Evaluating Latency of Distributed Algorithms Using Petri Nets

Nicole Sergent
Département d'Informatique
Ecole Polytechnique Fédérale de Lausanne
CH - 1015 Lausanne (Switzerland)
sergent@lse.epfl.ch

Abstract

The time it takes to a distributed algorithm to finish (the distributed algorithm latency) cannot be directly measured in asynchronous systems if the algorithm starts and ends on different processors, since the system has no global time. A simple method to evaluate this latency is to build and simulate a unified model which includes the network and the distributed algorithm sub-models. In this paper we introduce a network model for UDP (User Datagram Protocol) which allows to establish a relationship between the number of messages exchanged in a distributed algorithm and the communication delays. As an application, we consider the two-phase commitment algorithm. Numerical results derived from simulating the model are compared with data obtained from performance measures.

1. Introduction

Usually, the performance of distributed algorithms is measured in terms of the number of messages exchanged during the execution. This is not the only relevant metric, assessing the algorithm latency may be also important, especially for real-time applications.

In distributed systems there is no shared memory nor global clock, so the communication and the synchronisation between processes on different nodes are done by message passing. The distributed algorithm latencies cannot be measured if computation begins and ends on different processors because processes do not have access to a global time. Another major impediment in the performance measures approach is that results are usually biased by unpredictable network and stations loads, not related to the studied algorithm. From these two reasons stems the interest in modelling the distributed algorithm and the underlying network using a formal description technique. By simulating this model it is possible to evaluate the algorithm latency with pretty good accuracy.

A distributed algorithm may be seen as an upper layer application based on a lower layer, the network. Between these two layers there is a strong interconnection. The number of messages generated by the algorithm has a direct influence on the end-to-end delay experienced by a message in the underlying network. This is particularly obvious in Ethernet based networks, where allocating the bus becomes a bottleneck, but this may be also true for networks capable of carrying several messages simultaneously, like FDDI. For some algorithms [3], the number of generated messages depends on the communication delays. This interrelation between the network and the algorithm makes it difficult to build an analytical model from which to derive the algorithm latency. An analytic modelling approach is presented in [10], for algorithms with a deterministic number of exchanged messages.

A more direct approach is to use a formal description technique, such as Petri nets[12], for modelling both the network and the distributed algorithm and then to derive the algorithm latency from simulations.

Since it is impossible to provide a universally valid network model, we focus on a very common network protocol, the User Datagram Protocol (UDP) [2]. First, a simple model for UDP in a LAN was built and validated through performance measures.The UDP model was translated into a Petri net model, which is reusable for all kinds of distributed algorithms based on exchanging messages.

As an application, we have chosen the two-phase commitment protocol [5]. The performance of any other distributed algorithm may be studied within this framework. Using the hierarchy concept of Petri nets, we build a unified model, which is simulated with a different number of participant processes. Approximate performance measures for the two-phase commitment protocol are used to validate the unified model.

The model is scalable with respect to the number of participant processes, i.e. we may increase the number of processes without modifying the model structure, which would be time-consuming. This is an important asset of our modelling approach over the more difficult performance measures.

2. Petri net model

The classical Petri net [11], [12] is a directed bipartite graph with two kinds of nodes, places and transitions. The nodes are connected by directed, weighted arcs. Arcs do not connect nodes of the same type. Places are graphically represented by circles and transitions by rectangles. Places may contain a number of tokens, drawn as black dots.

A place p is called an *input* place for a transition t if there exists a directed arc from p to t. A place p is called an *output* place of t if there exists a directed arc from t to p. The disposition of tokens on the places is called a *marking*. M(p) is the number of tokens on the place p. The initial marking of a net is the contents in tokens of each place before executing the net. During the execution the number of tokens may change.

A transition is *enabled* if each input place contains "enough" tokens, corresponding to the input arc weights. An enabled transition is eager to fire. When transition t fires, it removes the enabling tokens from the input places and produces new tokens for the output places. A transition t produces multiple tokens for an output place p if the weight of the arc connecting t to p is more than one (the number of produced tokens is the weight of the arc).

Petri nets have been used in many application areas, like communication protocols, distributed systems, flexible manufacturing systems, hardware design, information systems. Classical Petri nets describing real systems tend to be complex and extremely large. This leads to extensions of the initial Petri nets, like introducing "colour" and "hierarchy" [9]. Such extensions are necessary for modelling complex systems but are not sufficient for modelling time issues. The "time" extension was introduced in Petri nets to allow to use them in the area of the performance evaluation. We will informally present the three extensions (for more details see [9]).

