowing the actual location of the other nets.

## 2.3 A Communication Package

We have tried the server solution (in C) and the messenger solution (in Java). Here we are going to limit ourselves to a description of the messenger solution, because it results in a more readable Design/CPN package.

The actual implementation of the mailbox and the messenger processes are beyond the scope of this paper. Let it suffice to say that much effort went into the handling of concurrency and into the protocol. The actual network programming turned out to be very easy in Java, contrary to C where network calls consitute the majority of the code.

At the moment the pure subprocess solution is not implemented, but it could be done in about three weeks, if this is required due to performance reasons. The Design/CPN library, which we are going to describe now, and the protocol on the pipes would not change at all, making an upgrade possible without any noticeable difference for the users.

All basic message handling is done on a single I/O page that can be reused in every distributed system of nets, see Fig. 4. It encapsulates all the transitions that set up the connection and send or receive messages.

In the communication package described here every Design/CPN process is assigned an arbitrary character string that can be used as the mailbox address. The I/O page is given the mailbox address on which it will listen via the "in" port place `connect`.

The ML variable `messenger` has to contain the path and name of the executable that starts a messenger process and should be defined in the global declaration node, e.g.

```
val messenger = "/home/cpnuser/bin/messenger";
```

This path name is not passed as a token, because it does not often change and because an incorrect path might result in errors that are undetectable in the ML code.

Outgoing messages must be put into the "in" port place `send` in the form of an address/data pair. Both will be handed on to the messenger through the output stream, preceded by the command `send`. Messages are received from the messenger through the input stream. It is important to test whether there is any pending input with the ML function `can_input`, because `input_line`[1] suspends until a newline character (`\n`) is read. The page provides access to the received messages using the "out" port place `received`. The data to be sent and received is always a character string. Other datatypes must be converted to a string format, e.g. by using the predefined `mkst_col'`*`colour`* functions.

Tokens can be put into the `polling` place or be removed from there in order to either start or stop polling for new messages. Design/CPN only allows to stop an "automatic"

[1]The function `inputLine` used here is just a customized version of `input_line` that cuts off the newline at the end of the result string.

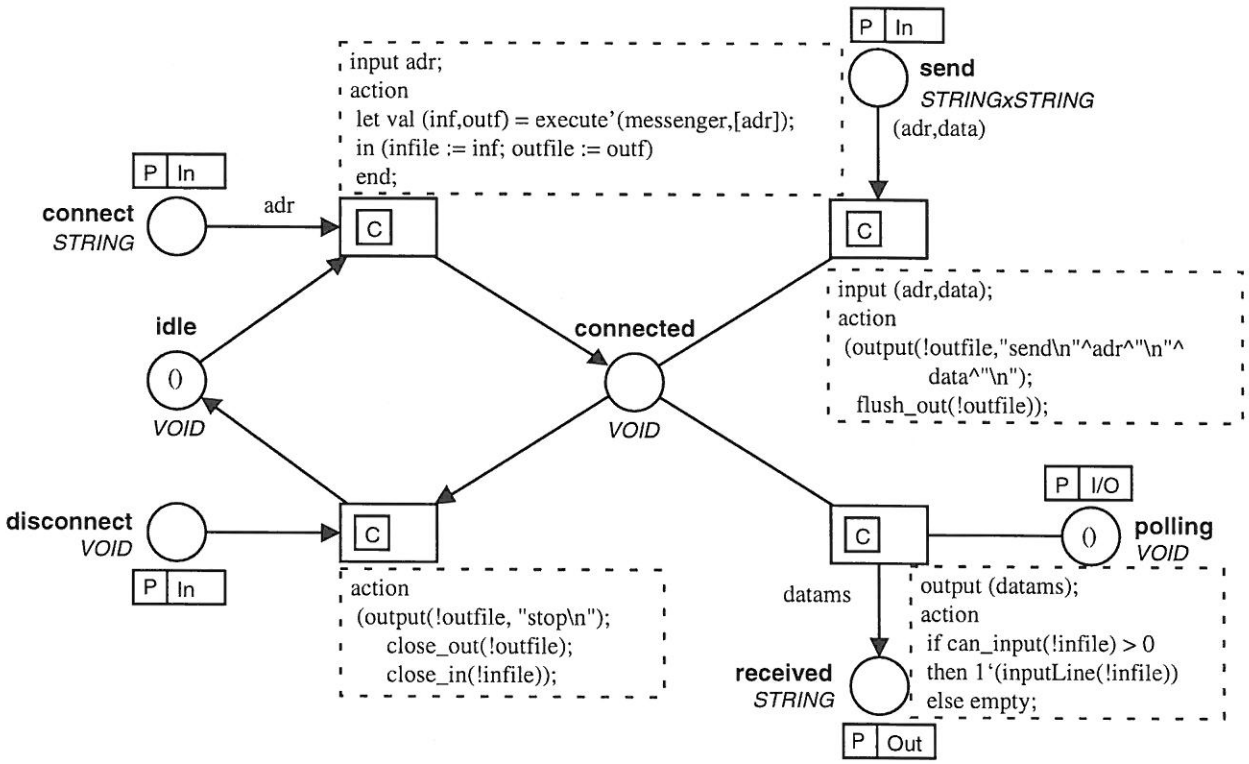



Figure 4: A subpage for string message I/O (`IOPage`)

simulation run after a given number of steps or in case there are no more enabled transitions. Removing the `polling` token offers the possibility to stop an automatic simulation run without disconnecting. Furthermore, this feature should be used with nets where it is known when to expect incoming messages, e.g. only as answers to query messages that have been sent before. It is much more efficient not to poll for new messages while the net is working locally and not expecting any message input. If the `polling` port place is not assigned on the "parent" page, the initial marking of one token will be used and polling will always be activated as long as the connection is up. It should be recalled that Design/CPN uses the initial marking of a port place only if no socket place is assigned to it.

Finally, there is a facility to close the connection and shut down the messenger process, because these processes might stay alive when they are not terminated properly. If one wants to do so, one has to change the marking of the `disconnect` place to one token and fire the `disconnect` transition.

Because pipes cannot be uniquely represented as strings, they cannot be used as token colours, hence they have to be stored in reference variables. These are not allowed to be used in arc inscriptions, but in code regions only. We defined two *global* instance variables, `infile` and `outfile`. If one wants to use the subnet more than once, because there is some need to run multiple messenger processes, one can also use *instance* reference variables which have different values for each page instance. This may be desired when many net fragments are developed and tested within one simulator, before they are finally split into many independent nets. The reason why we used global reference variables is that we normally do not use more than one instance of the I/O page and also that instance variables cannot be reached from ML code via Design/CPN's *ML evaluate* feature, e.g. to close the streams manually.

# 3 Accessing Java from Design/CPN

It should be clear that the message passing scheme used in our architecture does not rely on specific Petri net techniques, so that Design/CPN could be complemented by programs in arbitrary languages that support TCP. Again we choose Java as our example language, even though experiments have also been done with C (for details see [1]).

A straightforward implementation would provide only the basic routines to send and to receive string messages, leaving the programmer with the task of making the necessary calls. But we can increase the developers' productivity by defining a standard message format, so that we can supply reusable parsing algorithms.

On top of that, we can provide algorithms that actually perform the call that was requested by Design/CPN, so that the message passing framework becomes completely invisible to Java programs. Quite the same can be achieved on the side of Design/CPN where we make a Java method call look like an ordinary substitution transition (see section 3.4).

## 3.1 The Message Format

The message format should contain all the necessary information to make a Java method call: the object whose method is invoked, the caller that awaits a return message, the method name for the called Java object, and a list of parameters. We have to distinguish these *call messages* from another type of messages which we call *return messages.* A return message is much simpler, as the method name and the caller can be omitted. What remains is just the target object (which is the sender of the call message) and a list of parameters, which in case of Java may only be of length one or zero (if a method is of return type `void`).

One suggestion for a suitable message format that has been implemented has the main aim to keep the net inscriptions and functions simple on the side of Design/CPN. As Design/CPN is based on the functional language ML, it can handle lists very well. Thus, a message is implemented as a list of the components mentioned above.

However, Design/CPN-colours are strongly typed, so a union type of all different types that may appear in a message has to be defined. We end up with a message colour definition as follows:

```
color OBJECT = union Null
                     + RC:RCLASS
                     + RI:RINSTANCE
                     + M:METHOD
                     + Int:INTEGER + Str:STRING + Bool:BOOLEAN + Real:REAL;
color MSG = list OBJECT;
```

with `RC` being a colour that represents a reference to a remote class, `RI` representing a reference to a remote instance, `M` declaring a method name and `Int`, `Str`, `Bool` and `Real` being the constructors for the basic datatypes available in ML and Java. Other types, especially arrays, could be added, if desired.

This very general message format needs additional constraints for well-formed call and return messages, but offers the advantage to define both with the same Design/CPN colour. In order to distinguish call and return messages easily, we chose to put the method name in the first position of the object list instead of using the order of the object-oriented dot notation, where the receiving object is named first. A message starting with a method

name is assumed to be a call message, or else a return message. The complete sequence of a call message is

```
[M(method-name), invoked-object, caller-object, param1, param2, ... ]
```

A return message is simply

```
[receiver-object, result]
```

where *result* is optional and the *receiver-object* is the former *caller-object*.

References are defined as tuples of a class and a process identifier the latter of which is used to locate remote objects. Unlike a class reference, an instance reference also contains an ID as a third component that makes the triple a globally unique identifier.

```
color RCLASS = product CLASS * PID;
color RINSTANCE = product CLASS * PID * RID;
```

These tuples could have been coded into a single string, but using tuples we can apply the built-in Design/CPN pattern matching capabilities to select the information that is needed to send and receive messages.

All colours that have not been declared are simply defined as `STRING` in our implementation.

Alternatively, the structure of a message given at the beginning of this section could be directly translated into Design/CPN-colours. For a call message, this would result in a four-tuple of the object being invoked, the method name, the calling object and a list of parameters, which again is a list of objects defined as the message above.

## 3.2 A Sample Message Communication

To illustrate the message format, we will provide a code fragment using the big integer numbers provided by the standard Java library. Real-world applications will of course access more complex classes, but the example is sufficient for illustration purposes.

```
BigInteger bigNum1 = new java.math.BigInteger("42");
BigInteger bigNum2 = bigNum1.pow(42);
String bigStr = bigNum2.toString();
```

Obviously, one call and one return message have to be sent for each statement. When sending messages, we have to provide a unique name for the sender. We can for example use the constant class name `cpn`, construct the "mailbox" name as *hostname:process-ID* (of the Design/ML process), and use a sequence number for every sender being active concurrently. Thus, we end up with the first sender called `RI(("cpn","cpnhost:12345","1"))`[2].

At first, the net has to invoke a constructor method of the class `BigInteger`. To do so, we also have to provide a mailbox address where a special Java process is waiting for messages (e.g. `javaserver`) and send a `new` message like this:

```
[M("new"), RC(("java.math.BigInteger","javaserver")),
           RI(("cpn","cpnhost:12345","1")),
           Str("42")]
```

[2]The double brackets appear due to the standard Design/CPN `mkst_col` function which produces a pair of brackets for both the tuple *and* the union constructor `RC`. Although most brackets may be omitted in net inscriptions, interactive input, and any other expressions that are evaluated by ML, only this syntax is accepted by the predefined Design/CPN function `input_col` used for reading colour values from the input stream.

The message becomes much easier to read if we replace the objects by CPN variables, which normally is the case in net inscriptions:

```
[M("new"), bigIntegerClass, cpn1, Str("42")]
```

The Java program then performs some actions (which are explained in section 3.6) and answer with a reference to the newly created object:

```
[cpn1, RI(("java.math.BigInteger","javaserver","xyz123"))]
```

Let us call the new object `bigNum1`. Note that the given reference contains its class, the host (or rather: mailbox name) where it can be reached and some automatically generated ID. Now, we want to compose the message corresponding to the second statement. The net has to invoke the `pow` method of the given object with the parameter value 42 (this time as an integer). We can reuse `cpn1` as the sender, because no other return messages are expected and there is no concurrency involved.

```
[M("pow"), bigNum1, cpn1, Int(42)]
```

Java will answer with a message like

```
[cpn1, RI(("java.math.BigInteger","javaserver","abc456"))]
```

Let us call the given instance `bigNum2`. Now we convert this number to a string:

```
[M("toString"), bigNum2, cpn1]
```

Note that this method call does not take any parameters. Java responds

```
[cpn1, Str("15013093754529657235677...797056873877723589353301 6064")]
```

Have a look at the first message again, but this time using the alternative message format mentioned in section 3.1:

```
(bigIntegerClass, "new", cpn1, [Str("42")])
```

The only gain is that we can omit the union constructor `M` for the method name, but on the other hand, additional brackets appear. Also, we have to define a different colour for return messages. When receiving a message, it normally cannot be anticipated whether it is a call or a return message, so we would have to define another union colour consisting of call *or* return messages, which again makes the message format somewhat longwinded.

All in all, the perfect message format is partly a matter of taste, but we believe that the list format offers most advantages for Design/CPN. If you want customized net inscriptions, there is also the possibility of defining special colour sets in ML. Using your own string representation, you are no longer dependent on ML syntax. However, this approach takes more time to implement and you still have to tackle the problem of integrating CPN variables into net inscriptions.

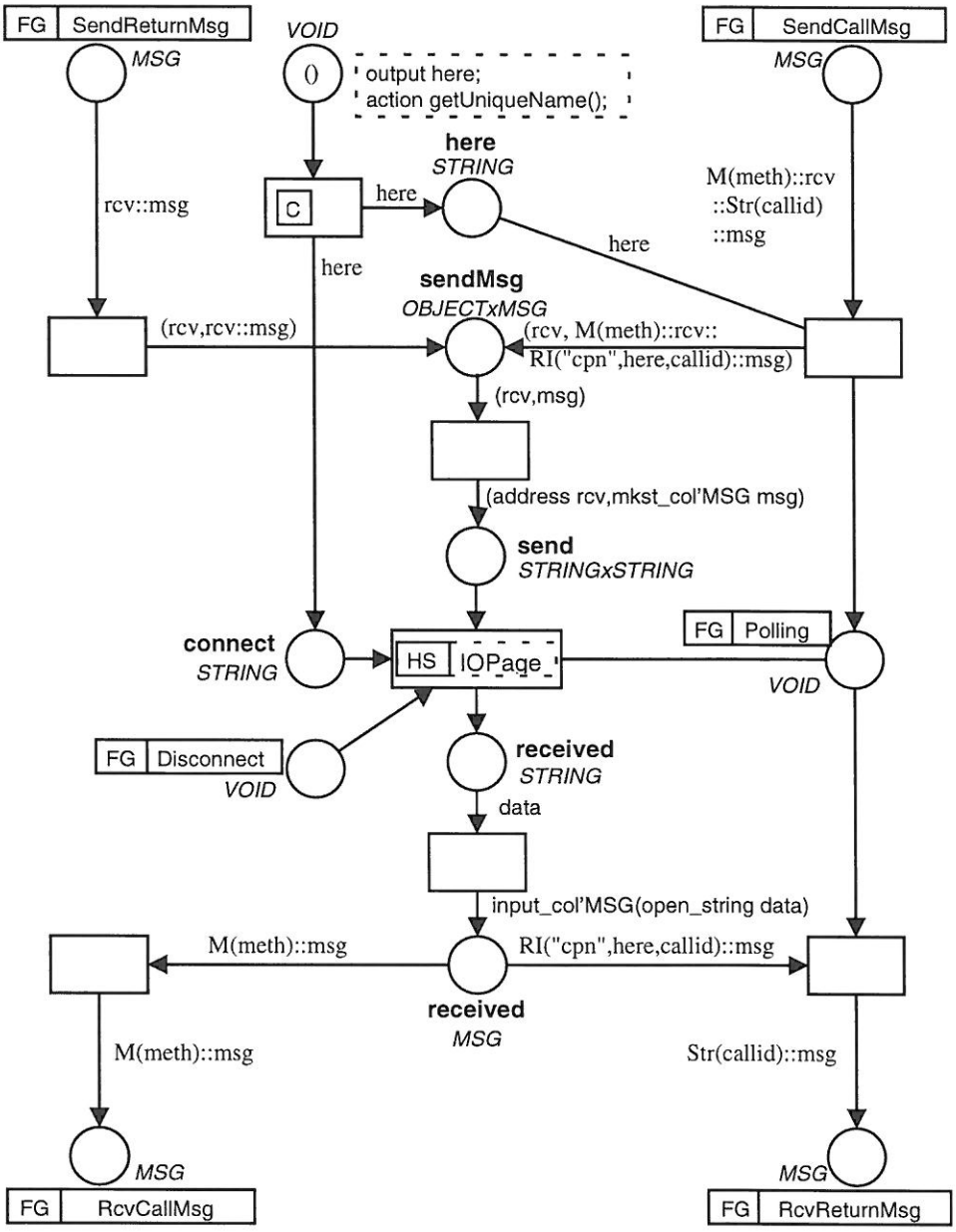



Figure 5: A message translation page

### 3.3 A Message Translation Page

On top of the message I/O page, another page is built that translates the message to and from the string representation which is used externally (see figure 5). Pages like this one must be created individually according to the input and output types that result from the conversion. The pages are expected to share a common structure, so they can be copied from a standard template and then be modified. Since most of the conversion is done by the predefined ML functions `mkst_col'colour` and `input_col'colour`, the effort is usually neglegible.

In our solution, a unique name for connecting to the mailbox is generated by a custom ML function `getUniqueName()` according to the convention described in the previous section. The `disconnect` socket is transformed to a global fusion place, so that the network

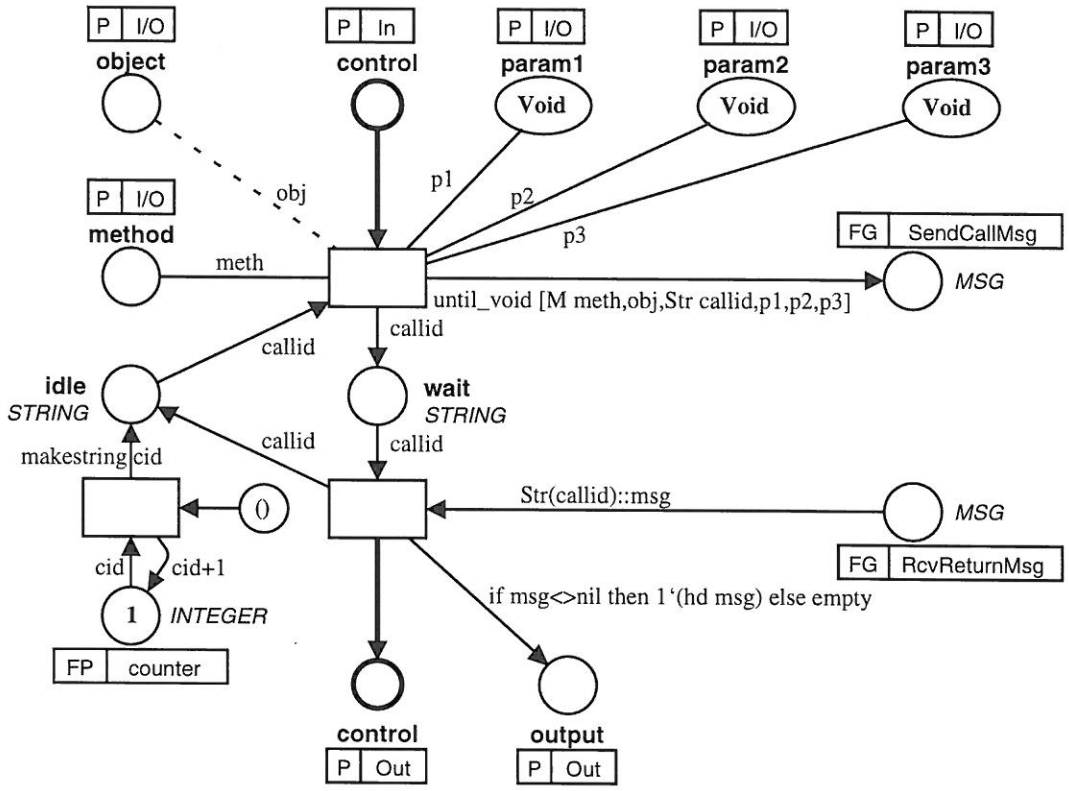



Figure 6: A subpage wrapping calls to Java methods (`Call`)

connection may be terminated from any page in the diagram. Access to the `send` and the `receive` place are provided via global fusion sets, too. Alternatively, the translation page could have defined port places, so that it could have been used as a subpage to a net that wants to send messages. We chose the former solution because the translation page is expected to be used by several other pages or page instances and it is neither necessary nor desirable to get multiple instances of this page. A single, global page is of course more efficient in simulation runs.

### 3.4 A Graphical Petri Net Notation for Method Calls

A further possibility to avoid complex arc inscriptions and to abstract from the message format is to choose a notation that is more adequate for Petri nets. A (Java) method call always goes through the same steps: The message is constructed from the components mentioned in section 3.1, with a unique identifier being constructed as the sender. Then, the message is sent and the caller waits for a return message, which is again decomposed into its components.

Figure 6 shows a subpage that implements this behaviour. The interface consists of all components of a call and the corresponding return message. The maximum number of parameters has been chosen to be three in this net but may easily be extended. No practical problems are to be expected, since the maximum of parameters needed can be determined at compile-time and should not be too high, anyway. Moreover, note the distinction between data and control flow: A special place named `control` indicates the control state of the call, while all other input data is read by test arcs[3] only. A unique ID is assigned to

[3]In fact, Design/CPN does not support test arcs that do not move any tokens. An arc with no arrows is

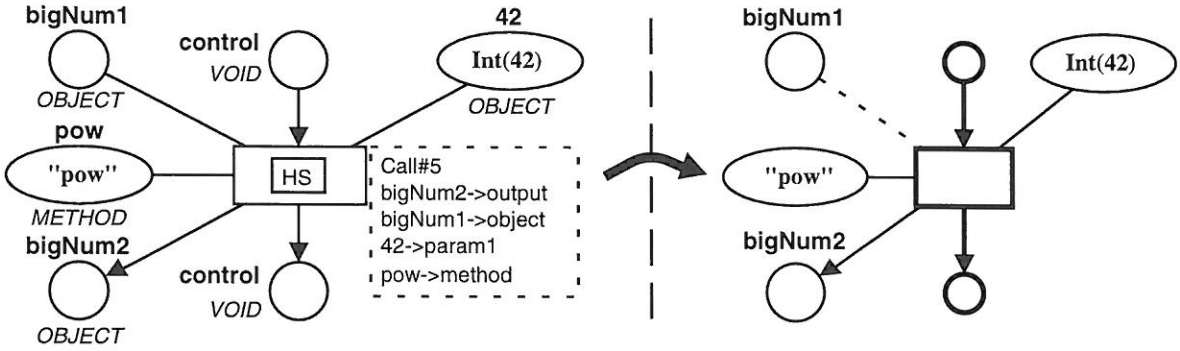



Figure 7: An example of using the `Call` subpage, illustrating the special graphical notation

each instance of the page by using the page instance fusion place `counter`. Each page instance may only invoke one method at a given time (asserted by the place `idle`), but several instances of the `Call` subpage may act concurrently. To handle parameter input, we again take advantage of the Design/CPN feature that the initial marking of a port place is only used if the port place remains unassigned. Thus, all parameter places that are not used by a substitution transition remain `Void` and are suppressed in the construction of the call message by the ML function `until_void` defined in the global declaration node. The `output` port place should only be used if the called method actually returns a result, because then the arc expression will produce no token (not even a black one). A separate control token is produced to indicate that the method call is finished. It may be used to enable the next transition.

To make the nets calling Java more readable, some Design/CPN regions have been suppressed in the following figures. However, a special graphical notation makes the syntax of the nets clear without ambiguity, as may be seen in figure 7, where the second line of Java code from section 3.2 is shown as a net. The left-hand side contains all regions and inscriptions, while the right-hand side shows how these are translated into graphical representations: All control places and control arcs are shown in bold style. They are always of colour `VOID` ("black tokens") as they store control information only. Data places and arcs are shown in normal style and all have the colour `OBJECT` unless they are a `method` place, which can be recognized by their position directly to the left or to the right to a `Call` substitution transition. If an arc has no arc inscription, it is either connected to a substitution transition (Design/CPN neglects inscription of such arcs, anyway) or it has the hidden arc inscription `()` which is one black token.

Any transition that invokes an object method, furtherto referred to as an invocation transition, is indicated by the presence of a dotted arc with no arrows. Since all these transitions are substitution transition refined by the `Call` subpage, the hierarchy substitution region (`HS`) may be omitted. In order to still state a precise port assignment, the following rules apply: The arc connecting a socket place to the `object` I/O port is exactly the dotted one. The input and output control sockets can be recognized as the bold places with input and output arcs, respectively. The input parameters are connected with test arcs (no arrows) in normal style. If more than one input parameter is used, the arcs have to be labelled with the parameter index $pn$ (this notation is not needed in this paper). The output port is assigned to the socket that is connected to the substitution transition via an output arc in normal style.

treated as an arc with arrows in both directions, thus removing and putting back identical tokens. However, this is not semantically different as long as an interleaving semantics is assumed.

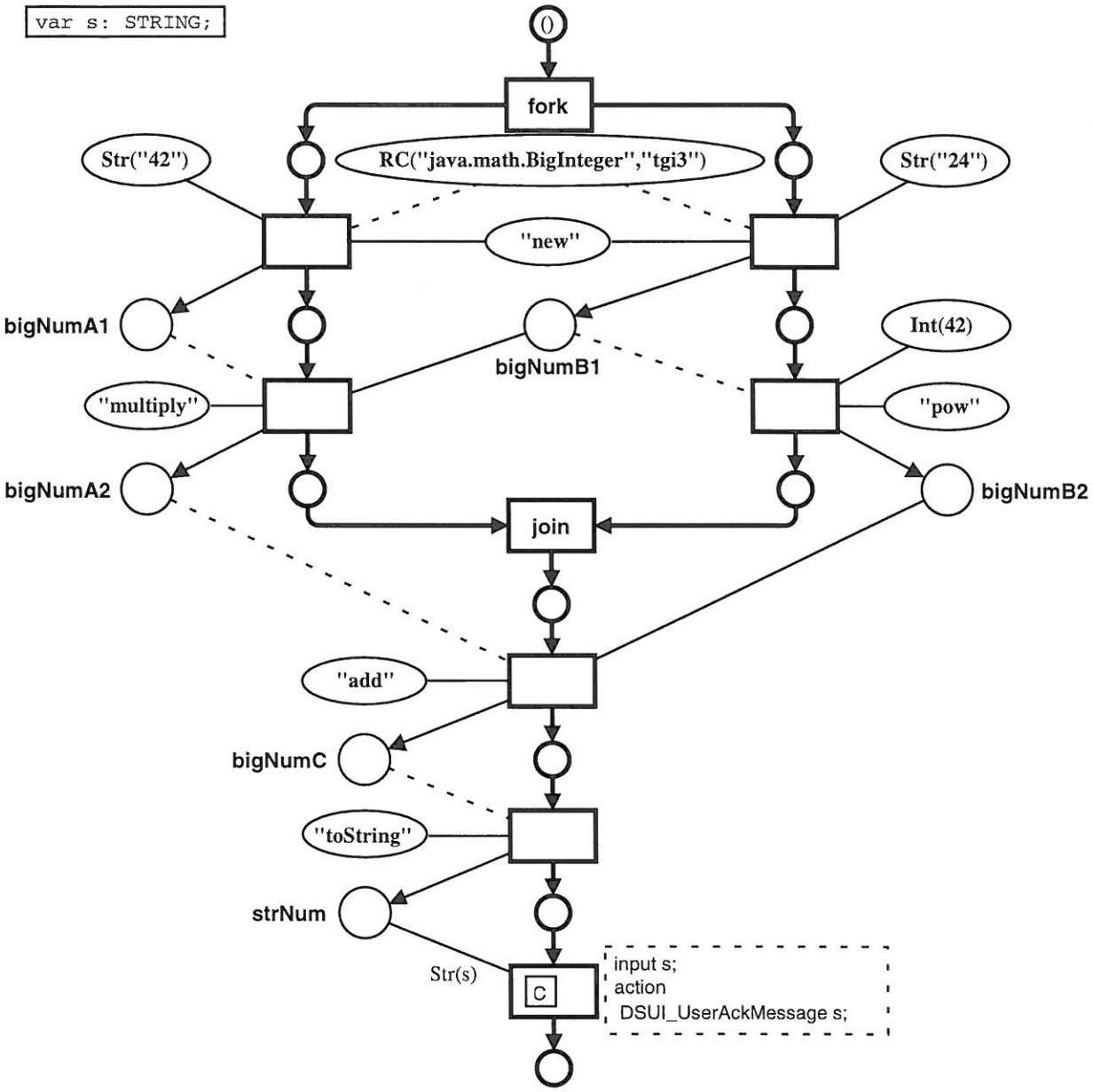



Figure 8: A Petri net performing concurrent calculations with Java `BigIntegers`

### 3.5 An Example Net Calling Java

In the following example net some more features of Java-calling Petri nets are illustrated. Why would we take the overhead of constructing a Petri net to call Java methods? Two main reasons are that Petri nets have a graphical representation (thus, we have something like a visual programming front-end for Java) and that they are superior in specifying concurrency.

In the example in figure 8, again a computation with big integers is performed, but this time, concurrency is exploited. To specify concurrency, we use transitions to fork and to join control flow. Since both "threads" that are created send messages to Java, we have to take care that the return messages are correctly associated. This is ensured by every instance of the `Call` subpage using a different identifier.

There have also been other test applications. In [4] a game was developed that simulates a stock exchange using a Coloured Petri Nets. The net was augmented by calls to a graphical user interface programmed in Java, resulting in a game that can be played by multiple players across a network. The new net shows a dramatic increase in usability that would have been impossible with other tools.

### 3.6 Processing Method Calls in Java

We now briefly describe how the Java side of our framework treats the incoming messages. As a first step, a message is converted into an internal representation using a straightforward top-down down-parser.

Now some remote references might point to an object of the local Java process. Class references are immediately resolved to ordinary Java classes.[4] Then the framework has to translate the remote instance references into local Java objects using a special table of externally known objects.

Afterwards the framework determines the class of the invoked object and uses the standard `reflection`[5] package to get a list of all public methods which the object supports. It chooses the appropriate method from the list and calls it in an individual thread. Now other calls can be concurrently received and parsed.

After the completion of the method call, the result is sent back to the caller whose address was specified in the message. If a reference to a Java object is returned, the framework generates a unique external name for it and inserts this name in the table of externally known objects. Design/CPN will then receive the message and forward the result.

If a Design/CPN process interacts with multiple Java processes, remote references might point to an object of a remote Java process instead of a local Java object. This case, too, is handled by our framework, e.g. one could, if desired, store a reference to an object on one process in a hashtable on another process.

There are two possibilities for a CPN process to get to know Java objects. To start with, such an object may be created by a special program that also invokes the framework and inserts the object into the table of externally known objects under a well-known name.

Then, the CPN process may access the `new` method of a class reference. In section 3.2 we have given an example of such a call. In fact, the message looks like a normal method call, which is not the case in the Java programming language, where `new` is a keyword with a special syntax. To keep the message format clean, we decided to use `new` like a static method provided by every class instead.

This allows the creation of arbitrary Java objects from a Design/CPN net, even for built-in classes like hashtables, windows, etc. Although this device is extremely powerful, a note of caution has to be: The creation of new objects using the `new` method provides complete access to the Java environment, thereby opening up huge security holes. But Java programs can deliberately reduce their access rights by means of the `SecurityManager` interface. If this does not prove sufficient, it is still possible to protect the access to either the Java framework or to the entire message handling system by passwords or other techniques.

[4] Java allows the loading of classes at runtime requiring only a string representation of the class name. Even the creation of classes at runtime is possible.

[5] The `reflection` mechanism is a powerful feature of the Java environment that allows the complete analysis of objects, classes and methods at runtime. Thus, Java programs can view a *reflected* image of themselves as if in a mirror. Additionally, the modification, the creation, and the invocation of objects is supported.

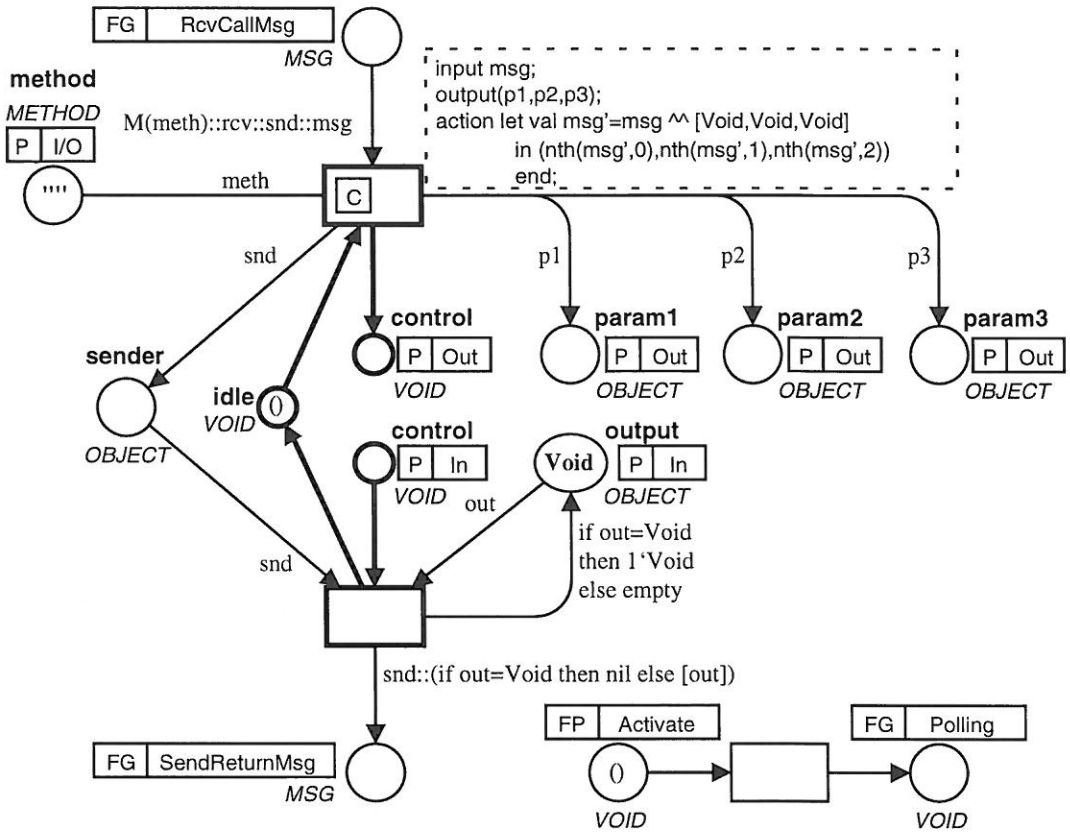



Figure 9: A subpage for calls to a CPN (`CPNCall`)

# 4 Accessing Design/CPN from Java

In the last section, we have shown how to call Java from Design/CPN. This approach is useful when the major part of the application is modeled with Design/CPN and Java is used to complement the application.

In other cases, an application or at least large parts may have already been implemented in Java, but Petri nets are utilized for modelling use cases or workflows that are contained in the application. Then, one may wish to design and run these parts with Design/CPN and call Petri nets from Java. The basic idea is that a system modeler can use the most appropriate tool for representing and solving the problem at hand.

All one has to do to extend our approach to support this feature, is firstly to implement generic Java proxy objects that call Design/CPN nets via messages. Secondly, Design/CPN nets have to be extended to be able to receive and process call messages, not only return messages. In a sense, a Design/CPN process has to behave like a single (static) object, thus providing some methods that may be called.

## 4.1 Designing a Subpage for Calls to Design/CPN Nets

Again, we have tried to find a very general solution, i.e. a net subpage that can be re-used for any Design/CPN net implementing some method call. Figure 9 shows such a subpage which is basically the counterpart to the net presented in figure 6. The two transitions at the bottom ensure that the net is able to receive messages all the time, in contrast to a net that just calls Java methods and polls for return messages if some call has been sent.

The upper transition checks all incoming messages whether they are call messages to the method name given in the port place `method`. If so, the parameters contained in the call message are distributed into separate places (again, the maximum number of parameters is restricted to three) and a token is put into the `control` output port place to specify that the "parent" net may start now. The receiver of the message is ignored, because the whole net is treated like a single object. If this net had not been the receiver of the message, the message would not have been delivered there. The `idle` place is needed because this simple version of calling a net does not support concurrent calls to the same net, while it does allow concurrency within the net. This restriction is lifted when our object Petri net approach is used as discussed in the conclusion. After the parent net has finished its task, it has to put a token into the `control` input port place. It should remove all input parameter tokens as well as all tokens that were produced during the run, lest the next call fails or produces unexpected results.[6] When a socket place is assigned to the `output` place, some token has to be present that is used as a return parameter. The subpage then constructs a return message to the sender of the call message and puts it into the global fusion place `SendReturnMsg`, so that it is sent by the `Message` page and the `IOPage`.

## 4.2 An Example of a Net that can be Called

To illustrate the use of this subpage, we specify a net that implements a method executing some example workflow by W. v.d. Aalst, cited in [14].

The example was introduced as follows. When a criminal offence happens and the police has a suspect, a record is made by an official. This is printed and sent to the secretary of the Justice Department. Extra information about the history of the suspect and some data from the local government are supplied and completed by a second official. Meanwhile the information on the official record is verified by a secretary. When these activities are completed, a prosecutor determines whether the suspect is summoned or charged, or whether the case is suspended.

In figure 10, a workflow for this case is modeled using our object-oriented Petri net notation introduced in section 3.4, extended by the feature that an invocation transition may consume input parameters. In the upper part, the interface places of the method `crimeCase` are connected to the subpage `CPNCall` through a substitution transition. The arcs that point to the bottom border of the figure are actually connected to this transition, too. Note that tokens for the places `official1`, `official2`, `secretary`, and `prosecutor` have to be provided elsewhere. We do not give any further details on how the invocation transitions might be refined. In fact, the activities may be implemented as code regions, as subpages, in Java or even as other nets that may be called through the message interface.

In this case the call of the workflow from Java simply looks like

```
RemoteCPN cpn=new RemoteCPN("cpnhost:12345");
String decision=(String)cpn.execute("crimeCase","Roger Rabbit",
                                    "murder of Marvin Acme");
```

assuming that `cpnhost:12345` is the hostname / process-ID of the Design/ML process (and thus its mailbox name) simulating the workflow net, that the suspect is Roger Rabbit, and that he is accused of murder, where the decision of the prosecutor (as well as the input parameters) is implemented as a `String`.

[6] It is not too complicated to check this property automatically using place invariants.

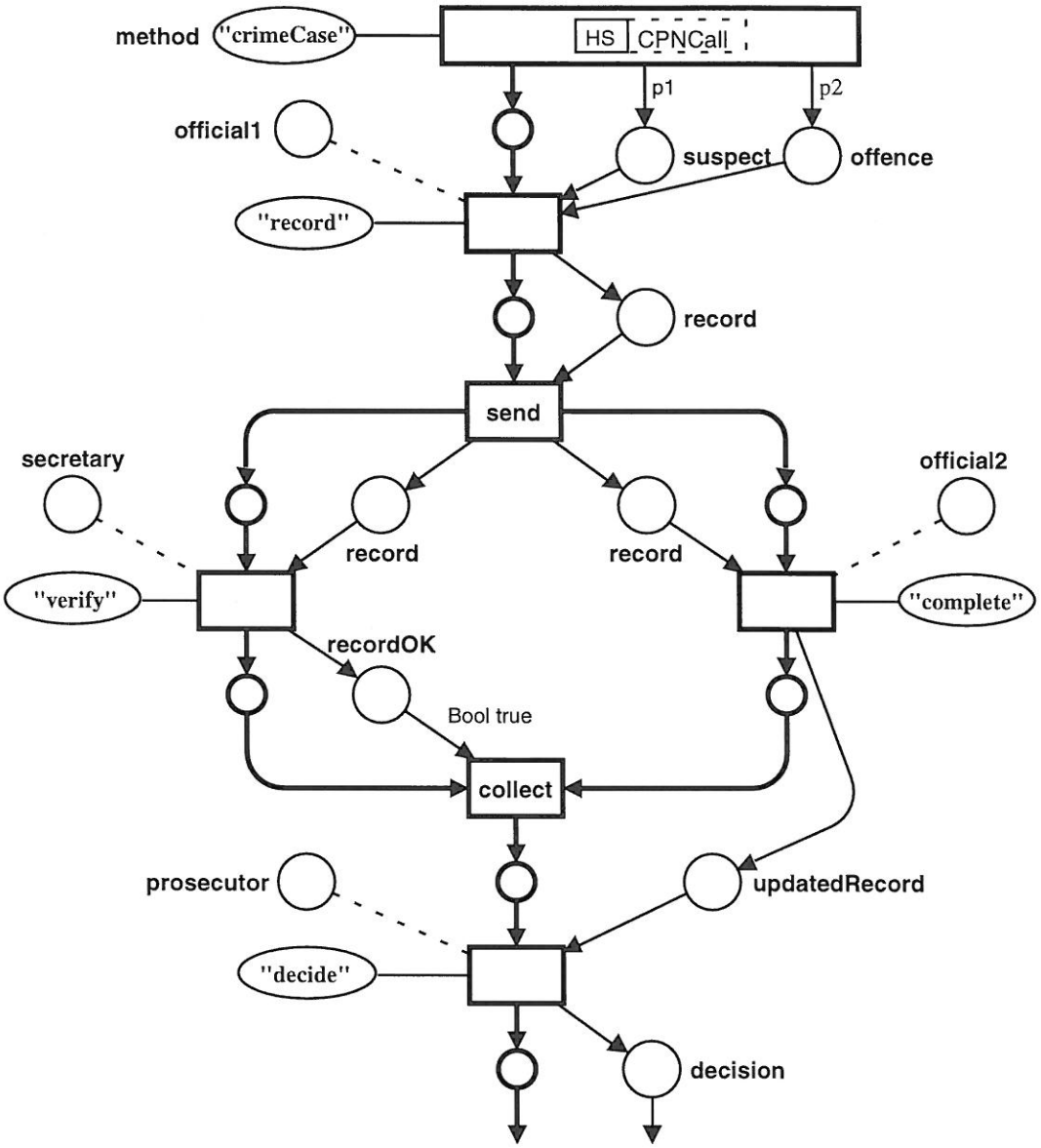



Figure 10: An example of a net that can be called from Java

## 5 Possible Applications and Benefits

Petri nets are already a concurrent formalism, so a net model documents the intended concurrency without the need to distribute it. But there remain at least two cases where a distributed net model is favourable even in the absence of interaction with Java:

- Need for performance gains. Whether any significant performance improvements are possible depends on several factors. It must be possible to split the application into parts that communicate by messages. This is often naturally the case, but existing net models might not show the possible splitting lines clearly. What is more important is that the nets should require as little communication messages as possible. Alternatively, the net might be demanding in terms of ML evaluation time, e.g. animation or optimization algorithms. In both cases, the communication cost might be dominated by the computation cost.

- Real concurrency. The Design/CPN simulator is a single-threaded application. Thus, no real concurrency, not even multi-threading is available. Multi-threading would be especially desirable if complex or time-consuming ML functions are executed, which is normally done within one simulation step, delaying all transitions. Now, code may be moved to a separate diagram (or any external program, see below), enabling the calling net to continue working while the computation is performed. The drawback of polling for the answer is in some cases preferable compared to blocking the whole simulation.

- Necessary distribution. Some applications require access to distributed resources, e.g. a visualization module that accesses various screens or a game that needs input from many players. Such applications cannot naturally be realized without a distributed simulator.

However, the greatest benefit of our communication framework comes from the interoperability with other programming environments:

- The access to Java processes enables the developer of a net model to incorporate much more complex GUIs into a net. This improves the interaction with the user of the net, but might also be used to animate and visualize the simulated process or to generate more elaborate statistics and debugging information.

  Although there are some GUI libraries for Design/CPN already—the most notable one being Mimic/CPN described in [13]—they do not match the flexibility of Java or comparable languages. Moreover, there is a lack of rapid prototyping tools which greatly speed up the GUI development.

- Distributed computing also allows multiple users to participate in a single simulation from different terminals.

- Processes that are controlled by Design/CPN might also handle general I/O devices. This may simplify the control OF a system by a net, a possibility already mentioned in the context of the security system presented in section 1.5 of volume 3 of [7]. There it was proposed to extract parts of the ML code generated from the net for the execution on a stand-alone microprocessor. Such a task might be considerably simplified if the connection to the outside world remains the same during the translation.

- It becomes possible to reuse code that was not developed in Design/CPN or ML. In this area the standard container classes come to mind immediately, but in fact there is a wide range of programs for Java covering almost all aspects of algorithms and data structures as well as various I/O and network libraries.

There are additional benefits when Coloures Petri Nets can also be called from the outside:

- It is possible to move gradually from a Petri net prototype to an implementation using Java. Although this will require that the nets themselves are structured in an object-oriented way, it remains a viable route.

- The specification of workflows with Petri nets has attracted much interest in the past years. Using the framework it seems possible to use Coloured Petri Nets for executable prototypes of workflows and in the future maybe even for final products.

## 6 Outlook

The framework presented here improves the usability of Design/CPN in some of the most important areas: distribution, interoperability, appropriate modelling, and graphical user interfaces. A smooth transition from a specification within Design/CPN via distributed Design/CPN modules to distributed Java modules seems possible. Some applications have already been implemented thereby documenting the gained flexibility.

In the near future we are going to extend the framework to object-oriented Petri nets in the sense of [8], [11], and [10]. In those approaches the structure of the nets represents most object-oriented features without extensions of Coloured Petri Nets themselves. Tool support of the object-oriented techniques could then further simplify the development process.

Additionally, Artificial Intelligence concepts as already used in [12] are going to be extended by providing a connection to Prolog. This will allow us to use available tools for logic programming directly from Design/CPN models.

If Design/CPN itself was extended by multi-threaded simulation of net-models, this would capture some concurrency aspects of our framework. The distribution and interaction with Java would still be essential benefits of our framework, which would even be improved, because blocking input could be used instead of polling.

Our communication framework already bears some resemblance to other distributed object-oriented architectures. Although many functions and just as many concepts are still missing, it does no longer seem far fetched that Petri nets might one day be usable with systems like CORBA [2].

## 7 Acknowledgments

Much of the Java implementation described in this paper was done during a project at the University of Hamburg. We would like to thank the participants for their efforts, especially Matthias Ernst, Torben Kroll, Heiko Rölke, and Eberhard Wolff.

A message passing library in C was implemented by Heinrich Biallas and has been documented in [1].

In [4] Margret Freund-Breuer and Olaf Fricke developed the stock exchange model that was used as a first test case for our Java interaction framework.

## References

[1] Heinrich Biallas. Realisierung der verteilten Ausführung von gefärbten Petrinetzen. Diplomarbeit, Universität Hamburg, Fachbereich Informatik, November 1997.

[2] The OMG Corba Page. WWW page at `http://www.omg.org/corba/`. Contains references to the current specification of CORBA/IIOP (on 05-15-98, this is version 2.2), a discussion of the Object Management Architecture, and many more links.

[3] Design/CPN Online. WWW page at `http://www.daimi.aau.dk/designCPN/`.

[4] Margret Freund-Breuer and Olaf Fricke. Spezifikation mit gefärbten Petri-Netzen am Beispiel des Börsenspiels. Studienarbeit, Fachbereich Informatik, Universität Hamburg, September 1993.

[5] Dirk Hauschildt. A petri net implementation. Fachbereichsmitteilung FBI-HH-M-145/87, Universität Hamburg, Fachbereich Informatik, 1987.

[6] The Java Home Page. WWW page at `http://java.sun.com`. Contains references to online material as well as to many introductory books and technical papers.

[7] Kurt Jensen. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, 1992: Vol. 1, 1994: Vol. 2, 1997: Vol. 3.

[8] Christoph Maier and Daniel Moldt. Objektorientierte Konzepte – Dargestellt mit gefärbten Petrinetzen. Fachbereichsbericht FBI-HH-M-261/96, Universität Hamburg, Fachbereich Informatik, August 1996.

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

[10] Daniel Moldt. *Höhere Petrinetze als Grundlage für Systemspezifikationen.* Dissertation, Universität Hamburg, Fachbereich Informatik, August 1996.

[11] Daniel Moldt and Christoph Maier. Coloured Object Petri Nets - A Formal Technique for Object Oriented Modelling. In B. Farwer, D. Moldt, and M.-O. Stehr, editors, *Petri Nets in System Engineering, Modelling, Verification and Validation*, pages 11–19, Fachbereich Informatik, Univ. Hamburg, 1997. FBI-HH-B-205/97.

[12] Daniel Moldt and Frank Wienberg. Multi-agent-systems based on coloured petri nets. In Pierre Azéma and Gianfranco Balbo, editors, *Application and Theory of Petri Nets 1997*, number 1248 in Lecture Notes in Computer Science, pages 82–101, Berlin, 1997. Springer Verlag.

[13] Jens Linneberg Rasmussen and Mejar Singh. *Mimic/CPN – A Graphics Animation Utility for Design/CPN. User's Manual Version 1.5.* Computer Science Department, Aarhus University, December 1995.

[14] Rüdiger Valk. On Concurrency in Communicating Object Nets, 1998. Accepted for publication at the International Conference on Application and Theory of Petri Nets (ICATPN).

# Executable Models of Influence Nets Using Design/CPN

**Lee W. Wagenhals, Insub Shin, Alexander H. Levis**
System Architectures Laboratory
School of Information Technology and Engineering
George Mason University
Fairfax, VA 22030, USA

## Abstract

This paper describes a methodology for converting an influence net to an executable model, implemented using the Colored Petri Net formalism and tools, so that it can be used to assess the impact of a set of controllable events or actions on outcomes of interest; specifically the impact of various sequences and timing of those actionable events. With this methodology, alternative courses of action, first defined using influence nets, can be refined by adding sequence and timing information for analysis and comparison. The techniques developed offer a means to integrate intelligence and operational planning models to improve course of action development. The paper includes a description of the automated algorithms that convert an influence net to a Colored Petri Net and illustrates how that model can be used for the analysis of alternative courses of action.

## 1 Introduction

In the command and control environment, planning and selecting specific courses of action to achieve objectives and goals involves two types of modeling and analysis activities (Figure 1). The first involves models that attempt to assess potential events and outcomes based on incomplete and uncertain understanding of both physics based and perception based processes. Such activity is primarily the forte of intelligence analysts. Probabilistic modeling techniques that provide inferences are often used to suggest what outcomes might occur given sets of controllable actions and uncertain exogenous events. The second type of activity, planning, involves the generation and evaluation of detailed actions and activities to accomplish the mission goals. In general detailed models and algorithms are available for planning the courses of action to be taken.

These two classes of activity involve different ways of viewing problems, and the models created have different characteristics. In many cases, the executable models used in planning require parameters that can be derived from the probabilistic models. It follows, then, that being able to place both modeling techniques in a common environment can enhance the overall planning process.

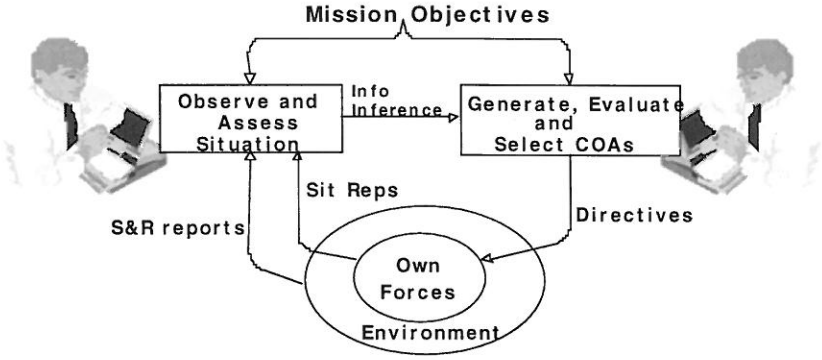



Figure 1. High Level Model of Major Command And Control Activities

This research concentrated on influence nets which are one type of probabilistic modeling. In particular, the software application, Situation Influence Assessment Module (SIAM) developed by SAIC [Rosen and Smith, 1996] was used. Design/CPN, was chosen as the target environment into which the influence net models would be converted by an automatic algorithm written using ML[1] code and many of the functions built into the Design/CPN application.

The remainder of this section provides a brief description of Influence Net modeling, motivation for the research, and a statement of the problem. Section 2 describes the conversion algorithm including design considerations and decisions. Section 3 provides an example that illustrates the process of course of action development and evaluation. The last section summarizes the results of this effort and includes a brief discussion of future directions.

### 1.1 What is an Influence Net

Influence nets are an extension to traditional Bayesian inference nets, sometimes called Bayesian belief nets or causal probability networks. Bayesian nets have both a graphical and a mathematical component and are used to model complex domains where uncertainty exists. They are directed acyclic graphs that represent the factorization of joint probability distributions of n random variables. They include an inferencing procedure for updating the joint probability distribution via Bayes rule as new information is received about any of the random variables. Thus they can be used as powerful modeling constructs for diagnostics and estimating outcomes given evidence. Because their construction requires the creation of marginal and conditional probability tables, they tend not to be easily accessible to decision makers not familiar with probability theory. Influence nets were developed to extend the traditional Bayes net structure and allow the creation of useful models by analyst unfamiliar with probability theory or who are unable to spend the time needed to fully specify a complete Bayesian net.