Adding colour in Petri nets was a step comparable with the introduction of types in the programming languages. Each token may carry complex typed information and this information may influence the firing rule of transitions. Colours considerably reduce the size of Petri nets but preserve formal analysis possibilities.

The basic Petri nets are not suitable for timed applications, so several propositions were made to include time in the model [11]. We consider the model where each token has a timestamp which indicates when the token becomes available. The timestamp of an output token, produced by a transition t, is equal to the sum of the firing time and a *delay* specified by the transition t [13].

The introduction of hierarchical Petri nets [8] is comparable with the development of structural languages, where subroutines provide reusable code. In hierarchical Petri nets complex systems can be decomposed into subsystems, more easy to specify and design. The modeller may concentrate on the details of a subsystem, ignoring all the other aspects of a system.

3. The UDP/IP model

The UDP/IP protocol is widely used for message exchange between distant processes connected through a LAN (Local Area Network). The analysis presented in this paper considers that UDP/IP is built on top of an Ethernet (CSMA/CD) bus [2], so the protocol layers involved in a message exchange are as presented in Figure 1.

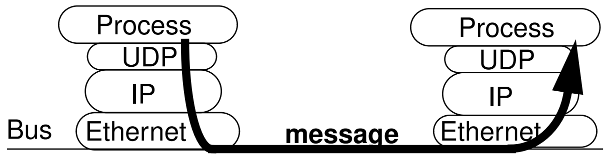

Figure 1 Communication between two processes

To simplify our analysis, we make several assumptions:
- the stations connected on the bus are identical
- there is at most one process running on a station
- the LAN is private, i.e. there is no other message passing on the network except those generated by the considered processes
- there is no message loss, nor transmission error
- at the process level, the instructions are executed in a null time delay.

The size of the messages at the process level is 4 bytes, the IP datagram size is 32 bytes and the Ethernet frame size is 64 bytes. We assume point-to-point communications, implemented via UDP sockets by datagrams. The network controller [2] has its own processor, which is responsible for implementing the Ethernet protocol.

Sending a message from the process level to the network controller requires the allocation of the station's CPU. If the operation succeeds, the CPU of the sending station will be unavailable for t_S time units, the delay for a send operation; t_S is a constant depending on the UDP/IP implementation and the operating system characteristics. If a process attempts to send a message while the CPU is busy, the message will have to wait until the CPU becomes idle. We denote this waiting time by t_{wS}.

For receiving a message, a process has to allocate the CPU of the host station for t_R time units, counted from the moment the last bit of the frame has been read by the network controller. If the CPU is busy, the message will wait t_{wR} time units before being received by the process. The network controller is available immediately after the last bit of a message has passed, so it can detect a new frame.

For sending the bits of a frame on the bus, the controller needs t_t time units, which represents the transmission time; t_t can be computed as L/D, where L is the frame length in bits and D is the number of bits which are transmitted in a second. For D=10 Mbits/s, we have t_t=52 µs. The propagation time t_p represents the time it takes for a bit to pass on the bus from one station to another. For a maximal distance of 500 meters on the bus between two stations, t_p is 2,5 µs. The time needed to send a frame on the bus is t_n, which is the sum of t_t, t_p and other memory access delays. This means t_n>54.5 µs. If the bus is busy, a frame will have to wait for t_{wn} time units, before being transmitted by the controller. Sending a message occupies the bus during t_t+t_p time units, in order to simplify the model we consider the bus unavailable for t_n time units.

The end-to-end communication delay T experienced by a message exchanged between two distant processes is $T=t_{wS}+t_S+t_{wn}+t_n+t_{wR}+t_R$. While t_S,t_n and t_R are constants, t_{wS},t_{wn} and t_{wR} are variable delays depending on the network and stations loads [1].

4. Model validation

The model was validated against the performance results of a real system, consisting in a network of SUN stations (SS2-GX, 1 x CPU SPARC, 40 Mhz, 64MB RAM), interconnected over an Ethernet CSMA/CD bus (no repeaters) at 10 Mbits/s. We run several simple tests. The results of the first three tests are used to identify the t_S,t_n and t_R constants and the last test validates the model. The exchanged messages have a special identifier which makes sure there is no message loss during our performance measures.

4.1 Finding the constants

1) In the first test there are two processes, p_1 and p_2, running on different stations and exchanging the messages m_1 and m_2, as in Figure 2.

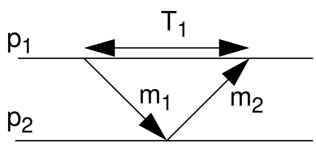

Figure 2 The first test

Process p_1 sends a message m_1 to p_2 and waits for a reply. After receiving the reply from p_2, p_1 records the elapsed time T_1. This test is analysed using the UDP/IP model. Process p_1 sends m_1 at time $t=0$, so m_1 arrives at the controller at $t=t_S$, since the first station's CPU is available ($t_{wS}=0$). At this moment the bus is idle, so m_1 passes on the network ($t_{wn}=0$) and is received by the controller of the second station at $t=t_S+t_n$. The second station's CPU is idle, so m_1 will be received by p_2 at $t=t_S+t_n+t_R$. The reply m_2 is sent at $t=2t_S+t_n+t_R$ and received by the controller of the first station at $t=2t_S+2t_n+t_R$. Finally, p_2 receives m_2 at $t=2t_S+2t_n+2t_R$ ($t_{wR}=0$, since the first station's CPU is available). Figure 3 presents the results of this analysis.

	t_{wS}	t_S	t_{wn}	t_n	t_{wR}	t_R
m_1	0	t_S	0	t_n	0	t_R
m_2	0	t_S	0	t_n	0	t_R

Figure 3 Waiting times in the first test

The model gives $T_1=2t_S+2t_n+2t_R$ and the performance measures result is T_{1exp}=1.336 ms. The first equation is:

$$T_1=T_{1exp} \qquad (1)$$

2) In the second test there are three processes, p_1, p_2 and p_3, exchanging four messages, as in Figure 4.

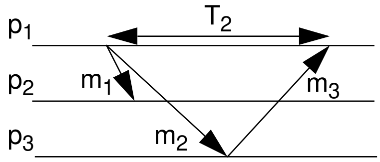

Figure 4 The second test

Process p_1 sends two messages, to p_2 and p_3 respectively, and waits for a reply from p_3. The analysis is similar to that performed for the first test, except for the message m_2 waiting times, t_{wS} and t_{wn}. The message m_2 arrives at the controller at $t=2t_S$, since it has to wait for m_1 to be sent. If $t_S>t_n$, m_1 has already passed on the bus and the bus is idle, so $t_{wn}=0$. If $t_S<t_n$, m_2 has to wait for the bus ($t_{wn}=t_n-t_S$). So $t_{wn}=max(0,t_n-t_S)$. These results are presented in Figure 5, where the constants t_S,t_n and t_R are omitted.

	t_{wS}	t_{wn}	t_{wR}
m_1	0	0	0
m_2	t_S	$max(0,t_n-t_S)$	0
m_3	0	0	0

Figure 5 Waiting times in the second test

We have $T_2=3t_S+2t_n+2t_R+max(0,t_n-t_S)$, or

$$T_2 = \begin{cases} 3t_S + 2t_n + 2t_R, & \text{if } t_S \ge t_n \\ 2t_S + 3t_n + 2t_R, & \text{if } t_S < t_n \end{cases}$$

Our performance measures provided T_{2exp}=1.605 ms. The second equation is: $T_2=T_{2exp}$ (2)

3) In the third test, two processes, p_1 and p_2, exchange the m_1, m_2, m_3 and m_4 messages, as depicted in Figure 6.

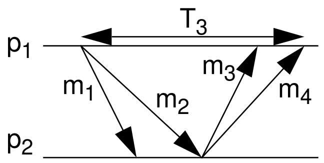

Figure 6 The third test

Process p_1 sends two messages to p_2 and waits for two replies. After receiving the second reply, p_1 records the elapsed time T_3. Analysing this test provides the results from Figure 7.

	t_{wS}	t_{wn}	t_{wR}
m_1	0	0	0
m_2	t_S	$max(0,t_n-t_S)$	t
m_3	0	0	0
m_4	t_S	$max(0,t_n-t_S)$	t

Figure 7 Waiting times in the third test

The message m_1 is received by p_2 at $t_1=t_S+t_n+t_R$. The message m_2 is received by the controller of the second station at $t_2=2t_S+t_n+max(0,t_n-t_S)$. If $t_1<t_2$ then m_2 is immediately received by p_2, otherwise it has to wait the reception of m_1 by p_2.