Influence net modeling incorporates an intuitive graphical influence diagramming technique for model construction. The SIAM implementation used in this investigation automatically creates a Bayesian model that allows for rigorous analysis using non mathematical inputs from the analyst. To construct an influence net, a modeler creates "influence nodes" that depict events that are part of a set of cause-effect relationships appropriate to the situation being modeled. The modeler also creates influence links between the nodes to graphically illustrate causal relationships between events. Each influence link can either be promoting or inhibiting, as identified by the terminator of the connecting link (arrow head for promoting, ball for inhibiting).

Figure 2 shows a simplified and hypothetical example of the topology of an influence net that might be created by intelligence analyst to assess a potential military situation and evaluate the effectiveness of a combination of diplomatic and military options. This net represents a situation where a Country B is in a posture that is threatening to its neighbors, creating instability. Country G is contemplating conducting a covert mission to destroy a key facility in Country B. If successful, the mission should reduce the confidence of the leader of Country B. The covert mission will be more effective if it can occur at the same time as a diplomatic mission visits a neighboring country. Additional influence will occur if the international community sanctions against Country B.

[1] ML stands for Meta Language. It is a functional programming language.

With SIAM, in order to generate the underlying quantitative marginal and conditional probability values, the modelers indicate, through a graphical user interface (GUI) the "strength" of each influence link to either promote or inhibit the effect event at the head of each link given the event at the tail. This approach is based on "Causal Strengths" logic [Chang, et. al. 1994] and the algorithm for converting causal strengths to marginal and conditional probability transition matrices to evaluate the cumulative likelihood of any event in the influence net using traditional probability calculations. Once constructed, influence nets can be used by analysts and decision makers to examine events over which they have some control to determine which events have the best chance of creating the outcomes desired by the decision maker.

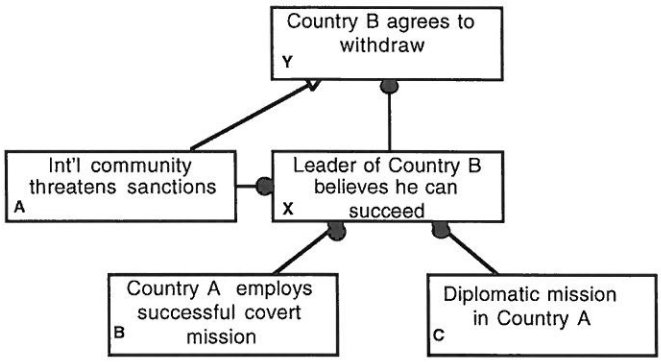


Figure 2. Topology of an Influence Net

### 1.2 Motivation for this effort

Influence nets produce static equilibrium models that relate cause to effect. There is no time or dynamic behavior captured in such models. They are most useful in assessing situations involving perceptions, rationale, and decisions. They are particularly useful if there is no underlying "physical" phenomenon that can be represented in analytical and executable dynamic models or if the underlying physical phenomena are not known.

One of the prime uses of influence nets is the evaluation of chains of causes and effects to determine which actions have the greatest likelihood of causing the desired outcome. In this sense, the influence net helps a decision maker decide *what* should be done. Detailing *how* the actions should be executed, in many cases, will involve the time-phasing of the actions. This can be modeled more effectively by a discrete event dynamical system. In many cases, such models require input parameters that can be estimated from influence nets. This effort was motivated by the notion of converting influence net models into an environment that can incorporate discrete event dynamical models so that both situation assessment models and dynamic models of courses of action can be interconnected to bring together the advantages of both modeling techniques into a single modeling environment. With such a process, it will be possible for analysts familiar with situation assessments to create influence nets in the environment with which they are most familiar and then to transfer their work in an automated way into an environment suitable for the execution of discrete dynamical models for evaluation of the selected courses of action.

### 1.3 Statement of Problem

The objective of this research was to design, develop, and test an algorithm that could take an output file from an influence net modeling software application and automatically generate an executable Colored Petri Net model. This model could then be interconnected to other executable CPN models. The Colored Petri Net model would perform the same forward probability propagation calculations that the influence net application performs. In addition, the algorithm should allow time delays to be added to the probability propagation at various nodes, allow easy input of marginal probabilities to the initial nodes for evaluating impact of different events, and allow the user to designate nodes for which a time history of change in probability could be collected for analysis.

## 2 Design of the Conversion Algorithm

In this effort, a UNIX-based influence net modeling application called Situation Assessment Module (SIAM) developed by Rosen and Smith (1996) was used. The design process started by understanding both the information available within a SIAM generated model and the calculations used by SIAM to propagate marginal probabilities given a change in probability of the initial events in the influence net. This understanding was used to design an output file that would be generated from SIAM and design a set of Colored Petri Net subnets that would replicate the probability propagation process. A multi step process for reading that file and converting it to a Colored Petri Net that would implement the forward probability propagation calculation was then developed.

This section first describes the SIAM processes and information and gives a brief description of the multi step process that accomplishes the conversion. This is followed by a more in depth description of the conversion process including descriptions of the Colored Petri Net that is created.

### 2.1 Influence Net Calculations and Export Information

The initial requirements for the design of the conversion algorithm focused on the probability calculations of SIAM that would be replicated in the Colored Petri Net. The calculation to be implemented is as follows.

Each node in the influence net represents some variable of interest and the variable is Boolean, in that it represents a logical statement that is either true (T) or false (F). Probabilities are associated with each node X expressed as a marginal probability P[X]. For each node in the influence net that has n parents ($n > 0$), let $Q_i$ be the $i^{th}$ set of parent states. A set of conditional probabilities, $P[X|Q_i]$, can be defined for each set of parent states of node X. SIAM uses the simplifying assumption that all the parent states are independent. With this assumption, letting $P[q_j \in Q_i]$ be the marginal probability of the $j^{th}$ parent state in $Q_i$, then the marginal probability of that node X is

$$P[X] = \sum_{i=1}^{2^n} \left[ P[X \mid Q_i] \times \prod_{j=1}^{n} P[q_j] \right]$$

$$q_j \in Q_i$$

Case 1: {**P(X|A, B, C, D**}

Case 2: {**P(X|¬A, B, C, D**}

• •
• •
• •

Case 16: {**P(X|¬A, ¬B, ¬C, ¬D**}

Figure 3. Conditional Probabilities for Node with Four Parents

Given this calculation, the following information was needed from SIAM to create the conversion: for each node in the influence net, the name of the node and its identification number, its position in x and y coordinates on the diagram, the number of parents of the node, its initial marginal probability, and a vector of the $P\{X|Q_i]$ representing the complete set of conditional cases for that particular node. For example a node **X** with four parents, **A, B, C,** and **D,** would have $2^4 = 16$ conditioning cases as shown in Figure 3.

For this effort, a special export routine was added to SIAM by Rosen and Smith that generates a text file with the above information for any influence net created in SIAM. Figure 4 shows an example of part of such an export files

### 2.2 Overview of the conversion algorithm

To meet the objectives of the procedure described in Section 1.3, a seven step algorithm was created in Design/CPN 3.0 which allows code to be written in the ML language to read files and create and draw CPN models based on the information in those files. The following briefly describes each step.

1. Read the text file from the Influence net application. For each node in the influence net initialize appropriate variables with:

```
"Caesar Baseline Net"          / Net Title
93                             // Number of Nodes

1                              / ID of Node @X=1411 Y=248
"Country A employs SOF mission"      // Name of
0.52025                        // Marginal Belief of Node
1                              / Number of Parents
88                             // Parent 1 "Country B selects options "
0.00050                        // (1|88) = (1|F)
0.77650                        // (1|88) = (1|T)

2                              / ID of Node @X=30 Y=34
"UN flies inspection flight" // Name of Node
0.66650                        // Marginal Belief of Node
0                              / Number of Parents

3                              / ID of Node @X=5 Y=294
"UN threatens sanctions "// Name of Node
0.66650                        // Marginal Belief of Node
0                              / Number of Parents

4                              / ID of Node @X=7 Y=199
"Leader of Country B believes he will succeed"// Name of Node
0.13845                        // Marginal Belief of Node
3                              / Number of Parents
1                              / Parent 1" Country A employs SOF mission"
2                              / Parent 2" UN flies inspection flight"
3                              / Parent 3" UN threatens sanctions
president"
0.93573                        // (4|5,59,3) = (4|F,F,F)
0.38379                        // (4|5,59,3) = (4|F,F,T)
0.80214                        // (4|5,59,3) = (4|F,T,F)
0.12467                        // (4|5,59,3) = (4|F,T,T)
0.01429                        // (4|5,59,3) = (4|T,F,F)
0.00141                        // (4|5,59,3) = (4|T,F,T)
0.00464                        // (4|5,59,3) = (4|T,T,F)
0.00046                        // (4|5,59,3) = (4|T,T,T)
```

Figure 4 Partial Sample of Export File

node ID number
node name
node location
conditional probabilities list
list of parents
node marginal probability

2. Create a page containing auxiliary boxes and arrows that replicate the topology of the influence net. For each node in the influence net, draw an auxiliary box and label it with the node name and ID number. Using the List of parents for each node, draw the arcs from parent nodes to child notes. A sample of a replica of a 93 node influence net drawn with the algorithm is shown in Figure 5. For each node on the page, use the Design/CPN function that returns lists of input nodes and output nodes to generate a 3 tuple composed of 3 lists: a list of the node ID numbers of the parents (inputs), a list containing the node ID number, and a list of the node ID numbers of the children (outputs). These tuples of lists will be used in later routines to create the Petri Net.

3. Partition the net into sets of nodes that will be placed on individual pages in the Colored Petri Net.

4. For each page, create and interconnect subnets for each node of the original influence net that perform the probability propagation and updating for each node of the influence net.

5. Designate fusion places that interconnect each of the pages so that the topology of the Colored Petri Net is the same as that of the influence net.

6. Create a control panel page that allows the initial nodes of the net to be given marginal probability values and provides places to collect a history of the changes in marginal probability of nodes designated by the analyst.

7. Initialize the net by reading a file containing time delay information for each node.

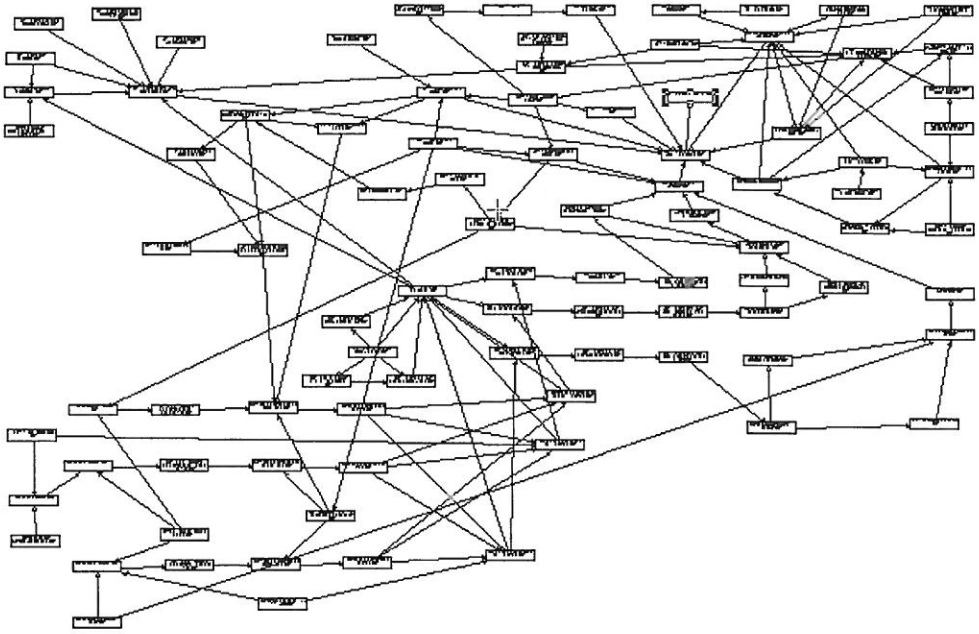


Figure 5 Sample Influence Net

The ML code is contained in several auxiliary boxes on a page in Design/CPN as summarized in Table 1. Each box of code is executed, one box at a time, until the complete CPN model has been created. At the end of each step, an appropriate dialog box appears indicating the step has been completed.

| ML Code Boxes that Comprise the Conversion |
|---|
| 1. Read the Input File; Construct Influence Net |
| 2. Cluster Groups of Nodes for Paging |
| 3. Draw Each Page of the Colored Petri Net |
| 4. Create Fusion Place Ports and Sockets |
| 5. Create a Control Panel Page |
| 6. Initialize the Net with Timing Information |

Table 1. ML Code Segments for Reading SIAM File and Building CPN Model

Once the Colored Petri Net has been constructed, a syntax check is made and the model is switched to the simulator mode. Once in the simulator, the model can be executed using different markings in the Control Panel Page that represent probabilities of the input nodes. The resultant history of changes in probability of the designated nodes is contained in the output places and can be saved as text files for analysis using a spread sheet program such as Excel. In addition to evaluating the influence diagram in the Colored Petri Net, existing Design CPN models representing the details of courses of action can be loaded into the appropriate pages of the influence net model and connected using fusion places.

### 2.3 Conversion Algorithm Design Details

Several constraints were encountered that had to be addressed in the design of the conversion algorithm. First, the current version of the influence net modeling application

(SIAM) creates one flat net on a single page. This is not a problem for small nets of less than 15 to 20 nodes; however, larger nets cause the equivalent CPN model to have an excessive number of objects on a single page in Design/CPN adversely affecting the run time of the simulation. The solution was to partition the CPN model into multiple pages that can be interconnected significantly decreasing the execution time of the CPN model.

Several schemes for accomplishing this partitioning were considered. To achieve the best performance, the goal of any partitioning scheme should be to create clusters of nodes that minimize links between clusters. Each cluster will be place on its own page in the Colored Petri Net. Our initial design used a simplistic partitioning scheme by clustering nodes based on node identification number. For example, if a network had 60 nodes, and it was decided to partition the net into four pages, page 1 contained nodes whose identification numbers were between 1 and 15, page two contained nodes 16 through 30 , etc. Since node numbers are assigned in a random fashion by the influence net modeling utility, it is not likely that this method will result in an optimal partitioning.

An algorithm to automatically create an effective clustering has not been written at this time as this was not the central focus of this research. In the currently implementation it has been left to the analyst to "read" the diagram and select the groups of nodes for each cluster. In general, humans heuristically can do a good job of creating a set of clusters that have a minimum or near minimal number of interconnections. At this time, the analyst determines the clusters and then enters the ID numbers of the nodes that are in each cluster. In future research we will implement one of several graph theoretic algorithms to do the clustering automatically.

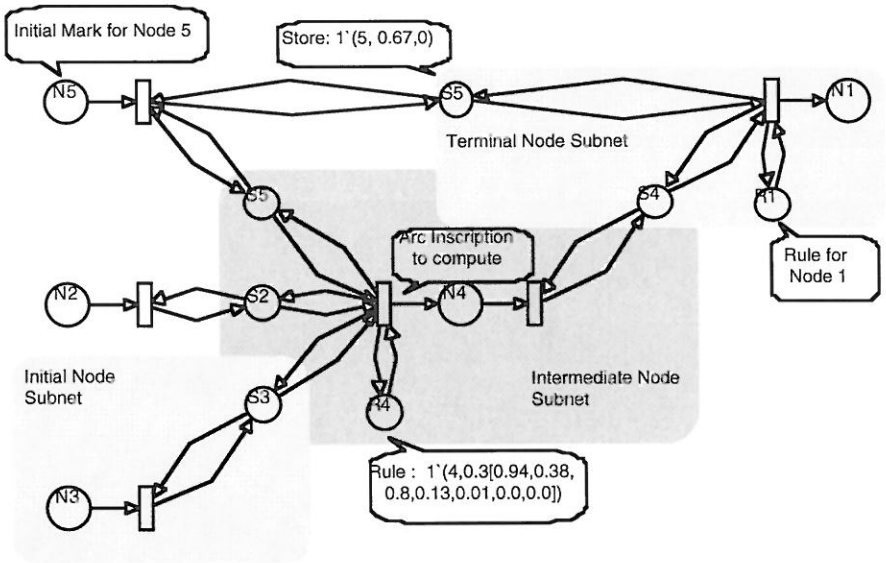



Figure 6. Structure of CPN of Influence Net

In step 4, the 3 tuples generated in Step 2 (they contain lists of parents and children for each node along with the lists indicating which nodes go on the different pages generated in Step 3) are used to generate the Colored Petri Net. Each node of the influence net is converted to one of three standardized subnets depending on whether the node is an initial node (one with no parents), a terminal node (one with no children) or an intermediate node (one with at least one parent and at least one child). Figure 6 shows the structure of the Colored Petri Net generated from a five node influence net with the topology of the influence net in Figure 2. Its is composed of five standardized subnets, three for the initial

nodes (only the one for node 2 is highlighted) and one each for the intermediate and terminal nodes.

Figure 7 shows the CPN in more detail with the arc inscriptions and guard functions needed to understand how this structure executes. The Global Declaration Node for the CPN is shown in Figure 8.

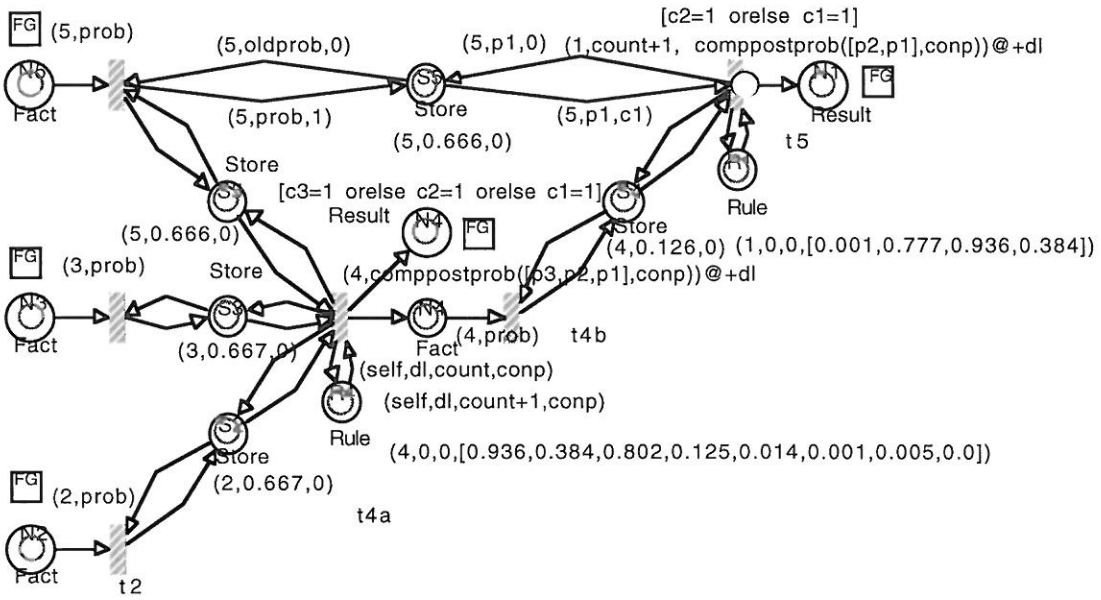



Figure 7 Annotated Color Petri Net

The Colored Petri Net uses four types of places. Places of the timed color set Fact contain tokens that are a tuple composed of node ID number (integer) and the probability (real). Places of color set Store hold tokens that are a 3 tuple of Node ID number, the probability of the node, and a control (integer). Places of color set Result hold tokens that are a 3 tuple of Node ID, a counter (integer) used to determine the sequence number of the token, and the probability of the node. Places of color set Rule hold a single token that is a 4 tuple composed of the node ID number, the time delay associated with that node, a counter, and a list of the conditional probabilities for that node.

In Figures 6 and 7, the places labeled N5, N2, and N3 (color set Fact) are initial nodes corresponding to nodes **A, B,** and **C** in the influence net of Figure 2. Each of their subnets is composed of a single transition with a single input place and as many output places as there are children of the node. These output places, labeled "S", are of color set Store connected in a self loop for updating the marginal probability of the node. Figure 7 shows the arc inscriptions that remove the token with the "old probability" and replace it with a token with the new "probability." Notice that when a new token is placed in the Store place, its control is set to 1.

Each intermediate node in the influence net is converted to a subnet structure with two transitions that are interconnected with a single place of labeled "N" of color set Fact. The first transition is connected in a self loop with the output Store places of each of its parents (so it can "read" the marginal probability of its parents). The arc inscriptions show that, when the transition fires, it removes the input tokens and replaces them with a token with the same probability but with the control set to zero. This first transition is also connected

with a self loop to a place labeled "R" (color set Rule) that contains a list of the conditional probabilities for that node and a sequence number. Each time the transition fires, it reads the conditional probability list, the sequence number and the time delay and returns a token with the same conditional probability list, and time delay and increments the sequence number by one. The output arc from the first transition to the N place has an arc inscription that computes the marginal probability of the node given the marginal probabilities of the parents and the list of conditional probabilities contained in the rules The arc inscription uses the computpostprob function contained in the Global Declaration Node to perform the probability calculation contained in the equation in section 2.1. Its input arguments are the list of probabilities that are read from the parent Store places using variables p1, p2,...and the list of conditional probabilities contained in the rule using variable conp. The N place is timed and the arc inscription incorporates the time delay contained in the rule.

Because its input places always contain tokens, the first transition is enabled whenever the guard function evaluates as true, i.e., when at least one of the control variables of the input tokens is equal to one, meaning a new update has occurred. For example, in Figure 7, the guard function for the first transition of node 4 is [c3=1 orelse c2=1 orelse c1=1].

The second transition in the intermediate subnet is connected in a self loop to a Store place for each child. Note that the second transition with its preset and poset have the same structure and arc inscriptions as the subnet for the initial nodes. Said another way, the structure of the initial node is the subnet of the intermediate node obtained by removing the first transition and its preset.

```
(* ===== Global Declaration Node =====*)

color FactID = int;
var f,self: FactID;
color Control = int;
var c1,c2,c3,c4,c5,c6,c7,c8,c9,c10: Control;
color Delay = int;
var td,dl: Delay;
color Counter = int;
var count: Counter;
color RealCS = real;
var prob,oldprob,p1,p2,p3,p4,p5,p6,p7,p8,p9,p10 : RealCS;
color RealList = list RealCS;
var conp,pli,problist: RealList;
color Rule = product FactID  * Delay * Counter * RealList;
color Fact = product FactID * RealCS timed;
color FactTime = product FactID * RealCS * Delay ;
color Store = product FactID * RealCS * Control;
color Result = product FactID *  Counter * RealCS 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 comppostprob(pli,problist)=multlist (map (probdet pli)
(map (detbase (length(pli))) (constlist (length(problist))))) problist;
```

Figure 8 Global Declaration Node

The subnet structure for each terminal node is similar to the structure of the intermediate nodes without the second transition (N1 is the only terminal node in Figures 5 and 6). There is a slight difference; the output place, which is of color set Result, contains a sequence number that ensures the list of output tokens are ordered in the sequence in which the transition fired.

Notice that the intermediate node subnet also has a Result place connected as an output of the first transition of that subnet. This place and its arc are automatically generated for each node designated by the user to collect time history data. It has the same arc inscription as the terminal node.

Before describing how the algorithm builds the Colored

Petri Net just described, it is instructive to understand how the net executes. As previously stated, the final Colored Petri Net contains initial markings for all Store places and all Rule places. The control element of each Store token is initialized to zero and the counters in the Rule tokens are initialized to zero. In the absence of any tokens in any Fact places, no transitions will be enabled. The CPN performs the forward probability propagation whenever a token is entered into a Fact place. This normally occurs in the initial node input places. When such a token is added, it sets off a chain reaction that characterizes the probability propagation. For example in the CPN of Figure 7, if a token 1`(2, 1.0)@[0] is introduced into the N2 place, transition t2 will fire replacing the token in S2 with a token (2. 1.0, 1). This will enable transition t4a because the control has changed to 1. Transition t4a will "read" all three probabilities of its parents and compute a new posterior probability. The result will be placed in the Result place with sequence number 1 and in the intermediate Fact place. If there is a time delay associated with this node, the token will be given the appropriate time stamp. When the simulation time advances, transition t4b will be enabled and fire, updating place S4. This will trigger the same action for node 1 as occurred with node 4. The result will be Stored in Result place 1. Of course several tokens can be placed in the initial input places to see the effect of several events. Each token precipitates a chain reaction like the one just described.

In Step 4, the algorithm that draws the CPN executes a sequence of actions. First, it draws the appropriate subnet for each node of the influence net . The subnet drawn depends on the 3 tuple that contains the lists of parents and children generated for each node in step 2. For example, for each intermediate node, a subnet like the one in Figure 9 will be drawn on the page in the same location as its counterpart in the influence net.

Next the algorithm draws the Store places, one for each arc in the influence net drawn in step two. If two nodes are on the same page, the Store place is drawn halfway between the first transition of the parent and the second transition of the child. Then, one pair of arcs with arc inscriptions are drawn between the Store place and the second transition of the parent place and a second set of arcs is drawn between the Store place and the first transition of the child. If the two nodes are on different pages, the Store place of a parent is drawn at a fixed location near the first transition and is designated as a fusion place. Arcs are then drawn between the first transition of the child node and this Store place. Note that the second set of arcs is not drawn at this time. This procedure is completed for each page until all nodes have be constructed.

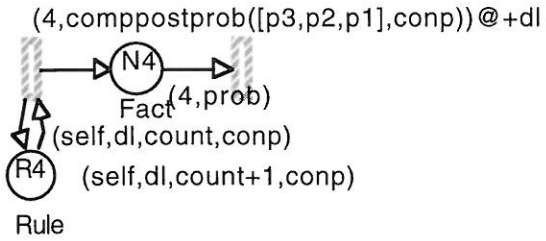



Figure 9. Initial Subnet for Intermediate Node

In Step 5, the algorithm uses the list that associates each node with its children to connect the pages together using fusion places. This only needs to be done when a parent and child are on different pages. Starting with the first page, the algorithm searches until it finds a node with a child on a different page. It then creates a Store place near the second transition of the node subnet and adds it to the group of fusion places that was created on the child's page when that fusion place was created in Step 4. The algorithm continues until all nodes have been searched and, as a result, all fusion places appropriately grouped.

In step 6, the algorithm creates a separate control panel page. The control panel page enhances the usability of the network by providing a convenient single input place to input

the probabilities of initial nodes in the net and places to collect the output data of any node in the network as designated by the analyst[2]. The control panel page for the CPN in Figure 6 is shown in Figure 10. All of the places are fusion places. The box in the middle is an auxiliary node that serves no function other than indicate the relationship of the control panel page to the other pages in the Colored Petri Net. The figure shows the form of the output data that is collected in each Result node. Notice that the listing of the tokens is ordered by sequence number so that it gives a time history of the probability state of the node on which the data is collected.

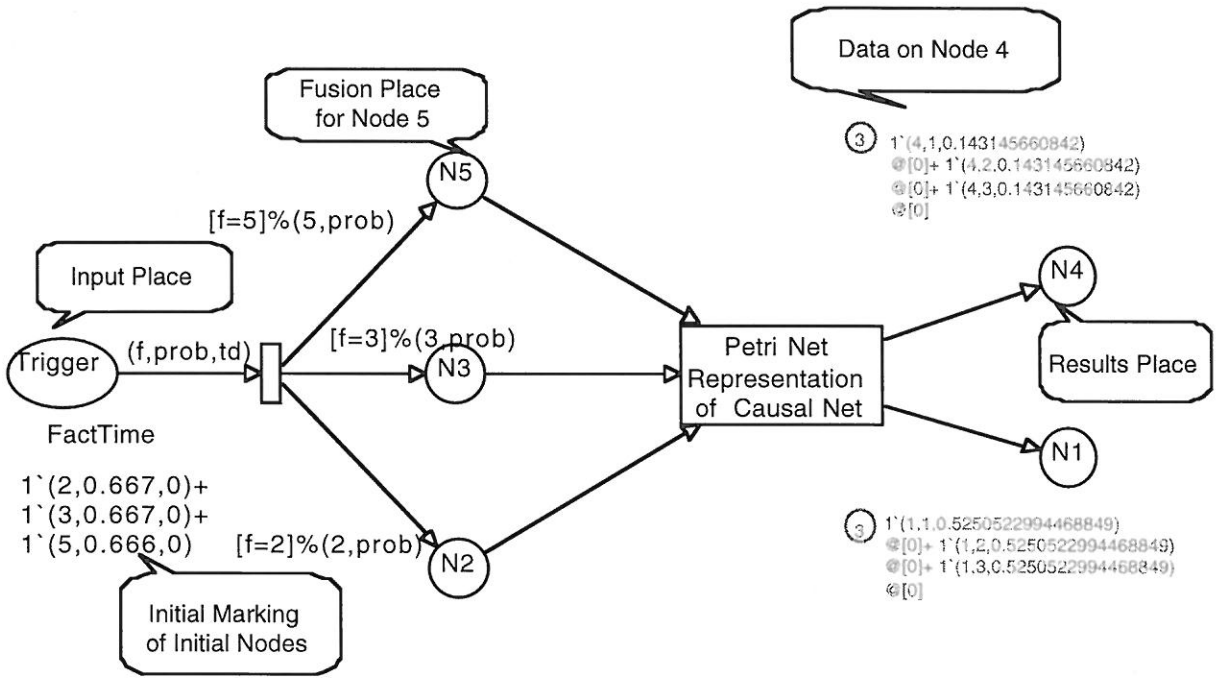



Figure 10. Control Panel Page

To create the control panel page, the algorithm searches each page for initial nodes and designates their input places as fusion places. It also searches for Results places and designates them as fusion nodes. The algorithm then creates a new page and creates Fact places for each input node and adds them to the appropriate group of fusion places. If performs a similar operation for the Results places. The algorithm creates the Trigger place (color set Fact Time) and the transition shown in Figure 10, and draws the appropriate arcs. Finally, an auxiliary box is created and auxiliary connectors are drawn been each fusion place and the auxiliary box.

In the last step, the algorithm creates the initial marking regions for the entire network. It first provides a dialog box asking the user to navigate to and select the text file that has the time delays associated with each node in the influence net. After reading this file, it creates the initial marking regions for all the rule places using the variable values assigned when the SIAM export file was read in Step 1. It also creates the initial marking regions for every Store place in the net. Finally, it creates an initial marking region for the Trigger place on the control panel page by creating a list of the marginal probabilities for the initial nodes and converting it to an initial marking like the one shown in Figure 9.

[2] The analyst can "instrument" the net by designating the nodes from which a history of the probability changes will be recorded. An additional instrumentation place is automatically added to each subnet of these nodes to store status tokens each time a node is updated.

### 2.4 Difficulties encountered and overcome

The development of the algorithm was initially accomplished on both Motorola 68030 CPU and PowerPC Macintosh computers using Design/CPN 3.0. The code was then ported to a SUN Sparcstation Ultra 1 with little difficulty.

During the development, several difficulties with Design/CPN were encountered for which work arounds were devised.

As was mentioned in Section 2.3 we found that when using any influence net of reasonable size, the net had to be partitioned to decrease the simulation run rimes. We also found a significant difference in performance with different computers. For example, the time to perform the conversion with a Power Macintosh 7500/100 was approximately 2 hours compared to 25 minutes for the SUN Ultra 1 for a 93 node influence network. We found a similar 4 to 1 improvement on execution of the simulator for the same net.

One difficulty occurred when running the application on the PowerPC Macintosh. It is our conjecture that there is a limitation on the maximum number of tokens that can be present in any one place. This limitation was exceeded in Macintosh versions of the algorithm when using a influence net containing 93 nodes and 23 initial places. The Macintosh version of Design/CPN allows one to observe the running of the ML code. We were unable to do this in the UNIX version. Observation of ML code revealed error messages of "out of range" once the number of tokens exceeded approximately 900 in the single terminal place of the 93 node net; thus our conjecture on the maximum number of tokens.

The design relies on a great deal of information encoded on the tokens. In particular, the size of the Rule token grows exponentially with the number of parents. We had several nodes with 8 or 9 parents. This generates lists of conditional probabilities that have 256 or 512 real numbers. We conjecture that some of the problems we had occurred because we exceeded memory limitations in these tokens. The problems disappeared when we used a rounding function to hold the precision of probability calculations to three significant digits. The large number of characters in the string of rules also makes the viewing of large networks difficult. Figure 11 shows a portion of a page for a 93 node net with many of the arc inscriptions suppressed.

One problem that was difficult to overcome occurs when saving the state of the CPN model in the simulator. If each page of the CPN model is designated as a prime page and then the state is saved, the resulting file will contain a corrupted multiplicity factor of minus 5147 for several of the pages. Of course the model will not execute properly. The reason for this anomaly is not known. The following work around, suggested by Kjeld H. Mortensen of the University of Aarhus, Denmark, eliminated the problem. When the model is created in the editor, only the first page of the net is designated as prime. After switching to the simulator , the state can be saved and reloaded without any problems. Of course after loading the state, all of the pages have to be designated as prime. The initial state command is then issued using the CPN pull-down menu, and after the re-initialization that now includes all the pages, the model can be executed to collect data for analysis.

## 3.0 Example

The following example illustrates the procedure for creating an influence net of a politico-military situation, using it to select high pay-off actionable events, converting the net to a CPN, and executing the CPN to collect data for course of action evaluation.

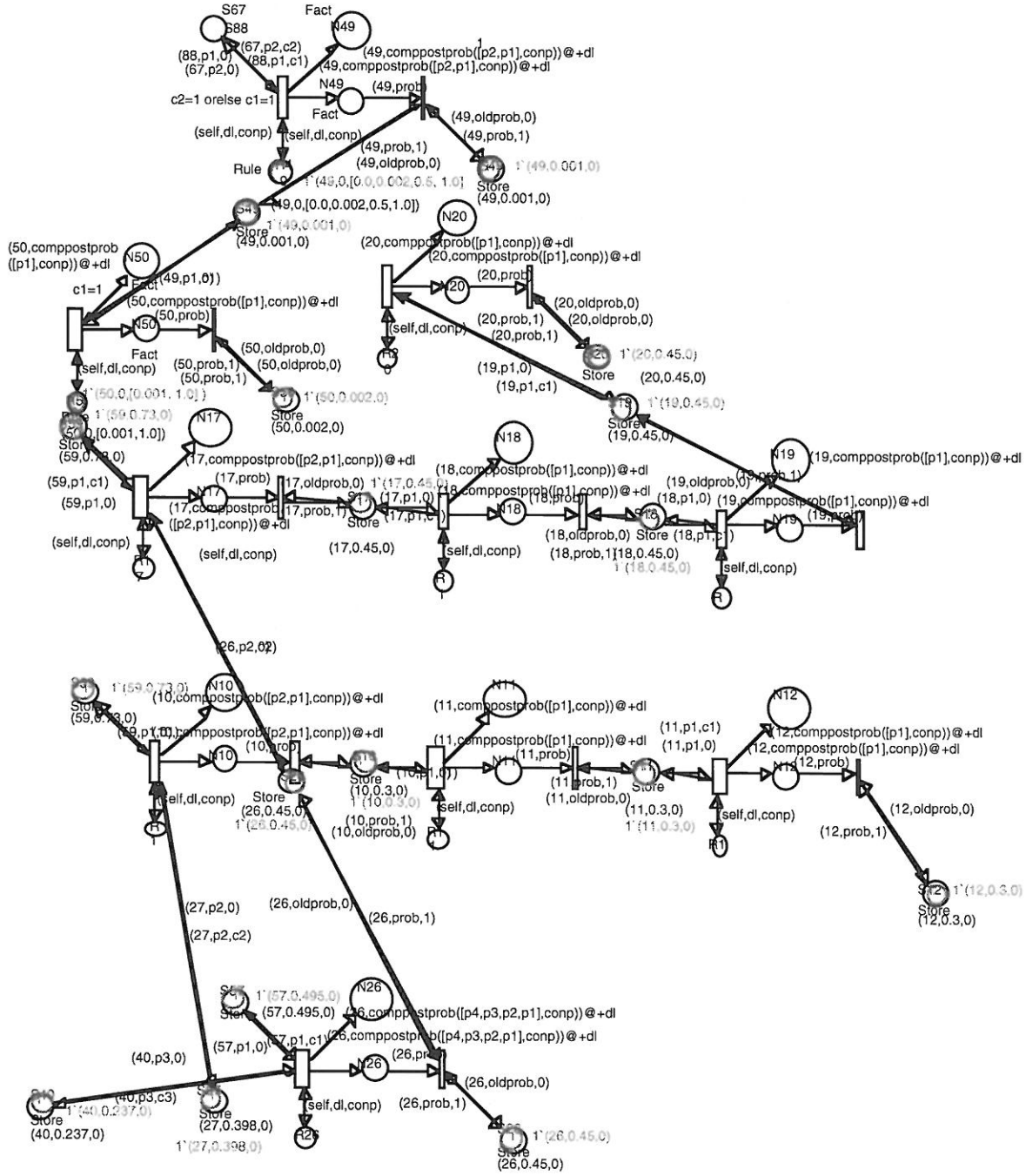



Figure 11. Portion of One Page of 93 Node Net.

In this example, a hypothetical country, Witmania has undertaken a program to develop and produce weapons of mass destruction (WMD). Country G, the good guys, seeks actions to cause Witmania to stop its program. The intelligence community of country G has sufficient knowledge about the decision making processes of Witmania to create an influence net. The net has one overall objective or target node: "Witmania stops its WMD program." From this target node, nodes representing various diplomatic efforts, surveillance flights, and psychological operations that can effect this decision along with their causal links are created. The influences of actions and perceptions of the leader of

Witmania are also included. The intelligence analysts also include several military actions, including the use of a covert mission to destroy the WMD production facility, the use of a Unmanned Air Vehicle (UAV) as a diversion, and a neutral flight. The influence net models the reaction of the Witmania integrated air defense system (IADS) to the covert operation, UAV and neutral flights. The overall result is a large influence net created in SIAM with over 90 nodes, including 23 nodes for actionable events. Figure 3 in section 2 is a thumbnail view of the structure of the influence net.

The intelligence analyst conducts a sensitivity analysis of the influence net to determine which controllable events have the greatest impact on the target node, stopping the WMD program. This analysis indicates that while diplomacy has some impact, the combined effects of the covert operation, UAV, and neutral flight significantly increase the likelihood that the WMD program will be stopped. As a result, it is decided to investigate combinations of these actions in more detail. Figure 12 shows a notional operational concept for the various courses of action under consideration.

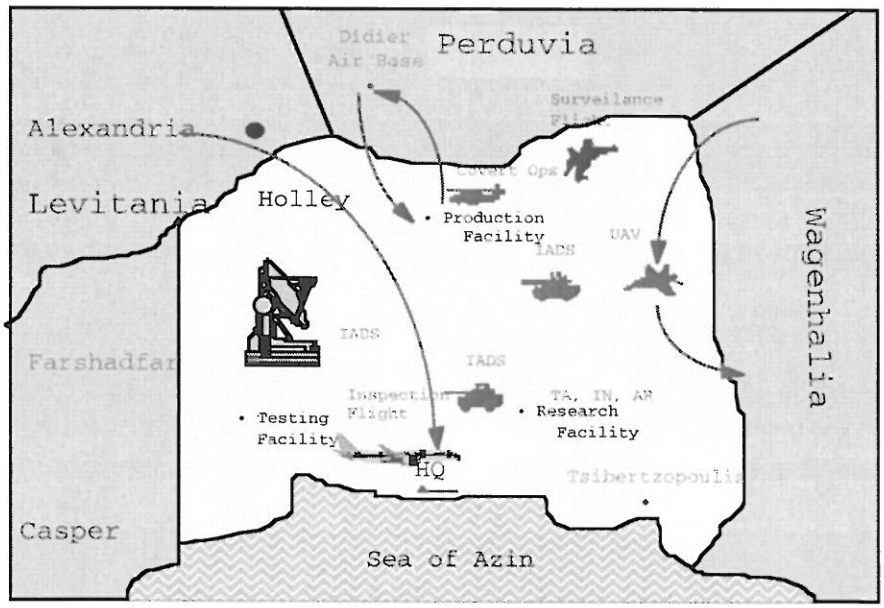



Figure 12. Operational Concept for Witmania

Using the output file from SIAM and the conversion algorithm created in Design/CPN the influence net is converted to a colored Petri Net. Figure 11 shows a portion of one of the pages of the CPN. First, analysis of the sequences is conducted to eliminate any sequences that will not have the desired impact. Then the timing information is added to the best sequences to analyze the temporal aspects of the courses of action.

It was decided to include the diplomatic events in all courses of action because these actions would take place prior to military action and may produce the desired results. The diplomatic efforts are composed of a sequence of four actions: Initial diplomatic efforts, surveillance flights, using the results of the surveillance flight to advertise the existence of the WMD program to the international community, and local psychological operations to influence public opinion against the WMD program. Figure 13 shows how the probability that the WMD program will be stopped increases from 26% to 48% as this sequence of diplomatic actions takes place.

Since the sensitivity analysis using SIAM indicated that the target node probability would increase significantly if a covert mission and UAV flight were conducted in conjunction with a neutral flight, the CPN model is executed using the complete set of 13 combinations of sequence of these actions shown in Table 2.

| Neutral Flight | UAV | Covert Mission |
|---|---|---|
| 1 | 1 | 1 |
| 1 | 1 | 2 |
| 1 | 2 | 2 |
| 1 | 2 | 1 |
| 1 | 2 | 3 |
| 1 | 3 | 2 |
| 2 | 1 | 1 |
| 2 | 1 | 2 |
| 2 | 2 | 1 |
| 2 | 1 | 3 |
| 2 | 3 | 1 |
| 3 | 1 | 2 |
| 3 | 2 | 1 |

Table 2. Sequences for 3 Actions

The control panel page of the CPN model has been set up to collect probability data on several intermediate nodes in addition to the target node. These include the probabilities that the Integrated Air Defense System (IADS) stands down due to the neutral flights, that the UAV survives, that the covert mission evades the IADS, and that the overall cover mission is a success. Figure 14 shows the results for the third sequence of Table 2: the neutral flight followed by the covert mission concurrently with the UAV. Similar charts are created for the 12 other sequences. The 13 charts show that the covert mission success dominates the outcome of the target node. The neutral flight increases the likelihood that IADS will stand down and the UAV has a 60% chance of survival. The overall conclusion is that the UAV provides little cover for the covert mission; covert mission success is independent of the UAV flight.

After reducing the number of candidate courses of action by analyzing the impact of the various sequences of actions, timing data including processes times for the various nodes in the net and starting times for the actionable events in the CPN model, is added to the model. For example, estimates of the time delays in passing information through the

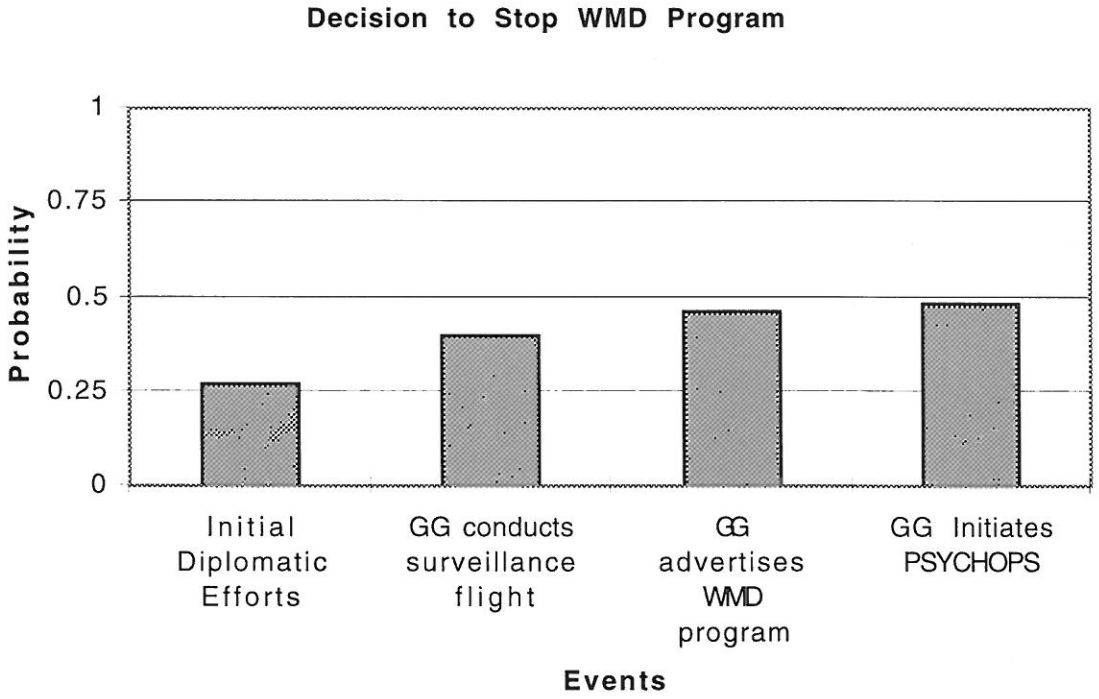



Figure 13. Changes in Probability of Target Node Due to Diplomatic Actions

IADS and the time required for decisions to be made to stand down the IADS or have the IADS engage the covert and UAV missions are added to the appropriate nodes of the CPN.

In addition, the planned time for the flight of the covert mission, the UAV and the neutral flight are incorporated in the tokens representing these events. The model is executed again for several combinations of these event times and data collected on the key nodes of the model.

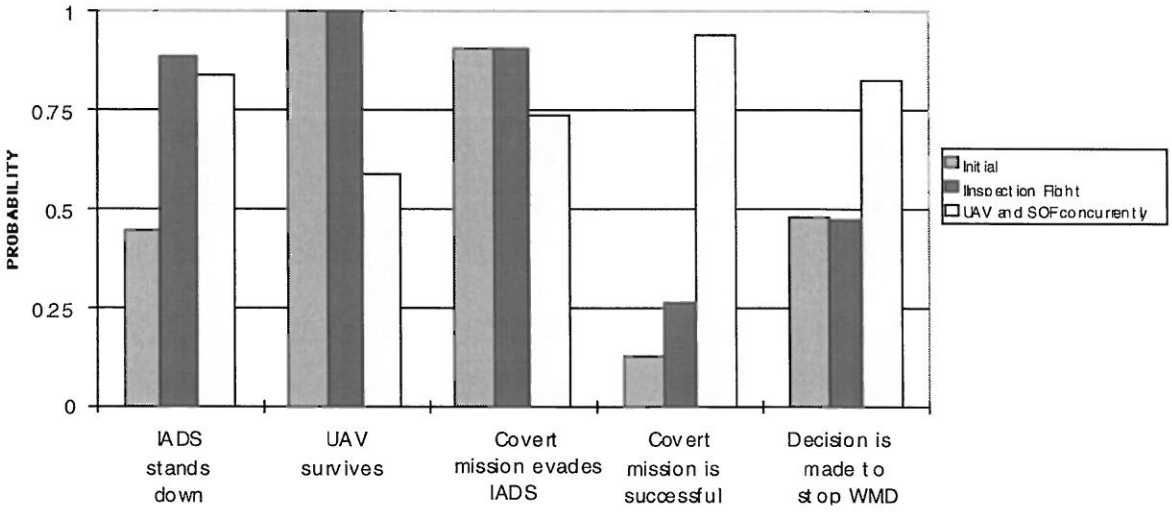



Figure 14. Results of Inpsection Flight Followed by SOF and UAV

The data collected indicate the time phased changes in probability of each event for which data was collected. Figure 15 shows an example to the changes in probabilities that the IADS will stand down due to the neutral flight and that the UAV will survive. Note that the advent of the neutral flight causes the probability of the IADS standing down to increase. As the UAV enters the IADS air space its probability of survival begins to decease until approximately 25 minutes into the mission. Then there is a slight increase due to the interaction with the neutral flight. As the UAV mission lingers in the air space beyond 35 minutes, its probability of survival again decreases. This suggests that if the UAV is to be used, it should be kept in the IADS airspace for no more than 35 minutes. Other time histories of the probabilities of events are examined until the best time windows are determined.

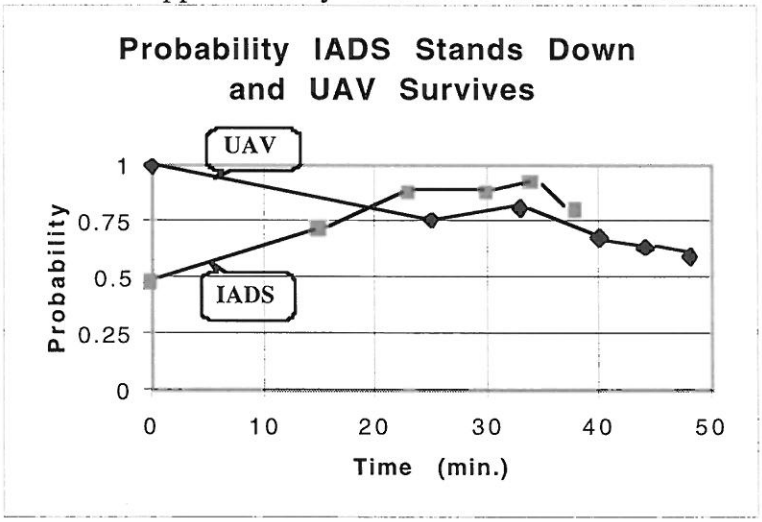



Figure 15. Timing Comparison of Two Events

## 4 Summary and Areas for Further Research

This work represents a successful integration of influence nets with discrete event models in a common environment. It is believed that this technology will allow a closer coupling between models designed to assess situations and compare potential courses of

action, and dynamical models that can be used to provide detailed planning and evaluation of those courses of action.

To date we have not connected together any real operational models and influence nets in this common environment. Future research is planned to do this and develop guidelines and rules for constructing and interconnecting such models that is consistent with both the underlying mathematics of the models and the objectives of the enterprise. In addition, a sound methodology using the interconnected models for developing and evaluating courses of action based on measures of performance and measures of effectiveness for the mission objectives needs to be developed. When fully established, this capability will enhance understanding of collaboration between teams of experts, each using tools and models appropriate to its discipline for situation assessment and course of action generation and evaluation.

## Ackowledgements

This work was sponsored, in part, by the United States Air Force Research Laboratory (George Mason University participated as a sub contractor to QuesTech, Inc.) and by the Office of Naval Research under contract no. N0014- 93-1-0912. Special thanks goes to Mr. Steve Hendricks and Mr. Larry Simmons of QuesTech for their contribution in defining the hypothetical scenario used in this effort, and to Dr. Julie Rosen and Mr. Wayne Smith of SAIC for there assistance with SIAM.

## References

[Buede and Wagenhals, 1996]. Dennis M. Buede and Lee W. Wagenhals. *Influence Diagram Representation of Dynamic, Distributed Decision Making*, Proc.of the Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey CA, June 25-28, 1996.

[Chang *et al*, 1994] K.C. Chang, Paul E. Lehner, Alexander H. Levis, S. Abbas K. Zaidi, and Xinhai Zhao. *On Causal Influence Logic*, Technical, George Mason University, Center of Excellence for C3I, December 3, 1994.

[Rosen and Smith, 1996] Julie A. Rosen and Wayne L. Smith. *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.

# Experimental Symbolic Analysis of Net Systems

**Hartmann J. Genrich**
GMD – Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven, D–53754 Sankt Augustin, Germany
hartmann.genrich@gmd.de

## 1 Introduction

This note addresses certain aspects of using Petri nets in the analysis of computerized systems: engineering systems, manufacturing systems, workflow systems, communication systems – whatever Petri net models are used for in practice. Any such system when adequately presented by a Petri net I call a *net system*. And throughout this note *Petri nets* means *higher-level* ones like those represented by *PrT-nets* in theory and *Design/CPN* in practice.

Technically, the note contains little new for those familiar with Petri nets and with the notions of *S-invariants* and *T-invariants* in particular, the solutions of homogeneous linear equation systems based on the incidence matrix of a Petri net. Rather, I wish to give a twist to the understanding and usage of invariants. I hope to change our focus from S-invariants to those *state variables* or *S-quantities* that may or may not be *constant*, and from T-invariants to those *changes* that may or may not be *cyclic* and to the *process variables* or *T-quantities* that can be associated with them.

Form the very beginning when discovering the PrT-nets Kurt Lautenbach and I were intrigued by the idea of operating upon the higher-level representation of a net system *symbolically* – in the fashion of symbolic logic or computer algebra.[1] I was always convinced that this should be simple and straightforward. In recent years, in order to find out how little, or much, was actually needed I started to conduct some experiments. To support them I extended some Standard ML code I had written earlier (on top of Design/CPN's interface) to extract the incidence matrix of an executable net model.

PrT-nets introduced the notion of an *individual* – *value* and *variable* – into the theory of Petri nets. Hence certain basic transformations of the expressions annotating a PrT-net come in quite naturally; in particular, consistent substitution of individual variables. (For a complete set of *equivalence transformations* for PrT-nets, see [2]. Only very few of those, however, play a role in the sequel.). To do the S-invariant analysis symbolically I took a step further. I identified *S-quantities* that are connected to the markings of a Petri net and *T-quantities* that are connected to the changes and started to study their properties and relationships in a playful, computer-algebraic fashion.

Over the time this SML code grew into a kind of package whose kernel consists of two basic elements:

- the application of a combination of transformation rules to a combination of expressions and,
- the reduction of the result by simplifications some of which are built-in and others can be declared ad-hoc.

The package might be a software engineer's nightmare but it allowed me to perform some small yet non-trivial experiments – about the results of which and some conclusions I want to talk in this note. The style of the note remains narrative rather than technical. I use a simple CPN model of a production schema as a running example and demonstrate what I can find out using my 'analysis package'.

# 2 The Example

The running toy example is shown in figure 1. It presents a simple production schema consisting of several nested loops; it is denoted by $\Pi$ in the sequel. It has been extracted from a more complex net system in the field of chemical engineering – a recipe for batch process control [3] – but it could as well have been derived from the transaction processing in a database system or the processing of work objects in a workflow system or, just some piece of software.

The example has been built and run by means of Design/CPN [6] and hence it follows its syntactical conventions. The main declarations of its datatypes (colour sets) and other non-logical symbols are shown in table 1. In the symbolic analysis of Petri nets that we demonstrate in this note they play a minor role only.

For the time being treat the shaded transitions as auxiliary (comment); they indicate how the system may interact with other systems or its environment. Also assume that the pairs of places $s_1$ and $s_1'$ respectively $s_3$ and $s_3'$ are fused together. Then table 2 presents the net system $\Pi$ in an equivalent form, namely by its *incidence matrix* $C$. It has a row $C_s$ for each place $s$ and a column $C^t$ for each transition $t$. Each entry $C_s^t$ is the combination of arc expressions between $s$ and $t$, inputs to $t$ distinguished from outputs by a negative sign.

The incidence matrix is the basis for studying the net system $\Pi$ in merely symbolical terms. Table 3 shows what the package extracts from the CPN model in simulation state – when, in general, all submodels are linked properly and it is about to be executed.

# 3 Combinations of Pieces

In physical systems, quantities like displacement, force, energy, charge, temperature, etc. are real-valued, continuous functions of time with sufficiently high derivatives. For net systems we do not expect that much structure. However, a minimum of numbership should be found with the quantities in net systems, too. That is, we call some observable that we associate with a net

system, a *quantity* only if we can *add* its values to each other and *multiply* them by *scalars*.[1] These scalars may be integer, rational or real numbers; they form a ring $R$.

Structures whose elements can be added to each other and multiplied by numbers are called *modules*. Every marking of a place, for example, is an element of a particular module; it is an integer combination of structured tokens, elements of the datatype (colour set) associated with that place.

For the rest of this paper all quantities have values that are combinations of some *pieces*. These pieces can be viewed as different units for expressing a value of the quantity – like the various coins and notes of a currency. Mere numbers themselves as values then are combinations of the unit ().

- Let $D$ be a set respectively a datatype and $R$ a ring of scalar coefficients respectively the datatype *int* or *real*.[2] A combination of pieces from $D$ with coefficients from $R$ is a mapping $\kappa : D \to R$ such that for finitely many pieces $d$ only the coefficient $\kappa(d)$ is different from zero.
- $\mathcal{L}(D, R)$ denotes the set of combinations of pieces from $D$ with coefficients from $R$.

I use the following notation:

- Listing of pieces:

$$3{\cdot}p - 2{\cdot}q + r := \{d \mapsto \begin{cases} 3 : d = p \\ -2 : d = q \\ 1 : d = r \\ 0 : otherwise \end{cases} \mid d \in D\}$$

- Null:

$$\mathbf{0} := \{d \mapsto 0 \mid d \in D\}$$

- Boolean coefficients:

$$[\top]{\cdot}d := d \qquad [\bot]{\cdot}d := \mathbf{0} \qquad [b_1, \ldots, b_k]{\cdot}d := [b_1]{\cdot}\ldots{\cdot}[b_k]{\cdot}d$$

- Combination of combinations:

$$(\kappa_1 + \kappa_2)(d) := \kappa_1(d) + \kappa_2(d) \qquad \text{(addition)}$$
$$(k \cdot \kappa)(d) := k \cdot \kappa(d) \qquad \text{(multiplication with scalars)}$$

Let $D$ and $D'$ be two sets of pieces and $R$ a set of scalars. A *linear* mapping $m$ from $\mathcal{L}(D, R)$ into $\mathcal{L}(D', R)$ I call a *distribution*; it can be represented as a function $\delta : D \to \mathcal{L}(D', R)$ that 'distributes' every piece of $D$ over a combination of pieces of $D'$. Because of the linearity of $m$,

- $m(\kappa) = \sum_{d \in D} \kappa(d) \cdot \delta(d)$ (note that $\kappa(d) \neq 0$ for finitely many $d$ only)

[1] An observable is an operator that assigns to every state or every process of the net system exactly one value. A proposition is not a quantity in our sense since its truth values cannot be added or multiplied by numbers.

[2] Actually, neither the datatype *int* of integers nor the datatype *real* of floating point numbers form a ring.

- *Example*: Let $m$ be defined by $\delta = \{(x, y, c) \mapsto c{\cdot}x - y\}$. Then

  $m(2{\cdot}(A, B, 3){+}3{\cdot}(A, C, -1)) = 2{\cdot}(3{\cdot}A{-}B){+}3{\cdot}(-A{-}C) = 3{\cdot}A{-}2{\cdot}B{-}3{\cdot}C$

A higher-level Petri net is full of expressions for combinations of pieces. The markings of a place $s$ and the arc inscriptions of the adjacent arcs are expressions all denoting combinations of pieces of the *colour set* of $s$. The *guard* of a transition $t$ is not a combination by itself but is a boolean coefficient that is common to all inscriptions of the arcs adjacent to $t$.

What about combinations of expressions. If we just view them as pieces of datatype *string*, we may combine them arbitrarily. However, in order for the addition of two expressions $u$ and $w$ to make sense, two conditions must be satisfied.

- All values of $u$ and $w$ must belong to the same $\mathcal{L}(D, R)$.
- $u$ and $w$ must have the same *scope* of consistent substitution of individual variables.

The first condition is satisfied for any two expressions in the reach of the same place; the second one is satified for any two expressions in the reach of the same transition. In general, however, two arc expressions at the same place or at the same transition do not satisfy both conditions. Before they may be combined, they must be transformed – at a place by substitutions that get rid of all individual variables (*bindings* in CPN terminology), at a transition by distributions that have the same range.

Let $u$ be (an expression for) a combination of pieces and $\tau$ a transformation rule (substitution or distribution). The application of $\tau$ to $u$ is denoted by $u{\odot}\tau$. The two conditions above for combining expressions guarantee that

$$(u + w){\odot}\tau = u{\odot}\tau + w{\odot}\tau$$

What about combining two transformation rules $\sigma$ and $\tau$ such that for every expression $u$

$$u{\odot}(\sigma + \tau) = u{\odot}\sigma + u{\odot}\tau$$

Here the conditions are

- Either both $\sigma$ and $\tau$ are distributions of the same type $D \to \mathcal{L}(D', R)$ where $D$ is the colour set of the place to which $u$ belongs
- or, both $\sigma$ and $\tau$ are substitutions of the individual variables (CPN variables) of the transition to which $u$ belongs.

# 4 Changes

To get acquainted with the example, let us now consider what happens with one of the tokens of the initial marking of place $s_0$. It is the batch $(X, 3)$, *'produce 3 units of substance* X*'*, that on place $s_0$ appears as an incoming order.

The processing of the batch begins with an occurrence of transition $t_1$ for $\{c, r, x \leftarrow\!\shortmid 3, [D1, D3], X\}$ which takes the batch $(X, 3)$ from the *incoming*

*orders*, $s_0$, puts a copy of the recipe for $X$, $r = R(x) = [D1, D3]$, on place $s_6$, removes the multi-set of devices listed in $r$, (dvc $r$) $= 1{\cdot}D1 + 1{\cdot}D3$, from the *pool of available devices*, $s_4$, puts an empty container for substance $X$ on *outgoing results*, $s_5$, and puts the status vector for the batch on place $s_1$ where the production cycle begins.

All this happens, at the chosen level of abstraction, as a single indivisible *event*. However, it may occur *only if* the binding for $c$ and $r$ satisfies the list of constraints stated in the *guard* of $t_1$, $[\ c > 0, r = Rx, r \neq [] \ ]$, that is if the count $c$ and the recipe $r$ are meaningful and non-trivial.

The result of applying $\{c, r, x \leftarrow\!\shortmid 3, [D1, D3], X\}$ to $t_1$ is shown in table 4(a). Note that in our symbolic approach no evaluation takes place. Rather, individual variables are replaced by terms and the resulting expressions may be transformed by some simple equivalence transformations as reduction rules.

Through transition $t_1$ the kernel of the processing of the batch has been entered. It consists of two loops. The inner one, determined by transitions $t_2$ and $t_3$, follows the recipe step by step until one unit of substance $x$ is done. A single production step is identified, for the sake of simplicity, with utilizing a particular device (resource). In the outer loop, when one unit is done, $t_4$ puts it on place $s_3$ for being accumulated at $s_5$. And, as long as more units have to be produced yet, $t_4$ also restores the recipe and decreases the count in the status vector, and enters another round.

Transition $t_5$ accumulates the single units of a substance in the corresponding container that was put on place $s_5$ by transition $t_1$. If the whole batch is done, $t_5$ also returns the set of devices that were reserved for the batch in the beginning and removes the copy of the recipe used for substance $X$ as done – thus completing the processing of a single batch.

Let us now summarize the whole processing of batch $(X, 3)$ by adding up, for each transition, all the bindings for which this transition occurs. In doing so we deliberately disregard any order in which the transitions occur for the listed bindings, and whether the transition occurrences are enabled or not. The result is shown in table 4(b). It is a *T-vector* (vector whose index set is the set $T$ of transitions) whose entries are combinations of bindings. We call such a T-vector a *change* in the sequel.

The rows $C_s$ of the incidence matrix $C$ (see table 2) are T-vectors, too, whose entries are combinations of arc expressions. If we apply, for each transition $t$, the combinations of bindings of the change in table 4(b) to the corresponding arc expressions and then add-up all transformed columns, we get the net effect that the change has on the markings of each place. Table 5 shows what the package delivers – before and after some built-in simplifications for list expression are applied.

Formally, the *effect* of a change $P$ is the *matrix product* $C{\cdot}P$ between the incidence matrix $C$ and the one-column matrix (T-vector) $P$. The result is exactly what we expect. The batch $(X, 3)$ is transformed from an unprocessed order into a container with 3 units of substance $X$. All internal places are unaffected. Their markings are restored to the initial ones. On the inner part of the system – transitions $t_2, t_3, t_4$ – the change is *cyclic*.

# 5 S-Quantities

Let $\Sigma$ be a net system. A state variable is an operator that for some module $\mathcal{D} = \mathcal{L}(D, R)$, assigns to every conceivable marking of $\Sigma$ an element of $\mathcal{D}$. If a state variable $Q$ is linear it can be represented by an S-vector of distributions: for every place $s$ there is a distribution $Q_s : D_s \to \mathcal{D}$ that assign to every token of the colour set of $s$, $D_s$, the corresponding value of $Q$. Each $Q_s$ determines the form in which the variable $Q$ appears on place $s$. I call a such linear state variable $Q$ of a net system an *S-quantity.*

Table 6 shows three S-vectors of distributions. They denote different S-quantities of our production schema $\Pi$. The S-quantity *load*, for example, gives the work-load of $\Pi$ at an arbitrary marking in terms of the number of units of each substance. It shows how the work objects (batches) are spread over the places at the various stages of the production process, i.e. the various forms in which *load* appears in $\Pi$.

$$
\begin{array}{rcll}
 & & \Delta s_0 \odot \{(x,c) \mapsto c{\cdot}x\} & \text{waiting batches} \\
 & + & \Delta s_1 \odot \{(x,c,\_,\_) \mapsto c{\cdot}x\} & \text{batch between stations} \\
\Delta load & = & + \quad \Delta s_2 \odot \{(x,c,\_,\_,\_) \mapsto c{\cdot}x\} & \text{batch at a station} \\
 & + & \Delta s_3 \odot \{(x,\_,\_) \mapsto 1{\cdot}x\} & \text{piece done} \\
 & + & \Delta s_5 \odot \{(x,c',\_) \mapsto c'{\cdot}x\} & \text{pieces stored}
\end{array}
$$

Each entry of the vector specifies the combination of pieces that the tokens of the corresponding place contribute to the current work-load. For the initial marking shown in figure 2, the value of *load* is

$$
\begin{array}{rl}
 & (1{\cdot}(X,3) + 1{\cdot}(Y,2)) \odot \{(x,c) \mapsto c{\cdot}x\} \\
+ & \mathbf{0} \odot \{(x,c,\_,\_) \mapsto c{\cdot}x\} \\
+ & \mathbf{0} \odot \{(x,c,\_,\_,\_) \mapsto c{\cdot}x\} \\
+ & \mathbf{0} \odot \{(x,\_,\_) \mapsto 1{\cdot}x\} \\
+ & \mathit{AvlDvc}^0 \odot \{\_ \mapsto \mathbf{0}\} \\
+ & \mathbf{0} \odot \{(x,c) \mapsto c{\cdot}x\} \\
+ & \mathit{Recipes}^0 \odot \{\_ \mapsto \mathbf{0}\} \qquad = \quad 3{\cdot}X + 2{\cdot}Y
\end{array}
$$

Table 7 shows the result of applying every entry of the S-vector *load* to the entries of the corresponding row of the incidence matrix (table 2). The bottom row, grad (*gradient*), is the combination of the rows for places $s_0, \ldots, s_6$. It gives for each transition $t$ the effect that the corresponding elementary changes $\Delta t$ have to the S-quantity *load*. We call it the *defect* of *load* at transition $t$.

All entries of grad *load* are equal to zero except for transition $t_4$. The batch count $c$, however, remains greater than zero for all meaningful processes of the production schema (see below). And for $c > 0$, the defect of *load* at $t_4$ reduces to $\mathbf{0}$ as well. Hence grad *load* $= \mathbf{0}$ for all meaningful processes; the S-quantity *load* is constant.

By the same procedure the other two S-quantities of table 6, *recipe* and *count*, can be shown to be constant or, as the terminology is in net theory, are

*S-invariants.* The quantity *count* is an auxiliary S-quantity; that its defect is **0** at every transition proves that a batch count which is non-negative initially may never become negative.

The different forms of the quantity *recipe* show how a recipe is decomposed and re-composed during the production process. There is a twist, however: the coefficient of the combination on place $s_6$ is negative. Hence we can split *recipe* into two non-trivial quantities with non-negative coefficients, $recipe^{\geq 0}$ and $recipe^{\leq 0}$, such that

$$recipe = recipe^{\geq 0} - recipe^{\leq 0}$$

Since *recipe* is constant (see table 8) and its value for the initial marking is **0**, we get

$$recipe^{\geq 0} = recipe^{\leq 0}$$

The quantity $recipe^{\leq 0}$ is a *copy* of $recipe^{\geq 0}$: every piece $(x, r)$ on place $s_6$ is a copy of the recipe of a batch being processed.

The *flow* of S-quantities in net systems – between subsystems determined by disjoint sets of places – has two aspects[5]:

- *flux* or *exchange* between subsystems: *load*, for example, is exchanged between the outer part and the inner part of the system through transitions $t_1$ and $t_4$;
- *influence* between subsystems: $recipe^{\geq 0}$ and $recipe^{\leq 0}$ are copies maintained by transitions $t_1$ and $t_5$.

# 6 T-Quantities

Not all interesting quantities that we associate with a net system may be functions of markings. There are also quantities that are functions of changes. For example, the production costs for a batch in a production schema, the total time during which a resource is idle respectively occupied, or the *work* performed by a component or subsystem during a particular process – all those quantities cannot be expressed, in general, as a function of markings but rather as functions of changes.

A linear operator that for some module $\mathcal{D} = \mathcal{L}(D, R)$ assigns an element of $\mathcal{D}$ to every change in a net system $\Sigma$ is called a *T-quantity* of $\Sigma$.

Assume we want to express the production costs for batches of our production schema $\Pi$, in terms of entire units of some currency. We assign to transition $t_2$ some costs involved using device $d$ during the production of a single unit of substance $x$, and to transition $t_5$ some costs for the transport and storage of a single unit of $x$. Other costs are ignored. The corresponding cost functions are $K_2(x, d)$ and $K_5(x)$, respectively. The T-quantity *cost* is then represented by a T-vector as shown in table 9.

The entries of *cost* for each transition $t_i$ are expressions for integer combinations of the unit () whose free variables belong to the scope of $t_i$. For every binding of $t_i$ they denote an integer value. For $K_2$ and $K_5$ as given as in table 9, the costs for processing batch $(X, 3)$, for example, are

$$3{\cdot}(K_2(X, D1){\cdot}() + K_2(X, D3){\cdot}() + K_5(X){\cdot}()) = 27{\cdot}()$$

Also the T-vector grad *load* of table 9 denotes a T-quantity, namely the effect that any change has on *load*. In general:

The T-vector of defects (the gradient) grad $q$ of an S-quantity $q$ is a T-quantity.

The value of a T-quantity, $k$, for some change $u$ is independent of the order in which the $\Delta t$ of $u$ may occur in an actual process. It is just the dot product between $k$ and $u$.

Now assume there are two different changes, $u_1$ and $u_2$, both leading from the same marking $M^1$ to the same marking $M^2$. Then we may ask whether the values of $k$ along $u_1$ and $u_2$ are the same or not. If it is the case that for any pair of markings $(M^1, M^2)$ the value of $k$ is the same along all changes that connect $M^1$ to $M^2$, we call $k$ *path-indifferent* ('path' in the reachabilty graph). In physical systems, work is a 'path-indifferent' quantity iff the forces are conservative – no friction, for example.

A T-quantity $k$ is path-indifferent iff its value along any cyclic change is **0**. And it is a linear-algebraical fact that

**Theorem:** For a T-quantity $k$ there is an S-quantity $\widehat{k}$ such that $k = \text{grad}\,\widehat{k}$ – a *potential function* for $k$ – iff $k$ is path-indifferent.

The costs for the processing of any set of batches as given by the T-quantity *cost* are independent of the various ways in which these processes may be intertwined. And since there are no cyclic processes in the production schema $\Pi$ as long it is not completed by an environment, *cost* is path-indifferent. A potential function *COST* is shown in table 10. The proof that

$$cost = -\text{grad}\ COST$$

eventually convinced me of the usefulness of that little symbolic analysis package; I could not have done without it.

# 7 An Interpretation

Talking about the *gradient* of an S-quantity and the *potential function* of a T-quantity suggests that I see a certain similarity with the dynamic description of physical systems. Here is the essence of what I have in mind. It seems to me that the interpretation that I give below to the incidence matrix may help to understand why net systems are so successful as discrete presentations of dynamic systems. However, for the time being I want to stress that by using some notation of calculus I do not claim more than similarity.

**S-Quantities:**

The state coordinates of a net system are its places. The elementary (the smallest yet not infinitesimal) changes are the transitions. The incidence matrix is the matrix of partial derivatives of the coordinates with respect to the changes:

$$\frac{\partial s}{\partial t} := C_s^t$$

A binding of the individual variables of a transition $t$ gives an event at $t$. I denote the binding as $\Delta t$. Then I get the effect of that event on a place $s$ as

$$\Delta s \;=\; C^t_s \odot \Delta t \;=:\; \frac{\partial s}{\partial t}\Delta t$$

The components $(Q_1, \ldots, Q_n)$ of the S-vector representing an S-quantity $Q$ are the partial derivatives of $Q$ with respect to the coordinates $s_i$:

$$Q_i =: \frac{\partial Q}{\partial s_i}, \quad Q_i(C^t_i) = \frac{\partial s_i}{\partial t} \odot \frac{\partial Q}{\partial s_i} \;=:\; \frac{\partial Q}{\partial s_i}\frac{\partial s_i}{\partial t}, \quad \frac{\partial Q}{\partial t} \;:=\; \sum_{i=1}^{n} \frac{\partial Q}{\partial s_i}\frac{\partial s_i}{\partial t}$$

For the effect of an event $\Delta t$ on the S-quantity $Q$ we now get

$$\begin{aligned} \Delta Q \;&=\; \sum_{i=1}^{n} Q_i(C^t_i \odot \Delta t) \;=\; \sum_{i=1}^{n} (C^t_i \odot \Delta t) \odot Q_i \;\stackrel{!}{=}\; \sum_{i=1}^{n} (C^t_i \odot Q_i) \odot \Delta t \\ &=\; \sum_{i=1}^{n} (\frac{\partial s_i}{\partial t} \odot \frac{\partial Q}{\partial s_i})\Delta t \;=\; (\sum_{i=1}^{n} \frac{\partial Q}{\partial s_i}\frac{\partial s_i}{\partial t})\Delta t \;=\; \frac{\partial Q}{\partial t}\Delta t \end{aligned}$$

Note that the essential step in this derivation depends on the assumption that the distributions $Q_i$ and the substitution $\Delta t$ – as transformations of expressions – commute. This is always the case as long as the distributions do not introduce additional free individual variables (which they cannot do if they denote proper functions).

Hence the heart of the S-invariance technique, the dot product between the column $C^t$ of the incidence matrix and the S-vector of components $(Q_1, \ldots, Q_n)$ of an S-quantity $Q$ yields the partial derivative of $Q$ with respect to the transition $t$.

$$\frac{\partial Q}{\partial t} \;=\; \sum_{i=1}^{n} \frac{\partial Q}{\partial s_i}\frac{\partial s_i}{\partial t} \;=\; (C^t)^{\intercal} \cdot Q$$

**Result (S-Quantities):**

- The matrix product of the transposed incidence matrix $C^{\intercal}$ with the S-quantity $Q \cong (Q_1, \ldots, Q_n)$ yields the *gradient* of $Q$.

$$\operatorname{grad} Q \;=\; (\frac{\partial Q}{\partial t_1}, \ldots, \frac{\partial Q}{\partial t_m}) \;=\; C^{\intercal} \cdot Q$$

- The components $(Q_1, \ldots, Q_n)$ are the different *forms* in which $Q$ appears on the places $s_i$ (like energy appears in different forms in physical systems).

$$\Delta Q \;=\; \frac{\partial Q}{\partial s_1}\Delta s_1 + \ldots + \frac{\partial Q}{\partial s_n}\Delta s_n$$

**T-Quantities, Fields:**

Quantities like *work* or *cost* assign a value to every process rather than every state. In a net system a process $P$ is a partially ordered set of event

occurrences. Let $\{\Delta t_i\}$ denote an enumeration of the occurrences of $P$ that is consistent with the ordering (a 'firing sequence').

$$\Delta t_i < \Delta t_k \Rightarrow i < k$$

Let $\mathcal{F} = (\ldots, \mathcal{F}_t, \ldots)$ be a T-vector of expressions denoting values of the same domain $\mathcal{D} = \mathcal{L}(D, R)$. I call it a *field*; it represents a T-quantity $F$ that assigns to every process $P \cong \{\Delta t_i\}$ a value by virtue of the *'line integral'* of $\mathcal{F}$ along $P$.

$$F(P) = \sum_i \mathcal{F}_t \odot \Delta t_i =: \int_P \mathcal{F}\, dt$$

The gradient of an S-quantity $Q$ is such a field. Its line integral along a process $P$ is the accumulated effect that $P$ has on $Q$. If, conversely, for a field $\mathcal{F}$ there is an S-quantity $U$ such that

$$\mathcal{F} = -\operatorname{grad} U,$$

$U$ is called a *potential function* of $\mathcal{F}$. If it exists it is unique up to an additive constant.

**Result:**

For net systems as well as for physical systems the following holds:

**Theorem**: For a field $\mathcal{F}$ there exists a potential function $U$ iff for any two processes $P_1$ and $P_2$ connecting the same states the line integral of $\mathcal{F}$ has the same value,

$$M_1 \underset{\overrightarrow{P_2}}{\overset{\overrightarrow{P_1}}{\rightrightarrows}} M_2 \;\Rightarrow\; \int_{P_1} \mathcal{F}\, dt = \int_{P_2} \mathcal{F}\, dt$$

or, equivalently, iff for any *cyclic* process (T-invariant, reproduction component) $R$ the value of the line integral is zero.

$$C \cdot R = 0 \quad \Rightarrow \quad \oint_R \mathcal{F}\, dt = 0$$

# 8 Future Work, Conclusions

The package so far is a nice gadget. It helped me to test some ideas on symbolic manipulation of higher-level nets. It doesn't need much theoretical foundation but uses simple transformations and simplifications of expressions in a straightforward fashion.

I have a lot of ideas how to improve and extend it. For example, it already provides the basic ingredients to do symbolic reachability analysis. Only one more technical device is needed, namely *individual parameters* to allow transition occurrences with partial bindings. They consist of individual variables with a suffix of the form $\hat{}i$ for the $i^{th}$ partial occurrence. Their scope is global.

For example, assume that $t_1$ is to occur for substance $X$ and substance $Y$ but count and recipe are left unspecified. Let the current value of a unique

counter for partial occurrences be 5. The resulting *parametrized marking* difference is shown in table 4(c).

Note that in order to add the marking difference to some given, ordinary or parametrized marking, one has to distribute the guard as a common boolean coefficient to all arc expressions of $t_1$.

Before parametrized markings and other features may be added, the package has to be thoroughly and professionally re-designed and re-implemented. Its tolerance for extensions has reached its limits.

For a re-implementation there are mainly two options. Either, continue using *Standard ML* or, switch to a computer algebra system like *Mathematica*. The SML option makes it part of the Design/CPN tool but requires implementing of a good deal of features that any good computer algebra package already has. The Mathematica option makes it an independent tool that requires a good link to Design/CPN.

I don't know yet. If you have an opinion, let me know of it.

# References


[1] Genrich, H.J.; Lautenbach, K.: *System modelling with high-level Petri nets.* Theoretical Computer Science 13 (1981) 109–136

[2] Genrich, H.J.: *Equivalence Transformations of PrT-Nets.* Advances in Petri Nets 1989 (G. Rozenberg, Ed.), LNCS 424. Berlin : Springer (1989) 179–208

[3] Genrich, H.J.; Hanisch, H.-M.: *Modelling and Analysis of Recipes.* Workshop on Analysis and Design of Event-Driven Operations in Process Systems, Imperial College, April 1995.

[4] Jensen, K.: *Coloured Petri Nets and the Invariant Method.* Theoretical Computer Science 14 (1981) 317–336

[5] Petri, C.A.: *Grundsätzliches zur Beschreibung diskreter Prozesse.* 3. Colloquium über Automatentheorie (W. Händler, E Peschl, H. Unger, Hrsg.).Basel, Stuttgart : Birkhäuser (1967) 121–140

[6] Design/CPN reference manual. Version 3.00. Aarhus : DAIMI, Aarhus University (1996)

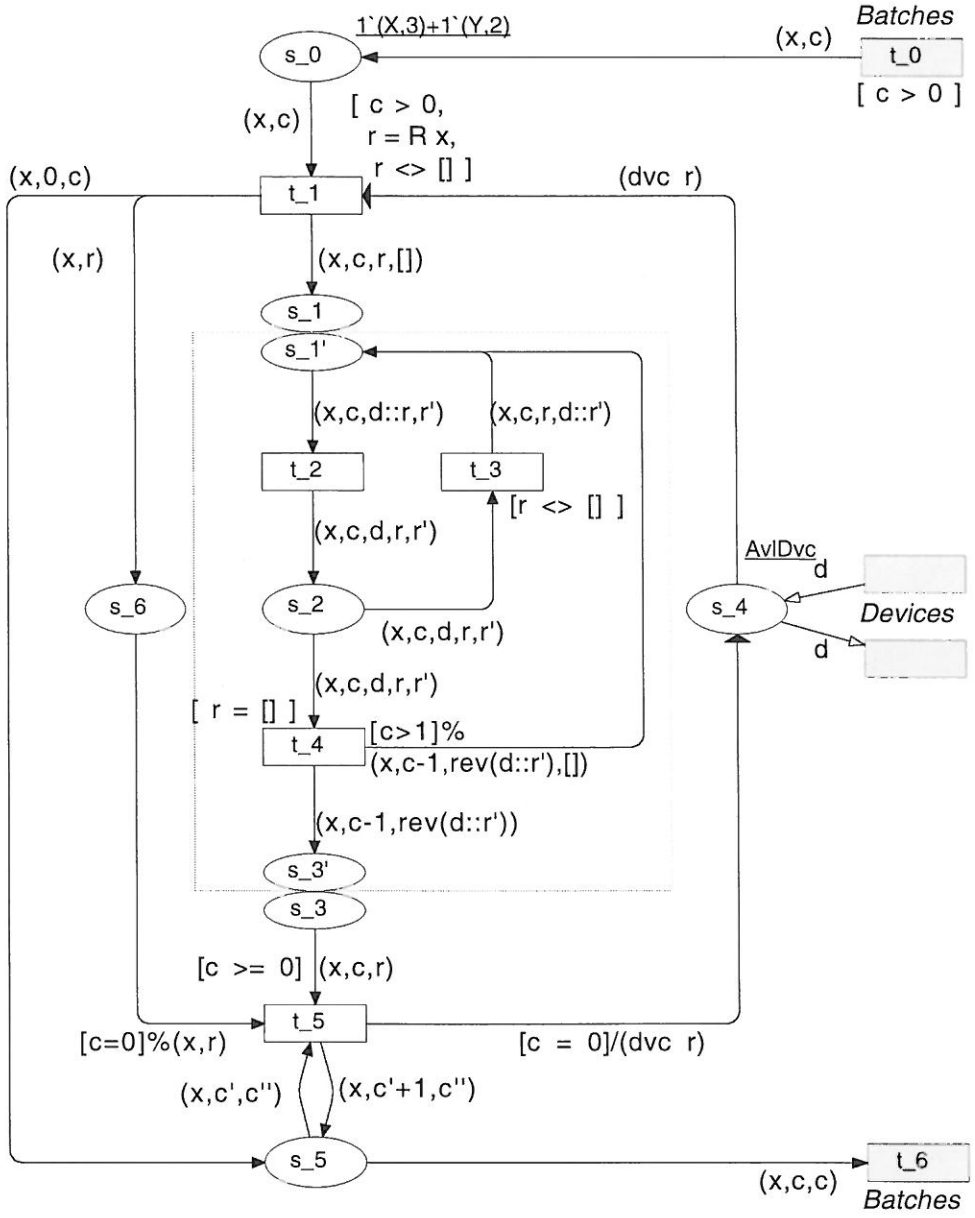



Figure 1: Net System – Production Schema

```
color Sbs = union X + Y + Z;                    var x,x' : Sbs;
color Cnt = int;                                var c,c',c'' : Cnt;
color Bch   = product Sbs * Cnt;

color Dvc = union D1 + D2 + D3 declare ms;      var d,d' : Dvc;
color Stp = Dvc;
color Rcp = list Stp;                           var r,r' : Rcp;

fun dvc r = (list_to_ms o remdupl) r;

val p_Rec = ref (fn X => [D1,D3] | Y => [D1,D2,D3,D1] | _ => []);
fun R d = !p_Rec d;

val AvlDvc  = 2 * Dvc;
```

Table 1: Global Declarations for the Production Schema

| | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ |
|---|---|---|---|---|---|
| $\Gamma$ | $[\, c > 0,$ $r = R\, x$ $r \neq [\,] \,]$ | | $[\, r \neq [\,] \,]$ | $[\, r = [\,] \,]$ | $[\, c \geq 0 \,]$ |
| $s_0$ | $-(x, c)$ | | | | |
| $s_1$ | $(x, c, r, [\,])$ | $-(x, c, d::r, r')$ | $(x, c, r, d::r')$ | $[c{>}1]\cdot$ $(x, c{-}1, rev(d::r'), [\,])$ | |
| $s_2$ | | $(x, c, d, r, r')$ | $-(x, c, d, r, r')$ | $-(x, c, d, r, r')$ | |
| $s_3$ | | | | $(x, c{-}1, rev(d::r'))$ | $-(x, c, r)$ |
| $s_4$ | $-(dvc\ r)$ | | | | $[c{=}0]\cdot(dvc\ r)$ |
| $s_5$ | $(x, 0, c)$ | | | | $-(x, c', c'')$ $+\ (x, c'{+}1, c'')$ |
| $s_6$ | $(x, r)$ | | | | $-[c{=}0]\cdot(x, r)$ |

$\Gamma$ = *Guard*: boolean coefficient common to all entries of the respective column

Table 2: Incidence Matrix of the Production Schema with *Guards*

```
( show_Mat_S (!IncMatrix);
  show_FusionSets ();
  show_FusionReps () );
```

```
t_1: c,r,x
Guard: [c>0,r=R x,r<>[]]
Pre:
s_4:      (dvc r)
s_0:      (x,c)
Post:
s_1:      (x,c,r,[])
s_5:      (x,0,c)
s_6:      (x,r)
-----
t_2: c,d,r,r',x
Guard:
Pre:
s_1':     (x,c,d::r,r')
Post:
s_2:      (x,c,d,r,r')
-----
t_3: c,d,r,r',x
Guard: [r<>[]]
Pre:
s_2:      (x,c,d,r,r')
Post:
s_1':     (x,c,r,d::r')
-----
t_4: c,d,r,r',x
Guard: [r=[]]
Pre:
s_2:            (x,c,d,r,r')
Post:
s_3':           (x,c-1,rev(d::r'))
s_1':     [c>1]‘(x,c-1,rev(d::r'),[])
-----
t_5: c,c',c'',r,x
Guard: [c>=0]
Pre:
s_3:            (x,c,r)
s_6:      [c=0]‘(x,r)
s_5:            (x,c',c'')
Post:
s_5:            (x,c'+1,c'')
s_4:      [c=0]‘(dvc r)
=====
FG_S1: s_1,s_1'
FG_S3: s_3,s_3'
=====
FG_S1: s_1
FG_S3: s_3
=====
```

Table 3: Incidence Matrix Extracted by the Analysis Package

| | $t_1$ |
|---|---|
| $\Gamma$ | [ 3>0,<br>$[D1, D3]{=}R\ X$,<br>$[D1, D3]{\neq}[]$ ] |
| $s_0$ | $-\ (X, 3)$ |
| $s_1$ | $(X, 3, [D1, D3], [])$ |
| $s_2$ | |
| $s_3$ | |
| $s_4$ | $-$ (dvc $[D1, D3]$) |
| $s_5$ | $(X, 0, 3)$ |
| $s_6$ | $(X, [D1, D3])$ |

$t_1 \odot \{c, r, x \leftarrow 3, [D1, D3], X\}$

(a)

| | |
|---|---|
| $t_1$ | $1{\cdot}\{x, c, r \leftarrow X, 3, [D1, D3]\}$ |
| $t_2$ | $1{\cdot}\{x, c, d, r, r' \leftarrow X, 3, D1, [D3], []\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 3, D3, [], [D1]\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 2, D1, [D3], []\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 2, D3, [], [D1]\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 1, D1, [D3], []\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 1, D3, [], [D1]\}$ |
| $t_3$ | $1{\cdot}\{x, c, d, r, r' \leftarrow X, 3, D1, [D3], []\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 2, D1, [D3], []\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 1, D1, [D3], []\}$ |
| $t_4$ | $1{\cdot}\{x, c, d, r, r' \leftarrow X, 3, D3, [], [D1]\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 2, D3, [], [D1]\}$ |
| | $+\ 1{\cdot}\{x, c, d, r, r' \leftarrow X, 1, D3, [], [D1]\}$ |
| $t_5$ | $1{\cdot}\{x, c, c', c'', r \leftarrow X, 2, 0, 3, [D1, D3]\}$ |
| | $+\ 1{\cdot}\{x, c, c', c'', r \leftarrow X, 1, 1, 3, [D1, D3]\}$ |
| | $+\ 1{\cdot}\{x, c, c', c'', r \leftarrow X, 0, 2, 3, [D1, D3]\}$ |

Summary of processing of batch $(X, 3)$

(b)

| | |
|---|---|
| $s_0$ | $-[\ c\hat{}5{>}0, r\hat{}5{=}R\ X, r\hat{}5{\neq}[]\ ]{\cdot}(X, c\hat{}5) \ + \ -[\ c\hat{}6{>}0, r\hat{}6{=}R\ Y, r\hat{}6{\neq}[]\ ]{\cdot}(Y, c\hat{}6)$ |
| $s_1$ | $[\ c\hat{}5{>}0, r\hat{}5{=}R\ X, r\hat{}5{\neq}[]\ ]{\cdot}(X, c\hat{}5, r\hat{}5, [])+[\ c\hat{}6{>}0, r\hat{}6{=}R\ Y, r\hat{}6{\neq}[]\ ]{\cdot}(Y, c\hat{}6, r\hat{}6, [])$ |
| $s_2$ | |
| $s_3$ | |
| $s_4$ | $-[\ c\hat{}5{>}0, r\hat{}5{=}R\ X, r\hat{}5{\neq}[]\ ]{\cdot}(\text{dvc}\ r\hat{}5) \ + \ -[\ c\hat{}6{>}0, r\hat{}6{=}R\ Y, r\hat{}6{\neq}[]\ ]{\cdot}(\text{dvc}\ r\hat{}6)$ |
| $s_5$ | $[\ c\hat{}5{>}0, r\hat{}5{=}R\ X, r\hat{}5{\neq}[]\ ]{\cdot}(X, 0, c\hat{}5) \ + \ [\ c\hat{}6{>}0, r\hat{}6{=}R\ Y, r\hat{}6{\neq}[]\ ]{\cdot}(Y, 0, c\hat{}6)$ |
| $s_6$ | $[\ c\hat{}5{>}0, r\hat{}5{=}R\ X, r\hat{}5{\neq}[]\ ]{\cdot}(X, r\hat{}5) \ + \ [\ c\hat{}6{>}0, r\hat{}6{=}R\ Y, r\hat{}6{\neq}[]\ ]{\cdot}(Y, r\hat{}6)$ |

Partial bindings and parameters: $\Delta M = t_1 \hat{\odot}^{5}(\{x \leftarrow X\} + \{x \leftarrow Y\})$

(c)

Table 4: Changes as Combinations of Bindings

```
( val change_X_3 =
  [
  (t_1, [(1,"(x,c,r)" <-+ "(X,3,[D1,D3])") ]),
  (t_2, [(1,"(x,c,d,r,r')" <-+ "(X,3,D1,[D3],[])"),...  ]),
   ...
  (t_5, [ ...,(1,"(x,c,c',c'',r)" <-+ "(X,0,2,3,[D1,D3])") ]) ];
  EFFECT change_X_3; );
```

```
s_0:  -       (X,3)
s_1:          (X,3,[D1,D3],[])
      -       (X,1,D1::[D3],[])
      -       (X,1,D3::[],[D1])
      +       (X,1,[D3],D1::[])
      -       (X,2,D1::[D3],[])
      -       (X,2,D3::[],[D1])
      +       (X,2,[D3],D1::[])
      -       (X,3,D1::[D3],[])
      -       (X,3,D3::[],[D1])
      +       (X,3,[D3],D1::[])
      + [1>1]'(X,1-1,rev(D3::[D1]),[])
      + [2>1]'(X,2-1,rev(D3::[D1]),[])
      + [3>1]'(X,3-1,rev(D3::[D1]),[])
s_3:  -       (X,0,[D1,D3])
      -       (X,1,[D1,D3])
      -       (X,2,[D1,D3])
      +       (X,1-1,rev(D3::[D1]))
      +       (X,2-1,rev(D3::[D1]))
      +       (X,3-1,rev(D3::[D1]))
s_4:  -       (dvc[D1,D3])
      + [0=0]'(dvc[D1,D3])
      + [1=0]'(dvc[D1,D3])
      + [2=0]'(dvc[D1,D3])
s_5:  -       (X,1,3)
      -       (X,2,3)
      +       (X,0+1,3)
      +       (X,1+1,3)
      +       (X,2+1,3)
s_6:          (X,[D1,D3])
      - [0=0]'(X,[D1,D3])
      - [1=0]'(X,[D1,D3])
      - [2=0]'(X,[D1,D3])
=====
```

```
( setRed_coeff red_bool;
  setRed_piece (map_tuple [(2,red_int),(3,red_list),(4,red_list)]);
  REDUCE_IT () );
```

```
s_0:  - (X,3)
s_5:    (X,3,3)
=====
```

Table 5: Effect of a Change Before and After Simplification

| | *load* | *recipe* | *count* |
|---|---|---|---|
| $s_0$ | $(x, c) \mapsto c{\cdot}x$ | $\_ \mapsto \mathbf{0}$ | $(\_, c) \mapsto [c{>}0]{\cdot}c{\cdot}()$ |
| $s_1$ | $(x, c, \_, \_) \mapsto c{\cdot}x$ | $(x, \_, r, r') \mapsto (x, (rev\ r')\hat{}\hat{}r)$ | $(\_, c, \_, \_) \mapsto [c{>}0]{\cdot}c{\cdot}()$ |
| $s_2$ | $(x, c, \_, \_, \_) \mapsto c{\cdot}x$ | $(x, \_, d, r, r') \mapsto (x, (rev\ d{::}r')\hat{}\hat{}r)$ | $(\_, c, \_, \_, \_) \mapsto [c{>}0]{\cdot}c{\cdot}()$ |
| $s_3$ | $(x, \_, \_) \mapsto 1{\cdot}x$ | $(x, c, r) \mapsto [c{=}0]{\cdot}(x, r)$ | $(\_, c, \_) \mapsto [c{\geq}0]{\cdot}1{\cdot}()$ |
| $s_4$ | $\_ \mapsto \mathbf{0}$ | $\_ \mapsto \mathbf{0}$ | $\_ \mapsto \mathbf{0}$ |
| $s_5$ | $(x, c) \mapsto c{\cdot}x$ | $\_ \mapsto \mathbf{0}$ | $(\_, c') \mapsto c'{\cdot}()$ |
| $s_6$ | $\_ \mapsto \mathbf{0}$ | $(x, r) \mapsto -(x, r)$ | $\_ \mapsto \mathbf{0}$ |

Table 6: Some S-Quantities of the Production Schema

| | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ |
|---|---|---|---|---|---|
| $\Gamma$ | $[\ c > 0,\ r \neq [\,]\ ]$ | | $[\ r \neq [\,]\ ]$ | $[\ r = [\,]\ ]$ | $[\ c \geq 0\ ]$ |
| $s_0$ | $-c{\cdot}x$ | | | | |
| $s_1$ | $c{\cdot}x$ | $-c{\cdot}x$ | $c{\cdot}x$ | $[c{>}1]{\cdot}(c{-}1){\cdot}x$ | |
| $s_2$ | | $c{\cdot}x$ | $-c{\cdot}x$ | $-c{\cdot}x$ | |
| $s_3$ | | | | $x$ | $-x$ |
| $s_4$ | | | | | |
| $s_5$ | $0{\cdot}x$ | | | | $-c'{\cdot}x + (c'{+}1){\cdot}x$ |
| $s_6$ | | | | | |
| grad *load* | $\mathbf{0}$ | $\mathbf{0}$ | $\mathbf{0}$ | $x - c{\cdot}x$ $-[c{>}1]{\cdot}x$ $+[c{>}1]{\cdot}c{\cdot}x$ | $\mathbf{0}$ |

Table 7: Incidence Matrix Transformed by S-Quantity *load*

```
DEFECT recipe;

t_1:  -            (x,r)
      +            (x,(rev[])^^r)
t_2:               (x,(rev d::r')^^r)
      -            (x,(rev r')^^(d::r))
t_3:  -            (x,(rev d::r')^^r)
      +            (x,(rev(d::r'))^^r)
t_4:    [(c-1)=0]`(x,(rev(d::r')))
      -            (x,(rev d::r')^^r)
      +     [c>1]`(x,(rev[])^^(rev(d::r')))
=====

( setRed_coeff red_bool;
  setRed_piece (map_tuple [(2,red_list)]);
  REDUCE_IT () );

t_4:    [c=1]`(x,(rev r')^^[d])
      + [c>1]`(x,(rev r')^^[d])
      -       (x,(rev r')^^[d]^^r)
=====

( Guards_to_IT [t_4];
  print_IT () );

t_4:    [r=[],c=1]`(x,(rev r')^^[d])
      + [r=[],c>1]`(x,(rev r')^^[d])
      -     [r=[]]`(x,(rev r')^^[d]^^r)
=====

let val sub_r = subst [("r","[]")]
    val rep_cf = replace [("[c>1]","1-[c=1]")] (* c>0!  *)
in
setRed_coeff (red_bool o rep_cf o sub_r);
setRed_piece ((map_tuple [(2,red_list)]) o sub_r);
REDUCE_IT ()
end;
=====
```

Table 8: Defect of *recipe* With Simplification

| | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ |
|---|---|---|---|---|---|
| | T-Vectors | | | | |
| type | $\{x, c, r\}$ | $\{x, c, d, r, r'\}$ | $\{x, c, d, r, r'\}$ | $\{x, c, d, r, r'\}$ | $\{x, c, c', c'', r\}$ |
| $\Gamma$ | $[\, c > 0,$ $r = R\, x$ $r \neq [] \,]$ | | $[\, r \neq [] \,]$ | $[\, r = [] \,]$ | $[\, c \geq 0 \,]$ |
| | T-Quantities | | | | |
| *cost* | **0** | $K_2(x, d)\cdot()$ | **0** | **0** | $K_5(x)\cdot()$ |
| grad *load* | **0** | **0** | **0** | $x - c\cdot x$ $-[c{>}1]\cdot x$ $+[c{>}1]\cdot c\cdot x$ | **0** |

$K_2 =$

| | $X$ | $Y$ | $Z$ |
|---|---|---|---|
| $D1$ | 4 | 3 | 0 |
| $D2$ | 2 | 2 | 0 |
| $D3$ | 3 | 5 | 0 |
| $D4$ | 1 | 1 | 0 |

, $K_5 =$

| $X$ | $Y$ | $Z$ |
|---|---|---|
| 2 | 1 | 0 |

Table 9: T-Quantities and Other T-Vectors

```
(
val COST = [
(s_0,[(1,    "(x,c)"     +-> "-            c*(H(x, R x) + K_5(x))'()")]),
(s_1,[(1, "(x,c,r,r')"  +-> "-     (H(x,r)+(c-1)*H(x,(rev r' )^^r))'
                                                              ()")]),
(s_2,[(1,"(x,c,d,r,r')" +-> "- (H(x,r)+(c-1)*H(x,(rev (d::r'))^^r))'
                                                              ()")]),
(s_3,[(1,   "(x,c,r)"    +-> "-                                0'()")]),
(s_5,[(1, "(x,c',c'')"  +-> "-                    (c''-c')*K_5(x)'()")])
];
defect COST;
Guards_to_IT [t_1,t_4];
setRed_coeff (map_arg "H" (map_tuple [(2,red_list)]));
REDUCE_IT ()
);
```

```
t_1:  -      [c>0,r<>[],r=R x]*H(x,r)*c'()
      +    [c>0,r<>[],r=R x]*H(x,R x)*c'()
t_2:  -                            H(x,r)'()
      +                       H(x,[d]^^r)'()
t_4:                       [r=[]]*(H(x,r))'()
      +   [c>1,r=[]]*H(x,(rev r')^^[d])'()
      -  [r=[]]*(H(x,(rev r')^^[d]^^r))'()
      +  [r=[]]*H(x,(rev r')^^[d]^^r)*c'()
      - [c>1,r=[]]*H(x,(rev r')^^[d])*c'()
t_5:                               K_5(x)'()
=====
```

```
( setRed_coeff_at [t_1] (subst [("r","R x")]);
  setRed_coeff_at [t_4] ((replace [("[c>1]","1-[c=1]")]) o
                                        subst [("r","[]")]));
  setRed_coeff ((map_arg "H" (map_tuple [(2,red_list)])) o
                                                    red_bool);
  REDUCE_IT () );
```

```
t_2:  -      H(x,r)'()
      + H(x,[d]^^r)'()
t_5:         K_5(x)'()
=====
```

```
( define ("H", "(x,[])     +-> 0 |
               (x,[d])     +-> K_2(x,d) |
               (x,r^^r') +-> H(x,r)+H(x,r')" );
  setRed_coeff (eval_defs ["H"]); REDUCE_IT () );
```

```
t_2:    K_2(x,d)'()
t_5:      K_5(x)'()
=====
```

Table 10: A Potential Function of *cost*

# DEPENDABILITY EVALUATION OF A SIMPLE MECHATRONIC SYSTEM USING COLOURED PETRI NETS

Gilles MONCELET[*,**], Søren CHRISTENSEN[***], Hamid DEMMOU[**], Mario PALUDETTO[**], José PORRAS[*]

**PSA Peugeot Citroën, 18 rue des Fauvelles, 92256 La Garenne Colombes cedex, France*
*Tel: + 33 (0)1 47 69 83 36*
***LAAS/CNRS, 7 avenue du colonel Roche, 31077 Toulouse cedex, France*
*Tel: + 33 (0)5 61 33 62 00, E-mail: moncelet@laas.fr*
****Computer Science Department, Aarhus University, Ny Munkegade 116, DK-8000 Aarhus C, Denmark*
*Tel: +45 89 42 32 65, E-mail: schristensen@daimi.aau.dk*

**Abstract:** Mechatronic automotive systems are hybrid systems. Modelling and simulation of the interactions between continuous and discrete parts is essential to evaluate dependability. In this paper we show how a simple mechatronic system can be modelled in the CPN formalism. Quantitative dependability evaluation is obtained thanks to Monte-Carlo simulation. We use the DesignCPN Occurrence Graph tool to validate the model and make a qualitative analysis of the system.

## 1. MOTIVATION

Mechatronic systems mix electric, mechanic, hydraulic and electronic technologies and use a computer control [GUY 94]. Some mechatronic systems like active suspension, automatic gear box, engine control, anti-skating system are already available on today's cars. The aim of the control system is to observe the operative part through physical variables measured by the sensors, and choose the suitable command processed by the actuators. Two kind of actions are possible : continuous or discrete actions. The continuous control process estimate the output error compared to a target value and calculate the new continuous action to reduce the error. A discrete control process detect some event (typically, a threshold overshoot) and choose a new discrete state for the system. A reconfiguration system is a discrete control system dedicated to react against faults of the system components. The architecture of a typical mechatronic system is given by Figure 1. In this article, we deal with discrete control processes only.

In the early design stage of a new mechatronic system, designers have to deal with dependability evaluation [LER 92, HEN 96]. From a functional model, the Preliminary Risk Analysis identifies the events that lead to a catastrophic event, also called « feared events ». The fault tree method is then used for a qualitative and quantitative dependability evaluation.

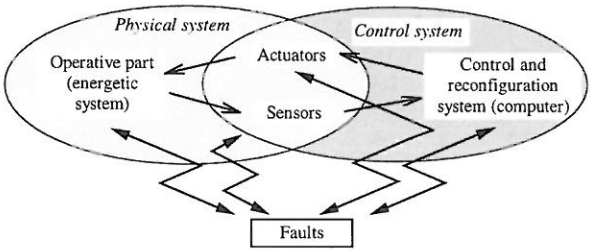



Figure 1: Architecture of mechatronic systems

A fault tree gives the Boolean conditions of occurrence of a feared event. These conditions are written in terms of elementary events, the faults of the basic components of the system [PAG 80]. Efficient algorithms and tools allow

today to compute the feared event occurrence probability given the elementary event failure rates. But, this representation is static and does not take into account reconfigurations.

An alternative to the fault trees is to model the structural and functional interactions between the components of the system in the State Graph formalism [PAG 80]. The modelled states are the operating and fault states of the system. State graphs can describe any kind of finite discrete event system by enumerating the states, but the number of states grows drastically with the number of parallel activities generated by the system. Petri Nets are well suited to model discrete event systems with concurrent and synchronised activities and to cope with the combinatory explosion of the number of states.

For a quantitative dependability evaluation, it is necessary to take into account time as a variable. In a mechatronic system, the delay of state change of a device is captured by associating a delay to a place or a transition in the corresponding Petri Net.

Delays related to repair and fault process are generally modelled by random variables with exponential distribution functions. A Petri Net containing only stochastic time delays is known as a Stochastic Petri Net [FLO 85]. If we allow immediate firing transitions (for synchronisation modelling), the model obtained is the Generalised Stochastic Petri Net. In both cases, the successive marking of the net can be represented by a Markov Chain and therefore, dependability will be evaluated analytically. Many dependability studies on computer systems use this method [FOT 97].

More generally, it may be useful to model state changes of a device that do not represent fault or repair process, but a change related to the regular behaviour of the system. In this case, the delay of the state change is modelled by the designers with a distribution function on a time interval [ERE 96]. Dependability results are then generally obtained by Monte-Carlo simulation: many hihistories are simulated during the mission time and the average number of hihistories that reached feared event is computed.

The delay of a state change may also depend on the physical evolution of a continuous process. Inversely, the configuration of the system influences the evolution of the continuous process. This is typically the case in mechatronic systems where the control system is more particularly devoted to constrain some process variable within specified limits. As a consequence of an initiating event, some process variable might cross these limits, and the control system modifies the system configuration to influence the evolution of the process and bring back the system between its regular limits.

This hybrid point of view is essential to evaluate dependability of mechatronic systems. Indeed, both continuous and discrete parts dynamics influence the dependability of the mechatronic system. Reconfigurations will succeed only if it take place during a « grace period » which goes from the date when the control boundary is exceeded to the date when the feared event occurs. The duration of the « grace period » depends on the dynamic of the operative part and the duration of a reconfiguration depends on the control system and actuators dynamics [MAR 96].

Today, tools for modelling and simulation of hybrid systems exist. The control part is modelled by means of Petri Nets or State Charts. The continuous part is generally modelled by differential algebraic equations. But it is yet difficult to achieve numerical integration for Monte-Carlo simulation in a reasonable time.

A way of solving the problem is to derive an abstract model of the operative part. Indeed, it is often possible to transform differential algebraic equations into explicit and purely algebraic ones. By means of Coloured Petri Nets (CP-nets), the operative part will be modelled in this way. The behaviour of all parts of the system can be captured in a CP-net.

For all these reasons, Coloured Petri Nets [JEN 92, JEN 94] were chosen to model our system for simulation purposes. We use the DesignCPN tool [JEN 97].