We have

$$t = max(0, t_1 - t_2) \Rightarrow t = \begin{cases} 0, & \text{if } t_1 \le t_2 \\ t_1 - t_2, & \text{if } t_1 > t_2 \end{cases}$$

$$t = \begin{cases} 0, & \text{if } (t_S > t_n) \text{ and } (t_S > t_R) \\ 0, & \text{if } (t_n > t_S) \text{ and } (t_n > t_R) \\ t_R - t_S, & \text{if } (t_S > t_n) \text{ and } (t_R > t_S) \\ t_R - t_n, & \text{if } (t_n > t_S) \text{ and } (t_R > t_n) \end{cases}$$

From the model, $T_3=4t_S+2t_n+2t_R+2max(0,t_n-t_S)+2t$.

The performance measures result is T_{3exp}=1.92 ms. The third equation is $T_3=T_{3exp}$ (3)

The system of equations (1),(2) and (3) gives two solutions:

(i) t_S=0.269 ms, t_n=0.107 ms, t_R=0.292 ms and

(ii) t_S=0.107 ms, t_n=0.269 ms, t_R=0.292 ms.

We expect $t_S \cong t_R$, which will be confirmed by the fourth test.

4.2. Validating the constants

4) In the fourth test there are six processes which exchange messages, as depicted in Figure 8.

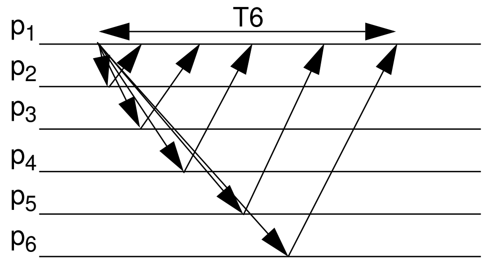

Figure 8 The fourth test

The performance measures give T_{6exp}=2.928 ms. The first solution of our model gives T_6=2.805 ms. The second solution gives T_6=3.473ms. We have to reject the second solution and validate only the first one. The relative error is 4.2%.

5. The UDP/IP Petri net model

We have built our Petri nets using EXSPECT, a software tool developed by the Eindhoven University of Technology [4], [6], [7], [13]. The language EXSPECT consists of two parts, a functional part and a dynamic part. The functional part is used to define types and functions. The dynamic part is used to specify the net. EXSPECT provides a hierarchy construct, called system. A system is an aggregate of transitions, places and (possibly) subsystems. Graphically, squares with a marked bottom right-hand corner denote transitions, rectangles denote systems, circles denote places, crossed circles denote one-capacity places (stores). Exspect allows interactive net simulation.

The UDP/IP Petri net model is presented in Figure 9. For readability reasons we do not include arc inscriptions,

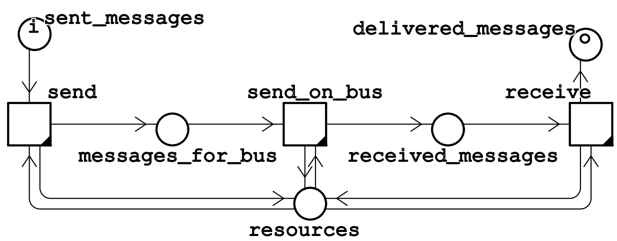

Figure 9 The UDP/IP model

written in a functional language, together with the nets.

Instead we present in detail the functioning of the net.

When a process sends a message to another process, it deposes a token in the «sent_messages» place. The token carries several data, such as the source and destination process identifiers, the message type and the message contents. The tokens from the «messages_for_bus» place model messages which are waiting for the bus at the sending controller level. If several messages are waiting for the bus, the one who will occupy the bus is chosen randomly (i.e. all the messages have the same priority). Messages received at the controller level but not yet received by the process are modelled as tokens in the «received_messages» place. Tokens in the «delivered_messages» place model messages received at the process level. The send and receive operations have the same priority on a station.