## 2. CASE STUDY AND MODELLING

### *2.1 Case study*

We study a simple mechatronic system (Figure 2), derived from a more complex system, whose purpose is to maintain a level of pressure (P) in the range [Pmin,Pmax]. The functional constraints are given here after:

- If P>Pmax then electrovalve is closed,
- If P<Pmin then electrovalve is opened,
- If P>Palarm_max or P<Palarm_min then the system fails

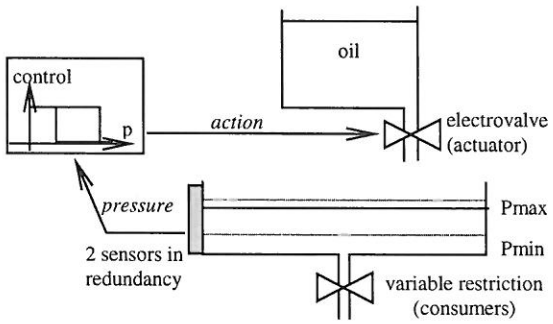



Figure 2 : Principle of the studied system

### *2.2. Functional and dysfunctional model*

The only fault considered here is the leak of the tank. The system can be then modelled by algebraic relations as follows (P is the tank pressure, V the volume, $Q_{in}$ the input flow, $Q_{out}$ the output flow, $Q_{consumers}$ the consumption flow, $Q_{leak}$ the leak flow and $Q_{pump}$ the pumpflow):

(r1): P = f(V), f reversible,

(r2): $V = (Q_{in} - Q_{out})*(t - t_0) + V_0$ with Qout = Qconsumers + Qleak, Qconsumers (t) and Qleak (t) can be whatever step functions,

(r3): if the electrovalve is opened then $Q_{in} = Q_{pump}$ else $Q_{in} = 0$.

Relation (r1) and (r2) are explicit algebraic equations, which allows to compute the date of a threshold overshoot by P ($P=P_{threshold}$), given an initial state ($P=P_0$, $V=V_0$ at time $t=t_0$):

$$t_{threshold} = t_0 + [f^{-1}(P_{threshold}) - V_0] / (Q_{in} - Q_{out})$$

### *2.3. CP-nets*

We suggest to represent our system by three successive CP-nets ranging from the more easy to read to the more efficient in terms of simulation time.

The so called « specification » model describes at each sampling date the interactions between operative part and control. This representation is close to the way of thinking of the designers. The principle of the CP-net is the following (Fig. 3): for each device, one place contains a token describing the state of the system and the associated

starting date. In our example, such a state is described by the pressure and volume in the tank, the phase of the system (position of the electrovalve), the level of consumption and the kind of leak. Each time a discrete event occurs like a change of consumption, a change of electrovalve position or the occurrence of a leak (transitions *Change Consumption, Change Actuator* or *Leak* are respectively fired) the state is updated and a time stamp is associated to the token indicating the date when the next feared event will appear (transition *Failure* fired). The present state is calculated knowing the previous state and associated starting date, according to relations (r1) and (r2). At each sampling date, the control system read the pressure in the tank (transition *Read Pressure*) and update the command (place *pos req*).

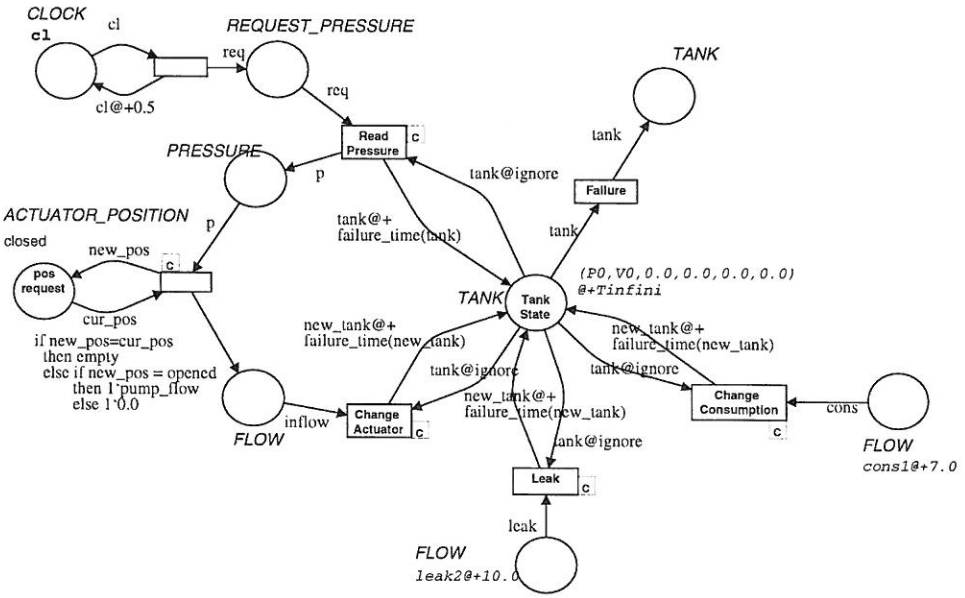



Figure 3 : The « specification » CP-net

We can remark that calculating the new command at each sample date is useless. Indeed, the command changes only once given pressure limits are overshoot. Three pressure intervals are sufficient to describe the command effect. We must add two prohibited intervals which define the feared events (Figure 4).

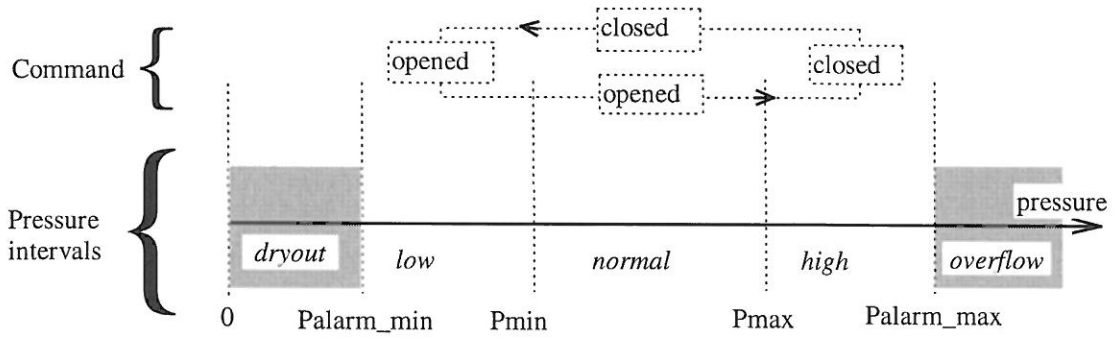



Figure 4 : Pressure intervals and the related command

By using these predefined intervals, it's possible to build an abstract model which represents the same behaviour as modelled by the « specification » model (Figure 5).

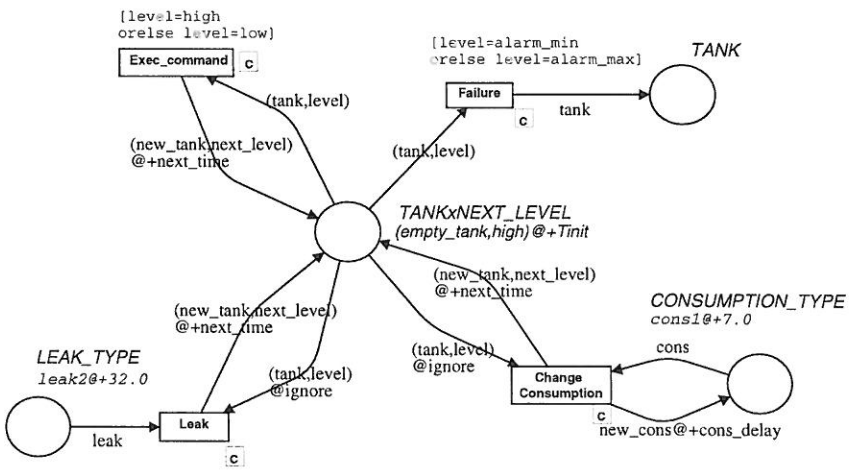



Figure 5 : The abstract CP-net

Each time an external event occurs (leak or change of consumption), the new state (pressure and volume in the tank, pressure interval, position of the electrovalve, level of consumption and the kind of leak are contained in the *new_tank* variable on arc inscriptions), the occurrence date and the current pressure interval are updated. Then, the next interval that will trigger a decision (command or failure) and the delay when this interval will be reached are calculated (*next_level* and *next_time* variables). A « jump » to the decision level can be realised, the state is then updated. According to the level reached (guards on the transitions Exec_command and Failure decide which transition is enabled), the command is calculated or a failure is detected. Moreover, as far as the command is concerned, the next decision level and delay when it will be reached is calculated. Notice that a threshold overshoot is detected by the control system after a delay due to the sampling. This effect can be taken into account by adding to the ideal date of a command switch a small $\Delta t$ corresponding to this delay. Behaviours of both models are then strictly the same (Figure 6).

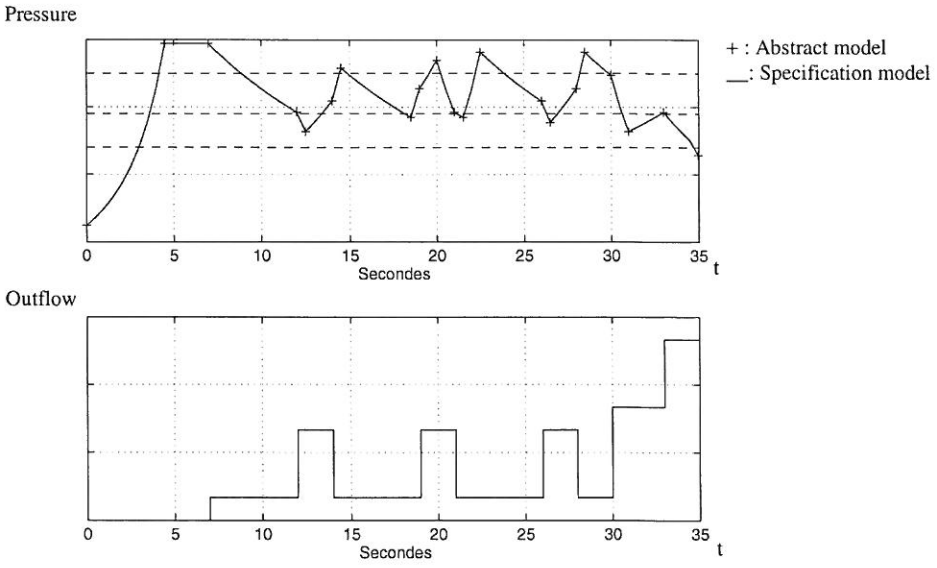



Figure 6 : Comparison of behaviours simulated by the specification and abstract models with a 0.5 seconds sampling period

At last, an optimised and equivalent model is deduced from the previous one, so that simulation can be done without the simulator of the tool DesignCPN. This model is written in ML code (the functional language used by

DesignCPN) and shares the data structures and functions used by the abstract model. As shown on Table 1, these successive models allow to improve simulation times.

| | Specification model | Abstract model | ML function |
|---|---|---|---|
| Simulation time in seconds | 745 | 75 | 1.5 |
| Acceleration | 1 | 10 | 500 |

Table 1 : Computation times for a 10 hours mission run and a 0.5 seconds sampling period

## 3. DEPENDABILITY EVALUATION

The system described till now fails only if a leak occurs. Since this fault leads surely to the dryout feared event, it is not necessary to build such a model to evaluate probability to get into the dryout state : we only need to have the leak probability. But suppose that one failed component can be repaired in a very variable time (several reparations may be needed) and that the time available for the reparation depends on the continuous process. We need then to model the system from this hybrid point of view.

This is actually the case we study now. The electrovalve is an actuator which may fail on demand and remain blocked in the current position. Designers experimented that if the electrovalve is blocked in closed position, it may be possible to unblock it by "shaking" it (electric pulse train) during a time called *Tshake*. If the shaking failed, the shaking will be repeated after an idle period which lasts *Twait*. This strategy is also useful when the pump has failed. Indeed, it also is possible to prime it again if the electrovalve move quickly between its extreme positions, as it is the case when it is shaken.

This new control policy could be chosen each time the pressure get under a pressure limit called *Pshake* (due to the closed blocked electrovalve or a pump failure), and abandoned as soon as the pressure grows again.

### *3.1. Quantitative modelling*

We modelled the electrovalve faults, and the corresponding control policy as suggested before. The abstract model is given in Figure 7.

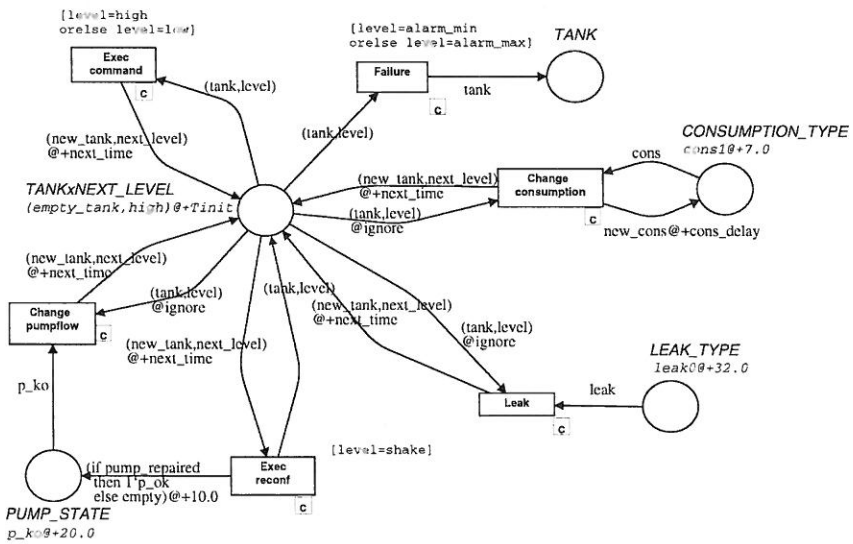



Figure 7 : Abstract model with possible repair of the pump and electrovalve

Monte-Carlo simulations showed that for a given set of parameters (mission time, sampling period, fault rates, *Pshake*, *Tshake*, Twait and consumption profile), the probability occurrence of the "dryout" event at the end of the mission time could be divided by ten with a 50 % of reconfiguration success for the electrovalve, and 50% of reconfiguration success for the pump (Figure 8). We could so observe and show the interest of this strategy. A simulation of 1000 histories each corresponding to 1000 hours of real time behaviour takes 40 hours.

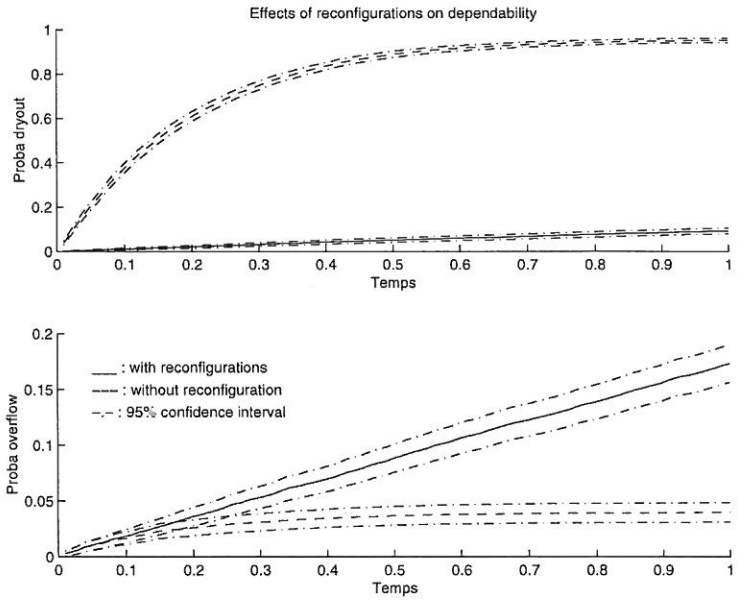



Figure 8 : Effects of reconfigurations on dependability

We can notice that one side effect of the reconfiguration used is that the probability to get the « overflow » event increase with time more quickly than without the reconfiguration. This is due to the fact that failure rates are very high : less « dryout » events let more chance to get « overflow » events.

*3.2. Qualitative modelling*

The Occurrence Graph is generally used to verify dynamic properties like boundedness or liveness properties, and more generally to validate the model. It contains all the possible states the system can reach and how they are reached (an arc represents a transition firing and so an event, a place represents a PN marking and so a state of the system).

But the Occurrence Graph cannot be exhaustive, and so useful for a model validation, if tokens can take an infinite number of possible values. That is the case in our models where time is explicitly represented by a continuous variable. Indeed faults may happen at any time and so an infinite number (non countable) of histories can occur.

A solution is to consider the associate qualitative model where time is not handled any more, but only the order of the events sequences is considered. In the previous quantitative model, the domains of the continuous process variables is naturally divided into discrete levels The evolution of the system is determined by the actual discrete state and the occupancy duration of this state. To get the associate qualitative model, we only need to eliminate explicit time references. Time will be modelled then by the order when events occur. The Occurrence Graph of the associate qualitative model can be so completely build. The qualitative model of the studied system is given in Figure 9 (the Occurrence Graph contains 21 nodes and 34 arcs).

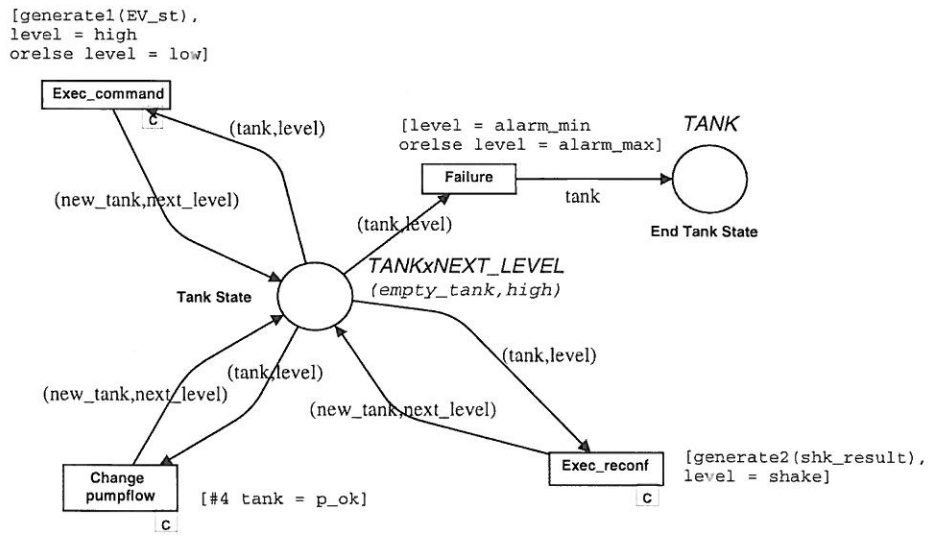



Figure 9 : Qualitative model derived from the abstract model of Figure 7

To validate the model, we can prove, for example, that dead markings represent no other states than faulty states (no dead markings due to a dead lock). We can also verify that some well known scenarios happen as expected by building it directly thanks to the Occurrence Graph tool or the simulation tool.

We can also explore systematically all the possible scenarios (and possibly find some unexpected ones). This analysis is based on the strongly connected component (Scc) of the occurrence graph.

Indeed, the notion of Scc has a useful interpretation from the dependability point of view. Any state of a Scc can be reached from any other state of the Scc (Figure 10). Two non exclusive cases are possible in the case of hybrid systems :

- the Scc contains a set of states covered cyclically during a possible infinite time which corresponds to the supply of a service in a nominal or degraded operating mode,
- the Scc represents a transitory evolution which ends when conditions of a feared event are detected (when a process variable crosses a given threshold, the system reaches an absorbing state ).

Any arc which goes out of a Scc means that the associated event prevents the system from going back to the

previous operating mode or transitory state. These events are called critical events (thick arrows on Figure 10). There are two kinds of critical events :

- non repairable faults,
- detection of a feared event occurrence.

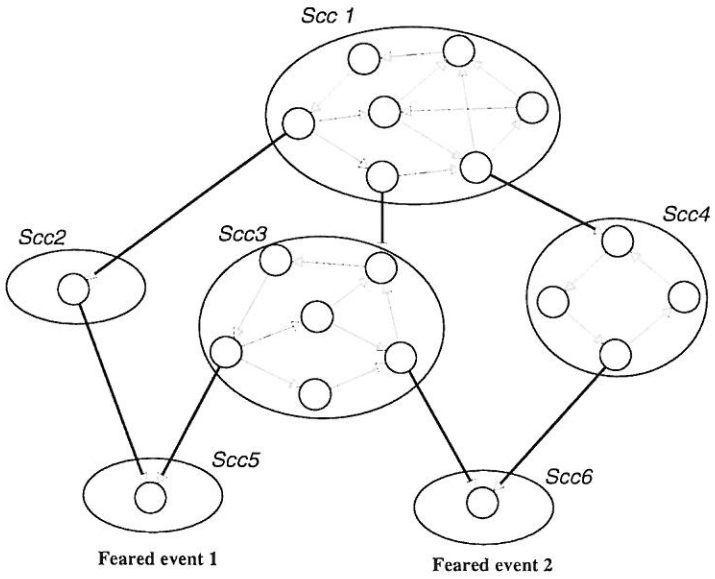



Figure 10 : Exemple of Strongly Connected Component Graph

The analysis is done in three steps and consists on :

- defining a partition of dead markings. Each of the disjoint sets corresponds to one expected feared event,
- interpreting all the non trivial Sccs and finding the functionning mode it represents,
- finding and interpreting all the possible paths (sequence of events - one event is caracterized by a binding element) from each Scc to the next Sccs or feared events. These paths define trees of bindings and one branch describes one of the sequences of events that occur when the system leaves the considered functionning mode. Many branches of these trees can be merged if we group together bindings that represent the same events.

This work of interpretation involves much of the knowledge the designer have on his system and cannot be done automatically. It could be difficult and require a long time for complex systems.

Note that the dead markings describe generally the set of faults which lead to the considered feared event. But it gives no information on the order in which these faults occurred, which may be non trivial for complex systems. Indeed, some sequence of faults may lead to a feared event whereas the same set of faults in a different order may not. Moreover, a fault may trigger a feared event only in presence of a particular solicitation of the system. In this case, the dead markings give no information about the sequence of solicitations (combined with faults) that lead to the feared events. The analysis of the paths going out of the Sccs is so generally essential.

For our example, we can prove that only one Scc (related to a total of 15 Scc nodes) contains a cycling set of states corresponding to the support of the acceptable level of pressure. We can prove that only two kinds of events trigger the exit of this Scc and directly lead to feared events :

- a reconfiguration failed to repair the pump or the closed blocked electrovalve,
- the electrovalve remained opened blocked when asked to close.

## 4. CONCLUSION AND PROSPECTS

CP-nets is well adapted to describe an hybrid system model, with the assumption that the operative part can be modelled by explicit algebraic equations. Moreover, an ML code for Monte-Carlo simulations can be deduced and validated through an abstract CP-net. We first got dependability evaluation using this method on a simple mechatronic system.

To complement this quantitative dependability evaluation approach, an important issue is to determine all the possible scenarios, particularly rare scenarios which cannot be easily shown by Monte-Carlo simulation. We showed how useful is the DesignCPN occurrence graph analysis tool in some simple case.

The next step will be to use these approaches on a more consequent system.

## BIBLIOGRAPHY


[ERE 97] Jean-François Ereau et Malecka Saleman: « Modelling and Simulation of a Satellite Constellation based on Petri Nets », Annual Reliability and Maintenability Symposium, Proceedings 1996.

[FOT 97] Nicolae Fota: « Spécification et Construction Incrémentale de Modèles de Sûreté de Fonctionnement - Application au CAUTRA », thèse présentée au LAAS, 1997.

[FLO 85] G. Florin et S. Natkin: « Les réseaux de Petri stochastiques », Techniques et Sciences Informatiques, vol. 4, n°1, 1985.

[GUY 94] Jacques Guyot: « Mechatronic components design in the automotive industry », Proceedings of the 2nd Japan-France congress on Mechatronics, Japan, 1994.

[HEN 96] Valéry Hénault: « Méthodologie de développement des systèmes électroniques embarqués automobiles, matériels et logiciels, sûrs de fonctionnement », thèse présentée à l'IRESTE, septembre 1996.

[JEN 92] Jensen, K. (1992) Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 1, Basic Concepts. EATCS Monographs on Theoretical Computer Science, Springer-Verlag.

[JEN 94] Jensen, K. (1994) Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use. Vol. 2, Analysis Methods. Monographs in Theoretical Computer Science. Springer-Verlag.

[JEN 97] Jensen, K.; Christensen, S.; Huber, P.; Holla, M. (1997) Design/CPN Reference Manual. Computer Science Department, University of Aarhus, Denmark. On-line http //www.daimi.aau.dk/designCPN/.

[LER 92] Alain Leroy et Jean Pierre Signoret: « Le risque technologique », Collection « Que sais-je ? », 1992.

[MAR 96] M. Marseguerra & E. Zio: Monte Carlo approach to PSA for dynamic process systems, Reliability Engineering & System Safty, vol. 52, 1996

[PAG 80] A. Pagès et M.Gondran: « Fiabilité des systèmes », collection de la Direction des Etudes et Recherche d'Electricité de France, 1980.

# Modeling The Execution Architecture of Mobile Phone Software System by Colored Petri Nets

**Jianli Xu**
Nokia Research Center
P.O.Box 45
Helsinki, Finland
+358 9 4376 6634
jianli.xu @research.nokia.com

**Juha Kuusela**
Nokia Research Center
P.O.Box 45
Helsinki, Finland
+358 9 4376 6325
juha.kuusela @research.nokia.com

## ABSTRACT

We present an application of Colored Petri Nets for modeling of software architecture. Our work demonstrates how to use architectural modeling to control properties of industrial scale products and product families. The new software architecture of a mobile phone family is modeled by Colored Petri Nets and analyzed using Design/CPN tool. The model describes the execution architecture and allows us to analyze both time and space performance of the software system. Our experience shows that a special purpose model can be constructed also for large systems and this model predicts accurately enough system properties to be useful. The insight this gave us to the internal operation of the software system together with the estimates on its performance guided us through the early phases of architectural design. We continue to maintain the model and expect it to be very valuable tool in configuring different product family members.

### Keywords

Software Architecture, Execution Architecture, Modeling, Colored Petri Net, Performance

## 1 INTRODUCTION

### 1.1 Architectural Models

Software architecture [1] provides a global view to software system and accordingly software architecture models can be used to predict and control system wide properties. In the ARES[1] project we have built several models of module architecture showing how these models support configuration of variants in a product family [2]. We have also tried several different techniques for modeling execution architectures [3].

Architectural models are used to analyze system wide properties and they have to be complete in respect to that property. Consequently, architectural models often have to be limited to a single viewpoint. The architectural model documented in this paper concentrates on managing dynamic system properties and on predicting system performance.

[1] ARES is supported by the European Commission under ESPRIT framework IV contract #20477 and is pursued by Nokia, ABB, Philips, Imperial College, Technical University of Madrid, and Technical University of Vienna.

From industrial perspective architectural modeling is not yet a mature field. In order to make architectural modeling more useful for software engineers:

- Modeling techniques should be classified according to the reasoning they can support.
- Pre-made models should exist for different architectural constructs allowing analysis of different attributes.
- Models should be composable so that the composite model will correctly predict the properties of the composed system
- Model-checking techniques should be more practical. Models can be partitioned to overcome computational limitations but no guidance exists on how to do it.
- Handbooks on how to model different architectural constructs should exists, currently the results depend entirely on the personal skills of the architect.

Due to various reasons, documented examples of architectural modeling have been small systems. The example used in this paper shows that it is possible to create specific models for analyzing interesting architectural properties of large systems. Several architecture description languages (ADLs) have been developed to model architectures and their properties (see [4]). Here we demonstrate that general purpose modeling techniques like Colored Petri Net[5] and Design/CPN tool[6] can also be applied.

Architectural models share the system structure. The decomposition of the model into parts follows the decomposition of the system and mapping from the entities of the model to the entities of the system is clear. This means that architectural models directly assist in designing and configuring different system variants. Since architectural models share the system structure, it is easy to see how well the model conforms to the system. This increases confidence that the model gives correct predictions.

## 1.2 Why Colored Petri Net is Chosen for Architectural Modeling

Now in software research community, several common ADLs have been developed to support descriptions for components, connectors, and other aspects of software architecture, like styles, constrains, or design rationale [4][9], they emphasize the structural aspects of the system but are rather weak at describing the performance aspect. In [10] the Unified Modeling Language (UML) [11] is adapted as a ADL in order to integrate the power of ADLs with the day-to-day usefulness of UML.

In designing the architecture of mobile phone software system, we use UML as the ADL for architecture description. Because UML is a standard now and most of our designers can understand it, UML descriptions of the architecture not only provide a standard definition of the system structure and system terminology, but also provide a common and easy language for our designers to communicate with each other. In the architecture descriptions we use UML diagrams instead of those box and line diagrams traditionally used in our software development process.

Designing the architecture of a software system include organizing the system as a composition of components; developing global control structures; selecting protocols for communication, synchronization, and data access; assigning functionality to design elements; physically distributing the components; scaling the system and estimation performance; defining the expected evolutionary paths; and selecting among design alternatives. With UML we can analyze the problem domain and create an architecture model of the solution domain addressing the above architectural aspects , but it is very hard or may be impossible to assess the correctness of our architecture design and how well the designed architecture meets the system requirement. A formal model of the architecture design is needed for specifying and proving the system properties, especially the performance and dynamic properties. Our designers dream about an executable formal model of their architecture design, so that they

can use the model to run different use cases, estimate the performance more accurately, try different system configurations, and select the right design alternatives.

Currently there are many formal modeling methods or languages, such as Z, VDM, CCS, CSP, LOTOS, temporal logic and Petri Nets, just to mention a few. Colored Petri Nets (CPN) have the following advantages over other methods in modeling:

- A CPN model is an intuitive description of the modeled system. It can be used as a specification or presentation. CPN diagrams resemble many of the informal drawings, which are made by designers while they construct and analyze a system.
- CPN supports hierarchical descriptions. This means that we can model large and complex systems in a manageable and compositional way.
- CPN can be extended with time. Therefore we can use the same modeling method for specification and validation of both functional (logical) properties and performance properties.
- CPN has computer tools supporting model building, simulation and formal analysis. Design/CPN is one of the most successful tools, and it has already been used in many practical systems of many application areas.

CPN offers two mechanisms that are of special interest for architectural modeling. First is the flexibility of token definition and manipulation. It is possible to use tokens to model various architectural elements. Second is the concept of sub-page. Sub-pages can be used as a module definition mechanism for various purposes.

Token flexibility comes from the inscription language CPN ML. CPN ML is based on the functional language Standard ML (SML) [7]. Each token in CPN has its own value of a pre-defined data type. Different token types can be used to represent different architectural elements. In our case (see the later parts of this paper), components, tasks, messages, events and even use cases are all described by different types of tokens. The value of tokens can be investigated and modified by the transitions corresponding to different system behaviors.

Sub-pages can be used to created hierarchical system model. The model can be developed either top-down or bottom-up. CPN provides well-defined interfaces between sub-models, and sub-models can be reused. This feature of CPN is very useful in reconfiguring the model in order to analyze alternative polices and mechanisms. It can also be used as a mechanism to define the dimensions of variation in the product family.

The paper is organized as follows. Section 2 in brief introduction to the mobile phone software system and its architecture design. Section 3 describes how the CPN model is created. Section 4 shows the simulation conducted on the CPN model, and section 5 briefly describes the formal analysis of the model. Section 6 summarize our experience and results.

## 2 SOFTWARE SYSTEM OF A NEW FAMILY OF MOBILE PHONES

The product family of Nokia mobile phones consists of a large number of products for each mobile communication standard. As the market has evolved, requirements have changed substantially. The phone has become a part of a distributed system with software running concurrently on handset, Data-card, PC or a car control system. Image and live video transmission have made the real time requirements harder. New requirements also increase the size of the product family and the variation inside the product family.

A new component-based software architecture has been designed to manage the complexity of the product domain and guide the development of new generations of Nokia mobile phones. This architecture defines software components, message interfaces between components, essential use cases, component grouping and deployment structure.

We follows the "4+1" views architecture design approach [12][13] and use UML to describe the architecture. Figure 1 shows the simplified software architecture of the mobile phone family as a UML class diagram (from the logical view). There are three major parts in the architecture: *System Objects*, *Utilities* and *Communication & Control Kernel*. *System Objects* can be *Client* objects or *Server* objects. *Client* objects interact with users to provide them the custom features of the mobile phone, they are mostly user interface components. *Server* objects manage and control the functional resources inside the system provide basic system services to *Client* objects and other *Server* objects. Examples of resources are call, short message, data call, phone number book, window, energy, and so on. During the execution time, *System Objects* are grouped into concurrently executing threads called tasks of the operating system. *System Objects* communicate with each other through the *Communicator* in *Communication & Control Kernel*. *Communicator* is responsible for message routing and both local and remote message passing between *System Objects*, it has different policies for inter-device and intra-device communication, and different mechanisms for inter-task and intra-task communication. *Communication & Control Kernel* is also responsible for system object management , task scheduling and event control. *Communication & Control Kernel* is the infrastructure of the system. Utilities package contains common facilities, such as display and memory management libraries, which can be accessed by all other components. *Communication & Control Kernel* is generic to all products in the family. Different products may have different configuration of system objects due to different product features and hardware resources.

The new architecture supports the need for different product configurations. However to be able to predict the performance of the system in different configurations before it is implemented, we have to develop a model of the execution architecture.

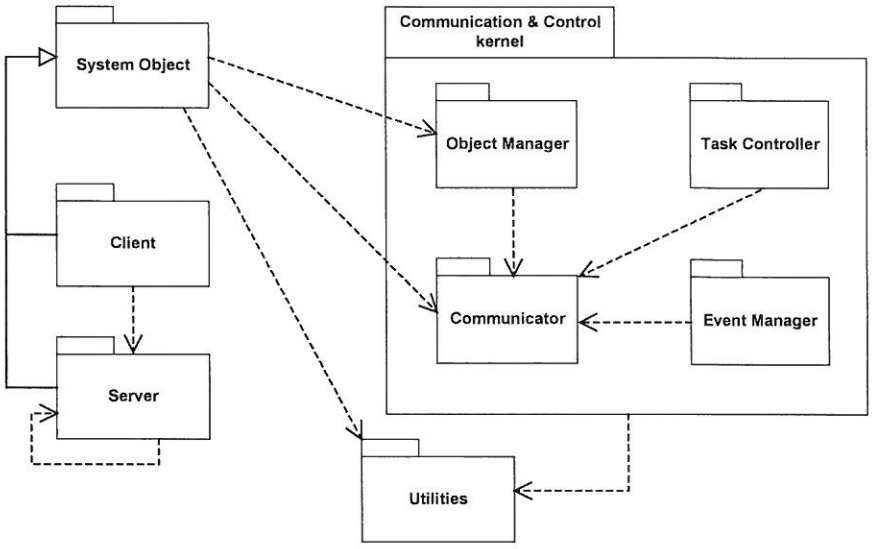



Figure 1. Module architecture of the software system

We used CPN as the modeling language. So far, we have used this model to:

- specify and verify the task control mechanism
- specify and verify the task communication mechanism
- evaluate different task divisions and allocations
- simulate typical use cases
- estimate the message buffer usage and message delays

We started the modeling just after the principal structure of the new software architecture had been outlined and modeling has continued along with the component architecture design and detailed system design. The CPN modeling iterates over three main steps: creating or modifying the model, simulating, and analyzing.

## 3 CREATING THE MODEL

Before we started to create the CPN model, we had spent about one and a half person months to prepare for the real work, including learning CPN and SML, learning how to use Design/CPN tool, and doing small case studies on Design/CPN tool. In the beginning of the project we used the Design/CPN tool on a Mac-PowerPC with 32Mb RAM, because at the same time we also tried a ADL tool as an backup of UML which only run on Mac-OS. After three months, in order to speed up the simulation and to perform the state space analysis of some real cases, we moved to a SUN workstation (Sun SPARK 2, with 496Mb RAM). Before we moved to the new environment, we had already created the first CPN model. Because Design/CPN tools on these two different platforms do not share any data, we had to spend about two weeks to draw the whole model and test it again on the new platform.

The CPN model was created based on the structure and properties of this architecture. Figure 2 shows the top page of the model. The model contains data objects of devices, functional components (described as client or server), tasks, messages, message queues, and events. Interactions between objects task scheduling and message transmission mechanisms are described using CPN structure and inscriptions. This structure makes it possible to describe task allocation of components, components interactions, space and timing properties of message transmission mechanism, and task control.

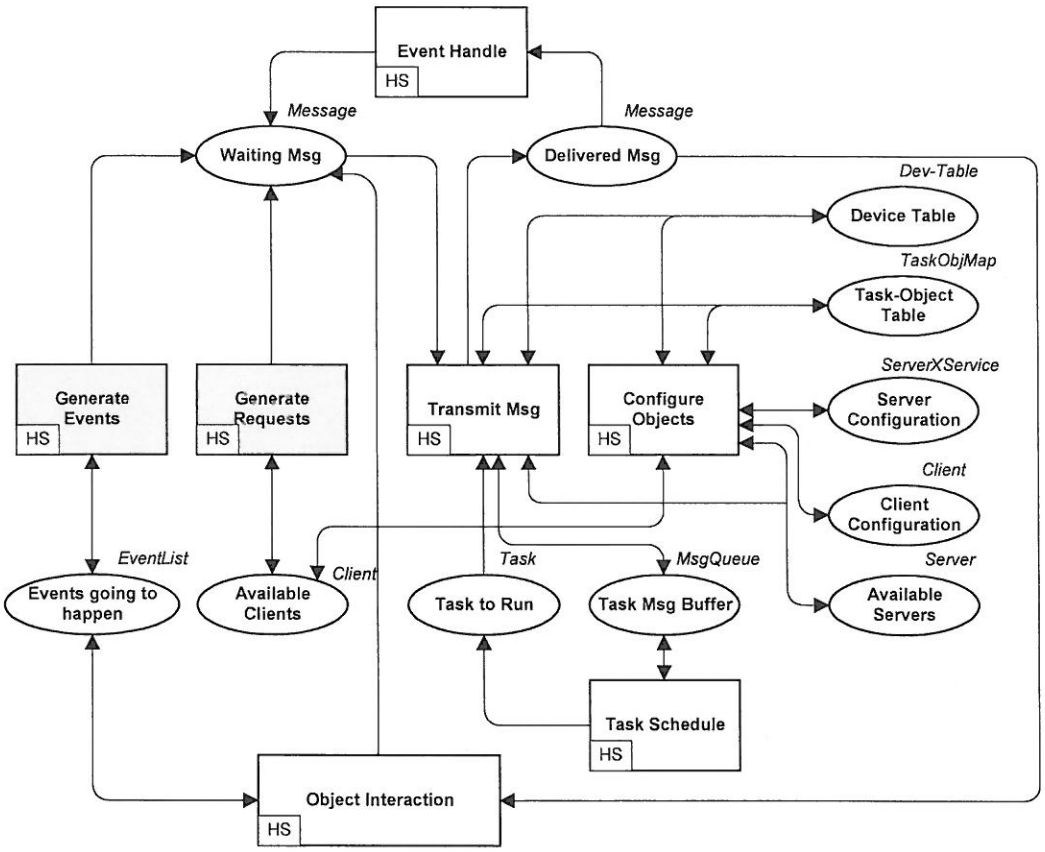



Figure 2. Top page of the CPN model

The mapping between the elements of the module architecture and the elements of the execution architecture is natural. Client and server objects are mapped to Client and Server

type tokens. Object Manager, Task Controller, Event Manager and Communicator modules are mapped to sub-pages *Configure Objects*, *Task Schedule*, *Event Handle* and *Transmit Msg*. The interactions between system objects are modeled by sub-page *Object Interaction*.

Since we are not interested in the detailed internal behavior of client or server objects in architectural modeling, client and server objects are abstracted by CPN tokens, which can be reconfigured by giving different initial markings. There are no corresponding components in the module architecture for sub-page *Generate Events* and *Generate Requests*. These two sub-pages are added to generate input (use cases) to the CPN model for the simulation purpose.

In order to support the development of an entire product family the model has to be a generic abstraction of the family and it has to be flexible enough to represent different device, component and task configurations. In our model, the part that has to be reconfigured and changed due to differences in different products is described by data variables and initial markings. They can be easily changed without affecting the overall structure of the model. For instance different initial markings in place *Device Table* represent different hardware configuration, different initial markings in place *Task-Object Table* give different task divisions, and different initial markings in places *Client Configuration* and *Server Configuration* describe different client and server configurations. The generic part of the model is its CPN structure, which represents the properties and mechanisms shared by the products in the family.

The basic system elements (such as client objects, server objects, tasks, devices, service types of each server, and messages between system objects, etc.) are defined by types of tokens. Compositions of the basic system element token types are used to describe higher level tokens, like configuration tables and use cases.

Figure 3 shows the most important token definitions taken from the global definition node of the model. The use cases has been classified into two categories and defined by `Usecase_C` and `Usecase_S` respectively. `Usecase_C` represents the use cases initiated by clients when a mobile phone user starts to use a certain feature, for example, makes a call, answers a call , sends a short message, or check a phone number, etc. `Usecase_S` represents the use cases started by the system (by server or hardware events), such as indicates an incoming call, indicates a short message received, indicates battery low, etc. Selected use cases of both types are described as initial markings in sub-page *Generate Requests* and *Generate Events* before we simulate the execution of them. There UML descriptions of the architecture have defined all the important use cases that fully cover the system functionality, they are collected in the use-case view of the architecture [*]. The use-case view can be used as input for use case definitions in the CPN model.

```
color   Resource =          with NONE | CALL | SMS | PNB | WIN | ENG | DATA |
                            KEY | DISPLAY |…;
color   DeviceType =        with HandSet | PDA | DataCard | SIMReader | PC| …;
color   Media =             with XBUS | YBUS | RS232 |INFRD| ANY|…; (*
                            communication connections between devices *)
color   DeviceState=        with Active | Deactive | Busy | Not_available;
color   DeviceID =          int;
color   Device =            record d_type: DeviceType * d_id: DeviceID;
color   Dev_Table =         product Device*DeviceState * Media;
color   ObjectID =          int;
color   Object =            record res: Resource * dev: Device * obj_id:
                            ObjectID timed; (* for a client object res=NONE,
                            for a server object res shows the resource it
                            controls *)
color   Client =            Object;
color   Server =            Object;
color   Service =           with Create_Call | Answer_call | Release_Call |
                            Send_SM | Receive_SM | Read_PN | write_PN |
                            Create_Win | Destroy_Win | …; (* important service
                            functions and event functions provided by servers
                            are defined here, about 50 items*)
color   ServerXService=     product Server * Service;
color   TaskID =            int;
color   Priority =          int;
color   Task =              record dev: Device * t_id: TaskID * pri: Priority;
color   TaskObjMap =        product Task * Object;
color   MsgType =           with REQ | RESP | CONF | IND;
color   MsgID =             record res: Resource * msg_fun: Service *
                            msg_type: MsgType;
color   MsgLength =         int;
color   Sender =            Object;
color   Receiver =          Object;
color   SubBlock =          MsgID;
color   Message =           product Receiver * MsgID * Sender * SubBlock *
                            MsgLength; (* when necessary more field can be
                            added, like a field of data content. *)
color   Event =             MsgID;
color   MsgList =           list Message;
color   MsgQueue =          product Task * MsgList;
color   EventList =         list Event;
color   ServiceReq =        product Resource * Service;
color   ReqList =           list ServiceReq;
color   Usecase_C =         Product Client * ReqList;
color   Usecase_S =         EventList;
Color   …;
```

Figure 3 Important token type definitions in the global definition node

Figure 4 shows a simplified example of sub-page *Generate Requests*. In sub-page *Generate Requests*, an user may send a request to the system in every based on a probability distribution "x_distribution(p)". The distribution function and the value of "UserReqTime" is decided by the experience and some statistical analysis on our current products. An available client will send a service request to a server when an user request comes. The service request is taken from the use cases predefined in the initial marking of place The initial marking of place *ClientUseCase* describes the following use cases: a client on a PC ("PcClient") will send s short message and then add a new item into the phone number book in the handset; a client in the handset ("HandsetClient") will get a number from the phone number book, make a call by that number, and finally release the call. The service request is converted into a message and is send to the corresponding server. Message routing and transmission is carried out in sub-page *Transmit Msg*. After the server receives the request message, it will perform the request and send a response message to the client. While the server processing the service request, it may exchange and a few more messages with the client. These client-server interactions are modeled in sub-page *Object Interaction*.

Similar methods are used for describing and generating `Usecase_S` type of use cases (in sub-page *Generate Events*) and hardware signals (in sub-page *Wait Signal*).

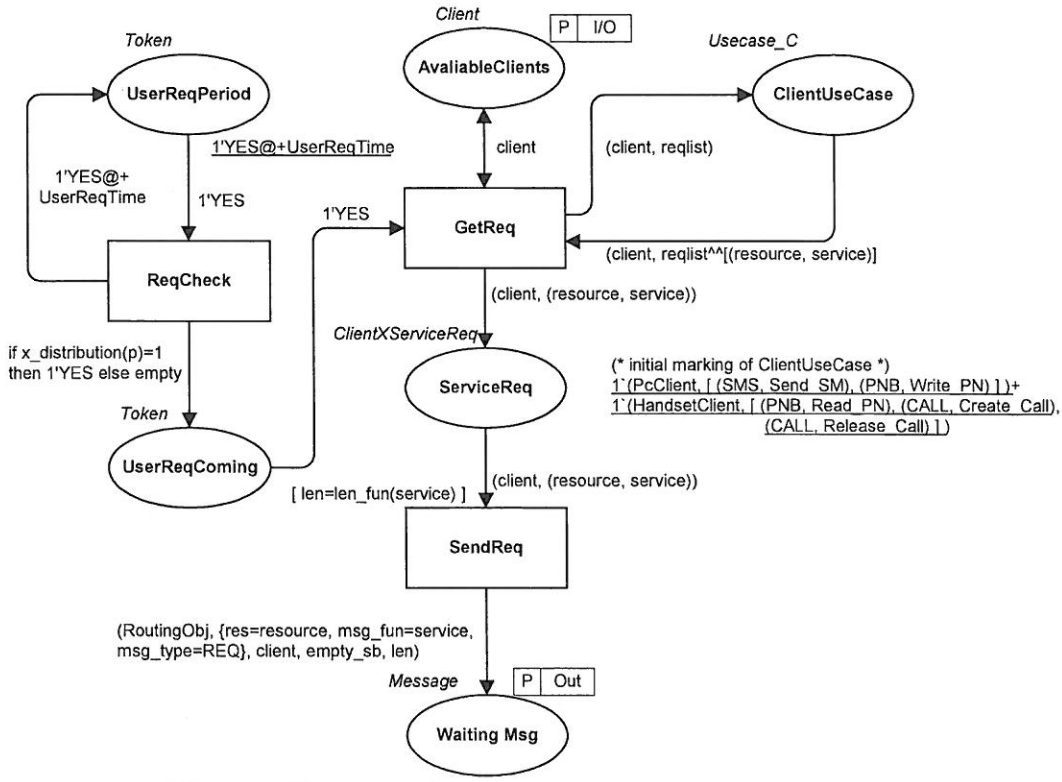



Figure 4 Example of use case description and initiation

In sub-page *Object Interaction*, in order to focus on architecture properties, we only modeled some most important interactions, such as call handling and short message handling interactions, to a reasonable detail level, and abstracted other interactions just as request-response or indication-confirmation pairs.

The model is hierarchical as shown in Figure 5. Sub-pages (sub-models) in the hierarchy can be replaced or reconfigured. The model building on Mac platform took about one person month. The first model has been changed greatly through simulation to the current model.

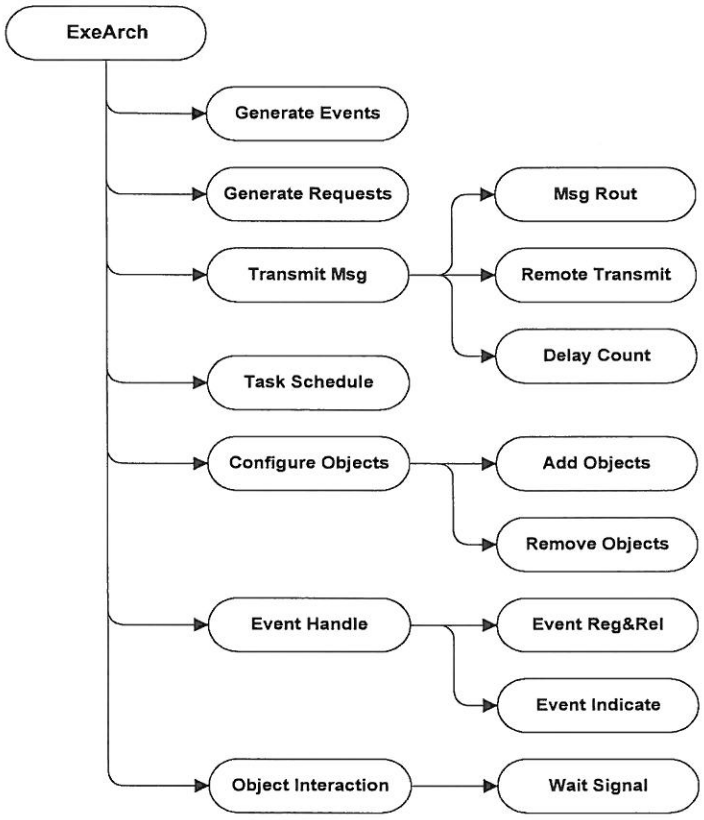



Figure 5. Hierarchy of the CPN model

## 3.1 Simulation

Deep understanding of the system structure and experience in CPN is required to develop a correct model. As a starting point, we used the component architecture model that shows all components and their interfaces. We structured the execution architecture model by grouping these components into sub-pages. Each sub-page corresponds to a certain class of actions (interactions between components) and a group of actors (components) defined in the module architecture. This structural correspondence helps in validating the model.

Once the kernel part of the model was ready, we ran several use cases. Based on the system specification we know how the system should behave in each use case and we could debug our model. Design/CPN simulator provides interactive simulation, so we can run the model step by step watching the state change and token flow for every step. During this early stage of simulation, we found many inconsistencies. Because this is our first real application of CPN, most of the inconsistencies were errors in the model, but we also found several problems in the original architecture.

After the model was stable, we added time parameters, including message delays on different message links, task-switching time of the operation system, and event processing time. We also defined necessary statistical variables and computations to update them to get quantitative results. We created the input parts (sub-page Generate Requests and Generate Events) of the model to automatically generate the streams of user requests, events and network signals according to different probability distributions. Using these estimates we could compute

- the message buffer usage in the worst case
- minimum, average and maximum message delays (total values)
- and number of task switches needed for each transaction

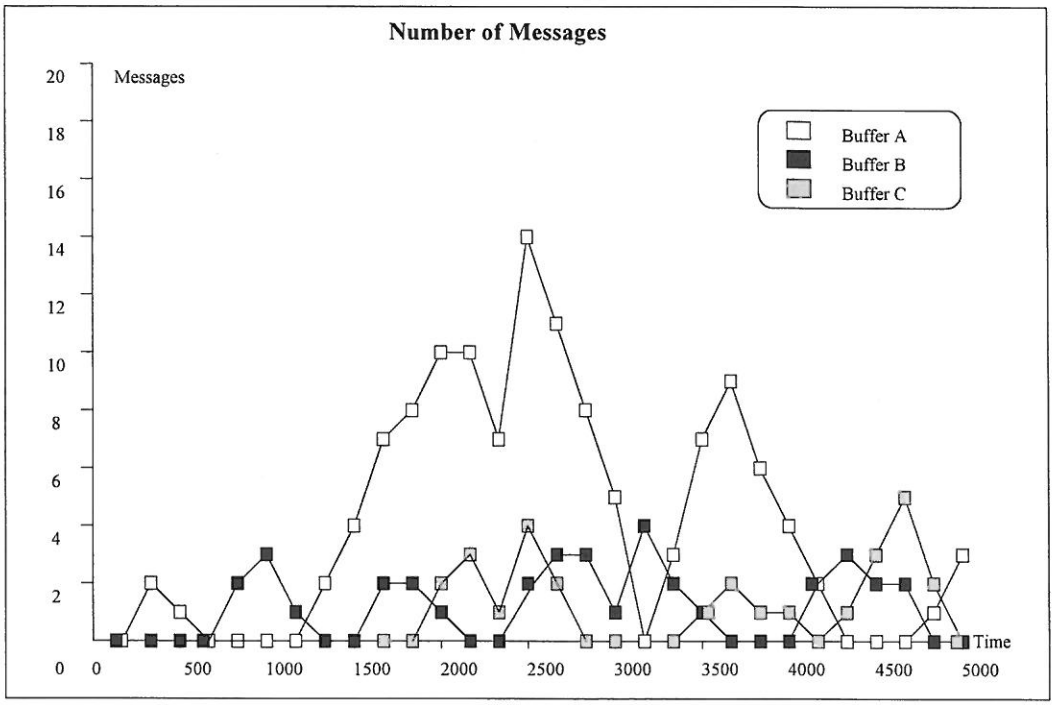



Figure 6. Line chart of message buffer usage

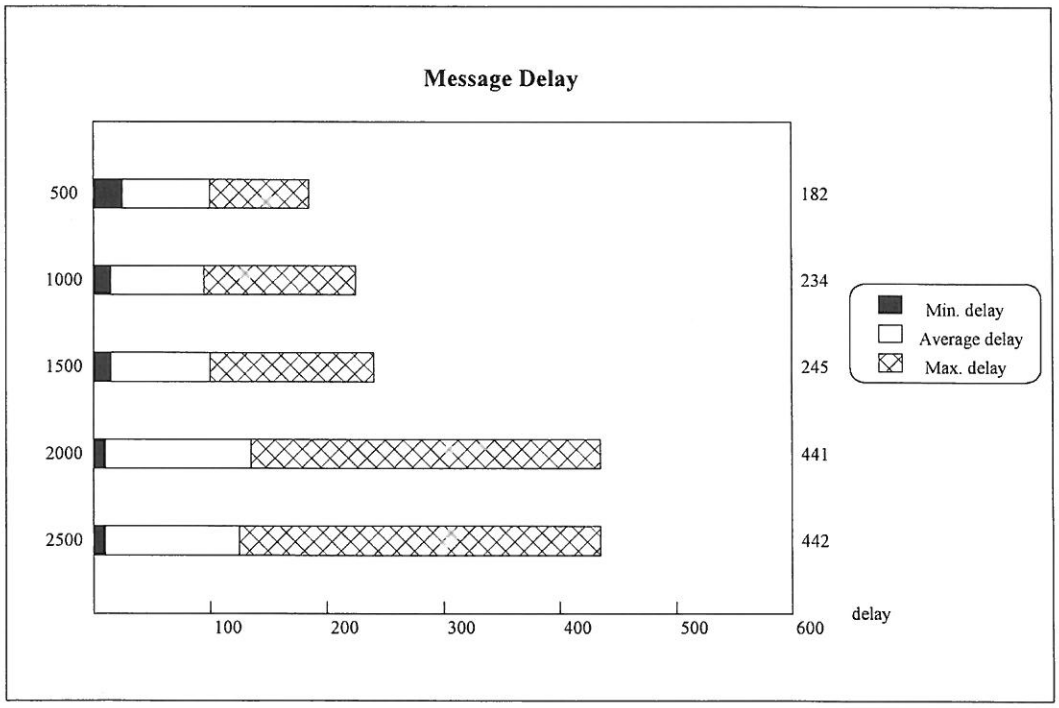



Figure 7. History chart of message delays

Simulation allows us to compare the performance and timing properties of different communication mechanism and control policies. Figure 6 and 7 show the statistic results of a simple use case simulation displayed by chart facilities of the Design/CPN tool. Figure 6 is the line chart of messages in three message buffers of the system at different time, and Figure 7 is a history chart of message delays during the simulation of the same use case, it is updated every 500 time units. From this history chart of message delays we can see the maximum and minimum message delays so far. During or after the simulation, we can accumulate and access statistics about the values of statistic variables generated during the simulation, and show the results by charts or store them in data files.

By simulating a lot of typical use cases and some extreme use cases, we estimated the maximum size of message buffers and the average transaction processing time. Estimated results are based on the actual timing properties of the existing system components and the assumption of the timing properties of new components to be developed.

Sometimes it is necessary to work the other way around. In another project for IP access systems, the total size of packet buffers was determined by hardware configuration. Based on the fixed space requirements, we estimated the timing parameters and used them as timing requirements to the component design and implementation.

We have found some potential problems of the new module architecture through simulation. The communication mechanism may be the bottleneck of the system, overall there are too many task switches, and too many steps in event handling. Description of potential problems together with quantitative analysis of their impact is valuable feedback for the system architect. If these problems emerge in the later phases of implementation, the cost to handle them is not predictable.

### 3.2 Formal analysis

We did some case studies of formal analysis on the Mac platform. Because of the limitation of processing power and memory restrictions of the platform, we cannot analyze the model as a whole. We have analyzed several important sub-models one by one with the occurrence graph (state space) tool [8]. The tool provides a set of standard queries investigating dynamic properties of the CPN model, such as boundedness, liveness, and home property. Unfortunately, we had to greatly simplify the sub-models in order to get results. We have not done any occurrence graph analysis after we moved to the SUN platform, but we are planning to do it soon.

Now we present an example of occurrence graph analysis. In the initial marking (state) there is one *Create_Call* request, and one *Call_Comging* event. That means these two things may happen in any order or even at the same time. One typical scenario of the use case implied by this initial marking is the call collision case, the user is making a call and at the same time, a call comes from the network. The occurrence graph tool successfully generated the full state space after we revised and simplified the model for several times. The standard analysis report of the full state space is given in Appendix, and part of the occurrence graph is shown in figure 8.

In the analysis report, the dead marking is also the home marking. It means that all of the interactions will end at the same state (marking [3659]). We checked this marking in the simulator and found it corresponds to a state where all the service requests and events have been processed and the system is ready to accept new request and event. So we can say that under this initial marking the interaction protocols of call control is correct (no deadlock and safely ended). The analysis report also gives the boundedness properties of every place in the CPN model. For example, under the given initial marking, the maximum number of active tasks is 5 (the upper bound of place *ActiveObjects*) and the upper bound of the message buffer *MsgToBeSent* is 5. The cyclic occurrence sequences in those strongly connected components are caused by processing the network signals generated by one transitions which simulates the radio interface to the mobile network.

State space analysis is also an efficient way to debug CPN model. We have found several non-trivial errors in the model during occurrence graph analysis. We found that in most of the time when we could not generate the occurrence graph or SCC, there were some problems in the model, for example unnecessary loops or problematic arc inscriptions. We have not gain many experience of formal analysis so far by just doing small case studies. We are planning to analyze the whole model on the new platform in the near future.

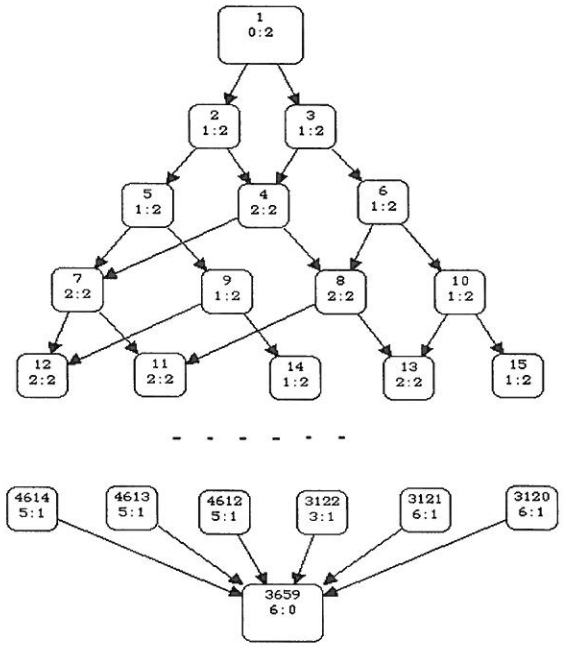



Figure 8. Part of the occurrence graph

## 4 CONCLUSION

Modeling has been successful. The requirements placed on the system execution architecture have been checked using the CPN model. Alternative mechanisms and policies have been evaluated using the model. We have also found some potential problems. Modeling provides valuable feedback to the development of the architecture. Design/CPN tool has greatly supported our work in designing the new architecture, CPN and Design/CPN tool are fit for this application area quite well if we use them properly.

We have demonstrated that architectural models can be built for industrial systems and general-purpose formal modeling techniques can be applied to architectural modeling. Modeling can provide quantitative results to be used as a guide in developing the system further. Existing tools have limited analytical power but simulation can be used to compensate. The execution architecture model also demonstrates the important benefit of architectural modeling: structural similarity helps in keeping the design and model coherent.

Our main problem was the lack of documentation on how to model software architectures using these techniques. We had to figure out how to model different constructs, where to approximate, how to model the input to the system and how to monitor the behavior. We have now applied the same technique also in modeling another system and it turned out to be a lot easier and much more effective than in the first case. We hope that the documentation of our positive experiences will make architectural modeling a more wide spread activity.

## 5 ACKNOWLEDGEMENTS

We would like to thank the organizer of this workshop for such a good learning opportunity. We are very grateful to all the reviewers for their constructive comments to our paper.

## 6 REFERENCES


1. E. Rechtin, M. Maier, "The art of systems architecting", CRC Press, Boca Raton, USA, 1997.
2. A.Ran, J.Kuusela, "Selected Issues in Architecture of Software Intensive Products", Proceedings of ISAW-2, ACM, 1996.
3. Jeff Kramer and Jeff Magee, "Analysing Dynamic Change in Software Architectures: A case study", 4th IEEE International Conference on Configurable Distributed Systems, Annapolis, May 1998
4. Nedad Medvidovic and David S. Rosenblum, "Domains of Concern in Software Architectures and Architecture Description Languages", Proceedings of the USENIX Conference on Domain-Specific Languages, pages 199-212, Santa Barbara, CA, October 15-17, 1997
5. K. Jensen. "Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use", Volume 1. Monogr. in Theor. CS, Springer-Verlag, 1992.
6. Computer Science Department of University of Aarhus and Meta Software Corporation. "Design/CPN Reference Manual", Cambridge, MA, USA. 1993.
7. L. C. Paulson. "ML for the Working Programmer", 2nd Edition. Cambridge University Press, July 1996.
8. Computer Science Department of University of Aarhus, "Design/CPN Occurrence Graph Manual", Version 3.0. Aarhus, Denmark, 1996.
9. M. Shaw, "Truth vs Knowledge: The difference between what a component does and what we know it does", Proceedings of 8th International Workshop on Software Specification and Design, March 1996.
10. J. E. Robbins, and et al, "Integrating Architecture Description Languages with a Standard Design Method", Presented at the Second EDCS Cross Cluster Meeting in Austin, Texas. (can be got from http://www.ics.uci.edu/pub/arch/sw-and-pubs.shtml)
11. Rational, "UML Notation Guide", version 1.1, September 1997, http://www.rational.com/UML
12. P. Kruchten, "The 4+1 View Model of Software Architecture", IEEE Software, November 1995.
13. H. Eriksson and M. Penker, "UML Toolkit", Wiley Computer Publishing, John Wiley & Sons, Inc., 1998.

# 7 APPENDIX

**Example of state space analysis report**

```
----------------------------------------------------------------------

Occurrence Graph
  Nodes:  4614
  Arcs:   16732
  Secs:   1795
  Status: Full

Scc Graph
  Nodes:  3912
  Arcs:   11202
  Secs:   35

Boundedness Properties
----------------------------------------------------------------------
Best Integers Bounds                Upper        Lower
CallTrans'ActiveObjects 1           5            0
CallTrans'CallComing 1              1            1
CallTrans'CallCreated 1             2            0
CallTrans'CallMsgForApp 1           5            0
CallTrans'CallMsgForServer 1        2            0
CallTrans'MO_S1_Finished 1          1            0
CallTrans'MsgDelivered 1            3            3
CallTrans'MsgToBeSent 1             5            0
EventIndicate'ActiveObjects 1       5            0
EventIndicate'CallComing 1          1            1
EventIndicate'EventonGoing 1        1            0
EventIndicate'EventsHappend 1       1            0
EventIndicate'EventsSubscribed 1 1  1            0
EventIndicate'MsgDelivered 1        3            3
EventIndicate'MsgToBeSent 1         5            0
Exec'ActiveObjects 1                5            0
Exec'AvailableServers 1             3            3
Exec'EventsHappend 1                1            0
Exec'EventsSubscribed 1             1            0
Exec'MsgDelivered 1                 3            3
Exec'MsgToBeSent 1                  5            0
Exec'WantedServices 1               1            0
MsgTransmit'ActiveObjects 1         5            0
MsgTransmit'AvailableServers1       3            3
MsgTransmit'MsgDelivered 1          3            3
MsgTransmit'MsgToBeSent 1           5            0
        ...

Home Properties
----------------------------------------------------------------------
Home Markings:  [3659]

Liveness Properties
----------------------------------------------------------------------
Dead Markings:  [3659]
Dead Transitions Instances: All
Live Transitions Instances: None
```

# Petri Net analysis of the MASCOT Pool IDA Communication Mechanisms

Mustafa A. Jiffry, King's College, University of London, UK

## Abstract

The paper is concerned with modelling and analysis of the MASCOT Pool IDA communication mechanisms. Both the Four-slot fully asynchronous mechanism and the Two-slot conditionally asynchronous mechanism are investigated. The mechanism properties are defined first, to provide the basis for Petri nets modelling. Place/Transition nets and Coloured Petri nets are used to produce the models. The Design/CPN software tool is used to verify those communication mechanisms. It is shown that the method adopted is an effective way for verification of the correctness of interprocess communication mechanisms used by parallel architectures of distributed real-time systems.

## 1. Introduction

The Four-slot fully asynchronous communication mechanism was first developed by Simpson, and presented publicly in the IEE Proceedings [1]. The mechanism provides a new technique for solving the problem of accurate data transfer between concurrent processes using parallel architectures within distributed real-time data processing systems. Generally the techniques which have been used to provide concurrent access to shared data, were based on the mutual exclusion principle. This new form of fully asynchronous communication mechanism is not based on the mutual exclusion principle, instead it uses co-operative access control. Co-operative access control uses control variables to steer both the reader and writer processes in-order to preserve data integrity. The control variables are used as indices for the mechanism steering strategy, and are not part of any conditional statement. Because there is no existence of any form of wait protocol, the reader and writer processes are fully asynchronous and do not have to wait for each other to maintain data integrity.

## 2. Background

This section contains an informal introduction to MASCOT, the interprocess communication of the Pool IDA, and the Design/CPN software tool. The reader can refer to a more formal presentation in the literature [1,2,3,4,5,6,7,8,9,10].

## 2.1 MASCOT

MASCOT (Modular Approach to Software Construction Operation and Test) is a software design method for real-time systems, based on data flow concepts in which a key feature is the use of special symbols to represent the real time dynamics of concurrent processes

communicating through shared memory. Figure 1 shows the communication mechanism of reference data being represented by a MASCOT pool IDA.

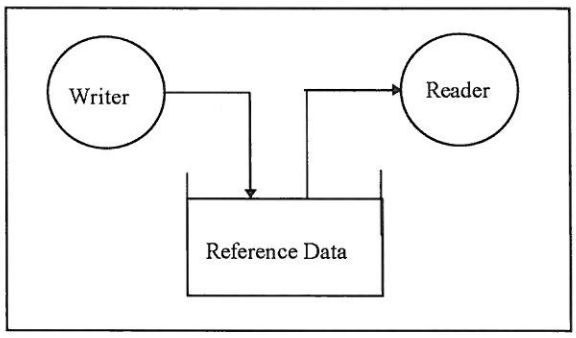



**Figure 1 Shared Pool IDA**

In MASCOT there are two fundamental classes of components:

1. Activity - the data processing component
   An activity is a single sequential program thread that can be independently scheduled. The activities communicate through IDAs. The IDA provides the necessary synchronisation and mutual exclusion facilities. The graphical representation of activities have rounded boundary and are often shown as circles.

2. Intercommunication Data Area (IDA) - the data communication component
   An IDA is a passive element which services the data communication needs of activity components. It can contain its own private data areas, and provides procedures which activities use for the transfer of data. The graphical representation of IDAs are rectangular shapes with special forms for the channel and pool.

MASCOT supports several forms of IDAs which are based on their behaviour :

- Channel IDA
  The channel IDA allows message data to be passed from one process to another. It supports data communication between producers and consumers. It can contain one or more items of information. Writing to a channel adds an item without changing items already in it (a non-destructive write operation). The read operation is destructive, since it removes an item from the channel. A channel can become empty and because its capacity is finite, it can become full.
- Pool IDA
  The pool IDA allows reference data (a table or dictionary) to be passed from one process to another. The reference data is retained within the pool, where it can be consulted at any time by the reader and updated at any time by the writer. The write operation is destructive and the read operation is non-destructive.

The classification of the communication models in MASCOT is based on its synchronous form. The following presents the distinction between reference and message data for a single-writer and a single-reader;

1. Reference data (Pool IDA)
   - Fully asynchronous (Four-slot mechanism).

- Conditionally asynchronous (Two-slot mechanism).

2. Message data (Channel IDA)
   - Loosely synchronous (Bounded buffer).
   - Fully synchronous (Rendezvous).

## 2.2 Interprocess communication of the Pool IDA

The reader and the writer process co-operate dynamically by means of control variables so that concurrent access to any data record never occurs. The control variables are bit type, so their values are always guaranteed to be coherent, and where reading and writing operations can be concurrent and need not interfere with each other. They can be implemented by a single latch (a bistable latch or a D-type flip-flop).

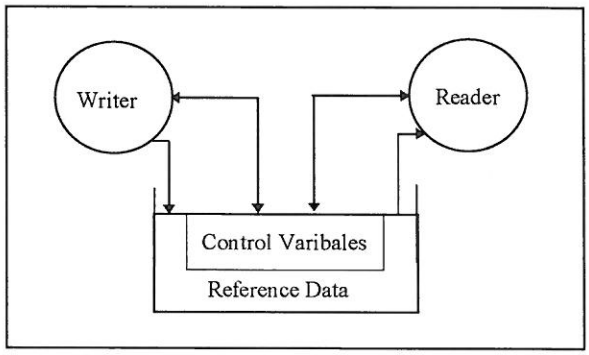



**Figure 2 Co-operative access control**

Therefore, the mechanism design is based entirely on co-operative access control by means of bit control variables (single binary digit of 0 or 1) and these variables are never used in conditional statements which would affect process timing. Figure 2 shows the interprocess communication mechanism which is based on co-operative access control.

## 2.3 The Design/CPN software tool

Petri nets allow a system to be modelled where the resulting Petri net model can be analysed to reveal information about the structure and dynamic behaviour of the modelled system. Coloured Petri nets belong to a class of high-level nets which can model complex systems in a manageable way. The practical use of Coloured Petri nets is highly dependent upon the existence of adequate computer tools in-order to assist the modeller with the following [2] :

- Storing and handling all the details of his CP-net model.
- Obtaining better results (e.g. where the CPN editor provides precision and quality drawing capabilities, and the simulator provides a computer support for complex analysis methods).
- Getting faster results (e.g. where the CPN editor is used to create and modify CP-net models, and the simulator is used to construct occurrence graphs).
- The possibility of making interactive presentation of the analysis results.
- The possibility of hiding technical aspects of the CP-net theory inside the tools.

The Design/CPN is an interactive computer tool to build and execute CP-nets [3,4,5]. It requires X-Windows/Unix, or MacOs (Macintosh) platform. The tool provides the following :

- An editor to create and manipulate CP-nets.
- Syntax checker for validating CP-nets.
- A simulator for executing and debugging CP-nets.
- Facilities for organising a CP-net into a hierarchy of modules.
- Animation and charting facilities for displaying simulation results.

## 3. The Mechanism Properties

The mechanism we are considering involves the case of a single writer communicating with a single reader by using a slot, which is a data area in shared memory capable of holding a single data record in transit. The reader and writer are individually represented by a process, which is an independent thread of execution defined by a series of sequential operations. Control variables are being used to regulate or direct the writing and reading of data. Both processes (reader and writer) can run in an endless loop to perform its dedicated function.

The author is trying to define the mechanism properties which will ensure through logical reasoning that the resulted Petri net model is a correct model. In other words, the mechanism software design and the Petri net model represent the same system. Defining these properties will provide the basis for Petri nets modelling. The main properties of the fully asynchronous communication mechanism have been defined by Simpson [1]. These properties relate to the behaviour of the mechanism and to how the interprocess communication is regulated. Additional properties relating to model representation will be introduced for two purposes. Firstly, to make the Petri net model a true reflection of the software design. Secondly, to help with clarity of presentation of the analysis. The properties list is as follows;

1. **Asynchronism**
   Neither process (reader or writer) may affect the timing of the other as a direct result of its communication operations.
2. **Data coherence**
   Data must always be passed as a coherent set. Interleaved access to any data record by the reader and the writer is not permitted.
3. **Data freshness**
   The latest completed data record produced by the writer must always be made available for use by the reader.
4. **Mutually independent cycles**
   Each process (reader and writer) will be represented by a mutually independent Petri net cyclic model, since each process is expected to run in an endless loop.
5. **Reflecting the software design**
   The model must always reflect the software design forms proposed by Simpson [1]. The software module will consist of global and local control variables, and access algorithms for both the reader and writer processes. The execution of any single statement within each access algorithm will be represented by the occurrence (firing) of a single transition in the corresponding Petri net model. Global control variables will have global scoping, so they will keep their values between cycle execution (analogous to procedure or function calls). They will be represented by a place stating their current bit value (0 or 1). Local control variables will have local scoping, so their values will be set dynamically during Petri nets execution, and maintained throughout the process execution cycle as required. This property

will make sure that the dynamic execution of the Petri net model will be the same as the execution of the software design. If we break point the execution of the Petri net model at any step, we should be able to tell, for each process, what is the current statement to be executed next and what is the current value of each control variable. The results should be identical to those which would be obtained if we were to break point the execution of the software design at the corresponding statement.

6. **Reflecting the way the mechanism is using control variables**
   The control variable is a bit type (single binary digit). It is used to regulate or steer the co-operative processes, in-order to maintain data integrity. The Petri net model must maintain the use of each control variable as intended by the proposed mechanism software design.
7. **Reflecting the concurrent process execution**
   Petri nets execution is naturally concurrent, since the transition firing is never predetermined. When modelling the access of control variables, attempt will be made to ensure that transitions will be concurrently enabled. Therefore, if there exist any two enabled transitions having the same input place, they might be in conflict for the same enabling token. Removing all token conflicts will make all the transitions to be concurrently enabled. Here, we have two situations:
   - **Both processes want to read the same control variable**
     In this situation, the number of tokens will be doubled, so each transition will have its own set of enabling tokens. Therefore, we are matching the hardware design, since reading a latch is done on the rising or falling clock signal edge.
   - **One process is reading and the other is updating the same control variable**
     In this case, the hardware design can accommodate writing and reading of the bit variable, since the latch will lock its input (write value) on the rising edge, and deliver its output on the falling edge of the clock signal (or visa versa). Therefore, the control variable bit value will be written on the rising edge and read on the falling edge of the clock signal (or visa versa). The Petri nets execution will simulate this situation, since the firing of either transitions is never predetermined. Let us consider the following scenario:
     *Tw2*, *Tr0* are two enabled transitions which require the same enabling token from the same input place (see Figure 8). The execution sequence of *Tw2*, *Tr0* results in writing the control variable followed by reading it. But the execution sequence of *Tr0*, *Tw2* results in reading the control variable followed by writing it. Both of these sequences are permitted by the fair execution case of Design/CPN.
8. **Metastability**
   The phenomenon of metastability occurs whenever a shared control variable is read close to the time it is being changed. However, the execution of the Petri net model does not allow it to happen because the two transitions (see Figure 8, where *Tw2* writes and *Tr0* reads the same shared control variable *latest*) will not fire at the same time, since they will be in conflict (requiring the same enabling token *latest*). In the case of the hardware bistable latch, the clock signal pulse width (the time duration between the rising and falling edges) can be used to prevent metastability. Therefore, the shared control variables are always stable and having a valid state of 0 or 1.
9. **Detecting error condition**
   Detecting data coherence errors should not load the Petri net model by the creation of conflicts between transitions requiring the same enabling token.
10. **Naming convention for places and transitions**
    For Place/Transition nets, the author has used a naming convention which reflects the software design, by assigning the software design statements as names to transitions and the control variables values as names to places. Although this has resulted in duplicate names (which is not allowed by Petri nets), it is however intended to give a clear description of the

use of those transitions and places. Of course, for Coloured Petri nets, unique descriptive names are used in accordance with the Petri nets rule.

## 4. The One-Slot Mechanism

This mechanism contains one possible place for data in transit. This is a null form of co-operative access control. There is no need for control variables to regulate or steer data access, since there exists only one slot to store the data in transit.

```
mechanism one-slot;
  var data : TypeData :=null;

  procedure write(item : TypeData);
    begin  data := item;  end;

  function reader : TypeData;
    begin  read := data;  end;

end.
```

**Figure 3 The One-slot mechanism (in a Pascal like language)**

The software design of the One-slot mechanism as in Figure 3, defines a single slot (to hold the data) and two access algorithms (write and read). The single slot is initialised to null in-order to ensure data coherence when the reader accesses the mechanism before the first write. Integrity of the One-slot mechanism is only preserved if write and read never overlap by scheduling the data access.

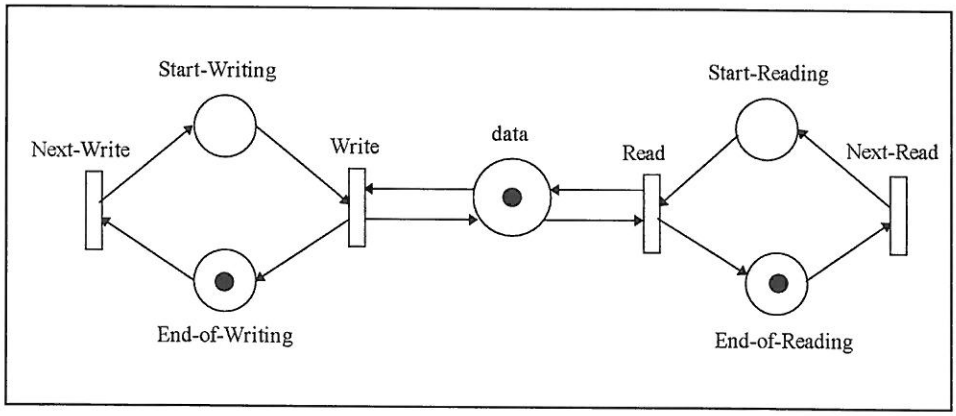



**Figure 4 Place/Transition net model for the One-slot mechanism**

Figure 4 shows the Place/Transition net model for the One-slot mechanism. The Coloured Petri net model is similar, since the tokens within the Coloured Petri net model would carry no information except for their presence or absence.

## 5. The Two-slot Communication Mechanism

The software design of the Two-slot mechanism, is shown in Figure 5. This design consists of the following;

1. A two slots array *data[bit]* to hold information in transit, and it is pre-set to *null*, in-order to ensure that a read before the first write will obtain *null* data value.
2. A control variable *latest*, which indicates the latest data written. It is pre-set to zero.
3. The write algorithm selects alternate slots for writing, and at the end of each write it indicates the latest data.
4. The read algorithm always starts to read data from the last completely written slot.

```
mechanism two-slot;
  type bit = 0..1;
  var  data : array[bit] of TypeData := (null, null);
       latest : bit := 0;

  procedure write(item : TypeData);
    var wx : bit;
    begin
      wx := NOT latest;
      data[wx] := item;
       latest := wx;
    end;

  function read : TypeData;
    var rx : bit;
     begin
        rx := latest;
        read := data[rx];
      end;

end.
```

**Figure 5 The Two-slot algorithm (in Pascal like language)**

The author will be using Place/Transition nets and Coloured Petri nets to model the Two-slot mechanism. The properties established earlier would be preserved, so that the Petri net model can reflect the mechanism behaviour under investigation. Each mechanism cycle (reading and writing) would be modelled separately, in-order to reduce the complexity and provide a simple model to present each cycle on its own. Combining both into a single diagram is a trivial step. Also, when the author combines both cycles into a single Coloured Petri net model, the folding power of Coloured Petri net will be demonstrated.

## 5.1 Place/Transition net model of the Two-slot Mechanism

Figure 6 shows the PT-net model of the read access algorithm. *Start-Reading* is the place which establishes the entry point of the read access function. The initial marking represents the initialisation values defined by the software design as indicated by Figure 5. The

transition *rx=latest* is the first executable statement, and *Read=data[rx]* is the second executable statement. The places *latest=0* and *latest=1* represent the global control variable latest having a value of 0 or 1. When a token is present at the place *latest=0*, it will mean that, the control variable latest is set to have a zero (0) bit value. Since all the properties set out earlier are being encapsulated within this PT-net model, the model represents the same reading access algorithm defined by the software design form of the Two-slot mechanism.

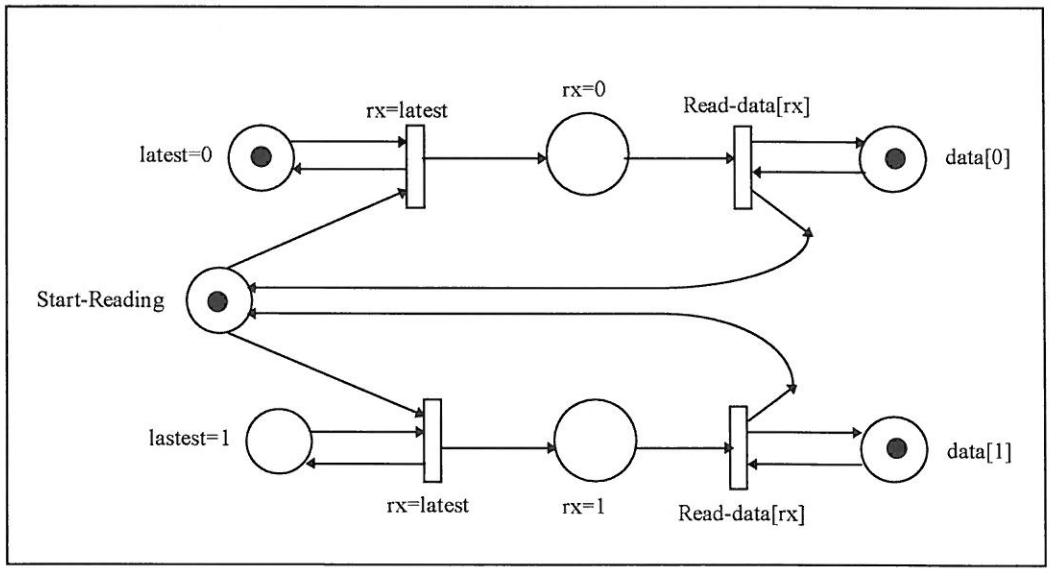



**Figure 6 PT-net model for the reading cycle of the Two-slot mechanism**

The model given in Figure 6 reveals how the mechanism keeps the reader process from jumping to the other slot. The reader does not have the ability to switch over until the writer indicates the presence of a new fresh data by changing the bit value of the control variable *latest*. This control variable steers the reader to the latest completely written data.

Figure 7 shows the PT-net of the write access algorithm. The write access procedure consists of an entry point and three executable statements. The place *Start-Writing* marks the entry point, and the transitions *wx=Not(latest)*, *Write-data[wx]*, and *Latest=wx* represent the three executable statements. The initial marking represents the initialisation values of the mechanism software design. The mechanism uses the control variable *latest* to point to the slot containing the freshest data. Therefore, the PT-net model of the write access algorithm must reflect the use of the control variable *latest* (as stated in the previously defined properties list). The transition *latest=wx* will exchange the bit value of this control variable from 0 to 1, by removing the token from place *latest=0* and depositing it in place *latest=1*, and visa versa. Since the PT-net model guarantees encapsulation of the previously defined properties, we can claim that this is a correct model.

It can be seen in Figure 7 how the writer jumps between slots, regardless whether the reader is accessing the slot or not. The writer does not have any knowledge of whether the data slot to be used next, is free and is not used by the reader. Therefore, if the writer process is faster than the reader process, a data coherence violation will arise. Simpson refers to this mechanism as the swung buffer [1], since the writer process writes alternate data items to alternate slots which are then swung into visibility (by the indication of the *latest* control variable bit value) for use by the reader.

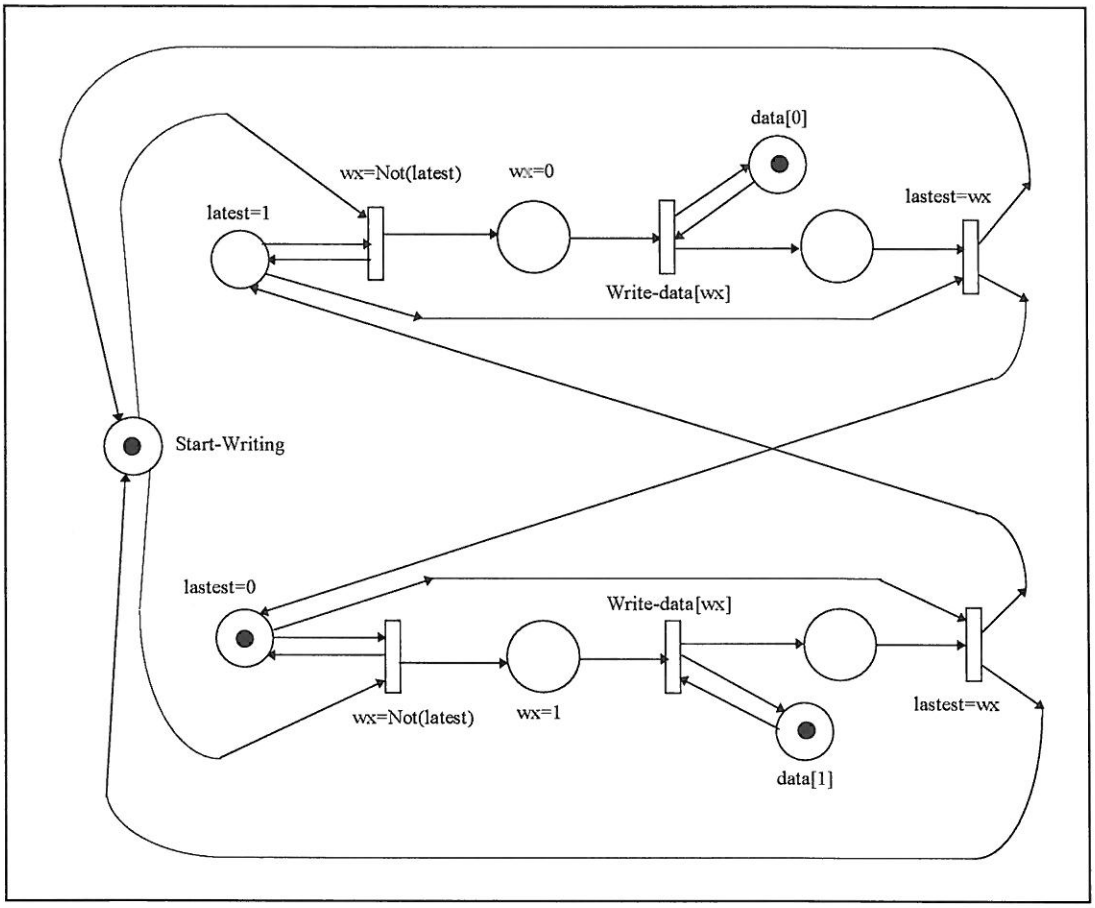



**Figure 7 PT-net model for the writing cycle of the Two-slot mechanism**

## 5.2 Coloured Petri net model of the Two-slot Mechanism

Coloured Petri net is a powerful net type, which can describe complex systems in a manageable way. In the PT-net model, we had to represent the mechanism state based on the control variable bit value, with two identical separate subnets. But now a more compact representation can be achieved, by folding the two read access function subnets into a single subnet, and the same can be done with the two subnets of the write access procedure. This is only a one level folding, and we can see the reduction of complexity by using a high level net. The original PT-net model (for both processes) consists of twelve (12) places and ten (10) transitions, whereas the equivalent CP-net model consists of seven (7) places and five (5) transitions.

The folding power of Coloured Petri net is quite a temptation for the modeller to use in-order to reduce the Petri net model size, but it can lead to complication in presentation of the analysis. It is possible to fold the Petri net model further by two levels: first by combining both processes (reader and writer) and second by combining the mechanism progression steps. If the described folding is done, there will be a violation of the previously defined properties list which would effect the clarity of presentation of the analysis. The author is aiming for a simple and precise Petri net model, in-order to reveal all the information that can be obtained towards the understanding of the mechanism behaviour.

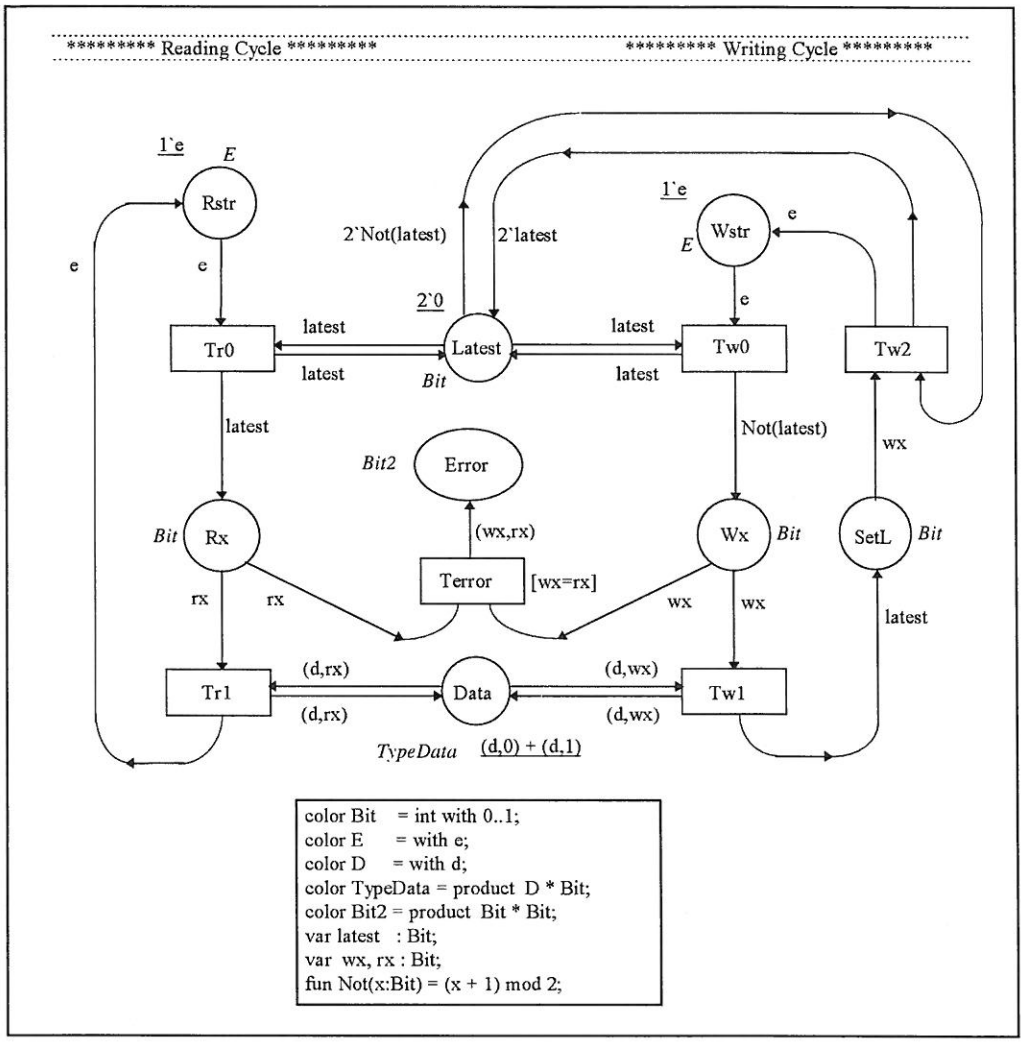



**Figure 8 CP-net model for the Two-slot mechanism with error detection**

The CP-net, shown in Figure 8, resulting from the one level folding of the reader and writer PT-net models, is the complete Coloured Petri net model of the Two-slot mechanism. The property of reflecting the concurrent execution of both processes is preserved by setting the *Latest* place initial marking with two tokens (2`0), in-order to concurrently enable both transitions *Tr0* (represents *rx=latest* ) and *Tw0* (represents *wx=Not(latest)* ).

In this study, it is possible to identify locations in the CP-net model were metastability can occur. In Figure 8, for example, the simultaneous firing of *Tw2* and *Tr0* will result in metastability. However, *Tw2* and *Tr0* can not fire at the same time because the CP-net execution does not allow them since they are in conflict for the same enabling token *latest*.

## 5.3 Verification of the Two-slot Communication Mechanism

The Two-slot mechanism is not a fully asynchronous communication mechanism. The author needs to detect error conditions which may take place. The only property violation this mechanism may commit is that of data coherence, since data freshness is accounted for. Data coherence error arises when both processes (writer and reader) try to access the same data slot.

Design/CPN is used to verify and prove the occurrence of violation of data coherence. An error detection facility must be added to satisfy the properties list defined earlier.

Figure 8 shows the proposed error detection facility. The CP-net model will halt its execution, when the state of data coherence violation arises. There is no loading on the original CP-net model, if it executes without violating the data integrity property. When a data coherence violation takes place, then three transitions (*Terror*, *Twl*, *Trl*) will be in conflict by requiring the same enabling token. The guard ( *[wx=rx]* ) associated with the *Terror* transition makes sure that the occurrence of the error condition is the only way that will lead to enabling the *Terror* transition. When the error condition is enabled, it will mean that the three transitions (*Terror*, *Twl*, *Trl*) are enabled, and either *Terror* will fire or *Twl* and or *Trl* will fire in accordance with the Petri nets rule. But we are seeking a full occurrence graph (state space), which means we are interested in the possibility of *Terror* to occur or not.

Since we now have the complete CP-net model with error detection, we can proceed with the Design/CPN tool to generate the occurrence graph, which is the CP-net state space (some times it is referred to as reachability tree) that could be presented graphically.

```
Statistics
------------------------------------------------------------------------

 Occurrence Graph
   Nodes: 20
   Arcs:  38
   Secs:  0
   Status: Full


 Boundedness Properties
------------------------------------------------------------------------

 Best Upper Multi-set Bounds
 NewData 1      1`(d,0) + 1`(d,1)
 NewLatest 1          2`0+ 2`1
 NewError 1       1`(0,0) + 1`(1,1)
 NewRstr 1          1`e
 NewRx 1          1`0+ 1`1
 NewSetL 1      1`0+ 1`1
 NewWx 1        1`0+ 1`1
 NewWstr 1          1`e

 Best Lower Multi-set Bounds
 NewData 1      1`(d,0) + 1`(d,1)
 NewLatest 1          empty
 NewError 1       empty
 NewRstr 1          empty
 NewRx 1          empty
 NewSetL 1      empty
 NewWx 1        empty
 NewWstr 1          empty


 Liveness Properties
------------------------------------------------------------------------
 Dead Markings:  [15,20]
 Dead Transitions Instances: None
```

**Figure 9 Design/CPN (partial) statistical report for the Two-slot**

Figure 9 presents a partial statistical report generated by using Design/CPN to run the CP-net model of the Two-slot mechanism. The occurrence graph generated is full and small in size (20 nodes in total). Full state space of the CP-net model is required for verifying the mechanism, in-order to identify all the possible states the mechanism can reach. Data coherence error is reachable, by checking the liveness and boundedness properties of the CP-net model. The liveness property indicates that the CP-net model can have two states of dead markings, they are state number 15 and 20. These states (15 and 20) can be expanded by Design/CPN to show the marking of all the places of the CP-net model. By using the best upper multi-set bounds of the boundedness property, we can see that the place *Error* can have an upper bound marking of *1`(0,0)+1`(1,1)*. This means, data coherence condition is reachable when both processes, the reader and writer, are either accessing *data[0]* or *data[1]*, since we have both cases of *wx=rx=0* and *wx=rx=1*. Because the occurrence graph is small in size, an attempt is made to use Design/CPN to draw it (as seen in Figure 10).

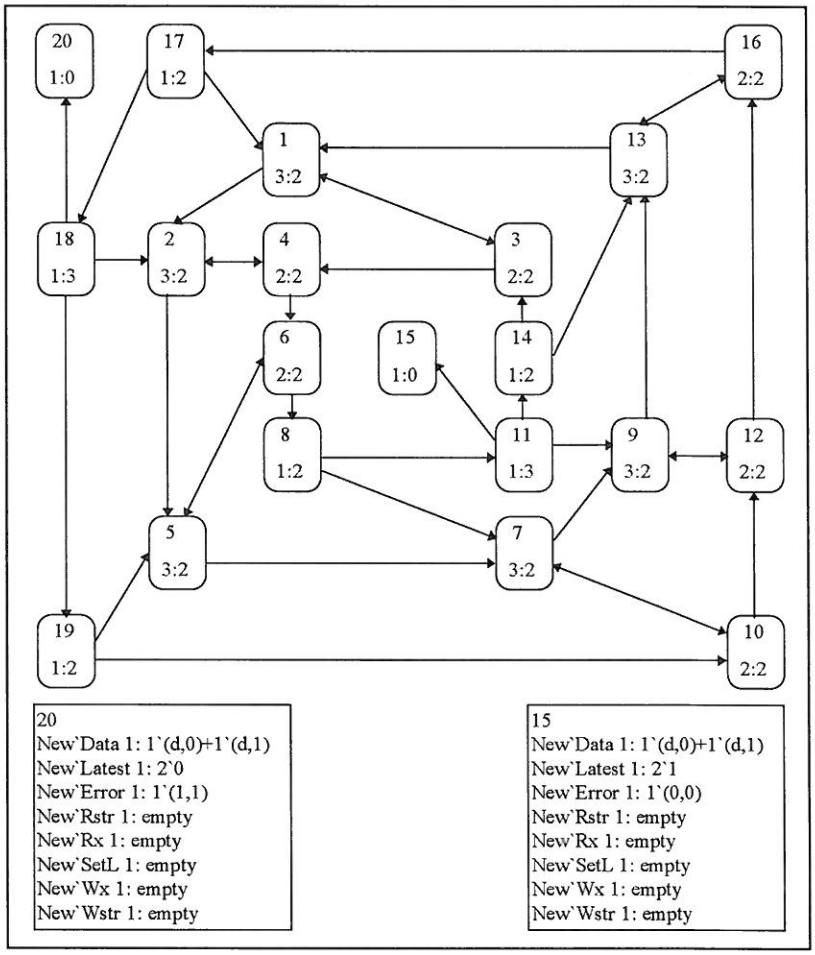



**Figure 10 Occurrence graph of the Two-slot mechanism with error detection**

Data coherence error is violated by the writer process. When it executes faster than the reader, it wants to write a new data record, at the same time, the reader did not finish or about to start reading the old data record. This situation is highlighted by the token values of both violation states (15 and 20), the value of the control variable *latest* is changed by the writer to have the same value as the reader was using to enter its reading cycle. For example, let us investigate the state number 20. The reader starts its cycle by accessing the control variable as *latest=1*, the write just finished writing its data record, and proceeds to exchange the bit value of the

control variable *latest* to be *latest=0*. Then it will try to write a new data record, but the reader is not through yet. This scenario can be see from the token value of the place *Latest* (token of 2`0 means *latest*=0) and the token value of the place *Error* (token of 1`(1,1) means *wx=rx*=1).

There is a good way to cross check the validity of the CP-net model proposed, by using the boundedness property of CP-net. As stated previously in the properties list, the CP-net model must reflect the software design. We can see that the upper multi-set bounds of all the places can never exceed the maximum number of token limits. In other words, the place *Latest* can either have *2`0* or *2`1* and the place *Rx* can either have *1`0* or *1`1*. Therefore, the CP-net is precisely reflecting the control variables use.

Finally, we conclude that the Two-slot communication mechanism is conditional asynchronous, the condition would be to assure the speed of accessing and utilising the data record by the reader to be equal or faster than the writer. Let us say the time it takes the reader to access the data record is $R_{access}$ and to utilise it is $R_{between}$ (between access), and similar for the writer. Also, we have the two functions to return the maximum and minimum time of these accessing and utilising operations. Then the condition would be :

$$Min(W_{access}) + Min(W_{between}) \quad >= \quad Max(R_{access}) + Max(R_{between})$$

## 6. The Four-slot Communication Mechanism

```
mechanism four-slot;
  type bit = 0..1;
  var  data : array[bit,bit] of TypeData := ( (null,null), (null,null) );
       s : array[bit] of bit := (0, 0);
       latest, r : bit := 0, 0;

  procedure write(item : TypeData);
    var wp, wx : bit;
    begin
      wp := NOT r;
      wx := NOT s[wp];
      data[wp,wx] := item;
       s[wp] := wx;
       latest := wp;
    end;

  function reader : TypeData;
    var rp, rx : bit;
     begin
        rp := latest;
        r := rp;
        rx := s[rp];
        read := data[rp,rx];
      end;

end.
```

**Figure 11 The Four-slot algorithm (in Pascal like language)**

Figure 11 shows the Four-slot software design. The mechanism consists of four data slots, control variables, and two access algorithms.
The Four-slot software design can be described as follows;

1. A four slots array *data[bit, bit]* to hold data in transit, organised as two pairs of two slots. It is pre-set to *null*, in-order to ensure that a read before the first write will obtain the *null* value.
2. A control variable array *s[bit]* of two elements. Updated by the writer to indicate the index of the slot which contains the last written data, it is pre-set to zero.
3. A control variable *latest* to indicate the last written slot pair, it is pre-set to zero.
4. A control variable *r* to indicate the pair about to be, being, or last read. It is pre-set to zero.
5. Write access algorithm selects the pair of slots that are not being, or to be read. Then, it selects alternate slots within the previously selected pair. Finally it declares the position of the latest written data.
6. Read access algorithm selects the pair of slots that contain the freshest declared data, then it reads the last written data within that pair.

The methodology will follow the same pattern already established, where the author will be using Place/Transition nets, then Coloured Petri nets to model the Four-slot mechanism. The properties established earlier would be preserved, in-order for the Petri net model to reflect the mechanism behaviour under investigation. Each mechanism cycle (reading and writing) will be modelled separately.

## 6.1 Place/Transition net model of the Four-slot Mechanism

The Place/Transition net model of the reading cycle of the Four-slot is presented in Figure 12. The place *Start-Reading* marks its entry point, and the initial marking is based on the initialisation values defined by the software design. You can see the equivalence between the software design statement execution and the transition firing of the PT-net model.

This algorithm uses the control variable *latest* to steer it towards the slot pair which holds the freshest data. The use of the control variable *r* is more complicated, since the reader process is responsible for updating its value. In-order to reflect this use as required by the previously defined properties list, we need to set the correct bit value for the *r* control variable. When the place *latest=0* has a token, it implies that the control variable bit value is zero (0), and the same goes with the place *r=0*. If the software design statement *r=rp* executes (see Figure 11), then the PT-net model must set the bit value of *r* to be the same as *rp*. This means, if *rp=0 and r=0* the bit value of *r* does not change. On the other hand, if *rp=0 and r=1* the bit value of *r* must be exchanged with *r=0*. This way the PT-net model is reflecting the use of the control variables.

Figure 13 shows the PT-net model of the Four-slot write access algorithm, where the place *Start-Writing* sets the entry point. The mechanism use of the control variables is more complicated here. The control array *s* is used to point to the latest written slot within the corresponding pairs. This means *s[0]* points to the pair *data[0,0]* and *data[0,1]*, but the value of *s[0]* points to the last slot used within those two pairs. The PT-net model is reflecting this use by exchanging the value of *s[0]=0* and *s[0]=1*. The use of the *latest* control variable is reflected by either keeping it unchanged when *wp=latest*, and exchanging it when *wp<>latest*.

Therefore, all the properties defined previously are being encapsulated within these Petri net models, which makes them correct models for the reader and writer processes.

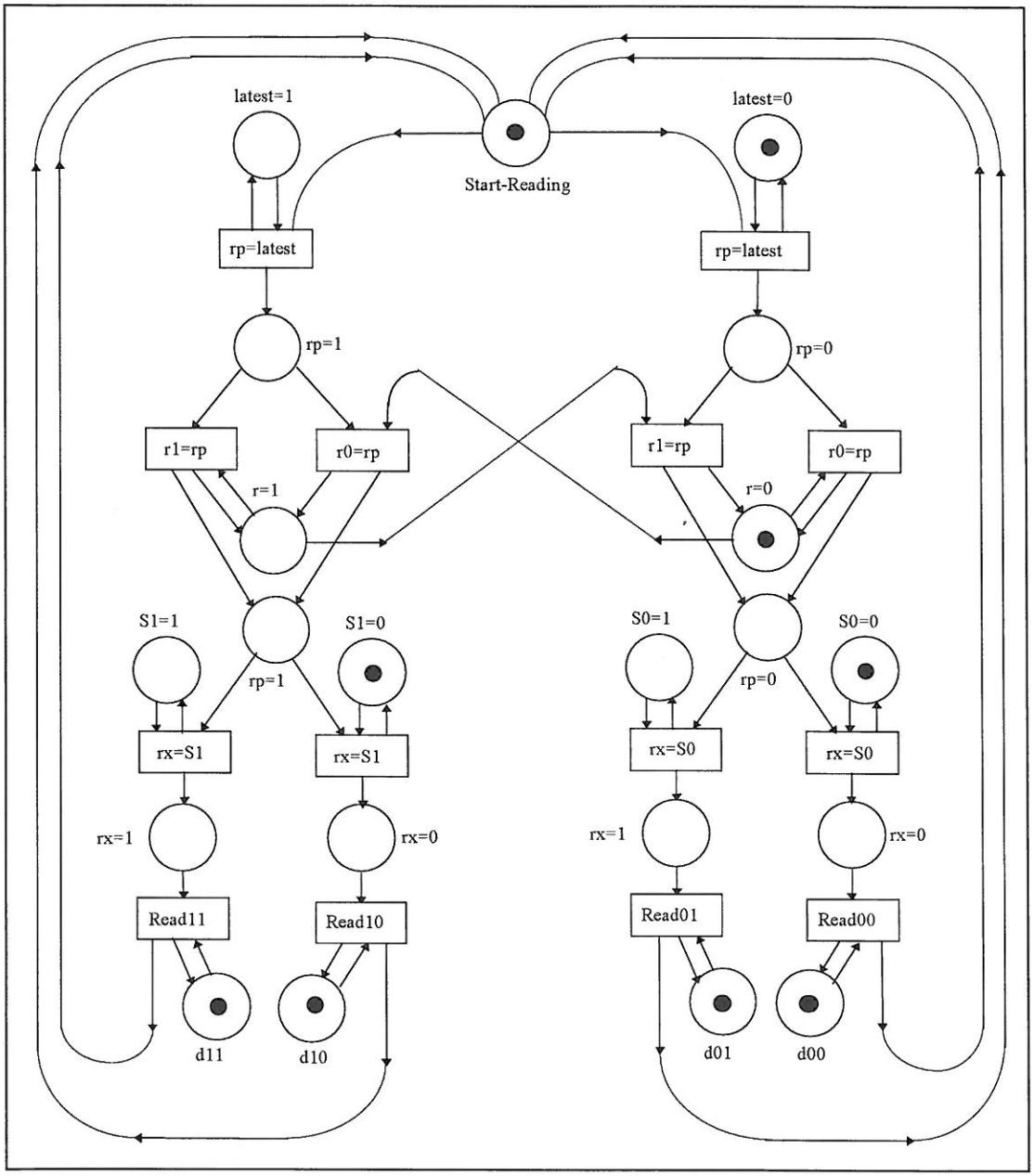



**Figure 12 PT-net model for the reading cycle of the Four-slot mechanism**

In Figure 12 the reading cycle steering strategy is as follows;

1. The writer steers the reader to the pair having the latest completely written data by means of the control variable *latest*. The selected pair becomes the reader currently used pair.
2. The reader steers the writer away from the pair it is currently using, by means of the control variable *r*.
3. The writer steers the reader to the slot within the reader currently used pair which has the freshest data, by means of the control variable *s[latest]*.
4. The reader access the freshest data slot, and re-starts its reading cycle

The reader will keep reading the same slot (within the currently selected pair), until the writer steers the reader to a pair having the new freshest data.