To arrive at the controller, a message from the «sent_messages» place needs the station's CPU, which is modelled as a token residing in the «resources» place. This place contains a token for each station CPU and a token which models the bus. Colours allow to distinguish the different resources. The «send» transition models a send operation from the process level to the controller level. It consumes two input tokens, from the «sent_messages» and «resources» places and produces two output tokens, delayed with t_S, for the «resources» and «messages_for_bus» places. The «send_on_bus» transition models the bus allocation and the message transmission. This transition may fire if it can consume the «bus» token (the bus is idle) and a message token from the «messages_for_bus» place. Firing will produce a message token, in the «received_messages» place, and the bus token, in the «resources» place, both delayed with t_n. The «receive» transition models the message reception and the station's CPU allocation during t_R time units.

6. The two phase-commitment algorithm

To ensure consistency in a distributed data base, transaction updates involving several sites must be integrated in the database by an atomic action. Protocols for ensuring transaction atomicity are called *commit protocols*.

The simplest commit protocol is proposed in [5]. This protocol uses a designated process, the coordinator. In the first step, the coordinator sends a «vote-request» message to all other participant processes and each process votes on whether to commit («yes») or to abort («no») the transaction (see Figure 10). In the second step, the coordinator gathers all the votes and informs each participant of the outcome. The protocol is not fault-tolerant, it blocks if one of the processes crashes. The *termination time* (latency) for this protocol is the time elapsed from the moment the coordinator sends the «vote-request» message to the first participant until all the participants have received the coordinator's decision.

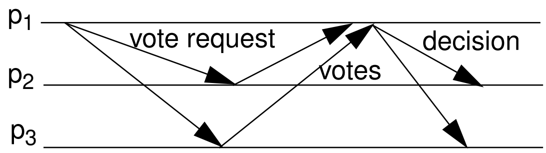

Figure 10 The two-phase commitment protocol with 3 processes

The performance measures we have made to evaluate this latency are approximate: we measure the time elapsed from the moment the coordinator sends the transaction to all the participants until the coordinator gathers all votes from the participants (this is possible since the measurements are made on the same station) and we add half of the round-trip time.

7. The unified model

The unified Petri net model for the two-phase commitment protocol is presented in Figure 11. The «network» system from Figure 11 is the Petri net from Figure 9 (we are using here the hierarchy concept [8] of Petri nets to hide the complexity of the UDP/IP model inside the unified model). The «sent_messages» and «delivered_messages» places are the connections between the two Petri nets.

Before executing the net, the *n* processes, modelled by *n* coloured tokens, are situated in the «start» place. Each token has a colour which models the *pid* of the corresponding process. One of the processes (for instance, the one which has the greatest pid) is the coordinator. The «vote_request» transition firing consumes the coordinator token and produces *n-1* token messages, one for each participant, in the «sent_messages» place. The coordinator token is put in the «waiting_replies» place, reflecting the new state of the coordinator process. Each participant waits in the «start» place until it detects a «vote request» message token in the «delivered_messages» place. Colours allow a participant to detect whether the message is addressed to him or not. Then the «send_vote» transaction fires and the vote token is put in the «sent_messages» place. A message token contains the same data as in the UDP/IP Petri net model, i.e. the source and destination identifiers, the message type and the message contents. After sending his vote, a participant proceeds to a new state, reflected by a token output in the «waiting_decision» place.

The coordinator stores the number of the votes he has currently received in a special place, «votes», represented as a crossed circle, which contains only one token. After receiving *n-1* votes, the «receive_votes» transaction will output *n-1* message tokens in the «sent_messages» place and a coordinator token in the «decided» place. The «finish» transition detects the algorithm end, when all the participants have decided, and records the current time in the «results» place, using the current time value from the «time» store, updated by the EXSPECT simulator.