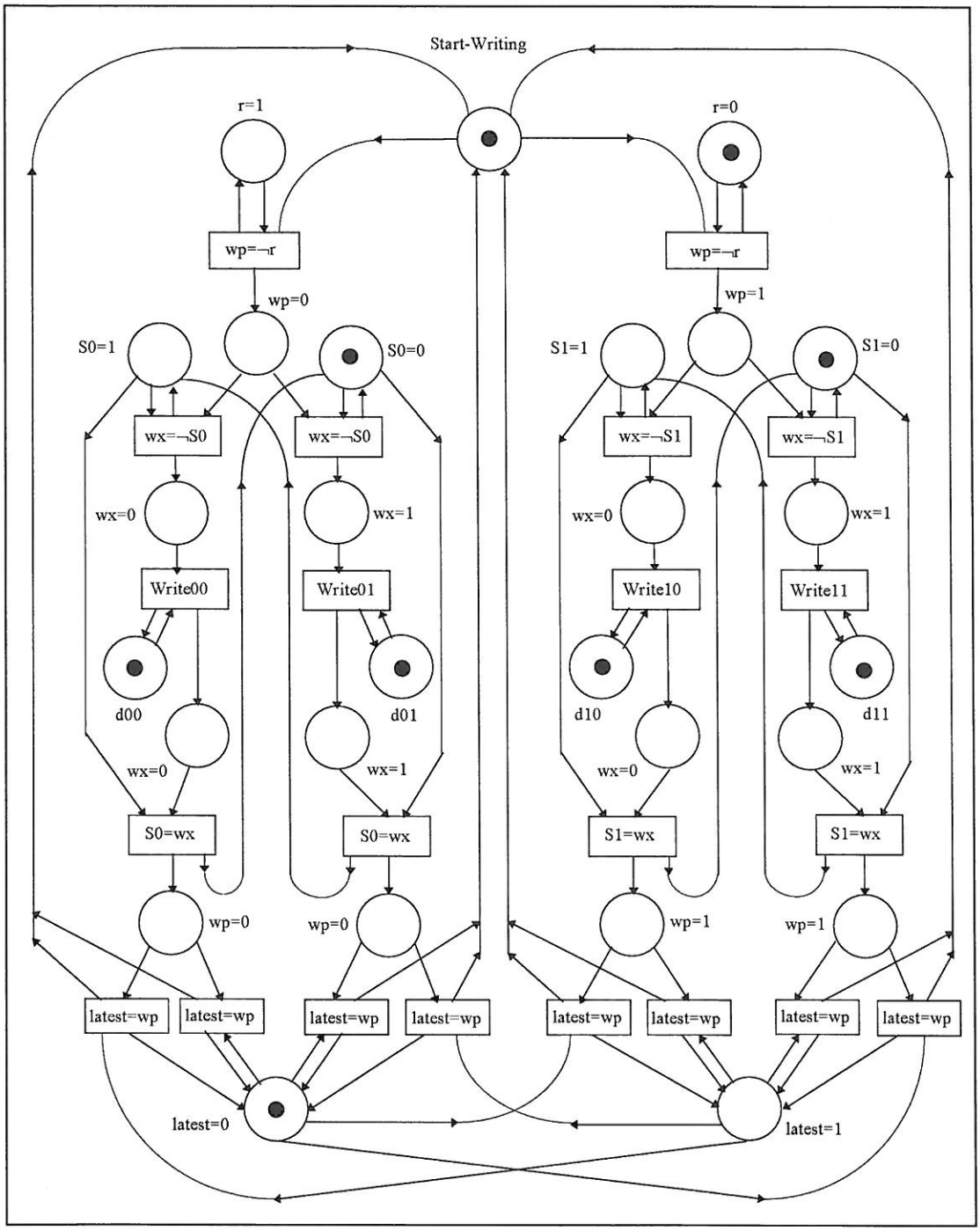



**Figure 13 PT-net model for the writing cycle of the Four-slot mechanism**

In Figure 13 the writing cycle steering strategy is as follows;

1. The writer avoids the reader currently used pair and chooses the free pair.
2. The writer chooses the slot having the oldest data within the free pair to write its new data.
3. When the writer has completely written its data, it indicates the new freshest data location:
   - The slot which holds the new freshest data (by the control variables *s[latest]*).
   - The pair which holds the new freshest data (by the control variable *latest*).

   Afterwards, the writer re-starts its writing cycle.

The Two-slot mechanism is a one way steering strategy (where the reader keeps chasing the freshest data), but it can be seen from the previous description that the Four-slot mechanism is a two way steering strategy. The writer steers the reader towards the freshest slot, and the reader steers the writer away from the pair it is using. In other words, the reader keeps chasing the freshest data and the writer keeps avoiding the pair used by the reader.

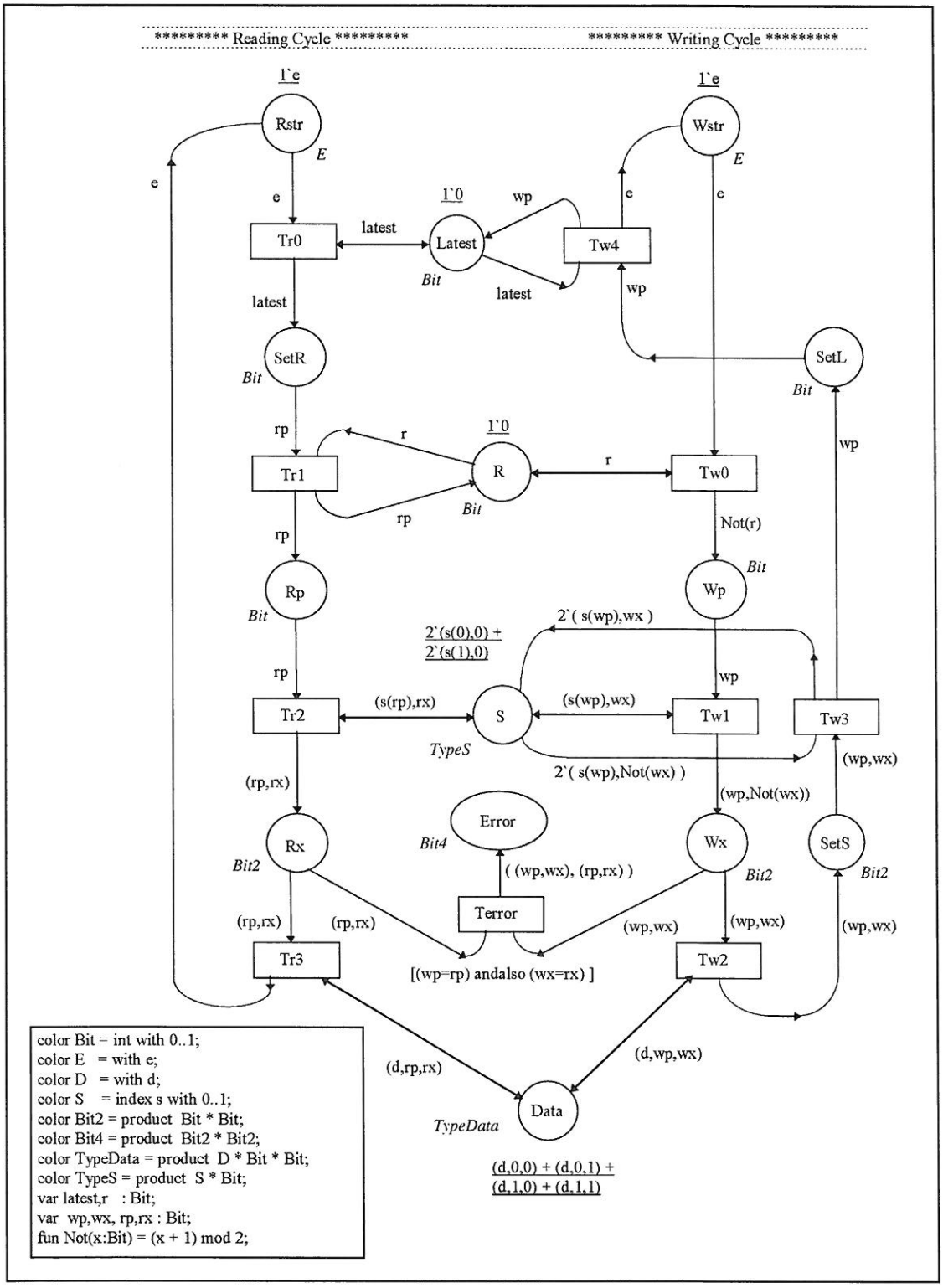



**Figure 14 CP-net model of the Four-slot mechanism with error detection**

## 6.2 Coloured Petri net model of the Four-slot Mechanism

The Four-slot mechanism could be constructed directly from the software design, but the author is trying to stress on the power of CP-net modelling and how it simplifies the resulting model into a manageable one. It also provides a good vehicle for verification.

The PT-net model has a big advantage in presenting the analysis, since it reveals all the hidden information about the mechanism behaviour. The CP-net model does not clearly describe the underlying steering strategy used by the mechanism. Therefore, both modelling techniques (Place/Transition nets and Coloured Petri nets) have their own advantages.

There exist two levels of folding within each access algorithm. The first level is the outer subnet, and the second level is the inner subnet. Figure 14 presents the resulting CP-net model.

## 6.3 Verification of the Four-slot Communication Mechanism

As shown in Figure 14, the proposed error detection facility will halt the execution of the CP-net model, when data coherence violation is committed.

Figure 15 shows the Design/CPN statistical report of the Four-slot mechanism. The occurrence graph generated is full, but unfortunately it is big in size (totals to 576 nodes), so no attempt will be made to draw it. Full state space is required to verify the mechanism data coherence property. By checking both the liveness and boundedness properties of the CP-net model, we can conclude that data coherence error can never occur. The liveness property indicates that the CP-net has no dead marking, so data coherence error can never be reached. Also, the best upper multi-set bounds of the boundedness property, shows that the place *Error* has a maximum marking value to be *empty*. The dead transitions instances shows that the transition *Terror* can never occur. All of these are proving that the data coherence property can never be violated.

Finally, we conclude that the Four-slot communication mechanism is fully asynchronous, since both processes, the reader and writer, do not have to wait for each other or to restrict their timing to access and utilise the data in-order to maintain data integrity.

```
Statistics
------------------------------------------------------------------------

  Occurrence Graph
    Nodes:  576
    Arcs:   1152
    Secs:   5
    Status: Full

  Boundedness Properties
------------------------------------------------------------------------

  Best Upper Multi-set Bounds
  New'Data 1        1`(d,0,0) + 1`(d,0,1) + 1`(d,1,0) + 1`(d,1,1)
  New'Error 1       empty
  New'Latest 1      1`0+ 1`1
  New'R 1           1`0+ 1`1
  New'Rp 1          1`0+ 1`1
  New'Rstr 1        1`e
  New'Rx 1          1`(0,0) + 1`(0,1) + 1`(1,0) + 1`(1,1)
  New'S 1           2`(s(0),0) + 2`(s(0),1) + 2`(s(1),0) + 2`(s(1),1)
  New'SetL 1        1`0+ 1`1
  New'SetR 1        1`0+ 1`1
  New'SetS 1        1`(0,0) + 1`(0,1) + 1`(1,0) + 1`(1,1)
  New'Wp 1          1`0+ 1`1
  New'Wstr 1        1`e
  New'Wx 1          1`(0,0) + 1`(0,1) + 1`(1,0) + 1`(1,1)

  Best Lower Multi-set Bounds
  New'Data 1        1`(d,0,0) + 1`(d,0,1) + 1`(d,1,0) + 1`(d,1,1)
  New'Error 1       empty
  New'Latest 1      empty
  New'R 1           empty
  New'Rp 1          empty
  New'Rstr 1        empty
  New'Rx 1          empty
  New'S 1           empty
  New'SetL 1        empty
  New'SetR 1        empty
  New'SetS 1        empty
  New'Wp 1          empty
  New'Wstr 1        empty
  New'Wx 1          empty

  Liveness Properties
------------------------------------------------------------------------

  Dead Markings: None
  Dead Transitions Instances:

  New'Terror 1
```

**Figure 15 Design/CPN (partial) statistical report for the Four-slot**

# 7. Conclusion

This paper presents a novel Petri nets modelling method for verifying the correctness of interprocess communication mechanisms of the MASCOT Pool IDA. Also, the modelling method provides a clear description of the mechanism behaviour. The Two-slot and Four-slot communication mechanisms have been analysed and verified using Place/Transition nets, Coloured Petri nets, and Design/CPN. This presents a formal correctness verification of both the Two-slot and Four-slot communication mechanisms used by the MASCOT Pool IDA.

The method developed can be used as a tool for analysis of real-time systems such as MASCOT networks. Fundamental to the method is the definition of the properties which provide the basis for Petri nets modelling. Coloured Petri nets and the support of Design/CPN make up a very powerful modelling and analysis tools. The Coloured Petri nets dynamic nature can be used to follow the path of execution of a real-time system, without the cost and effort of building a test system. Future work will aim at establishing the properties of a simple real-time system represented by a MASCOT network, and then utilising Coloured Petri nets with the support of Design/CPN to analyse the system.

## 8. References


1. Simpson, H., 'Four-slot fully asynchronous communication mechanism', IEE proceedings on Computers and Digital Techniques, Jan 1990, Vol 137 No 1, pp. 17-30.

2. Jensen, K., 'Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use-Volume 1', Second Edition, Springer, 1996.

3. Meta Software Corporation, 'Design/CPN Tutorial for X-Windows - Version 2.0', Cambridge MA, USA, 1993.

4. Meta Software Corporation, 'Design/CPN Reference Manual for X-Windows - Version 2.0', Cambridge MA, USA, 1993.

5. University of Aarhus, 'Design/CPN Occurrence Graph Manual Version 3.0', Aarhus, Denmark, 1996.

6. British Aerospace Dynamics, 'The Official Handbook of MASCOT (Version 3.1)', Royal Signals and Radar Establishment , June 1987.

7. Simpson, H., 'The MASCOT Method', Software Engineering Journal, May 1986, pp.103-120.

8. Simpson, H., 'A Data Interaction Architecture (DIA) for real time embedded multiprocessor systems', Proceedings of RAe Conference on Computing Techniques in Guided Flight, April 1990.

9. Bennett, S., 'Real-Time Computer Control - An introduction - Second Edition', Prentice Hall, 1994.

10. Peterson, J.L., 'Petri net theory and the modelling of systems', Prentice Hall,1981.

# A CPN Model of an Internet Object Cache

B. Kolics and K. M. Hangos
Computer and Automation Research Institute Hung. Acad. Sci.
H-1518 Budapest P.O. Box 62, Hungary
E-mail: Bertold.Kolics@sztaki.hu, hangos@scl.sztaki.hu

**Abstract**

A request-level timed stochastic CPN model of a leaf Internet object cache is proposed in this paper. The leaf cache is modelled as a discrete time discrete event dynamic system together with its adjustable input-, measurable output, disturbance and state variables.

The CPN model has been verified and validated against real measured data. The developed model will be used form model-based optimal selection of the most influental tuning parameters modelled as input variables on the performance of the cache.

## 1 Introduction

Internet object caching is a widely used technique in the current Internet environment. The primary function of object caches is to store popular objects (originating from Web, FTP, Gopher servers in general) and in this way they reduce repetitious accesses to the same resource. As a result, both network bandwidth is saved and users perceived latency is reduced.

Several object caching software products exist which provide a rich set of configuration settings, so cache administrators can easily configure the software products. However, the initial configuration settings may not reflect the given state of network, operating system and other cache influencing parameters, i.e. these settings represent rather a static than a dynamic model. Some articles has already claimed that adaptive approach outperforms other, non-dynamic cache models [1],[2].

Therefore the aim of the project was to develop a *request-level CPN model* of a leaf cache, to develop methods for its verification and validation against

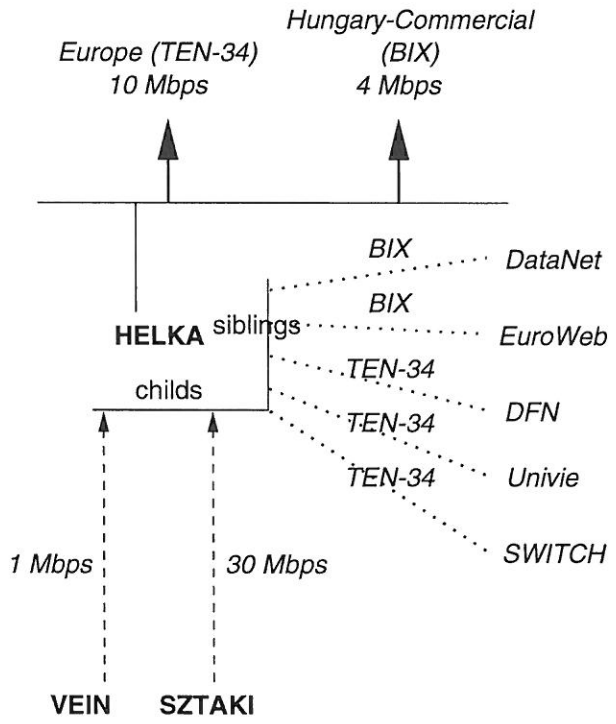



Figure 1: Cache hierarchy used for modelling a leaf cache

measured data in order to be able to investigate the effect of the most influential tuning parameters on the performance of the cache.

The paper is organized as follows. First the modelled system, the leaf cache is described as a dynamic system together with its input, output, state and disturbance variables. Thereafter the CPN model of the cache is developed starting from the modelling assumptions with all of its elements: hierarchy pages, places and transitions. Thereafter the methods and results of the model verification follows and some simulation results are shown. Finally conclusions are drawn.

# 2 The Internet Object Cache

## 2.1 System description

A simplified dynamic model of a leaf cache in a cache hierarchy is developed. The cache server has a parent on a higher level. The cache hierarchy with the place of the investigated leaf cache server is shown on Fig. 2.1.

The cache accepts requests from the clients and delivers the requested

objects to them either from its cached objects or downloaded either from the hierarchy or from the original source.

If the requested object is not present in the leaf cache then it sends a query to the hierarchy, in this case to its parent, to deliver it. It is assumed that after a given time-out the cache is able to download the requested object itself from the original source.

## 2.2 The modelling goal

The aim of our modelling is to find ways to tune the most influencing cache tuning parameters in a model-based and adaptive way. Therefore the model includes:

- the cache tuning parameters which influence the performance of web caches primarily,
- the relevant set of state variables which describes the whole caching process,
- the change of relevant internal cache variables over time,
- the most important measurable variables characterising the operation and performance of the cache.

Because of the above modelling goal a *request level mechanistic dynamic model* is needed which is able to describe the non-linear dynamic behaviour of the cache over a wide operation region.

## 2.3 Dynamic (time varying) variables

The performance of a proxy cache is characterised by the *hit-rate* and by the *response time* of the individual requests. Moreover, the quality of service based on cached WWW-objects is also important where the ratio of stale (i.e. *not up-to-date responses*) is of importance. The above variables can be regarded as *measurable output variables* of the proxy cache as a dynamic system and will naturally be the subject of optimisation or control.

The major dynamic variables influencing the behaviour of a proxy cache are that of the characteristics of the requests. The *number of requests per hour* is a time-varying and random variable which can be regarded as a *disturbance* to the dynamic system. Its average value characterises the *load of the cache*. However, the quality of the requests (the "difficulty" of the requests) also plays role but it is rather difficult to associate a measurable variable (or variables) to it.

The set of *state variables* of the proxy cache system consists of the variables which characterise the actual status of the cache comprising its "past" into their value. The actual status of the resources of the proxy cache computer relevant to caching play role here, such as *the size and the content of the cache memory and disc space, the content and organisation of the object disc directories* etc. The state variables in our CPN model reflect the current state of the caching system using a reduced set of cache-related variables and parameters.

Finally, the potential *input variables*, i.e. the measurable variables which can be directly influenced, are surely among the 100 tuning knobs available for tuning proxy caches. ¿From this set the most influencing and important variables are *the available memory size, disk size, the processor capacity, the sizes of the object disc directories, file handlers and the disc speed.*

# 3 The CPN model

## 3.1 Modelling assumptions

In order to obtain a simple and feasible model of the caching system the following *assumptions* have been made.

1. The cached objects are placed in a set (without ordering).
2. The query of parent and neighbours is performed in a broadcast manner with specified time-out.
3. The name resolution procedure is not described, it is assumed that its impact is negligible.
4. If there is no response from the hierarchy within the specified time-out then the object is retrieved directly from the source server.
5. The processing of the required freshness of the requests (as specified by the client) is not described.
6. The transmission time of the request from the client is assumed to be negligible compared to the service time of the other processing steps therefore this processing step is not modelled.
7. The effect of the hierarchy related tuning parameters is not investigated therefore random variables are used to describe the average behaviour of the hierarchy towards the cache.

With the assumptions above a hierarchical stochastic timed CPN model [6] is developed which is described in a top-down manner.

The model has been implemented and tested using the Design/CPN package [7] containing a CPN editor, a macro-language for declarations and procedures and a CPN simulator. The model is described using the syntax of the Design/CPN package. The model pages of the CPN model are described in details below. The contents of the declaration node is found **in the Appendix.**. The PAGE IDENTIFIERS of the CPN model are denoted by SMALLCAPS, **places** by **bold** and *transitions* by *italic* letters.

## 3.2 Most abstract view of the cache model: the SUPERPAGE

This page contains the top hierarchy page of the model with the following main elements: places and transitions.

| | |
|---|---|
| **Arrival** | arrival of requests (input from a file) |
| *Registration* | a unique identifier is given to the requests |
| **Registered** | registered requests, each request has a unique `(URL, identifier)` touple |
| *Is it in the cache?* | check if the given URL is present in the cache |
| **Existence tested** | requests are marked for local or remote processing |
| *Local object processing* (substitution transition) | for requests present in the cache detailed description of the processing steps is on page LOCAL |
| **Local object processing done** | request has been processed locally (with appropiate time delay) |
| *Transmit to client* | with random time delay |
| *Remote object processing* (substitution transition) | for requests missing from the cache (the detailed description of the processing steps is on page REMOTE) |
| **Exit** | end of request processing (output to a file). |

## 3.3 Initialization: page INPUT

The timed requests are read from a file on this page and are transmitted to the place ARRIVAL.

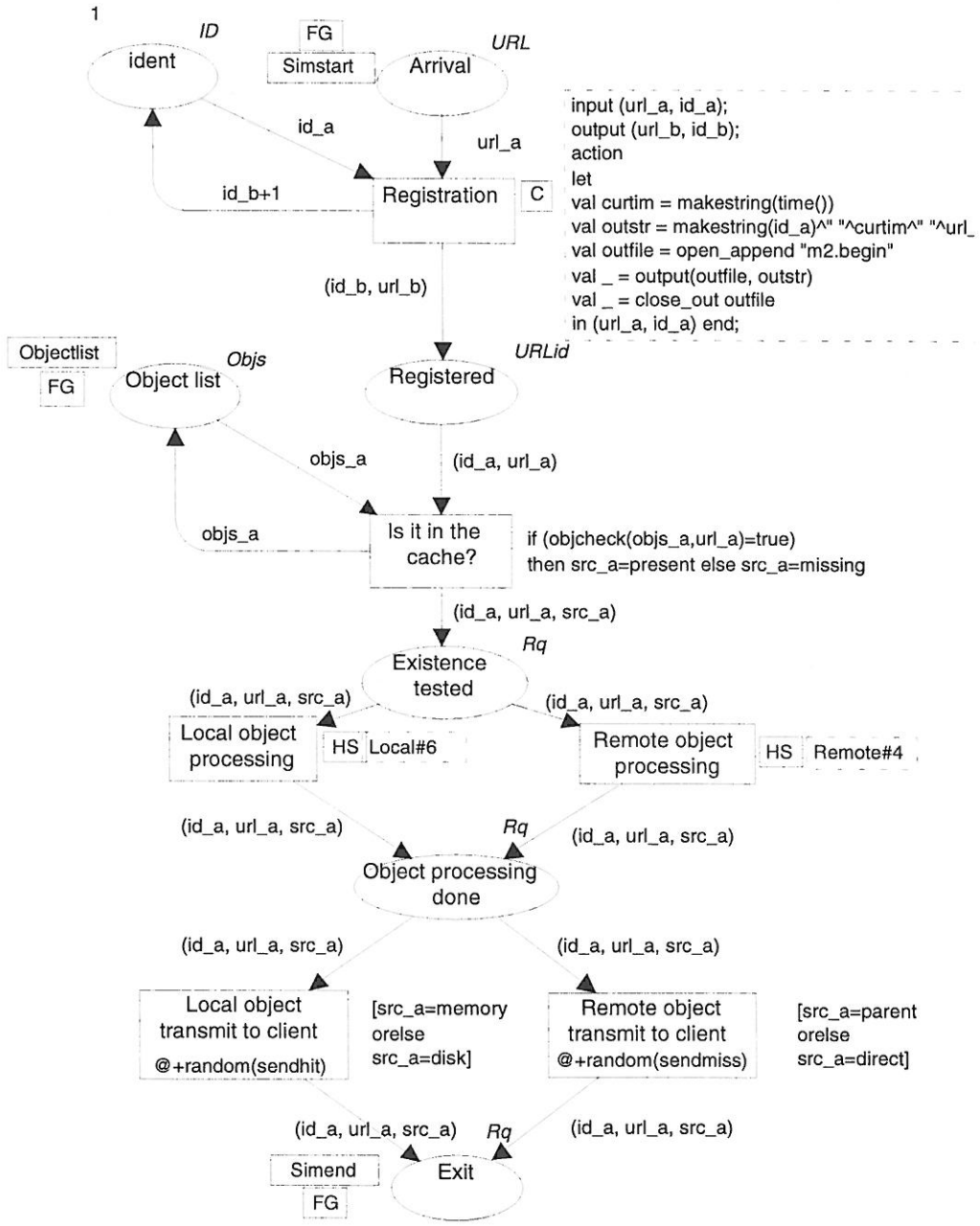



Figure 2: The SUPERPAGE page of the model

## 3.4 Local object processing: pages LOCAL and MEMORY UPDATE

On this page requests are read from the memory or disc and the last referenced time of the objects is updated.

| | |
|---|---|
| *Local object processing* | the requested object is present in the cache |
| **Local object** | local request |
| *Locate object* | check if the request is in the memory or on disc |
| **Located first** | request marked with memory or disc |
| *Disk I/O* | input of the object from disc (its time stamp is incremented by a random delay time based on measured data) |
| **Memory update needed** | we need to store objects read from disc in memory |
| *Memory updated* | memory has been updated (last reference time of the object is updated). If there is no more space in memory then an object is deleted from memory based on last referenced time (see in details on page MEMORY UPDATE). |
| **Memory objects** | the list of objects stored in memory manipulated by the branch coming from *Memory updated* |
| **Update enabled** | the object list can be updated after memory update |
| *Read from memory* | the object is in memory, we read it and update memory |
| **Memory objects** | list of objects in memory updated by the branch coming from *Read from memory* |
| **Local object** | local object processing done |

## 3.5 Remote object processing: page REMOTE

The processing steps when the request is not in the cache are collected on this page. The object is downloaded from the hierarchy or directly from the source. Thereafter the object is stored both in the memory and on the disc marked with the actual system time as its last referenced time. The garbage collection from disc is also performed here based on the last referenced time of the objects.

| | |
|---|---|
| *Is the object in the cache?* | the object is not in the cache |
| **Remote object needed** | remote processing is needed |
| *Request broadcast* | send request: a random variable on the output arc of this transition describes if the response comes from the |

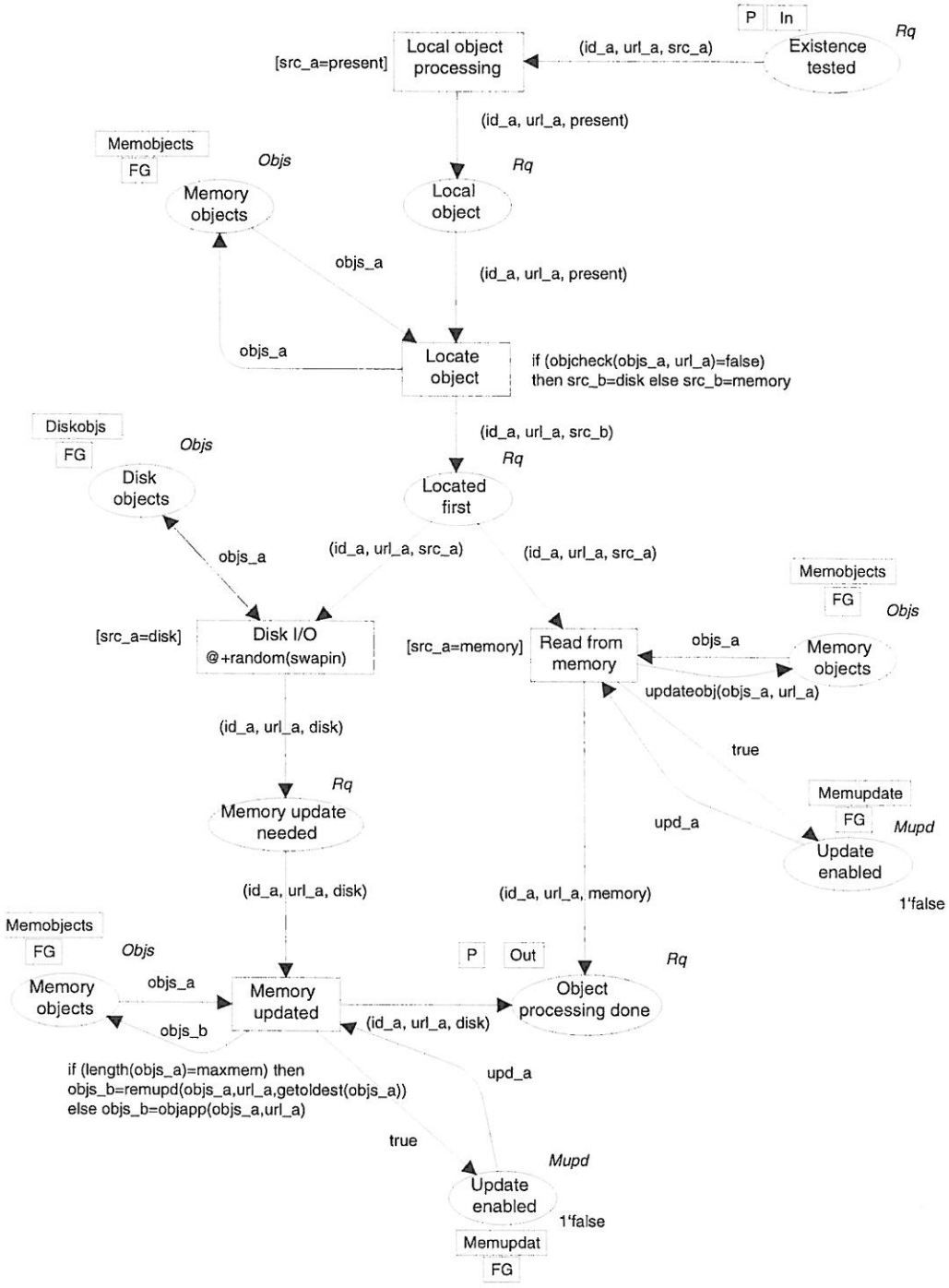



Figure 3: The LOCAL page of the model

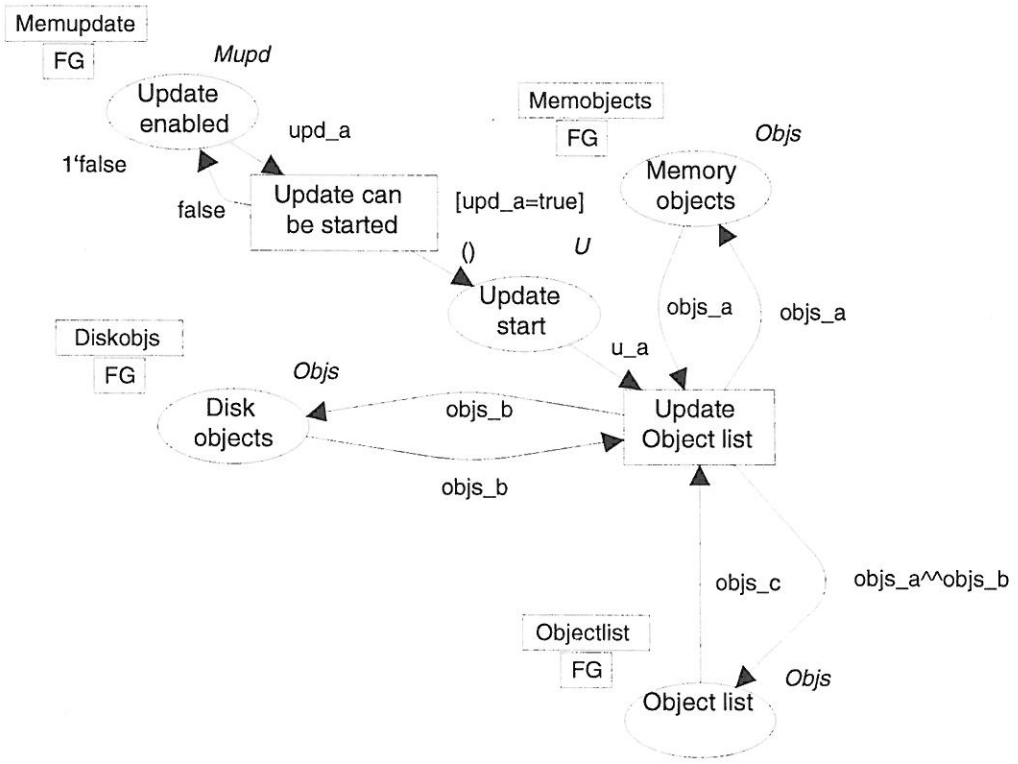



Figure 4: The UPDATE page of the model

| | |
|---|---|
| | parent or we have to download the object directly from its source |
| **Object or timeout** | object to be downloaded marked by its source (directly or from parent) |
| *Timeout, direct connection* | the object is downloaded directly from its source, its timestamp is incremented by a constant (tunable parameter) time-out value and a random time associated to the direct download |
| *Object comes from hierarchy* | the object is downloaded from the hierarchy (i.e. from the parent), its timestamp is incremented by a random value chosen from a multiset representing the download time from the hierarchy |
| **Remote object received** | the object has been received |
| *Update memory* | update the content of the memory |
| **Disk update follows** | in parallel with the transmission to the client we have |

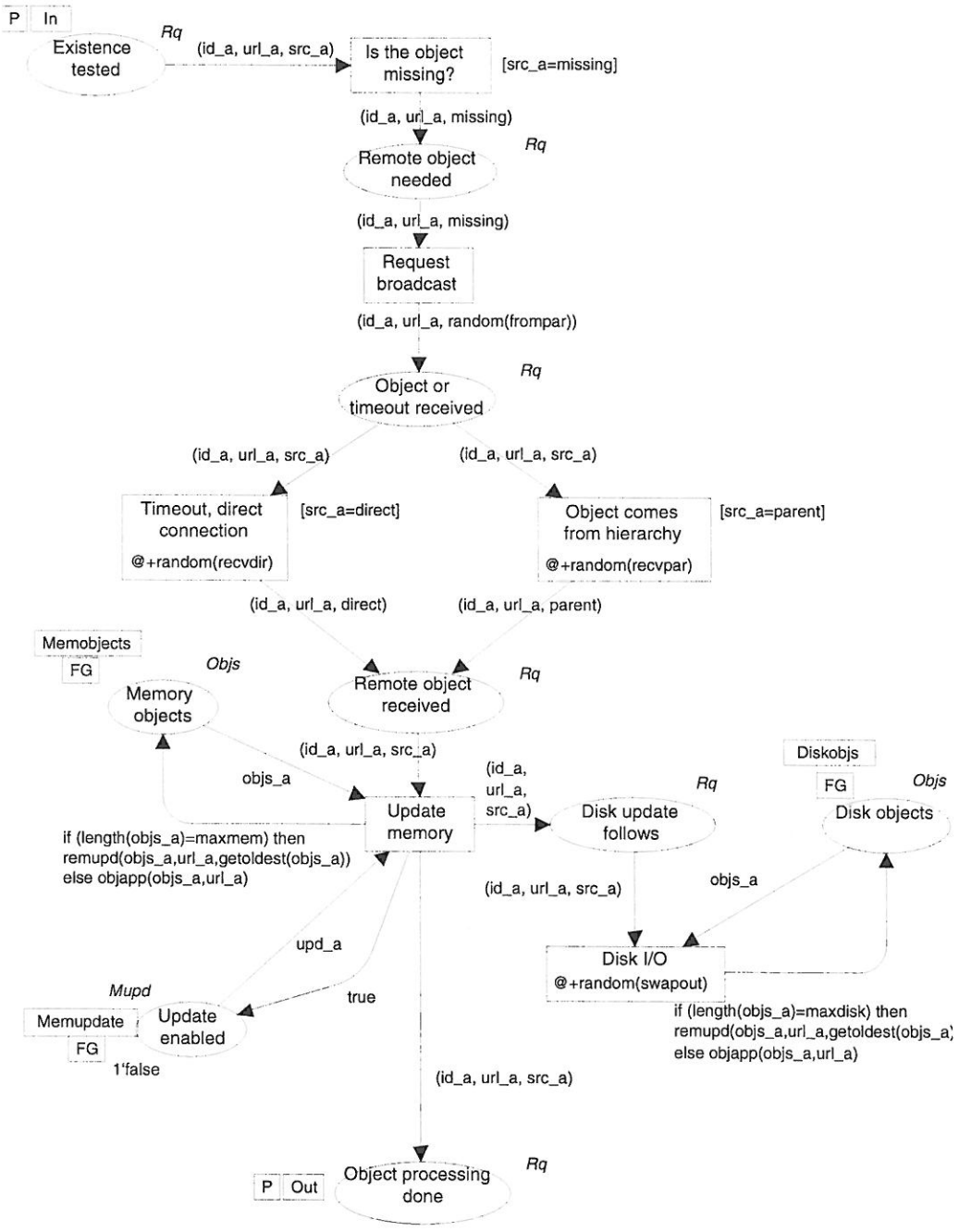



Figure 5: The REMOTE page of the model

to write the object to the disc, too

## 3.6 The most important tokens present in the CPN model

### 3.6.1 Tokens associated to requests (color Rq)

Requests are described by special coloured tokens where their colour is a vector of colours with elements

- request identifier: `id` (unique identification number)
- object identifier: `url` (Uniform Resource Locator)
- response source: `src` (local/from parent/from neighbour)

Moreover the requests possess their `time stamp`.

### 3.6.2 Tokens associated to stored objects (color Objs)

These are color set type tokens present on the places **Memory objects**, **Disk objects** and **Object list**. The color `Objs` is a list of records where an `url` and a `time stamp` showing the last usage time are put into a record.

The detailed description of all the other tokens used and all the procedures written is found in the **Appendix: The contents of the declaration node**.

## 3.7 Measurable variables and parameters of the model

### 3.7.1 Variables

The most important dynamic variables described in Section 2.3 are represented in the CPN model above in the following way.

1. *disturbance variables*
   The cache load is modelled through the time dependent set of the request tokens on the place **Arrival**.

2. *output variables*
   Both the response time and the hit-rate as the main characteristic variables are taken into account. The response time as a distribution is computed from the `time stamp` of the arrived request tokens on the place **Exit** while the hit-rate can be obtained from the `src` color item.

3. *state variables*
   The state variables of the cache system are represented by the list of the cached objects on the places **Object list**, **Memory objects** and **Disk objects**.

4. *potential input variables*
   The input variables are mainly present in various constraints, i.e. upper limit on available resources in the CPN model, such as

   - the maximal number of objects on the place **Object list**,
   - the time-out value associated to the transition *Request received*,
   - the maximal number of objects on the place **Memory objects**,
   - the maximal number of objects on the place **Disk objects**.

### 3.7.2 Parameters

In addition to the dynamic variables the CPN model contains *parameters* which characterise the modelled cache system. The model parameters are mostly response times of various kind associated to the relevant transitions in the CPN and can also be measured. These parameters possess random and slowly time varying nature, i.e. they can be characterised by slowly time varying distributions. The following table summarises the model parameters and their location in the CPN model.

| *Identifier* | *Meaning* | *Transition* |
|---|---|---|
| `disk-io` | disk i/o response time | *Disk I/O* |
| `trans-resp` | transmission time from the from hierarchy | *Object comes from hierarchy* |
| `send-client` | transmission time to the client | *Transmit to client* |

# 4 Simulation using the CPN model

## 4.1 Simulation input

A simulation run is performed by specifying the initial marking which represent the following cache variables:

- a set of request tokens put on the place **Arrival**,
- a set of tokens put on the places **Object list**, **Memory objects** and **Disk objects**.

The set of request tokens is placed in an input file generated from the log files of the caching software and read to the CPN simulator.

### 4.2 Simulation output

As a result of the completed simulation

- a set of processed request tokens on the place **Exit** and
- a set of new tokens on the places **Object list**, **Memory objects** and **Disk objects**

is generated.

Since we are primarily interested in distribution of the processing time of each request, the tokens collected at the place **Exit** are written out to an output file for further processing.

### 4.3 Simulation results

All the branches have been tested individually using specially designed simulation inputs and observing the simulation output. We do not present any slides from the simulation run, because the large number of tokens makes it impossible to follow the simulation on paper.

## 5 Model verification

To be able to eliminate irrelevant details from our model and to verify it against real behaviour we need to measure the static and dynamic characteristics of the modelled object cache. The dynamic variables (input, state, output and disturbance variables) of our CPN model have been measured using special tools. In addition, the model parameters above have also been determined from measurements either directly or in an indirect way.

### 5.1 Measurements

The caching software used at the modelled cache was Squid-1.1.17. [8] This software generates two log files which are useful for analysing the cache activity. We applied a patch made available by Alex Rousskov [9] which enhances the log file records by some time profiling data. It must be stressed that these measured data contain per request based measurements since the profiling data are measured by the caching software for every request. Details on the measurements and measured data have been reported elsewhere [4].

The model parameters have also been determined from these measured data.

- *Disk input/output*: the distribution of time needed for swapping an object from/to the disk to/from the virtual memory
- *Processing time of local object*: the distribution of time needed for the service of a local object (i.e. an object that can be serviced from the cache)
- *Retrieval time from the hierarchy*: the distribution of time needed for object retrieval from a different cache
- *Retrieval time from the source server*: the distribution of time needed for object retrieval from the source server
- *Request arrival rate*: the distribution of time between successive client requests

## 5.2 Comparison of the simulated model output and the measured output variables

The initial marking, i.e. the simulation input of a simulation run have been generated from the measured and evaluated per request based log files in different cases, i.e. for different cache load and time interval. The model output is generated therefrom by performing a simulation run and collecting the simulation output.

The model output is then compared to the records of the same log files the simulation input was generated from. The comparison is shown on Fig. 5.2. We can say that good correspondence has been found between the measured and simulated data based on our log files. The shape of the two data sets shows identical nature. The difference between the simulated and the measured data originates from *the gross tuning parameters*: we used a very limited number of data to specify the distribution of the model parameters. (The specification of model parameters can be seen in the Appendix).

# 6 Conclusion and Future work

A novel stochastic timed CPN model of an Internet object leaf cache has been developed and verifyed in the paper. The model has been implemented and tested in the Design/CPN software environment.

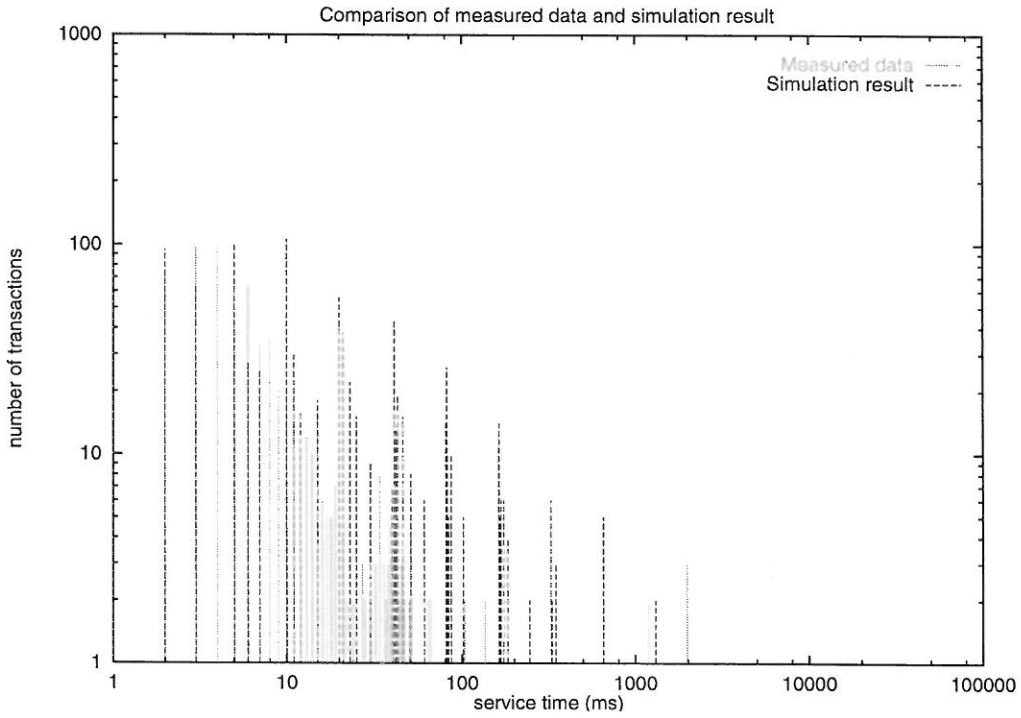



Figure 6: Comparison of measured data and simulation result

The aim of the modelling has been to use the model for control and optimization purposes: to develop a model-based adaptive methodology for tuning the cache parameters based on measured data and on the desired objective (loss) function of the cache performance.

Having verified the CPN model the following model validation steps are needed which will be the subject of our further research.

1. validation of the model output against measured data in different cache load regimes,
2. model sensitivity analysis with respect to the variations in model parameters,
3. model sensitivity analysis with respect to the variations of the potential input variables,
4. model refinement and simplification according to the aboves,
5. adaptive tuning experiments.

## Acknowledgement

This work has been partially supported by the Hungarian National Research Fund (OTKA) through grant T026575 which is gratefully acknowledged.

## References

[1] Evangelos P. Markatos: Main memory caching of Web documents. In *Proceedings of the Fifth International World-Wide Web Conference*, Paris, France, May 1996.

[2] Roland P. Wooster, Marc Abrams: Proxy caching that estimates page load delays. In *Proceedings of the Sixth International World-Wide Web Conference*, Santa Clara, California - USA, April 1997

[3] B. Kolics: Toward Intelligent Internet Object Caching (Diploma thesis). University of Veszprém, 1997.

[4] B. Kolics, K. M. Hangos, I. Tétényi: Experimental Investigation of Internet Object Cache Characteristics, *INET'98 Conference*, July 21–24, Genéve, Switzerland

[5] Ho, Y. C., Cassandras, C.: A New Approach to the Analysis of Discrete Event Dynamic Systems. *Automatica*, **19**, 149–167 (1983)

[6] Kurt Jensen: Coloured Petri Nets, *Basic Concepts, Analysis Methods and Practical Use*, Vol. 1–3, Springer Verlag, 1992-1997

[7] Design/CPN Online home page:
`http://www.daimi.aau.dk/designCPN/`

[8] Squid Internet Object Cache home page: `http://squid.nlanr.net/`

[9] Squid Profiling Statistics home page:
`http://www.cs.ndsu.nodak.edu/~rousskov/research/cache/squid/profiling/`

# Appendix: The contents of the declaration node

```
color ID = int; (* request identifier *)
color URL = string declare input_ms timed; (* object identifier *)
color Lrt = int; (* last referenced time *)
color Rt = with simple|conditional; (* request type *)
color Src = with present|missing|memory|disk|parent|neighbour|direct;
(* object source *)
color Obj = product URL * Lrt declare input_ms; (* object string
*)
color Objs = list Obj; (* object string list *)
color URLs = list URL; (* URL list *)
color URLid = product ID * URL timed;
color Rq = product ID * URL * Src timed declare output_ms; (* request
record *)
color Lrq = product ID * URL * Src * Lrt timed; (* local request
record *)
color U = unit; (* unit for initialization *)
color Mupd = bool; (* used for memory update enabling/disabling *)

var id_a,id_b:  ID;
var url_a, url_b:  URL;
var urls_a:  URLs;
var urlms:  URL ms; (* URL multiset variable *)
var objs_a, objs_b, objs_c:  Objs;
var src_a, src_b:  Src;
var lrt_a:  Lrt;
var upd_a:  Mupd;
var u_a:  U;
var rq_a:  Rq;

val maxmem = 1000; (* maximum number of objects in memory *)
val maxdisk = 100000; (* maximum number of objects on disk *)
val reqarrival =4`1+804`2+758`4+532`8+628`16+1119`32+1771`64+3164`128+
3938`256+3056`512+2921`1024+2109`2048+1117`4096+791`8192+389`16384+
142`32768+18`65536+4`131072+2`262144+90`524288;
(* time delay between successive requests *)
val swapout = 17`4+2199`8+3933`16+17742`32+35209`64+7807`128+3118`256+
908`512+266`1024+203`2048+75`4096+3`8192+1`32768;
(* swapout time *)
```

```
val swapin = 17‘2+6246‘4+4115‘8+4724‘16+8076‘32+11239‘64+6753‘128+3143‘256+
1222‘512+312‘1024+129‘2048+32‘4096+5‘8192+1‘16384+1‘32768;
(* swapin time *)
val sendhit = 12‘2+4168‘4+11007‘8+6074‘16+5061‘32+7470‘64+6290‘128+3868‘256
+1734‘512+574‘1024+330‘2048+139‘4096+76‘8192+69‘16384+57‘32768+
58‘65536+18‘131072+19‘262144+7‘524288+1‘1048576;
(* time it takes to process a 'hit' object *)
val sendmiss = 9‘1+1068‘2+16224‘4+19586‘8+4807‘16+6159‘32+8617‘64+
11361‘128+9841‘256+6904‘512+4796‘1024+4674‘2048+3322‘4096+2113‘8192+
1124‘16384+623‘32768+363‘65536+170‘131072+79‘262144+57‘524288+
33‘1048576+25‘2097152;
(* time it takes to process a 'miss' object *)
val recvpar = 3‘4+101‘8+155‘16+592‘32+1500‘64+2888‘128+3762‘256+4834‘512
+3238‘1024+2609‘2048+2393‘4096+1425‘8192+672‘16384+287‘32768+197‘65536+
87‘131072+35‘262144+32‘524288+19‘1048576+9‘2097152;
(* hierarchy response time *)
val recvdir = 21‘8+360‘16+257‘32+618‘64+1184‘128+1584‘256+1515‘512+
1680‘1024+1136‘2048+841‘4096+575‘8192+351‘16384+132‘32768+60‘65536+
21‘131072+4‘262144+2‘524288;
(* direct object retrieval time *)
val frompar = 293‘direct+607‘parent;
(* relation of direct and hierarchy responses *)
(* derived from log file analysis *)

fun getmin(q:  Obj, p:  Obj):Obj = if (#2(q)<#2(p)) then q else p;
(* returns the object with the lower *)
(* last referenced time value *)

fun last_used(l:  Objs, a:  Obj, i:  int):Objs*Obj*int = if (i=length(l))
then (l,a,i) else last_used(l,getmin(a,nth(l,i)),i+1);
(* returns the object string which has the *)
(* lowest last referenced value in the list *)
(* in the second field *)

fun remove_last_used(l:  Objs):Objs =
ms_to_list(list_to_ms(l)-1‘#2(last_used(l, (" ",maxage),startpoint)));
(* removes the last referenced object string *)
(* from an object string list *)

fun objcheck(objlst, myurl) = exists (fn elm => elm = (myurl, #2(elm)))
objlst;
```

```
(* tests whether an URL is in the list *)

fun updateobj(objlist, myurl) = map (fn elm => if (myurl=#1(elm))
then (myurl, time()) else elm) objlist; (* updates the timestamp
of an URL in a list *)

fun remupd(objlist, myurl, oldurl) = map (fn elm => if (oldurl=#1(elm))
then (myurl, time()) else elm) objlist; (* removes oldurl from list
and inserts myurl *)
(* to list with actual timestamp *)

fun geto3(l:Objs, i:int, j:int):int = if (#2(nth(l,i))<#2(nth(l,j)))
then i else j;
(* assistant function of getoldest *)
fun geto2(l:Objs, i:int, j:int):Objs*int*int = if (i=length(l)) then
(l,i,j) else geto2(l,i+1,geto3(l,i,j)); (* assistant function of
getoldest *)

fun getoldest(l:Objs):URL = #1(nth(l,#3(geto2(l,0,0))));
(* returns the URL that has the lowest LRT value *)

fun objapp(l:Objs, u:URL):Objs = l^^[(u,time())];
```

# Timed Colored Petri Net Models of Distributed Memory Multithreaded Multiprocessors

W.M. Zuberek[†], R. Govindarajan[‡] and F. Suciu[†]

[†]Department of Computer Science
Memorial University of Newfoundland
St.John's, Canada A1B 3X5

[‡]Supercomputer Education and Research Center
Indian Institute of Science
Bangalore 560 012, India

## Abstract

Distributed–memory multithreaded multiprocessors are composed of a number of (multithreaded) processors, each with its memory, and an interconnecting network. The long memory latencies and synchronization delays are tolerated by context switching, i.e., by suspending the current thread and switching the processor to another 'ready' thread provided such a thread is available. Because of very simple representation of concurrency and synchronization, timed Petri net models seem to be well suited for modeling and evaluation of such systems. Colors are used to represent the progress of remote memory access requests in the interconnecting network as well as to fold the models of individual processors.

This paper describes timed colored Petri net models of several multithreaded multiprocessor architectures, and presents some performance characteristics obtained by evaluation of these models.

## 1. Introduction

Multithreaded processors utilize the simple and efficient sequential execution technique of control–flow combined with data–flow like concurrency [14]. This supports the conceptually simple but quite powerful idea of rescheduling rather than blocking when waiting for data, e.g., from large and distributed memories, and thus can be used for tolerating long data transmission latencies. Multithreading makes multiprocessing far more efficient because the cost of moving data between distributed memories and processors can be hidden by other activities. The same hardware mechanisms can be used to synchronize interprocess communication and to alleviate operating system overheads.

Several multithreaded architectures have recently been proposed which differ in the implementation of multithreading [1, 3, 5, 7, 9]. Switching from one thread to another can be performed under different circumstances [4]:

- Switching on every instruction: one instruction is picked from each of runnable threads and is inserted into the processor's pipeline; if there are many threads, then each stage of the pipeline is executing an instruction from a different thread, and no instruction dependency problems exist [18].
- Switching on block of instructions: blocks of instructions from different threads are interleaved.
- Switching on every load: whenever a thread encounters a load instruction, the processor switches to another thread after that load instruction is issued; the context switch is irrespective of whether the data is local or remote [3].
- Switching on remote load: processor switches to another thread only when current thread encounters an access to remote memory [1].

The nodes of a multithreaded multiprocessor are linked by an interconnecting network. This network can be a two–dimensional torus–like network or a hypercube–type connection. It is assumed that all messages in the system are routed along the shortest paths, but in a non–deterministic manner. That is, whenever there are multiple (shortest) paths between the source and destination, any of the paths is equally likely to be taken. Consequently, the traffic on the links of the interconnecting networks is assumed to be uniformly distributed. The delay of each message is proportional to the number of hops between the source and destination nodes, and it also depends upon the traffic in the chosen path. A pair of network interfaces, one for outbound and the second for inbound traffic, connect the network switches with processor nodes.

Petri nets have been proposed as a simple and convenient formalism for modeling systems that exhibit parallel and concurrent activities [16, 15]. In order to take the durations of these activities into account, several types of Petri nets *with time* have been proposed by assigning *firing times* to the transitions or places of a net. In timed nets [20], deterministic or stochastic (exponentially distributed) firing times are associated with transitions, and transition firings occur in real–time, i.e., tokens are removed from input places at the beginning of the firing period, and they are deposited to the output places at the end of this period. In color nets [11], attributes (called colors) are associated with tokens, so different activities can be assigned to tokens of different types (i.e., colors), within the same structure of the net. For modeling of multithreaded architectures, colors are used for two different purposes. One use of token attributes is to forward requests of remote memory accesses to target nodes and than send the responses back to home nodes. The second use is to fold all the (identical) processor subnets.

Analyzing the performance of multiprocessor architectures is rather involved as it depends on a number of parameters related to the architecture — memory latency time, context switching time, switch delay in the interconnection network — and a number of application parameters — number of parallel threads, runlengths of threads, remote memory access pattern and so on. The performance of multithreaded architectures have been evaluated using discrete–event simulation [9, 12, 7], analytical models — using either queuing networks or Petri nets [2, 17], or using trace-driven simulation [19]. Event–driven simulation of the timed colored net models [21] was used to obtain the results presented in this paper.

This paper presents Petri net models of several multithreaded multiprocessor architectures, discusses the development of colored net models and includes some performance results as an illustration of model analysis. The influence of architectural and application parameters on the performance of the system is also discussed.

## 2. Timed Petri nets

This section first recalls elementary concepts of place/transition nets and colored nets, and then reviews the basics of timed Petri nets.

### 2.1. Basic concepts of Petri nets

The basic marked place/transition Petri net $\mathcal{M} = (P, T, A, m_0)$ is usually defined as a system composed of a finite, nonempty set of places $P$, a finite, nonempty set of transitions $T$, a set of directed arcs $A$, $A \subset P \times T \cup T \times P$, and an initial marking function $m_0$ which assigns nonnegative numbers of so called tokens to places of the net, $m_0 : P \to \{0, 1, ...\}$. Usually the set of places connected by (directed) arcs to a transition is called the *input set* of a transition, and the set of placed connected by (directed) arcs outgoing from a transition, its *output set.*

A place is shared if it belongs to the input set of more than one transition. A net is conflict–free if it does not contain shared places. A shared place is (generalized) free–choice if all transitions sharing it have the same input sets. Each free–choice place determines a class of free–choice transitions sharing it, and free–choice classes of transitions determined by different free–choice places are disjoint. It is assumed that selection of a transition for firing in each free–choice class of transitions is a random process which can be described by (free–choice) probabilities assigned to transitions. Moreover, it is usually assumed that the random choices in different free–choice classes are mutually independent.

A shared place which is not free–choice, is a conflict place. Enabled transitions sharing a conflict place are called conflicting transitions. The class of conflicting transitions is defined in a transitive way, i.e., two transitions are conflicting if they share a conflict place or if there exists another transition such that it shares a conflict place with one of these two transitions and is conflicting with the other transition. The conflicting relation is an equivalence relation in the set of transitions, which implies a partition of this set into disjoint classes of conflicting transitions. For each conflicting class, the probabilities of firings can be determined on the basis of relative frequencies of transition firings [10]; the probability of firing an enabled transition $t$ is determined as the ratio of $t$'s (relative) frequency to the sum of relative frequencies of all transitions in the conflict class of $t$. Quite often such relative frequencies (and probabilities of firings) are dynamic, depending upon the marking function, for example, by using the number of tokens in a place rather than a fixed, constant number as the relative frequency. The determination of conflicting transitions depends upon the marking function, so the probabilities of firing conflicting transitions must be determined in a dynamic way.

In basic nets, the tokens are indistinguishable, so their distribution can be described by a simple marking function $m : P \to \{0, 1, ...\}$. In colored Petri nets [11], tokens have

attributes called colors, so a marking function becomes $m : P \to C \to \{0, 1, ...\}$ where $C$ is a finite set of token colors (or attributes). Token colors can be modified by (firing) transitions and also a transition can have several different occurrences (or variants) of its firings for different combinations of (colored) tokens in its input places. This is captured by a definition of a colored net $\mathcal{C} = \{P, T, A, C, a, m_0\}$, where the *arc* function $a$ describes the numbers of colored tokens required in input places and deposited to output places for different *colors* of transitions' firings (or different occurrences of transitions), $a : A \to C \to C \to \{0, 1, ...\}$ (the definition is slightly different from the one used by Jensen [11], but is consistent with it). It should be observed that if $C$ contains just one color, the colored net reduces to a 'standard' place/transition net.

The basic idea of colored nets is to 'fold' an ordinary place/transition net. The original set of places is partitioned into a set of disjoint classes, and each class is replaced by a single place with token colors indicating which of the original places the tokens belong to. Similarly, the original set of transitions is partitioned into a set of disjoint classes, and each class is replaced by a single transition with occurrences indicating which of the original transitions the firing corresponds to. Any partition of places and transitions will result in a colored net. One of the extreme partitions will combine all original places into one place, and all original transitions into one transition; this will create a very simple net (one place and one transition only) but with quite complicated rules describing the use of colors. The other extreme partition will create one–element classes of places and transitions, so the colored net will be isomorphic to the original net, with only one color. To be useful in practice, colored nets must constitute a reasonable balance between these two extreme cases.

### 2.2. Timed Petri nets

In order to study performance aspects of Petri net models, the duration of activities must also be taken into account and included into model specifications. Several types of Petri nets 'with time' have been proposed by assigning firing times to the transitions or places of a net. In timed nets, firing times are associated with transitions (or occurrences), and transition firings are 'real–time' events, i.e., tokens are removed from input places at the beginning of the firing period, and they are deposited to the output places at the end of this period (sometimes this is also called a 'three–phase' firing mechanism as opposed to a 'one–phase', instantaneous firings of nets without time or stochastic nets).

In timed nets, all firings of enabled transitions are initiated in the same instants of time in which the transitions become enabled (although some enabled transition cannot initiate their firings). If, during the firing period of a transition, the transition becomes enabled again, a new, independent firing can be initiated, which will overlap with the other firing(s). There is no limit on the number of simultaneous firings of the same transition (sometimes this is called 'infinite firing semantics'). Similarly, if a transition is enabled 'several times' (i.e., it remains enabled after initiating a firing), it may start several independent firings in the same time instant.

A timed (colored) net can be defined as a quadruple $\mathcal{T} = (\mathcal{N}, m_0, c, f)$, where $\mathcal{N}$ is a (colored) net structure, $m_0$ is the initial marking function, $c$ is the choice function which assigns free–choice probabilities to free-choice occurrences of transitions and relative frequencies of firings to (potentially) conflicting occurrences of transitions, $c : T \to C \to \mathbf{R}^+$,

and $f$ is the firing time function which assigns the (average) firing times to occurrences of transitions, $f : T \to C \to \mathbf{R}^{\oplus}$.

The firing times of some transitions may be equal to zero, which means that the firings are instantaneous; all transitions with zero firing times are called *immediate* (while the other are called *timed*). Since the immediate transitions have no tangible effect on the (timed) behavior of the model, it is convenient to assume that first all (enabled) immediate transitions are fired, and then (still in the same time instant), when no more immediate transitions are enabled, to start the firings of (enabled) timed transitions. It should be noted that such a convention introduces the priority of immediate transitions over the timed ones, so the conflicts of immediate and timed transitions should be avoided. Similarly, the free–choice classes of transitions must be 'uniform', i.e., all transitions in each free–choice class must be either immediate or timed.

The behavior of timed nets can be described by states and state transitions where each state represents the distribution of (remaining) tokens in places as well as distribution of firing transitions (some transitions can fire several times). The transitions between states can be combined into a graph of reachable states. For timed nets with exponentially distributed firing times, this graph is simply a Markov chain of the stochastic process represented by the timed net. For nets with deterministic firing times, this graph is a semi–Markov (or embedded Markov) chain. In both cases standard techniques developed for Markov process can be used to find stationary properties of states (if the state space is finite) and then derive performance characteristics from these stationary probabilities [20].

## 3. Multithreaded processors

In the multithreaded execution model, a program is a collection of partially ordered threads, and a thread consists of a sequence of instructions which are executed in the conventional von Neumann model. Scheduling of different threads follows the data–driven approach.

A Petri net model of a single processor for the case when context switching is performed for all long–latency (local and remote) memory accesses, is shown in Fig.3.1, which also shows the two switches, *Tsinp* and *Tsout*, for incoming and outgoing traffic, respectively. The interconnection of the node with its four neighbors (for the case of a two–dimensional torus–like interconnecting network; the interconnection networks are discussed in greater detail in Section 4) is also shown in Fig.3.1.

The execution of (ready) threads is modeled by transition *Trun* with place *Proc* representing the (available) processor (if marked) and *Ready* – the pool of threads waiting for execution. The initial marking of *Ready* represents the average number of threads, $n_t$, one of important model parameters. It is assumed that this number does not change in time.

The firing time of *Trun* is exponentially distributed (all other firing times are deterministic) and its average value represents the runlength of threads, i.e., the average number of instructions executed before context switching occurs. The runlength, $\ell_t$, is another important parameter of the model.

*Mem* is a free–choice place, with a random choice of either accessing local memory (*Tloc*) or remote memory (*Trem*); in the first case, the request is directed to *Lmem* where

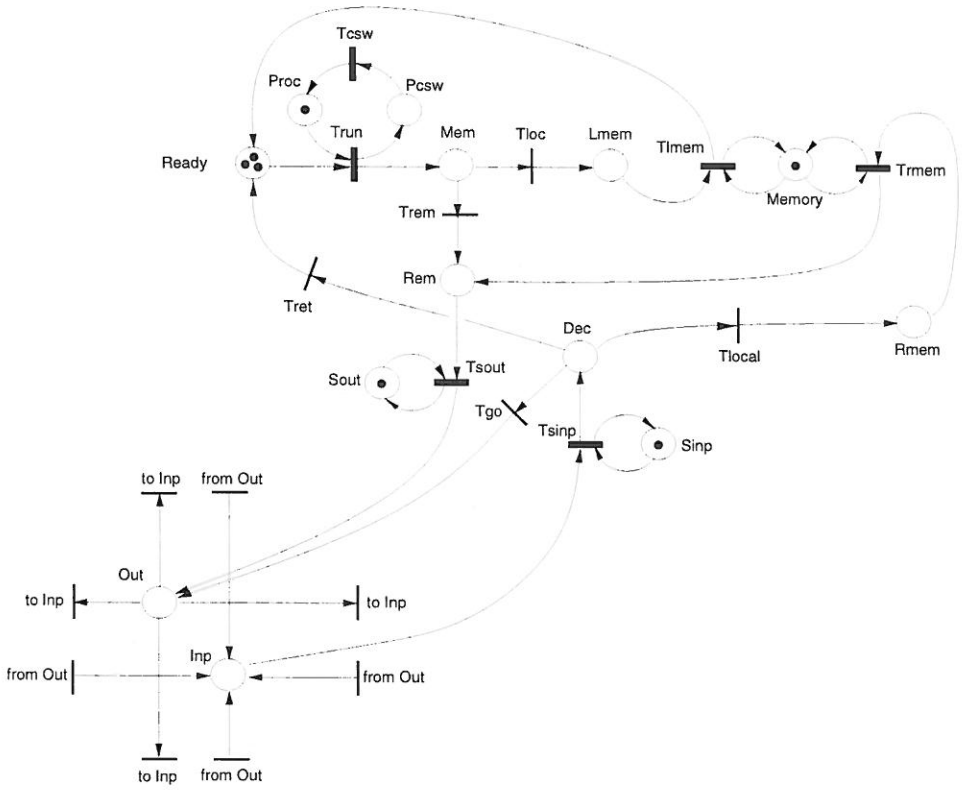



Fig.3.1. Petri net model of a single processor; context switch on all memory accesses.

it waits for availability of *Memory*, and after accessing the memory, the thread returns to the queue of waiting threads, *Ready*. *Memory* is a shared place with two conflicting transitions, $Trmem$ (for remote accesses) and $Tlmem$ (for local accesses); the resolution of this conflict (if both accesses are waiting) is based on marking–dependent (relative) frequencies determined by the numbers of tokens in *Lmem* and *Rmem*, respectively.

The free–choice probability of $Trem$, $p_r$ (or of $Tloc$, $p_\ell = 1 - p_r$), is another basic parameter of the model.

Requests for remote accesses are directed to *Rem*, and then, after a sequential delay (the switch modeled by *Sout* and $Tsout$), forwarded to *Out*, where a random selection is made of one of the four (in this case) adjacent nodes (all nodes are selected with equal probabilities). Similarly, the incoming traffic (memory access requests from remote nodes and responses to remote accesses) are collected from all neighboring nodes in *Inp*, and, after a sequential delay (*Sinp* and $Tsinp$), forwarded to *Dec*. *Dec* is a free–choice place with three transitions sharing it: $Tret$, which represents the satisfied requests reaching their 'home' nodes; $Tgo$, which represents requests as well as responses forwarded to another node (another 'hop'); and $Tlocal$, which represents remote requests accessing the memory at the destination node; these remote requests are queued in *Rmem* and served by $Trmem$ when the memory module *Memory* becomes available.

The traffic outgoing from a node (place *Out*) is composed of requests and responses for-

warded to another node (transition $Tgo$), responses to requests from other nodes (transition $Trmem$) and remote memory requests originating in this node (transition $Trem$).

There are six timed transitions in the model shown in Fig.3.1. It is convenient to assume that all firing times are expressed in terms of the 'processor cycle time' used as a 'unit of time'; then the parameters and their typical values are:

| *symbol* | *parameter* | *typical values* |
|---|---|---|
| $n_t$ | the (average) number of threads | 2,...,20 |
| $\ell_t$ | thread runlength | 5,10,20 |
| $t_{cs}$ | context switching time | 1,10 |
| $t_m$ | memory cycle time | 10 |
| $t_s$ | switch delay | 5, 10 |
| $p_\ell$, $p_r$ | probability of accesses to local/remote memory | 0.1,...,0.9 |

Fig.3.2 shows a processor's model for the case when the context switching is performed for remote memory accesses only (if the time of context switching is comparable with the access time to local memory, context switching for accesses to local memory is not justified).

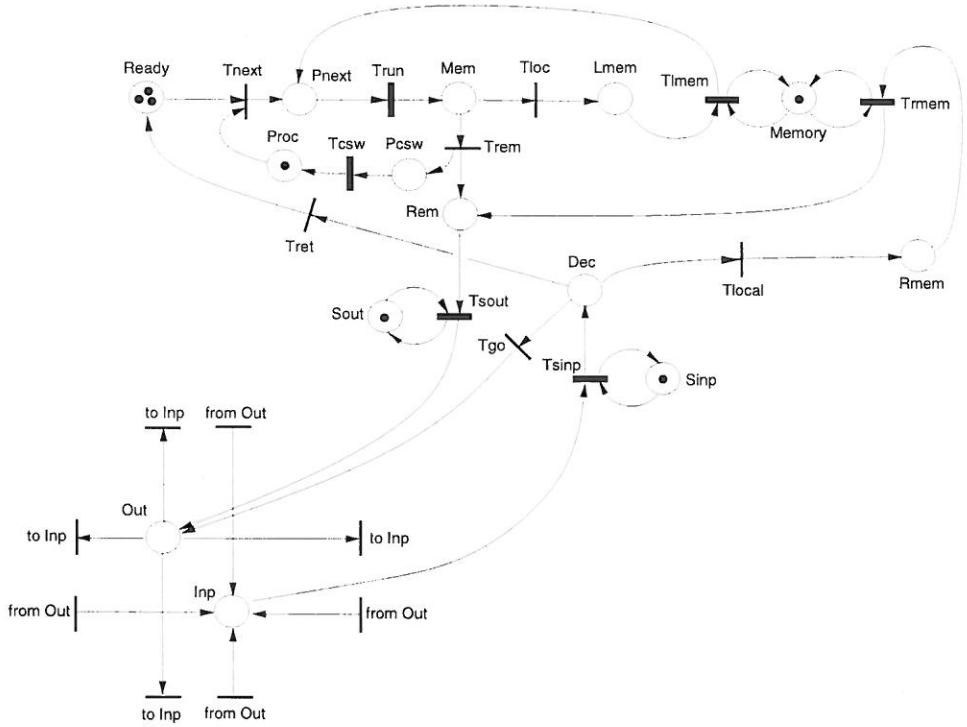



Fig.3.2. Petri net model of a single processor; context switching on remote memory accesses.

Selection of a thread for execution is represented by transition *Tnext*. During accesses to local memory (*Tlmem*), the processor is waiting for the completion of the access, after which the execution of the same thread continues (transition *Trun*). For accesses to remote memory, transition *Trem* initiates context switching by depositing a token in *Pcsw*; *Tcsw* represents context switching (by its firing time), after which the processor is available for execution of another thread from *Ready*, the pool of waiting threads.

## 4. Interconnecting networks

Processors are usually connected by either a torus–like network or a hypercube–type connection. Fig.4.1 shows a 16–node two–dimensional torus network where each node contains a processor (as in Fig.3.1 or Fig.3.2), and all connections between pairs of nodes are two–directional.

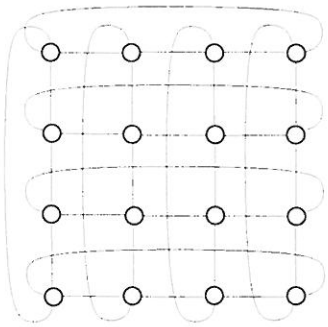


Fig.4.1. Torus–like network for a 16–processor system.

The free–choice probability of $Tgo$ can be determined from the 'traffic patterns' in the interconnecting network. Assuming that all (remote) memory requests are uniformly distributed over the nodes, the average number of hops can be calculated from the lengths of the (shortest) paths between the nodes. For a 16–processor system, for each node there are 15 remote nodes, 4 of which are at the distance of 1 hop, 6 at the distance of 2 hops, 4 at the distance of 3 hops, and 1 node at the distance of 4 hops, as sketched in Fig.4.2, where "0" denotes the 'reference node'. The average distance is thus:

$$\frac{4*1+6*2+4*3+1*4}{15} \approx 2 \text{ hops}$$

If a simple geometric distribution of the number of hops is assumed, the average value for this distribution is:

$$\frac{1}{1-p} = 2 \text{ hops}$$

so $p$, the probability that a request is forwarded to a next node (i.e., the free–choice probability of $Tgo$) is $p = 0.5$. The model of geometric distribution is very simple (as shown in Fig.3.1 and Fig.3.2) but this distribution does not restrict forwarded requests to 4 hops; consequently, there is a small probability that some requests can be forwarded beyond the limit of 4 hops. Also, the probability density function of the number of hops for the geometric distribution does not represent the real distribution very accurately; it appears that the real distribution can be modeled very conveniently by colored nets, as described further.

In order to realistically represent the two–directional traffic in the interconnecting network, i.e., streams of requests (for remote accesses) and streams of responses (to these requests) that 'return' to the original nodes, the model uses two colors of tokens in the interconnecting network, "F" for forward moving requests and "B" for backward moving responses. The tokens (requests) generated by $Trem$ are thus of color "F", while those

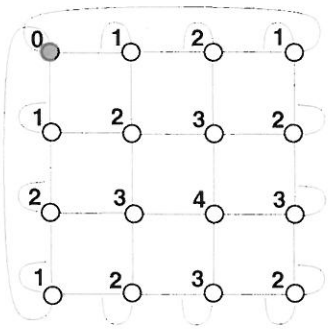



Fig.4.2. The minimal distances between nodes.

generated by *Trmem* are of color "B". The stream of colored tokens reaching *Dec* is separated so that only tokens of color "F" enable *Tlocal* and only tokens of color "B" enable *Tret* (*Tgo* is enabled by tokens of both colors). Consequently, the free–choice probability of *Tlocal* is 0.5 for color "F" (and 0 for color "B"), and that of *Tret* is 0.5 for color "B" (and 0 for color "F"); free–choice probability of *Tgo* is 0.5 for both colors, "F" and "B".

There is one more consequence of using colored tokens in the interconnecting network: the switches represented by *Tsinp* and *Tsout* have different occurrences for different colors of tokens, and these occurrences are in conflict because of sharing common places *Sinp* and *Sout*, respectively. The solution used to resolve these conflicts is based on marking–dependent (relative) frequencies, determined by the numbers of tokens in *Inp* and *Rem*, respectively (similarly to the conflict resolution of *Tlmem* and *Trmem*).

If the two occurrences of transitions *Tsinp* and *Tsout* are denoted by "F" and "B", the following tables describe the arc function $a$ for arcs incident with the two switches ("U" denotes the 'universal' color, when only one color is needed):

| *Tsinp* | *Inp* | *Sinp* | *Dec* | *Sinp* |
|---|---|---|---|---|
| F | F:1 | U:1 | F:1 | U:1 |
| B | B:1 | U:1 | B:1 | U:1 |

| *Tsout* | *Rem* | *Sout* | *Out* | *Sout* |
|---|---|---|---|---|
| F | F:1 | U:1 | F:1 | U:1 |
| B | B:1 | U:1 | B:1 | U:1 |

where "$X : n$" denotes a function $a : C \to \{0, 1, ...\}$ such that $a(X) = n$, and for all other colors $Y \in C - \{X\}$, $a(Y) = 0$.

The geometric distribution of the number of hops in the network (Fig.3.1 and Fig.3.2) can be refined to a more accurate, but also more complicated representation. Since the exact probabilities of making another hop are known (Fig.4.2), the real values of these probabilities can be used in the model instead of the geometric distribution; Fig.4.3(a) shows the real probability density function of the number of hops for remote memory accesses in a 16–processor system while Fig.4.3(b) shows the density function of the geometric distribution with the same average value as the distribution in Fig.4.3.(a) (this average value is 2).

In order to model the distribution of Fig.4.3(a), four "forward" colors are needed (say, F1, F2, F3 and F4) as there are four (discrete) values in this distribution, and four "backward colors" (for example, B1, B2, B3, B4). Transition *Trem* generates a token of color F1 (for the first hop); if a token of color F1 continue for another hop (free–choice transition *Tgo*), its color is changed from F1 to F2 (by occurrence F1 of *Tgo*); similarly, if a token of color F2 continues for another hop, its color is changed to F3 (by occurrence F2 of *Tgo*),

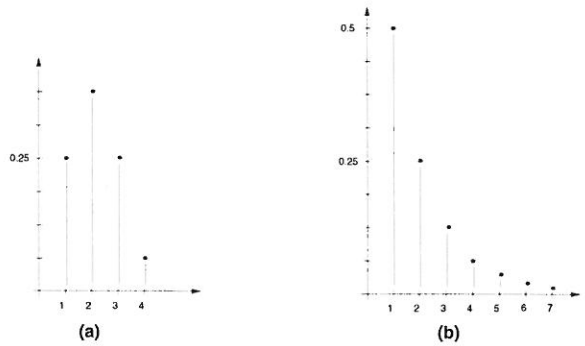



Fig.4.3. The distributions of the number of hops; 16 processors.

etc. If a token of color Fi accesses memory (through *Tlocal*, *Rmem* and *Tmem*), its color changes from Fi to Bi, and then *Tgo* forwards all Bi tokens "decreasing" their colors from B4 to B3, from B3 to B2, and from B2 to B1. Tokes of color B1 enable only *Tret*, and, after firing this transition, are returned to *Ready* (with the 'universal' color U).

The free–choice transitions *Tlocal* and *Tgo* will have four occurrences (say F1, F2, F3, and F4) with the following free–choice probabilities:

| *occurrence* | *Tlocal* | *Tgo* |
|---|---|---|
| F1 | 4/15 | 11/15 |
| F2 | 6/11 | 5/11 |
| F3 | 4/5 | 1/5 |
| F4 | 1.00 | 0 |

These probabilities correspond to the distribution in Fig.4.3(a), "rescaled" for consecutive hops; after the first hop, the probability of accessing memory is 4/15 (there are 4 nodes in distance of 1 hop in Fig.4.2); the probability of accessing memory after 2 hops is 6/15 (Fig.4.3(a)), or 11/15*6/11 (11/15 to continue after the first hop, and 6/11 to select memory accessing after the second hop), and so on. After 4 hops, in a 16–processor system there is no choice; the requests have to access the memory.

The free–choice probabilities of *Tgo* and *Tret* are:

| *occurrence* | *Tgo* | *Tret* |
|---|---|---|
| B1 | 0 | 1.00 |
| B2 | 1.00 | 0 |
| B3 | 1.00 | 0 |
| B4 | 1.00 | 0 |

The second major use of the colors of tokens is to "fold" all subnets representing the processors (or nodes). Such a "folded" model is shown in Fig.4.4, with 16 colors, N$ij$, $i = 1, 2, 3, 4$, and $j = 1, 2, 3, 4$, representing the original array of 4×4 processors.

Transition *Tnet* has 16*4 occurrences, which are denoted N$ijk$, $i = 1, 2, 3, 4$, $j = 1, 2, 3, 4$, $k$ =N,E,S,W. These occurrences correspond to the 4 connections (N – north, E – east, etc.) of each of the 16 nodes in the original network. All these occurrences form free–choice

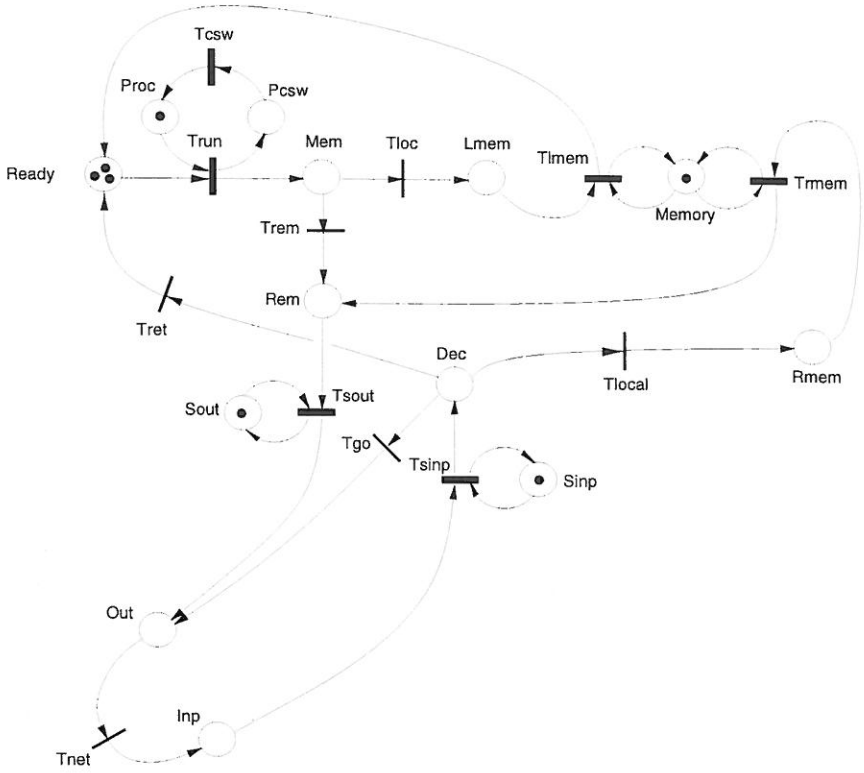



Fig.4.4. Colored net model of a multiprocessor system; context switching on all memory accesses.

classes in groups of 4, for different values of $k$; if the traffic in the network is uniformly distributed, all free–choice probabilities of these occurrences are equal. A partial definition of the arc functions, showing the transformation of token colors, is:

| *Tnet* | *Out* | *Inp* |
|---|---|---|
| N11N | N11:1 | N41:1 |
| N11E | N11:1 | N12:1 |
| N11S | N11:1 | N21:1 |
| N11W | N11:1 | N14:1 |
| N12N | N12:1 | N42:1 |
| N12E | N12:1 | N13:1 |
| N12S | N12:1 | N22:1 |
| N12W | N12:1 | N11:1 |
| .... | ... | ... |

Since *Tnet* is the only transition with such a large number of occurrences, it may be more reasonable to replace *Tnet* by four free–choice transitions, representing the choice of a neighbor N, W, S or E, as shown in Fig.4.5, each transition with 16 occurrences corresponding to 16 nodes. A partial definition of the arc functions for this solution is:

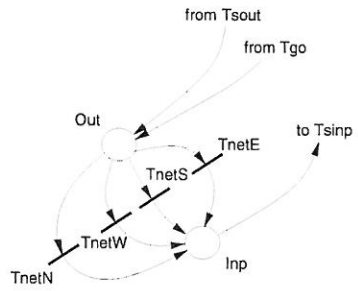



Fig.4.5. Refined colored model of the interconnecting network.

| *occurrence* | *Out* | *TnetN* | *TnetE* | *TnetS* | *TnetW* |
|---|---|---|---|---|---|
| N11 | N11:1 | N41:1 | N12:1 | N21:1 | N14:1 |
| N12 | N12:1 | N42:1 | N13:1 | N22:1 | N11:1 |
| N13 | N13:1 | N43:1 | N14:1 | N23:1 | N12:1 |
| N14 | N14:1 | N44:1 | N11:1 | N24:1 | N13:1 |
| N21 | N21:1 | N11:1 | N22:1 | N31:1 | N24:1 |
| N22 | N22:1 | N12:1 | N23:1 | N32:1 | N21:1 |
| ... | ... | ... | ... | ... | ... |
| N44 | N44:1 | N34:1 | N41:1 | N14:1 | N43:1 |

For hypercube–type interconnecting networks, the main difference is in the average number of hops that a request must make to reach its destination (and then return to the home node). In a net with $N = 2^n$ nodes (or processors) spanning along $n$ dimensions, each node is connected with $n$ other nodes. An example of a 4–cube network (for $2^4 = 16$ nodes) is shown in Fig.4.6.

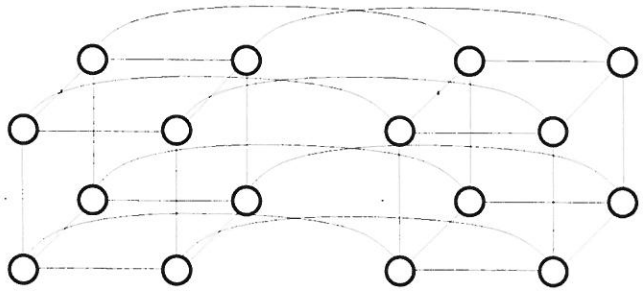


Fig.4.6. 4–cube interconnecting network obtained from two 3–cubes.

For example, in a 64–processor system, each node is connected with 6 other nodes ($64 = 2^6$), and the average number of hops is equal to 3.1. In a two–dimensional torus–like interconnecting network with 64 nodes, each node is connected with 4 neighbors, and the average number of hops is equal to 4. Consequently, in a torus–line network there will be more traffic and greater service demands for switches than in a hypercube–type network with the same number of nodes and the same parameters. The delays of remote memory accesses will thus be longer, and the processors utilization will be lower for the torus–like interconnecting network.

It should be observed that for a 16–processor system the two types of interconnecting networks are equivalent; each processor is connected to 4 other processors, the average numbers of hops are the same, so the throughputs are the same, and processor utilizations also.

## 6. Performance results

All results presented in this section have been obtained by simulation of the corresponding net models [21].

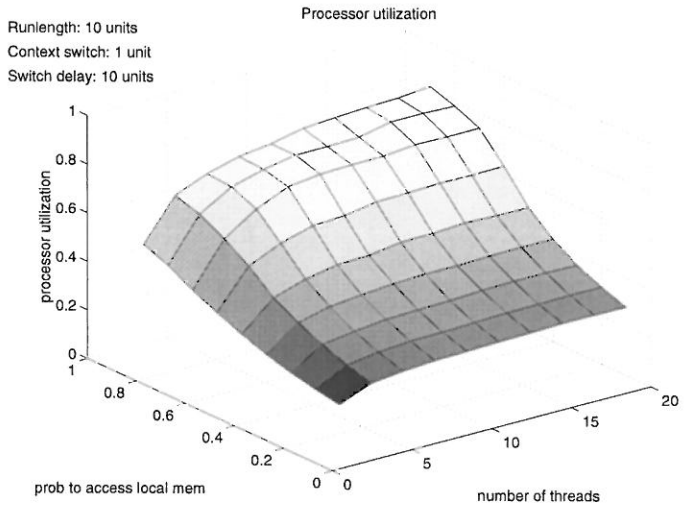



Fig.5.1. Processor utilization ($t_s = 10, t_{cs} = 1$).

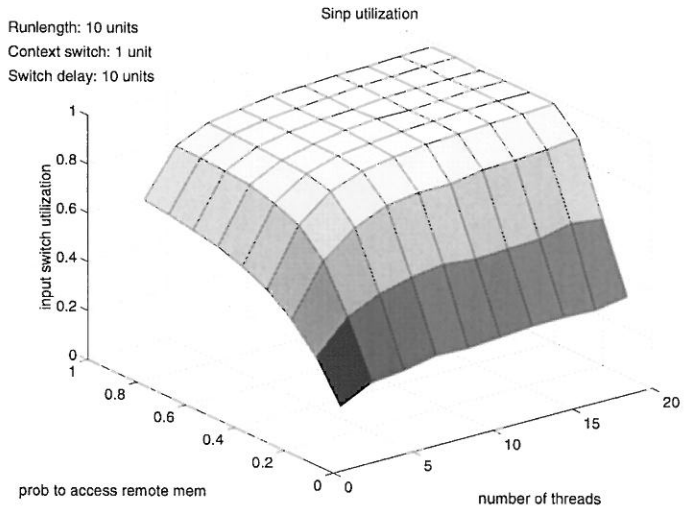



Fig.5.2. Input switch utilization ($t_s = 10, t_{cs} = 1$).

Fig.5.1 shows the utilization of the processor as a function of the (average) number of threads, $n_t$, and the probability of accesses to local memory, $p_\ell$; the remaining parameters

are: context switching time $t_{cs} = 1$, runlength $\ell_t = 10$, memory cycle $t_m = 10$, and switch delay $t_s = 10$.

With the exception of the region corresponding to $p_\ell$ close to one (and $p_r$ close to zero), the utilization tends to "saturate" very quickly at the levels significantly below the maximum value. This is a characteristic indication that some other (than processor) component limits the performance of the system, and becomes the bottleneck [6]. Indeed, it appears that the input switch *Tsinp* is too "slow", and is utilized is almost 100 %, as shown in Fig.5.2 (note that probability of remote accesses, $p_r$, is used in Fig.5.2 rather than $p_\ell$, so the "front" of Fig.5.2 corresponds to the "back" of Fig.5.1).

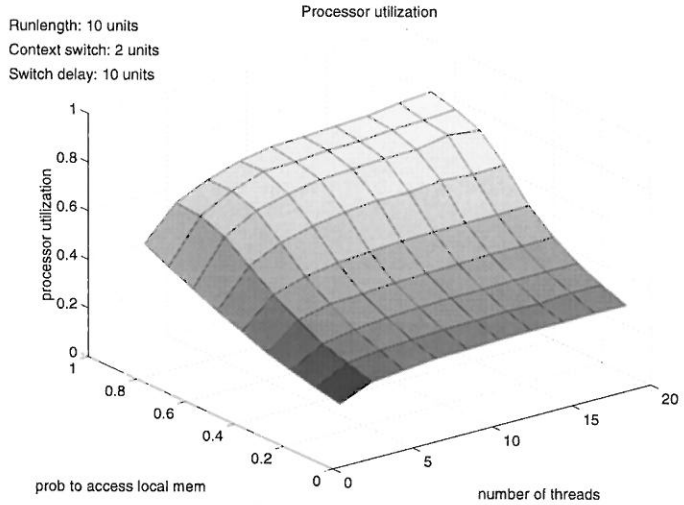



Fig.5.3. Processor utilization ($t_s = 10, t_{cs} = 2$).

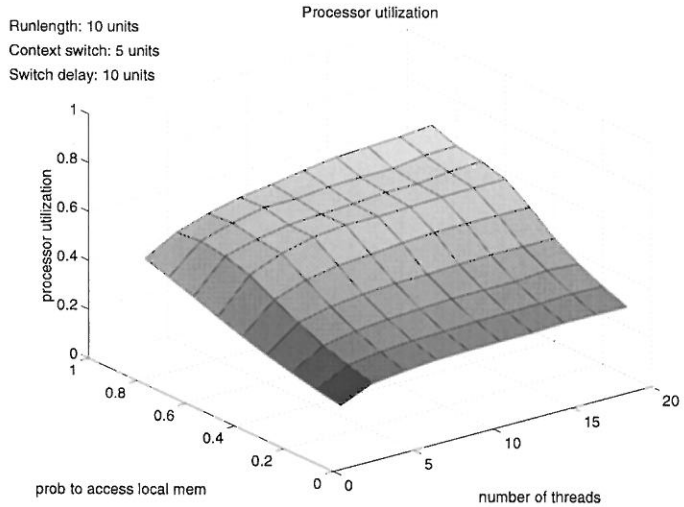



Fig.5.4. Processor utilization ($t_s = 10, t_{cs} = 5$).

The influence of the context switching time can be observed in Fig.5.3 and Fig.5.4, which show the processor utilization as a function of the number of threads and the probability of accesses to local memory, as in Fig.5.1, but for the context switching times equal to 2 and 5, respectively; the maximum values of the processor utilization clearly decrease when

the context switching time increases (for the "saturated" region they remain practically the same).

It can be shown that for the probability $p_\ell$ close to 1, for large numbers of threads, and for $l_t \geq t_m$, the upper bound on the value of processor utilization is equal to $l_t/(l_t + t_{cs})$; for Fig.5.3, this bound is equal to 0.83, and for Fig.5.4 is equal to 0.67 (for Fig.5.1 the bound is equal to 0.91).

Fig.5.5 shows the utilization of the input switch for a smaller values of the switch delay, $t_s = 5$, with all other parameters are as in Fig.5.1. It can be observed that the saturated region is reduced to less than one half of that shown in Fig.5.2.

The consequence of the "relaxed" utilization of the input switch is that the utilization of the processor (and the performance of the whole system) improves, as shown in Fig.5.6 (again, the "front" part of Fig.5.6 corresponds to the "back" part of Fig.5.5, as the probability of remote accesses is used in Fig.5.5 rather than local ones used in Fig.5.6).

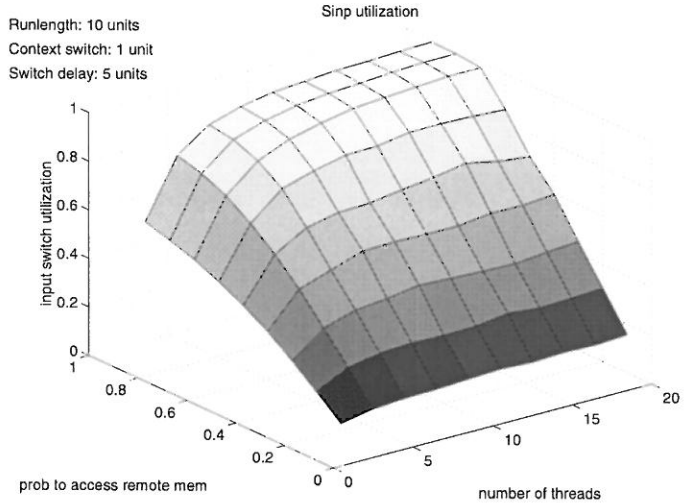



Fig.5.5. Input switch utilization ($t_s = 5, t_{cs} = 1$).

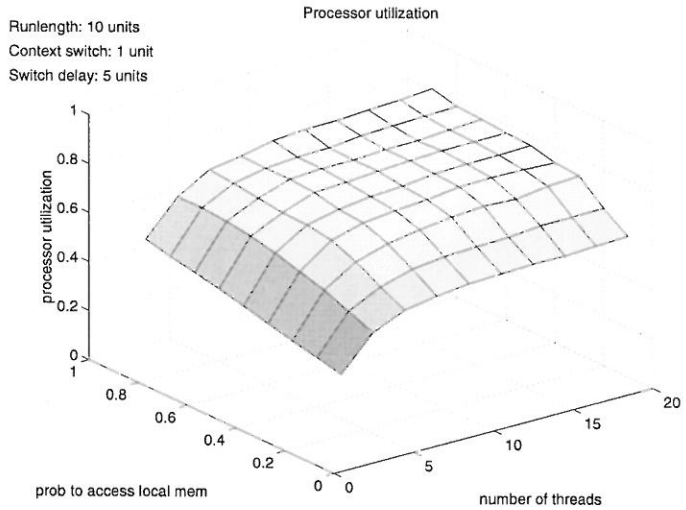



Fig.5.6. Processor utilization ($t_s = 5, t_{cs} = 1$).

Further reduction of the switch delay results in even better performance of the system.

Many other performance characteristics of multithreaded multiprocessor systems can be obtained in a similar way. Some other results are shown in [8].

## 5. Concluding remarks

It has been shown that a simple colored Petri net model of a fairly complex system can be derived in a systematic way, taking advantage of similarities between components of the system. The simple structure of the model does not mean, in general, that analysis is as straightforward as the model might imply.

The presented model (Fig.4.4) does not actually depend upon the number of processors in the original system, although some model parameters (for example, free–choice probabilities) must be adjusted if the number of processors changes. The whole interconnection network is represented by a single transition in Fig.4.4 (or a single class of free–choice transitions in Fig.4.5), and this representation is valid for torus–like and hypercube–type connections as well as many other types of interconnecting networks (the number of free–choice transitions in Fig.4.5 changes with the number of node connections).

The colored model captures very nicely the most important features of the system, illustrates the flow of requests, and represents the decision and conflicts which must be resolved by some additional criteria. The simplicity and clarity of the model seems to be one of major advantages of the proposed approach. Many less important details are hidden in the descriptions of model elements.

Even very simple performance analyses indicate that the system's behavior is quite sensitive to the values, and in fact relations between the values of model parameters [22]. If service demands to different components (such as processor, memory or switches) are not balanced, the performance of the whole system is limited by the most demanded component (called the bottleneck); any improvement in the performance of the system can be achieved only if the performance of this critical component is improved.

A throughput–preserving approach to model simplification has been proposed in [8]. Taking advantage of symmetries in the multiprocessor system, the model is substantially reduced by removing most of the nodes and by adjusting the throughputs of the remaining switches to the same values as in the original system; since the relative values of these throughputs can easily be derived from the steady–state conditions, the adjustments can be made without the knowledge of the actual values. In effect, the performance of the whole system can be obtained by evaluating a much simpler model. More research is needed in this direction to understand the limitations of such an approach.

## References


[1] Agrawal, A., Lim, B-H., Kranz, D., Kubiatowicz, J., "April: a processor architecture for multiprocessing"; Proc. 17-th Annual Int. Symp. on Computer Architecture, pp.104–114, 1990.

[2] Agrawal, A., "Limits on interconnection network performance"; IEEE Trans. on Parallel and Distributed Systems, vol.2, no.4, pp.398–412, 1991.

[3] Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Posterfield, A., Smith, B., "The Tera computer system"; Proc. Int. Conf. on Supercomputing, Amsterdam, The Netherlands, pp.1–6, 1990.

[4] Boothe, B. and Ranade, A., "Improved multithreading techniques for hiding communication latency in multiprocessors"; Proc. 19-th Annual Int. Symp. on Computer Architecture, pp.214–223, 1992.

[5] Culler, D.E., et al., "Fine–grain parallelism with minimal hardware support: a compiler controlled threaded abstract machine"; Proc. 4-th Int. Conf. on Architectural Support of Programming Languages and Operating Systems, Santa Clara, CA, pp.164–175, 1991.

[6] Ferrari, D., "Computer systems performance evaluation"; Prentice–Hall 1978.

[7] Govindarajan, R., Nemawarkar, S.S., LeNir, P., "Design and performance evaluation of a multithreaded architecture"; Proc. First IEEE Symp. on High–Performance Computer Architecture, Raleigh, NC, pp.298–307, 1995.

[8] Govindarajan, R., Suciu, F., Zuberek, W.M., "Timed Petri net models of multithreaded multiprocessor architectures"; Proc. 7-th Int. Workshop on Petri Nets and Performance Models, St. Malo, France, pp.153–162, 1997.

[9] Hirata, H., et al., "An elementary processor architecture with simultaneous instruction issuing from multiple threads"; Proc. 19-th Annual Int. Symp. on Computer Architecture, pp.136–145, 1992.

[10] Holliday, M.A., Vernon, M.K., "Exact performance estimates for multiprocessor memory and bus interference"; IEEE Trans. on Computers, vol.36, no.1, pp.76–85, 1987.

[11] Jensen, K., "Coloured Petri nets"; in: "Advanced Course on Petri Nets 1986" (Lecture Notes in Computer Science 254), Rozenberg, G. (ed.), pp.248–299, Springer Verlag 1987.

[12] Keckler, S.W., Dally, W.J., "Processor coupling: integration of compile-time and runtime scheduling for parallelism"; Proc. 19-th Annual Int. Symp. on Computer Architecture, pp.202–213, 1992.

[13] King, P.J.B., "Computer and communication systems performance modelling"; Prentice–Hall 1990.

[14] Moore, S.W., "Multithreaded processor design"; Kluwer Academic 1996.

[15] Murata, T., "Petri nets: properties, analysis and applications"; Proceedings of IEEE, vol.77, no.4, pp.541–580, 1989.

[16] Reisig, W., "Petri nets - an introduction" (EATCS Monographs on Theoretical Computer Science 4); Springer Verlag 1985.

[17] Saavedra–Bareera, R.H., Culler, D.E., von Eicken, T., "Analysis of multithreaded architectures for parallel computing"; Proc. 2-nd Annual Symp. on Parallel Algorithms and Architectures, Crete, Greece, 1990.

[18] Smith, B.J., "Architecture and applications of the HEP multiprocessor computer System"; Proc. SPIE – Real-Time Signal Processing IV, vol. 298, pp. 241–248, San Diego, CA, 1981.

[19] Weber, W.D., Gupta, A., "Exploring the benefits of multiple contexts in a multiprocessor architecture: preliminary results"; Proc. 16-th Annual Int. Symp. on Computer Architecture, pp.273–280, 1989.

[20] Zuberek, W.M., "Timed Petri nets – definitions, properties and applications"; Microelectronics and Reliability, vol.31, no.4, pp.627–644, 1991 (available through anonymous `ftp` at `ftp.cs.mun.ca/pub/publications/91-MaR.ps.Z`).

[21] Zuberek, W.M., "Modeling using timed Petri nets – event–driven simulation"; Technical Report #9602, Department of Computer Science, Memorial Univ. of Newfoundland, St. John's, Canada A1B 3X5, 1996 (available through anonymous `ftp` at `ftp.cs.mun.ca/pub/techreports/tr-9602.ps.Z`).

[22] Zuberek, W.M., Govindarajan, R. "Performance balancing in multithreaded multiprocessor architectures"; Proc. 4-th Australasian Conf. on Parallel and Real–Time Systems (PART'97), Newcastle, Australia, pp.15–26, 1997.

# Using Design/CPN for the Schedulability Analysis of Actor Systems with Timing Constraints

Libero Nigro and Francesco Pupo

*Dipartimento di Elettronica Informatica e Sistemistica*
*Università della Calabria, I-87036 Rende (CS) - Italy*
*Voice: +39-984-494748 Fax: +39-984-494713 Email: {l.nigro, f.pupo}@unical.it*

*Abstract.* This work is concerned with the use of Design/CPN for the modelling and analysis of actor-based distributed real-time systems. Coloured Petri Nets are used to obtain a formal and operational model of a specified system, as part of an iterative development process. Functional and timing properties of an achieved CPN model can be validated by means of simulation and occurrence graphs. The resultant approach facilitates transformations from analysis down through to the design and implementation in object-oriented languages like C++ and Java.

**Keywords**: Actors, Modularity, Real Time, Design/CPN, Temporal Analysis, Occurrence Graphs.

## 1 Introduction

Many distributed software systems can be represented, abstracting away from time aspects, as a set of asynchronous, autonomous components which interact one to another in order to co-ordinate local activity. The Actor model (Agha, 1986) is a well established computational framework suitable for building open and re-configurable general distributed applications.

In real-time systems time management is fundamental: application correctness depends not only on the functional results produced by computations, but also on the time at which such results are generated. Timing correctness is related to guaranteeing that a given logical behaviour is provided at the right time, not before nor after a due time.

In the last years, Agha et al. (Ren and Agha, 1995) (Ren et al., 1996) (Saito and Agha, 1995) have proposed extensions to the actor model in order for it to be applicable to real-time systems. The extensions rely on capturing message *interaction patterns* among actors through a *RTsynchronizer* construct (Ren and Agha, 1995). RTsynchronizers are declarative in character. They refer to groups of actors and specify timing constraints on the execution of relevant messages. Their concrete application depends on the possibility of ensuring a global time notion in a distributed system. Moreover a selected scheduling structure is required in order to guarantee that the timing constraints of messages are ultimately met. A major benefit of the use of RTsynchronizers is modularity. Actors are firstly defined according to functional issues only. Then timing aspects are separately specified through RTsynchronizers which affect scheduling.

The possibility of modelling actor systems through high level Petri nets has been previously investigated. In (Sami and Vidal-Naquet, 1991) an equivalence between actors and the formalism of CPN (Jensen, 1992) is provided with the goal of formalising actor semantics. The work of Agha et al. described in (Agha et al., 1992) is concerned with the use of Predicate Transition nets (Genrich, 1987) in order to support the visualisation of actor programs. Predicate Transition nets (PrT-nets) and CPN are formally equivalent and can be considered as two slightly different dialects of the same language. But none of these approaches has analysis aims. Moreover, only functional aspects are covered.

In (Nigro and Pupo, 1996) the temporal analysis of object-based real-time systems through TER nets (Ghezzi et al., 1991) was explored. However, the lack of mechanisms for modularity and compositionality, which are essential in the modelling of complex systems, was among the motivations leading to the choice of CPN (Jensen, 1994) (Jensen, 1997) in the work described in this paper. CPN temporal extension allows to exploit powerful analysis tools and techniques in the modelling and verification of timing aspects of actor systems.