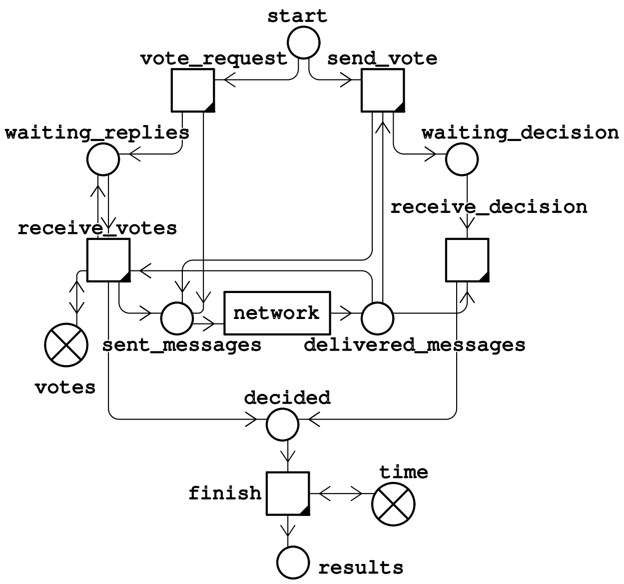

Figure 11 The two-phase commitment model

The model was validated by the invariant analysis technique [9],[12] and by performance measures, executed in the same environment as for the UDP/IP validation. A special analysis tool, the IAT, found the place invariant M(start)+M(waiting_replies)+M(waiting_decision)+M(decided)=constant [4]. This invariant proves that the number of processes does not change during the net execution. Figure 12 depicts the two-phase-commitment latency over the number of participant processes. To minimise the bias from extra-loads on stations and network, the approximate performance measures were done at unsocial hours. The simulation results correspond to the performance measures results within 6%.

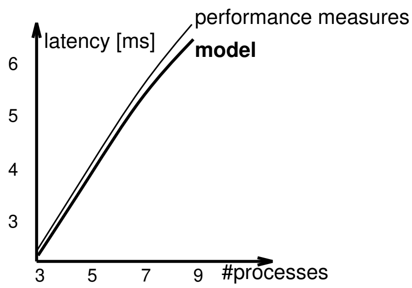

Figure 12 Simulation and performance measures results

8. Conclusions

This paper presents a modelling approach, based on Petri nets, for the evaluation of the distributed algorithm latencies. As an application, we have chosen the two-phase commitment protocol, for which a Petri net model was developed. The performance results obtained by simulating this model were proved to be close enough to the performance measures derived from the real system. An accurate model of the UDP/IP protocol was developed using Petri nets and integrated into the two-phase commitment algorithm model. The advantage of using Hierarchical Coloured Timed Petri nets and the EXSPECT tool is the possibility to make a good trade-off between the size of the net and the length of the code. The model is scalable: while the size (the number of processes involved in the atomic commitment) of the modelled system increases substantially, the size of the model remains constant and the modelling time grows insignificantly. Simulating the model is more attractive than performance measures, since the results are not biased by unpredictable system loads or by approximations due to the lack of global time. The UDP/IP model may be reused for the evaluation of other distributed algorithms.

References:

[1] Cabrera L. F., Hunter E., Karels M. and D.Mosher.1988, User-Process Communication Performance in Networks of Computers, *IEEE Transactions on Software Engineering*, vol. 14, no.1.

[2] Comer D.E. and L.D. Stevens, *Internetworking with TCP/IP,* Volume II, 1991, Prentice-Hall.

[3] Chandra T. D. and S. Toueg. 1996. "Unreliable failure detectors for reliable distributed systems". *Journal of ACM*, 34(1), pp. 225-267.

[4] ExSpect Reference Manual - Release 5.0., Bakkenist ExSpect, Netherlands, november 1994.

[5] Gray J.N., Notes on Database Operating Systems, Operating Systems: An Advanced Course, Springer-Verlag, 1979.

[6] Hee K.M. Van, Somers L.J., Voorhoeve M., Executable Specifications for distributed information systems.. In Information System Concepts: An In-depth Analysis, North-Holland, 1989.

[7] Hee K.M. Van, Houben, Dietz J.L.G., Modeling of discrete dynamic systems-framework and examples, In Information Systems, Vol. 14, No.4, Pergamon Press, 1989, pp.277-289.

[8] Huber P., Jensen K., Shapiro R.M., Hierarchies in Coloured Petri Nets, In Advances in Petri Nets 1990, LNCS 483.

[9] Jensen K., Coloured Petri nets: basic concepts, analysis methods and practical use. , Springer 1994.

[10] Menascé D., Yesha Y., On a Unified Framework for the Evaluation of Distributed Quorum Attainment Protocols, IEEE Transactions on Software engineering, vol.20, no. 11, november 1994.

[11] Molloy M.K., Petri Net Modelling, The Past, the Present and the Future, International Conference on Petri Nets and Performance Models, 1989.

[12] Peterson J.L., Petri Nets Theory and the modelling of systems, Prentice Hall 1981.

[13] Voorhoeve M., Reus B.M. de, ExSpect Language Tutorial - Release 5.0, Bakkenist ExSpect, Netherlands, december 1994.