This paper first describes a variant of the Actor model (Kirk et al., 1997) (Nigro and Pupo, 1997) where reflective actors are used to capture RTsynchronizers, then an embed of the chosen actor model into Design/CPN is proposed as a part of an iterative and incremental system development life (Verber et al., 1997). The proposed actor model centres on an integrated approach where actors, message passing, timing constraints and scheduling are all components of a time-predictable framework. In order to achieve the visualisation of both functional and timing issues of a modelled system, the approach depends upon the facilities and tools provided by Design/CPN, i.e., the high-level character of CPNs, the hierarchical and modular net constructions, the primitive time dimension and the verification of timing properties through the generation of occurrence graphs. As a benefit of the methodology, the validation process is accorded to the final development phases, in the sense that the same concepts and mechanisms move unchanged from analysis down to the design and implementation into an object oriented language like C++ or Java.

# 2 An actor-based framework for time-dependent systems

A modified version of the Actor model (Agha, 1986) was designed (Kirk et al., 1997) (Nigro and Pupo, 1997) to support the construction of real-time applications. It centres on actors directly modelled as finite state machines. As in the Actor model three basic operations are available:

- *new*, for the creation of a new actor as an instance of a class which directly or indirectly derives from a basic Actor class. The data component of an actor includes a set of *acquaintances*, i.e., the known actors to which messages can be sent
- *send*, for transmitting an *asynchronous message* to a destination actor. The message can carry data values. The sender continues immediately after the send operation
- *become*, for changing the current state of the actor. All of this modifies the way the actor will respond to expected messages. The processing of an unexpected message can be postponed by storing it into states or data.

Differences from the Actor model were introduced for ensuring real-time behaviour. Each object no longer has an internal thread. Rather, concurrency is provided by a *control machine* (see later in this paper ) which transparently buffers all the exchanged messages among the actors residing on a same processor, and delivers them according to a control strategy. In other terms, concurrency relies on a light-weight mechanism: *message processing interleaving*, which costs a method invocation in an object-oriented language. Only one message processing can be in progress within an actor at any instant in time. Message execution can't be suspended nor interrupted. All of this contributes to a deterministic computation of message response times.

Instead of associating a single mail queue to every actor, a few message queues (e.g., one) are handled by the control machine. A customisable scheduler hosting timing constraints avoids control machine dependencies from the policies and mechanisms of an operating system.

## 2.1 Basic Components

At the system level an application consists of a collection of subsystems/processors, linked one to another by a (possibly) deterministic interconnection network (e.g., CAN bus (Kirk, 1995)). A subsystem hosts a group of actors which are orchestrated by a control machine. Basic components are the control machine and the application actors. The control machine is articulated into the following sub-components (see also Fig. 1):

- a *local clock*, which contains a "real" time reference clock for all the actors in the subsystem
- a *plan*, which arranges message invocations along a timeline
- a *scheduler*, which filters message transmissions, applies to them a set of timing control clauses and schedules them on the plan. The scheduler is actually split into two actors: the *input filter* (iFilter) and the *output filter* (oFilter). iFilter is responsible of the scheduling actions of *just sent messages*; oFilter can be specialised to verifying timing violations at the dispatch time of a message
- a *controller*, which provides the basic control engine which repeatedly selects (*selector* block) from the plan the next message to be dispatched, and delivers it (*dispatcher* block) to its receiver actor.

A control machine is naturally event-driven. Asynchronous messages are received either from the external environment or from within a subsystem, and are processed by the relevant actors. A message processing can

generate further messages and so on. From this point of view, the message plan of the control machine could reduce to a FIFO queue.

The time-driven paradigm (Nigro and Tisato, 1996), on the other hand, adds the facility of scheduling messages to occur at certain due times. A possible organisation of the message plan consists of a set of partial ordered *time windows* of message invocations. A *message invocation* includes a message *msg*, its destination actor and two pieces of timing information: $t_{min}$ and $t_{max}$. It is intended that *msg* cannot be dispatched before $t_{min}$ and should be delivered before $t_{max}$ to avoid a timing violation. The selector component can choose the next dispatch message on the basis of an *Earliest Deadline First* (EDF) strategy.

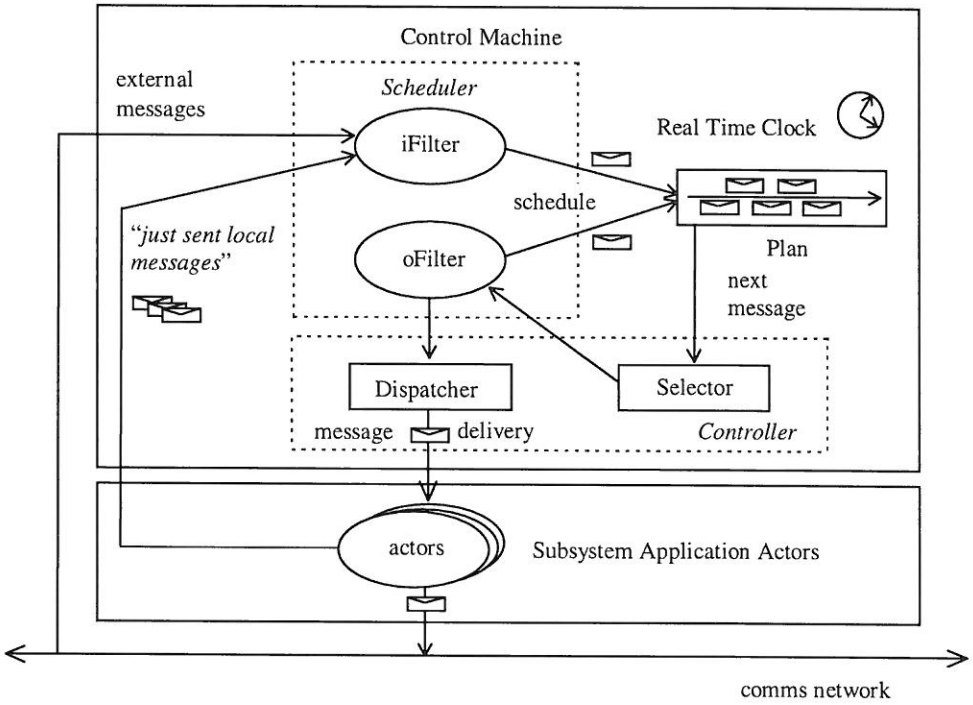



Figure 1: Architecture of an actor-based RT subsystem.

The control machines of a whole system can interact one to another in order to fulfil system-level requirements. To this purpose a suitable *interaction pattern* (protocol) (Agha, 1996) can be introduced which is responsible of inter-subsystem synchronisation (e.g., real-time barrier synchronisation (Saito and Agha, 1995)). An actor system can require both local and global co-ordination and control. A selected subsystem can host as a *master* control machine, where the operator, through a friendly and asynchronous graphical user interface, can issue configuration or monitoring commands.

## 2.2 Timing Constraints

A timing constraint normally specifies a time interval between the occurrences of a group of causally connected messages. A violation of a timing constraint can require a recovery action to be carried out. The following operations are provided for the expression of timing constraints and scheduler programming:

- msg.*cause()*, which returns the identity of a message msg' whose processing caused the generation of the message msg
- msg.*iTime()*, which returns the time at which the message msg was dispatched
- msg.*deadline()*, which returns the upper limit of msg's time window
- *schedule*( msg, $t_{min}$, $t_{max}$ ), which schedules on the control machine plan the message msg with the associated time window [$t_{min}$..$t_{max}$]
- *now()*, which returns the current value of local clock.

It is worthy of note that for the *iTime* function to be applicable to an incoming network message, a notion of a global time has to be provided (Ren et al., 1996).

A filter is fed of the timing parameters of a subsystem. A timing constraint is identified by a given *pattern*, i.e., a boolean expression involving message parameters and local data of the filter. If the pattern is satisfied the corresponding timing constraint is applied, possibly executing a scheduling action.

The following are some common examples of timing constraints. For simplicity, the are expressed in Java syntax. A framework is assumed where the external environment is sensed by periodic terminator actors and an environment condition starts a chain of messages ("thread of control") whose processing can be required to terminate within a deadline. A terminator synchronously operates a corresponding device (e.g., a sensor).

*Periodic message*
The time clause is associated with an actor (e.g., a terminator) which has a repetitive message (e.g., ReadCO). After being processed, the message is sent again by the actor to itself. Message periodicity is ensured externally to the actor in the iFilter:

```
if( m instanceof ReadCO && m.cause() == m ) schedule( m, m.cause().iTime()+P, m.cause().iTime()+P );
```

where *P* is the message period, which is data local to the iFilter.

*Deadline on a message*
A message can be required to be handled within a given deadline *D* measured since the invocation time of its cause:

```
if( m instanceof DeadlineMessage ) schedule( m, now(), m.cause().iTime()+D );
```

*Urgent time clause*
A message can be scheduled to occur immediately, e.g., for safety conditions:

```
schedule( m, now(), now() );
```

*Default time clause*
A default time clause can schedule a message to occur according to the deadline of its cause:

```
schedule( m, now(), m.cause().deadline() );
```

A more weak time clause requiring dispatching a message "as soon as possible" can be achieved by:

```
schedule( m, now(), infinite );
```

# 3 An example

The following considers a simplified model of a software controlled crane (Bergmans and Aksit, 1996) that can lift and carry containers from arriving trucks to a buffer area, from where they are taken for further handling. To carry the containers, the crane uses a magnetic latching mechanism. Actors can naturally model the crane control system (Crane), the buffer area (BoundedBuffer) and the operator (Console). Terminator actors are introduced for controlling the engine, the magnet, and the periodic item detector, left and right position sensor physical devices. Actor Crane understands the following messages:

- *On*, *Off* (turn on and off the magnet)
- *Loaded*, *UnLoaded* (raised by the *ItemDetector*)
- *Forward*, *Backward* (start a forward/backward crane movement)
- *RightStop*, *LeftStop* (raised respectively by *RightSensor* and *LeftSensor*)
- *Emergency* (stops crane and turns off the magnet in emergency situations).

In addition, Crane exports (synchronous) accessor state inquiries: *isOn*, *isOff*, *Moving*, *Stationary*, *isLoaded*, *isUnLoaded*, which return information about crane current state. The Engine and the Magnet actor classes have

messages for turning their state on and off. The Engine has also a message to setup the direction movement (forward/backward). The ItemDetector senses a container into the latching mechanism and sends a Loaded or Unloaded message to crane. Position sensors capture crane limit positions where the engine should be turned off. They transmit a RightStop or LeftStop message respectively.

Starting from a loaded container, first a Loaded message is received by crane. Loaded causes a Forward message to be sent by crane to itself. Processing Forward in turn generates messages for setting up the engine rotation movement and turning it on. Then the crane enters a MOVING state where it waits for a RightStop message. After that, the engine is turned off and a Put message is sent to the buffer area. After receiving a reply from put, the container is released by turning off the magnet and a Backward message is sent to the crane itself for the backward movement.

The crane system can also be controlled by the operator at the console under the restriction that On or Off requests are rejected while crane is moving. However, an Emergency message has to be processed within an assigned deadline *d*. Fig. 2 shows an iFilter which schedules the crane system. It is initialised with the relevant timing attributes: period of sensors and emergency deadline.

```
public class iFilter extends Actor{
  long p /*sensors period*/, d /*emergency deadline*/;
  iFilter( long p, long d ){ this.p=p; this.d=d; }
protected void handler( Message m ){
  if( m instanceof ReadItemDetector || m instanceof ReadLeftSensor || m instanceof ReadRightSensor ) {
    //periodic message
    if( m.cause() == m ) //schedule m according to period p
      ControlMachine.schedule( m, m.cause().iTime()+p,m.cause().iTime()+p );
    else //initialisation
      ControlMachine.schedule( m, now(), now() );
  }
  else if( m instanceof Emergency )
    ControlMachine.schedule( m, now(), now()+d );
  else if(   m.sender instanceof Operator || m.cause() instanceof ReadItemDetector ||
             m.cause() instanceof ReadLeftSensor || m.cause() instanceof ReadRightSensor )  //weak time constraint
    ControlMachine.schedule( m, now(), ControlMachine.infinite )
  else //default time constraint
    ControlMachine.schedule( m, now(), m.cause().deadline() )
}//handler
}//iFilter
```

Figure 2: An iFilter for scheduling the crane system.

It should be noted that self-driving periodic actors send to themselves the first periodic message during initialisation (i.e., within the constructor).
Exchanged messages during normal, automated behaviour, are handled by a default weak timing constraint with $t_{max}$ being infinite. However, an Emergency thread of messages is managed according to its deadline.

# 4 Modelling Actor Components by CP-nets

Basic system components (actors, control machine, scheduler, ...) can be 1-to-1 mapped on to CPN-subnets (pages). Mapping rules are described by modelling the crane system example presented in section 3. Figure 3 represents the hierarchy page of a CPN model, which has a node for each page in the model. An arc between two nodes indicates that the source node contains a transition (*substitution transition*) whose details are contained in the destination node. The page of the destination node is called a *subpage*. Each node is inscribed by the name and number of the corresponding page, while each arc is inscribed with the name of the corresponding substitution transition.

The hierarchy page shows the net structure of the model. At an highest level it is composed of three parts:

- the System Description part, which consists of the page *Crane_System*. It provides the most abstract view of the system

- the Actors part, which consists of eleven pages (*Actors, Crane, BoundedBuffer, Console, SensorSet, LeftSensor, RightSensor, ItemDetector, Actuators, Magnet, Engine*)
- the Control Machine part, which consists of five pages (*ControlMachine, iFilter, Selector, oFilter, Dispatcher*).

Crane_System is the topmost page in the hierarchy and it is shown in Fig. 4. It contains *Actors* and *ControlMachine* transitions communicating through the interface places CMPIn and CMPOut.

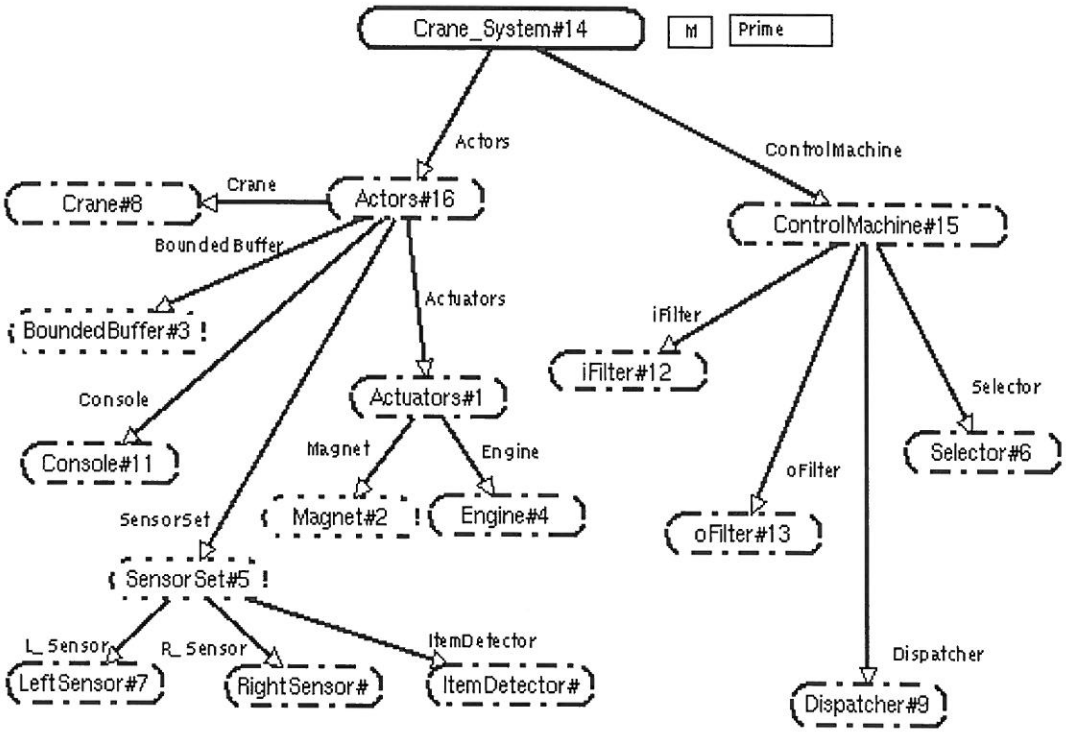



Figure 3: Hierarchy page of the crane system.

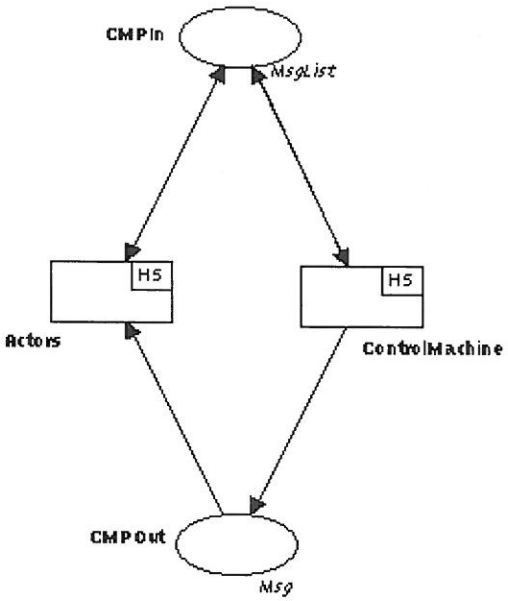



Figure 4: The Crane_System page.

Actors and ControlMachine in Fig. 4 are substitution transitions and from the hierarchy page it can be deduced that they correspond respectively to *Actors*#16 and *ControlMachine*#15 pages. The page Actors#16 is shown in Fig. 5 and contains some substitution transitions which represent the actor group in the system. Some of these

are directly substituted by subpages modelling corresponding actors (e.g., the *BoundedBuffer* page portrayed in Fig. 6), others abstract a group of logically related actors (e.g. SensorSet and Actuators) that are refined at a further sub-level.

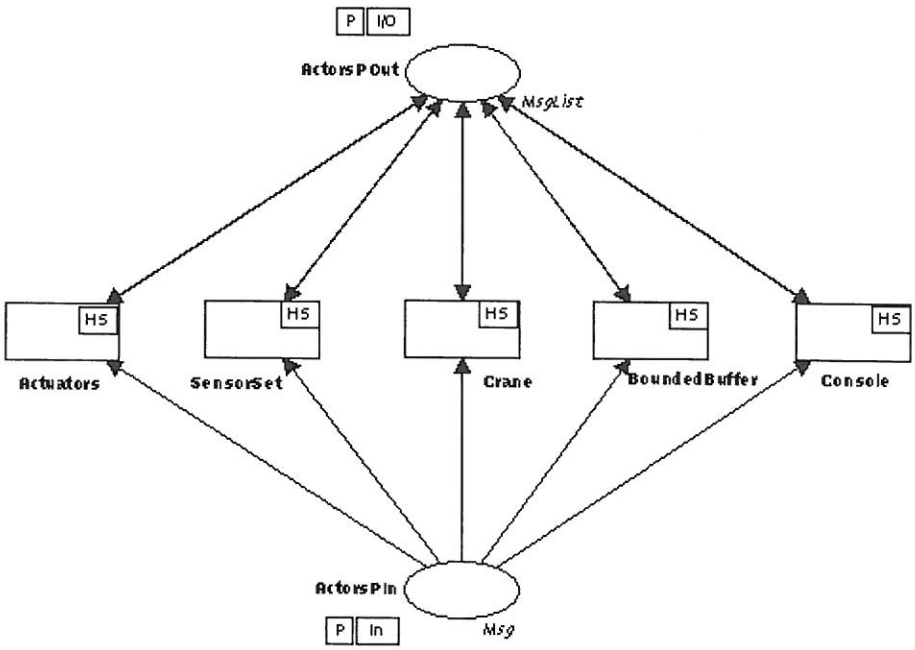



Figure 5: The Actors page.

In order to favour the analysis task of a system, actors can be modelled by a single transition (*action transition*) annotated by an estimation of its worst-case-execution-time. The actor behaviour is then captured by ML arc expressions and some local state places (see Fig. 6). This way two advantages can be achieved. The first one is related to a reduction of the net complexity and therefore to a minimising of the state explosion problem of the occurrence graph method. The second one is concerned with a reduction of the "distortion risks" during the translation of the verified model to the target application. This is obtained by expressing, owing to the high level character of the ML language, actor state transitions and actions in a form very close to the final implementation language.

The BoundedBuffer actor can receive a *Get* or *Put* message and can find itself into one of three states: *Empty*, *Partial* and *Full*. An incoming unexpected message, e.g., a *Put* in the *Full* status, is deferred by storing it in a local queue (token in the DEQ place). Deferred messages are re-scheduled to be received again as soon as the actor changes its current state.

The Control Machine component is modelled by the ControlMachine#15 page shown in Fig. 7. It contains four substitution transitions (*iFilter*, *Selector*, *oFilter* and *Dispatcher*) corresponding to the basic sub-components of the control machine, and five places, two of which (CMPIn and CMPOut) represent the interface with the system actors. CMPIn and CMPOut are port places corresponding to the socket places CMPIn and CMPOut of Crane_System page. These sockets are also assigned respectively to the ports ActorsPOut and ActorsPIn of the page Actors#16. The places Plan, CMP1 and CMP2 allow the control machine components to communicate.

The iFilter component is modelled by the page iFilter#12 depicted in Fig. 8. It contains a transition (iFilter) and four places. At each occurrence of the iFilter transition, the ifilt(...) function is invoked which takes the plan message list from the iFP2 place and the just sent message list from the iFP1 place, applies the time clauses of Fig. 2 and generates a new plan message list. Place iFP3 holds a copy of the last dispatched message which is used as a repository for inquiring about the causal message and its attributes of a message under scheduling.

The CMMonitor place is a member of a Fusion Global set and contains a token of the CMMon colour set which ensures that scheduling, selecting and dispatching activities in the control machine are strictly sequenced.

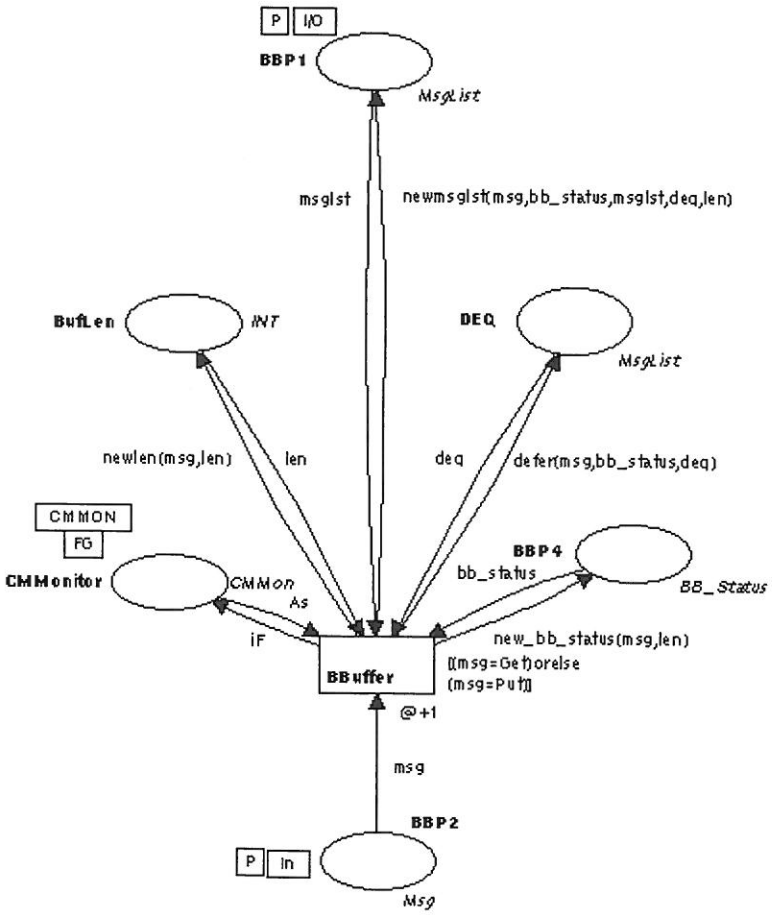



Figure 6: The Bounded Buffer page.

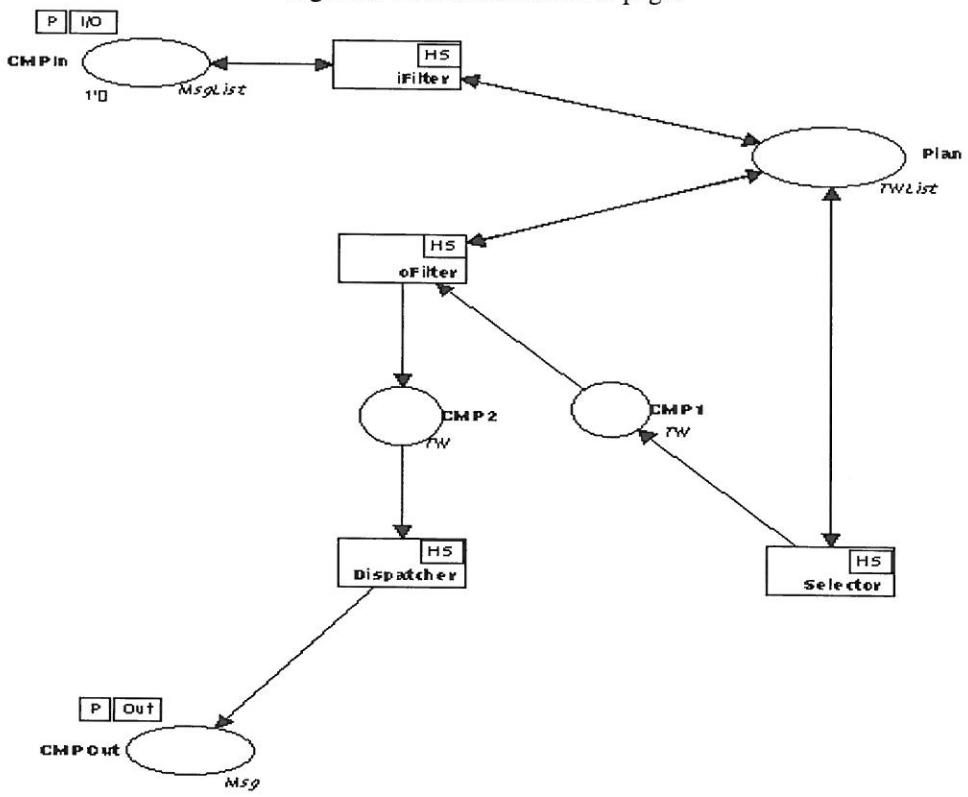



Figure 7: The Control Machine page.

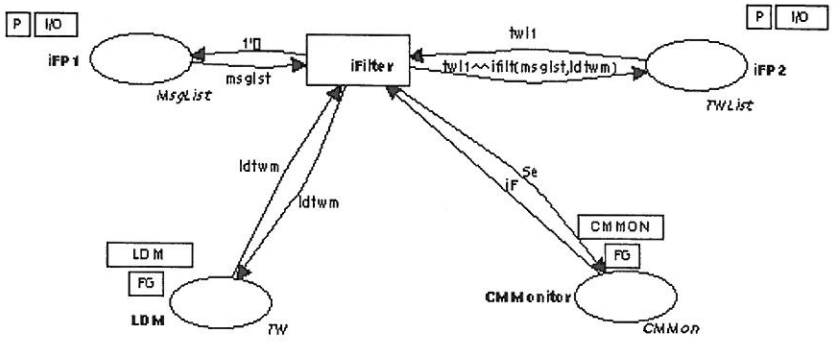



Figure 8: The iFilter page.

The iFP2 place corresponds to the socket place Plan in the page ControlMachine#15. This socket node is assigned also to the Plan port node of the page *Selector#6* which is represented in Fig. 9. This page models the selection of the next message to be dispatched from the Plan. In this case an EDF strategy is used. First the Plan

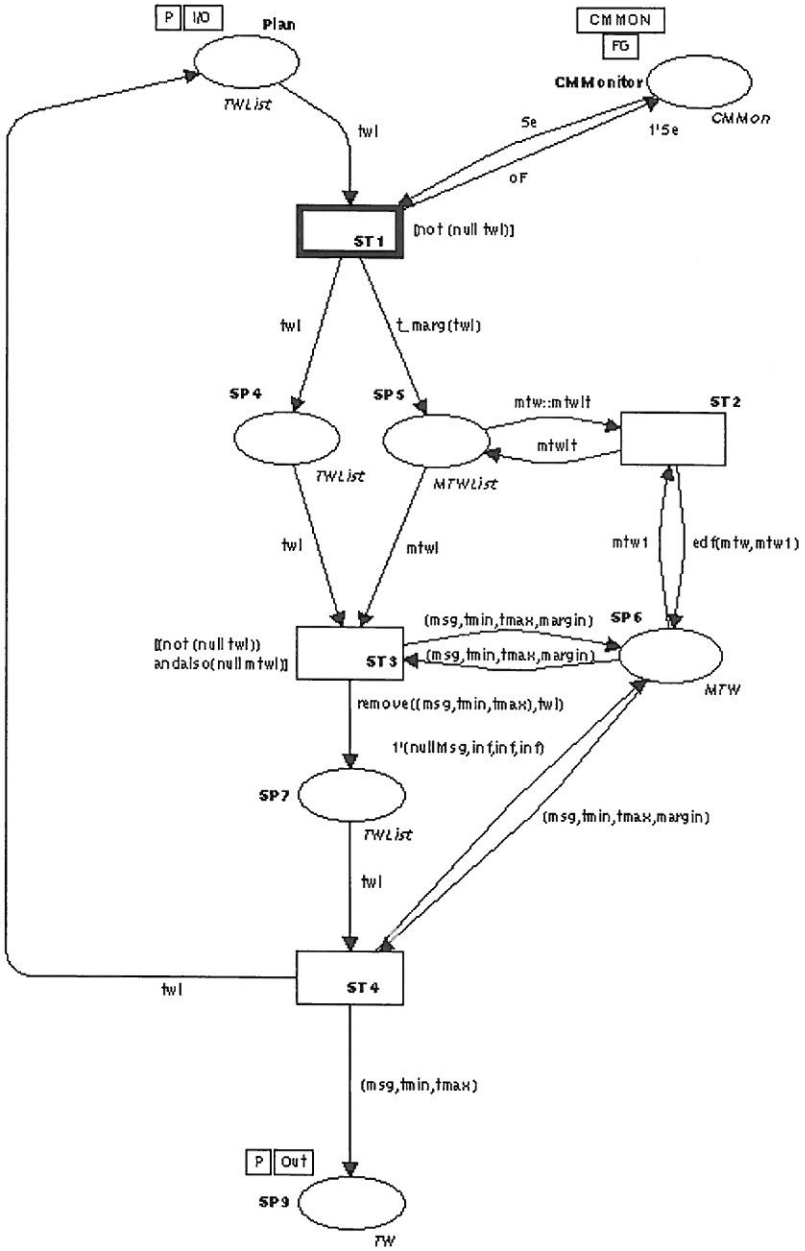



Figure 9: The Selector page.

message list is duplicated into the SP4 and SP5 places. The t_marg() function appends to each message item its *temporal margin*, i.e., the difference between the *tmax* value of the message time window and the current time. Transition ST2 edf() function is responsible of determining in the SP6 place the message with the minimum temporal margin which is removed (ST3 transition) from the message list copy in the place SP4. Finally, the firing of the ST4 transition updates the Plan message list and copies the selected message into the SP9 place for the subsequent dispatch phase.

It should be noted that for the purposes of the Crane example, the oFilter page doesn't apply any function to the message to be dispatched. Therefore, this page is omitted.

The Dispatcher page depicted in Fig. 10 is in charge of delivering the selected message to its relevant actor. The time inscription of the Dispatch transition:

$$@+if\ (tmin>time())\ then\ (tmin-time())\ else\ 0$$

allows to timestamp the message token in the DP2 place with the amount of time possibly required to advance the system time to the message *tmin*.

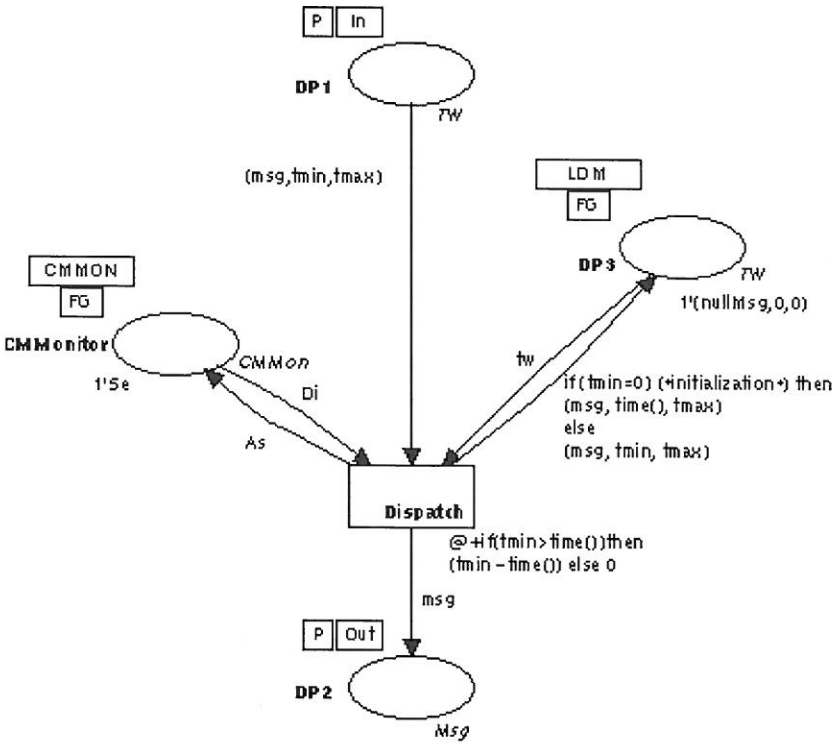



Figure 11: The Dispatcher page.

The destination actor receives the dispatched message, processes it by firing its action transition and generates a list of new messages. All these messages get timestamped with the occurrence time of the action transition augumented by the action duration. After that the control machine resumes its activity into the iFilter component thus starting a new cycle of the control loop.

# 5 Analysis of an Actor-based CPN Model

The analysis process of an actor-based CPN model aims at verifying and possibly correcting the specification of timing constraints and action worst-case-execution-times in order to ensure *schedulability* (e.g., Tsai et al., 1995). In particular, the analysis has to assert that every thread is completed within its deadline.

Both informal and formal analysis methods can be applied. The informal analysis can be conducted by simulating the model (*specifications testing* (Ghezzi et al., 1991)), i.e., by providing an initial marking and

then by tracing one or more possible resulting behaviours. By observing these behaviours the analyst can realise whether or not the specified system meets functional or timing requirements. Temporal behaviour can be verified by analysing the multiple concurrent threads especially under peak-loads.

Formal analysis consists in defining general properties of the net model which reflect special types of desirable (or undesirable) behaviours of the specified system, and then using the specification to formally prove (or disprove) such properties. For this purpose the *occurrence graph* (OG) method (Jensen, 1994) and the Occurrence Graph Tool (Occ Tool) (Christensen et al., 1996) can be used.

An OG is a directed graph which has a node for each reachable marking and an arc for each occurring binding element. An arc links the node of the marking in which the associated binding element occurs to the node of the marking resulting from the occurrence. Such a graph may become so large, even for relatively small nets, that it cannot possibly be generated even with the most powerful computer. Another limitation is dependency from the initial marking: each possible initial marking may originate a different occurrence graph.

The OGs generated for the analysis of an actor system are always timed, i.e., each node represents a system state and contains a time value and a timed marking. Since the typical periodic behaviour of such an RT system, it is possible to reduce the size of an OG by choosing a suitable simulation time during which meaningful system activity (e.g., peak-load conditions) occurs. In addition, the adopted modelling techniques (i.e., colour sets as intervals or enumerated values, and the adopted net structure) purposely contribute to keep under control the state space explosion problem.

All standard dynamic properties for a CP-net can be derived from its OG, e.g., boundedness, home, liveness, and fairness properties. They all can be investigated by available functions in the OG Tool. Moreover ML functions can be written for issuing non-standard inquiries to OG.

For instance, the maximum number of simultaneously enabled transition instances can be found by the MaxTransEnabled function in Fig. 12.

```
fun MaxTransEnabled () : int
  = SearchNodes (EntireGraph,
                 fn _ => true,
                 NoLimit,
                 fn n => length( remdupl( map ArcToTI( OutArcs n ) ) ),
                 0,
                 max);
```

Figure 12: Predictable system behaviour check function.

When invoked on any generated OG for the Crane CPN model, the function returns 1:

```
MaxTransEnabled ();
1 : int;
```

thus confirming that the modelled system always evolves in a predictable manner. Indeed, when a system consists of a single subsystem there is always only one transition enabled (belonging either to an actor or to an internal component of the control machine).

Similarly, one such a function can capture the verification of a temporal property in positive or negative form, provided an OG is generated for a suitable simulation period.

To exemplify, it is possible to carry out the verification of properties like the following: "is it always true that for each instance of a given transition firing (representing, e.g., the beginning of a thread) there always (as the OG state space allows) exists an instance of the corresponding transition firing (modelling, e.g., the end of the thread) such that the time distance between them is less than a fixed time interval (e.g., deadline of thread execution) ?"

The negative form of a property can be more immediate. In this case the existence of a single occurrence of a searched event that contradicts the property is sufficient to assert that the property doesn't hold.

In the crane example it can be verified (see Fig. 13) that for each occurrence of the *Emergency* event it always (in the generated OG) follows a transition instance corresponding to turning off the magnet (*Magnet* action transition) such that the temporal distance between the two events is less than a deadline *d*.

```
fun EmerDeadln(): Node list =
  PredAllNodes(fn n =>
                let
                  val Cr_Em = StripTime(Mark.Crane_System'CMPOut 1 n);
                in
                  if Cr_Em == 1`Cr_Emergency then
                    null(PredArcs(EntireGraph,
                                  fn a =>
                                    if ArcToTI(a) = TI.Magnet'Magnet 1 then
                                      ((CreationTime(DestNode(a))-CreationTime(n))<d)
                                      andalso(DestNode(a)>n)
                                    else false,
                                  1))
                  else false
                end)
```

Figure 13: Temporal property check function.

When invoked, the EmerDeadln() function always returns an empty list, proving that the list built with magnet transition instances which are causally connected to an Emergency event and temporally internal to the deadline interval, is never empty:

```
EmerDeadln();
val it = [] : Node list
```

The CP-net modelling of an actor system supports an *incremental development* through a modified spiral lifecycle model (Verber et al., 1997). Functional and temporal analysis can start as soon as a CPN specification has been produced and be based on the required obligations (periods, thread deadlines, ...) and a preliminary estimation of actor message processing times. As the temporal information about actors get more accurate, i.e., the project is tuned to a physical target architecture, the CPN model can be applied again to check the system remains schedulable. The key point is that the CPN model closely mirrors the design/implementation models of a system. Therefore, conclusions drawn from CPN analysis can directly be interpreted at the lower levels of development.

The experimental analysis of the Crane system was performed on a Sun Sparc 5 with 24 MB of physical RAM.

# 6 Modelling Issues of Distributed Systems

This section highlights how the CPN modelling techniques previously described can be adapted to support distributed systems. A simple example is sketched in Fig. 14 where two subsystems interact one to another by means of a network which introduces a constant transmission delay. In a more realistic case the network component could model a CAN priority bus (Nigro and Pupo, 1998).

The first subsystem (Site1) contains a pressure sensor and a valve. The second subsystem (Site2) contains a pressure controller which sends a message to open the valve when the pressure value exceeds a given limit. Site1 and Site2 share the basic Control Machine components which are instantiated for each subsystem.
The Network page is illustrated in Fig. 15. It models the message routing from a subsystem to another. The NBufOut place has a message list for each subsystem. A network message received into the NBufIn place is routed by the arc inscriptions to the correct message list in the NBufOut.

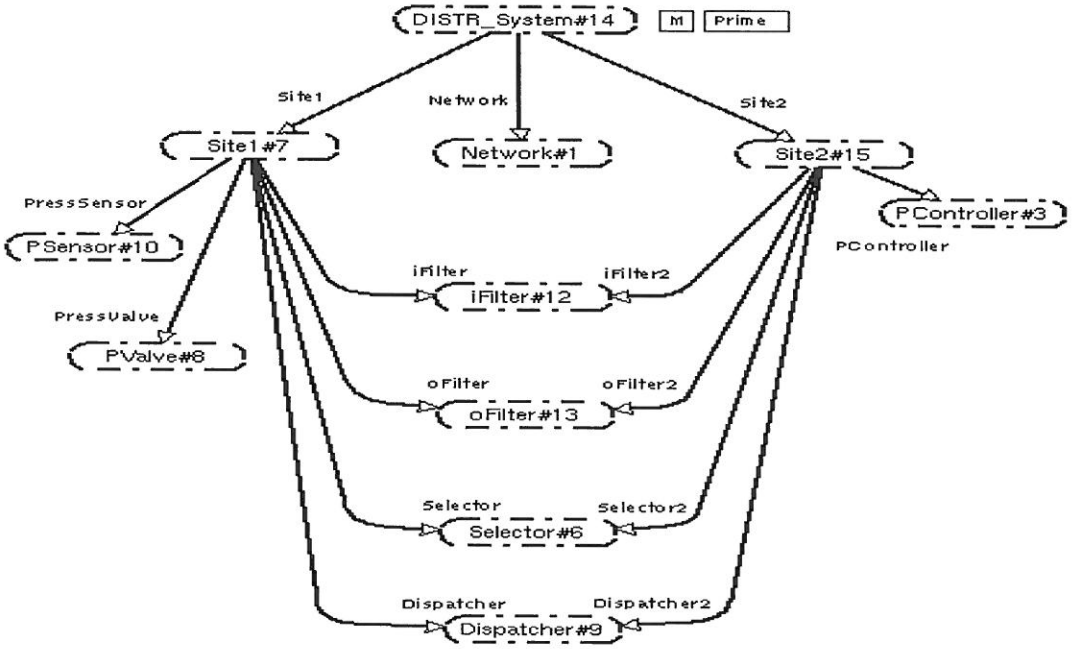



Figure 14: A simple distributed system.

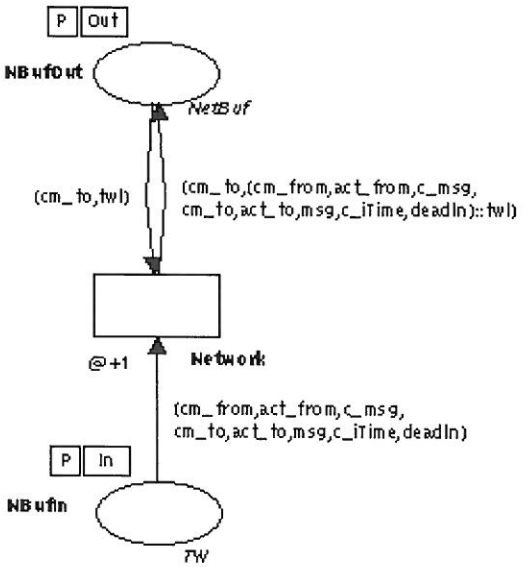



Figure 15: The Network page.

Fig. 16 shows the Site1 page. The identity of the control machine component instances is explicitly maintained by a token in the InstP11 and InstP12 places. Incoming network messages are scheduled on the plan along with the local generated messages. Outgoing network messages are deposited into the ToNetP place which corresponds to the NBufIn place of the Network page in Fig. 15.

It is worth noting that the two subsystems concurrently execute but the Design/CPN time notion ensures that the timing behaviour of each subsystem is always kept coherent with the time evolution of the whole system. For instance, a selected message by a control machine remains pending into the dispatch place if its timestamp is greater than the current CPN clock time. Only when no other transition in the model exists which can fire by incrementing the simulation time of a lower amount, the message is eventually delivered to its destination actor. As a consequence, the analysis of system-wide temporal properties, e.g., concerning threads of control which cross the boundaries of multiple subsystems, can be carried out according to the same techniques described in the section 5.

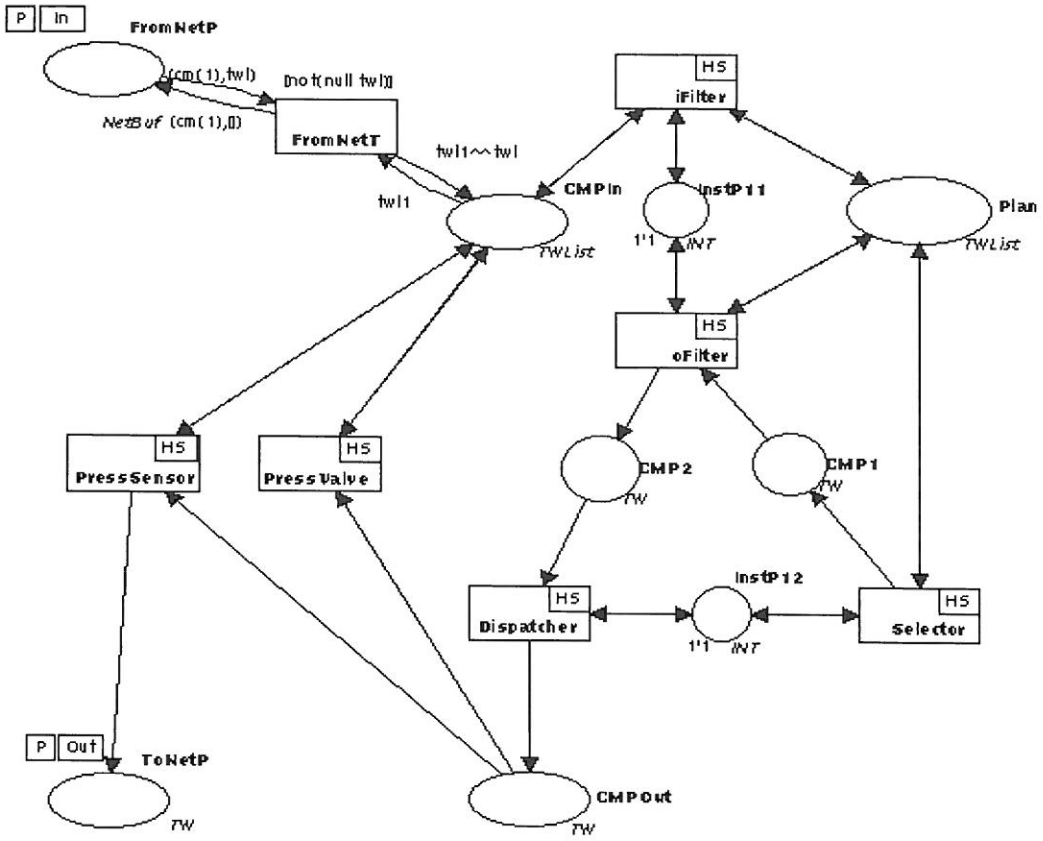



Figure 16: The Site1 page.

# 7 Conclusions

This paper describes an actor-based framework suited to the development of real-time distributed systems and shows a possible formalisation in terms of CP-nets. Benefits of the formalisation are the possibility of achieving visualisation and verification support in the context of the Design/CPN tool.

Like in (Verber et al., 1997) (Nigro and Pupo, 1996) both temporal and functional aspects can be considered since the early stages of a project using a modified spiral-model design life cycle where specifications can gradually be refined as the development becomes more and more specific. In particular, as the action execution time estimates become more precise, schedulability and timing behaviour can be checked that remain fulfilled. Temporal properties can be analysed by generating and exploring occurrence graphs.

The proposed approach facilitates transition to the final development phases, in the sense that a modelled and verified system can directly be transformed into the implementation terms of an object oriented language.

Directions of further work include

- experimenting with the proposed CPN modelling to real-life time-dependent systems
- improving the support of distributed systems by a better exploitation of page reuse and instance management mechanisms, in the presence of deterministic networks like CAN (Kirk, 1995) (Tindel et. al., ) which assign priority to inter-subsystem messages to control their transmission delay
- improving property analysis by using timed CTL (Cheng et al., 1996) for expressing queries on the state space of a generated occurrence graph.

The actor-based approach described in this paper is also in current use in the realisation of

- distributed measurement systems (Grimaldi et al., 1998)
- distributed simulation of cellular networks (Beraldi and Nigro, 1998)
- multimedia applications (Fortino and Nigro, 1998).

## Acknowledgments

Work carried out under the financial support of the Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Methodologies and Tools of High Performance Systems for Distributed Applications".

The authors wish to thank L.M. Kristensen and K. H. Mortensen of the CPN group at the University of Aarhus for their support during the experimental work with the Design/CPN tools.

# References

Agha G. (1986). *Actors: A model for concurrent computation in distributed systems*. MIT Press.

Agha G. (1996). Abstracting interaction patterns: a programming paradigm for open distributed systems. *Formal Methods for Open Object-based Distributed Systems*, Vol. **1**, Najm E. and Stefani J. B. (eds), Chapman & Hall.

Agha G., S. Miriyala and Y. Sami (1992). Visualizing Actor Programs using Predicate Transition Nets. *Journal of Visual Languages and Computation*, **3**(2), pp. 195-220, June.

Beraldi R. and L. Nigro (1998). Performance of a Time Warp based simulator of large scale PCS networks. *Simulation Practice and Theory*, **6**(2), pp. 149-163, February.

Bergmans L. and M. Aksit (1996). Composing synchronisation and real-time constraints, *J. of Parallel and Distributed Computing*, September .

Cheng A., Christensen S. and K.H. Mortensen (1996). Model checking Coloured Petri Nets: Exploting strongly connected components. WoDES'96, August 20. http://www.daimi.aau.dk/designCPN/libs/askctl.

Christensen S., K. Jensen and L. Kristensen (1996). The Design/CPN Occurrence Graph Tool. User's manual version 3.0. Computer Science Department, University of Aarhus. http://www.daimi.aau.dk/designCPN/.

Fortino G. and L. Nigro (1998). QoS centred Java and actor based framework for real/virtual teleconferences. *Proc. of SCS EuroMedia98*, Leicester (UK), Jan. 5-6, pp. 129-133.

Grimaldi D., L. Nigro and F. Pupo (1998). Java based distributed measurement systems. To appear on *IEEE Trans. on Instr. and Measurement*.

Genrich H. J. (1987). Predicate/transition nets. In *Advances in Petri Nets*, W. Brauer, W. Reisig and G. Rozenberg (eds.), New York, Springer Verlag.

Ghezzi C., D. Mandrioli, S. Morasca and M. Pezzè (1991). A unified high-level Petri net formalism for time-critical systems. *IEEE Trans. on Software Engineering*, **17**(2), pp. 160-172, February.

Kirk B. (1995). Real time protocol design for control area networks. *Proc. of Real Time'95*, Ostrava (Cz Rep.), pp. 251-268.

Kirk B., L. Nigro and F. Pupo (1997). Using real time constraints for modularisation. Springer-Verlag , LNCS 1204, pp. 236-251.

Jensen K. (1992). *Coloured Petri Nets - Basic concepts, analysis methods and practical use*. Vol. **1**: Basic concepts. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Jensen K. (1994). *Coloured Petri Nets - Basic concepts, analysis methods and practical use*. Vol. **2**: Analysis methods. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Jensen K. (1997). *Coloured Petri Nets - Basic concepts, analysis methods and practical use*. Vol. **3**: Practical use. EATCS Monographs on Theoretical Computer Science. Springer-Verlag.

Jensen K., S. Christensen, P. Huber and M. Holla. (1996). Design/CPN. A reference manual. *Computer Science Department*, University of Aaurus. Online: http://www.daimi.aau.dk/designCPN/.

Nigro L. and F. Pupo (1996). Modelling and analysing DART systems through high level Petri nets, Springer-Verlag, LNCS 1091, pp. 420-439.

Nigro L. and F. Tisato (1996). Timing as a programming in-the-large issue. *Microsystems and Microprocessors*, **20**, pp. 211-223, June.

Nigro L. and F. Pupo (1997). A modular approach to real time programming using actors and Java. *Proc. of 22nd IFAC/IFIP Workshop on Real Time Programming*, Lyon, 15-17 September, 83-88.

Nigro L. and F. Pupo (1998). Actors and Coloured Petri Nets in the development life cycle of distributed real time systems. To be presented at *IFAC Large Scale Systems'98*, Univ. of Patras Rio(Greece), 15-17 July.

Ren S. and G. Agha (1995). RTsynchronizer: language support for real-time specification in distributed systems. *ACM SIGPLAN Notices*, **30**, pp. 50-59.

Ren S., G. Agha and M. Saito (1996). A modular approach for programming distributed real-time systems. *J. of Parallel and Distributed Computing*, Special issue on Object-Oriented Real-Time Systems.

Saito M. and G. Agha (1995). A modular approach to real-time synchronisation. In *Object-Oriented Real-Time Systems Workshop*, 13-22, *OOPS Messenger, ACM SIGPLAN*.

Sami Y. and G. Vidal-Naquet (1991). Formalization of the behaviour of actors by coloured Petri nets and some applications. *PARLE '91*.

Tsai J.J.P., S.J. Yang and Y.-H. Chang (1995). Timing Constraints Petri Nets and their application to schedulability analysis of real-time system specification. *IEEE Trans. on Software Engineering*, **21**(1), pp. 32-49, January.

Tindel K., A. Burns and A.J. Wellings (1995). Analysis of hard real time communications. *Real Time Systems*, *9*, pp. 147-171.

Verber D., M. Colnaric, A.H. Frigeri and W.A. Halang (1997). Object orientation in the real-time system lifecycle. *Proc. of 22nd IFAC/IFIP Workshop on Real Time Programming*, Lyon, 15-17 September, pp. 77-82.